Lenses for the Mere Mortal

PureScript Edition

Brian Marick

Lenses for the Mere Mortal: PureScript Edition

Brian Marick

This book is for sale at http://leanpub.com/lenses

This version was published on 2018-08-20

Contents

Introduction

Lens libraries, such as Haskell's `Control.Lens` or PureScript's `Data.Lens`, have the reputation of being hard to learn, or at least hard to learn well. That's not to say that lots of people haven't learned them. Many have. But many others have had a hard time, me included.

This book teaches lenses differently. I don't claim I've found the single best way to teach them, just that this way is worth a try, whether you're learning lenses for the first time or you've bounced off some other tutorial that didn't work for you.

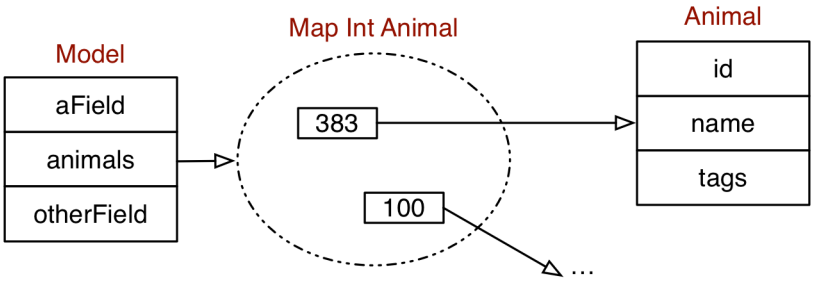
Since you might be new to lenses, I'll start by describing what they're good for. But first:

Pedanticism alert!

The previous paragraphs used the word "lens." From now on, I'll use the word "optics" (a common alternate term). That's because the type `Lens` is only one of several related types, so people end up switching awkwardly between "lens" to refer to a type and "lens" to refer to the general idea. I'll use "lens" for the type and "optic" for the idea.

Optics are useful

Optics are types that can be used to "focus in on" a particular element in a deeply nested data structure. As a simple example, consider this:[1]



If that's the heart of your program – as it is for one of mine – you'll probably have a lot of code that fetches or changes a particular `Animal` within the `Model`. In PureScript, an `Animal` can be fetched like this:

```
> Map.lookup 3838 model.animals
(Just { id: 3838, name: "Genesis", tags: ["mare"] })
```

If we wanted a function that fetched an animal at a particular `id`, it could look like this:[2]

```
viewAnimal :: Animal.Id -> Model -> Maybe Animal
viewAnimal id = _.animals >>> Map.lookup id

> viewAnimal 3838 model
(Just { id: 3838, name: "Genesis", tags: ["mare"] })
```

Now consider that `viewAnimal` can be thought of as having two responsibilities:

1. Knowing the *path* from a `Model` to a particular `Animal`.
2. Knowing how to retrieve a (possibly missing) `Animal`, given that path.

What if we split those two responsibilities up into two functions?

[1] This structure, with all of a browser app's state enclosed in a `Model`, is typical of the Elm Architecture and PureScript frameworks like Spork or Pux.

[2] I'm using point-free style for `viewAnimal` – omitting the `model` argument – to make this work-in-progress look more like actual optic definitions.

```
> view (_animal 3838) model
(Just { id: 3838, name: "Genesis", tags: ["mare"] })
```

`_animal` creates a path-knower (an "optic"), and `view` uses it.

That seems silly, especially since the implementation of `_animal` would be the same as `viewAnimal`:

```
_animal id = _.animals >>> Map.lookup id
```

... and `view` does nothing but step out of the way:

```
view optic whole = optic whole
```

But wait! You've probably heard the acronym `CRUD`, representing four universal database operations: `CREATE`, `READ`, `UPDATE`, and `DELETE`. `view` is our `READ`, but it's not the only thing to do with a particular animal. You might want to replace it. Doing that in straight PureScript is annoyingly different than viewing it:

```
model { animals = Map.insert 3838 newAnimal model.animals }
```

Wouldn't it be nicer if we could use the same (`_animal 3838`) value with a `set` function?

```
> set  (_animal 3838) newAnimal model
-- compare to:
> view (_animal 3838)           model
```

Actually, we could get cute here. Since `view` returns a `Maybe Animal`, we can make `set` conceptually parallel by having it take a `Maybe Animal`. Then setting a `Just` will insert or update:

```
> set (_animal 3838) (Just newAnimal) model
```

... and setting a `Nothing` will delete:

```
> set (_animal 3838) Nothing model
{ animals: (fromFoldable []) }
```

`set` won't work with our previous implementation of `_animal`. A value that works wtih both `view` and `set` requires a much more complicated type. But you don't have to fully understand that type to use its constructors and compose their results into longer paths. So, even though records are one of the more awkward types to create optics for, defining `_animal` is not much code:

```
-- Create a lens for a record field using `lens`.
-- Record lenses always follow the same template,
-- so you can cut, paste, and substitute.
_animals =
  lens _.animals $ _ { animals = _ }

-- Add an optic for the `Map`.
_animal id =
  _animals <<< at id
```

Once you've got that, you can not only `view` and `set` an animal, but also `UPDATE` it using `over`:

```
> over (_animal 3838) (map $ Animal.addTag "new tag") model
                             ^^^^^^^^^^^^^^^^^^^^^^^^

{ animals: (fromFoldable [(Tuple 3838
   { id: 3838, name: "Genesis", tags: ["mare","new tag"]
                                             ^^^^^^^^^

})]) }
```

If you imagine writing the same update in straight PureScript, I think you'll appreciate the work that optics can save you.

Why are optics notoriously confusing?

I have a lot of experience explaining things to people, and I believe optics aren't *inherently* confusing. Rather, they're traditionally taught in a way that doesn't work for an audience of people like me. (Which, fortunately for my career as an explainer, includes a lot of people.)

Here are the problems I see:

1. Optic explanations are written by people suffering from the curse of knowledge: they know too much to understand (or remember) what beginners don't know.
2. Programming is easier when you don't have to understand the implementation of something to be able to use it. Such sharp separation seems impossible for optics. Because of that, explaining them is a delicate dance of revealing *just enough* of the implementation, and at a pace that doesn't overwhelm the beginner.

   That's made harder because the implementation of optics is immensely clever and perhaps even profound. Naturally, those who understand it want to share its coolness – but that impulse doesn't serve the beginner well.

3. Haskell's seminal `Control.Lens` (and consequently, reimplementations of it) uses operators heavily. Although there are named functions behind those operators, the operators dominate beginner documentation. That means that the task of understanding optics is coupled to the need to memorize operators like `^?`, `^..`, `^.`, `||~`, `?=`, `<>~`, and friends.

   Many people (me!) are not good at such simultaneous memorization.

4. The documentation is too scattered. There's no clear through line[3] for beginners to follow, no path that doesn't threaten to suddenly dump you into depths you're not ready to handle.

This book's approach

This book is different.

1. I'm learning optics as I write, so I'm not cursed by non-beginner knowledge.
2. I came to optics because I got tired of writing special-case code to traverse structures containing `Map`s or sum types. However, such common types are underused in optics tutorials.

   For example, creating a `Prism` for a sum type requires the explanation of some tedious details. I suspect that's why so many `Prism` explanations use other types. But sum types are what `Prism` is *for*! Other types are just a bonus.

   Therefore, this book is (roughly) ordered by the types *on which you use optics* rather than the optic types themselves. For example, an optic for `Map` comes early in the book because working with `Map`s is probably a pressing need for people learning optics. That's true even though `Map` optics are un-profound variations of more fundamental optics. Un-profound – but really confusing to the novice, and so worth careful discussion.
3. When it comes to optics, sadly, the compiler isn't your friend – at least not as much as it is with other types. It's a consequence of the subtlety and generality of the implementation, but that's cold comfort when the compiler throws something like this at you:

   ```
   forall t1 t3. At t1 Int t3
     => (forall p. Strong p =>
     p (Maybe t3) (Maybe t3) -> p t1 t1)
   ```

   ... and you realize that `At` and `Strong` aren't types you know anything about or use in optic type annotations.

   For that reason, the book contains a number of examples of what you see from the compiler. For each, you're told some of what output means... but only enough to let you make practical progress toward understanding how to *use* optic types.

   (I do plan on adding a fairly comprehensive description of the implementation at the end of the book, but it will be optional. I'll count this book a failure if you need those chapters to use optics effectively.)

[3] "Though line" (or "throughline") is an idea originating in the theater. It connects all the events in a play so that what happens next seems an unsurprising (though perhaps still shocking) consequence of what the audience has seen before. A short description, from theater. Another short one, from novel writing. A longer description, from story writing.

4. This book will *show* what other sources expect you to *deduce*. For example, what's special about the `At` optic is captured by this type annotation:

   ```
   at :: a -> Lens' m (Maybe b)
   ```

   But it took me *forever* – and considerable frustration – to understand the implications. That's (mostly) because I was a novice. Forcing you to do the same would be a cruel and wasteful hazing ritual. So I spell implications out.

Why PureScript?

The ancestor of all the optics libraries of the kind described here is for Haskell. If I'd started out wanting to write the most broadly useful book, I'd have documented that one.

However, this book's origin story is… kind of random. I was a backend programmer (focused on Clojure and Elixir) who started writing browser apps. I progressed through Elm and its Monocle optics package. Then I tried out PureScript. I had a lot of trouble understanding PureScript's optics library, even while leaning heavily on documentation for the Haskell original. So I decided to explain PureScript's version. I don't use Haskell, so I had no particular incentive to switch.

What does that mean for you? Especially if you're a Haskell programmer?

PureScript is close, syntactically and semantically, to Haskell. Its optic package has a different underlying implementation but an API closely modeled after Haskell's. So it's my hope that Haskell programmers will find this book useful. But I won't know if it is until some people try it and tell me.

Getting ready

This book assumes Purescript 0.12.0 or later.

The code for exercises and examples is in https://github.com/marick/purescript-lenses. Clone that repo and run `bower install`.

Insert here the usual speech about how much more you get out of a book when you do the exercises. (It's true.)

About the cover

The cover is "Portrait of Giovanni Arnolfini and his Wife," by Jan van Eyck (1434). I chose it because it's frequently mentioned in support of a theory that the Old Masters used optical devices to help them paint. In the case of van Eyck, the claim is that he used a concave mirror to project an image that he used to trace outlines. Notice that there's a concave mirror between the two people. It's been suggested that that's the very mirror he used to create the painting. (Some people can't resist dropping hints about their clever hacks.)

Your problems and questions

For the moment, either send problems to me or post issues in the Github repo.

Please include the version of the book. This is **version five**.

Change log

version five

- A chapter on `Prism` optics.

version four

- A chapter on `Index` optics.

version three

- A first chapter on `Traversal`.

version two

- Chapter 3 is on refactoring and lenses
- Chapter 2's discussion of `At` lenses is considerably improved. (I hope!) The `Ix` optic has been deferred until after a chapter on `Traversal`.
- Added appendices for (1) a concise optics cheatsheet, (2) tables for all possible optic compositions, and (3) a summary of how to interpret the raw profunctor-function types you get from the compiler.
- Reworked this introduction as the book's approach solidifies.
- Upgraded to PureScript 0.12 and changed text to match changed names and locations.
- I'm now following the convention of naming lenses with a _ prefix. So Chapter 1's tuple lens is named `_first`. When I refer to the built-in version, I'll use `_1` instead of `first`.

version one

- The `Lens` type, using tuples and records as examples.
- `At` and `Ix` optics.

Thanks for the help

These people provided helpful comments on drafts or answered my questions:

cheater
Christoph Hegemann
Dan Lien
Dave Fayram
Gary Burgess
Jeremy W. Sherman
Josh Bohde
Josh Graham
Mark Seemann
Matt Parsons
Phil Freeman
Soham Chowdhury

I apologize if I overlooked someone.

1. Tuples and records (lenses for product types)

Synopsis

Types: `Lens`, `Lens'`
Constructors: `lens`
Functions: `view`, `set`, `over`
Predefined optics: `_1`, `_2`

Consider these ways of getting a value from some kind of container:

```
> fst $ Tuple 5 "hi"
5

> _.a {a : 5}
5

> Array.index [0, 5, 50] 1
(Just 5)

> Map.lookup "a" $ Map.singleton "a" 5
(Just 5)
```

For the first two examples, the compiler guarantees that the getter (`fst` or `_.a`) has something to get. The second two don't have that guarantee, so they must return a `Maybe`.

In this chapter, we'll work on types with the first property. Those are handled by the `Lens` optic.

I encourage you to experiment in the repl. To that end, you can find this chapter's optic and data definitions in src/Product.purs. Throughout the book, I'll start such files with `import` statements to paste into the repl:

```
{- Paste the following into the repl

import Product
import Data.Lens (lens, view, set, over, _1, _2)

import Data.Tuple
import Data.String as String
-}
```

With luck, they'll keep you from getting "I don't know what that token means" errors from the repl. I don't want to increase the total amount of avoidable annoyance in the world.

The `psc-ide` package is a backend that can be integrated into an editor. Among other things, it can add and update `import` statements for you. See "A quick introduction to psc-ide." There is an Emacs integration, an Atom integration, and quite likely others.

For this book, I'm *not* assuming that you're using `psc-ide`, so I'll be explicit about imports.

1.1 Tuples

`Lens` values are constructed with the `lens` function. It takes two arguments, a *getter* and a *setter*. The `Product` module shows a long-winded way of constructing a lens for the first element of a `Tuple`:

```
_first =
  lens getter setter
  where
    getter = fst
    setter (Tuple _ kept) new =
      Tuple new kept
```

- There's something of a convention of naming optics with a _ prefix. I'll follow that convention in this book.
- Getters always have the type `whole -> part`.
- Setters always have the type `whole -> part -> whole`.
- A setter always produces a new `whole` with exactly and only the `part` replaced. (In this example, the unchanged component is named `kept`.)

