Optics, lenses and prisms

Motivation

OOP *universe* · *galaxy* (*milky*) · *sun* · *earth* · *mff* · *ms* · *rooms* ["S6"] · *molecules* [45] · *atom* [0] ⇝ "N"

Haskell (*head* · *atoms* · (!!23) · *molecules* · (!!"S6") · *rooms* · *ms* · *mff* · *earth* · *sun* · (!!*milky*) · *galaxies*) *universe* ⇝ "N"

Motivation

OOP *universe* · *galaxy* (*milky*) · *sun* · *earth* · *mff* · *ms* · *rooms* ["S6"] · *molecules* [45] · *atom* [0] ⇝ "N"

Haskell (*head* · *atoms* · (!!23) · *molecules* · (!!"S6") · *rooms* · *ms* · *mff* · *earth* · *sun* · (!!*milky*) · *galaxies*) *universe* ⇝ "N"

OOP *universe* · *galaxy* (*milky*) · *sun* · *earth* · *mff* · *ms* · *rooms* ["S6"] · *molecules* [45] · *atom* [0] = "Au"

Haskell ☹

Problem

Substructure access in OOP does not really return data, but *references*.

Haskell doesn't have references (`IORef` doesn't count).

What now?

Problem

Substructure access in OOP does not really return data, but *references*.

Haskell doesn't have references (`IORef` doesn't count).

What now? Software engineering! Simulate the references!

What can we do with a reference?

- get
- set
- make a sub-reference

Which functional value can have all these properties at once?

Attempt 1:

```
data Ref big small = Ref
  { view :: big → small
  , over :: (small → small) → (big → big)
  }
```

Attempt 1

data Ref *big small* = Ref
  $\{ view :: big \to small, over :: (small \to small) \to (big \to big) \}$

data Two a = Two a a **deriving** Show

x = Ref $(\lambda(\text{Two } a \_) \to a)$ $(\lambda f\ (\text{Two } a\ b) \to \text{Two } (f\ a)\ b)$
y = Ref $(\lambda(\text{Two } \_\ b) \to b)$ $(\lambda f\ (\text{Two } a\ b) \to \text{Two } a\ (f\ b))$

t = Two 1 2

$view\ x\ t \quad \leadsto 1$
$over\ y\ (+1)\ t \quad \leadsto$ Two 1 3

$set\ x\ v = over\ x\ (const\ v)$
$set\ x\ 5\ t \quad \leadsto$ Two 5 2

Attempt 1

(Ref *v1 o1*) ‘*sub*‘ (Ref *v2 o2*) =
  Ref (*v2* ∘ *v1*) (*o1* ∘ *o2*)

t = Two (Two 1 2) (Two 3 4)

: *t x* ‘*sub*‘ *y* ⇝ Ref (Two (Two *b*)) *b*

view (*x* ‘*sub*‘ *y*) *t* ⇝ 2
over (*x* ‘*sub*‘ *y*) (∗10) *t* ⇝ Two (Two 1 20) (Two 3 4)

set r x = *over r* (*const x*)
set (*x* ‘*sub*‘ *y*) 1234 *t* ⇝ Two (Two 1 1234) (Two 3 4)

Improvements

- We want dot access with $x \cdot y \cdot x$!
- What if we want something better than OOP?
  - Safe reference to objects that may not exist?
    $left :: \mathrm{Ref}\ (\mathrm{Either}\ a\ b)\ a$?
  - Smart references to more objects (multi-edit)?
    Clamp in R: $x\ [x < 0] = 0; x\ [x > 1] = 1;$
  - Type-changing references?

Improvements

- We want dot access with $x \cdot y \cdot x$!
- What if we want something better than OOP?
  - Safe reference to objects that may not exist?
    left :: Ref (Either *a b*) *a*?
  - Smart references to more objects (multi-edit)?
    Clamp in R: $x\ [x < 0] = 0; x\ [x > 1] = 1;$
  - Type-changing references?

Let's start with the hardest part.

NB.: Because Ref *big small* represents a 'zoom' on a small object, it's usually called a Lens.

Optics, attempt 2

We have:

$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$

We want:

$$(\cdot) :: \mathsf{Lens}\ x\ y \to \mathsf{Lens}\ y\ z \to \mathsf{Lens}\ x\ z$$

Lenses thus must be functions!

$$\textbf{type}\ \mathsf{Lens}\ a\ b = \_ \to \_$$

At the same time we know:

$$set' :: (\_ \to \_) \to (b \to b) \to (a \to a)$$
$$get' :: (\_ \to \_) \to (b \to b) \to (a \to b)$$

Which type $(\_ \to \_)$ can easily produce both $(a \to a)$ and $(a \to b)$?

