# A look through the lens



Alex Gryzlov
Adform

2015-10-21

# Functional programming

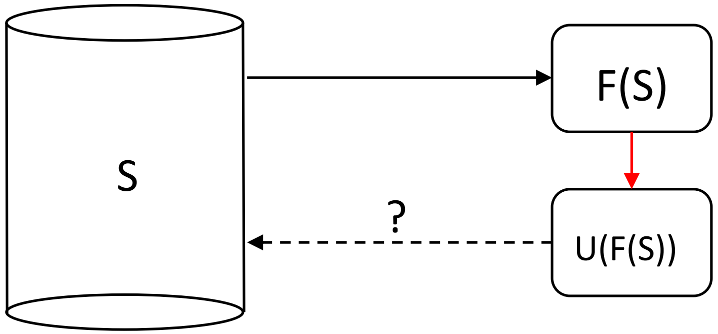
- Declarativity (laziness, higher-order functions)
- Immutability
- Referential transparency (pure functions)

# Immutability

- Easier reasoning – no hidden state
- Safer and cleaner concurrent programming
- Better caching

# View update problem

- First formulated by E. Codd in 1974
- Some database object S
- Mapped into view state F(S)
- View state is changed to U(F(S))
- How to translate back T(U)(S)?



# Bidirectional Programming

- Let's invent a new style of programming
- Programs that can be run "backwards"
- Interfaces and implementations
- Sources and views

- Pierce & Foster's *Harmony*/*Boomerang*
- *Augeas* config manager (has bindings for *Puppet* and *SaltStack*)

# Intermission - Purescript

- Haskell-like statically typed functional language
- Compiles to JS
- Strict evaluation
- Explicit `forall`s (∀)
- Row types & row polymorphism

```
> let showPerson { first: x, last : y } = y ++ ", " ++ x
> :type showPerson
∀ r. { first :: String, last :: String | r } -> String
```

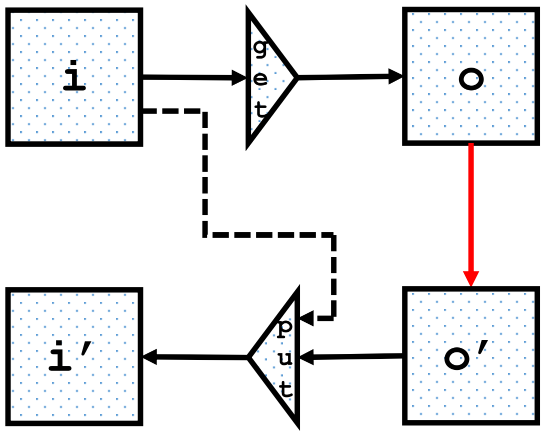
- Effects instead of IO monad

# Bidirectional Programming

Bijectiveness is too strong!

**get** :: ∀ I O. **I** → **O**

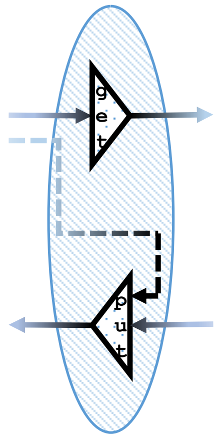
**put** :: ∀ I O. **O** → **I** → **I**




Defining `get` and `put` simultaneously:

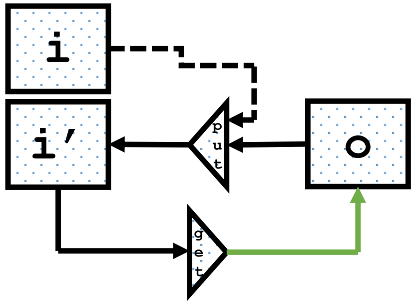
- Infer `put` from `get` somehow
- Provide `put+get` pairs as building blocks ← *lenses*