Once you have a `Lens` like `_first`, you use its getter via the metaphorically-named function `view`:

```
> view _first $ Tuple "one" 1
"one"
```

You've "focused in" from a whole to what I'll call the *part* or *focus* (depending on what I want to emphasize.)

You use the lens's setter with the non-metaphorical `set`:

```
> set _first "harangue" $ Tuple "one" 1
(Tuple "harangue" 1)
```

Notice that the argument order is the opposite of the original setter's: with `set`, the new `part` comes before the `whole` that's to be "modified."

You can transform the focus element with the differently-metaphorical `over`:

```
> over _first String.toUpper $ Tuple "one" 1
(Tuple "ONE" 1)
```

One magical thing about lenses is that `set` and `over` can produce a different type than they consume. Here, for example, I replace a string with its length:

```
> over _first String.length $ Tuple "one" 1
(Tuple 3 1)
```

> `Data.Lens` defines two `Tuple` lenses, `_1` and `_2`. `_1` is the same as our `_first`, and `_2` works on the second `Tuple` element:
>
> ```
> > set _2 "no-longer-an-Int" $ Tuple "one" 1
> (Tuple "one" "no-longer-an-Int")
> ```
>
> (Strictly, they're defined in `Data.Lens.Common`. The `Data.Lens` module exists just to re-export names from many other `Data.Lens.*` modules. Once you understand optics, it'll be a good idea for you to browse its documentation.)

1.2 Records

In this section, I'll use the following record (also found in `Product`) as an example:

```
duringNetflix = { subject : "Brian"
                , object : "Dawn"
                , action : "cafuné"
                , count : 0
                }
```

("Cafuné" is a Portguese idiom meaning "stroking or running one's fingers through a loved one's hair.")[1]

Because every record is a different type, there's no equivalent to the predefined `_1` and `_2` lenses. You'll have to define a new lens for each field you want to work with. Here's a verbose way of defining a lens that focuses on the `action` field:

```
_action =
  lens getter setter
  where
    getter = _.action
    setter whole new = whole { action = new }
```

Here's a terser definition that focuses on the `count` field:

```
_count = lens _.count $ _ { count = _ }
```

Now our three optics functions work with both `Tuple` and `Record` lenses:

```
> view _action duringNetflix
"cafuné"

> over _action String.toUpper duringNetflix
{ action: "CAFUNÉ", count: 0, object: "Dawn", subject: "Brian" }
          ^^^^^^^^

> set _count 999 duringNetflix
{ action: "cafuné", count: 999, object: "Dawn", subject: "Brian" }
                    ^^^^^^^^^^
```

Changing the type of a record's field works as it did for tuples:

[1] I'm learning Portuguese, and I'm not very good at languages. Se meus exemplos são errados, por favor, seja tolerante.

```
> :t duringNetflix
{ subject :: String
, object :: String
, action :: String       <<<<----
, count :: Int
}

> :t over _action String.length duringNetflix
{ subject :: String
, object :: String
, action :: Int          <<<<----
, count :: Int
}
```

1.3 Composing lenses

Lenses are most useful for nested data structures. Here I nest last section's `duringNetflix` inside a tuple:

```
both = Tuple "example" duringNetflix
```

Lenses use the normal composition operator `(<<<)` to construct a path into a nested structure:

```
> _bothCount = _2 <<< _count
> view _bothCount both
0

> both # set _bothCount 55 # view _bothCount
55
```

Note that the path goes from left to right, even though the composition operator points in the other direction.

1.4 Composition exercise

Start from the file named `ProductExercises.purs`. You can find my solution in `ProductSolutions.purs`.

1. Define a lens for the `object` field of the `duringNetflix` record. Name it `_object`.
2. Using a lens, create a value `stringified` that is a version of `duringNetflix` with that record replaced by a String:

```
> stringified
(Tuple "example"
       "{ action: \"cafuné\",
          count: 0,
          object: \"Dawn\",
          subject: \"Brian\" }")
```

(I've added line breaks so the single line doesn't sprawl past the margin.)

3. Using lens composition, convert `both` into this structure:

   ```
   > ... both ...
   (Tuple "example" "Dawn")
   ```

   That is, the result's second element, `"Dawn"`, was originally the `object` value in the original tuple's second element.

1.5 Types

The mechanism underlying lenses leads to complexity in the types the compiler infers. Let's look at our `_first`:

```
> :t _first
forall p t6 t7 t8. Strong p =>
  p t6 t8 -> p (Tuple t6 t7) (Tuple t8 t7)
```

There's a certain "Where's Wally?"[2] character to such types: you need to pattern-match signals out of the noise. The type `Strong` is the signal you have a lens rather than some other optic. You can ignore the rest of that line.

All optics have four crucial types, and it's easiest to find them by thinking of what `set` does. So notice the two uses of `Tuple`:

```
> :t _first
forall p t6 t7 t8. Strong p =>
  p t6 t8 -> p (Tuple t6 t7) (Tuple t8 t7)
               ^^^^^         ^^^^^
```

They tell us `set` will consume one tuple and produce another. We can do a blink-comparison[3] between the tuples to identify the single focus element that changes:

[2] "Where's Wally" (named "Where's Waldo?" in the USA and Canada) is a series of children's books. Wally wears a distinctive red-and-white-striped shirt and matching hat. Each page of the book hides him inside groups of people doing various strange things. The reader's task is to find him.

[3] A blink comparator is the astronomical device that was used to discover Pluto. It's loaded with two images of the same star field, taken some time apart. Either image can be projected to the same location. The astronomer switches rapidly between the two, making any object that's moved appear to jump back and forth. The human perceptual system is really good at detecting such changes.

```
  p t6 t8 -> p (Tuple t6 t7) (Tuple t8 t7)
                    ^^             ^^
```

Type variables `t6` and `t8` differ because, as we've seen, `set` and `over` can change the type of the focus element.

The rest of `_first`'s type signature is only relevant if you care how lenses are implemented. For now, we don't.

When annotating your own lenses, you can simplify the type signature with the `Lens` type alias. It takes four types, which are traditionally named `s`, `t`, `a`, and `b`:

```
type Lens s t a b = ...
```

s t a b

- `s` represents the type of the `whole` that functions like `set` take.
- `t` represents the type of the `whole` that `set` produces.
- `a` represents the original `part` given to `set` or `over`.
- `b` represents the type of the value that replaced the `a`.

Note: if `b` narrows to a different concrete type than `a` does, `t` will necessarily be a different type than `s`.

Here are `Lens` annotations for two of our lenses:

```
_first :: forall a b ignored .
          Lens (Tuple a ignored)
               (Tuple b ignored)
               a b

_action :: forall a b ignored .
           Lens {action :: a | ignored }
                {action :: b | ignored }
                a b
```

Notice that the `_action` lens will work on *any* record with an `action` field:

```
> set _action 5 {action : "figure"}
{ action: 5 }
```

I myself rarely use that feature. Instead, I give my records type aliases and define lenses in terms of them. So, in `Product`, I've added the type alias `Event` for `duringNetflix`'s type, and my type annotation for `_count` looks like this:

```
_count :: Lens Event Event Int Int
```

Notice this type annotation doesn't let `set` or `over` change the type of the `count` field. Since that's often the case, there's a shorter type alias called `Lens'`.

```
_count :: Lens' Event Int
```

Note the apostrophe after `Lens`. You'll find two-type aliases like `Lens'` for all the four-type aliases like `Lens`. Here's the definition of `Lens'`:

```
type Lens' s a = Lens s s a a
```

Reading types

Even after type annotations, the printed type of a lens is wordy:

```
> :t _action
forall p a b ignored.
  Strong p => p a b
              -> p
                   { action :: a
                   | ignored
                   }
                   { action :: b
                   | ignored
                   }
```

When faced with something like that, I often simplify by looking at the type of a partial application, like this one of `set`:

```
> :t set _action "String"
forall t5 t7.
  { action :: t7
  | t5
  }
  -> { action :: String
     | t5
     }
```

1.6 Composition exercise two

`Data.Tuple.Nested` has some convenience functions and types for working with nested two-element `Tuples`:

```
> import Data.Tuple.Nested
> fourLong = tuple4 1 2 3 4

> fourLong
(Tuple 1 (Tuple 2 (Tuple 3 (Tuple 4 unit))))

> :t fourLong
Tuple Int (Tuple Int (Tuple Int (Tuple Int Unit)))
```

The getter functions are named `get1`, `get2`, `get3`, and so on. They're convenient because each works on a tuple of the listed size *or larger*:

```
> get1 fourLong
1

> get2 fourLong
2
```

As you might expect, the compiler objects when it knows the tuple is too short:

```
> get5 fourLong
Could not match type
 Unit
with type
 Tuple t4 t5
```

There are two parts to this exercise. There are hints following the second part. My solution is in the second half of `src/ProductSolutions.purs`.

The `Data.Tuple.Nested` documentation lists but does not (yet) describe types you'll want to use. So here's the description I wish it had:

- The `Tuple`*n* types describe tuples created with the corresponding `tuple`*n* constructors (or their equivalent). The nested structure has *n+1* elements, including the trailing `unit`.
- The `T`*n* types describe tuples *at least* *n* elements long, counting the "tail" of the structure (possibly just `unit`) as `1`. That is, an `id` specialized to take tuples of length two or longer would use `T3` in its type:

  ```
  t2Id :: forall a b rest. T3 a b rest -> T3 a b rest
  t2Id = id
  ```

You might also want to look at the `source` to see how the types are used.

Part 1:

`Tuple.Nested` comes with no setters. To start this exercise, implement `set1`, `set2`, and `set3`:

```
> set2 fourLong "TWOTWOTWO"
(Tuple 1 (Tuple "TWOTWOTWO" (Tuple 3 (Tuple 4 unit))))
```

Continue working in `src/ProductExercises.purs`: it has all the imports you'll need.

Part 2:

Create lenses `_elt1`, `_elt2`, and `_elt3` that combine the appropriate getters and setters. You're done when the following works:

```
> over elt3 ((*)60000) fourLong
(Tuple 1 (Tuple 2 (Tuple 180000 (Tuple 4 unit))))
```

Don't forget to annotate your lenses with their type.

Hint: Start with `set1` and define `set2` in terms of it.

Hint: You want to use the `T`*n* types instead of the `Tuple`*n* types.

1.7 Lens laws

`lens` requires a getter and setter whose types are related to each other:

```
lens :: forall s t a b.
        (s -> a) -> (s -> b -> t) -> Lens s t a b
        ^getter^    ^^^^setter^^^
```

However, not every type-compliant pair will produce a *well-behaved* lens. Proper behavior is defined by three *lens laws*. I'll describe each law three ways: in words, algebraically (with ≡ meaning "produces the same result as"), and with an example.

Note: the law names are conventional, even though lens packages usually use `view` instead of `get`.

set-get:

`view` retrieves what `set` puts in.

```
set _lens >>> view _lens ≡ id
```

```
> set _1 "NEW" (Tuple 1 2) # view _1
"NEW"
```

get-set:

If you set the focus to the value it already has, the whole isn't changed.

```
view _lens >>> set _lens ≡ const   -- or...
set _lens (view _lens whole) ≡ id  -- for any `whole`

> tuple = (Tuple 111 222)
> view _1 tuple
111

> set _1 111 tuple
(Tuple 111 2222)
```

set-set:
Setting the focus twice is the same as setting it once. (This, for example, rules out a setter that not only changes the focus but also increments a counter.)

```
--- for any value `new`:
set lens new >>> set lens new ≡ set lens new

> tuple = Tuple "OLD" "-"
> set _1 "NEW" tuple
(Tuple "NEW" "-")

> set _1 "NEW" tuple # set _1 "NEW"
(Tuple "NEW" "-")
```

1.8 Law exercise

For each of the laws, write an incorrect lens – one that fails to satisfy it. (Because the laws are interrelated and not necessarily minimal, you might not be able to write a lens that fails *only* that law.)

1. A lens where you don't get back what you put in (set-get).
2. A lens where taking something out and putting it back in results in a different `whole` (get-set).
3. A `set` that's affected by an earlier `set` (set-set).

My solution is, again, in `src/ProductSolutions.purs`.

1.9 What now?

`Map` is a common data structure. There are two optics to help you work with them (and also other collection-like types, such as `Array`). One is a variant of `Lens`, so we'll look at it next.

2. Maps (`At` lenses)

Synopsis

Types: `class At` (a `Lens`)
Constructors: `at`
Functions: `setJust`
Predefined optics:

Maps differ from tuples and records in that elements can be deleted and added as well as updated:

```
> Map.insert "key" 5 Map.empty
(fromFoldable [(Tuple "key" 5)])

> Map.delete "key" $ Map.singleton "key" 5
(fromFoldable [])
```

Let's suppose we want a `Lens` for maps. Like the last chapter's optics, it should work with `view`, `set`, and `over`.

2.1 A custom implementation

Let's think through some properties of such a custom lens. For concreteness, we'll assume it focuses in on the `"key"` element:

```
_key :: forall focus. Lens' (Map String focus) ?????
_key =
  lens getter setter
  where
    getter = ???
    setter = ???
```

The getter can't return the `focus` element, because the map might not contain `"key"`. It has to return a `Maybe`, just as `Map.lookup` does. That means we have this lens type:

```
_key :: forall focus. Lens' (Map String focus) (Maybe focus)
                                               ^^^^^^^^^^^^^
```

And we might as well use `Map.lookup` as the getter:

```
_key =
  lens getter setter
  where
    getter = Map.lookup "key"
             ^^^^^^^^^^^^^^^^
    setter = ???
```

What about the setter? If we have a lens, we should obey the lens laws. In particular, we should be able to `set` back what we `view` and not change the map:

```
  set _lens (view _lens whole) ≡ id  -- for any `whole`
```

That means that `set` must itself take a `Maybe`, and the setter must handle both cases (lines 5-8).

```
1  _key =
2    lens getter setter
3    where
4      getter = Map.lookup "key"
5      setter whole wrapped =
6        case wrapped of
7          Just new -> ???
8          Nothing -> ???
```

What if the setter's argument is `(Just new)`? Then `new` must be inserted into the map at the `"key"`:

```
    setter whole wrapped =
      case wrapped of
        Just new -> Map.insert "key" new whole
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^
        Nothing -> ???
```

`Map.insert` is what database people call an "upsert": it replaces a value if present, but otherwise inserts it. That's consistent with the lens laws. In particular, if you `set` a `Just focus`, a `view` on the result will return that same `Just focus`, whether or not the `set` created a new value or replaced an old one.

