Asymmetric Lenses

Tony Morris

24 July 2012, Brisbane Functional Programming Group

Lens is CoState Comonad Coalgebra

It is true

- A Lens is exactly the Coalgebra for the CoState Comonad[a b]
- This is a concise, fully-describing statement, so we can go home now...
- ...or we can examine the practical implications

[a] perhaps better known as the Store Comonad
[b] *hat-tip Russell O'Connor*

The modify Problem

Record types

```
data Address = Address {
    street :: String
  , suburb :: String
}

data Person = Person {
    name :: String
  , address :: Address
}
```

The modify Problem

For example

- Supposing we wish to reverse or upper-case a person's name
- modifyName :: (**String** −> **String**) −> Person −> Person
- We have to write this function for *every* record field.

It Gets Worse

Code blowout

- We have to write many other functions, per record field
  - setName :: **String** -> Person -> Person
  - setNameAddress :: (**String**, Address) -> Person -> Person

And Even Worse

EEK!

- These functions must also be written as record fields are embedded
- How to set a person's suburb?

```
setSuburb :: String -> Person -> Person
setSuburb s p = p {
   address = address p {
      suburb = s
   }
}
```

- "My code looks like a gigantic Greater Than symbol!"

Introducing Lenses

Lens is a Data Type

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

Example Lens

```
nameLens :: Lens Person String
```

A universal modify

A universal modify

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

modify :: Lens a b -> (b -> b) -> a -> a
modify (Lens set get) m target =
   set target (m (get target))
```

Composing Lenses

- We now have a universal modify function to run with any lens
- How do we handle embedded fields?
  - A person has an address and an address has a suburb
  - How can we get to the person's suburb?

Composing Lenses

Compose the name and suburb lenses

```
nameLens :: Lens Person Address
suburbLens :: Lens Address String
```

Can we compose them?

```
Lens Person Address
-> Lens Address String
-> Lens Person String
```

Lens is a Semigroupoid

What is a Semigroupoid?

```
(>>>) ::
  semi a b
  -> semi b c
  -> semi a c
```

Example Semigroupoids

- (->)
- Monad m => Kleisli m
- Lens

Traversing the Graph

Since Lens is a Semigroupoid . . .

- We can compose to an arbitrary depth
- To reverse a person's suburb

```
reverseSuburb :: Person -> Person
reverseSuburb =
  modify
    (addressLens >>> suburbLens)
    reverse
```

- No more code shaped like a Greater Than symbol!

Lens Libraries

Not only does a Lens give rise to a Semigroupoid, but it also maps a set onto itself

Identity Lens

```
identityLens :: Lens a a
identityLens = Lens (const id) id
```

A Semigroupoid that has an identity element is called a *Category*. Lens is a category.

Lens Libraries

Two split lenses that map onto a value of the same element type produce a lens that can merge.

Split Lens

```
(|||) ::
  Lens a x
  -> Lens b x
  -> Lens (Either a b) x
Lens s1 g1 ||| Lens s2 g2 =
  Lens (either
          (\a -> Left . s1 a)
          (\b -> Right . s2 b))
       (either g1 g2)
```

Lens Libraries

Two disjoint lenses can be paired.

Lens Tensor

```
(***) ::
  Lens a b
  -> Lens c d
  -> Lens (a, c) (b, d)
Lens s1 g1 *** Lens s2 g2 =
  Lens (\(a, c) (b, d) -> (s1 a b, s2 c d))
       (\(a, c) -> (g1 a, g2 c))
```

Lens Libraries

And More

- **unzip** ::
  Lens s (a, b)
  -> (Lens s a, Lens s b)

- factor ::
  Lens (**Either** (a, b) (a, c))
  (a, **Either** b c)

- distribute ::
  Lens (a, **Either** b c)
  (**Either** (a, b) (a, c))

Lens Values

First Lens

```
fstLens :: Lens (a, b) a
fstLens =
  Lens (\(_, b) a -> (a, b)) fst
```

Second Lens

