


[artyom.me]



Hello, I'm Artyom. You can write me at yom@artyom.me, in Telegram ([@lightgreen](#)), Skype ([mayangreen](#)), or in IRC (puregreen at freenode). You can write me if you're bored, I like chatting with random people. Or you can [hire me](#). I'm also [on Github](#) (but most of the projects are [here](#)). And I've got a [Telegram channel](#) where I'm posting stuff that is too small to be posted here, as well as links, quotes, and so on.

This site was last updated on **April 27, 2017** under intriguing and mysterious circumstances. If you're unsure whether I'm alive (or at least have internet), see my [Last.FM](#). Here's a [random TVTrope](#), [xkcd comic](#), and [kitten](#) for you.

typo? bad phrasing? mistake? select and press Ctrl-Enter or (or [open an issue on Github](#))

<<< “lens over tea” >>>

lens over tea #1: lenses 101, traversals 101, and some implementation details

Okay, so I never could really understand how [lens](#) works. This is an attempt to understand lens as completely as possible, including the ideas behind it, the implementation, and the design choices. (A later update: it's not that good for actually learning how to use lenses, and frankly should be fully rewritten, but I don't have the time.)

There are already articles which explain how to *use* lenses and how to *make* basic lenses, but some things don't seem to be explained well anywhere:

- zooming
- prisms/isos
- indexed stuff
- implementation details
- category theory “mumbo-jumbo”
- performance

(or they are explained but in such a way that an “ordinary person” can't understand them). That's why I decided to write a series of articles about lens. I hope I'll learn something in the process, and maybe you will too (if I don't completely suck at explaining stuff, that is).

Regarding implementation details: I spend perhaps too much time discussing those, so if you're reading these posts just to learn how to use lenses, it might be a waste of time and I recommend you to look for another tutorial.

I'm not assuming that you've ever used lenses, or know category theory, or profunctors, or anything of the sort. I *am* assuming that you can use `Monoid`, `Functor`, `Applicative` and `Monad`, but nothing beyond that; if you don't know these either, read

- [LYAH on functors and monoids](#)
- [LYAH on monads](#)
- a [School of Haskell tutorial](#)
- an [explanation with pictures](#) if you like pictures (I don't, but maybe you do, who knows).

To be honest, tho, you probably shouldn't be touching lens until you know those things – unless you only need lenses to do record updates and other simple stuff, in which case read [Gabriel Gonzales's lens tutorial](#) and the [Wikibook chapter on lenses](#) (which is actually a good lenses tutorial in general).

Disclaimers

I tried to write in such a way that I myself (the *past* self, I mean) would've been able to understand what I've written – which means explaining everything many times. When something is explained only once, the reader has to think while reading, and there's nothing worse than having to think while reading.

...Fine, maybe there was a bit of sarcasm in the last sentence, but I definitely know that there is at least 1 person (yeah, me) who only reads articles while drinking tea or waiting for something to compile. So, in order for me to learn anything, there *has* to be an article about it somewhere which can be read while drinking tea – and this is an article about lenses which can be read while drinking tea.

Disclaimer #1: you might need more than 1 cup.

Disclaimer #2: as usual with my posts, I've no idea what I'm writing about, so, if anybody says I'm wrong about something, I'm most likely indeed wrong about it.

Disclaimer #3: this post is an exploration more than an explanation, so don't expect the text to be structured the way normal blog posts are. If I knew how to structure it all in such a way that it wouldn't be a tangled braindump, I would've done that.

Okay, enough of ~~chit-chat~~ important and appropriate disclaimers. Here goes.

Lenses 101

A lens allows us to get some value from a Big Value, and to update some value in the Big Value. (Admittedly, it's a *slightly* simplistic view, but it'll do for now.) E.g. `(1, True)` is a “big” value which contains `1` and `True`. Here's how to get `1` from `(1, True)`:

```
> fst (1, True)
1
```

However, `fst` is only a *getter* – it doesn't allow updating the value. Moreover, there's no `setFst` function in Prelude, which means that changing `1` to `8` in `(1, True)` is a huge, giant, enormous pain:

```
> (8, snd (1, True))
(8, True)
```

It might not *seem* like “huge, giant, enormous pain”, but it's still like, say, a small papercut – and a thousand papercuts *are* a pain. Moreover, for me even one papercut is a pain...

Okay, fine, sorry for sharing my stupid phobia of paper cuts with you, but please at least agree that -having separate functions to get a value and set a value- isn't all that nice. Can we do better?

Getter + setter = lens

One solution is to combine a getter and a setter in a single structure:

```
data Lens s a = Lens
  { getter :: s -> a
  , setter :: a -> s -> s }
```

While we're at it, let's write our first lens – the `ix` lens, which accesses the *i*-th element of a list.

```
setIth :: Int -> a -> [a] -> [a]
setIth index new list
  | index < 0      = error "setIth: negative index"
  | null list      = error "setIth: index too large"
  | old:rest <- list = if index == 0
                        then new : rest
                        else old : setIth (index-1) new rest

ix :: Int -> Lens [a] a
ix i = Lens { getter = (!! i)
             , setter = setIth i }
```

I don't like this approach either. Think about it: to increment the 1000th element of a list, we have to find it using `getter`, then increment it, then `setter` would have to find it *again* in order to set the new value. It's okay when we're talking about tuples or `Map`s, but when I think about having to traverse a long list twice in order to do something with a value, I feel slightly bad.

The fix is easy – it's enough to replace `setter :: a -> s -> s` with `modifier :: (a -> a) -> s -> s`. However, I have another objection: in order to get to the value, `modifier` *still* has to reimplement the functionality of `getter`. Code duplication is wrong. Can we do better?

Removing the getter

Hm... what about making `modifier` return the value before the modification? Then `getter` isn't even needed:

```
import Data.Bifunctor
```

```

type Lens s a = (a -> a) -> s -> (a, s)

-- ix :: Int -> (a -> a) -> [a] -> (a, [a])
ix :: Int -> Lens [a] a
ix index f list
  | index < 0          = error "ix: negative index"
  | null list          = error "ix: index too large"
  | old:rest <- list = if index == 0
                      then (old, f old : rest)
                      else second (old:) $ ix (index-1) f rest

```

- `second` is a function from `Data.Bifunctor` (you can find it in [bifunctors](#) if you're not on GHC 7.10 or later yet), which can be used to apply a function to the second element of a tuple (among other things):

```

> second ("second " ++) (True, "element")
(True, "second element")

```

So, the line

```
second (old:) $ ix (index-1) f rest
```

is the same as

```

let (x', s') = ix (index-1) f rest
in  (x', old : s')

```

- We could use `second` from `Control.Arrow` instead, but I chose `Data.Bifunctor` because bifunctors are much easier to understand than arrows.
- In a nutshell, a bifunctor is merely a functor with 2 parameters instead of 1. `Either` and `(,)` are both bifunctors (which means that you can use `first` and `second` on things like `Either a b` or `(a, b)`).
- If you're getting bored, here's a question for you: is `(->)` a bifunctor?
- (Yeah, we could use `let` and avoid scary bifunctors entirely, but I wanted to have an excuse to introduce them. Or we could use `fmap` since it's the same as `second` for tuples – but I don't like using `fmap` as a sort of a magic wand. It's better to state your intentions explicitly by using `map` for lists, or `second` for tuples, or `.` for functions, etc.)

Now, to get the value using `ix`, it's enough to give our lens `id` as the “modifying” function (actually, we can even give `undefined` – it doesn't matter):

```
> fst $ ix 3 id [7, 4, 1, 8]
8

> ix 3 undefined [7, 4, 1, 8]
(8, [7, 4, 1, *** Exception: Prelude.undefined

> fst $ ix 3 undefined [7, 4, 1, 8]
8
```

To set the value, we can use `const`:

```
> snd $ ix 3 (const 1000) [7, 4, 1, 8]
[7, 4, 1, 1000]
```

Okay, cool. We have a way to get, set and modify the *i*th element of a list, and we're not duplicating any code. Next objection: what if we want to change the value several times?

Monads to the rescue!

Hm, sorry, I'm not explaining very clearly. So, again:

- Imagine that there's a list.
- Traversing it once takes 1s of time (it's a huge list).
- And you for whatever reason need 1000 lists, all differing in *one* element, like this:

```
--
      ↓
[ [6, 1, 0, 3, 2, ..., 100, 8, 1, 0, ...]
, [6, 1, 0, 3, 2, ..., 233, 8, 1, 0, ...]
, [6, 1, 0, 3, 2, ..., 754, 8, 1, 0, ...]
, [6, 1, 0, 3, 2, ..., 138, 8, 1, 0, ...]
-- and so on
```

- So, currently you'll have to call the lens 1000 times, and each time it would have to traverse the list again to get to the element you want to change.
- That's not nice.