That `set-get` law also tells us how `Nothing` must behave: if we `set` a `Nothing`, a subsequent `view` must produce `Nothing` – whether or not the original map had a `"key"`. That can be implemented with `Map.delete`, giving us this complete definition of `_key`:

```
_key :: forall focus .
        Lens' (Map String focus) (Maybe focus)
_key =
  lens getter setter
  where
    getter = Map.lookup "key"
    setter whole wrapped =
      case wrapped of
        Just new -> Map.insert "key" new whole
        Nothing -> Map.delete "key" whole
                   ^^^^^^^^^^^^^^^^^^^^
```

2.2 `At.at`

You wouldn't want to write a lens like `_key` more than once, so let's generalize the code above to a function. While we're at it, let's allow non-`String` keys:

```
_atKey :: forall key focus . Ord key =>
                  ^^^
          key -> Lens' (Map key focus) (Maybe focus)
          ^^^^^^             ^^^
_atKey key =
       ^^^
  lens getter setter
  where
    getter = Map.lookup key
                        ^^^
    setter whole wrapped =
      case wrapped of
        Just new -> Map.insert key new whole
                               ^^^
        Nothing -> Map.delete key whole
                              ^^^
```

We can further generalize. `atKey` works with `Map`, but not the related type `Foreign.Object`. (`Foreign.Object` is a thin facade over JavaScript objects. It acts like roughly like a `Map` that requires `String` keys.) We could copy the `_atKey` definition, do a little substitution, and then have an `atKey` variant for `Foreign.Object`. But that sort of mechanical transformation is what type classes are for, so it's not surprising there's a type class with both `Map` and `Foreign.Object` instances. It's called `Data.Lens.At.At`, and it has the single required member `at`:

```
at :: a -> Lens' m (Maybe b)
```

Once we've imported `Data.Lens.At`, we can write polymorphic code like this:

```
> set (at "key") (Just 3) $ Map.empty
> set (at "key") (Just 3) $ Object.empty
```

Useful things to know about `At` lenses

1. The `setJust` function wraps the new `set` value in `Just` for you:

   ```
   > setJust (at "key") 3 $ Map.empty
        ^^^^            ^
   ```

2. Since `over` is a `view`, transformation, and `set`, you have to cope with the `Maybe` result from `view`. That is, this won't work:

   ```
   > over (at "key") negate $ Map.singleton "key" 3
     No type class instance was found for
       Data.Ring.Ring (Maybe Int)
                      ^^^^^^^^^^^
   ```

   You need to lift the `negate`:

   ```
   > over (at "key") (map negate) $ Map.singleton "key" 3
                     ^^^
   (fromFoldable [(Tuple "key" -3)])
   ```

3. Neither `set` nor `over` inform you when they do nothing because the key is missing. You have to use `view` first if you want to check.

Sets

You can think of `Set` as a `Map` all of whose keys have the "nothing to see here, boss" value: `unit`. So you can use `(at 1)` to test for the existence of `Set` element `1`:

```
> view (at 1) $ Set.singleton 1
(Just unit)
```

`view`'s result for a `Set` is kind of useless, so you might convert it into a boolean value:

```
> isJust $ view (at 1) $ Set.singleton 1
true
```

You can delete a `Set` element using `Nothing`:

```
> set (at 23.5) Nothing $ Set.singleton 23.5
(fromFoldable Nil)
```

It's a bit awkward, but you add a `Set` element by giving `set` the value (`Just unit`):

```
> set (at 'b') (Just unit) $ Set.singleton 'a'
               ^^^^^^^^^^^
(fromFoldable ('a' : 'b' : Nil))
```

That perhaps looks a bit better with `setJust`:

```
> setJust (at 'b') unit $ Set.singleton 'a'
  ^^^^^^^          ^^^^
(fromFoldable ('a' : 'b' : Nil))
```

2.3 Types

The type of an `At` lens is characteristically obscure:

```
1  > :t at 3
2  forall t1 t3. At t1 Int t3
3    => (forall p. Strong p =>
4    p (Maybe t3) (Maybe t3) -> p t1 t1)
```

Line 3's tells use we have a `Lens` by using `Strong`. A clue that we have an `At` lens is the use of `Maybe` on the last line:

```
1  > :t at 3
2  forall t1 t3. At t1 Int t3
3    => (forall p. Strong p =>
4                  ^^^^^^
5    p (Maybe t3) (Maybe t3) -> p t1 t1)
6      ^^^^^^^^^^^^^^^^^^^^
```

That's not a strong enough clue, though. Consider the folowing record, a lens for one of its fields, and the lens's type:

```
type Rec= { field :: Maybe Int }

_field :: Lens' Rec (Maybe Int)
_field = lens _.field $ _ { field = _ }

> :t _field
forall p.
  Strong p => p (Maybe Int) (Maybe Int)
  ^^^^^^         ^^^^^^^^^^^ ^^^^^^^^^^^
              -> p
                   { field :: Maybe Int
                   }
                   { field :: Maybe Int
                   }
```

That looks like an `At` lens, but creation and deletion don't work right. For example, I can't add a `field` to a record that doesn't have one:

```
> set _field (Just 3) { different : Just 8 }
Error found:
  Could not match type
    ()
  with type
    ( different :: Maybe Int
    | t0
    )
```

So the better indicator of an `At` lens is the constraint on the `At` type class (line 2):

```
1  > :t at 3
2  forall t1 t3. At t1 Int t3
3                ^^^^^^^^^^^^
4    => (forall p. Strong p =>
5    p (Maybe t3) (Maybe t3) -> p t1 t1)
```

... although I find that clue easy to overlook. (The `At` blends in with the surrounding `t1`, and `t3` type variables.)

Once you've determined you have an `At` lens, creating a `Lens'` type annotation is similar to the previous chapter:

```
_at3 :: forall whole part. At whole Int part =>
        Lens' whole (Maybe part)
_at3 = at 3
```

If you're content to work with a single `whole` type (like `Map`) rather than the `At` type class, you can further simplify:

```
_at3' :: forall part. Lens' (Map Int part) (Maybe part)
_at3' = at3
```

A note on type narrowing

At this point, an exercise had you create a version of `at` that only allowed sets. The solution would have a type like this:

```
_element :: forall a . Ord a =>
            a -> Lens' (Set a) (Maybe Unit)
_element = at
```

Unfortunately, there's a known issue with that. Specifically, the compiler fails to match the declared type annotation against the actual type signature of `at`.

I mention this here because you might run into the same issue. The solution is to add a parameter to `_element`:

```
_element :: <same annotation>
_element x = at x
             ^      ^
```

2.4 Composing `At` lenses

You can add `At` lenses to the end of a pipeline of lenses:

```
> :t _1 <<< _2 <<< at 3
forall t13 t14 t19 t5 t8.
  Strong t8 => At t14 Int t19 =>
  t8 (Maybe t19) (Maybe t19) ->
    t8 (Tuple (Tuple t13 t14) t5) (Tuple (Tuple t13 t14) t5)
```

The result is an `At` lens because it has `At t14 Int t19` in its type.

But that's perhaps not the most attractive type. Write a friendlier one using `Lens'`. My answer follows.

You might have been able to go straight to the solution, but I goofed up when I tried that, so let's start with some helpful renaming:

```
forall profunctor focus focusHolder _1_ _2_ .
  Strong profunctor => At focusHolder Int focus =>
  profunctor (Maybe focus)
             (Maybe focus) ->
    profunctor (Tuple (Tuple _1_ focusHolder) _2_)
               (Tuple (Tuple _1_ focusHolder) _2_)
```

I should explain the type variable names.

`profunctor`:

`Profunctor` is a type class that we'll consider magic for now (and, really, forever, if you skip the entirely optional implementation chapters). All optics are, in the end, a function from one profunctor to another:

```
forall blah blah blah .
  blah blah => blah blah =>
  profunctor a b -> profunctor s t
```

`s`, `t`, `a`, and `b` mean what they meant in the last chapter: original `whole`, new `whole`, original `part`, new `part`.

Naming the profunctor `profunctor` makes the type easier to read than would leaving it as `t8`. Most people name profunctors `p`, but I was an English major.[1] Every so often I like to strike a blow for words over symbols.

`focus` and `focusHolder`:

`focus` is the type of the value we ultimately care about. It's not the `s t a b` "`a`" value because that's a wrapping `Maybe`. Our focus is on the type within the `Maybe`.

Similarly, we don't care that the `focus` is buried inside a nested tuple structure. What matters to us is the relationship between the `focus` and the `At` instance that holds it, so I'm calling the enclosing structure `focusHolder`.

[1] BA English Literature, University of Illinois, 1981.

`_1_` and `_2_`:
: These represent the parts of the tuples that are ignored by the composed optic.

  I should confess that I'm unhealthily obsessive not just about words, but also about naming conventions. I want such "ignorables" to (1) signal that they're uninteresting, (2) be unobtrusive enough that they're easy to ignore, (3) not be a letter `a` because the word and symbol look so much alike that I'm always thinking the symbol is the word, and (4) not even be any single letter because even those that don't confuse when seen in running text mess up global search and replace. (Like replacing `b` in code that contains `Maybe`.)

  Please tolerate my quirks, just as I tolerate yours.

Here's the renaming, again:

```
forall profunctor focus focusHolder _1_ _2_ .
  Strong profunctor => At focusHolder Int focus =>
  profunctor (Maybe focus)
             (Maybe focus) ->
    profunctor (Tuple (Tuple _1_ focusHolder) _2_)
               (Tuple (Tuple _1_ focusHolder) _2_)
```

Given the more meaningful type variables, the conversion to `Lens'` form is easier:

```
composed :: forall focus focusHolder _1_ _2_ .
  At focusHolder Int focus =>
  Lens'
    (Tuple (Tuple _1_ focusHolder) _2_)
    (Maybe focus)
composed = _1 <<< _2 <<< at 3
```

And `composed` works as you probably expect:

```
> threeDeep = Tuple (Tuple "_1_" $ Map.singleton 3 "match") "_2_"

> view composed threeDeep
(Just "match")

> over composed (map String.toUpper) threeDeep
(Tuple (Tuple "_1_" (fromFoldable [(Tuple 3 "MATCH")])) "_2_")
```

Oops

Having added an `At` lens after a `Lens`, let's do the reverse:

```
> at 3 <<< _1
  Could not match type
    Tuple t3
  with type
    Maybe
```

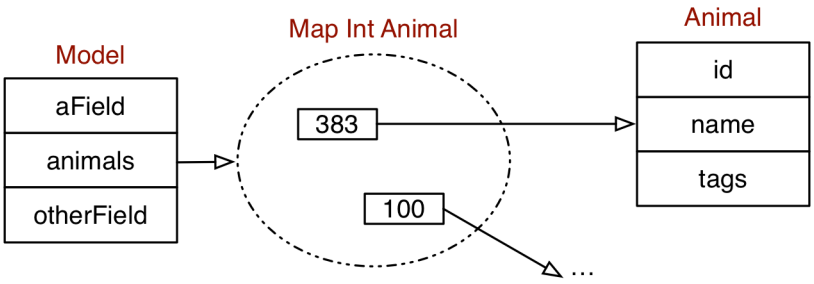
That doesn't work. You can't even compose two `At` lenses:

```
> at 3 <<< at 2
Error found:
  Could not match type
    Unit
  with type
    Int
```

The problem is that the second lens expects some `whole` type, but it gets a `Maybe whole`. This problem is easily fixable, but I'm not going to tell you how yet.

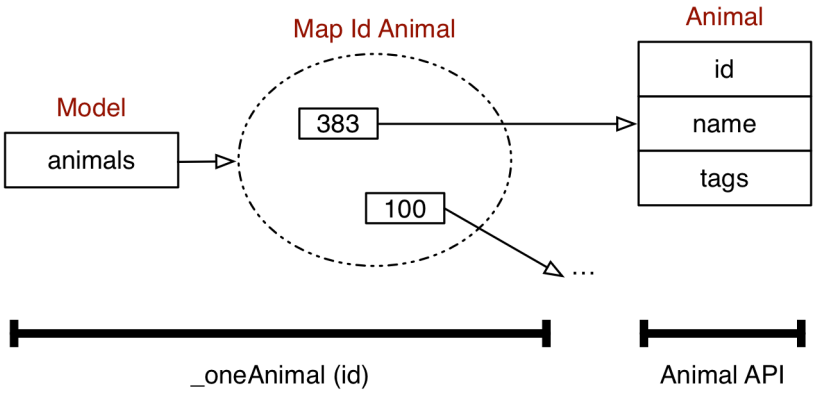
2.5 What now?

It seems wrong to leave `At` lens composition half finished. Remember the very first diagram in the book?



It's extremely common for a `Map`'s values to be structures of some type: perhaps records, perhaps something even more complicated. If we don't know how to append optics to `At` lenses, aren't we making optics useless for their most common case?

