Profunctor optics, a categorical update

(Extended abstract)

Mario Román, Bryce Clarke, Fosco Loregian, Emily Pillmore,
Derek Elkins, Bartosz Milewski and Jeremy Gibbons

September 5, 2019

NWPT'19, TALTECH

Part 1: Motivation

Optics

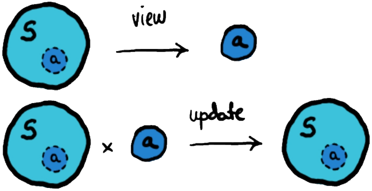
Optics are composable data accessors. They allow us to access and modify nested data structures.

- Each family of optics encodes a data accessing pattern.
  - Lenses access subfields.
  - Prisms pattern match.
  - Traversals transform lists.
- Two optics (of any two families!) can be directly composed.

Lenses

Definition (Oles, 1982)

$$\mathbf{Lens}\,(A, S) := (S \to A) \times (S \times A \to S).$$

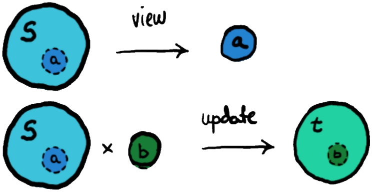



```
data Lens a s = Lens
  { view :: s -> a
  , update :: s -> a -> s
  }
```

Lenses

Definition (Oles, 1982)

$$\mathbf{Lens}\left(\begin{pmatrix}A\\B\end{pmatrix},\begin{pmatrix}S\\T\end{pmatrix}\right) := (S \to A) \times (S \times B \to T).$$




```haskell
data Lens a b s t = Lens
  { view :: s -> a
  , update :: s -> b -> t
  }
```

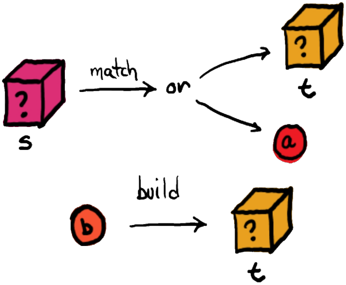
Lenses

```
let taltech = Building { address = Address
     { street = "Akadeemia tee 21"
     , zipcode = 19086
     , city = "Tallinn" }
  , name = "Tallinn University of Technology" }

>>> taltech^.address
"Akadeemia tee 21, 19086 Tallinn, Estonia"

>>> taltech^.address.street
"Akadeemia tee 21"

>>> taltech^.address.street <~ "Ehitajate tee 5"
Building "Tallinn University of Technology"
 @ "Eitahate tee 5, 19086 Tallinn, Estonia"
```

Prisms

Definition

$$\textbf{Prism}\left(\binom{A}{B},\binom{S}{T}\right) = (S \to T + A) \times (B \to T).$$




```
data Prism a b s t = Prism
  { build :: b -> t
  , match :: s -> Either a t
  }
```

Prisms

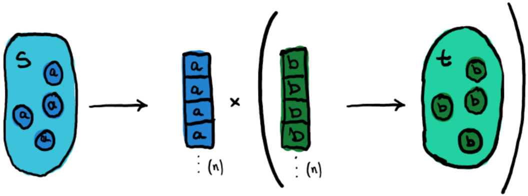
```haskell
divide :: Int -> Prism Int Int
divide n = Prism (n *) match
  where match x = if mod x n == 0
          then Just (div x n)
          else Nothing

>>> 42 ^? divide 3 . divide 2 . divide 7
Just 1

>>> divide 3 # 4
12
```

Adapted from Penner's @opticsbyexample

Traversals

Definition

$$\textbf{Traversal}\left(\begin{pmatrix}A\\B\end{pmatrix},\begin{pmatrix}S\\T\end{pmatrix}\right)=\left(S\to\sum_n A^n\times(B^n\to T)\right).$$




```
data Traversal a b s t = Traversal
  { extract :: s -> ([a] , [b] -> t)
  }
```

The problem of modularity

- How to compose any two optics?
- Even from different families of optics (lens+prism+traversal).
- Simple but tedious code.
- Every pair of families needs special attention.

```haskell
-- Given a lens and a prism,
street  :: Lens  Address Street
address :: Prism String  Address

-- the composition is neither a lens nor a prism.
parseStreet :: String
            -> Either String (Street , Street -> Postal)
parseStreet s = case match address s of
  Left  addr -> Left  addr
  Right post -> Right (view street post, set street post)
```

