More Optics

Haskell and Cryptocurrencies

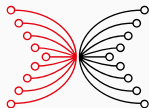Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK
Dr. Polina Vinogradova, IOHK

2019-02-21

Goals

- Profunctors
- Prisms
- Isos
- Polymorphic Updates

Profunctors

Profunctor

Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A profunctor P from $\mathcal{C}$ to $\mathcal{D}$ is a functor $P : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$.

Explicitly, this means that for objects X, $Y \in \mathrm{Ob}(\mathcal{C})$, PXY is an object of $\mathcal{D}$, and for morphisms $f : X' \to X$ and $g : Y \to Y'$ in $\mathcal{C}$, $Pfg : PXY \to PX'Y'$ is a morphism in $\mathcal{D}$, such that $P1_X1_Y = 1_{PXY}$ and such that for morphisms $f' : X'' \to X'$ and $g' : Y' \to Y''$ in $\mathcal{C}$, $P(ff')(g'g) = Pf'g' \circ Pfg : PXY \to PX''Y''$.

$$
\begin{array}{ccc}
X & Y & PXY \\
\end{array}
$$

Let $\mathcal{C}$ be a category. The standard example for a profunctor is

$$\mathrm{Mor}_{\mathcal{C}}(\cdot, \cdot) : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \underline{\mathrm{Set}}.$$

For morphisms $f : X' \to X$ and $g : Y \to Y'$ in $\mathcal{C}$, composition with f and g gives

$$\underbrace{X' \xrightarrow{f} \overbrace{X \xrightarrow{\alpha} Y}^{\in \mathrm{Mor}_{\mathcal{C}}(X,Y)} \xrightarrow{g} Y'}_{\in \mathrm{Mor}_{\mathcal{C}}(X',Y')},$$

and the laws follow immediately from the category laws.

The **profunctors**-package (by Edward Kmett, the author of the **lens**-library) defines

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

(in `Data.Profunctor`).

The Haskell analogue to the standard example from the last slide is:

```
instance Profunctor (->) where
  dimap ab cd bc = cd . bc . ab
```
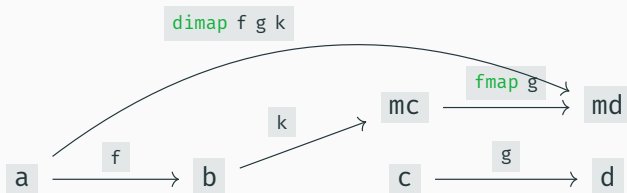
In `Control.Arrow`, we find the following definition:

```haskell
newtype Kleisli m a b =
  Kleisli {runKleisli :: a -> m b}
```

If `m` is a `Monad`, then `Kleisli m` is a `Profunctor`:

```haskell
instance Monad m => Profunctor (Kleisli m) where
  dimap f g k =
    Kleisli (fmap g . runKleisli k . f)
```

The `tagged` package, again by Edward Kmett, contains another example in `Data.Tagged` :

```
newtype Tagged a b = Tagged {unTagged :: b}
```

```
instance Profunctor Tagged where
  dimap _ g t = Tagged (g (unTagged t))
```

$$
\begin{array}{ccc}
& & t \longmapsto gt \\
a \xrightarrow{\ f\ } b & & c \xrightarrow{\ g\ } d
\end{array}
$$

Let's look at one other type class related to profunctors (from `Data.Profunctor.Choice`):

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

Note that `left'` can be defined given `right'`, and vice versa.

All the `Profunctor` instances we have seen until now are also istances of `Choice` :

```
instance Choice (->) where
  left' f  = either (Left . f) Right
  right' f = either Left (Right . f)
```

All the `Profunctor` instances we have seen until now are also istances of `Choice`:

```
instance Monad m => Choice (Kleisli m) where
  left' k  = Kleisli (either
    (fmap Left . runKleisli k)
    (return . Right))
  right' k = Kleisli (either
    (return . Left)
    (fmap Right . runKleisli k))
```

All the Profunctor instances we have seen until now are also istances of Choice :

```haskell
instance Choice Tagged where
  left'  = Tagged . Left . unTagged
  right' = Tagged . Right . unTagged
```

Prisms

Reminder – lenses and traversals

In the last lecture, we learned about lenses and traversals.