Let's change the `Lens` type again:

```
-- This is needed so that we can have constraints in type synonyms.
{-# LANGUAGE RankNTypes #-}

type Lens s a = Monad m => (a -> m a) -> s -> (a, m s)
```

How does it work? Well, everybody nowadays seems to know that `[]` is a monad (if you don't – read a chapter of [LYAH](#)), and this is how the type of `ix` looks now when it's been specialised to use the list monad:

```
ix :: Int -> (a -> [a]) -> [a] -> (a, [[a]])
```

Or in English:

Give me

- a position in a list of `a`s (“index”)
- a function generating several `a`s from an `a`
- a list of `a`s

and I'll give you many lists, each having the element at given index replaced by one of the elements generated from the original element. Oh, and I'll tell you what the original element was as well, in case you want to know.

Or in the Language Of Examples:

```
> ix 2 (\x -> [1..x]) [300, 100, 4, 600, 900, 400]
(4, [ [300, 100, 1, 600, 900, 400]
      , [300, 100, 2, 600, 900, 400]
      , [300, 100, 3, 600, 900, 400]
      , [300, 100, 4, 600, 900, 400] ])
```

The implementation doesn't differ much – we only add a couple of `liftM`s and that's it.

```
ix :: Int -> Lens [a] a
ix index f list
  | index < 0      = error "ix: negative index"
  | null list      = error "ix: index too large"
  | old:rest <- list = if index == 0
```

```

    then (old, liftM (: rest) (f old))
    else second (liftM (old :)) $ ix (index-1) f rest

```

Now, you might have a question: why specifically `Monad`?

Well, because then we can do all kinds of nifty things, like generating several values, or using `IO` in the modifying function, or `Maybe` ... Hm, wait. If we're only using `liftM` anyway, do we really need `Monad` here?

To hell with monads, functors to the rescue!

Ye-eah, we don't. All monads are functors (at least in theory, tho starting from GHC 7.10 [it's going to become "official"](#)), and `Functor f` is quite enough to implement `ix` and do fun stuff with it – why should we ask for more? (Spoiler: we will ask for more... but not until we need it.)

```

import Control.Applicative

type Lens s a = Functor f => (a -> f a) -> s -> (a, f s)

ix :: Int -> Lens [a] a
ix index f list
  | index < 0      = error "ix: negative index"
  | null list      = error "ix: index too large"
  | old:rest <- list = if index == 0
                        then (old, (: rest) <$> f old)
                        else second ((old :) <$>) $ ix (index-1) f rest

```

And now let's repeat. A lens allows us to *do something to a big structure* given that we know how to *do something to a part of it*. Note that “doing something” means more than “applying a function” – for instance, “randomly shuffling a list” (which requires IO) or “getting all permutations” (which you could say is done in list monad) are all examples of “doing something”.

Now, another nitpick. Is it really necessary to explicitly return the original value?

Setter is getter

Unfortunately, I don't know whether I would've guessed the right answer if I didn't know it

beforehand, but the answer is “it's not”, and here's why.

Meet `Storey` (don't bother googling the “canonical” name, it [doesn't seem to exist](#)). This is a *functor modifier* – it does what the original functor does, but it also attaches a value to it.

```
data Storey x f a = Storey x (f a)
  deriving Show

instance Functor f => Functor (Storey x f) where
  fmap f (Storey x fa) = Storey x (fmap f fa)
```

Or in English:

- `Storey x f a` is conceptually the same as `(x, f a)`.
- When `fmap` is used on a `Storey` value, it leaves the attached value alone and only changes the “main” value.

Now I'll show how to use it to get the original value. The trick lies in the different update function we'll give to the lens this time:

```
(\x -> Storey x [1..x])
```

It hides the original value – `x` – inside the functor. From now on all other operations will affect the *return value* – that is, `[1..x]` – but not the stored value; and then the stored value will get, unmodified, into the “outer world”, where we would be able to get it by unwrapping `Storey`.

So, let's remove the “original value” backdoor:

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

ix :: Int -> Lens [a] a
ix index f list
  | index < 0      = error "ix: negative index"
  | null list      = error "ix: index too large"
  | old:rest <- list = if index == 0
                      then (: rest) <$> f old
                      else (old :) <$> ix (index-1) f rest
```

And now look how `storey` lets us easily get both the original value and the updated structure in one package:

```
> ix 2 (\x -> Storey x [1..x]) [300, 100, 4, 600, 900, 400]
Storey 4 [ [300, 100, 1, 600, 900, 400]
          , [300, 100, 2, 600, 900, 400]
          , [300, 100, 3, 600, 900, 400]
          , [300, 100, 4, 600, 900, 400] ]
```

Ha!

Always, always benchmark

Just wondering: did this explanation seem legit to you?

So, currently you'll have to call the lens 1000 times, and each time it would have to traverse the list again to get to the element you want to change.

If so, this is yet another proof that you should Always Benchmark First [I think it's a quote, but I can't remember the source]. In reality, the functor-based lens is around 3x slower than an ordinary function, even if the latter is called 1000 times. Here's the reason:

- A list with its last element modified is a completely *new* list. 2 lists can share a tail, yep, but they can't share their beginnings.
- So, when you modify the last element of a list, you're creating a brand new list.
- When the functor lens changes some element 1000 times, it still has to create 1000 new lists. It's not a cheap operation.
- Both functor and ordinary lens create those lists during traversal. There's no actual difference in what's being done.
- So, the functor lens *can't* be faster, at least on lists. It would be faster for data structures where updating is cheaper than getting, but not for lists or trees (such as `Map` or `Seq`).
- One example of such a data structure is map implemented as a list of key–value pairs (updating is as simple as adding a new pair to the list, but getting requires traversing the list).

There are many reasons why lenses are awesome, but performance isn't one of them. (Not

saying lenses are slow... merely that they generally aren't faster than “usual” functions.)

Why preserve the type?

Nothing actually prevents us from changing the *type* of the part we're modifying as well (you might've noticed it by yourself already, but I still wanted to point this out). So, the actual type for lenses is:

```
type Lens s t a b = Functor f => (a -> f b) -> s -> f t

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

(`Lens'` means something like “simple lens”. Also, you can read `s` and `t` as “source” and “target”.)

Composing functors

You might wonder: shouldn't it be possible to somehow reuse the `(,)` instance for `Functor`? After all, the functor instance for `(,)` applies the function to the second part of the tuple:

```
> fmap not (1, False)
(1, True)
```

which is pretty similar to what `fmap` does to `Storey`:

```
fmap f (x, a) = (x,) $ f a
fmap f (Storey x a) = Storey x $ fmap f a
```

If you think about it a bit, `Storey` kinda looks like a *composition* of `(,)` and a functor. If we pretend for a moment that we can use `.` and `$` on types...

- `data Storey x f a = Storey x (f a)`
- `data Storey x f a = Storey x . f $ a`
- `data Storey x f = Storey x . f`

And since we already know that `Storey x` is like `(,) x ...`

- `data Storey x f = (,) x . f`

So, is there some kind of `.` for types? Yes, and it's called (kinda appropriately) `Compose`. Using it is really simple:

```
type Storey x f = Compose ((,) x) f
```

Actually, we don't even need this type declaration:

```
> -- Compare with the original:
> --           ix 2 (\x -> Storey x [1..x]) [300, 100, 4, 600, 900, 400]
> getCompose $ ix 2 (\x -> Compose (x, [1..x])) [300, 100, 4, 600, 900, 400]
(4, [ [300, 100, 1, 600, 900, 400]
      , [300, 100, 2, 600, 900, 400]
      , [300, 100, 3, 600, 900, 400]
      , [300, 100, 4, 600, 900, 400] ])
```

Do you completely understand what's going on?

- `f` and `g` may be functors, but `type FG a = f (g a)` won't automatically become one.
- However, `type FG a = Compose f g a` (or `type FG = Compose f g`) will.
- In our example, we couldn't use `ix` on `(\x -> (x, [1..x]))` because `(x, [1..x])` has type `(a, [a])`, which is a composition of 2 functors – `(,) a` and `[]` – but isn't a `Functor` itself.
- Applying `Compose` to `(x, [1..x])` is enough to make it a proper functor. Then `ix` can do its job.
- `getCompose` extracts the “functor-of-functor” value from `Compose`.

Good, now to the next question.

What if I don't need any functors?

If you just want to set or modify the value, use `Identity`, which is precisely the functor to use when you don't actually need one.

```
> -- Setting.
> runIdentity $ ix 2 (\x -> Identity 88) [0..4]
[0, 1, 88, 3, 4]

> -- Modification.
> runIdentity $ ix 2 (\x -> Identity (x * 44)) [0..4]
[0, 1, 88, 3, 4]
```

Since it happens ~~kinda~~ really often that you won't need a functor, it's better to have a function to do the wrapping–unwrapping of `Identity` for you. In lens it's called `over`:

```
over :: Lens s t a b -> ((a -> b) -> s -> t)
over l f = runIdentity . l (Identity . f)
```

So, our 2 examples can be rewritten like

```
> over (ix 2) (const 88) [0..4]
[0, 1, 88, 3, 4]

> over (ix 2) (* 44) [0..4]
[0, 1, 88, 3, 4]
```

What if I don't need any modifications?