Well, if you'll forgive some design advice from the object-oriented world, structures after a `Map` are often the right place to put an encapsulation boundary. That is, rather than an optic that reaches all the way down to an `Animal`'s `tags` array, perhaps we'd be better off with two partial paths. Like this:

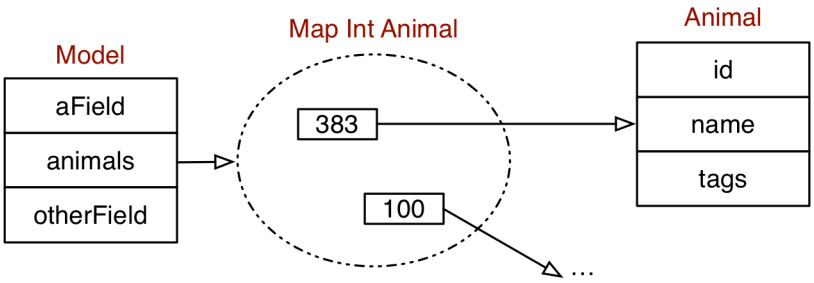


I'll show an example of that in the next chapter, which also gives me an excuse to talk about how optics help refactoring. After that, I'll describe another core optic type, `Traversal`. It solves our problem extending a path past an `At` lens, but that's really a side effect of its main purpose, which is to work with all elements within a collection.

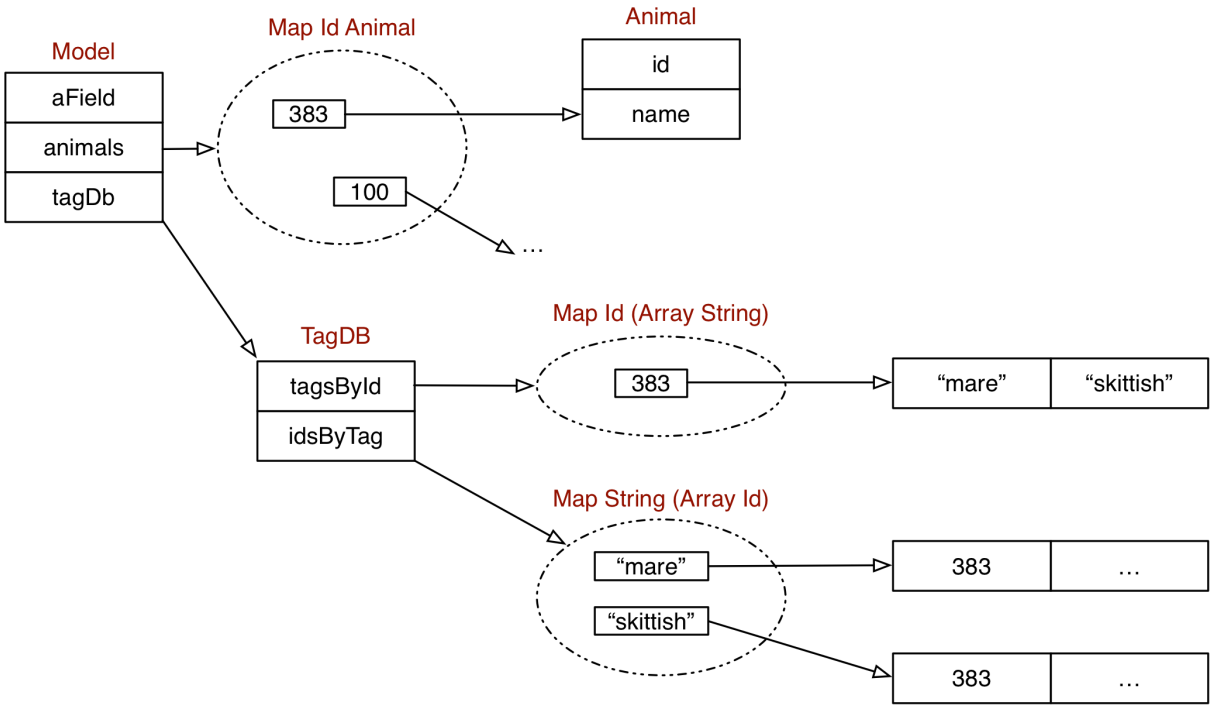
> There's a bug somewhere that causes ≡ to be rendered incorrectly in this chapter unless I include the text "Schrödinger's" (apostrophe required) in it. There's a quantum mechanics joke in there somewhere.

3. Optics and refactoring (optional)

Lenses can make changing your mind easier. In this chapter I'll show how they simplify refactoring this structure:



... into this one:



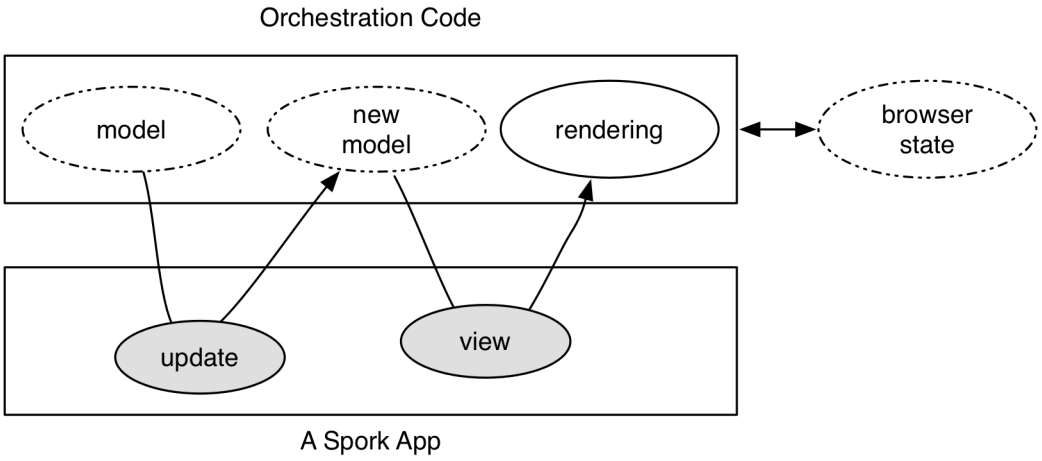
The refactoring is rather simple, but I hope it gives you a feel for how optics help you isolate knowledge of paths – and how that means a change to the interior of a deeply-nested structure doesn't require you to change all of the code that depends on it.

Yes, the compiler won't let you miss any places that need changing. But wouldn't it be nice if it pointed you at *fewer* places?

You can find sample code and repl imports in the Critter4Us directory.[1]

3.1 Architecture

Our code is modeled after the Spork framework, which is itself modeled after the Elm Architecture.



In such an architecture, all the data used to construct an HTML page is stored in a single record, traditionally named `Model`. In our example, the model will contain a collection of `Animal`s, each of which has various properties. In this example, we're interested in an animal's `tags`, which is an `Array String`.

The orchestration code receives `Action`s (Elm calls them `Msg`s) from various sources: button clicks, HTTP responses, timer ticks, and so on. Our example will have two actions:

```
data Action
  = AddAnimal Animal.Id String
  | AddTag Animal.Id String
```

All actions are fed into a single function, `update`, that changes the `Model`. The changed `Model` is normally used to construct a new `view`, which is displayed to the user. This example won't bother constructing a `view`. Instead, you'll explore changes at the repl.

Animals

The `Animal` type lives in `Critter4Us/Animal.purs`:

[1] `Critter4Us` is the name of an app I wrote long ago to manage reservations of teaching animals for a college of veterinary medicine.

```
module Critter4Us.Animal ...

type Id = Int
type Tags = Array String

type Animal =
  { id :: Id
  , name :: String
  , tags :: Tags
  }
```

That concrete structure, though, is hidden behind an interface that provides these functions:

```
named :: String -> Id -> Animal
addTag :: String -> Animal -> Animal
clearTags :: Animal -> Animal
```

> It's my habit to import modules as qualified names, using open imports only for types:
>
> ```
> import Critter4Us.Animal as Animal
> import Critter4Us.Animal (Animal)
> ```
>
> So I expect external code that creates an animal to look like this:
>
> ```
> Animal.named "bossy" 3838
> ```

The model

The `Model` type lives in `Critter4Us/Model.purs`, and is unrealistically small:

```
module Critter4Us.Model ...

type Model =
  { animals :: Map Animal.Id Animal
  }
```

`Model` also provides an interface to be used by `Main.update`:

```
initialModel :: Model
addAnimal :: Animal.Id -> String -> Model -> Model
addAnimalTag :: Animal.Id -> String -> Model -> Model
```

Notice that the model hides the `animals` `Map` inside itself. Changes made deep in the structure (like adding a tag) look no different than shallower changes (like adding an animal). Lenses help make changes easier, but they're not a replacement for thinking about encapsulation boundaries.

Examples

Here are three examples of using this little program. I've added whitespace for clarity. You don't have to type these commands yourself; instead, you can copy them from a comment in `Critter4Us/Main.purs`.

The first shows a starting value for the `Model`:

```
> initialModel
{ animals: (fromFoldable
  [(Tuple 3838 { id: 3838, name: "Genesis", tags: ["mare"] })]
) }
```

Here, I add an animal:

```
> update initialModel (AddAnimal 1 "Bossy")
{ animals: (fromFoldable
  [ (Tuple 1    { id: 1,    name: "Bossy",   tags: [] })  # <<<<---
  , (Tuple 3838 { id: 3838, name: "Genesis", tags: ["mare"] }
)]) }
```

... and here I add a tag:

```
> update initialModel (AddTag 3838 "skittish")
                                   ^^^^^^^^^^

{ animals: (fromFoldable
  [(Tuple 3838 { id: 3838, name: "Genesis",
                 tags: ["mare","skittish"]
                               ^^^^^^^^^^

})]) }
```

3.2 Optics

The implementations of `Model.addAnimal` and `Model.addAnimalTag` use `_oneAnimal`, a function that makes an `At` lens:

```
addAnimal :: Animal.Id -> String -> Model -> Model
addAnimal id name =
  setJust (_oneAnimal id) (Animal.named name id)
          ^^^^^^^^^^^^^^^

addAnimalTag :: Animal.Id -> String -> Model -> Model
addAnimalTag id tag =
  over (_oneAnimal id) (map $ Animal.addTag tag)
       ^^^^^^^^^^^^^^^
```

`_oneAnimal` uses `_animals`, a `Lens` from a record into its `animals` field:

```
_animals :: Lens' Model (Map Animal.Id Animal)
_animals =
  lens _.animals $ _ { animals = _ }
```

… which `_oneAnimal` composes with an `At` lens:

```
_oneAnimal :: Animal.Id -> Lens' Model (Maybe Animal)
_oneAnimal id =
  _animals <<< at id
           ^^^^^^^^^
```

The functions in `Animal` are also implemented with lenses, just because this old Lisp programmer prefers functions like `set` to a special record-update syntax. So, for example, here's `clearTags`:

```
clearTags :: Animal -> Animal
clearTags = set _tags []

_tags :: Lens' Animal Tags
_tags = lens _.tags $ _ { tags = _ }
```

Ah, Lisp syntax…

Once upon a time, Martin Fowler and I were staring at some Ruby code, trying to figure out why it wasn't working. Longtime Lisp programmer Richard P. Gabriel was sitting next to me, reading on his laptop. He leaned over, glanced at the code, said in his gravelley voice, "That language has too much syntax," and lost interest. Turns out he was right: it was an operator precedence problem. No such thing in Lisp.

3.3 Exercise

This app makes it easy to show all of an `Animal`'s tags. Let's pretend it were put into production. Within a millisecond, someone would want a report of all animals that share a particular tag. That feature request reveals that tags and animals should have been in a many-to-many relationship all along.

Therefore: in this exercise, we'll plug in a (crude) new type, `TagDb`, that makes such a report easy.

1. If you didn't already run the earlier examples in the repl, import these two modules:

   ```
   import Critter4Us.Main
   import Critter4Us.Model
   ```

2. Remove `tags` from the `Animal` type (in `Animal.purs`):

   ```
   type Animal =
     { id :: Id
     , name :: String
     , tags :: Tags     -- Delete this
     }
   ```

3. In `Model.purs`, uncomment the imports of `TagDb`:

   ```
   -- import Critter4Us.TagDb (TagDb)
   -- import Critter4Us.TagDb as TagDb
   ....
   ```

   and add the new type to the model:

   ```
   type Model =
     { animals :: Map Animal.Id Animal
     , tagDb :: TagDb     -- make this appear
     }
   ```

4. `:reload` the modules, causing them to be recompiled. You'll see errors. Fix them – and, by doing that, complete the switch to the new model. Note that I've already written `TagDb`, including some lenses.

   Note: when I first tried this exercise, I wrote code before the compiler told me I needed it. That turned out to confuse me, so I recommend you not do that.

My solution is in Critter4UsRefactored.

3.4 My solution

The core difference between my two versions of `Model` is that the implementation of `addAnimalTag` changed from this:

```
addAnimalTag :: Animal.Id -> String -> Model -> Model
addAnimalTag id tag =
  over (_oneAnimal id) (map $ Animal.addTag tag)
```

... to this:

```
addAnimalTag id tag =
  over _tagDb $ TagDb.addTag id tag
```

That required adding one new lens:

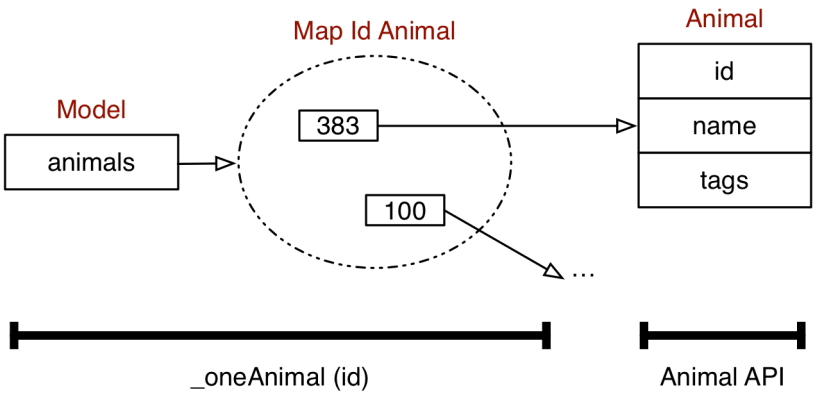
```
_tagDb :: Lens' Model TagDb
_tagDb =
  lens _.tagDb $ _ { tagDb = _ }
```

Since `Critter4UsRefactored.Animal` no longer contains `tags`, the `addTag` and `clearTags` functions had to be deleted, along with their supporting lens `_tags`.

