Profunctor Optics

Matthew Pickering

April 21st 2016

Introduction

- What is an optic?
- Van Laarhoven's Representation
- Profunctor Optics

What is an optic?

- A method of accessing part of a data structure

The Lens Hierachy

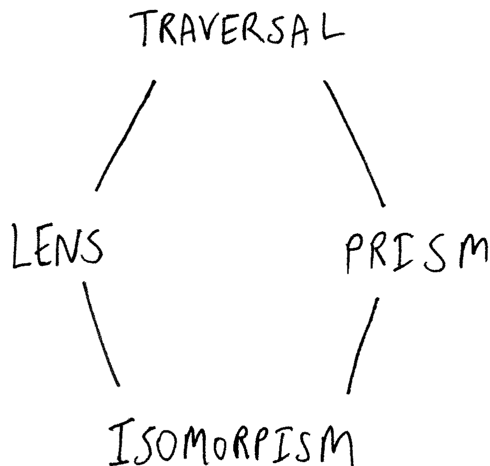

Figure 1:Simplified Lens Hierachy

Lens

```
Lens s a
```

- The *getter*: From `s` we can get out an `a`
- The *setter*: With an `s` and an `a` we can change `a` inside `s`

Examples

- The *getter*: From `s` we can get out an `a`
- The *setter*: With an `s` and an `a` we can change `a` inside `s`

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

```
_1 :: Lens (a, b) a
_2 :: Lens (a, b) b
```

Prism

```
Prism s a
```

- The *matcher*: With an `a` we can build an `s`.
- The *builder*: With an `s` we might be able to get a `a`.

Examples

- With an a we can build an s.
- With an s we might be able to get a a.

```
data Prism s a = Prism { match :: s -> Maybe a
                       , build :: a -> s }
```

```
_Just :: Prism (Maybe a) a
_Left :: Prism (Either a b) a
```

Traversal

```
Traversal s a
```

- A multi-target lens
- In a `s` we can access zero of more `a`s.

```
listTraverse :: Traversal [a] a

treeTraverse :: Traversal (Tree a) a
```

Isomorphism

```
Iso s a
```

- We can turn an `s` into an `a` and an `a` into an `s`.

Examples

- We can turn an s into an a and an a into an s.

```
data Iso s a = Iso { from :: s -> a
                   , to :: a -> s }

curry :: Iso ((a, b) -> c) (a -> b -> c)
assoc :: Iso ((a, b), c) (a, (b, c))
```

The Lens Hierachy

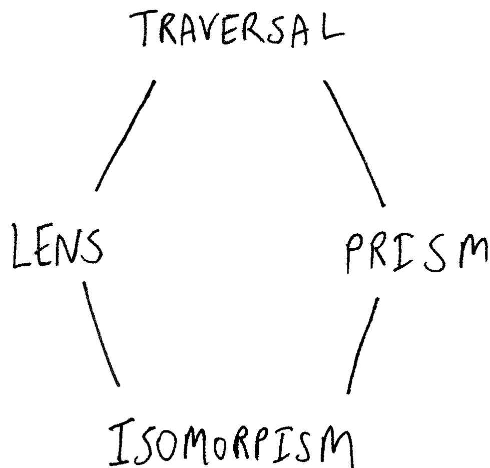

Figure 2:Simplified Lens Hierachy

Full Hierachy

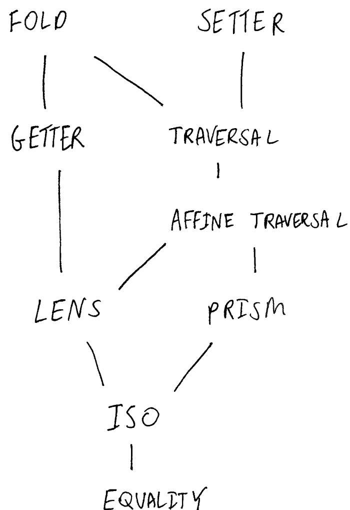

Figure 3:Complete Lens Hierachy

Van Laarhoven's Representation

- A representation in Haskell
- Subtyping
- Easy to compose different optics together