Oka-ay. If you only want to get the value, without updating anything, you can use `Const`, which is like our `Storey` but without the functor part – and it's also in the base libraries. Compare: this is the `Storey` version of a function which uses a lens to extract some value:

```
getByStorey lens s = x
  where
    Storey x _ = lens (\x -> Storey x (Identity x)) s
```

And this is the `Const` version:

```
getByConst lens s = x
  where
    Const x = lens (\x -> Const x) s
```

(`getByConst` is called `view` in lens.)

Please note that the main difference between `Const` and `Storey` is in their *constructors*, not types. Speaking in ~~silly~~ easily comprehensible metaphors, `Storey` is a box containing humanitarian aid, with a letter glued to the box, while `Const` is the same box with the same letter, except that... there's no humanitarian aid, because the box is too small for it. But you still insist that it *is* a box of humanitarian aid (merely containing 0 items) and therefore you are entitled to a free delivery (because your country has a law stating that every citizen can use post services free of charge if they're sending humanitarian aid). Your parcel is accepted, but the post workers hate you. You don't care – your grandpa in Africa would love to get your letter, nothing else matters.

Here are the definitions of `Const` and `Storey` for you to compare, and after that we will move on.

```
data Storey x f a = Storey x (f a)
data Const  x a   = Const  x
```

(By the way, we could implement `Storey` using `Const` and `Product` instead of `(,)` and `Compose`. Bonus points if you do it without opening the link.)

Test yourself

At this point you should be able to write the following (you can copy and paste this whole chunk into GHCi and it would work because I replicated type synonyms and imports as well) (oh, and there are `typed holes` added for your convenience, but if you write these functions merely by blindly following the types, it doesn't count):

```
{-# LANGUAGE
RankNTypes,
TupleSections
 #-}

import Control.Applicative

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

-- _1 :: Functor f => (a -> f b) -> (a, x) -> f (b, x)
```

```

_1 :: Lens (a, x) (b, x) a b
_1 = _

-- _2 :: Functor f => (a -> f b) -> (x, a) -> f (x, b)
_2 :: Lens (x, a) (x, b) a b
_2 = _

-- Make a lens out of a getter and a setter.
lens :: (s -> a) -> (s -> b -> t) -> Lens s t a b
lens get set = _

-- Combine 2 lenses to make a lens which works on Either. (It's a good idea
-- to try to use bimap for this, but it won't work, and you have to use
-- explicit case-matching. Still a good idea, tho.)
choosing :: Lens s1 t1 a b -> Lens s2 t2 a b
          -> Lens (Either s1 s2) (Either t1 t2) a b
choosing l1 l2 = _

-- Modify the target of a lens and return the result. (Bonus points if you
-- do it without lambdas and defining new functions. There's also a hint
-- before the end of the section, so don't scroll if you don't want it.)
(<%~) :: Lens s t a b -> (a -> b) -> s -> (b, t)
(<%~) l f s = _

-- Modify the target of a lens, but return the old value.
(<<%~) :: Lens s t a b -> (a -> b) -> s -> (a, t)
(<<%~) l f s = _

-- There's a () in every value. (No idea what this one is for, maybe it'll
-- become clear later.)
united :: Lens' s ()
united = _

```

Please stop drinking your tea now and *do* it. No, really, even if it's a totally stupid exercise. Haven't you ever been in a situation when you have to do something simple, and you look at the screen/keyboard and realise that you don't know how to start? You stare at the letters and symbols – “a”, “s”, “tuple”, “this goes there”, “if this accepts this then it would be like *this*”... – but brain refuses to do its job. You are despised by your peers and laughed at by your enemies. To not let this happen, *please* write the definitions for these functions.

Thanks.

[Now I'm wondering whether to admit that `united` took me more than 2min...]

Here's a hint for `<%~`: it involves the tuple functor.

Recap

(I promised explaining everything many times but didn't promise that explanations would be different every time, right?)

- Lenses are like `modify :: (a -> b) -> s -> t`, where `a` is a part of `s` (“source”), and `t` (“target”) is what happens to `s` when the `a`-typed subpart is replaced by the `b`-typed subpart. Except that lenses are actually like `ultraModify :: Functor f => (a -> f b) -> s -> f t ...` and this `f` here enables us to do *lots* of cool stuff.
- If you want a lens to be like an ordinary `modify`, use `Identity` in place of `f`. This is done by `over` in lens.
- If you want to use the lens as a getter, use `Const` as `f` – it would store the `a` value and “carry it to the outside” for you. This is done by `view` in lens.
- You *don't* need any special `overM` to update the value using `IO` or something – just apply the lens directly. `over` is merely a convenient shortcut to wrap and unwrap `Identity`.
- You can create a lens from a getter and a setter, but it might be less than optimal (because this way `modify` would necessarily be a combination of `get` and `set`, instead of a single “dive” into the data structure).

Bonus stuff:

- Functors can be composed with `Compose`.
- `second` from `Data.Bifunctor` or `Control.Arrow` maps over the second element of a tuple.
- `Bifunctors` are like functors with 2 type parameters.

Traversals 101

A lens focuses on a single value – it doesn't matter whether it's *actually* a single value or not, as long as *conceptually* it is. A couple of examples to illustrate what I'm talking about:

- We can make a lens for changing the absolute value of a number (which technically isn't even “contained” in the number):

```
_abs :: Real a => Lens' a a
_abs f n = update <$> f (abs n)
  where
    update x
      | x < 0      = error "_abs: absolute value can't be negative"
      | otherwise = signum n * x
```

To demonstrate, I'll square the `10` contained in `-10`:

```
> over _abs (^2) (-10)
-100
```

- We can make a lens for focusing on several elements in the list, as long as they are equal to the given one:

```
_all :: Eq a => a -> Lens' [a] a
_all ref = lens get set
  where
    get s      = ref
    set s new = map (\old -> if old == ref then new else old) s
```

To demonstrate, I'll change all `0` s in a list into `-8` s:

```
> set (_all 0) (-8) [100, 600, 0, 200, 0]
[100, 600, -8, 200, -8]
```

Note that `_all`'s behavior is probably different from what you expect. For instance, let's say I want to ask user for the replacement value each time I stumble upon a `0`:

```
> (_all 0) (const $ putStr "? new: " >> readLn) [100, 600, 0, 200, 0]
? new: 13
[100, 600, 13, 200, 13]
```

The user was asked for the value only once, but there are 2 `IO`s in the list. How come? Well, it's simple: a lens depends on a functor (`IO` in this case), and you *can't* execute an action several times using only the `Functor` interface.

It wasn't obvious to me, so it might not be obvious to you either... and thus I'd better elaborate. Let's say you have `launchMissiles :: IO ()`:

```
launchMissiles = putStrLn "kaboom!"
```

Using `Monad`, you can launch missiles twice:

```
> launchMissiles >> launchMissiles
kaboom!
kaboom!
```

You can even do it using `Applicative`:

```
> launchMissiles *> launchMissiles
kaboom!
kaboom!
```

But you can't do it using `Functor`; the only thing you can do is replace the returned `()` (what for? I've no idea):

```
> ":( " <$ launchMissiles
kaboom!
":( "
```

The road to power

The last example (with `_all`) makes one wonder: what additional power would we get if `Functor` in the definition of `Lens` was replaced with `Applicative`? Well, for one, we'd be able to write a better `_all`.

First, type synonyms for our “applicative lenses”:

```
type AppLens s t a b = Applicative f => (a -> f b) -> s -> f t
type AppLens' s a = AppLens s s a a
```

Now, here's how the type of `_all'` looks like:

- unexpanded: `_all' :: Eq a => a -> AppLens' [a] a`

- expanded:

```
_all' :: (Applicative f, Eq a) => a -> (a -> f a) -> [a] -> f [a]
```

A clarification about `_all`

`_all` is written using `lens` and simple getters and setters. Even if you have written `lens` after I asked you to, it still might not be clear to you how `_all` works (it wasn't for me). If it *is* very clear to you how `_all` works, skip this section; otherwise, read on.

- `fmap` allows acting on a value inside the functor:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

- The `_all` lens is provided with a value of type `a` (the “reference value”), a function of type `a -> f a`, and the list:

```
_all ref f list = ...
```

- Since we consider all equal values to be the *same* value, we can immediately combine `ref` and `f` to get the updated value:

```
f'new = f ref
```

We have to do it *anyway*, because there's no way to get a value of type `f a` otherwise (e.g. the given list might be empty; note that we can't use `pure` as it's in `Applicative` and not `Functor`). For instance, this means that in case of `IO`, the action would be executed even if there are no values equal to `ref` in the list.

- If we have the `ref` value and the `new` value, we can do a replacement in the list without resorting to functors at all – it's simply

```
getModifiedList new = map (\old -> if old == ref then new else old) list
```

- We don't have the `new` value, but we have `f'new` which contains it – so we can do

`getModifiedList <$> f'new` to get a value of type `f [a]`. The end.