So: the changes were limited to the top-level module `Model` and the deeply-nested `Animal`. (If I hadn't provided it in advance, I would have had to implement the deeply-nested `TagDb`.)

Those changes are significant in a way I think is best represented by two pictures.

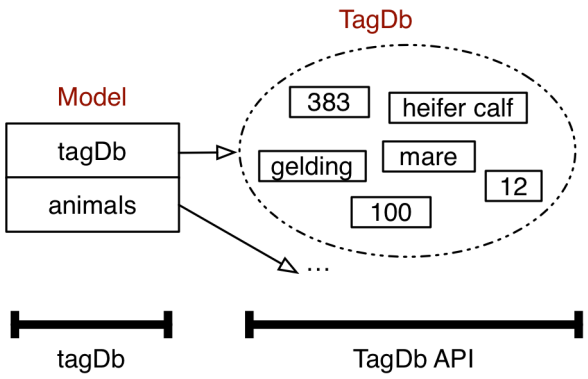
Here's the picture before the change to `TagDb`:



before

`Animal` provides an API to a self-contained data structure. It doesn't expose the optics it uses to the outside world. An `Animal` is reached via a traversal of a data structure, one that's captured by a single optic (`_oneAnimal id`) that composes the path from a `Model` to a `Map` and from a `Map` to (maybe) an `Animal`.

Here's the picture after the change:



after

It's also a two-part path, except that the path to the API is shorter (and it happens that the paths *behind* the API are longer and so use composed lenses that the `Animal` implementation didn't need).

Optics allow you to segment complex data structures into two endpoint types (here `Model` and `Animal`) connected by a separate value (a combined lens) that represents the path between them. If you're lucky, many restructurings can be done with no or very localized changes to intermediate structures.

To amplify that point:

It's often said that static languages allow easier refactoring than dynamic ones. That's roughly true (though I'm not yet as comfortable finding the right *order* for the small refactorings that make up a big one). The reason it's true was shown in the changes you just made to `Critter4Us`: the compiler reminds you of all the places you have to change.

However: if you keep refactoring, such floods of compiler errors can get oppressive. It's easy to reach a point where you decide the next improvement is just too much trouble. Lenses, I believe, can help push that point further away.

For example, consider the nightmare when an endpoint API that used to return a `String` now returns a `Maybe String`. That could require changes in modules all the way back to `Model`. Or it could require changing the optic that describes the path by changing (only) the last segment from a `Lens` to an `At` lens.

> I think of lenses as a tool to allow you to keep making steady progress while your domain model settles down.

3.5 What now?

The next chapter covers a completely new type of optic called a `Traversal`. Using it `over` can apply a function to every element of an array. It has powers, beyond that. As with O+ blood transfusions, `Traversal` is a sort of “universal donor”, compatible with all other optics.

4. Operating on whole collections (`Traversal`, part 1)

Synopsis

Types: `Traversal`, `Traveral'`
Constructors:
Functions: `firstOf`, `lastOf`, `preview`, `toListOf`
Predefined optics: `traversed`

In the previous chapters, you saw optics that focused on individual elements of collections. This chapter is about the `Traversal` type, an optic that focuses on *all* of the values in a collection. Traversals have two somewhat distinct purposes, depending on whether a collection contains effects. This chapter assumes the collection doesn't. So this chapter is about topics like these:

1. Using the usual optic functions (like `over`) to work with all the elements "at once." For example: negating every integer in a two-dimensional array (an array of arrays).
2. Reaching into a structure to extract a collection of (monoidal) values and combine them into a single value. For example: extracting a nested array of strings and appending them into a single string.

You can find sample code and repl imports in src/Traversal.purs.

4.1 Working with all the values

You hardly ever need to create your own `Traversal` optic: the predefined `traversed` will likely satisfy your needs. For example, here's how you negate every element of an `Array`:

```
> over traversed negate [1, 2]
[-1, -2]

> (over traversed negate []) :: Array Int
[]
```

You can also `set` every element:

```
> set traversed 99 [1, 2]
[99,99]
```

`traversed` is constrained to apply only to types that are instances of `Traversable`. For now, all you need to know is that many of your favorite types are instances. You saw `Array` above, but other types – not obviously "collections" – also work:

```
> over traversed negate (Just 3)
(Just -3)

> (over traversed negate $ Right 88) :: Either Int Int
(Right -88)
```

Notice that, like `map`, `traversed` applies to the `Right` value and not the `Left`:

```
> (over traversed negate $ Left 88) :: Either Int Int
(Left 88)
```

Also notice that you can't use `set` to change, say, a `Nothing` to a `Just`:

```
> set traversed 3 Nothing
Nothing
```

Like `map`, `set` can only change elements, not add them.

The `view` function doesn't do what you might guess (see below), so you use `toListOf` (defined in `Data.Lens.Fold`[1]) to do something close:

```
> toListOf traversed [1, 2, 3]
(1 : 2 : 3 : Nil)

> toListOf traversed (Just 3)
(3 : Nil)

> (toListOf traversed $ Left 5) :: List Int
Nil
```

Note that `List` is also `Traversable`:

[1] `Traversal` works with `Traversable` types. `Data.Lens.Fold` works with `Foldable` types. All the predefined `Foldable` types are also `Traversable`, so the only immediate practical effect of the distinction is that you get to guess which `Data.Lens.{Fold,Traversal}` module contains the documentation you want. A *lot* of it is in `Data.Lens.Fold`.

```
> over traversed negate (1 : 2 : 3 : Nil)
(-1 : -2 : -3 : Nil)
```

Beware of Data.Traversable

Data.Lens.Traversal suppplies traversed.
Data.Traversable supples traverse.
Both of those functions apply to the same types (Traversable).

Imagine the fun if you happen to open import both modules! And you type traverse when you meant traversed!

```
> import Data.Traversable
> over traverse negate [1, 2]
Error found:
  Could not match type
    Int
  with type
    t0 t1
```

My advice: always use import as with one of the modules.

4.2 view

view requires that the focus elements be monoids, and it monoidally-appends them together:

```
> view traversed ["D", "a", "w", "n"]
"Dawn"
```

If you use a non-monoidal type, you'll be greeted with this:

```
> view _trav_1 [(Tuple 1 1), (Tuple 23 2)]
Error found:
  No type class instance was found for
    Data.Monoid.Monoid Int

while checking that type forall t19 t20 t21 t22 t23.
Traversable t19 => Wander t20 => t20 t23 t22 -> t20
(t19 (Tuple t23 t21)) (t19 (Tuple t22 t21))
  is at least as general as type Forget t0 t0 t1 -> Forget t0 t2 t3
```

Train the pattern-matching part of your brain to recognize this pattern:

```
  No type class instance was found for
    Data.Monoid.Monoid Int

ttttt tttttttt ttta ttta tttttt ttt ttt ttt ttt ttt.
Traversable ttt => Wander ttt => ttt ttt ttt tt ttt
ttta tttttt ttt ttttt ttta tttttt ttt ttttt
```

… and you'll probably spend less time being deciphering such errors. (Wander is the profunctor used to create a Traversal.)

Integers don't support append, so you can't view a collection of them. The problem with integers is that both multiplication and addition seem like equally valid choices for append. If you want a monoid, you wrap Integers to signify your choice:

```
> append (Additive 1) (Additive 2)
(Additive 3)
```

The same wrapping allows view to work with them:

```
> view traversed $ map Additive [1, 2, 3]
(Additive 6)
```

Exercise: rewrite the above without using map. Use over instead. My answer is on the next page.

```
> view traversed $ over traversed Additive [1, 2, 3]
(Additive 6)
```

I was working on this in the same repl where I'd open-imported `traverse` from `Data.Traversable`. Naturally, I used `traverse` instead of `traversed` in the following:

```
> view traversed $ over traverse Additive [1, 2, 3]
[1,2,3] -- This is not (Additive 6)
```

In a later chapter, I'll explain the relationship between `Traversal` and `Traversable`. That will explain what my code was doing and why. The relationship isn't hugely important – it's more of an implementation detail – but it shows its face enough that understanding it will save you some stress.

4.3 Focusing on individual values

You can fetch individual values. The two most-commonly desired values have built-in functions (documented in `Data.Lens.Fold`):

```
> firstOf traversed [1, 2, 3]
(Just 1)

> lastOf traversed [1, 2, 3]
(Just 3)
```

They have to return a `Maybe` because the "container" might be empty.

You can "narrow" a `Traversal` down to a single element by creating a new traversal. To show that the result really is a `Traversal`, I'll add a type annotation and dare the compiler to contradict me. The `Traversal'` type alias has the normal whole/part form:

```
_element1 :: Traversal' (Array String) String
_element1 = element 1 traversed

> over _element1 String.toUpper ["no", "yes!", "no"]
["no","YES!","no"]

> firstOf _element1 ["no", "yes!", "no"]
(Just "yes!")
```

Narrowing a `Traversal` with `element` always works, but the resulting optic is inefficient. The next chapter shows you a type class whose instances are way more efficient than the general case.

4.4 `preview`

Let me repeat that last line of code:

```
> firstOf _element1 ["no", "yes!", "no"]
```

That reads a bit oddly, so let's use `preview` instead:

```
> preview _element1 ["no", "yes!", "no"]
(Just "yes!")
```

`preview` is a synonym of `firstOf`, typically used for traversals over a type that's either "empty" or contains a single value.[2] We'll be looking at other optics that use `preview` in later chapters.

4.5 Composing traversals

A composition of a `Traversal` and any other optic produces a `Traversal`. Deciding on a type annotation for the resulting optic can be a challenge. I'll show some examples here, but will save some for exercises.

In day-to-day work, I expect you'll more often look up the type in A Catalog of Compositions than figure it out yourself. Nevertheless, I think being *able* to figure it out is important. If you skip the exercises, make sure you understand why the solutions make sense.

`traversed <<< traversed`

Composed traversals are used for things that, in other languages, require nested loops.

```
> over (traversed <<< traversed) negate [ [1, 2], [3, 4]]
[[-1,-2],[-3,-4]]
```

Exercise: Use `view` to convert `[["1"], ["2", "3"]]` to `"123"`. Then use `view` to convert `[[1], [2, 3]]` to `Additive 10`. The answer is in src/Traversal.purs.

`traversed <<< lens`

Adding a lens to a traversal lets you use the lens to focus into each of the values traversed:

[2] They're called "affine traversals" in the API documentation.

```
_trav_1 = traversed <<< _1

> example = [ Tuple 1 2, Tuple 3 4 ]

> over _trav_1 ((*)1111) example
[(Tuple 1111 2),(Tuple 3333 4)]

> toListOf _trav_1 example
(1 : 3 : Nil)
```

Exercise: Predict the result of `preview _trav_1 example`. The answer is in src/Traversal.purs.

Creating a type annotation

As usual, the compiler's type for `_trav_1` is complicated:

```
> :t _trav_1
forall t19 t20 t21 t22 t23.
  Traversable t19 => Wander t20 =>
  t20 t23 t22 -> t20 (t19 (Tuple t23 t21)) (t19 (Tuple t22 t21))
```

`Wander` is the clue that we're looking at a `Traversal`, just as `Strong` is the clue we're looking at a `Lens`. So we know we're looking to reduce the mess into something of this form:

```
Traversal s t a b
```

Substitution helps. Start by changing the names after `Traversable` and `Wander` into `traversable` and `profunctor`, respectively:

```
1 > :t _trav_1
2 forall traversable profunctor t21 t22 t23.
3   Traversable traversable => Wander profunctor =>
4   profunctor t23 t22 ->
5       profunctor (traversable (Tuple t23 t21))
6                  (traversable (Tuple t22 t21))
```

All optics are profunctor functions of this form:

```
profunctor a b -> profunctor s t
```

So line 4 gives us better names for `t23` and `t22`:

```
> :t _trav_1
forall traversable profunctor t21 b a.
  Traversable traversable => Wander profunctor =>
  profunctor a b ->
             ^ ^
      profunctor (traversable (Tuple a t21))
                                     ^
                 (traversable (Tuple b t21))
                                     ^
```

And the right-hand side of the function gives us this particular optic's values for `s` and `t`:

```
s is (traversable (Tuple a t21))
t is (traversable (Tuple b t21))
```

We now have the `s t a b` types to use with the `Traversal` type alias (plus a better-named "don't care" type):

```
_trav_1 :: forall traversable a b _1_.
           Traversable traversable =>
             Traversal (traversable (Tuple a _1_))
                       (traversable (Tuple b _1_))
                       a b
```

lens <<< traversed

You can operate on `Traversable`s within a product type:

```
> _1_trav = _1 <<< traversed

> over _1_trav negate (Tuple [1, 2, 3] "don't care")
(Tuple [-1,-2,-3] "don't care")

> over _1_trav negate (Tuple (Just 3) "don't care")
(Tuple (Just -3) "don't care")
```

At lens <<< traversed

> If you just followed the link from the At lens chapter, the solution is to put traversed after the At lens:
>
> ```
> at 3 <<< traversed <<< _1
> ```

Remember how you couldn't compose another optic after an At lens because an At lens produces a Maybe focus but any following optic wants to consume focus? Does that problem also apply to Traversal?

