Fun with Profunctors

Phil Freeman

Agenda

- Different types of functor
- Profunctor examples
- Profunctor lenses

Functor

```
class Functor f where
  fmap :: (a → b) → f a → f b



data Burrito filling = Tortilla filling

instance Functor Burrito where
  fmap f (Tortilla filling)
    = Tortilla (f filling)
```

Contravariant Functors

```
class Contravariant f where
  cmap :: (a → b) → f b → f a
```

```
data Customer filling = Eat (filling → IO ())

instance Contravariant Customer where
  cmap f (Eat eat) = Eat (eat ○ f)
```

Aside

- Something is a **Functor** when its type argument only appears in *<u>positive position</u>*.
- Something is **Contravariant** when its type argument only appears in *<u>negative position</u>*.

Examples (positive, negative):

- a
- $a \rightarrow a$
- $a \rightarrow a \rightarrow a$
- $(a \rightarrow a) \rightarrow a$
- $((a \rightarrow a) \rightarrow a) \rightarrow a$

Can we mix the two?

Invariant Functors

```
class Invariant f where
  imap :: (b → a) → (a → b) → f a → f b
```

```
data Endo a = Endo (a → a)

instance Invariant Endo where
  imap f g (Endo e) = Endo (g ∘ e ∘ f)
```

Invariant Functors

What can we do with **Invariant** things?

Not much, but:

```
data Iso a b = Iso (a → b) (b → a)
```

```
iso :: Invariant f => Iso a b → f a → f b
iso (Iso to from) = imap from to
```

Invariant Functors

- Invariant functors can be quite tricky to work with in general.
- The Functor => Applicative => Monad hierarchy doesn’t seem to fit.
- To map, we have to be able to invert the function we want to map.

Instead, split the type argument into *two type arguments*.

Profunctors

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

instance Profunctor (→) where
  dimap f g k = g ∘ k ∘ f
```

Profunctors

What can we do with **Profunctors**?

We get the same lifting operation from before:

```
isoP :: Profunctor p => Iso a b → p a a → p b b
isoP (Iso to from) = dimap to from
```

Profunctors

```
swapping :: Profunctor p => p (a, b) (x, y)
                         → p (b, a) (y, x)
swapping = dimap swap swap

assoc :: Profunctor p => p ((a, b), c) ((x, y), z) →
                      → p (a, (b, c)) (x, (y, z))
assoc = dimap (\(a, (b, c)) → ((a, b), c)) (\((a, b), c) → (a, (b, c)))

-- Try composing these:
swapping ○ swapping :: Profunctor p => p (a, b) (x, y)
                                    → p (a, b) (x, y)
assoc ○ swapping    :: Profunctor p => p (a, (b, c)) (x, (y, z))
                                    → p (b, (c, a)) (y, (z, x))
```

Examples

```
data Forget r a b = Forget { runForget :: a → r }

instance Profunctor Forget where
  dimap f _ (Forget forget) = Forget (forget ∘ f)
```

Examples

```
data Star f a b = Star { runStar :: a → f b }

instance Functor f => Profunctor (Star f) where
  dimap f g (Star star) = Star (fmap g ∘ star ∘ f)
```

Examples

```
data Costar f a b = Costar { runCostar :: f a → b }

instance Functor f => Profunctor (Costar f) where
  dimap f g (Costar costar) = Costar (g ○ costar ○ fmap f)
```

Examples

```
data Fold m a b = Fold { runFold :: (b → m) → a → m }

instance Profunctor (Fold m) where
  dimap f g (Fold fold) = Fold $ \k → fold (k ○ g) ○ f
```

Examples