Profunctor optics

```haskell
let venues =
  [ "Taltech. Eitahate tee 5, 19086 Tallinn, Estonia"
  , "Oslotech. Gaustadalléen 21, 0349 Oslo, Norway"
  , "Linnateatteri. Puutarhakatu 8B, 20100 Turku, Finland" ]

-- We can compose lenses, prisms and traversals.
each    :: Traversal [String] String
address :: Prism     String   Address
country :: Lens      Address  String

>>> venues^.each.address.country %~ uppercase
[ "Taltech. Eitahate tee 5, 19086 Tallinn, ESTONIA"
, "Oslotech. Gaustadalléen 21, 0349 Oslo, NORWAY"
, "Linnateatteri. Puutarhakatu 8B, 20100 Turku, FINLAND" ]
```

Profunctor optics

```
class (Profunctor p) => Tambara mon p where
  action :: forall f a b . (mon f) => p a b -> p (f a) (f b)
```

A Tambara module is a profunctor endowed with a natural transformation $p(A, B) \to p(C \otimes A, C \otimes B)$ subject to some conditions. Every optic can be written as a function polymorphic on these Tambara modules. Why is this?

Outline

- Existential optics: a definition of optic.
- Profunctor optics: on optics as parametric functions.
- Composing optics: on how composition works.
- Case study: on how to invent an optic.
- Further work: and implementations.

Part 2: Existential optics

Parametricity as ends

- We write $\forall$ to denote polymorphism (*actually, ends*).
- We write $\exists$ to denote existential types (*coends*).

Parametricity (*Yoneda lemma*) implies the following rules.

- $\forall X.((A \to X) \to GX) \cong GA$
- $\exists X.((X \to A) \times FX) \cong FA$

Continuity implies the following.

- $((\exists C.FC) \to D) \cong (\forall C.FC \to D)$
- $(D \to (\forall C.PC)) \cong (\forall C.D \to PC)$

These are rules from *(co)end calculus*. See Loregian's "Coend calculus".

Parametricity as ends

- We write $\forall$ to denote polymorphism (*actually, ends*).
- We write $\exists$ to denote existential types (*coends*).

Parametricity (*Yoneda lemma*) implies the following rules.

- $\forall X.(\cancelto{X = A}{(A \to X)} \to GX) \cong GA$
- $\exists X.(\cancelto{X = A}{(X \to A)} \times FX) \cong FA$

Continuity implies the following.

- $((\exists C.FC) \to D) \cong (\forall C.FC \to D)$
- $(D \to (\forall C.PC)) \cong (\forall C.D \to PC)$

These are rules from *(co)end calculus*. See Loregian's "Coend calculus".

A definition of "optic"

Definition (Milewski, Boisseau/Gibbons, Riley, simplified)

Fix a monoidal class of endofunctors $\mathbf{M}$ (that is, a *constraint* satisfied by the identity and closed under composition, such as *Applicative* or *Traversable*).

An optic from (S, T) with *focus* on (A, B) is an element of the following type.

$$\mathbf{Optic}\left(\begin{pmatrix} A \\ B \end{pmatrix}, \begin{pmatrix} S \\ T \end{pmatrix}\right) := \exists M \in \mathbf{M}.\ (S \to MA) \times (MB \to T).$$

Intuition: The optic splits into some focus A and some *context* M. We cannot access that context, but we can use it to update.

```
data ExOptic mon a b s t where
  ExOptic :: (mon m) => (s -> m a) -> (m b -> t)
              -> ExOptic mon a b s t
```

Lenses are optics

```
(s -> a) × (s × b -> t) ≅ ExOptic (×) a b s t
```

Proposition (from Milewski, 2017)

Lenses are optics for the product.

Proof.

$$\begin{aligned} \exists C.\, (S \to C \times A) \times (C \times B \to T) &\cong && (\textit{Product}) \\ \exists C.\, (S \to C) \times (S \to A) \times (C \times B \to T) &\cong && (\textit{Yoneda}) \\ (S \to A) \times (S \times B \to T) & && \end{aligned}$$

Lenses are optics

```
(s -> a) × (s × b -> t) ≅ ExOptic (×) a b s t
```

Proposition (from Milewski, 2017)

Lenses are optics for the product.