Optics, attempt 2

$$(\bullet) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(\bullet) :: \text{Lens } x\ y \to \text{Lens } y\ z \to \text{Lens } x\ z$$

Parameter order must be ok:

$$\text{Lens } \mathit{big}\ \mathit{small} = \text{Something } \mathit{small} \to \text{Something } \mathit{big}$$

i.e., the lens convers ‘something that works with a part’ to ‘something that works with a whole’

Optics, attempt 2

$(\bullet) :: (b \to c) \to (a \to b) \to (a \to c)$
$(\bullet) :: \text{Lens}\ x\ y \to \text{Lens}\ y\ z \to \text{Lens}\ x\ z$

Parameter order must be ok:

$$\text{Lens}\ \mathit{big}\ \mathit{small} = \text{Something}\ \mathit{small} \to \text{Something}\ \mathit{big}$$

i.e., the lens convers 'something that works with a part' to 'something that works with a whole'

We need to use this Something in:

$\mathit{set'} :: (\text{Something}\ \mathit{small} \to \text{Something}\ \mathit{big}) \to (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{big})$
$\mathit{get'} :: (\text{Something}\ \mathit{small} \to \text{Something}\ \mathit{big}) \to (\mathit{small} \to \mathit{small}) \to (\mathit{big} \to \mathit{small})$

Optics, attempt 2

$$(\bullet) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(\bullet) :: \text{Lens } x\ y \to \text{Lens } y\ z \to \text{Lens } x\ z$$

Parameter order must be ok:

$$\text{Lens } big\ small = \text{Something } small \to \text{Something } big$$

i.e., the lens convers 'something that works with a part' to 'something that works with a whole'

We need to use this Something in:

$$set' :: (\text{Something } small \to \text{Something } big) \to (small \to small) \to (big \to big)$$
$$get' :: (\text{Something } small \to \text{Something } big) \to (small \to small) \to (big \to small)$$

Something *big* must be convertible to both *small* and *big*!

Optics, attempt 2

Let's add a small decoration

$$set' :: ((small \to \text{Set } small) \to (big \to \text{Set } big)) \to (small \to \text{Set } small) \to (big \to \text{Set } big)$$
$$get' :: ((small \to \text{Get } small) \to (big \to \text{Get } big)) \to (small \to \text{Get } small) \to (big \to \text{Get } big)$$

The user may create lenses that work with a decoration, and eventually remove it.

Optics, attempt 2

Let's add a small decoration

$$set' :: ((small \to \text{Set } small) \to (big \to \text{Set } big)) \to (small \to \text{Set } small) \to (big \to \text{Set } big)$$
$$get' :: ((small \to \text{Get } small) \to (big \to \text{Get } big)) \to (small \to \text{Get } small) \to (big \to \text{Get } big)$$

The user may create lenses that work with a decoration, and eventually remove it.

After the removal, we need this to hold:

$$\begin{aligned} \text{Set } small &\rightsquigarrow small \\ \text{Set } big &\rightsquigarrow big \\ \text{Get } small &\rightsquigarrow small \\ \text{Get } big &\rightsquigarrow small \end{aligned}$$

Optics, attempt 2

Let's add a small decoration

$$set' :: ((small \to \text{Set } small) \to (big \to \text{Set } big)) \to (small \to \text{Set } small) \to (big \to \text{Set } big)$$
$$get' :: ((small \to \text{Get } small) \to (big \to \text{Get } big)) \to (small \to \text{Get } small) \to (big \to \text{Get } big)$$

The user may create lenses that work with a decoration, and eventually remove it.

After the removal, we need this to hold:

$$\begin{aligned}
\text{Set } small &\rightsquigarrow small \\
\text{Set } big &\rightsquigarrow big \\
\text{Get } small &\rightsquigarrow small \\
\text{Get } big &\rightsquigarrow small
\end{aligned}$$

$$\begin{aligned}
id\ S &\rightsquigarrow S \\
id\ B &\rightsquigarrow B \\
const\ S\ S &\rightsquigarrow S \\
const\ S\ B &\rightsquigarrow S
\end{aligned}$$


Vorsicht
Funktor
2 m Abstand halten

Optics, attempt 2

$$
\begin{aligned}
&\mathit{runIdentity} :: \text{Identity } \mathit{small} \to \mathit{small} \\
&\mathit{runIdentity} :: \text{Identity } \mathit{big} \to \mathit{big} \\
&\mathit{getConst} :: (\text{Const } \mathit{small})\ \mathit{small} \to \mathit{small} \\
&\mathit{getConst} :: (\text{Const } \mathit{small})\ \mathit{big} \to \mathit{small}
\end{aligned}
$$

We get

```
set' :: ((small → Identity small) → (big → Identity big)) → (small → Identity small) → (big → Identity big)
set' = id
get' :: ((small → Const small small) → (big → Const small big)) → (small → Const small small) → (big → Const small big)
get' = id
```

$$
\begin{aligned}
&\mathit{over}\ \mathit{lens}\ f = \mathit{runIdentity} \cdot \mathit{lens}\ (\text{Identity} \cdot f) \\
&\mathit{view}\ \mathit{lens}\ f = \mathit{getConst} \cdot \mathit{lens}\ (\text{Const} \cdot f)
\end{aligned}
$$

What got better?