The Lens Hierachy

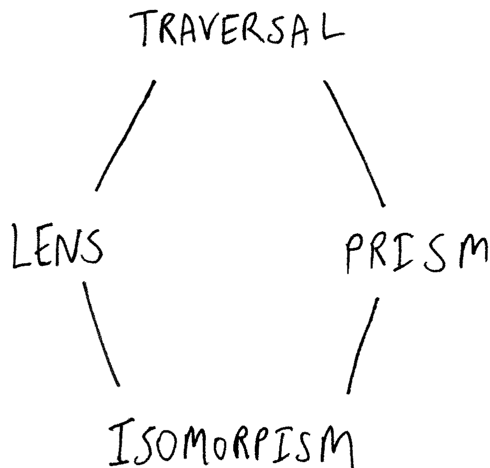

Figure 4:Simplified Lens Hierachy

Lenses

```
type LensVL s a
  = forall f . Functor f => (a -> f a) -> s -> f s
```

- This type is *isomorphic* to Lens

Recovering Lens

```
fromLensVL :: LensVL s a -> Lens s a
fromLensVL l =
  Lens { get = getConst . (l Const)
       , set = \s a ->
           runIdentity (l (const (Identity a) s) }

data Const a b = Const a
data Identity a = Identity a

get - (a -> Const a a) -> s -> Const a s

set - (a -> Identity a) -> s -> Identity s
```

- By choosing different `Functors` we get back the original definitions
- See (O'Connor 2011) and (Jaskelioff 2014)

Programming with Lenses

```
> getVL (_1 . _2) (("a", "b"), "c")
"b"
> setVL (_1 . _1) "d" (("a","b"), "c")
(("d", "b"), "c")
```

- NOTE: Normal Function Composition
- Note that the composition is *backwards* compared to normal function composition.
- Our accessors are *composable* in a way that normal record accessors are not.

Traversals

```
type TraversalVL s a
  = forall f . Applicative f => (a -> f a) -> s -> f s

class Functor f => Applicative f where ...
```

- `Applicative` is a subclass of `Functor`
- Natural generalisation of lenses.

Implementing Traversals

- Almost all traversals are just `traverse` from `Traversable`.
- Very well-understood idiom.

```
class Traversable t where
  traverse :: (a -> f b) -> t a -> f (t b)
```

- `TraversalVL` is a generalistion of the type of `traverse`.

Composing together lenses and traversals

```
_1 :: Lens (a, b) a

listTraverse :: Traversal [a] a
listTraverse = traverse

_1 . listTraverse :: Traversal ([a], b) a

listTraverse . _1 :: Traversal [(a, b)] a
```

Prisms

```
type PrismVL s a
  = forall p f. (Choice p, Applicative f)
    => p a (f a) -> p s (f s)
```

- Unecessary `f` constraint but needed to unify with the rest of the framework.
- Unclear relationship with `Lens`.

Isomorphism

```
type IsoVL s a
  = forall p f . (Profunctor p, Functor f)
    => p a (f a) -> p s (f s)
```

- Again, the `f` is not used but needed to unify.

Problems with the van Laarhoven representation

- Doesn't make obvious the relationship between the different parts of the framework.
- No obvious recipe to add new kinds of optics
- `profunctors` is not a base package and has a lot of dependencies.
-
  - People don't define `Prisms` and `Isos` because they don't want to incur these dependencies.

Profunctor Optics

- A generalisation of the van Laarhoven representation.
- A *recipe* to define new optics
- Some relationships are more obvious.