Proof.

$$\exists C.\ (S \to C \times A) \times (C \times B \to T) \cong \qquad \text{(Product)}$$

$$\exists C.\ \cancelto{S\,=\,C}{(S \to C)} \times (S \to A) \times (C \times B \to T) \cong \qquad \text{(Yoneda)}$$

$$(S \to A) \times (S \times B \to T)$$

Prisms are optics

```
(s -> Either a t) × (b -> t) ≅ ExOptic (+) a b s t
```

Proposition (Milewski, 2017)

Prisms are optics for the coproduct.

Proof.

$$\exists M.\, (S \to M + A) \times (M + B \to T) \cong \quad \text{(Coproduct)}$$

$$\exists M.\, (S \to M + A) \times (M \to T) \times (B \to T) \cong \quad \text{(Yoneda)}$$

$$(S \to T + A) \times (B \to T)$$

Prisms are optics

```
(s -> Either a t) × (b -> t) ≅ ExOptic (+) a b s t
```

Proposition (Milewski, 2017)

Prisms are optics for the coproduct.

Proof.

$$\begin{aligned}
\exists M.\, (S \to M + A) \times (M + B \to T) &\cong \quad \text{(Coproduct)} \\
\exists M.\, (S \to M + A) \times \cancelto{M = T}{(M \to T)} \times (B \to T) &\cong \quad \text{(Yoneda)} \\
(S \to T + A) \times (B \to T) &
\end{aligned}$$

Traversals are optics

```
s -> ([a], [b] -> t) ≅ ExOptic Series a b s t
```

Proposition

Traversals are optics for the action of polynomial functors $\sum_n C_n \times \square^n$.

That is,

$$\exists C.\ \left(S \to \sum_n C_n \times A^n\right) \times \left(\left(\sum_n C_n \times B^n\right) \to T\right) \cong \left(S \to \sum_n A^n \times (B^n \to T)\right).$$

Traversals are optics: proof

Again by the Yoneda lemma, this time for functors $C\colon \mathbb{N} \to \mathbf{Sets}$.

$$\begin{aligned}
\exists C.\ \left(S \to \sum\nolimits_n C_n \times A^n\right) \times \left(\sum\nolimits_n C_n \times B^n, T\right) &\cong \quad \textit{(cocontinuity)} \\
\exists C.\ \left(S \to \sum\nolimits_n C_n \times A^n\right) \times \prod_n (C_n \times B^n \to T) &\cong \quad \textit{(prod/exp adjunction)} \\
\exists C.\ \left(S \to \sum\nolimits_n C_n \times A^n\right) \times \prod_n (C_n \to (B^n \to T)) &\cong \quad \textit{(natural transformation)} \\
\exists C.\ \left(S, \sum\nolimits_n C_n \times A^n\right) \times \mathrm{Nat}\left(C_\square, (B^\square \to T)\right) &\cong \quad \textit{(Yoneda lemma)} \\
S \to \sum\nolimits_n A^n \times (B^n \to T) &
\end{aligned}$$

Programming libraries use traversable functors to describe traversals. Polynomials are related to these *traversable* functors by the work of Jaskelioff and O'Connor.

Traversals are optics: proof

Again by the Yoneda lemma, this time for functors $C\colon \mathbb{N} \to \mathbf{Sets}$.

$$
\begin{aligned}
\exists C.\left(S \to \sum_n C_n \times A^n\right) \times \left(\sum_n C_n \times B^n, T\right) &\cong \quad \textit{(cocontinuity)}\\
\exists C.\left(S \to \sum_n C_n \times A^n\right) \times \prod_n (C_n \times B^n \to T) &\cong \quad \textit{(prod/exp adjunction)}\\
\exists C.\left(S \to \sum_n C_n \times A^n\right) \times \prod_n (C_n \to (B^n \to T)) &\cong \quad \textit{(natural transformation)}\\
& \qquad C = B^{\square} \to T\\
\exists C.\left(S, \sum_n C_n \times A^n\right) \times \mathrm{Nat}\left(C_{\square}, (B^{\square} \to T)\right) &\cong \quad \textit{(Yoneda lemma)}\\
S \to \sum_n A^n \times (B^n \to T)
\end{aligned}
$$

Programming libraries use traversable functors to describe traversals. Polynomials are related to these *traversable* functors by the work of Jaskelioff and O'Connor.