  *(we need a type system to help us)*

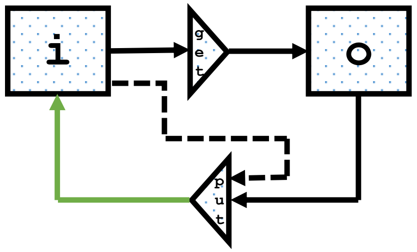


# Some laws

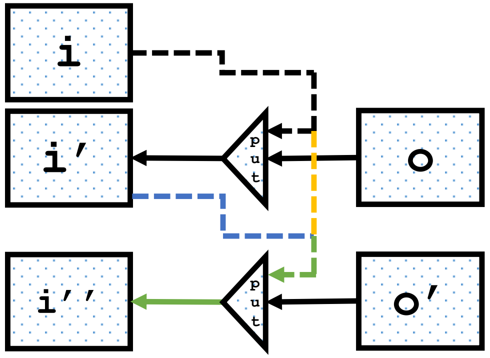
1. `get (put o i) = o`



2. `put (get i) i = i`



3. `put o’ (put o i) = put o’ i`



(last put “wins”/no side effects)

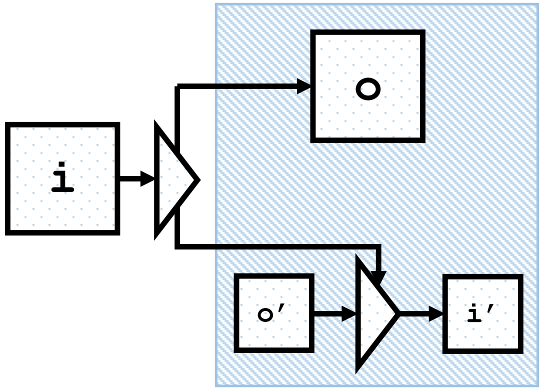
Together with 2nd law it means that updates can be “undone”

`put (get i) (put o i) = i`

# Approach #1 – Store

Let's change the notation a bit

```
Lens i o = (i -> o, o -> i -> i)
           (i -> o, i -> o -> o)
            i ->(o,      o -> i)
```




Store

This is a functor (over `i`):

```
fmap f (Store piece hole) = Store piece (f <<< hole)
```

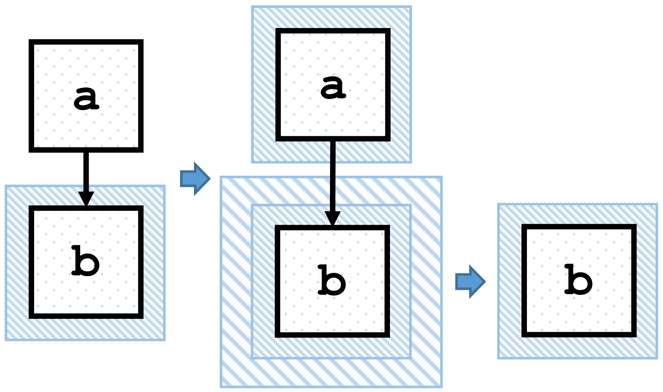
It is also a *comonad*…

# Intermission - comonads

Monad

```
return :: ∀ m a. a       -> m a
join   :: ∀ m a. m (m a) -> m a
```

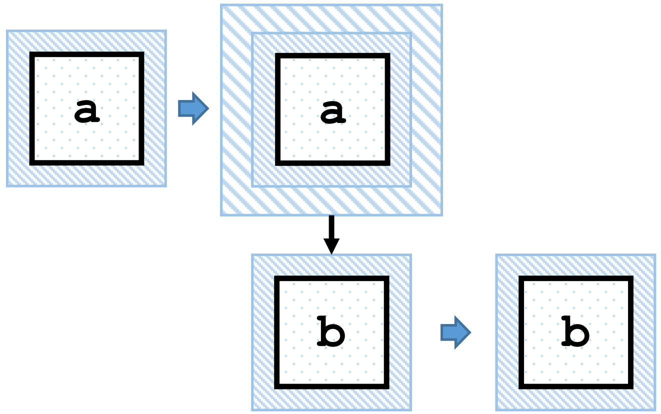
*+laws*




Comonad

```
extract   :: ∀ w a. w a -> a
duplicate :: ∀ w a. w a -> w (w a)
```

*+laws*




Useful for manipulating things that are “endless” or “in a context”