Profunctors

```
class Profunctor p where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
```

- A bifunctor which is

1. Contravariant in the first parameter
2. Covariant in the second parameter

- Intuitively, a profunctor `p` consumes `as` and produces `bs`.

```
instance Profunctor (->) where
  dimap f h g = h . g . f
```

- Folds
- Automata Libraries
- Database Libraries (`opaleye`)

Lifting Functors into Profunctors

```
data SubStar f a b = SubStar { runSubStar :: a -> f b }

instance Profunctor (SubStar f) where
  dimap f g (SubStar bfc) = SubStar (fmap g . bfc . f)

data UpStar f a b = UpStar { runUpStar :: f a -> b }

instance Profunctor (UpStar f) where
  dimap f g (UpStar fbc) = UpStar (g . fbc . fmap f)
```

The Lens Hierachy

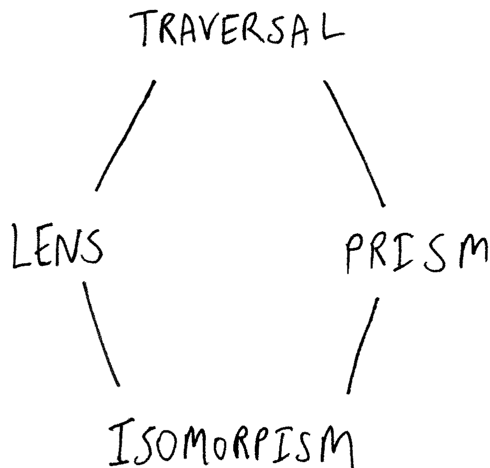

Figure 5:Simplified Lens Hierachy

Isomorphisms

```
type IsoP s a
  = forall p . Profunctor p => p a a -> p s s
```

- Possible to prove that this is isomorphic to the concrete representation of `Iso`.

Lenses

```
class Profunctor p => Strong p where
  _1 :: p a b -> p (a, c) (a, c)
```

- Nice operational interpretation.

```
type LensP s a = forall p . Strong p => p a a -> p s s

-- (a -> b) -> (a, c) -> (b, c)
instance Strong (->) where ...

-- Functor f => (a -> f b) -> (a, c) -> f (b, c)
instance Functor f => Strong (SubStar f) where ...
```

Recovering the van Laarhoven representation

- Choose `p = SubStar f`

```
p a a -> p s s
=~
Functor f => (a -> f a) -> s -> f s
```

- Can then choose `f` appropiately to recover `get` and `set`.

Prisms

```
class Profunctor p => Choice p where
  left' :: p a b -> p (Either a c) (Either b c)

-- (a -> b) -> Either a c -> Either b c
instance Choice (->) where ...

-- Functor f => (f a -> b) -> f (Either a c) -> Either b c
instance Phantom f => Choice (UpStar f) where ...

type Prism s a = forall p . Choice p => p a a -> p s s
```

Traversals

- Find a concrete representation and create a typeclass for it following the pattern.

Concrete Representation

- Need to find a normal form for functions of type `Applicative f => (a -> f b) -> f t`
- Remember the normal form for values built using the applicative interface..

1. `pure :: forall x . x -> f x`
2. `<*> :: forall x y . f (x -> y) -> f x -> f y`

```
pure f <*> x_1 <*> ... <*> x_n
```

- With the addition of `g :: a -> f b` we get the normal form

```
pure f <*> g a_1 <*> ... <*> g a_n
```

FunList

- This concrete representation is known as FunList by van Laarhoven.
- Also see (Bird et al. 2013)

```
data FunList a b t where
  Done :: t -> FunList a b t
  More :: a -> FunList a b (b -> t) -> FunList a b t
```

- `FunList` is `Traversable` in the first argument
- `FunList` is `Applicative` in the last argument

The Class

```
class (Strong p, Choice p) => Wander p where
  wander :: p a b -> p (FunList a c t) (FunList b c t)

instance Wander (->) where ...
instance Applicative f => Wander (SubStar f) where...
```