Naive reference:

$$\textbf{data}\ \mathrm{Ref}\ b\ s = \{ view :: b \to s,\ over :: (s \to s) \to (b \to b) \}$$

Problems:

- What if someone invents 3rd operation?
- No possibility to refer to more items
- The operation of 'picking the item' is implemented twice

Functorial reference:

$$\mathrm{Lens}\ b\ s :: \forall f.\mathrm{Functor}\ f \Rightarrow (s \to f\ s) \to b \to f\ b$$

- Implements a single operation $b \to (s, s \to b)$
- Pipeline: $b \rightsquigarrow s \rightsquigarrow f\ s \rightsquigarrow f\ b$
- Operations can be implemented by throwing in a functor
  $\mathrm{Identity}\ b \equiv b, (\mathrm{Const}\ s)\ b \equiv s, \ldots$

Lenses — implementation

Actual implementation of functorial lenses:

$$\text{Lens } big\ small :: \forall f.\text{Functor } f \Rightarrow (small \to f\ small) \to big \to f\ big$$

- gets a function that is able to wrap a small 'part' into a 'whole'
- should produce a function that is able to rewrap the 'whole'
- the implementation is strictly type-directed:
  1. we get the whole (as the 2nd parameter, of type *big*)
  2. we cut out a piece of type *small*
  3. we can have *small* wrapped into *f small* (using 1st parameter)
  4. from *f small* we want to make *f big*, but we only know *small* → *big*
  5. because *f* is a functor, we can do this with *fmap*

Lens can be trivially made from a getter and a setter.

Lenses — implementation

Actual implementation of functorial lenses:

$$\text{Lens } big\ small :: \forall f.\text{Functor } f \Rightarrow (small \to f\ small) \to big \to f\ big$$

- gets a function that is able to wrap a small 'part' into a 'whole'
- should produce a function that is able to rewrap the 'whole'
- the implementation is strictly type-directed:
  1. we get the whole (as the 2nd parameter, of type *big*)
  2. we cut out a piece of type *small*
  3. we can have *small* wrapped into *f small* (using 1st parameter)
  4. from *f small* we want to make *f big*, but we only know *small* → *big*
  5. because *f* is a functor, we can do this with *fmap*

Lens can be trivially made from a getter and a setter.

Lenses — implementation

Actual implementation of functorial lenses:

$$\text{Lens } big\ small :: \forall f.\text{Functor } f \Rightarrow (small \to f\ small) \to big \to f\ big$$

- gets a function that is able to wrap a small 'part' into a 'whole'
- should produce a function that is able to rewrap the 'whole'
- the implementation is strictly type-directed:
  1. we get the whole (as the 2nd parameter, of type *big*)
  2. we cut out a piece of type *small*
  3. we can have *small* wrapped into $f\ small$ (using 1st parameter)
  4. from $f\ small$ we want to make $f\ big$, but we only know $small \to big$
  5. because f is a functor, we can do this with *fmap*

Lens can be trivially made from a getter and a setter.

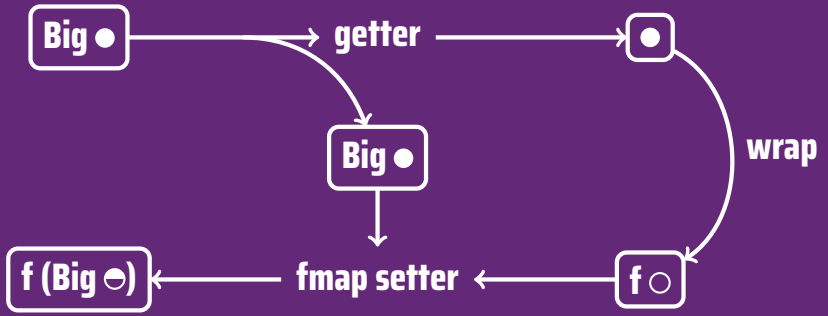
Lenses — implementation

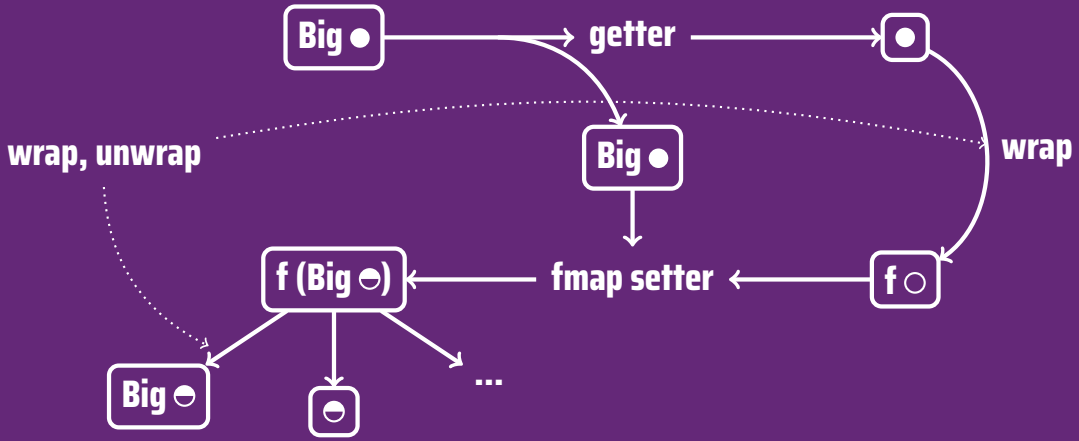
Actual implementation of functorial lenses:

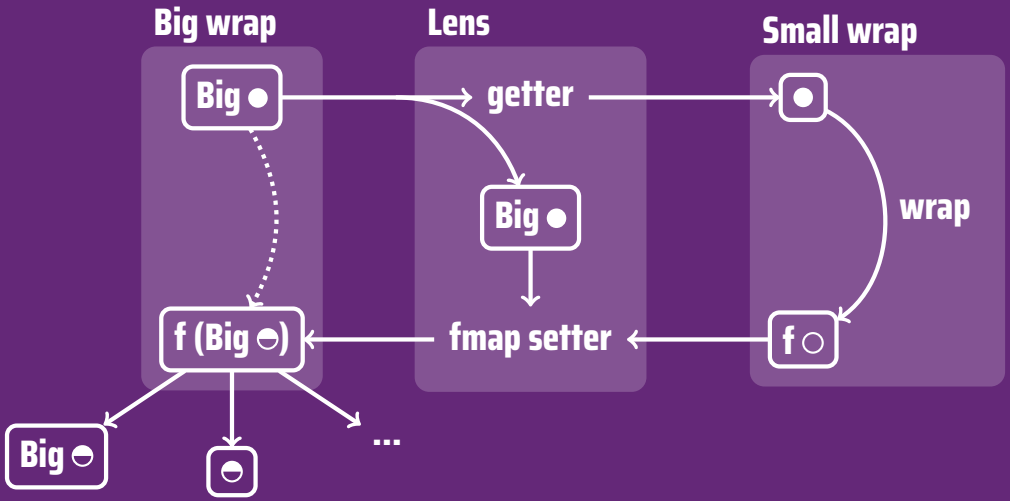
$$\text{Lens } big\ small :: \forall f.\text{Functor } f \Rightarrow (small \to f\ small) \to big \to f\ big$$

- gets a function that is able to wrap a small 'part' into a 'whole'
- should produce a function that is able to rewrap the 'whole'
- the implementation is strictly type-directed:
  1. we get the whole (as the 2nd parameter, of type *big*)
  2. we cut out a piece of type *small*
  3. we can have *small* wrapped into *f small* (using 1st parameter)
  4. from *f small* we want to make *f big*, but we only know *small* → *big*
  5. because *f* is a functor, we can do this with *fmap*

Lens can be trivially made from a getter and a setter.


Big ●
getter
●
wrap
Big ●
f (Big ◓)
fmap setter
f ○


Big ●
getter
●
wrap, unwrap
Big ●
wrap
f (Big ◓)
fmap setter
f ○
Big ◓
◓
...


Big wrap
Lens
Small wrap
Big
getter
Big
wrap
f (Big)
fmap setter
f
Big
...

Lenses – result

Overall lens type (needs `-XRankNTypes`):

$$\textbf{type}\ \text{Lens}\ a\ b = \forall f.\text{Functor}\ f \Rightarrow (b \to f\ b) \to (a \to f\ a)$$

What does it look like?

$$x :: \text{Lens}\ (\text{Two}\ a)\ a$$
$$x :: \text{Functor}\ f \Rightarrow (a \to f\ a) \to (\text{Two}\ a \to f\ (\text{Two}\ a))$$
$$x\ wrap\ (\text{Two}\ a\ b) = fmap\ (\lambda a' \to \text{Two}\ a'\ b)\ \$\ wrap\ a$$
$$y\ wrap\ (\text{Two}\ a\ b) = fmap\ (\lambda b' \to \text{Two}\ a\ b')\ \$\ wrap\ b$$

Generically from a getter and setter:

$$lens :: (a \to b) \to (a \to b \to a) \to \text{Lens}\ a\ b$$
$$lens\ getter\ setter\ wrap\ a = fmap\ (setter\ a)\ \$\ wrap\ (getter\ a)$$

Lenses vs. dots

Type tetris success:

$$
\begin{aligned}
&(\bullet) :: \mathsf{Functor}\ f \\
&\quad \Rightarrow ((b \to f\ b) \to (a \to f\ a)) \\
&\quad \to ((c \to f\ c) \to (b \to f\ b)) \\
&\quad \to ((c \to f\ c) \to (a \to f\ a))
\end{aligned}
$$

$$
(\bullet) :: \mathsf{Lens}\ a\ b \to \mathsf{Lens}\ b\ c \to \mathsf{Lens}\ a\ c
$$

(a, b are flipped in `Lens` definition!)