Writing a better `_all`

This time we want to do it without “cheating”, and honestly call `f` for each of the values we'll be replacing. So, we can already write the update function:

```
_all' :: Applicative f => ...
_all' ref ...
  where
    -- update :: a -> f a
    update old = if old == ref then f old else pure old
```

The only thing left is to apply it to the list and gather the results! If `f` was a monad, we could've use `mapM`:

```
_all' :: (Monad m, Eq a) => a -> (a -> m a) -> [a] -> m [a]
_all' ref f s = mapM update s
  where
    update old = if old == ref then f old else return old
```

(Do you see now how lenses are really just ordinary functions in clever disguises? If `_all'` was in some list library, betcha it would've been called `mapOnEqM` or something.)

However, `f` is an applicative functor and not a monad. How do you think is `mapM` for `Applicative` called? No, it's not `mapA` (I wish) – it's `traverse`.

With `traverse`, we can finally write `_all'`:

```
_all' :: Eq a => a -> AppLens' [a] a
_all' ref f s = traverse update s
  where
    update old = if old == ref then f old else pure old
```

```
> (_all' 0) (const $ putStr "? new: " >> readLn) [100, 600, 0, 200, 0]
? new: 11
? new: 22
[100, 600, 11, 200, 22]
```

Updating `view`, `over` and `set`

Unfortunately, you won't be able to use `view`, `over` and `set` with `_all'` yet. Remember their types?

```
view :: Lens s t a b -> s -> a
over :: Lens s t a b -> (a -> b) -> s -> t
set  :: Lens s t a b -> b -> s -> t
```

`Lens` won't do when we have an `AppLens`. Why? Because `AppLens`'s requirements to its arguments are *more strict* – in the same way you can't give `view` a “lens” of type `(a -> Maybe b) -> s -> Maybe t`. It's `view` that gets to choose what type it wants, not the lens.

This problem is solved by better stating what `view`, `over` and `set` need:

- `view` only uses `Const` functor – so its type would be:

```
-- old type:
--      ((a -> f a) -> s -> f s) -> s -> a
view :: ((a -> Const a a) -> s -> Const a s) -> s -> a
```

- `over` and `set` use `Identity` functor:

```
-- old type:
--      ((a -> f b) -> s -> f t) -> (a -> b) -> s -> t
over :: ((a -> Identity b) -> s -> Identity t) -> (a -> b) -> s -> t
```

(Not giving the type for `set` because it's almost the same.)

And now some convenient type synonyms (which mimic the ones in lens library):

```
type Getting s a = (a -> Const a a) -> s -> Const a s

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

Again: previously we were requiring the lens to work with *any* functor, even tho we only used `Const` and `Identity` functors. Now we only ask for what we actually use, which means that `view` and `over` can work with *more* types of lenses now.

Applying these type synonyms to the signatures gives us:

```
view :: Getting s a -> s -> a
over :: Setting s t a b -> (a -> b) -> s -> t
set  :: Setting s t a b -> b -> s -> t
```

With these modifications, we can finally apply `set` to `_all'`:

```
> set (_all' 0) (-8) [100, 600, 0, 200, 0]
[100, 600, -8, 200, -8]
```

(Oh, by the way, if your definitions of `<%~` and `<<%~` are broken now, you'll have to rewrite them.)

The `view` mystery

What about `view`? What do you think would happen if you tried to apply *it* to `_all'`?

Here are the definitions of `_all'` and `view` again for your convenience:

```
view :: Getting s a -> s -> a
view l = getConst . l Const

_all' :: Eq a => a -> AppLens' [a] a
_all' ref f s = T.traverse update s
  where
    update old = if old == ref then f old else pure old
```

A-and here's the result:

```
> view (_all' 0) [0, 1, 2]

<interactive>:
No instance for (Data.Monoid.Monoid a0) arising from a use of 'it'
The type variable 'a0' is ambiguous
Note: there are several potential instances:
  instance Data.Monoid.Monoid a => Data.Monoid.Monoid (Const a b)
    -- Defined in 'Control.Applicative'
  instance Data.Monoid.Monoid () -- Defined in 'Data.Monoid'
  instance (Data.Monoid.Monoid a, Data.Monoid.Monoid b) =>
    Data.Monoid.Monoid (a, b)
    -- Defined in 'Data.Monoid'
```

```
...plus 16 others
```

Oops. Okay, let's look at the inferred type:

```
> :t view (_all' 0) [0, 1, 2]
view (_all' 0) [0, 1, 2] :: (Data.Monoid.Monoid a, Num a, Eq a) => a
```

`Monoid`? Where does it come from? Remember, the original `_all` worked just fine:

```
> view (_all 0) [0, 1, 2]
0

> :t view (_all 0) [0, 1, 2]
view (_all 0) [0, 1, 2] :: (Num a, Eq a) => a
```

The mystery explained / exploiting `Monoid`

Take another look at the `Const` docs:

Instances:

- `Functor (Const m)`
- **`Monoid m => Applicative (Const m)`**
- `Foldable (Const m)`
- `Traversable (Const m)`
- `Generic1 (Const a)`
- `Generic (Const a b)`
- `Monoid a => Monoid (Const a b)`
- `type Rep1 (Const a)`
- `type Rep (Const a b)`

The original `_all` was satisfied with `Functor`; `_all'` uses `Applicative`, and it implies a constraint on the type of `Const`'s parameter. Since numbers generally aren't instances of `Monoid`, let's try something else. What about a list of lists?

```
> view (_all' [0]) [[0], [1], [2], [0]]
[0, 0]
```

```
> view (_all' [0]) [[1], [2]]
[]
```

A-ha, so *this* is how we can find out what values `_all'` is operating on. To make this more convenient, let's define 2 helper functions – `toListOf` (to get all values) and `preview` (to get just the first value). Additionally, I'd like to be able to check whether there is *any* value to operate on, so I also want to write `has`.

`toListOf` = all values

To write `toListOf`, look at `view` again:

```
-- unexpanded type:
--
--           Getting s a           -> s -> a
view :: ((a -> Const a a) -> s -> Const a s) -> s -> a
view l = getConst . l Const
```

When `view` is applied to `_all'`, the `Monoid` instance for `Const a` comes into play – but `a` *s* in general can't be combined meaningfully, and so it fails. (We can't define a generic `Monoid` instance for any `a` – it's impossible to write a generic `mempty :: Monoid a => a`, because it would have to pull an `a` out of thin air.) We can, however, trivially wrap `a` into a monoid (`[]` in this case):

```
toListOf :: ((a -> Const [a] a) -> s -> Const [a] s) -> s -> [a]
toListOf l = getConst . l (\x -> Const [x])
```

Then `[x]` *s* would be combined by the `Applicative` instance of `Const [a]`, and the result would contain all matching values in the list:

```
> toListOf (_all' 0) [0, 3, 1, 0]
[0,0]

> toListOf (_all' 0) []
[]
```

`preview` = first value

`preview` is based on the same idea – using an appropriate monoid to get the answer we

want- out of combined `Const` s. This time, however, we're only interested in the *first* value, not all of them. We could write our own monoid specifically for keeping the first value...

```
-- "Maybe a" because what if there aren't *any* values?
data First a = First (Maybe a)

instance Monoid (First a) where
  mempty = First Nothing

  mappend (First Nothing) y = y
  mappend      x          _ = x
```

...but it turns out it has been written already as `First` in the `Data.Monoid` module. Terrific.

```
preview
  :: ((a -> Const (First a) a) -> s -> Const (First a) s)
  -> s
  -> Maybe a
preview 1 = getFirst . getConst . 1 (\x -> Const (First (Just x)))
```

Just in case you want to see it in action:

```
> preview (_all' 0) [3, 2, 1, 0]
Just 0

> preview (_all' 0) []
Nothing
```

`has` = check for value

Now, `has` shall return a mere `Bool`. Yep, we could act on the output of `preview` (just check whether it's `Nothing` or `Just _`), but why do it when there's a monoid already available? In particular, I'm talking about the `Any` monoid, which is a simple wrapper over `Bool`:

```
newtype Any = Any Bool
```

With `Any` at hand, we can write `has` using the familiar pattern:

```
has :: ((a -> Const Any a) -> s -> Const Any s) -> s -> Bool
has l = getAny . getConst . l (\_ -> Const (Any True))
```

```
> has (_all' 0) [3, 2, 1, 0]
True

> has (_all' 0) [3, 2, 1]
False
```

No clumsy types!

Don't you think the types of functions we've written are kinda overly long? They're the same as the original `Getting s a`

```
type Getting s a = (a -> Const a a) -> s -> Const a s
```

with various monoids in place of `a` in `Const a`. Why not simply add an additional parameter to `Getting`?