- A `Lens s a` can be used to "zoom in" on an `a` that is "part" of an `s`.
- A `Traversal s a` "zooms in" on arbitrarily many `a`'s "contained" in `s`.
- Lenses and traversals can be composed with each other. Composing two lenses gives a lens again. Composing two traversals or a traversal and a lens gives a traversal.
- We can `set` and `view` the part(s) in focus, list them with `toListOf`, and we can apply a function to them via `over`.
- Effectful updates are also possible (and baked right into the definition).

Function `toListOf` accumulates all focused elements in the list monoid. By using the `First` monoid instead, we get `Just` the first focused element or `Nothing` if no element is focused:

```
preview :: (( a -> Const (First a) a)
           -> s -> Const (First a) s)
           -> s -> Maybe a
preview f s =
  getFirst
    (getConst (f (Const . First . Just) s))
```

Function `toListOf` accumulates all focused elements in the list monoid. By using the `First` monoid instead, we get `Just` the first focused element or `Nothing` if no element is focused:

```
GHCi> preview both ('x', 'y')
Just 'x'
GHCi> preview each []
Nothing
GHCi> preview _2 (42, False)
Just False
```

Lenses "zoom into" a factor of a product. What if we have a sum type instead? For example, consider the following type:

```haskell
data Result a = Ok a | Error String
```

We can't create lenses corresponding to the constructors `Ok` and `Error`. We need a new type of optics instead, a prism. A prism differs from a lens in two ways:

- The focused part of a prism is not always present.
- Given a part, the whole can be created from it.

In a sense that could be made precise, prisms are "dual" to lenses in the same way that coproducts/sums are dual to products.

Our plan

We want to understand prisms following steps similar to what we did for lenses:

- Give a concrete, "naive" definition of prisms.
- Explore how to compose prisms and thus turn those "naive" prisms into a category.
- Rewrite prisms in the van Laarhoven style, putting them into a form that allows us to freely compose prisms with lenses and traversals.

Where a (naive) lens was a getter/setter pair, a "naive" prisms
is a preview/review pair:

```haskell
data Prism s a = Prism
  { preview :: s -> Maybe a
  , review  :: a -> s
  }
```

```haskell
prism :: (s -> Maybe a) -> (a -> s) -> Prism s a
prism = Prism
```

A prism `p :: Prism s a` should obey the following two laws:

- review preview: If preview finds something, then reviewing that should reconstruct the original:
  If `preview p s = Just a`, then `review p a = s`.
- preview review: Previewing a reviewed `a` should find that `a`:
  `preview p (review p a) = Just a`.

Examples of prisms

Let us write a couple of example prisms…

```
_Ok :: Prism (Result a) a
_Ok = prism pr Ok
 where
   pr (Ok a)    = Just a
   pr (Error _) = Nothing
```

```
_Error :: Prism (Result a) String
_Error = prism pr Error
 where
   pr (Ok _)    = Nothing
   pr (Error e) = Just e
```

Trying our prisms

…and let us try them:

```
GHCi> preview _Ok (Ok 'x')
Just 'x'
GHCi> preview _Ok (Error "error")
Nothing
GHCi> preview _Error (Ok True)
Nothing
GHCi> preview _Error (Error "oups")
Just "oups"
```

```
GHCi> review _Ok 42
Ok 42
GHCi> review _Error "fail"
Error "fail"
```

Writing lenses for the fields of a record type by hand was easy, but tedious, and the same applies for writing prisms for sum types by hand.

Same as lenses, prisms can also be generated automatically, using Template Haskell or datatype-generic programming.

Naming Convention

Note that naming of prisms is "dual" to the naming of lenses: The constructor names have no underscore, the corresponding prisms do.

Exercise: some common prisms

As a quick exercise, please implement the following prisms:

```
_Just    :: Prism (Maybe a) a
_Nothing :: Prism (Maybe a) ()
_Left    :: Prism (Either a b) a
_Right   :: Prism (Either a b) b
_Nil     :: Prism [a] ()
_Cons    :: Prism [a] (a, [a])
_Id      :: Prism a a
```

Also write a function to compose two prisms:

```
compose :: Prism a x -> Prism s a -> Prism s x
```

Solution to the exercise

```haskell
_Just :: Prism (Maybe a) a
_Just = prism id Just

_Nothing :: Prism (Maybe a) ()
_Nothing = prism
  (maybe (Just ()) (const Nothing))
  (const Nothing)
```