Nope:

```
> _at3_trav = at 3 <<< traversed
>
```

Let's look at its type to see what we've got:

```
> :t _at3_trav
forall t5 t7 t8. At t5 Int t7 => Strong t8 =>
  Wander t8 =>
  t8 t7 t7 -> t8 t5 t5
```

t8 is the type variable for the profunctor, which means it's a Wander profunctor, which is the type used for Traversal. Because of that, we can build pipelines where the At lens corresponds to a structure's interior node:

```
> _at3_trav_1 = at 3 <<< traversed <<< _1
```

The result is yet another traversal. I'll spare you the compiler's type output and show a friendlier type annotation:

```
_at3_trav_1 :: forall a _1_ .
               Traversal' (Map Int (Tuple a _1_)) a
```

Here's an example of its use:

```
tupleMap = Map.fromFoldable [ (Tuple 3 (Tuple 8 "s"))
                            , (Tuple 4 (Tuple 1 "_2_"))]

> preview _at3_trav_1 tupleMap
(Just 8)

> set _at3_trav_1 444444 tupleMap
(fromFoldable [(Tuple 3 (Tuple 444444 "s"))
              ,(Tuple 4 (Tuple 1 "_2_"))])
```

traversed <<< At lens

Just like any other lens, an At lens can be added to a Traversal:

```
_trav_at3 = traversed <<< at 3
```

Let's see what that does:

```
mapArray = [ Map.singleton 3 "3"
           , Map.empty
           , Map.singleton 4 "4"
           ]

> toListOf _trav_at3 mapArray
((Just "3") : Nothing : Nothing : Nil)
```

That makes sense, since only the first Map has the right key. Notice that the return value must be a Maybe to allow the Nothing case.

Because At lenses allow creation, we can add a key to the maps that are missing them:

```
> set _trav_at3 (Just "NEW") mapArray
[(fromFoldable [(Tuple 3 "NEW")]),
 (fromFoldable [(Tuple 3 "NEW")]),
 (fromFoldable [(Tuple 3 "NEW"),(Tuple 4 "4")])]
```

Or we can change the single key with over:

```
> over _trav_at3 (map $ \s -> s <> "!+!" <> s) mapArray
[(fromFoldable [(Tuple 3 "3!+!3")]),
 (fromFoldable []),
 (fromFoldable [(Tuple 4 "4")])]
```

While this type combines the behavior of a traversal (working with all elements) and an `At` lens (inserting and deleting the focus element), its type is a `Traversal`. See the following exercises.

Composition exercises

The src/TraversalStart.purs file contains a number of definitions without type annotations. (They're commented out to keep the compiler from whining.) Add type annotations. If you're like me, some of them will require asking the compiler for the type, then changing the `tNN` variable names.

If you end up with any examples where the type is `(Traversal s s a a)` or, equivalently, `(Traversal' s a)`, create an example where a type-change would produce nonsensical results. That is, where you can say, "*this* use of `set` on *that* data structure would produce *this* new data structure, which is impossible."

Note: if you have an open repl, it'll be easier if you restart (or `:clear`) and paste the imports from `TraversalStart.purs`. Otherwise, you'll get annoying name clashes with `Traversal.purs`.

Solutions are toward the end of src/Traversal.purs.

4.6 A sort of base class

PureScript doesn't have inheritance, but the implementation of lenses is such that you can apply the `Traversal` functions to any type of optic. Lenses, for example:

```
> toListOf _1 (Tuple 8 "hi")
(8 : Nil)

> preview _1 (Tuple 8 "hi")
(Just 8)
```

I don't offhand see any use for this fact, but now you know.

4.7 What now?

The next chapter is a short one, describing the `Index` type class, which is usually preferable to using `element` and `traversed`.

5. Single elements in arrays and fixed-size maps (`Index`)

}

Synopsis

Types: `class Index` (a `Traversal`)
Constructors: `ix`
Functions: `preview, set, over`
Predefined optics:

`Index` is a type class with a constructor that's used like this:

```
ix 1
```

For many types, the resulting `Traversal` is more efficient than one created with

```
element 1 traversed
```

I prefer to ignore the underlying `Traversal` and think instead about `At`. Like an `At` lens, an `Index` optic focuses on a single element in a collection-like type. Like `At`, the element might be missing (so you expect to see `Maybe`).

The difference is in `set`. An `At` lens can be used to create or delete focus elements. You use `Index` optics when the underlying type either prevents that or you don't want it even if it's allowed.

For example, you'd use an `Index` optic with a `Map` when you want to signal (to a human reader) that it's meaningless (in the context of this program) to add a new key.

5.1 `ix`, `at`, and `Map`

You can find sample code and repl imports in src/Index.purs.

Here are examples of using both `At` and `Index` optics.

Creation looks the same:

```
> _at1 = at 1
> _ix1 = ix 1
```

Because `_at1` is a lens, you use `view`. Because `_ix1` is a traversal, you use `preview`. But the result is nevertheless a `Maybe`.

```
> m = Map.singleton 1 "a"

> view _at1 m
(Just "a")

> preview _ix1 m
(Just "a")
```

Because an `At` lens can create or delete, `set` takes a `Maybe`. Because an `Index` optic doesn't create or delete, it takes an unwrapped value:

```
> set _at1 (Just "new value") m
(fromFoldable [(Tuple 1 "new value")])

> set _ix1 "new value" m
(fromFoldable [(Tuple 1 "new value")])
```

It's important to know, though, that there's no obvious failure if the `Index` optic is used to `set` an object that doesn't exist. The structure is silently unchanged:

```
> set _ix1 "new value" Map.empty
(fromFoldable [])
```

Both `At` lenses and `Index` optics can be used with `over`:

```
> over _at1 (map String.toUpper) m
(fromFoldable [(Tuple 1 "A")])

> over _ix1 String.toUpper m
(fromFoldable [(Tuple 1 "A")])
```

Notice that the `Index` version doesn't need to use `(map String.toUpper)` because the `Nothing`-ignoring behavior is inherent in the definition of `Traversal`.

Like `set`, `over` is silent when the focus element is missing:

```
> over _ix1 String.toUpper $ Map.singleton 2 "irrelevant"
(fromFoldable [(Tuple 2 "irrelevant")])
```

5.2 Array

At lenses don't make sense for arrays. What does it mean to delete an element from the middle? Or to set the tenth element of a two-element array?

Since Index optics don't insert new elements or delete old ones, they don't have that problem.

The same ix 1 optic above can be used with arrays:

```
> preview _ix1 [1, 2]
(Just 2)

> set _ix1 "ignored" []
[]

> over _ix1 negate [0, 1]
[0,-1]
```

5.3 Types

The type of _ix1 is not too complicated:

```
1  > :t _ix1
2  forall t1 t3.
3  Index t1 Int t3 => (forall p. Wander p =>
4     p t3 t3 -> p t1 t1)
```

Notice the type class Index on line 3.

Most likely, you'll want to use Traversal' instead of Wander:

```
_ix1 :: forall s a .
        Index s Int a =>
          Traversal' s a
_ix1 = ix 1
```

5.4 Composition

As in the last chapter, I won't describe all the cases here. Three are left as an exercise (start with src/IndexStart.purs). Try writing the annotations from scratch, rather than by translating the type the compiler shows you.

`traversed <<< ix`

Recall that the type of `(traversed <<< traversed)` is this:

```
forall a b trav1 trav2 .
Traversable trav1 => Traversable trav2 =>
Traversal (trav1 (trav2 a))
          (trav1 (trav2 b))
          a b
```

Now let's consider `(traversed <<< ix 1)`. There are two differences:

1. The second optic works on not just any `Traversable` but specifically an `Index`.
2. Because an `Index` optic works on one of potentially many elements, the type of the composed optic does as well. So it requires a two-type `Traversal'` instead of a four-type `Traversal`.

Putting those facts together, we get this type:

```
forall trav indexed a.
Traversable trav => Index indexed Int a =>
  Traversal' (trav indexed) a
```

`ix <<< traversed`

Consider now `(ix 1 <<< traversed)`. That produces an error message:

```
> ix 1 <<< traversed
Error found:
  The inferred type
    forall t10 t11 t5 t8.
    Index t5 Int (t11 t10) => Wander t8 => Traversable t11 =>
    t8 t10 t10 -> t8 t5 t5

  has type variables which are not mentioned in the
  body of the type. Consider adding a type annotation.
```

I don't understand what the compiler is complaining about. If you do the usual substitution of meaningful names, you get this:

```
    forall a trav indexed p.
    Index indexed Int (trav a) => Wander p => Traversable trav =>
    p a a -> p indexed indexed
```

… which is easily rewritten into `Traversal'` form:

```
_ix1_trav :: forall indexed trav a.
             Index indexed Int (trav a) => Traversable trav =>
               Traversal' indexed a
_ix1_trav = ix 1 <<< traversed
```

… and which then compiles fine.

at <<< ix

Even though an `Index` optic is a `Traversal`, it can't be added directly after an `At` lens. As before, you must interpose a `traversed` to "unwrap" the `Maybe` result:

```
_at1_ix1 = at 1 <<< traversed <<< ix 1
```

That will not compile because of the "type variables are not mentioned" error from the previous section. As in that case, the error message can be renamed and converted to a `Traversal'`:

```
_at1_ix1 :: forall keyed indexed a.
            At keyed Int indexed => Index indexed Int a =>
            Traversal' keyed a
_at1_ix1 = at 1 <<< traversed <<< ix 1
```

5.5 What now?

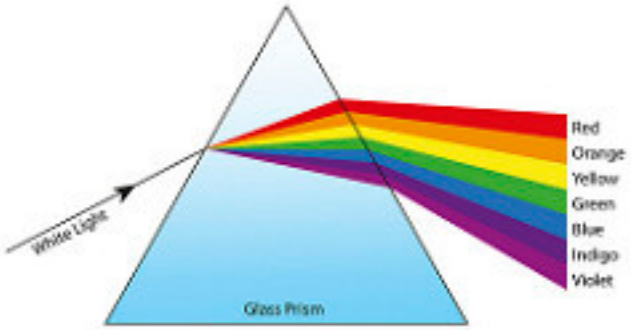
The last of the important lenses is `Prism`, which is used for sum types.

6. Sum types (prisms)

Synopsis

Types: `Prism`, `Prism'`
Constructors: `prism`, `prism'`, `only`, `nearly`
Functions: `preview`, `review`, `is`, `isn't`
Predefined optics: `_Right`, `_Left`, `_Nothing`, `_Just`

Prisms are optics used mainly to focus on one case of a sum type. It was hard for me to remember that `Prism` meant "optic for sum types." This picture might help you:[1]



Think of the distinct colors as corresponding to the distinct cases of a sum type.[2]

You can find sample code and repl imports in src/SumType.purs.

6.1 Using prisms

Consider a sum type used to describe how to fill a shape in a paint program:

[1] Picture credit: Siyavula Education, Creative Commons Attribution 2.0 Generic license.

[2] Pedantic me thinks that, since a `Prism` selects one case out of many, a better metaphor would be a detector placed in one of the color bands. Or maybe "color" itself.

```
data Fill
  = Solid Color
  | LinearGradient Color Color Percent
  | RadialGradient Color Color Point
  | NoFill
```

Suppose you have a lot of code that handles the `Solid` case specially but ignores any other case. Since many useful functions take `Maybe` arguments, it would be handy to convert a `Solid Color` into a `Maybe Color` and convert the `LinearGradient`, `RadialGradient`, and `NoFill` cases to `Nothing`.

With an appropriate prism, that's easy:

```
> preview _solidFill $ Solid Color.white
(Just rgba 255 255 255 1.0) -- this is white

> preview _solidFill NoFill
Nothing
```

With the same prism, you can go in the reverse direction, converting a plain `Color` into a `Solid Color` using a new function, `review`:

```
> review _solidFill Color.white
(Solid rgba 255 255 255 1.0)
```

(Think of `review` as the REverse of preVIEW.)

You can also test whether a particular `Fill` is (or isn't) a `Solid Color`:

```
> is _solidFill (Solid Color.white) :: Boolean
true

> isn't _solidFill (Solid Color.white) :: Boolean
false
```

Note the type annotation `:: Boolean`. That's needed because `is` and `isn't` are declared to produce not a `Boolean`, but an instance of `HeytingAlgebra`:

```
> :t is _solidFill
forall t1. HeytingAlgebra t1 => Fill -> t1
```

When using `is` in a function, you're unlikely to need a specific annotation.

`set` works with prisms, but `review` is simpler:

```
> set _solidFill Color.white $ Solid Color.black
(Solid rgba 255 255 255 1.0)

> review _solidFill Color.white
(Solid rgba 255 255 255 1.0)
```

Since the second argument to `set` has no effect on the result, it seems a pointless argument. That's true in this case, but not when prisms are composed with other optics:

```
> set (_1 <<< _solidFill) Color.white $ Tuple (Solid Color.black) 5
(Tuple (Solid rgba 255 255 255 1.0) 5)
```

6.2 Making a prism

As with the setters and getters that `lens` takes, there are two important functions for creating prisms.

1. A setter-like function applies a value constructor (like `Solid`) to a value. Notice that the setter doesn't set *part* of a whole, but rather creates an entirely new whole. So I'll name it `constructor` in code I write.
2. A getter-like function does a case analysis on a value of a sum type, and distinguishes between cases that should produce a `Just` and those that should produce a `Nothing`. I'll call this function `focuser` in code I write.

There are no fewer than four functions that make prisms.