```
data Mealy a b = Mealy { runMealy :: a → (b, Mealy a b) }

instance Profunctor Mealy where
  dimap f g = go
    where
      go (Mealy mealy) = Mealy $ (g *** go) ∘ mealy ∘ f
```

Strengthening Profunctors

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

class Profunctor p => Strong p where
  first  :: p a b → p (a, x) (b, x)
  second :: p a b → p (x, a) (x, b)

-- The function arrow is a strong profunctor
instance Strong (→) where
  first  f (a, x) = (f a, x)
  second f (x, a) = (x, f a)
```

Examples

```
instance Strong (Forget r)
instance Functor f => Strong (Star f)
instance Comonad w => Strong (Costar w)
instance Strong (Fold m)
instance Strong Mealy
```

Strengthening Profunctors

```
-- Try composing these:
first          :: Strong p => p a b → p (a, x) (b, x)
first ∘ first  :: Strong p => p a b → p ((a, x), y) ((b, x), y)
first ∘ second :: Strong p => p a b → p ((x, a), y) ((x, b), y)

-- These are starting to look a lot like lenses!
```

Strengthening Profunctors

```
-- Our new functions also compose with isos:
assoc ∘ first :: Strong p => p (a, b) (x, y)
                           → p (a, (b, z)) (x, (y, z))



-- These are starting to look a lot like lenses!
```

Lens Primer

...in GHCi

Profunctor Lenses

```
-- Let’s define our lens and iso types in terms of these classes:
type Iso  s t a b = forall p. Profunctor p => p a b → p s t
type Lens s t a b = forall p. Strong p     => p a b → p s t

-- Note: every Iso is automatically a Lens!
```

Lenses as Isos

```
-- Consider how we can construct a lens
type Lens s t a b = forall p. Strong p => p a b → p s t

-- All we have are dimap and first
dimap :: Profunctor p => (c → a) → (b → d) → p a b → p c d
first :: Strong p => p a b → p (a, x) (b, x)

-- Every profunctor lens has a normal form
nf :: (s → (a, x)) → ((b, x) → t) → Lens s t a b
nf f g pab = dimap f g (first pab)
```

Lenses as Isos

```
-- In other words:
iso2lens :: Iso s t (a, x) (b, x) → Lens s t a b
iso2lens iso pab = iso (first pab)

-- A lens asserts that
-- the family of types given by s and t
-- is (uniformly) isomorphic to a product

-- We can go the other way with some work:
lens2iso :: Lens s t a b → exists x. Iso s t (a, x) (b, x)
```

Lens Combinators

```
-- Let's write some useful functions with lenses:
get :: Lens s t a b → s → a
get lens = runForget (lens (Forget id))

set :: Lens s t a b → b → s → t
set lens b = lens (const b)

modify :: Lens s t a b → (a → b) → s → t
modify lens f = lens f

-- We didn't really need a full lens
```

Choice

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

class Profunctor p => Choice p where
  left  :: p a b → p (Either a x) (Either b x)
  right :: p a b → p (Either x a) (Either x b)

-- The function arrow has choice
instance Choice (→) where
  left  f (Left  a) = Left (f a)
  left  _ (Right x) = Right x
  right _ (Left  x) = Left x
  right f (Right a) = Right (f a)
```

Examples

```
instance Monoid m => Choice (Forget m)
instance Applicative f => Choice (Star f)
instance Comonad w => Choice (Costar w)
instance Monoid m => Choice (Fold m)
instance Choice Mealy
```

Choice

```
-- left and right compose as before:
left           :: Choice p => p a b → p (Either a x) (Either b x)
left ○ left    :: Choice p => p a b → p (Either (Either a x) y)
                                         (Either (Either b x) y)
left ○ right   :: Choice p => p a b → p (Either (Either x a) y)
                                         (Either (Either x b) y)
```

Choice

```
-- left and right also compose with isos and lenses
first ∘ left :: (Strong p, Choice p) => p a b
                                     → p (Either a x, y) (Either b x, y)
```

Prisms

```
-- We’ve rediscovered Prisms!
type Iso   s t a b = forall p. Profunctor p => p a b → p s t
type Lens  s t a b = forall p. Strong p     => p a b → p s t
type Prism s t a b = forall p. Choice p     => p a b → p s t