Solution to the exercise

```haskell
_Left :: Prism (Either a b) a
_Left = prism
  (either Just (const Nothing))
  Left
```

```haskell
_Right :: Prism (Either a b) b
_Right = prism
  (either (const Nothing) Just)
  Right
```

Solution to the exercise

```haskell
_Nil :: Prism [a] ()
_Nil = prism pr (const [])
 where
   pr []      = Just ()
   pr (_ : _) = Nothing
```

```haskell
_Cons :: Prism [a] (a, [a])
_Cons = prism pr (uncurry (:))
 where
   pr []       = Nothing
   pr (x : xs) = Just (x, xs)
```

```
_Id :: Prism a a
_Id = prism Just id
```

```
compose :: Prism a x -> Prism s a -> Prism s x
compose ax sa = prism
  (preview sa >=> preview ax)
  (review sa . review ax)
```

The category of prisms

Having defined `_Id` and `compose`, we can give a
`Category` instance for prisms:

```
instance Category Prism where
  id  = _Id
  (.) = compose
```

```
GHCi> preview (_Just . _Left . _Ok)
  (Ok (Left (Just 'x')))
Just 'x'
GHCi> review (_Nothing . _Right . _Ok) ()
Ok (Right Nothing)
```

Recall the formulation of lenses and traversals in van Laarhoven form:

```
type Lens s a      = forall f . Functor    f => (a -> f a) -> s -> f s
type Traversal s a = forall f . Applicative f => (a -> f a) -> s -> f s
```

Each `Prism s a` should be a `Traversal s a`, but neither is every lens a prism, nor is every prism a lens (can you think of one that is both?).

How can we possibly fit prisms into that scheme? There seems to be too little "wiggle room".

Let us write down the definitions of `Lens s a` and `Traversal s a` again, but this time in a slightly weird form:

```
type Lens s a      = forall f . Functor     f => (->) a (f a) -> (->) s (f s)
type Traversal s a = forall f . Applicative f => (->) a (f a) -> (->) s (f s)
```

Writing it like this makes us realize that there is indeed one more thing we can change: Instead of using `(->)`, we can use other profunctors.

It turns out that the correct definition for van Laarhoven prisms is the following:

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

This definition is not at all obvious, but we will see soon that it actually works.

One thing you can see already, by just looking at the constraints, is that both lenses and prisms are traversals and that lenses and prisms are – in general – incomparable.

First let's try to understand how to get a van Laarhoven prism, given a way to preview and to review:

```
prism :: (s -> Maybe a) -> (a -> s) -> Prism s a
```

So given functions `pr :: s -> Maybe a` and `rv :: a -> s` and some `p a (f a)` for a profunctor `p` implementing `Choice` and an applicative functor `f`, we have to produce a `p s (f s)`.

This is actually easier than it looks – the fact that we know nothing about `p` and `f` means we are confined to using `dimap`, `left'` and `right'`.
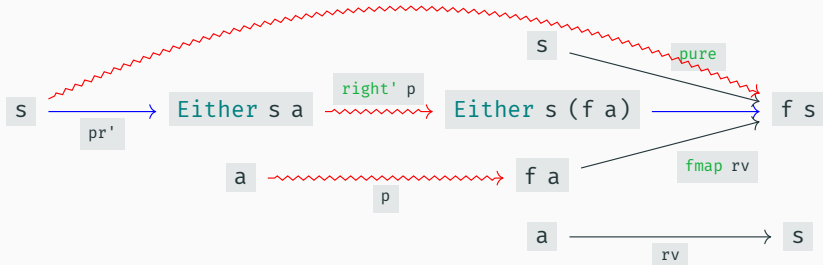
```
prism :: (s -> Maybe a) -> (a -> s) -> Prism s a
prism pr rv p =
  dimap pr' (either pure (fmap rv)) (right' p)
  where
    pr' s = case pr s of
      Nothing -> Left s
      Just a  -> Right a
```

Given a van Laarhoven prism, we want to be able to `preview` and to `review`.

The first is already done: We have defined `preview` for *traversals* earlier today, and every prism is in particular a traversal.

For `review`, we use the `Tagged` profunctor and the `Identity` functor:

```
review :: Prism s a -> a -> s
review p a = runIdentity
  (unTagged (p (Tagged (Identity a))))
```

From van Laarhoven prisms to naive prisms

Given a van Laarhoven prism, we want to be able to `preview` and to `review`.