```
type Getting r s a = (a -> Const r a) -> s -> Const r s
```

A-ha, this way it's much better (and also more similar to the actual types from lens):

```
view      :: Getting a      s a -> s -> a
toListOf  :: Getting [a]    s a -> s -> [a]
preview   :: Getting (First a) s a -> s -> Maybe a
has       :: Getting Any    s a -> s -> Bool
```

To make it easier to remember, here's what `Getting` approximately means in English:

`Getting r s a` is a function which, given some way to get `r` from `a`, will go over `a`'s in some `s` and return their combined `r`'s.

Or you could keep in mind that `Const r a` is the same as `r`, and mentally simplify the type to

```
type Getting r s a = (a -> r) -> s -> r
```

(and then `Setting s t a b` is simply `(a -> b) -> s -> t`.)

Why all this when `toListOf` is enough?

There's no single definite reason for not implementing everything in terms of `toListOf` once we have it, but...

- It would be “pretty odd” (as I got told on #haskell-lens).
- It would probably be slower.
- Aw c'mon, they're all one-liners anyway.

`toListOf` is subtly broken

(I mean, if you haven't noticed it already. Otherwise it's “obviously broken”).

Look at the definition of `toListOf` again:

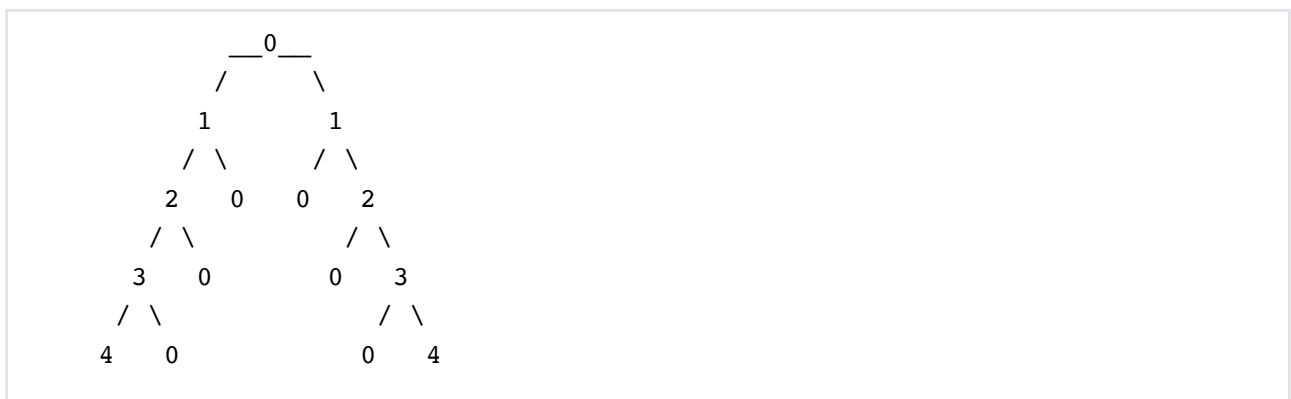
```
toListOf l = getConst . l (\x -> Const [x])
```

Notice that we always create a list consisting of 1 element. That's a million billion lists in case of a structure consisting of a million billion elements. All those lists would have to be *concatenated*. Are you starting to realise?

In case of the “structure” being a list, it doesn't matter – `traverse` would do the appends in the right order

```
[1] ++ ([2] ++ ([3] ++ ([4] ++ ...
```

and the whole operation would be cheap. However, it does matter when we're working with trees. Imagine a tree with 2 branches, one completely left-biased, the other – right-biased:



```

      / \
     ....

```

Let's say we cut this tree at depth 4 and do a traversal. Here are the appends that would happen (excluding appends to empty lists):

- `[4] <> [0]`
- `[3] <> [4,0]`
- `[3,4,0] <> [0]`
- `[2] <> [3,4,0,0]`
- `[2,3,4,0,0] <> [0]`
- `[1] <> [2,3,4,0,0,0]`
- `[0] <> [4]`
- `[3] <> [0,4]`
- `[0] <> [3,0,4]`
- `[2] <> [0,3,0,4]`
- `[0] <> [2,0,3,0,4]`
- `[1] <> [0,2,0,3,0,4]`
- `[1,2,3,4,0,0,0] <> [1,0,2,0,3,0,4]`
- `[0] <> [1,2,3,4,0,0,0,1,0,2,0,3,0,4]`

Generally, for such a tree of depth N (and containing $O(N)$ elements), there are N “bold” appends – the i -th append takes $O(i)$ time, and the total runtime is $O(N^2)$ for $O(N)$ elements. Ba-a-ad.

Difference lists

There's a well-known technique for making each append $O(1)$ when you have to do a lot of them, and it's called *difference lists*. I find existing explanations (e.g. in [Real World Haskell](#) and on [StackOverflow](#)) slightly confusing, so I'll explain it myself.

When we have a complex sequence of appends

```
((("a" ++ "b") ++ "c") ++ (("d" ++ "e") ++ "f") ++ ("g" ++ "h")) ++ ...
```

what we'd like is to somehow get a list of the “stuff being appended” and just do `concat` on it. The first idea is to append not lists, but *lists* of lists

```

type ConcatList a = [[a]]

(++) :: ConcatList a -> ConcatList a -> ConcatList a
(++) a b = a ++ b

```

and concat them all in the end:

```

toList :: ConcatList a -> [a]
toList = concat

```

But this approach, of course, won't work, as it doesn't make any difference when all “elementary” lists are a single element each. What we'd *really* like is to just say “these 2 lists have to go one after the other” and store this information without actually appending them yet... wait, we can do that with Haskell's data types!

```

data AppendList a = Append [a] [a]

```

Pfft, just kidding, this won't work. First of all, we want to append not simple, ordinary lists, but our `AppendList` S:

```

data AppendList a = Append (AppendList a) (AppendList a)

```

Then, where would *actual values* be stored? Let's add another constructor specifically for them:

```

data AppendList a = JustList [a] | Append (AppendList a) (AppendList a)

```

We don't need a special function for appending, because it's simply `Append`. But getting a list out of `AppendList` is slightly tricky:

```

doAppends :: AppendList a -> [a]
doAppends (JustList x) = x
doAppends (Append x y) = ...

```

Recursively converting `x` and `y` to lists and appending them with `++` wouldn't work—well, it would, but it wouldn't give any speed benefit. We need to *reorder*

`(a ++ b) ++ y ++ ...` into `a ++ (b ++ y ++ ...)`, remember? So, let's match on `x` as well:

```
doAppends (Append (JustList x) y) = x ++ doAppends y
doAppends (Append (Append a b) y) = doAppends (Append a (Append b y))
```

I won't show the benchmarks here – you can do some by yourself if you want (in a nutshell: use `Data.Tree`, generate the “evil tree” recursively, make a `Monoid` instance for `AppendList`, and use `traverse` with `Const (AppendList a)` to flatten it; to measure time in GHCi, do `:set +s` (I always confused `s` and `t` until I learned that `s` stands for “statistics”). But I'll tell you the results: an `AppendList`-based traversal is linear in time, while `[]`-based one is quadratic; the former flattens a tree of depth 2000k in 20s, while the latter only handles the depth of 11k in this time. [Just in case, “k” means “thousand” and not anything else it could mean.]

Now we can rewrite `toListOf` using `AppendList`:

```
instance Monoid (AppendList a) where
  mempty = JustList []
  mappend = Append