Lenses — implementation

$$set :: \text{Lens } a\ b \to b \to a \to a$$
$$set\ l\ v = runIdentity \cdot l\ (\text{Identity} \cdot const\ v)$$

$$over :: \text{Lens } a\ b \to (b \to b) \to a \to a$$
$$over\ l\ f = runIdentity \cdot l\ (\text{Identity} \cdot f)$$

$$view :: \text{Lens } a\ b \to a \to b$$
$$view\ l = getConst \cdot l\ \text{Const}$$

Lenses — usage

$$\mathit{set}\ x :: b \to \mathsf{Two}\ b \to \mathsf{Two}\ b$$
$$\mathit{over}\ (x \bullet y)\ (+1) :: \mathsf{Num}\ b \Rightarrow \mathsf{Two}\ (\mathsf{Two}\ b) \to \mathsf{Two}\ (\mathsf{Two}\ b)$$
$$\mathit{view}\ (x \bullet y \bullet x) :: \mathsf{Two}\ (\mathsf{Two}\ (\mathsf{Two}\ b)) \to b$$

A nice bonus: Lenses compose in the more “humane” order:

$$\begin{aligned} &\mathit{set}\ (x \bullet y)\ 10\ (\mathsf{Two}\ (\mathsf{Two}\ 1\ 2)\ (\mathsf{Two}\ 3\ 4)) \\ &\quad \rightsquigarrow \quad \mathsf{Two}\ (\mathsf{Two}\ 1\ 10)\ (\mathsf{Two}\ 3\ 4) \end{aligned}$$

Lenses — usage

$$set\ x :: b \to \text{Two}\ b \to \text{Two}\ b$$
$$over\ (x \cdot y)\ (+1) :: \text{Num}\ b \Rightarrow \text{Two}\ (\text{Two}\ b) \to \text{Two}\ (\text{Two}\ b)$$
$$view\ (x \cdot y \cdot x) :: \text{Two}\ (\text{Two}\ (\text{Two}\ b)) \to b$$

A nice bonus: Lenses compose in the more “humane” order:

$$set\ (x \cdot y)\ 10\ (\text{Two}\ (\text{Two}\ 1\ 2)\ (\text{Two}\ 3\ 4))$$
$$\rightsquigarrow \quad \text{Two}\ (\text{Two}\ 1\ 10)\ (\text{Two}\ 3\ 4)$$

Author: Twan van Laarhoven (2009)

✎ https://www.twanvl.nl/blog/haskell/cps-functional-references

Edward Kmett (`import Control.Lens`)

"newtype PlanT *k i o m a* = PlanT {*runPlanT* :: $\forall r.(a \to m\ r) \to (o \to m\ r \to m\ r) \to$

$(\forall z.(z \to m\ r) \to k\ i\ z \to m\ r \to m\ r) \to m\ r \to m\ r$} is a monad I actually use."

More optics!

- *allItems* :: Lens [*a*] _
- *left* :: Lens (Either *a b*) _

More optics!

- *allItems* :: Lens [*a*] _
- *left* :: Lens (Either *a b*) _

These concepts are called Traversal and Prism.

NB.: the types of stuff are going to get a bit out of hand now. Intuition from the Functorial lenses still holds.

More optics!

- *allItems* :: Lens [*a*] _
- *left* :: Lens (Either *a b*) _

These concepts are called Traversal and Prism.

NB.: the types of stuff are going to get a bit out of hand now. Intuition from the Functorial lenses still holds.

Traversal

If we want to apply optics to multiple 'small' parts, we need to be able to reconnect the wrappers later. Applicative is the usual solution:

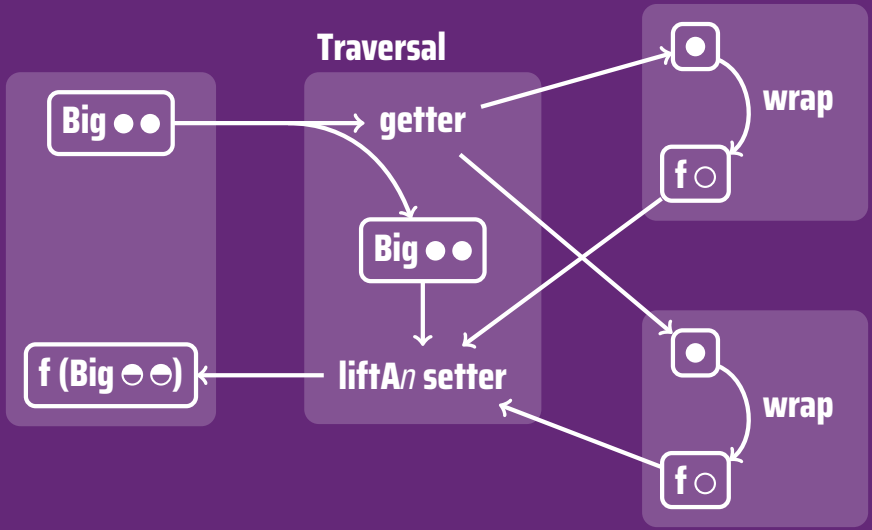
```
data Two a = Two a a deriving Show

both wrap (Two a b) = Two <$> wrap a <*> wrap b
```

