A Modern History of Lenses

Tony Morris

NICTA

November 25, 2014

Goals

- the motivation for lenses
- the definition of and nomenclature for lenses
- the problems encountered for lenses
- the proposed solutions and recent developments
- the current solution

Goals

- the motivation for lenses
- the definition of and nomenclature for lenses
- the problems encountered for lenses
- the proposed solutions and recent developments
- the current solution

Goals

- the motivation for lenses
- the definition of and nomenclature for lenses
- the problems encountered for lenses
- the proposed solutions and recent developments
- the current solution

Goals

- the motivation for lenses
- the definition of and nomenclature for lenses
- the problems encountered for lenses
- the proposed solutions and recent developments
- the current solution

Goals

- the motivation for lenses
- the definition of and nomenclature for lenses
- the problems encountered for lenses
- the proposed solutions and recent developments
- the current solution

Why Lenses?

the premise

We want to do programming

and anything but functional programming is **completely insane**.

Why Lenses?

If you accept that fact of matter

then you also accept that data types must be *immutable*.

Why Lenses?

OK let's try that

```
data Street =
  Street {
    name :: String
-- , ...
  }
```

Why Lenses?

```
data Employee =
  Employee {
    company :: Company
--  , ...
  }

data Company =
  Company {
    address :: Address
--  , ...
  }

data Address =
  Address {
    street :: Street
--  , ...
  }
```

Why Lenses?

Then your team leader says to you

> *Please set employer's street address to upper-case.*

Why Lenses?

ARGH!

```
upperStreetFirst ::
  Employee
  -> Employee
upperStreetFirst e =
  e {
    company = (company e) {
      address = (address (company e)) {
        street = (street (address (company e))) {
          name = map toUpper
            (name (street (address (company e))))
        }
      }
    }
  }
```

Why Lenses?

Scala insists on repeating history's mistakes

```
def upperStreetFirst(e: Employee): Employee =
  e.copy(company = e.company.copy(
    address = e.company.address.copy(
      street = e.company.address.street.copy(
        name = e.company.address.street.name.
          map(_.toUpper)
      )
    )
  )
```

Why Lenses?

We must subsume dysfunctional programming

because crushing victory is the best kind.

```
(company.address.street.name %= toUpper) e
```

We need lenses.

Why Lenses?

We must subsume dysfunctional programming

because crushing victory is the best kind.

```
(company.address.street.name %= toUpper) e
```

We need lenses.

What is a Lens?

Lens is a data structure

```haskell
data Lens target field =
  Lens {
    get :: target -> field
  , set :: target -> field -> target
  }
```

What is a Lens?

With three laws

- `get lens (set lens t f) == f`
- `set lens (get lens t) t == t`
- `set lens (set lens t f) f’ == set lens t f’`

What is a Lens?

For example

Formerly

```
company :: Employee -> Company
address :: Company -> Address
street :: Address -> Street
name :: Street -> String
```

What is a Lens?

For example

Becomes

```
company :: Employee `Lens` Company
address :: Company `Lens` Address
street :: Address `Lens` Street
name :: Street `Lens` String
```

What is a Lens?

Lenses do lots of interesting things

Lenses can compose to a new Lens

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

company :: Employee `Lens` Company
address :: Company `Lens` Address
company.address  :: Employee `Lens` Address
```

What is a Lens?

Lens comes in a small variety of formulations

```
data Lens target field =
  Lens {
    getset :: target -> (field -> target, field)
  }
```

What is a Lens?

Twan van Laarhoven lens

```haskell
data Lens target field =
  Lens {
    run :: forall f. Functor f =>
      (field -> f field) -> (target -> f target)
  }
```

What is a Lens?

We can derive functions from Lens

```
-- modify the current field of a target
(%=) :: Lens target field -> (f -> f) -> t -> t
Lens g s %= k =
  s <*> k . g
```

What is a Lens?

At this point, subsumption is achieved

We can do at least as well as dysfunctional programming

```
(company.address.street.name %= toUpper) e
```

- We have won.
- We have won at winning.

Problem?

But subsuming archaic ideas is not a noble goal

Can we do better? What other problems exist? Can we win winning against winning?

Problem?

JSON

```
data Json =
  JNull
  | JNumber Double
  | JArray [Json]
  | JObject [(Str, Json)]
  -- ...
```

Please set the object at "key" in the first array value to null.

Problem?

JSON

```
JArray [JObject [("key", JNumber 7)], JNumber 4]

JArray [JObject [("key", JNumber 7)], JNull]
```

Problem?

But what if

- We don't have an array?
- The array does not have a first value?
- The first value is not an object?
- The object does not have a "key"?

We need partiality in our lenses.

Problem?

But what if

- We don't have an array?
- The array does not have a first value?
- The first value is not an object?
- The object does not have a "key"?

We need partiality in our lenses.

Partiality

Partial Lens