```
sndLens :: Lens (a, b) b
sndLens =
  Lens (\(a, _) b -> (a, b)) snd
```

Lens Values

Lenses on collections

- `mapLens ::`
  `Ord k => k -> Lens (Map k v) (Maybe v)`
- `setLens ::`
  `Ord a => a -> Lens (Set a) Bool`

Partial Lenses

Sum types

- Partial lenses provide *nullability* through the lens
- As a regular lens corresponds to fields of a record type, a partial lens corresponds to constructors of a sum type

- ```
  data PartialLens a b =
    PartialLens (a -> Maybe (b -> a, b))
  ```
  ```

Example Sum Type

JSON Data Type

```
data Json =
    JNull
  | JBool Bool
  | JNumber Double
  | JString String
  | JArray [Json]
  | JObject [(String, Json)]
```

How do we navigate this data structure to perform retrieval and updates?

Like this

With a gigantic >symbol that's how

```
case j of
  JArray x ->
    case x of
      [] -> j
      (h:t) -> JArray (case h of
        JArray y ->
          case y of
            a:b:u -> JArray (b:a:u)
            _ -> j) t
            _ -> j
  _ -> j
```

I Kid

Partial Lenses

- compose as a semigroupoid

```
(>>>) ::
  PartialLens a b
  -> PartialLens b c
  -> PartialLens a c
```

- split and merge

```
(|||) ::
  PartialLens a x
  -> PartialLens b x
  -> PartialLens (Either a b) x
```

- . . . and all those other helpful bits too!

Example

Partial Lens for the JArray constructor

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

Example Partial Lens Values

Partial Lens for the head of a list

```
jHeadLens ::
  PartialLens [a] a
jHeadLens =
  PartialLens
    (\x -> case x of
      (h:t) -> Just (\i -> i:t, h)
      _     -> Nothing)
```

Language Support

- All this boilerplate generating lenses for fields and constructors is a small price
- But do we have to pay it?
- Can the language do it for us?
- We want to generate values with type a `Lens` b ...
- ... instead of a $->$ b as in Haskell, Scala and everyone else

Language Support

- Boomerang —A bidirectional programming language for ad-hoc, textual data.
- Roy Programming Language —Brian McKenna (TBD)
- Template Haskell
- *Your Programming Language*

Further Topics

- Lenses with Polymorphic Update
- Fusing Lenses on the target to the pair of set/get (Store)
- Optimal Integration into a General Purpose Programming Language
- Lenses must obey laws

```haskell
import Data.Map
import Data.Set
import qualified Data.Map as M
import qualified Data.Set as S

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

modify ::
  Lens a b
  -> (b -> b)
  -> a
  -> a
modify (Lens set get) m target =
  set target (m (get target))

(>>>) ::
  Lens a b
  -> Lens b c
  -> Lens a c
Lens s1 g1 >>> Lens s2 g2 =
  Lens (\a c -> s1 a (s2 (g1 a) c)) (g2 . g1)

nameLens ::
  Lens Person String
nameLens =
  Lens (\p n -> p { name = n }) name

addressLens ::
  Lens Person Address
addressLens =
  Lens (\a n -> a { address = n }) address

suburbLens ::
```

```
  Lens Address String
suburbLens =
  Lens (\a n -> a { suburb = n }) suburb

reverseSuburb ::
  Person
  -> Person
reverseSuburb =
  modify (addressLens >>> suburbLens) reverse

data Address = Address {
  street :: String
, suburb :: String
}

data Person = Person {
  name :: String
, address :: Address
}

(|||) ::
  Lens a x
  -> Lens b x
  -> Lens (Either a b) x
Lens s1 g1 ||| Lens s2 g2 =
  Lens
    (either (\a -> Left . s1 a) (\b -> Right . s2 b))
    (either g1 g2)

(***) ::
  Lens a b
  -> Lens c d
  -> Lens (a, c) (b, d)
Lens s1 g1 *** Lens s2 g2 =
  Lens
    (\(a, c) (b, d) -> (s1 a b, s2 c d))
```

```
    (\(a, c) -> (g1 a, g2 c))

unzipL ::
  Lens s (a, b)
  -> (Lens s a, Lens s b)
unzipL (Lens s g) =
  (
    Lens (\t a -> s t (a, snd (g t))) (fst . g)
  , Lens (\t b -> s t (fst (g t), b)) (snd . g)
  )

factor ::
  Lens
    (Either (a, b) (a, c))
    (a, Either b c)
factor =
  Lens
    (\e (a, ee) -> either (\b -> Left (a, b)) (\c -> Right (a, c)) ee)
    (either (\(a, b) -> (a, Left b)) (\(a, c) -> (a, Right c)))

distribute ::
  Lens
    (a, Either b c)
    (Either (a, b) (a, c))
distribute =
  Lens
    (\_ -> either (\(aa, bb) -> (aa, Left bb)) (\(aa, cc) -> (aa, Right cc)))
    (\(a, e) -> either (\b -> Left (a, b)) (\c -> Right (a, c)) e)

fstLens ::
  Lens (a, b) a
fstLens =
  Lens (\(_, b) a -> (a, b)) fst

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

