Putting Lenses to Work

John Wiegley

8 Apr 2017

Overview

Introduction

- Practical use of lenses, inspired by work
- Applied to the right problem, they are invaluable!

Lens

Lenses address some part of a "structure" that always exists

Tuple

view (^.)

```
(1,2,3) ^. _2
  ==> 2
```

view

```
view _2 (1,2,3)
  ==> 2
```

Tuple

set (.~)

```
(1,2,3) & _2 .~ 20
  ==> (1,20,3)
```

set

```
set _2 20 (1,2,3)
  ==> (1,20,3)
```

Records

- Possibly the least interesting use of `lens`
- For shallow use, barely different from access and update syntax
- "Distinguished products"

Records

```haskell
{-# LANGUAGE TemplateHaskell #-}

module Lenses where

import Control.Lens

data Record = Record
  { _field1 :: Int
  , _field2 :: Int
  }
makeLenses ''Record
```

Records (Classy)

`makeClassy` is an alternate lens builder that defines lenses as methods of a typeclass, making your record an instance of that class

Records (Classy)

```haskell
{-# LANGUAGE TemplateHaskell #-}

module Lenses (Record, HasRecord(..)) where

import Control.Lens

data Record = Record
  { _field1 :: Int
  , _field2 :: Int
  }
makeClassy ''Record
```

Records (Classy)

```haskell
class HasRecord r where
  record :: Lens' r Record
  field1 :: Lens' r Int
  field2 :: Lens' r Int
  {-# MINIMAL record #-}

instance HasRecord Record where
  ...
  field1 f (Record x y) = ...
  ...
```

Records

view

```haskell
Record 20 30 ^. field1
  ==> 20
```

set

```haskell
Record 20 30 & field1 .~ 1
  ==> Record 1 30
```

Records

Record lenses become quite useful when structure is deep

Records

With lens

```
v & foo.bar.baz +~ 1
```

Records

Without lens

```
let f = _foo v
    b = _bar f
    z = _baz b in
v { _foo = f {
      _bar = b {
        _baz = z + 1 } } }
```

Writing lenses by hand

```haskell
my_1 :: Lens' (Integer, Integer) Int
my_1 f (p1, p2) =
  (\n -> (toInteger n, p2))
    <$> f (fromIntegral p1)

my_1 :: Functor f
     => (Int -> f Int)
     -> (Integer, Integer)
     -> f (Integer, Integer)
```

Common operators

view	`v ^. l`
set	`v & l .~ x`
(set Just)	`v & l ?~ mx`
(incr)	`v & l +~ n`
(append)	`v & l <>~ x`
(apply)	`v & l %~ f`
(applyA)	`v & l %%~ f`

Prism

Prims address some part of a "structure" that may exist

ADTs

```haskell
{-# LANGUAGE TemplateHaskell #-}

module Lenses where

import Control.Lens

data ADT = Alpha Int Int
         | Beta Record
         | Gamma String

makePrisms ''ADT
```

ADTs

view (present)

```
Alpha 10 20 ^? _Alpha._2
  ==> Just 20
```

view (absent)

```
Gamma "hello" ^? _Alpha._2
  ==> Nothing
```

ADTs

set (present)

```
Alpha 10 20 & _Alpha._2 .~ 2
  ==> Alpha 10 2
```

set (absent)

```
Alpha 10 20 & _Beta.field1 .~ 2
  ==> Alpha 10 20
```

ADTs

With lens

```
v & _Beta.field1 +~ 1
```

ADTs

Without lens

```
case v of
  Beta z ->
    Beta (z { _field1 = _field1 z + 1 })
  _ -> v
```

Writing prisms by hand

```haskell
my_Left :: Prism' (Either Int Int) Int
my_Left = prism' Left $
  either Just (const Nothing)
```

Traversals

Traversals address many parts of a "structure" that may exist

Collections

preview

```
[1,2,3] ^? ix 1
  ==> Just 2
```

set

```
[1,2,3] & ix 1 .~ 20
  ==> [1,20,3]
```

Computations

preview

```
31415926 ^? digits.ix 2
   ==> Just 4
```

set

```
31415926 & digits.ix 2 .~ 8
   ==> 31815926
```

Computations

```haskell
digits :: Iso' Int [Int]
digits =
  iso (map (read :: String -> Int)
            . sequence . (:[]) . show)
        ((read :: String -> Int)
            . concatMap show)
```

Computations

set (flexible)

```
31415926 & digits.ix 2 .~ 99
  ==> 319915926
```

Monoids

- "Viewing" a traversal combines the elements using `Monoid`
- `^..` turns each element into a singleton list, so the `Monoid` result is a list of the elements

Monoids

A list of elements

```
[1,2,3,4] ^.. traverse
  ==> [1,2,3,4]
```

Using a Monoid

```
[1,2,3,4] ^. traverse.to Sum
  ==> Sum 10
```

Folds

allOf

```
allOf (traverse._2) even
  [(1, 10), (2, 12)]
  ==> True
```

Folds

allOf	andOf	anyOf
asumOf	concatMapOf	concatOf
elemOf	findMOf	findOf
firstOf	foldMapOf	foldOf
foldl1Of	foldl1Of’	foldlMOf
foldlOf	foldlOf’	foldr1Of
foldr1Of’	foldrMOf	foldrOf

More Folds

```
foldrOf'       forMOf_        forOf_
lastOf         lengthOf       lookupOf
mapMOf_        maximumByOf    maximumOf
minimumByOf    minimumOf      msumOf
noneOf         notElemOf      notNullOf
nullOf         orOf           productOf
sequenceAOf_   sequenceOf_    sumOf
toListOf       traverseOf_
```