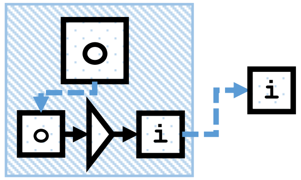
# Approach #1 – Store

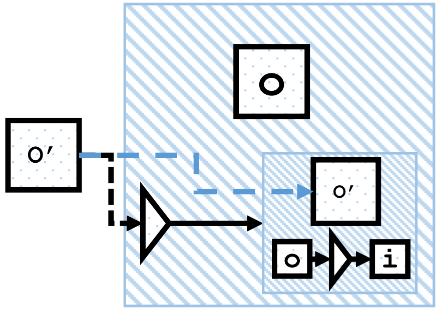
Going back to `Store`...

...It is also a *comonad:*

```
extract (Store piece hole) = hole piece

duplicate (Store piece hole) =
  Store piece (\piece’ -> Store piece’ hole)
```




In category theory, a functor F with a function `a -> F a` (+laws) is called

*F-coalgebra*

Turns out our Lens type `i ->(o, o -> i)` is a

***coalgebra for a Store comonad***

aka *O'Connor lenses*

# Usage - records

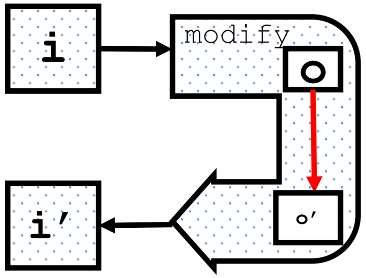
```
petrIvanov :: Person
petrIvanov = Person {
    firstName: "Petr" ,
    lastName: "Ivanov" ,
    address: Address {
        street: StreetRec {
            number: 12 ,
            streetName: "Lenina" ,
            designation: Street
        } ,
    city: "Minsk" ,
    country: Belarus
    }
}
```

# Problems

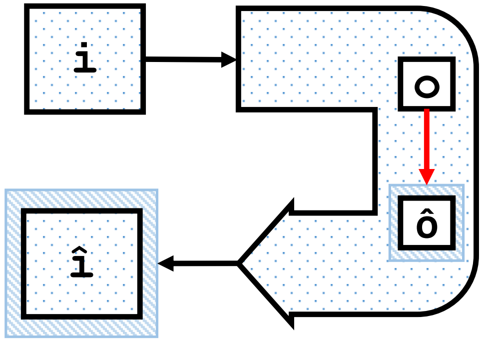
- Not easily composable
- Not too efficient
- Can’t change the type after modification

# Approach #2 – Modify

```
modify :: ∀ I O. (O -> O) -> I -> I
modify f i = put (f (get i)) i
```




```
type Lens I Î O Ô =
  ∀ F. (Functor F) => (O -> F Ô) -> I -> F Î
```

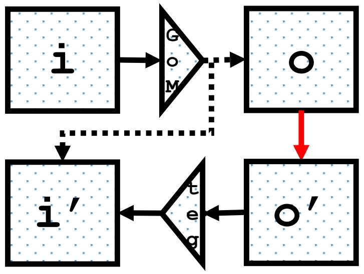



aka *van Laarhoven lenses*

# The Zoo

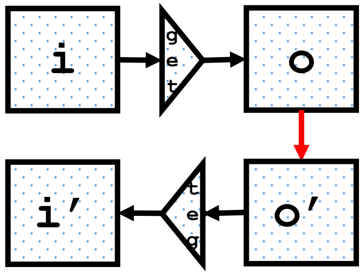
- Prism

```
getOrModify :: ∀ I O. I -> Either I O
reverseGet  :: ∀ I O. O -> I
```