The first is already done: We have defined `preview` for *traversals* earlier today, and every prism is in particular a traversal.

Remembering our lessons from the last lecture, we losen that signature to:

```
review :: (Tagged a (Identity a)
        ->  Tagged s (Identity s))
        -> a -> s
review p a = runIdentity
  (unTagged (p (Tagged (Identity a))))
```

Composability of prisms

The definitions of our example prisms `_Ok`, `_Error`,
`_Just`, `_Nothing`, `_Left`, `_Right`, `_Nil`, `_Cons` and
`_Id` stay literally the same. Now prisms are freely
composable with eachother and with lenses and traversables:

```
GHCi> set (_1 . _Just . each)
  (Just "Addis Ababa", 42) 'x'
(Just "xxxxxxxxxxx", 42)
GHCi> set (each . _Left . _2)
  [Left (True, 8), Right 'y', Left (False, 7)] 0
[Left (True, 0), Right 'y', Left (False, 0)]
```

Isos

Now we have learned about three types of optics, lenses, traversals and prisms. They relate to each other as follows:
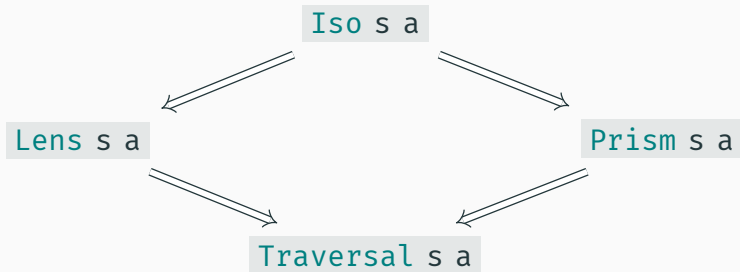

Our goal in this section is to understand a fourth type of optics, isos, which will complete that diagram's upper half: An iso is *both* a lens and a prism.

After all the work we have done up till now, defining isos will be relatively easy.

Isos

We want isos to be both lenses and prisms:

Because an iso is a lens, we have:

```
view :: Iso s a -> s -> a
```

Because an iso is also a prism, we also have:

```
review :: Iso s a -> a -> s
```

An `Iso s a` zooms from an `s` into an "isomorphic" part of type `a`, where the isomorphism is witnessed by `view` and `review`.

Therefore an iso `i :: Iso s a` should obey the following two laws:

- review view: `review i (view i s) = s`.
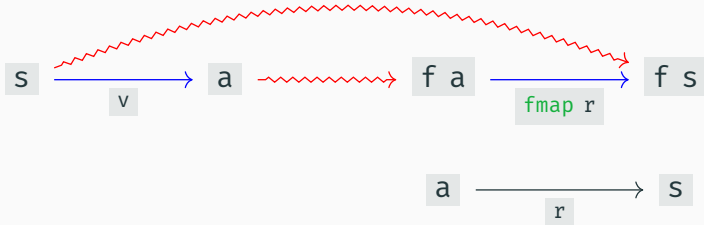- view review: `view i (review i a) = a`.

This time, we will *not* define "naive" isos first and then move on to van Laarhoven style. Instead, looking at the constraints defining lenses and prisms, we can immediately define

```
type Iso s a = forall f p . (Functor f, Choice p)
  => p a (f a) -> p s (f s)
```

As analogues to `lens` and `prism`, we also want a function

```
iso :: (s -> a) -> (a -> s) -> Iso s a
```

We will define this next.

```
iso :: (s -> a) -> (a -> s) -> Iso s a
iso v r = dimap v (fmap r)
```

Given an `Iso s a`, we also want to be able to get its inverse, an `Iso a s`:

```
re :: Iso s a -> Iso a s
re i = iso (review i) (view i)
```

Iso example

As an example, consider the following iso:

```
maybeToEither :: Iso (Maybe a) (Either () a)
maybeToEither = iso
  (maybe (Left ()) Right)
  (either (const Nothing) Just)
```

```
GHCi> view maybeToEither (Just 'x')
Right 'x'
GHCi> view maybeToEither Nothing
Left ()
GHCi> view (re maybeToEither) (Right True)
Just True
GHCi> view (re maybeToEither) (Left ())
Nothing
```