```haskell
sndLens =
  Lens (\(a, _) b -> (a, b)) snd

mapLens ::
  Ord k =>
  k
  -> Lens (Map k v) (Maybe v)
mapLens k =
  Lens
    (\m -> maybe (M.delete k m) (\v' -> M.insert k v' m))
    (M.lookup k)

setLens ::
  Ord a =>
  a
  -> Lens (Set a) Bool
setLens a =
  Lens
    (\s p -> (if p then S.insert else S.delete) a s)
    (S.member a)

data PartialLens a b = PartialLens (a -> Maybe (b -> a, b))

pmodify ::
  PartialLens a b
  -> (b -> b)
  -> a
  -> Maybe a
pmodify (PartialLens f) m target =
  fmap (\(s, g) -> s (m g)) (f target)

(>>>>) ::
  PartialLens a b
  -> PartialLens b c
  -> PartialLens a c
PartialLens f >>>> PartialLens g =
```

```
  PartialLens (\a ->
    do (x, b) <- f a
       (y, c) <- g b
       return (x . y, c))

jArrayLens ::
  PartialLens Json [Json]
jArrayLens =
  PartialLens
    (\j -> case j of
      JArray x -> Just (JArray, x)
      _ -> Nothing)

jHeadLens ::
  PartialLens [a] a
jHeadLens =
  PartialLens
    (\x -> case x of
      (h:t) -> Just (\i -> i:t, h)
      _ -> Nothing)

data Json =
  JNull
  | JBool Bool
  | JNumber Double
  | JString String
  | JArray [Json]
  | JObject [(String, Json)]

(||||) ::
  PartialLens a x
  -> PartialLens b x
  -> PartialLens (Either a b) x
PartialLens f |||| PartialLens g =
  PartialLens
    (either
```

```
      (fmap (\(p, q) -> (Left . p, q)) . f)
      (fmap (\(p, q) -> (Right . p, q)) . g))

(****) ::
  PartialLens a b
  -> PartialLens c d
  -> PartialLens (a, c) (b, d)
PartialLens f **** PartialLens g =
  PartialLens
    (\(a, c) -> do (b, b') <- f a
                   (d, d') <- g c
                   return (\(t, u) -> (b t, d u), (b', d')))

punzipL ::
  PartialLens s (a, b)
  -> (PartialLens s a, PartialLens s b)
punzipL (PartialLens f) =
  (
    PartialLens (fmap (\(a, (p, q)) -> (\k -> a (k, q), p)) . f)
  , PartialLens (fmap (\(b, (p, q)) -> (\k -> b (p, k), q)) . f)
  )

pfactor ::
  PartialLens
    (Either (a, b) (a, c))
    (a, Either b c)
pfactor =
  PartialLens
    (Just . \e ->
      (
        \(a, ee) -> either (\b -> Left (a, b)) (\c -> Right (a, c)) ee
      , either (\(a, b) -> (a, Left b)) (\(a, c) -> (a, Right c)) e
      ))

pdistribute ::
  PartialLens
```

```
    (a, Either b c)
    (Either (a, b) (a, c))
pdistribute =
  PartialLens
    (Just . \(a, e) ->
      (
        either (\(aa, bb) -> (aa, Left bb)) (\(aa, cc) -> (aa, Right cc))
      , either (\b -> Left (a, b)) (\c -> Right (a, c)) e
      ))
```