Vocabulary review

Class	Read	Write	Count	Example
Getter	y		1	`to f`
Lens	y	y	1	`_1`
Iso	y	y	1	`lazy`
Prism	y?	y?	1?	`only`
Fold	y?		0*	`folded`
Setter		y?	0*	`mapped`
Traversal	y?	y?	0*	`traverse`

Common operators

toListOf	`v ^.. l`
preview	`v ^? l`
(demand)	`v ^?! l`

Map

at (present)

```
alist [(1,"x"), (2,"y")] ^. at 1
  ==> Just "x"
```

at (absent)

```
alist [(1,"x"), (2,"y")] ^. at 1
  ==> Just "x"
```

Map

non (present)

```
alist [(1,"x"), (2,"y")]
  ^. at 2.non "z"
  ==> "y"
```

non (absent)

```
alist [(1,"x"), (2,"y")]
  ^. at 3.non "z"
  ==> "z"
```

Map

ix view (present)

```
alist [(1,"x"), (2,"y")] ^? ix 1
  ==> Just "x"
```

ix view (absent)

```
alist [(1,"x"), (2,"y")] ^? ix 3
  ==> Nothing
```

Map

ix view (demand)

```
alist [(1,"x"), (2,"y")] ^?! ix 1
  ==> "x"
```

Map

ix set (present)

```
alist [(1,"x"), (2,"y")] & ix 1 .~ "z"
  ==> alist [(1,"z"), (2,"y")]
```

ix set (absent)

```
alist [(1,"x"), (2,"y")] & ix 3 .~ "z"
  ==> alist [(1,"x"), (2,"y")]
```

Map

failing

```
alist [(1,"x"), (2,"y")] ^? ix 1
  ==> Just "x"
```

State

use

```
use _1 `evalState` (10, 20)
  ==> 10
```

uses

```
uses _1 negate `evalState` (10, 20)
  ==> -10
```

State

preuse

```
preuse (ix 1) `evalState` [1, 2, 3, 4]
  ==> Just 2
```

preuses

```
preuses (ix 1) negate
  `evalState` [1, 2, 3, 4]
  ==> Just (-2)
```

State

set

```
(ix 1 .= 5) `execState` [1, 2, 3, 4]
  ==> [1, 5, 3, 4]
```

set (monadic)

```
(ix 1 <~ pure 5) `execState` [1, 2, 3, 4]
  ==> [1, 5, 3, 4]
```

State

over

```
(ix 1 %= negate)
  `execState` [1, 2, 3, 4]
  ==> [1, -2, 3, 4]
```

State

zoom

```
zoom _1 (_2 .= 4) `execState` ((1, 2), 3)
  ==> ((1, 4), 3)
```

Lens

multi-set

```
((1,2,3) & _1 .~ 10
         & _2 .~ 20
         & _3 .~ 30)
  ==> (10,20,30)
```

Lens

stateful multi-set

```
((1,2,3) &~ do _1 .= 10
               _2 .= 20
               _3 .= 30)
  ==> (10,20,30)
```

We didn’t cover…

`ALens`	`LensLike`	`Writer`
`lens-action`	`lens-aeson`	`thyme`
Indexed lenses	Zippers	Exceptions
Arrays	Vectors	`FilePath`
`Numeric.Lens`		

partsOf

indices

```
[2,4,1,5,3,6]
  & partsOf (traversed.indices odd)
  %~ reverse
  ==> [2,6,1,5,3,4]
```

partsOf

filtered

```
[2,4,1,5,3,6]
  & partsOf (traverse.filtered (< 4))
  %~ reverse.sort
  ==> [3,4,2,5,1,6]
```

partsOf

each

```
(3,1,2,4,5) & partsOf each %~ sort
  ==> (1,2,3,4,5)
```

partsOf

set

```
"Hello, World"
  & partsOf (traverse.filtered isAlpha)
  .~ "Howdy!!!"
  ==> "Howdy, !!!ld"
```

partsOf

multiple

```
"00:00:00"
  & partsOf (itraversed.
             indices (>= 3).
             filtered (== '0'))
  .~ "1234"
  ==> "00:12:34"
```

ViewPatterns

lambda

```
(\(view _2 -> Left x) -> x) (10, Left 20)
  ==> 20
```

biplate

strings

```
((("foo", "bar"), "!", 2 :: Int, ()))
   ^.. biplate :: [String])
  ==> ["foo","bar","!"]
```

biplate

ints

```
((("foo", "bar"), "!", 2 :: Int, ())
    ^.. biplate :: [Int])
  ==> [2]
```

biplate

chars

```
((("foo", "bar"), "!", 2 :: Int, ())
   & biplate %~ toUpper)
  ==> (("FOO","BAR"),"!",2,())
```

biplate

with partsOf

```
(("foo","bar"),"!", 2 :: Int, ())
  & partsOf biplate
  %~ (reverse :: String -> String)
  ==> (("!ra","boo"),"f",2,())
```

biplate

filtered

```haskell
(("foo","bar"),"!", 2 :: Int, ())
  & partsOf (biplate.filtered (<= 'm'))
  %~ (reverse :: String -> String)
  ==> (("!oo","abr"),"f",2,())
```

biplate

head

```
((("foo","bar"),"!", 2 :: Int, ())
    & (biplate
          :: Data s
          => Traversal' s String)._head
    %~ toUpper)
  ==> (("Foo","Bar"),"!",2,())
```