Demo:

```
a = Two (Two 1 2) (Two 3 4)

set both 5 a              ⇝ Two 5 5
set (both • x) 5 a        ⇝ Two (Two 5 2) (Two 5 4)
set (x • both) 5 a        ⇝ Two (Two 5 5) (Two 3 4)
over (both • y) (+10) a   ⇝ Two (Two 1 12) (Two 3 14)
```


Traversal
Big ●●
getter
wrap
f ○
Big ●●
f (Big ◓◓)
liftA*n* setter
wrap
f ○

```
:t view both
```

```
:t view both
```

view both :: **Monoid *c* ⇒ Two *c* → *c***

...but where did the Monoid come from?!

view both :: Monoid $c \Rightarrow$ Two $c \to c$

Recall the effect-gluing instance for Const:

instance Monoid $b \Rightarrow$ Applicative (Const b)

Use:

view both (Two (Sum 1) (Sum 2))
$\rightsquigarrow$ Sum {*getSum* = 3}

view both (Two [1, 2] [3, 4])
$\rightsquigarrow$ [1, 2, 3, 4]

Traversing

Conveniently, *traverse* is a traversal for everything Traversable:

$$traverse :: (\text{Applicative } f, \text{Traversable } t) \Rightarrow (a \to f\ b) \to (t\ a \to f\ (t\ b))$$

(The type is just a little more decorated Lens $a\ b$)

$$over\ (traverse \cdot y)\ (+1)\ [\text{Two } 1\ 2, \text{Two } 3\ 4] \rightsquigarrow [\text{Two } 1\ 3, \text{Two } 3\ 5]$$

$$listOf\ l = getConst \cdot l\ (\lambda x \to \text{Const}\ [x])$$

$$listOf\ both\ \$\ \text{Two}\ (\text{Two } 1\ 2)\ (\text{Two } 3\ 4) \rightsquigarrow [\text{Two } 1\ 2, \text{Two } 3\ 4]$$

$$listOf\ (both \cdot y)\ \$\ \text{Two}\ (\text{Two } 1\ 2)\ (\text{Two } 3\ 4) \rightsquigarrow [2, 4]$$

$$listOf\ (y \cdot both)\ \$\ \text{Two}\ (\text{Two } 1\ 2)\ (\text{Two } 3\ 4) \rightsquigarrow [3, 4]$$

Traversing

SQL exercise:

ident *wr s = wr* *s* – don't do anything lens (same as id)

ignored wr s = pure s – discard changes

filtered cond f s = **if** *cond s* **then** *f s* **else** *pure s*

over (*both* ∘ *both* ∘ *filtered even*) (*'div'*2) $ Two (Two 1 2) (Two 3 4)

⇝ Two (Two 1 1) (Two 3 2)

Iso

Iso is a representation-changing lens (the name comes from isomorphisms).

```
asString :: (Show a, Read a) ⇒ Lens a String
asString ff a = fmap read $ ff (show a)

set (both • asString • traverse • filtered (≡ '3')) '5' $ Two 103 430
    ⇝ Two 105 450
```

(NB: without the explicit type, *asString* doesn't know to return the same *a*)

Prism

Prism is a special case of Traversal, that is guaranteed to match at most one target.

- *view* doesn't require monoids, instead we have *preview* with Maybe (implemented via First functor)
- it can be reversed (*review*, *re*)

Practical alternative definition: Iso that is total in only one direction.

Prism

preview *_Right* (Right 5) ⇝ Just 5
preview *_Right* (Left 5) ⇝ Nothing

view (*re* *_Right*) 3 ⇝ Right 3
review *_Right* 3 ⇝ Right 3

preview (*prefixed* `"tele"`) `"telescope"` ⇝ Just `"scope"`
preview (*prefixed* `"tele"`) `"orange"` ⇝ Nothing
review (*prefixed* `"tele"`) `"graph"` ⇝ `"telegraph"`
over (*prefixed* `"tele"`) *reverse* `"telegraph"` ⇝ `"telehparg"`
over (*prefixed* `"tele"`) *reverse* `"carrot"` ⇝ `"carrot"`

Intuition: Prisms generalize the pairing of data constructors and their pattern matches.

Lens library

`Control.Lens` implements the Van Laarhoven lenses with many interesting side benefits.