As another quick exercise, please implement the following isos:

```
reversed :: Iso [a] [a]
curried  :: Iso ((a, b) -> c) (a -> b -> c)
flipped  :: Iso (a -> b -> c) (b -> a -> c)
swapped  :: Iso (a, b) (b, a)
swapped' :: Iso (Either a b) (Either b a)
```

Iso exercise solutions

```haskell
reversed :: Iso [a] [a]
reversed = iso reverse reverse
```

```haskell
curried :: Iso ((a, b) -> c) (a -> b -> c)
curried = iso curry uncurry
```

```haskell
flipped :: Iso (a -> b -> c) (b -> a -> c)
flipped = iso flip flip
```

```haskell
swapped :: Iso (a, b) (b, a)
swapped = iso swap swap
  where
    swap (a, b) = (b, a)
```

```haskell
swapped' :: Iso (Either a b) (Either b a)
swapped' = iso swap swap
  where
    swap = either Right Left
```

Polymorphic Updates

Comparing **traverse** and our **Traversal** …

```
traverse ::
  ( Traversable t
  , Applicative f
  )                      => (a -> f b) -> (t a) -> f (t b)
type Traversal s a =
  forall f . Applicative f => (a -> f a) -> s      -> f s
```

… we can't help but notice that **traverse** is more general than our **Traversal** : With **traverse** , you can change the type, with **Traversal** you cannot.

Comparing **traverse** and our **Traversal** …

```
traverse ::
  ( Traversable t
  , Applicative f
  )                        => (a -> f b) -> (t a) -> f (t b)
type Traversal s a =
  forall f . Applicative f => (a -> f a) -> s      -> f s
```

… we can't help but notice that **traverse** is more general than our **Traversal** : With **traverse** , you can change the type, with **Traversal** you cannot.

Let's change that!

We redefine `Traversal` as

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

and rediscover our old type as

```
type Traversal' s a = Traversal s s a a
```

Polymorphic updates for the other optics

We similarly redefine `Lens`, `Prism` and `Iso` as

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

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

```
type Iso s t a b = forall f p . (Functor f, Choice p)
  => p a (f b) -> p s (f t)
```

and again rediscover our old types as

```
type Lens'  s a = Lens  s s a a
type Prism' s a = Prism s s a a
type Iso'   s a = Iso   s s a a
```

So now each of our optics is parameterized by *four* type parameters `s`, `t`, `a` and `b`.

The meaning of those is the same in all cases:

An optic zooms in from type `s` to elements of type `a`, and if we change the focussed elements to type `b`, the final result is of type `t`.

As before, we have functions `iso`, `lens` and `prism` to create optics.

Two of those, `iso` and `lens` won't come as a surprise, their signatures change, but their implementations stay literally the same:

```
lens :: (s -> a) -> (s -> b -> t) -> Lens s t a b
lens gt st f s = st s <$> f (gt s)
```

```
iso :: (s -> a) -> (b -> t) -> Iso s t a b
iso sa as = dimap sa (fmap as)
```

Creating prisms for polymorphic updates

The signature of `prism` for the creation of prisms, though, may come as a surprise:

```
prism :: (s -> Either t a) -> (b -> t) -> Prism s t a b
prism pr rv p =
  dimap pr (either pure (fmap rv)) (right' p)
```

If no `a` is found in the given `s`, we still need to create a `t`, so a `Maybe a` is not sufficient anymore – we now need an `Either t a`.

Note, however, that the *implementation* of `prism` has actually become simpler – `Either t a` plays nicer with `Choice` than `Maybe a` does.

For the special case `s = t`, we can still regain our old convenience by defining

```haskell
prism' :: (s -> Maybe a) -> (a -> s) -> Prism' s a
prism' pr = prism pr'
  where
    pr' s = case pr s of
      Nothing -> Left s
      Just a  -> Right a
```

Apart from this, most of our functions can keep their implementations, but gain greater generality.

Polymorphic update examples

```
GHCi> set (both . each) ([1, 2, 3], [4, 5]) 'x'
("xxx", "xx")

GHCi> over (each . swapped' . _Left) length
  [Right "Addis Ababa", Left True, Right "x"]
[Right 11, Left True, Right 1]

GHCi> over reversed (zip [1..]) "Addis Ababa"
[ ('A', 11), ('d', 10), ('d', 9), ('i', 8)
, ('s', 7), (' ', 6), ('A', 5), ('b', 4)
, ('a', 3), ('b', 2), ('a', 1)]
```