Unification of optics

All the usual optics are of this form. Some new ones arise naturally.

Name	Concrete	Action
Adapter	$(S \to A) \times (B \to T)$	Identity
Lens	$(S \to A) \times (B \times S \to T)$	Product
Prism	$(S \to T + A) \times (B \to T)$	Coproduct
Grate	$((S \to A) \to B) \to T$	Exponential
Affine Traversal	$S \to T + A \times (B \to T)$	Product and coproduct
Glass	$((S \to A) \to B) \to S \to T$	Product and exponential
Traversal	$S \to \Sigma n.A^n \times (B^n \to T)$	Polynomials
Setter	$(A \to B) \to (S \to T)$	Any functor

Unification of optics

```haskell
-- This definition generalizes all the optics we care about.
data ExOptic mon a b s t where
  ExOptic :: (mon f) => (s -> f a) -> (f b -> t)
          -> ExOptic mon a b s t

-- Under suitable definitions for the constraints.
type Lens      s t a b = ExOptic (×) a b s t
type Prism     s t a b = ExOptic (+) a b s t
type Traversal s t a b = ExOptic Series a b s t
type Setter    s t a b = ExOptic Functor a b s t
```

Part 3: the Profunctor representation theorem

Tambara modules

Definition (from Pastro/Street)

A Tambara module is a profunctor P together with a family of morphisms satisfying some coherence conditions.

$$P(A, B) \to P(MA, MB), \qquad M \in \mathbf{M}.$$

Pastro and Street showed they are algebras for a monad.

$$\Psi Q(X, Y) = \exists M, A, B.\, Q(A, B) \times (MA \to X) \times (Y \to MB)$$

We call Tmb to the Eilenberg-Moore category for the monad.

```
class (Profunctor p) => Tambara mon p where
  action :: forall f a b . (mon f) => p a b -> p (f a) (f b)
```

Profunctor representation

Theorem (Boisseau/Gibbons)

Optics are functions parametric over Tambara modules.

$$\mathbf{Optic}((A,B),(S,T)) \cong \forall P \in \mathrm{Tmb}.\ P(A,B) \to P(S,T)$$

```
type ProfOptic mon a b s t = forall p . Tambara mon p
  => p a b -> p s t

theorem :: forall mon a b s t .
  ExOptic mon a b s t -> ProfOptic mon a b s t
theorem (ExOptic l r) = dimap l r . (action @mon)
```

Part 4: Composition of optics

Composing optics via coproducts

When we compose two optics in Haskell, the compiler joins the constraints. Is this an optic according to the definition? If so, for which action?

```
forall p . Tmb(+) p =>          p a b -> p s t
forall p . Tmb(×) p =>          p x y -> p a b
-----------------------------------------------
forall p . Tmb(+) p, Tmb(×) p => p x y -> p a b
```

- In other words, P has a bialgebra structure.
- This is the same as P having algebra structure for the coproduct monad (Kelly, Adamek).
- We prove the coproduct monad is the monad for the coproduct action.

Composing optics via coproducts

When we compose two optics in Haskell, the compiler joins the constraints. Is this an optic according to the definition? If so, for which action?

```
forall p . Tmb(+) p =>          p a b -> p s t
forall p . Tmb(×) p =>          p x y -> p a b
------------------------------------------------
forall p . Tmb(+,×) p =>        p x y -> p a b
```

- In other words, P has a bialgebra structure.
- This is the same as P having algebra structure for the coproduct monad (Kelly, Adamek).
- We prove the coproduct monad is the monad for the coproduct action.

Composing optics via distributive laws

- The folklore is that lenses and prisms compose into the optic for the action of a *single* sum and product.

```
D × (C + A)
```

- Haskell actually composes lenses and prisms into the optic for the action of *multiple* sums and products.

$$D_n + C_n \times (.. + C_3 \times (D_3 + C_2 \times (C_2 + D_1 \times (C_1 + A))))$$

In which sense is folklore right?