Bonusy:

- TemplateHaskell
- useful operators

Other libraries:

- `Lens.Micro` (less funcionality with negligible bloat)
- `Optics.Optic` (easier encoding, less scary types, doesn't support dot access)
- `Prolens` (encoding in profunctors gives much scarier types and lots of speed, absolutely no bloat and minimal side functionality)

@IMPUREPICS

HOW MANY LEVELS OF LENSES ARE YOU ON?

```
obj.getter()
obj.setters()
```

```
get: s -> a
set: a -> s -> s
```

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

```
type Lens s t a b =
∀p. Strong p
=> p a b
-> p s t
```

You can use TemplateHaskell to automatically make optics for your ADTs:

```
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens

data Two a = Two {_x :: a, _y :: a} deriving Show
makeLenses ''Two

x :: Functor f ⇒ (a → f a) → Two a → f (Two a)
y :: Functor f ⇒ (a → f a) → Two a → f (Two a)
```

(the underscores are a convention)

Name collisions are solved via typeclasses:

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}

import Control.Lens

data Two a = Two {_twoX :: a, _twoY :: a}
makeFields ''Two

data Three a = Three {_threeX :: a, _threeY :: a, _threeZ :: a}
makeFields ''Three

x :: (Functor f, HasX s a) => (a -> f a) -> s -> f s
```

(the prefixing with underscored constructor name is a convention)

What follows is a random review of useful lensy constructions.

Operators!

prefix	infix	State
view	`^.`	
set	`.~`	`.=`
over	`%~`	`%=`
toListOf	`^..`	
preview	`^?`	
	`+~, -~, *~`	`+=, -=, *=`

Tool: `&` is a flipped `$`.

Operators and State utils

```
(0, 1, 0, 1, 0, 1) & _1 .~ 7
                   & _2 %~ (5*)
                   & _3 .~ (-1)
                   & _4 .~ "orange"
                   & _5 +~ 2
                   & _6 *~ 3
  ⇝ (7, 5, -1, "orange", 2, 3)
```

```
runPhysics :: State SomeWorld ()
runPhysics dT =
  zoom (gameObjects . actors) $
    do Vec x y ← use (player . speed)
       player . position . x += dT * x
       player . position . y += dT * y
```


MY INNOCENT
CODEBASE
@impurepics

import Data.Aeson.Lens

Using prisms instead of (ugly!) pattern matches:

```
"[1, \"x\"]" ^? nth 0 . _Number
  ⇝ Just 1.0
```

```
"[1, \"x\"]" ^? nth 1 . _Number
  ⇝ Nothing
```

```
"[10.5]" ^? nth 0 . _Integer
  ⇝ Just 10
```

```
"[{\"x\":1, \"y\":2}, {\"x\":3, \"y\":[]}]"
  ^.. values . members . _Integer
  ⇝ [1,2,3]
```

```
"[{\"x\":1, \"y\":2}, {\"x\":3, \"y\":4}]"
  ^.. values . key "x" . _Double
  ⇝ [1.0,3.0]
```

Indexes, filters, ...

```
import Control.Lens.Indexed
import Control.Lens.Combinators
```

Indexed lenses contain a 'hidden' index that can be extracted and accessed later

```
['a'..'z'] ^.. itraversed . filtered (∈ "hello")
    ⇝ "ehlo"
['a'..'z'] ^.. itraversed . filtered (∈ "hello") . withIndex
    ⇝ [(4,'e'),(7,'h'),(11,'l'),(14,'o')]
['a'..'z'] ^.. itraversed . indices odd
    ⇝ "bdfhjlnprtvxz"
```

Cool utilities:

```
has (element 10) [1,2,3]    ⇝ False
is _Left (Right 5)          ⇝ False
hasn't _Left (Right 5)      ⇝ True
```

```
"Some random words" & worded
  %~ show . length
    ⇝ "4 6 5"
"AAAAARGGGHH!" & _tail . mapped
  %~ toLower
    ⇝ "Aaaaargggghh!"
```

IO!

What else can the GHC runtime do for you?

- UNIX-like files & networking
- concurrent computation and synchronization
- parallel computation
- references and actual pointers
- FFI

Communication of the user with the program

- environment variables ('env')

```
import System.Environment

getEnv :: String → IO String
lookupEnv "PATH" :: IO (Maybe String)
getEnvironment :: IO [(String, String)]
```

Communication of the user with the program

- environment variables ('env')

```
import System.Environment

getEnv :: String → IO String
lookupEnv "PATH" :: IO (Maybe String)
getEnvironment :: IO [(String, String)]
```

- commandline parameters

```
getArgs :: IO [String]
getProgName :: IO String
getExecutablePath :: IO FilePath
```

Communication of the user with the program

- exit-code

```
import System.Exit

die "okay sorry" :: IO a

exitWith ExitSuccess :: IO a
exitWith (ExitFailure 3) :: IO a
```

Communication of the user with the program

- exit-code

```
import System.Exit

die "okay sorry" :: IO a

exitWith ExitSuccess :: IO a
exitWith (ExitFailure 3) :: IO a
```

- reading and writing from/to file descriptors

