Lenses from the ground up

David Peterson

What is a Lens?

Good question!

- A pure functional approach to manipulating both the content and structure of (often) deeply-nested data structures.

Why use a Lens?

Another good question!

— Provides a powerful mechanism for manipulating data structures, and composing these manipulations to perform higher-order operations.

Let's take a step back!

— The focus of this talk will be on exploring some *very basic* lens(-like) operations.

— Our goal is to gain an intuitive understanding of what lenses *are*, and how they work.

— The lenses we define here are not quite the same as *real* lenses. They are much simpler, and much less powerful.

Tuples

Let's start with something simple: *tuples*.

A *tuple* is just two associated values wrapped up, so that you can treat it as one *thing*.

```
> (1,2)
(1,2)

> (,) 42 42
(42,42)

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

Tuples

Let's define `get` and `set` functions for the *first* element of the tuple, i.e. here: (**1**,2).

```
-- retrieve the value x
get1 :: (x, y) -> x
get1 (x, _) = x

 -- replace the value x with x'
set1 :: x' -> (x, y) -> (x', y)
set1 x' (_, y) = (x', y)
```

Tuples

```
> get1 (1,2)
1

> set1 10 (1,2)
(10,2)
```

Tuples

Similarly we can define `get` and `set` functions for the *second* element of the tuple, i.e. (1,**2**)

```
get2 :: (x, y) -> y
get2 = snd

set2 :: y -> (x, y) -> (x, y)
set2 y' (x, _) = (x, y')
```

Tuples

```
> get2 (1,2)
2

> set2 0 (1,2)
(1,0)
```

None of this has anything to do with *Lens* ... it's just vanilla *pattern matching*.

Defining a Lens

Using standard record syntax, define a type constructor called `Lens`

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

To create a *Lens*, you need to pass two functions, one of type `a -> b`, and the other of type `b -> a -> a`.

Defining a Lens

Hey, we already have some!

```
get1 :: (x, y) -> x
set1 :: x -> (x, y) -> (x, y)
```

Here the tuple type `(x, y)` corresponds to `a`, and `x` corresponds to `b` (in `Lens a b`).

In Lens terminology, we call `a` the *object*, and we call `b` the *focus*.

Defining a Lens

So, let's make a *lens* ...

```
_1 = Lens get1 set1
```

Defining a Lens

Recall from Haskell record syntax, we automatically get *helper methods* for each named record field.

```
> :t get
Lens a b -> a -> b
--            ^^^^^^
--               |-- this is the signature of get1

> :t set
Lens a b -> b -> a -> a
--            ^^^^^^^^^^^
--                 |--  this is the signature of set1
```

Using a lens

Before, we had

```
> get1 (1,2)
1
```

Now with our lens, we have

```
> get _1 (1,2)
1
```

Using a lens

Similarly, before we had

```
> set1 5 (1,2)
(5,2)
```

Now with our lens, we have

```
> set _1 5 (1,2)
(5,2)
```

Using a lens

We have decoupled the `set` and `get` operations from the specific location on which they operate.

Intuitively, this sounds like a good thing, right?

I think so.

Using a lens

Let's make another lens ...

```
_2 = Lens get2 set2
```

Alternatively (and perhaps more typically), we can use *anonymous* functions rather than named ones.

```
_2 = Lens (\(_, y) -> y) (\y (x, _) -> (x, y))
```

Using a lens

```
> get _2 (1,2)
2

> set _2 5 (1,2)
(1,5)
```

Is that all?

NO!

Lens composition

Lenses *compose*!

```haskell
-- Define a composition operator (>-) ...

(>-) :: Lens a b -> Lens b c -> Lens a c

(>-) l1 l2 = Lens (get l2 . get l1) $
                  (\part whole -> set l1 (
                      set l2 part (get l1 whole)) whole)
```

Admittedly, composing `set` is a little funky!

Lens composition

Let's do it!

```
_1_1 = _1 >- _1

_1_2 = _1 >- _2


-- Looking at the types can be helpful!

> :t _1_1
_1_1 :: Lens ((c, b1), b) c
--            ^^^        ^^^

> :t _1_2
_1_2 :: Lens ((a, c), b) c
--               ^^^    ^^^
```

Lens composition

```
> get _1_1 ((1,2),(3,4))
1

> get _1_2 ((1,2),(3,4))
2

> set _1_1 5 ((1,2),(3,4))
((5,2),(3,4))

> set _1_2 5 ((1,2),(3,4))
((1,5),(3,4))
```

Is that all?

NO!

Shortcut operators

```
(.~) :: Lens a b -> b -> a -> a
(.~) = set
infixr 4 .~

(^.) :: a -> Lens a b -> b
(^.) = flip get
infixl 8 ^.
```

Shortcut operators

```
-- Get

> (1,2) ^. _1
1

> (1,2) ^. _2
2


-- Set

> _1 .~ 4 $ (1,2)
(4,2)

> _2 .~ 5 $ (1,3)
> (1,5)
```

Over

`over` is like `fmap` for a lens

```
over :: Lens a b -> (b -> b) -> a -> a
over l f a = set l (f (get l a)) a
(%~) = over
infixr 4 %~
```

```
> _1 %~ (*2) $ (3,2)
(6,2)

> _2 %~ (*2) $ (3,2)
(3,4)
```

Lens Laws

execrabilis ista turba, quae non novit legem

— Francis Bacon

Lens Laws

Three of them.

1. Get-Set Law
2. Set-Get Law
3. Set-Set Law

Get-Set Law

```haskell
get_set_law :: Eq a => Lens a b -> a -> Bool
get_set_law l =
  \a ->
    set l (get l a) a == a
```

Doing a `set` using a value obtained from a `get` is equivalent to doing *nothing at all*.

Get-Set Law

```
> get_set_law _1 (3,2)
True
> get_set_law _1 (99,23)
True
> get_set_law _1 (1,2)
True
```

Basically equivalent to

```
> let v = (1,2) ^. _1 -- get
> _1 .~ v $ (1,2) -- set
(1,2)
```

Set-Get Law

```
set_get_law :: Eq b => Lens a b -> b -> a -> Bool
set_get_law l =
  \s a ->
    get l (set l s a) == s
```

Doing a `set` operation followed by a `get` operation returns the value that was set.

Set-Get Law

```
> set_get_law _1 5 (1,2)
True
> set_get_law _1 5 (3,2)
True
> set_get_law _1 5 (2,1)
True
```

Set-Set Law

```
set_set_law :: Eq a => Lens a b -> b -> b -> a -> Bool
set_set_law l =
  \s1 s2 a ->
    set l s2 (set l s1 a) == set l s2 a
--  ^^^^^^^^^ ^^^^^^^^^^^^
--      |           |- first (inner) set operation (s1)
--      |- second (outer) set operation (s2)
```

If perform one set operation (s1) followed by a second set operation (s2), only the result of the second operation is preserved.

Set-Set Law

```
> set_set_law _1 12 24 (1,2)
True
```

Basically equivalent to

```
-- First set operation (s1)
> _1 .~ 12 $ (1,2)
(12,2)

-- Second set operation (s2)
> _1 .~ 24 $ (12,2)
(24,2)

-- (12,2) is gone!
```

Is that all?

Cue laughter!

We're just getting started.

References / Next Steps

1. David Peterson, Lets.TupleLens (code for this talk!)
2. Tony Morris, Let's Lens (a whole lens course!)
3. Gabriel Gonzales, Control.Lens.Tutorial (on hackage)
4. Joseph Abrahamson, A Little Lens Starter Tutorial