`prism'` uses `Maybe`

The `prism'` function (note the `'`) expects a focuser that returns a `Maybe`. Here's a verbose example:

```
_solidFill :: Prism' Fill Color
_solidFill = prism' constructor focuser
  where
    constructor = Solid
    focuser fill = case fill of
      Solid color -> Just color
      otherCases -> Nothing
```

This is terser:

```
_solidFill' :: Prism' Fill Color
_solidFill' = prism' Solid case _ of
  Solid color -> Just color
  _ -> Nothing
```

`prism` uses `Either`

The non-apostrophized `prism` asks the focuser to produce an `Either`:

```
_anotherSolidFill :: Prism' Fill Color
_anotherSolidFill = prism Solid case _ of
  Solid color -> Right color
  otherCases -> Left otherCases
```

Despite the use of `Either`, `preview` continues to produce a `Maybe`:

```
> preview _anotherSolidFill (Solid Color.white)
(Just rgba 255 255 255 1.0)

> preview _anotherSolidFill NoFill
Nothing
```

... so the information in a `Left` is lost.

Since `prism` and `prism'` are so similar, just use whichever is more convenient at the moment.

`only` tests for a single value

Suppose you only care about identifying whether a value is `(Solid Color.white)`. That is: you don't want to extract the wrapped `Color`, you just want to ask if it's one particular value:

```
> (is _solidWhite $ Solid Color.black) :: Boolean
false
```

In that case, `only` is more convenient than `prism` or `prism'`:

```
_solidWhite = only (Solid Color.white)
```

The resulting `Prism` has `Unit` as its focus type. While `preview` and `review` are available, they seem pretty useless:

```
> preview _solidWhite (Solid Color.white)
(Just unit)

> review _solidWhite unit
(Solid rgba 255 255 255 1.0)
```

`only` requires that the sum type implement `class Eq`. If it doesn't, see the next constructor. (I derived `Eq` for `Fill` to make these examples work.)

`nearly` doesn't require `Eq`

Just like `only`, `nearly` produces a prism that matches a *single value*. Don't be fooled by the name into thinking it accepts values "close enough" to some reference value. The difference from `only` is that `nearly` uses a predicate you supply instead of forcing the sum type to implement `Eq`.

It happens that the type wrapped by `Solid` (`Color`) does support equality. So we can get the effect of `_solidWhite` without deriving `Eq` for `Fill`:

```
_solidWhite' :: Prism' Fill Unit
_solidWhite' =
  nearly (Solid Color.white) case _ of
    Solid color -> color == Color.white
    _ -> false
```

The first argument to `nearly` allows `review` to obey the prism laws (see below). It probably has no practical value, unless you want to ask `_solidWhite'` what value it matches:

```
> review _solidWhite' unit
(Solid rgba 255 255 255 1.0)
```

Perhaps that seems more useful if you enjoy, as so many do, avoiding meaningful variable names:

```
> review pm unit
(Solid rgba 255 255 255 1.0)
```

6.3 Prism laws

As with other optics, there are laws that govern whether a prism is well-behaved or not. They relate the constructor to the focuser function. They are:

review-preview:
: `preview` retrieves what was given to `review`.

```
review lens >>> preview lens ≡ Just

> Color.white # review _solidFill # preview _solidFill
(Just Color.white)
```

preview-review:
If `preview` retrieves something, `review` can create the original from that something.

```
if  preview prism s ≡ Just a
then review prism a ≡ s

> Solid Color.white # preview _solidFill <#> review _solidFill
(Just (Solid Color.white))
```

Exercise

I want a prism that selects the central `Point` from a `RadialGradient`, ignoring the two colors. Given a radial `Fill` like this one:

```
> fillRadial
(RadialGradient (Color.white) (Color.black) (1.0, 3.4))
```

... the `_centerPoint` prism would produce this:

```
> preview _centerPoint fillRadial
(Just (1.0, 3.4))
```

Why is that an ill-formed prism? (Answer in the next section.)

6.4 Value constructors with more than one argument

The previous exercise asked about a prism with this type:

```
_centerPoint :: Prism' Fill Point
```

The implementation of the focuser is similar to what you've already seen:

```
_centerPoint = prism' constructor focuser
  where
    focuser = case _ of
      RadialGradient _ _ point -> Just point
      _ -> Nothing
```

The constructor is a problem. It must take a `Point` and use the constructor `RadialGradient` to make a `Fill`. But we have only one of the three values `RadialGradiant` requires. We've no option but to use constants for the other two (line 8):

```
1 centerPoint = prism' constructor focuser
2   where
3     focuser = case _ of
4       RadialGradient _ _ point -> Just point
5       _ -> Nothing
6
7     constructor point =
8       RadialGradient Color.black Color.white point
```

But then consider this:

```
> fillRadial # preview centerPoint <#> review centerPoint
(Just (RadialGradient rgba 0 0 0 1.0 rgba 255 255 255 1.0 (1.0, 3.4)))
                           ^^^^^^^^^      ^^^^^^^^^^^^^^^
```

Compare that to the original:

```
> fillRadial
      (RadialGradient rgba 255 255 255 1.0 rgba 0 0 0 1.0 (1.0, 3.4))
                           ^^^^^^^^^^^^^^^      ^^^^^^^^^
```

The preview-review round trip doesn't work.

Because of that, `preview` must deliver all the wrapped values. I'll use a record to contain them:

```
type RadialInterchange =
  { color1 :: Color
  , color2 :: Color
  , center :: Point
  }
```

That gives this `Prism` definition:

```
_centerPoint' :: Prism' Fill RadialInterchange
_centerPoint' = prism constructor focuser
  where
    focuser = case _ of
      RadialGradient color1 color2 center ->
        Right {color1, color2, center}
      otherCases ->
        Left otherCases

    constructor {color1, color2, center} =
      RadialGradient color1 color2 center
```

That definition satisfies the preview-review law:

```
> previewed = preview _centerPoint' fillRadial
> map (review _centerPoint') previewed
(Just (RadialGradient rgba 255 255 255 1.0 rgba 0 0 0 1.0 (1.0, 3.4)))
> fillRadial
      (RadialGradient rgba 255 255 255 1.0 rgba 0 0 0 1.0 (1.0, 3.4))
```

6.5 Types

Raw prism types are pleasingly straightforward:

```
> :t _centerPoint'
forall p.
  Choice p => p
                { color1 :: Color
                , color2 :: Color
                , center :: Point
                }
                { color1 :: Color
                , color2 :: Color
                , center :: Point
                }
              -> p Fill Fill
```

`Choice` is the clue that the profunctor is used to make a `Prism`.

6.6 Composing prisms

You can compose prisms with any other optic. I'll show most of them, leaving `At` lenses and `Index` topics for the composition appendix.

`prism <<< prism`

There are predefined prisms for `Maybe` and `Either`, named `_Just`, `_Nothing`, `_Left`, and `_Right`.

So to focus on a `Solid Color` within a `Right`, we'd do this:

```
> preview (_Right <<< _solidFill) (Right $ Solid Color.white)
(Just rgba 255 255 255 1.0)

> preview (_Right <<< _solidFill) (Left $ Solid Color.white)
Nothing

> preview (_Right <<< _solidFill) (Right NoFill)
Nothing
```

When used to make this compound structure, `review` will use both constructors:

```
> review (_Right <<< _solidFill) Color.white :: Either Fill Fill
(Right (Solid rgba 255 255 255 1.0))
```

... which I think is cool.

The `:type` of this composed optic tags it as another prism:

```
> :t _Right <<< _solidFill
forall t5 t8. Choice t8 =>
              ^^^^^^
  t8 Color Color -> t8 (Either t5 Fill) (Either t5 Fill)
```

... so the type annotation looks like those of other prisms:

```
_right_solidFill :: forall _1_.
                    Prism' (Either _1_ Fill) Color
_right_solidFill = _Right <<< _solidFill
```

`traversed <<< prism`

Adding a `Prism` to a `Traversal` produces a `Traversal`. Here's a sample of the raw type:

```
> :t traversed <<< _solidFill
forall t7 t8.
Traversable t7 => Wander t8 =>
  t8 Color Color -> t8 (t7 Fill) (t7 Fill)
```

There's no `Choice` profunctor, just the `Wander` that's a sign of a `Traversal`. So this is an appropriate type:

```
_traversed_solidFill :: forall trav .
                        Traversable trav =>
                          Traversal' (trav Fill) Color
_traversed_solidFill = traversed <<< _solidFill
```

`preview` works on such a composition, though it behaves as we saw in the `Traversal` chapter. For example, `preview` returns only the `Color` for the first element of this array:

```
> preview _traversed_solidFill [ Solid Color.white, Solid Color.black ]
(Just rgba 255 255 255 1.0)
```

To see all the colors, you use `toListOf`:

```
> toListOf _traversed_solidFill [ Solid Color.white, Solid Color.black ]
(rgba 255 255 255 1.0 : rgba 0 0 0 1.0 : Nil)
```

`review` doesn't work because the composed optic isn't the right type:

```
> review _traversed_solidFill Color.white
Error found:
  No type class instance was found for
    Data.Lens.Internal.Wander.Wander Tagged
```

`prism <<< traversed`

Adding a `Traversal` onto a `Prism` produces an optic with a type like this:

```
> :t _Right <<< traversed
forall t10 t11 t12 t6 t9.
Choice t9 => Traversable t12 => Wander t9 =>
^^^^^^                          ^^^^^^
t9 t11 t10 -> t9 (Either t6 (t12 t11)) (Either t6 (t12 t10))
```

The profunctor is both a `Choice` and a `Wander`. So is it a `Prism` or a `Traversal`? One easy way to check is to use `review`:

```
> :t review (_Right <<< traversed)
Error found:
  No type class instance was found for
    Data.Lens.Internal.Wander.Wander Tagged
```

As an exercise, try to write the correct type.

```
_right_traversed :: forall trav a b _1_ .
                    Traversable trav =>
                    Traversal (Either _1_ (trav a))
                              (Either _1_ (trav b))
                              a b
_right_traversed = _Right <<< traversed
```

It does indeed behave like a `Traversal`:

```
> preview _right_traversed (Right [1, 2, 3])
(Just 1)

> (set _right_traversed 888 (Right [1, 2, 3])) :: Either Int (Array Int)
(Right [888,888,888])

> (set _right_traversed 888 (Left 33)) :: Either Int (Array Int)
(Left 33)
```

`lens <<< prism`

Adding a prism to a lens once again gives a profunctor that implements two type classes:

```
> :t _1 <<< _solidFill
forall t5 t8.
Strong t8 => Choice t8 =>
t8 Color Color -> t8 (Tuple Fill t5) (Tuple Fill t5)
```

Again, `review` doesn't work on it, so it can't be typed as a `Prism`. But `preview` does work with the optic, even though the type doesn't include the `Traversal`'s telltale `Wander` profunctor:

```
> preview (_1 <<< _solidFill) $ Tuple (Solid Color.white) "ignore"
(Just rgba 255 255 255 1.0)

> preview (_1 <<< _solidFill) $ Tuple NoFill "ignore"
Nothing
```

It turns out that it *can* be typed as a `Traversal`:

```
_1_solidFill :: forall _1_ .
                Traversal' (Tuple Fill _1_) Color
_1_solidFill = _1 <<< _solidFill
```

And stranger still, it's a `Traversal` that doesn't demand a `Traversable`!

The mystery of the missing `Wander` is solved when you look at `Wander`'s definition. It turns out that `Wander` is shorthand for a profunctor that's both `Strong` and `Choice`:

```
class (Strong p, Choice p) <= Wander p where ...
```

In the case of this composition, `Strong` comes from one optic and `Choice` from the other, so the shorthand doesn't appear in the composed optic's type.

The result is a `Traversal` that "traverses" a type that has either one or zero values.

Revealed! Why `_Just` exists!

I mentioned earlier that `_Just` and `_Nothing` are predefined prisms.

`Just` seems pretty pointless:

```
> preview _Just (Just 3) -- an identity function
(Just 3)

> review _Just 3 -- a longer value constructor
(Just 3)

> (is _Just (Just 3)) :: Boolean  -- a more verbose `isJust`
true
```

However, remember the trick of using `traversed` to compose a lens (or anything else) onto an `At` lens?

```
at 3 <<< traversed <<< _1
```

That works because `at 3 <<< traversed` converts the `Lens` into a `Traversal` that has no effect other than to allow composition.

But `_Just`, like any `Prism`, can also convert a `Lens` into a `Traversal`. So you could equally well write this:

```
at 3 <<< _Just <<< _1
```

The latter is probably more idiomatic.

`prism <<< lens`

The type of a `Lens` added to a `Prism` is also contains `Choice` and `Strong`:

```
> :t _Left <<< _1
forall t10 t11 t5 t8 t9.
Choice t8 => Strong t8 =>
t8 t11 t10 -> t8 (Either (Tuple t11 t9) t5) (Either (Tuple t10 t9) t5)
```

So the appropriate type is, again, `Traversal`:

```
_left_1 ::forall a b _1_ _2_.
          Traversal (Either (Tuple a _1_) _2_)
                    (Either (Tuple b _1_) _2_)
                    a b
_left_1 = _Left <<< _1
```

6.7 Prisms aren't just for sum types (exercises)

Sum types aren't the only types that you can think of as being composed of mutually-exclusive cases. In this exercise, you'll work with some.

The src/SumTypeStart.purs file will save you from having to decide on `imports` yourself. src/SumTypeSolutions.purs contains my solutions.

1. Let's divide `String` into two cases: one containing a valid integer and one not.[3] Implement an `_intSource` prism that works like this:

   ```
   > preview _intSource "134"
   (Just "134")

   > preview _intSource "a35"
   Nothing
   ```

   Make sure `_intSource` obeys the prism laws.

   Hint: `SumTypeStart` imports `fromString` for you.

2. Using `_intSource` as a template, produce `int`. It produces the `Int` corresponding to a valid string:

[3] I got the idea for this exercise from Elm's Monocle

```
> preview _int "134"
(Just 134)

> preview _int "a35"
Nothing
```

Does it still obey the prism laws?