Composing optics via distributive laws

- The folklore is that lenses and prisms compose into the optic for the action of a *single* sum and product.

```
D × (C + A)
```

- Haskell actually composes lenses and prisms into the optic for the action of *multiple* sums and products.

```
Dn + Cn × (.. + C3 × (D3 + C2 × (C2 + D1 × (C1 + A))))
```

In which sense is folklore right? We show that the fact that $(\times)$ distributes over $(+)$ induces a distributive law between the Pastro-Street monads.

Composing optics via distributive laws

Monads can be joined in two ways.

- Taking their coproduct monad $S \oplus T$; and
- using a distributive law $ST \Rightarrow TS$ to induce a monad structure on the composition TS.

Composing optics via distributive laws

Monads can be joined in two ways.

- Taking their coproduct monad $S \oplus T$; and
- using a distributive law $ST \Rightarrow TS$ to induce a monad structure on the composition TS.

Families of optics can be joined in two ways.

- Taking their coproduct (as Haskell does),
- using a distributive law between them to induce optic structure on the composition.

Can we make this analogy precise?

Composing optics via distributive laws

Monads can be joined in two ways.

- Taking their coproduct monad $S \oplus T$; and
- using a distributive law $ST \Rightarrow TS$ to induce a monad structure on the composition TS.

Families of optics can be joined in two ways.

- Taking their coproduct (as Haskell does),
- using a distributive law between them to induce optic structure on the composition.

Can we make this analogy precise?

- Families of optics are a class of promonads (monoids in endoprofunctors).
- Coproducts of promonads correspond to their coproduct.
- Distributive laws between promonads are their distributive laws.

Part 4: Summary and further work

Summary

- Optics: a zoo of accessors used by programmers *[Kmett, lens library, 2012]*.
  - General definition: unified definition of optics as a coend.
  - Concrete cases: constructing new optics.
- Profunctor optics: for monoidal actions *[Pastro/Street, 2008]*, *[Milewski, 2017]* and general actions *[Boisseau/Gibbons, 2018]*.
  - Profunctor representation: can be composed easily.
  - Going from existential to profunctor and back is done in general.
- Composition of optics: what do we get when composing two optics.
  - Haskell considers coproducts of monads.
  - Composing with distributive laws is another natural choice.
  - What are other applications of promonads in programming?

Related and further work

- Lawful optics. Studied by *[Riley, 2018]*.
  - Programmers use lawful optics, optics with certain properties.
- Generalizations: in which other settings do we get useful results?
  - Enrichments over a cartesian Benabou cosmos $\mathcal{V}$.
  - We have extended the theorems for *mixed optics*.
- Implementation: developing libraries of optics.
  - A concise library in Haskell. `https://github.com/mroman42/vitrea/`
  - Derivations in Agda / Idris allow us to extract translation algorithms for optics. Everything we have been doing is constructive.

```
lensDerivation {s} {t} {a} {b} =
  begin
   ((∫exists c ∈ Set , ((s -> c × a) × (c × b -> t))))        ≅⟨ ≅-coend (λ c -> trivial) ⟩
   ((∫exists c ∈ Set , (((s -> c) × (s -> a)) × (c × b -> t)))) ≅⟨ ≅-coend (λ c -> trivial) ⟩
   ((∫exists c ∈ Set , ((s -> c) × (s -> a) × (c × b -> t))))   ≅⟨ yoneda ⟩
   ((s -> a) × (s × b -> t))
  qed
```

Some literature

Oles, 1982. *A category theoretic approach to the semantics of programming languages (PhD thesis).* Defines lenses for the first time.

Kmett, 2012. *Lens library.* Implements optics in Haskell.

Pickerings/Gibbons/Wu, 2016. *Profunctor optics: modular data accessors.* Derives lenses, prisms, adapters and traversals in Haskell.

Milewski, 2017. *Profunctor optics, the categorical view.* Tambara modules for lenses and prisms.

Boisseau/Gibbons, 2018. *What you needa know about Yoneda.* General definition of optics and a general profunctor representation theorem. Traversal as the optic for traversables.

Riley, 2018. *Categories of optics.* General framework for obtaining laws for the optics.