toListOf :: Getting (AppendList a) s a -> s -> [a]
toListOf l = doAppends . getConst . l (\x -> Const (JustList [x]))
```

Actual difference lists

...This is getting slightly offtopic, so I'll try to be more concise now.

A cool thing is that the compiler already does this reordering-stuff – when it simplifies expressions. Let's say we're evaluating `(x ++ y) ++ z`:

- The second `++` says: “let me look at the first element of `x ++ y` – if it exists, I'll emit it”.
- The first `++` says: “let me look at the first element of `x`”.
- Then it passes the element to the outer `++`.
- Then the outer `++` emits it and proceeds with the second element.

After all this, *each* element of `x` is “looked at” twice.

Now consider this alternative way to construct `(x ++ y) ++ z` using functions –

`((x ++). (y ++)). (z ++) $ []` (I just replaced each list with a function that appends this list to the argument). When evaluating *this* expression, the compiler will have to expand some `.` – and after this expansion the “application tree” would be practically flat! To understand better, you really should watch the succession of steps done by [stepeval](#), and then look at [these pictures](#). I've replicated the `stepeval` output here for your convenience, with bogus steps (e.g. `[a]` being “evaluated” into `a : []`) omitted. First, the normal version:

- `([a, b] ++ [c, d]) ++ [e, f]`
- `(a : [b] ++ [c, d]) ++ [e, f]` ← Here `a` has gone thru the first `++` ...
- `a : ([b] ++ [c, d]) ++ [e, f]` ← ...and only here – thru the second.
- `a : (b : [] ++ [c, d]) ++ [e, f]` ← Same with `b` –
- `a : b : ([] ++ [c, d]) ++ [e, f]` ← there are as many steps as `++` s.
- `a : b : [c, d] ++ [e, f]`
- `a : b : c : [d] ++ [e, f]` ← Only 1 step for `c`.
- `a : b : c : d : [] ++ [e, f]` ← Same with `d`.
- `a : b : c : d : e : f : []`

And now, the “difference list” version, which gets transformed into plain

`[a, b] ++ [c, d] ++ [e, f]` (if you strip the extraneous parens) after only 3 steps:

- `((([a, b] ++). ([c, d] ++)). ([e, f] ++)) $ []`
- `((([a, b] ++). ([c, d] ++)). ([e, f] ++)) []`
- `((([a, b] ++). ([c, d] ++)) ([e, f] ++) [])` ← Removed one `.`
- `([a, b] ++) ([c, d] ++) ([e, f] ++) []` ← Removed another `.`
- `a : [b] ++ ([c, d] ++) ([e, f] ++) []` ← Okay, here comes `a` already.
- `a : b : [] ++ ([c, d] ++) ([e, f] ++) []`
- `a : b : ([c, d] ++) ([e, f] ++) []`
- `a : b : c : [d] ++ ([e, f] ++) []`
- `a : b : c : d : [] ++ ([e, f] ++) []`
- `a : b : c : d : ([e, f] ++) []`
- `a : b : c : d : e : [f] ++ []`
- `a : b : c : d : e : f : [] ++ []`
- `a : b : c : d : e : f : []`

The important thing is that with difference lists we have to do only as many preliminary steps as there are lists to append; after that each element is generated in constant time. With ordinary lists, each element takes the time proportional to its depth in the tree – and this leads to quadratic behavior.

In case you're wondering, differential lists from the `Data.DList` module are twice as fast as our handwritten `AppendList`.

Introducing... monoids. Of endomorphisms! Under composition!

Turns out, however, that we don't even need the full power of difference lists.

`Data.Monoid` includes a very simple version that's enough for our purpose – `Endo`, which is, as the documentation helpfully suggests, “the monoid of endomorphisms under composition”. (As a side note, when I'm the king of Earth and its surroundings, I shall assemble a team of technical writers and bring all documentation in standard libraries to the level of `pipes`.)

Translated into ~~humanese English~~ language that -people who haven't studied category theory and don't really like googling- can understand, `Endo` is a `newtype` wrapper that lets us use the fact that functions of type `a -> a` form a monoid. (One reason why a wrapper is needed is that functions of the type `Monoid b => a -> b`, too, form a monoid – the same type can't have 2 instances of the same typeclass, and this instance is more important than the `Endo` one.) What monoid? Well, `id` is the identity element, and `.` is the binary operation, nothing interesting.

Have you guessed already how it can be used? With difference lists, each “list” looks like `(x ++)` – which has the type `[a] -> [a]`. It's an endomorphism! And we use `.` to append lists... and `.` is also function composition! Wow!

Um, I'm sorry for using 3 exclamation marks in a paragraph. I just thought it would make the text a bit less dull. (At night, many things seem duller than they are, and it's 2.30am here at the moment of writing.)

[Oh, and I'm also sorry for being meta in the last paragraph. I've noticed that some people are confounded by anything meta (especially in a conversation, for some reason); if you're one of those people, please accept my ~~condolences~~ apologies.]

Anyway, the final version of `toListOf` looks like this:


```
toListOf :: Getting (Endo [a]) s a -> s -> [a]
toListOf l = (`appEndo` []) . getConst . l (\x -> Const (Endo (x:)))
```

(`x:` looks better than `[x] ++`, don't you think? On the other hand, `(`appEndo` [])` looks worse than `toList` or `doAppends`. Whatever.)

Well, it's not *actually* final – in the next part of this series I'll have explained enough to match the true implementation in lens,

```
toListOf :: Getting (Endo [a]) s a -> s -> [a]
toListOf l = foldrOf l (:) []
```

but at least the type signature is “final”, and we'll have to be satisfied with that for the time being.

P.S. Did you know that originally lens used `[a]`, and it was only [changed to](#) `Endo [a]` in version 3.4? It helped me realise that lens isn't a shiny piece of perfect abstractions where everything follows absolutely rigorously from everything else and there's only one way to write things. Lens is alive. Lens is evolving. Some even [worry](#) it will become a “[new language] we all will have to learn [...] sooner or later” – and when it happens, we shall be prepared. Right?

[Soon on /r/haskell: “Lens, Pipes and Yesod Break Free, Fight Amid the Ruins of Haskell! SPJ Disappointed”.]

Get back to traversals, will you?

Ouch, sorry. Our `AppLens` is actually called `Traversal` in lens library! Here's the All-New Rebranded Definition:

```
type Traversal s t a b = Applicative f => (a -> f b) -> s -> f t
type Traversal' s a = Traversal s s a a
```

To celebrate this amazing revelation, here are some new `Traversal`s for you:

`each` = every element

`each` focuses on every element in a monomorphic container:

```
> set each 8 [1..3]
[8,8,8]

> over each toUpper (Data.Text.pack "blah")
"BLAH"

> each (\x -> print x >> return (-x)) (8, 19)
8
19
(-8,-19)
```

Its implementation is pretty simple. To support different types of containers, we create a typeclass containing `each` as its only method:

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

class Each s t a b | s -> a, t -> b, s b -> t, t a -> s where
  each :: Traversal s t a b
```

(The `s -> a, ...` bit is a [functional dependency](#) – without it we are going to get a bunch of “ambiguous type” errors whenever we try to actually use `each`).)

Now the best part: `each` is actually the *same* as `traverse` !

```
instance T.Traversable t => Each (t a) (t b) a b where
  each = T.traverse
```

Well, at least for `Traversable` types – that's `[]`, `Map`, `Maybe`, and so on... Hm. Tuples are `Traversable` as well, right?

```
> T.traverse (\_ -> putStr "? new: " >> readLn :: IO Bool) (1, 2)
? new: True
(1,True)
```

Darn. So our `each` isn't the same as lens's `each` after all:

```
> set each 8 (1, 2)
(1,8)
```

```
> set Control.Lens.each 8 (1, 2)
(8,8)
```

Of course, now I could just explain how to write `each` correctly (even tho it's pretty obvious and doesn't really need any explanation), but instead I'll dissect the actual [implementation of `Each`](#) in the lens library. “What a waste of time”, you might be thinking... Sorry.

Dissecting `Each`

The reason for our `each` being different from lens's `each` is that the `Traversable` instance for tuples isn't quite what we want – it's more general (`Traversable ((,) a)`), but the price to pay is that `traverse` can only work with the second element. Just in case, here's the definition:

```
instance Traversable ((,) a) where
  traverse f (x, y) = (x, ) <$> f y
```

(One reason – don't know whether there are more – for -not having the seemingly more useful instance for `(a, a)` in the base- is that you can't actually write it without resorting to newtypes, and this is because Haskell doesn't have [type-level lambdas](#).)

So, `Traversable` doesn't work as we want. No big deal, yeah – the only downside is that instead of a single instance like `Traversable t => Each ...`, there is:

- A kinda scary class declaration:

```
class Each s t a b | s -> a, t -> b, s b -> t, t a -> s where
  each :: Traversal s t a b
#ifdef HLINT
  default each :: (Applicative f, Traversable g, s ~ g a, t ~ g b)
    => LensLike f s t a b
  each = traverse
    {-# INLINE each #-}
#endif
```

- Several “default” instances without implementation:

```
instance Each [a] [b] a b
instance Each (NonEmpty a) (NonEmpty b) a b
instance Each (Identity a) (Identity b) a b
instance Each (Maybe a) (Maybe b) a b
instance Each (Seq a) (Seq b) a b
instance Each (Tree a) (Tree b) a b
```

- Other instances, such as the one for 2-tuples (there are similar ones up to 9-tuples there):

```
instance (a~a', b~b') => Each (a,a') (b,b') a b where
  each f ~(a,b) = (,) <$> f a <*> f b
  {-# INLINE each #-}
```

#ifndef

Let's start with the class declaration. The `#ifndef` section would be removed by the C preprocessor if the code is being analysed by [HLint](#) at the moment; to see why, it's enough to simply run HLint on this piece of code:

```
/tmp/lens-each:3:3: Warning: Parse error: default
Found:
  class Each s t a b | s -> a, t -> b, s b -> t, t a -> s where
    each :: Traversal s t a b
  >   default each :: (Applicative f, Traversable g, s ~ g a, t ~ g b)
        => LensLike f s t a b
    each = traverse

1 suggestion
```

Um, just so that it wouldn't be left unexplained... In a nutshell:

- You can enable `{-# LANGUAGE CPP #-}` and get your source processed by the [C preprocessor](#).
- It's mainly used to do things differently based on the version of the compiler, available libraries, OS, and so on.
- I won't describe how to use it yet (but I will when we stumble upon code which uses it more extensively). For now, you just need to know that `#ifndef` means “if not defined”, `HLINT` is the variable which would be in “defined” state when the code is

processed by HLint.

- Finally, it's “**C** preprocessor”, not “**C** plus plus preprocessor”.

`default`

So we've learned that HLint doesn't like this code because of `default`. Some googling shows us that `default` in type classes is enabled by the `DefaultSignatures` extension, which HLint apparently isn't yet aware of. If you aren't aware of it either – again, here's an “in a nutshell” explanation [hm, or should I call it “bullet list explanation”?]:

- You can give -methods of typeclasses- default implementations. It allows instance writers to use the -implementation they would've used anyway- without actually bothering to write it, and at the same time they can use a better implementation when it's available. For instance, the default implementation for `/=` is

`x /= y = not (x == y)` (makes sense, doesn't it), but the definition GHC uses for e.g. `Char` is

```
(C# c1) /= (C# c2) = isTrue# (c1 `neChar#` c2)
```

- Without this extension, the requirement for the default implementation is that it has the same type that the method has. You can't do something like this:

```
class Improve a where
  improve :: a -> a
  improve False = True
```

even if you're dead sure that `True` is better than `False`, because your default method has to work for all `a`s.