```
import System.IO

openFile "test" WriteMode :: IO Handle
hClose h :: IO ()

hGetChar h :: IO Char
hPutChar h 'c' :: IO ()
```

Communication of the user with the program

- sending and catching of signals

```
import System.Posix.Signals

raiseSignal sigSTOP :: IO ()
signalProcess sigKILL 1 :: IO ()
```

Communication of the user with the program

- sending and catching of signals

```
import System.Posix.Signals

raiseSignal sigSTOP :: IO ()
signalProcess sigKILL 1 :: IO ()


data Handler = Default | Ignore
    | Catch (IO ()) | CatchOnce (IO ()) | ...
installHandler sigTERM
   (Catch $ putStrLn "nope")
   Nothing
    :: IO Handler
```

(package `unix` contains most of the other low-level primitives)

How do you recognize a high-quality UNIX program?

How do you recognize a high-quality UNIX program?

It prints out an useful `--help`.

optparse-applicative

Abstraction: We will remember the optional commandline argument properties as monoids. The parameter parsers can be applicatively connected to a commandline parser that directly fills in a given data structure.

optparse-applicative

Abstraction: We will remember the optional commandline argument properties as monoids. The parameter parsers can be applicatively connected to a commandline parser that directly fills in a given data structure.

optparse-applicative

Abstraction: We will remember the optional commandline argument properties as monoids. The parameter parsers can be applicatively connected to a commandline parser that directly fills in a given data structure.

optparse-applicative

Abstraction: We will remember the optional commandline argument properties as monoids. The parameter parsers can be applicatively connected to a commandline parser that directly fills in a given data structure.

```
import Options.Applicative
import Data.Semigroup ((◇))

theOption :: Parser Bool
theOption = switch $ long "good"
                   ◇ short 'g'
                   ◇ help "Should it be really good?"

optInfo = info (helper ⟨∗⟩ theOption)
  $ progDesc "good shows current level of goodness"
  ◇ header "A program that shows if it's good."
  ◇ footer "See more information on www.good.program.hu"
```

optparse-applicative

Abstraction: We will remember the optional commandline argument properties as monoids. The parameter parsers can be applicatively connected to a commandline parser that directly fills in a given data structure.

```
import Options.Applicative
import Data.Semigroup ((◇))

theOption :: Parser Bool
theOption = switch $ long "good"
                   ◇ short 'g'
                   ◇ help "Should it be really good?"

optInfo = info (helper ⟨∗⟩ theOption)
  $ progDesc "good shows current level of goodness"
  ◇ header "A program that shows if it's good."
  ◇ footer "See more information on www.good.program.hu"
```

optparse-applicative

Abstraction: We will remember the optional commandline argument properties as monoids. The parameter parsers can be applicatively connected to a commandline parser that directly fills in a given data structure.

```
import Options.Applicative
import Data.Semigroup ((◇))

theOption :: Parser Bool
theOption = switch $ long "good"
                   ◇ short 'g'
                   ◇ help "Should it be really good?"

optInfo = info (helper ⟨∗⟩ theOption)
  $ progDesc "good shows current level of goodness"
  ◇ header "A program that shows if it's good."
  ◇ footer "See more information on www.good.program.hu"
```

optparse-applicative

```
main = do opt ← execParser $ optInfo
          putStrLn $ if opt
                     then "yeah it's good"
                     else "not good at all!"
```

```
$ ./good
not good at all!
$ ./good -x
Invalid option `-x'

Usage: ./good [-g|--good]
$ ./good -g
yeah it's good
```

optparse-applicative

```
main = do opt ← execParser $ optInfo
          putStrLn $ if opt
                     then "yeah it's good"
                     else "not good at all!"
```

```
$ ./good
not good at all!
$ ./good -x
Invalid option `-x'

Usage: ./good [-g|--good]
$ ./good -g
yeah it's good
```

```
$ ./good -h
A program that shows if it's good.

Usage: good [-g|--good]
  good shows current level of goodness

Available options:
  -h,--help                Show this help text
  -g,--good                Should it be really good?

See more information on www.good.program.hu
```

- Strings

```
optOutput = strOption
  $ long "output"
  ◇ short 'o'
  ◇ metavar "FILE"
  ◇ value "out.txt"
  ◇ showDefault
  ◇ help "Write output to FILE"
```

- Integers

```
optLevel = option
  $ long "level"
  ◇ short 'l'
  ◇ help "Level of whatever"
```

- combination (product)

```
data CmdLineOpts = CmdLineOpts {level :: Int,
                                output :: String}

allOptions = CmdLineOpts <$> optLevel <*> optOutput
```

- combination of alternatives

```
data OptionalOutput = FileOutput String | StdOutput

optionOptionalOutput = FileOutput <$> optOutput
  <|> flag' StdOutput (long "stdout"
                       <> help "write to stdout")
```