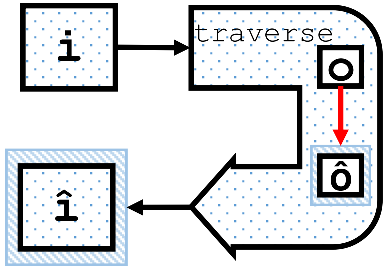

- Iso

```
get         :: ∀ I O. I -> O
reverseGet  :: ∀ I O. O -> I
```



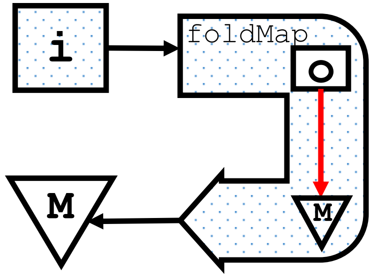

- Traversal

```
traverse :: ∀ F I O. (Applicative F) => (O -> F O) -> I -> F I
```




- Fold

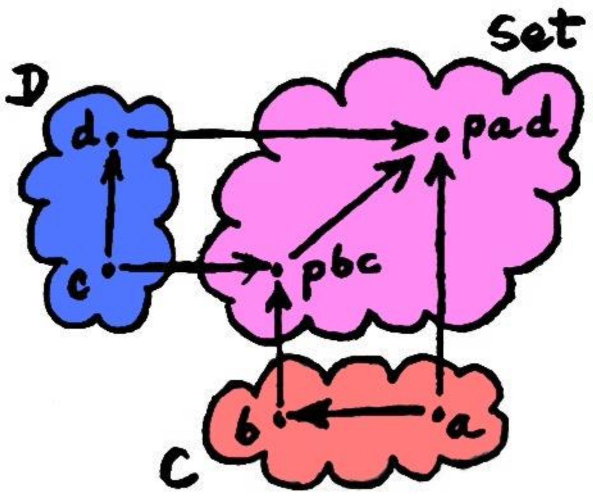
```
foldMap :: (Monoid M) => (O -> M) -> I -> M
```



# Usage – traversals & prisms

```
foo :: Array (Tuple
                (Either Int (Maybe String))
                (Maybe Boolean)
             )
foo   = [
        Tuple (Left 1) Nothing ,
        Tuple (Left 2) (Just true) ,
        Tuple (Right (Just "three")) (Just true) ,
        Tuple (Right (Just "four")) (Just false) ,
        Tuple (Right Nothing) Nothing
       ]
```

# Approach #3 – Profunctor




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

class (Profunctor p) <= Strong p where
   first :: ∀ a b c. p a b -> p (Tuple a c) (Tuple b c)
   second :: ∀ a b c. p b c -> p (Tuple a b) (Tuple a c)

class (Profunctor p) <= Choice p where
   left :: ∀ a b c. p a b -> p (Either a c) (Either b c)
   right :: ∀ a b c. p b c -> p (Either a b) (Either a c)

newtype Star f a b = Star (a -> f b)
instance profunctorStar :: (Functor f) => Profunctor (Star f) where
   dimap f g (Star ft) = Star (f >>> ft >>> map g)

class (Strong p, Choice p) <= Wander p where
   wander :: ∀ s t a b. (∀ f. (Applicative f) => (a -> f b) -> s -> f t)
          -> p a b -> p s t
```

# Approach #3 – Profunctor

```
type Optic p s t a b = p a b -> p s t

type Iso s t a b = ∀ p. (Profunctor p) => Optic p s t a b

type Lens s t a b = ∀ p. (Strong p) => Optic p s t a b

type Prism s t a b = ∀ p. (Choice p) => Optic p s t a b

type Traversal s t a b = ∀ p. (Wander p) => Optic p s t a b

type Fold r s t a b = Optic (Star (Const r)) s t a b
```

# Usage - UI

- UI is a profunctor (actually a monad ☹)
- Components access a local view model
- Child components can be embedded in a bigger component using lenses and traversals that focus on the respective sub-states
- Components provide a handler function that, given a new state, triggers an update of the UI and generates a view that is finally rendered using virtual-dom

# Other usages

- Monomorphic containers
- DB views
- GADTs
- …

# Links

- http://www.janis-voigtlaender.eu/papers/ThreeComplementaryApproachesToBidirectionalProgramming.pdf
- http://www.seas.upenn.edu/~harmony/

- http://artyom.me/lens-over-tea-1

- http://zrho.me/posts/2015-08-23-optic-ui.html
- http://www.haskellforall.com/2015/10/explicit-is-better-than-implicit.html