```haskell
data PartialLens target con =
  PartialLens (target -> Maybe (con -> target, con))
```

Partial Lens

For example

```
jArray ::
  PartialLens JSON [Json]
jArray =
  PartialLens (\j ->
    case j of JArray a ->
      Just (JArray, a)
    _                  ->
      Nothing
  )
```

Partial Lens

However

This structure violates many of our desirable lens properties that we had come to rely on. Our three laws do not translate.

The Polymorphic Update problem

Suppose we have this structure

```
data StringAnd a =
  StringAnd String a
```

The Polymorphic Update problem

And two values such as

```
aLens :: Lens (StringAnd a) a
aLens = ...

value :: StringAnd [Int]
value = StringAnd "abc" [1,5,10,100]
```

The Polymorphic Update problem

And we need to modify the `[Int]` field to a `String`. However,

```
(%=) ::
  Lens target field ->
  (field -> field) ->
  (target -> target)

(%=) ::
  Lens (StringAnd a) a ->
  (a -> a) ->
  (StringAnd a -> StringAnd a)
```

The Polymorphic Update problem

We want to *polymorphically update* the field

```
(%=) aLensPoly ::
  (field -> newfield) ->
  (StringAnd field -> StringAnd newfield)
```

Solutions

The Theory of Lenses

There have been many efforts to find a unifying theory of lenses to address the practical problems that we have identified.

An inexhaustive list follows.

Solutions

data-lens

- Started in 2008 by Edward Kmett; maintained by Russ O'Connor and me.
- Hit walls with doing polymorphic update and partiality when experimenting.
- Mostly abandoned now due to subsumption. The solution was ultimately found.

Solutions

data-lens

- Started in 2008 by Edward Kmett; maintained by Russ O'Connor and me.
- Hit walls with doing polymorphic update and partiality when experimenting.
- Mostly abandoned now due to subsumption. The solution was ultimately found.

Solutions

data-lens

- Started in 2008 by Edward Kmett; maintained by Russ O'Connor and me.
- Hit walls with doing polymorphic update and partiality when experimenting.
- Mostly abandoned now due to subsumption. The solution was ultimately found.

Solutions

fclabels

- Started in 2009 by Sebastian Visser.
- Originally only resolved the fundamental problems addressed by lenses.
- Now supports polymorphic update, but partiality is problematic.

Solutions

fclabels

- Started in 2009 by Sebastian Visser.
- Originally only resolved the fundamental problems addressed by lenses.
- Now supports polymorphic update, but partiality is problematic.

Solutions

fclabels

- Started in 2009 by Sebastian Visser.
- Originally only resolved the fundamental problems addressed by lenses.
- Now supports polymorphic update, but partiality is problematic.

Solutions

mubbip-woo!

Asymmetric Lenses in Scala

- A paper in 2012 by me.
- An effort to invite discussion and improvements outside of Haskell.
- Discussion flourished, but Scala and "improvements" remain as elusive as yowies.

Solutions

mubbip-woo!

Asymmetric Lenses in Scala

- A paper in 2012 by me.
- An effort to invite discussion and improvements outside of Haskell.
- Discussion flourished, but Scala and "improvements" remain as elusive as yowies.

Solutions

mubbip-woo!

Asymmetric Lenses in Scala

- A paper in 2012 by me.
- An effort to invite discussion and improvements outside of Haskell.
- Discussion flourished, but Scala and "improvements" remain as elusive as yowies.

Solutions

Lenses in Scalaz

- `scalaz.{Lens, PLens, IndexedLens, IndexedPLens}`
- Polymorphic update, but still partiality eludes us, like yowies.

Solutions

Lenses in Scalaz

- `scalaz.{Lens, PLens, IndexedLens, IndexedPLens}`
- Polymorphic update, but still partiality eludes us, like yowies.

The Solution

Control.Lens

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

- Twan van Laarhoven lens representation
- Polymorphic update
- but. . . Partiality? Multiple update?

The Solution

Control.Lens.Prism

```haskell
type Prism s t a b =
  (Applicative f, Choice p) =>
  p a (f b) -> p s (f t)
```

- Solves partiality.
- Importantly, is *principled*.
- Gives rise to diverse practical consequences.
- No more hacks or hitting walls!

The Solution

Control.Lens.Traversal

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

- View and update *multiple* values.
- `Fold` to *only view* multiple values.

The Solution

and it gets interesting. . .

- These structures are just functions.
- A `Fold` is a `Traversal`.
- A `Prism` is a `Traversal`.
- They are all a `Lens`.
- They all compose with `(.)` (regular function composition).

The Solution

and even more and more interesting. . .

But let's leave it here :)