3. Create a function `_word` such that:

```
> _dawn = _word "Dawn"
> preview _dawn "Dawn"
(Just "Dawn")

> preview _dawn "dawn"
Nothing

> preview _dawn "Brian"
Nothing
```

Can you make it obey the lens laws?

6.8 What now?

There are a few other miscellaneous optics that are probably worth knowing. They're covered in the next chapter.

7. Miscellaneous other optics

Not written yet. I'm not sure whether I should do more with the book. Perhaps what's here now represents the proverbial 80% of the benefit from 20% of the effort. (It'd probably be more like 40% of the effort.)

What do you think? Let me know via marick@exampler.com or @marick on Twitter.

Writing this book has been intellectually fun. But a book that few people know about isn't one I want to spend much effort on. (Fun *and* popular is better than just fun.) I'll be greatly motivated by tweets or, especially, reviews that call the static FP community's attention to this book.

It's a historical accident that the book is written in PureScript. If you want to translate it into Haskell, I'll help.

Thank you for reading this far.

Appendices

A cheat sheet for the optic types

`Lens`

Covered in chapter 1.

Purpose

Used for product types like tuples and records.

Creation and use

```haskell
_field :: Lens' RecordType FieldType
_field = lens _.field $ _ { field = _ }
```

Use `view`, `set`, and `over`.

Laws

- set-get: `view` retrieves what `set` puts in.
- get-set: If you `set` the focus to the value `view` returns, the whole isn't changed.
- set-set: Setting the focus twice to the same value is the same as setting it once.

(More detail on the laws)

`class At`

Covered in chapter 2.

Purpose

`At` lenses are a variant of `Lens`.

Used to work with a single focus in keyed collections like `Map`, where:

- elements can be missing
- new elements can be added
- existing elements can be deleted

Note: if you have such a collection but you want to disallow deletion or new elements, use `Index` traversals.

Creation and use

`At` lenses are created with `at` (which is not re-exported from `Data.Lens`). They differ from plain lenses in that `view` yields a `Maybe a`, and `set` requires a `Maybe a`:

```
import Data.Lens.At

_key :: forall a. Lens' (Map Int a) (Maybe a)
_key' = at "key"
```

- Use `set _key Nothing` to delete an element.
- `set _key (Just x)` will insert a new key/value pair if the focus is `Nothing`.
- `setJust _key x` is short for `set _key (Just x)`.
- Use `map` with `over`: `over _key (map negate)`

`Traversal`

Covered in chapter 4

Purpose

`Traversal` can be used to distribute an operation (like `set`) to all elements of a `Traversable` (collection-like) value. `Traversal` works nicely with empty values. It can be used between two incompatible optics in a composition pipeline.

Use

You don't have to construct a `Traversal`; `traversed` is general-purpose and supplied to you.

```
import Data.Lens

> set traversed 99 [1, 2]
[99,99]

> over traversed negate [1, 2]
[-1,-2]

> toListOf traversed [1, 2, 3]
(1 : 2 : 3 : Nil)
```

`view` appends all the focus elements together. The focus type must be a `Monoid`.

```
> view traversed ["2", "=", "1+1"]
"2=1+1"
```

`Maybe` is a `Traversable`. Because of that, `traversed` can be used to "pipe" the value of a `Just` result from an `At` lens into another lens:

```
> whole = Map.singleton "key" $ Tuple 1 "a"
> set (at "key" <<< traversed <<< _1) 888 whole
(fromFoldable [(Tuple "key" (Tuple 888 "a"))])
```

Using `Traversal` with effects

TBD

`class Index`

Covered in chapter 5

Purpose

An `Index` optic lets you focus on a single element in a collection-like value. The `ix` constructor is equivalent to this:

```
ix index = element index traversed
```

... but the optics it creates are more efficient.

An `Index` optic is similar to an `At` lens, but it's used in cases where the creation of new elements and deletion of old ones is either impossible or undesirable.

Creation and use

The constructor, `ix`, takes a single argument that represents an index (or key) into a structure:

```
import Data.Lens.Index (class Index, ix)

_ix1 :: forall s a . Index s Int a =>
        Traversal' s a
_ix1 = ix 1
```

Because an `Index` optic is a `Traversal`, the focus element is retrieved with `preview`:

```
> preview _ix1 ["a", "b", "c"]
(Just "b")
```

Unlike an `At` lens, `set` doesn't take a `(Maybe focus)`:

```
> set _ix1 "CCCCC" ["a", "b", "c"]
["a","CCCCC","c"]
```

... and the function argument to `over` doesn't have to be lifted with `map`:

```
> over _ix1 String.toUpper ["a", "abc_def"]
["a","ABC_DEF"]
```

Note that both `set` and `over` do nothing if the focus element is missing:

```
> set _ix1 "CCCCC" ["a"]
["a"]

> over _ix1 String.toUpper ["a"]
["a"]
```

`Prism`

Covered in chapter 6.

Purpose

A `Prism` lets you focus on one case of a sum type. It essentially converts the focus case into a `Just` and all the other cases into `Nothing`.

Creation

The two main constructors work with `Maybe` and `Either` types:

```
_solidFill :: Prism' Fill Color
_solidFill = prism' Solid case _ of
  Solid color -> Just color
  _ -> Nothing

_anotherSolidFill :: Prism' Fill Color
_anotherSolidFill = prism Solid case _ of
  Solid color -> Right color
  otherCases -> Left otherCases
```

There's no difference between the prisms produced. In the second prism, note that the value wrapped by `Left` is thrown away.

Two other contructors match a single value:

```
_solidWhite :: Prism' Fill Unit
_solidWhite = only (Solid Color.white)

_solidWhite' :: Prism' Fill Unit
_solidWhite' =
  nearly (Solid Color.white) case _ of
    Solid color -> color == Color.white
    _ -> false
```

The difference is that `only` requires the value be of a type that implements `Eq`, while `nearly` lets you define equality with a custom predicate.

Use

Use `preview` to extract values:

```
> preview _solidFill $ Solid Color.white
(Just rgba 255 255 255 1.0)

> preview _solidFill NoFill
Nothing
```

`is` and `isn't` test whether a value is of the focus case:

```
> is _solidFill (Solid Color.white) :: Boolean
true

> isn't _solidFill (Solid Color.white) :: Boolean
false
```

`set` works with prisms, but `review` is simpler:

```
> set _solidFill Color.white $ Solid Color.black
(Solid rgba 255 255 255 1.0)

> review _solidFill Color.white
(Solid rgba 255 255 255 1.0)
```

`over` works as always.

A catalog of compositions

Note: the indentations are bad on some of the type annotation examples. It's not fixable.

`Lens` `<<<` *optic*

Type appended	Example	Result type	Notes
`Lens`	`_1 <<< _1`	`Lens' (Tuple (Tuple a _1_) _2_) a`	
`Traversal`	`_1 <<< traversed`	`Traversable trav =>` `Traversal' (Tuple (trav a) _1_) a`	
`Prism`	`_1 <<< _Right`	`Traversal' (Tuple (Either _1_ a) _2_)`	

`At` lens	`_1 <<< at 1`	`At keyed Int a =>`
		`Lens' (Tuple keyed _2_) (Maybe a)`
		This is an `At` lens
			1

`Index` optic	`_1 <<< ix 1`	`Index indexed Int a =>` `Traversal' (Tuple indexed _1_) a`	

1. Use `traversed` or `_Just` to convert the resulting `At` lens to a `Traversal`:

```
_1 <<< at 1 <<< traversed
```

`Prism` `<<<` *optic*

Type appended	Example	Result type	Notes

`Lens`	`_Left <<< _1`	`Traversal (Either (Tuple a _1_) _2_)`
		`(Either (Tuple b _1_) _2_)`
		`a b`	

`Traversal`	`_Right <<< traversed`	`Traversable trav =>`
		`Traversal (Either _1_ (trav a))`
		`(Either _1_ (trav b))`
		`a b`	

`Prism`	`_Left <<< _Right`	`Prism' (Either (Either _1_ a) _2_) a`	

Type appended	Example	Result type	Notes
`At` lens			1
`Index` optic	`_Left <<< ix 1`	`Index indexed Int a =>` `Traversal' (Either indexed _1_) a`	

1. They can be combined, but I don't know of a reason why you ever would. You end up with nested `Maybe`s.

`Traversal` <<< *optic*

Type appended	Example	Result type	Notes

`Lens`	`traversed <<< _1`	`Traversable trav =>`
		`Traversal (trav (Tuple a _1_))`
		`(trav (Tuple b _1_))`
		`a b`	

`Traversal`	`traversed <<< traversed`	`Traversable trav1 =>`
		`Traversable trav2 =>`
		`Traversal (trav1 (trav2 a))`
		`(trav1 (trav2 b))`
		`a b`	

`Prism`	`traversed <<< _Left`	`Traversable trav =>` `Traversal (trav (Either a _1_)) a`	

`At` lens	`traversed <<< at 3`	`Traversable trav =>`
		`At keyed Int a =>`
		`Traversal' (trav keyed) (Maybe a)`	

`Index` optic	`traversed <<< ix 1`	`Traversable trav =>`
		`Index indexed Int a =>`
		`Traversal' (trav indexed) a`	

`At` lens <<< *optic*

The only optic that can be added after an `At` lens is a something that turns it into a `Traversal`.

Type appended	Example	Result type	Notes
`Traversal`	`at 1 <<< traversed`	`At keyed Int a =>` `Traversal' keyed a`	1
`Prism`	`at 1 <<< _Just`	`At keyed Int a =>` `Traversal' keyed a`	1

`Index` optic	`at 1 <<<`
	`traversed <<< ix 1`
		`At keyed Int indexed =>`
		`Index indexed Int a =>`
		`Traversal' keyed a`
			2

1. Note that `traversed` and `_Just` have the same effect. `_Just` is probably more idiomatic.
2. Although an `Index` optic is a `Traversal`, you still need to interpose `traversed` to unwrap the `At` lenses `Maybe` result.

`Index` optic `<<<` *optic*

Type appended	Example	Result type	Notes
`Lens`	`ix 1 <<< _1`	`Index indexed Int (Tuple a _1_) =>` `Traversal' indexed a`	

`Traversal`	`ix 1 <<< traversed`	`Index indexed Int (trav a) =>`
		`Traversable trav =>`
		`Traversal' indexed a`	

`Prism`	`ix 1 <<< _Left`	`Index indexed Int (Either a _1_) =>` `Traversal' indexed a`	

`At` lens	`ix 1 <<< at 3`	`Index indexed Int keyed =>`
		`At keyed Int a =>`
		`Traversal' indexed (Maybe a)`	

`Index` optic	`ix 1 <<< ix 1`	`Index indexed1 Int indexed2 =>`
		`Index indexed2 Int a =>`
		`Traversal' indexed1 a`	

Identifying an optic from type spewage

I see `Choice`

The `Choice` profunctor signals a `Prism`. This:

```
forall p. Choice p => p Color Color -> p Fill Fill
```

... can also be written like this:

```
Prism' Fill Color
```

I see `Strong` alone

A pattern like the following identifies a `(Lens s t a b)`:

```
forall p a b s t .
  Strong p =>
    p a b -> p s t
```

I see `Strong` and `Choice`

A profunctor that's both `Strong` and `Choice` is usually labelled as with `Wander`, but sometimes not. See "`Wander` alone" below.

I see `Strong` and `At`

If a type with a `Strong` constraint also includes a constraint on `(At whole key part)`, it's an `At` lens:

```
forall t1 t3.
  At t1 Int t3 => (forall p. Strong p =>
  ^^^^^^^^^^^^                ^^^^^^^^
    p (Maybe t3) (Maybe t3) -> p t1 t1)
```

I see `Index` and `Wander`

These two type classes signal an `Index` optic (a special kind of `Traversal` that refers to a single focus element in a `Traversable`):

```
forall t10 t11 t5 t8.
Index t5 Int (t11 t10) => Wander t8 => Traversable t11 =>
^^^^^                     ^^^^^^
t8 t10 t10 -> t8 t5 t5
```

I see `Wander` alone

`Wander` is the profunctor used by `Traversal`.

Here's the annotation for `traversed`:

```
> :t traversed
forall t a b. Traversable t =>
  (forall p. Wander p => p a b -> p (t a) (t b))
```

Notice the constraint that `traversed` is used with a `Traversable` type. Since the type will often be narrowed to something like `Array` or `Maybe`, `Traversable` might not appear but `Wander` will.

Here's the type of a simple composition:

```
> :t _1 <<< traversed
forall t12 t15 t16 t17 t18.
  Strong t15 => Traversable t18 => Wander t15 =>
    t15 t17 t16 ->
    t15 (Tuple (t18 t17) t12) (Tuple (t18 t16) t12)
```

The profunctor is `t15`. Notice that it's both `Strong` and `Wander`. Since any `Wander` profunctor is also `Strong`, the explicit mention of `Strong` is redundant. It has no effect on any type annotation you'd write:

```
_1_trav :: forall trav a b _1_ .
           Traversable trav =>
             Traversal (Tuple (trav a) _1_)
                       (Tuple (trav b) _1_)
                       a b
_1_trav = _1 <<< traversed
```

Composition in the reverse order produces a raw type without the `Strong`:

```
:t traversed <<< _1
forall t14 t15 t16 t17 t18.
  Traversable t14 => Wander t15 =>
    t15 t18 t17 ->
    t15 (t14 (Tuple t18 t16)) (t14 (Tuple t17 t16))
```

Since the `Strong` doesn't matter, a type annotation would have the same "shape":

```
_trav_1 :: forall trav a b _1_.
           Traversable trav =>
            Traversal (trav (Tuple a _1_))
                      (trav (Tuple b _1_))
                      a b
_trav_1 = traversed <<< _1
```

The only difference between these two annotations is a little shuffling of words.