- With `DefaultSignatures` enabled, however, you can choose one type (which can include constraints, etc.) for which you can provide the default method. For instance, lists of numbers:

```
class Improve a where
  improve :: a -> a
  default improve :: Num a => [a] -> [a]
  improve = map (*2)
```

- The only trouble you might encounter is that when your type is so specific that it doesn't even mention the type variables of the class, GHC will complain. This isn't allowed:

```
class Improve a where
  improve :: a -> a
  default improve :: Bool -> Bool
  improve _ = True
```

However, *this* is:

```
class Improve a where
  improve :: a -> a
  default improve :: (a ~ Bool) => a -> a
  improve _ = True
```

(`~` is a type equality constraint. If `Num a` reads as “`a` is a number”, then `a ~ Bool` reads as “`a` is the same as `Bool`”.)

Now you probably understand why there's a `default` signature in the class definition – because the default method only works for `Traversable` types. You also understand why it's

```
default each :: (Applicative f, Traversable g, s ~ g a, t ~ g b)
              => LensLike f s t a b
```

and not

```
default each :: (Applicative f, Traversable g)
              => LensLike f (g a) (g b) a b
```

– because the default signature has to mention all the type variables (`s`, `t`, `a` and `b` in our case).

LensLike

This is simply a convenient synonym:

```
type LensLike f s t a b = (a -> f b) -> s -> f t
```

The interesting thing is that it doesn't seem to be needed here. This declaration:

```
default each :: (Applicative f, Traversable g, s ~ g a, t ~ g b)
              => LensLike f s t a b
```

is supposed to be the same as:

```
default each :: (Traversable g, s ~ g a, t ~ g b)
              => Traversal s t a b
```

Why isn't the latter used instead of the former?

Instead of spending a day thinking about it and testing various theories, I asked Edward Kmett on `#haskell-lens` (well, I asked *anybody*, but it was he who happened to answer):

```
(06:57:55 AM) edwardk: because default signatures suck at dealing with
rank-2 types
(06:58:00 AM) edwardk: they just don't unify
(06:58:04 AM) edwardk: try it, it won't compile

...

(07:02:25 AM) edwardk: it may also be only on ghc < 7.6 or so
(07:02:41 AM) edwardk: i'd be happy to be wrong on this

...

(07:03:37 AM) edwardk: if that signature compiles on GHC 7.4 i'd take a
patch to switch
```

Ah, so there wasn't any reason – the signature indeed compiled and the patch got accepted. Huzzah.

INLINE

`{-# INLINE each #-}` means that the definition of `each` would be (roughly speaking) inserted directly into source code when compiling, instead of being called – this makes sense since in most cases it's simply a call to another function. Of course, I still wonder how much of a benefit it is, but benchmarking lens is going to be the topic of a separate article.

`a~a'`

Here's the code for the tuple instance again:

```
instance (a~a', b~b') => Each (a,a') (b,b') a b where
  each f ~(a,b) = (,) <$> f a <*> f b
```

The instance itself is easy to understand. However... There are 2 weird things here, and both involve a tilde (but these are *very* different tildes).

We already know that `~` means type equality. But why can't we just say

```
instance Each (a, a) (b, b) a b where
```

instead? What's the difference? Well, The difference is somewhat subtle, and it involves explaining how GHC works with instances.

- Each instance has a *context*. E.g.

```
instance Show a => Show [a]
```

has `Show a` as its context. Or, as described in Haskell report,

The part before the `=>` is the *context*, while the part after the `=>` is the *head* of the instance declaration.

- When you're doing something like `show x`, GHC looks at the type of `x` and tries to find the corresponding instance.
- However, when searching GHC does *not* look at the contexts – only at the heads. For example, the following code wouldn't even compile:

```
class Foo a where
  foo :: a -> Int

instance Num a => Foo a where
  foo x = 1

instance Show a => Foo a where
  foo x = 2
```


because `instance Num a => Foo a` and `instance Show a => Foo a` are “the same”, as they only differ in their contexts.

- If no instances overlap, GHC picks the one which matches and then checks whether the context is right. If the context isn't “right”, GHC will complain – but it won't look for other instances which could match.

Now, if we write

```
instance Each (a, a) (b, b) a b where
  each f ~(a,b) = (,) <$> f a <*> f b
```

and accidentally try to use `each` on a tuple with different types, what would GHC complain about? Right, that it can't find an instance:

```
> each Just (False, ())

<interactive>:
  No instance for (Each (Bool, ()) t0 b0 b0)
    arising from a use of 'it'
```

GHC doesn't know that there isn't an instance like `Each (Bool, ()) ...` defined anywhere and won't *ever* be defined; thus, its complaint is entirely reasonable. However, GHC also (usually) does better with more information rather than less, and we know that if `each` is used on a tuple, its elements *have* to be equal.

To give GHC the same knowledge, first we show it an instance for *arbitrary* tuples (remember that GHC is context-blind at the moment):

```
instance (a ~ a', b ~ b') => Each (a, a') (b, b') a b
```

GHC sees it and already commits to using it. Now it's time to look at the context, and *there* we tell it that those tuples' elements had to be of equal types. From now on GHC has no choice but to accept it – it can't backtrack and look for another instance.

For comparison, here's the error message GHC would actually give on

```
each Just (False, ()) :
```

```
> each Just (False, ())
```

```
<interactive>:
  Couldn't match type 'Bool' with '()'
  In the expression: each Just (False, ())
  In an equation for 'it': it = each Just (False, ())
```

Much better, isn't it?

`~(a,b)`

Here's the line again:

```
each f ~(a,b) = (,) <$> f a <*> f b
```

`~` here means a *lazy pattern*. Bullet list explanation:

- Haskell is lazy. If you don't touch `undefined`, it won't bite:

```
> (\x -> True) undefined
True
```

- Sometimes, unfortunately, we have to evaluate things a bit. For instance, when determining whether a list is empty of not:

```
null [] = True
null _  = False
```

Here `null` would have to look at the list *constructor* to make the decision.

Therefore, `null undefined` is `undefined`, while e.g.

`null (undefined:undefined)` is `False`.

- Sometimes looking at the constructor isn't really necessary. For instance, when the answer doesn't depend on it:

```
sillyVoid :: [a] -> ()
sillyVoid []      = ()
sillyVoid (x:xs) = ()
```

Is `sillyVoid` equal to `const ()`? No. Here's where it fails:

```
> const () undefined
```

```
( )

> sillyVoid undefined
*** Exception: Prelude.undefined
```

- A less silly example is `each` (as well as `***`, `first`, `second`, `bimap` and other tuple-related functions). Let's say we set the first element of a tuple to `False`, and the second to `True`. Do you expect the result to always be `(False, True)`?

```
> first (const False) $ second (const True) $ undefined :: (Bool, Bool)
(False, True)
```

Nice. What if we use our own function to do it?

```
> let setFT (x, y) = (False, True)

> setFT (undefined :: (Bool, Bool))
*** Exception: Prelude.undefined
```

Expected, but still isn't nice.

- `~` defers looking at the constructor until it's really needed. These pieces of code are equivalent:

```
f ~(x, y) = ... x ... y
```

and

```
f pair = ... (fst pair) ... (snd pair)
```

- So, if we want `over each (const ())` to always be the same as `const ((), ())`, we need that `~` there.
- These are also called *irrefutable* patterns, by the way. You can read more in the [Haskell Wikibook](#)
- Finally, keep in mind that not all tuple functions work like this. For instance, the `Monoid` instance for tuples doesn't – `(((), ()) <> undefined` should be `(((), ()))` “in theory”, but in reality it's `undefined`. This [has to do something with space leaks](#).

Okay, enough about `Each`. 2 more simple traversals and time for another recap.

`_head`, `_last` = first and last elements

`_head` and `_last` give access to the first and last element of a container (a list, vector, etc.). For simplicity, I'll only define `_head` for lists:

```
_head :: Traversal' [a] a
_head f []      = pure []
_head f (x:xs) = (:) <$> f x <*> pure xs
```

You might be wondering: “defining `_head` for lists is easy, why wouldn't he show how to define it for `Text` instead?”. Indeed, initially I wanted to define it for `Text` in a more efficient way than unconsing-and-consing, and was starting to feel bad when I couldn't think of a way to do it – but then I realised that it's impossible. Imagine the functor in question being `[]` – how can you construct many `Text`s, each with a different first character, without actually duplicating the character array? (Yep, I know that `Text` can [consist of chunks](#) – let's assume we're talking about strings shorter than one chunk.)