-- Note: every Iso is automatically a Prism!
```

0-1 Traversals

```
type AffineTraversal s t a b = forall p. (Strong p, Choice p) => p a b → p s t

first ○ left :: AffineTraversal (Either a x, y) (Either b x, y) a b
```

Prisms as Isos

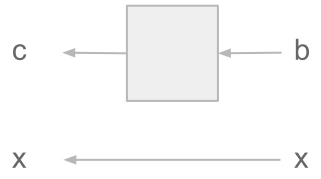
```
-- Normal forms for prisms:
iso2prism :: Iso s t (Either a x) (Either b x) → Prism s t a b
iso2prism iso pab = iso (left pab)

-- A prism asserts that
-- the family of types given by s and t
-- is (uniformly) isomorphic to a sum
```

Arrows

```
class (Strong a, Category a) => Arrow a

instance Arrow (→)
instance Applicative f => Arrow (Star f)
instance Comonad w => Arrow (Costar w)
instance Monoid m => Arrow (Fold m)
instance Arrow Mealy
```

Proc Notation

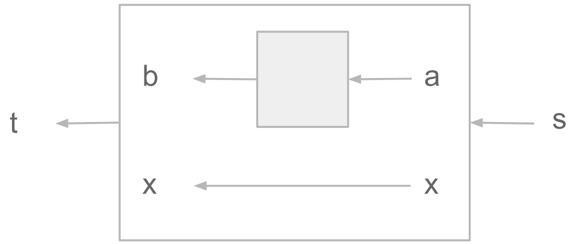
```
{-# LANGUAGE Arrows #-}

assoc :: Arrow a => a b c → a (b, x) (c, x)
assoc arr = proc (b, x) → do
              c ↞ arr ↞ b
              returnA ↞ (c, x)
```



Lenses in Pictures

```
nf :: Iso s t (a, x) (b, x) → Lens s t a b
nf iso pab = iso (first pab)
```

```
-- Many optics can be thought of in terms of
-- these “diagram transformers”
```



More Optics

```
type Optic c s t a b = forall p. c p => p a b → p s t

type Iso       = Optic Profunctor
type Lens      = Optic Strong
type Prism     = Optic Choice
type Traversal = Optic Traversing
type Grate     = Optic Closed
type SEC       = Optic ((→) ~)

class (Strong p, Choice p) => Traversing p where
  traversing :: Traversable t => p a b → p (t a) (t b)

class Profunctor p => Closed p where
  closed :: p a b → p (x → a) (x → b)
```

More Optics

Iso	**Profunctor**	$s_i \sim a_i$
Lens	**Strong**	$s_i \sim (a_i, x)$
Prism	**Choice**	$s_i \sim \text{Either } a_i\ x$
Traversal	**Traversing**	$s_i \sim t\ a_i$
Grate	**Closed**	$s_i \sim x \to a_i$
AffineTraversal	**Strong, Choice**	$s_i \sim \text{Either } (a_i, x)\ y$

Pros & Cons

Pros:

- More consistent API
- Many optics can be “inverted”
- We can apply our optics to more structures

Cons:

- Indexed optic story isn’t great
- Some changes to the API needed for performance

Real-World Example

```
-- data UI state = UI { runUI :: (state → IO ()) → state → IO Widget }

data UI a b = UI { runUI :: (a → IO ()) → b → IO Widget }

instance Profunctor UI
instance Strong UI
instance Choice UI
instance Traversing UI

type UI’ state = UI state state

animate :: UI’ state → state → IO ()
animate ui state = do
  widget ← runUI ui (animate ui) state
  render widget
```

Real-World Example

```
first       :: UI’ state → UI’ (state, x)

left        :: UI’ state → UI’ (Either state x)

traversing :: UI’ state → UI’ [state]
```