Recap

- A lens focuses on one value; a traversal – on many (may be 0 as well).
- Traversals are lenses with `Applicative` instead of `Functor`:

```
type Traversal s t a b = Applicative f => (a -> f b) -> s -> f t
```

- In order to let `view`, `over`, `set`, etc. work with both lenses and traversals, their types are specialised to `Getting` and `Setting`:

```
-- Mentally simplified to "(a -> r) -> s -> r".
type Getting r s a = (a -> Const r a) -> s -> Const r s

-- Mentally simplified to "(a -> b) -> s -> r".
type Setting s t a b = (a -> Identity b) -> s -> Identity t

view :: Getting a s a -> s -> a
```

```
over :: Setting s t a b -> (a -> b) -> s -> t
set  :: Setting s t a b -> b -> s -> t
```

- The `Applicative` instance for `Const` uses monoids to combine results, which means that by simply varying the monoid you can
 - get all traversed values (monoid: `Endo []`, function: `toListOf`)
 - get the first value (monoid: `First`, function: `preview`)
 - check for value (monoid: `Any`, function: `has`)
 - find the sum of traversed values (monoid: `Sum`)
 - find the product of traversed values (monoid: `Product`)
 - and so on (I haven't actually mentioned the last 2 in text, but hearing “`Sum`” and “`Product`” should be enough to figure it out)
- There are various predefined traversals, such as `each`, `_head` and `_last`. (Another few easy ones are `both`, `_tail` and `_init`. Other ones use scary indexed stuff and `Bazaar`s and who-knows-what-else, so we won't touch them yet.)

Bonus stuff:

- Difference lists are easy to implement and give you $O(1)$ appends (but $O(N)$ `head`).
- With default signatures there are more situations when you can add a default implementation to a method.
- `#haskell-lens` has all the answers.
- Type equality constraints (`~`) give you better error messages (and can be generally useful).
- GHC doesn't take constraints into consideration when searching for a matching instance.
- Irrefutable (“lazy”) patterns let you do lazy matching on values; however, it's rarely useful and you can always replace them by getters (e.g. `~(x, y)` by `fst` and `snd`).

Appendix: operators

There are many operators in lens – 124 in `Control.Lens` at the moment of writing (there are some other operators which aren't exported by this module). You don't have to use them all, but since people generally have different ideas about which operators they don't and do have to use... Anyway, here's the first bunch.

- `view` is `^.` (with arguments flipped):

```
> (1, 2) ^. _1
1
```

- `over` is `%~`:

```
> (_1 %~ negate) (1, 2)
(-1, 2)
```

- `&` is backwards function application:

```
> :t _1 %~ negate
_1 %~ negate :: Num a => (a, x) -> (a, x)

> (1, 2) & _1 %~ negate
(-1, 2)
```

It lets you create nice-looking chains:

```
blah & somePart %~ f
    & otherPart %~ g
    & ...
```

- `set` is `.~`:

```
> (1, 2) & _1 .~ 88
(88, 2)
```

- `<%~` is `over` which also returns the new value:

```
> [0..10] & ix 10 <%~ negate
(-10, [0,1,2,3,4,5,6,7,8,9,-10])
```

As there's no single “new” value in case of a traversal, you can't use `<%~` on traversals (well, or you can but you'll stumble upon the `Monoid` problem).

- `<<%~` is like `<%~`, but it returns the *old* value:

```
> [0..10] & ix 10 <<%~ negate
(10, [0,1,2,3,4,5,6,7,8,9,-10])
```

- Instead of `%`, you can put in `+`, `-`, `*`, `//`, `^`, `||`, `<>` and so on to get specialised versions:

```
> -- Multiply by 2.
> [0..10] & each *~ 2 -- Multiply by 2.
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

> -- Raise to the 3rd power and return the old value.
> [0..10] & ix 10 <<^~ 3
(10, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1000])
```

This already gives you ~30 operators.

(...Please don't use most of them. Honestly, stuff like `<<***~ x` is scary, and if it's not scary for you, it'll most likely be scary for people who would have to read your code.) Could you perhaps consider using e.g. `<<%~ (** x)` instead? Thanks.)

- `toListOf` is `^..`:

```
> (1, 2) ^.. each
[1, 2] -- `each` looks at all elements.

> [] ^.. _head
[] -- `_head` looks at the first element, if it exists

> [0..10] ^.. _head
[0] -- `[0..10]` has only 1 first element (surprise!).
```

It's simple to remember: `^.` (“viewing”) + `..` (“something list-y”).

- `preview` is `^?`:

```
> [0..10] ^? _head
Just 0
```

```
> [0..10] ^? each
Just 0
```

If you're implementing everything we've covered in a separate module (i.e. creating your own mini lens library), don't forget to add fixity declarations for these operators. You can take them from the source of lens, or find out using GHCi:

```
> import qualified Control.Lens as L

> :i L.&
(L.&) :: a -> (a -> b) -> b      -- Defined in 'Control.Lens.Lens'
infixl 1 L.&

> :i L.^?
(L.^?) :: s -> L.Getting (First a) s a -> Maybe a
      -- Defined in 'Control.Lens.Fold'
infixl 8 L.^?
```

A fixity declaration just describes

- the priority of the operator, from 0 to 9; 0 means “is applied last” (`$` has this priority) and 9 means “is applied first” (that's `.`, for instance)
- the fixity of the operator, which can be `infixr`, `infixl`, or `infix`; if several operators have equal priority and they are all `infixr` or `infixl`, the rightmost/leftmost operator would be applied first; if there's an `infix` operator in the mix, it's an error

To add a fixity declaration, just write it before or after the operator:

```
(!!!) = undefined

infixl 4 !!!
```

By the way, you can give fixity declarations to functions as well (which will affect their usage in backticks) – for instance, `elem` and `mod` are `infix 4` and `infixl 7` respectively.

To be continued

The next part will be about getters, setters and folds. I'll also describe `Lens` / `Traversal` laws, why our `ix` isn't a lens and `_all1'` isn't a traversal [I wonder, were those of you -who know about lens laws- already planning to write caustic comments on this topic?], and how lenses compose.

Oh, and the answer to the bonus points exercise is this:

```
(<%~) :: Lens s t a b -> (a -> b) -> s -> (b, t)
(<%~) l f s = l ((,) <$> f <*> f) s
```

And here's an even better-looking solution if you are willing to use `&&&`:

```
(<%~) l f = l (f &&& f)
```

And here's another one that I got by email after the post was published:

```
(<%~) l f = l (join (,) . f)
```

It's also faster because it computes the value-you-are-setting only once. All previous variants were doing this:

```
(<%~) l f = l (\x -> (f x, f x))
```

But the right thing to do is one of these:

```
(<%~) l f = l (\x -> let fx = f x in (fx, fx))
```

```
(<%~) l f = l $ (\t -> (t, t)) . f
```

Okay, and while I'm at it, here's a nice one for `<<%~`:

```
(<<%~) :: Lens s t a b -> (a -> b) -> s -> (a, t)
(<<%~) l f = l ((,) <*> f)

-- equivalent to ((,) <$> id <*> f)
-- equivalent to (\x -> (x, f x))
```

The answers to all other exercises are [here](#).

P.S.

[lens](#) is a huge package, so I took a chunk out of it and put into a separate library – [microlens](#). It has no dependencies, it compiles in about 4 seconds, and in many cases it's a good replacement for lens (i.e. if you want to use the power of lenses in your library and don't like heavy dependencies).

Don't use microlens for this series, tho; many of things I'm covering here are lens-specific (isomorphisms, prisms, indexed traversals, etc).

<<< “lens over tea” >>>

Read next: [Telegram channel](#)

[Get a Haskell mentor](#)
(or help us teach other people if you don't need a mentor)

9 Comments Artyom's site

 Login ▾ Recommend Share

Sort by Best ▾



Join the discussion...

**Constantine Kharlamov** • 9 months ago

Okay, I am again have no idea what the next training function is supposed to do, and hence how am I supposed to implement it.

```
choosing :: Lens s1 t1 a b -> Lens s2 t2 a b
          -> Lens (Either s1 s2) (Either t1 t2) a b
choosing l1 l2 = _
```

^ | ▾ • Reply • Share ›

**Artyom** Mod → Constantine Kharlamov • 9 months ago

If you have a lens that changes an `a` in an `s1`, and another lens that changes an `a` in an `s2`, then `choosing` makes a lens that can change an `a` in either `s1` or `s2`.

^ | ▾ • Reply • Share ›

**Constantine Kharlamov** → Artyom • 9 months ago

Please tell me, what do I do wrong, I've no slightest idea... The choosing type

```
Lens s1 t1 a b -> Lens s2 t2 a b
          -> Lens (Either s1 s2) (Either t1 t2) a b
```

upon simplification becomes

```
(a -> f b) -> s1 -> f t1    -- Lens s1 t1 a b
-> (a -> f b) -> s2 -> f t2  -- Lens s2 t2 a b
-> (a -> f b) -> (Either s1 s2) -> f (Either t1 t2)
```

And the implementation is straightforward:

```
choosing leftFunc leftVal _idk1 rightFunc rightVal _idk2 _idk3 ei
  case eitherVal of
    Left e -> (Left) <$> leftFunc e
    Right e -> (Right) <$> rightFunc e
```

But the compiler asks me where did I learn to calculate:

[see more](#)

^ | ▾ • Reply • Share ›

[view source](#)

[how it's made](#)

[hire me](#)

[stats](#)
[cat](#)

[comment by email](#)

[curiosity killed the](#)