haskell系列教程 II

by 韩冬@滴滴FP

- 函数式编程基本套路
- 函子抽象
- 透镜组

函数组合

```
(.) :: (b -> c) ->(a -> b) -> a -> c
f . g x = f (g x)
infixr 9 .

($) :: (a -> b) -> a -> b
f $ x = f x
infixr 0 $

-- 方便利用优先级省略括号，下面写法都可以
unlines (map reverse (lines ( "hello\nworld")))
(unlines . map reverse . lines) "hello\nworld"
unlines $ map reverse $ lines $ "hello\nworld"
unlines . map reverse . lines $ "hello\nworld"
-- "olleh\ndlrow\n"

reverseByLine :: String -> String
reverseByLine = unlines . map reverse . lines
```

常见函数

```
id :: a -> a       -- 直接返回参数
id x = x

const :: a -> b -> a          -- 忽略第二个参数
const x y = x

flip :: (a -> b -> c) -> b -> a -> c  -- 交换参数的顺序
flip f x y = f y x

flip map [1..10] $ \ x -> ...
-- (`map` [1..10]) $ \ x -> ...

(&) :: a -> (a -> b) -> b   -- flip ($)
x & f = f x
infixl 1 &

"hello\nworld" & lines & map reverse & unlines
-- "olleh\ndlrow\n"
```

递归

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

xs :: [Int]
xs = 1 : 1 : zipWith (+) xs (tail xs)

------------------------------------------------
data BinTree a = Nil | Node a (BinTree a) (BinTree a)

countNode :: BinTree a -> Int
countNode Nil = 0
countNode (Node _ left right) =
    countNode left + 1 + countNode right

countNode
  (Node 1 (Node 2 Nil Nil) (Node 3 Nil (Node 4 Nil Nil)))
-- 4
```

foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc [] = acc
foldr f acc (x:xs) = x `f` foldr f acc xs

foldr (+) 0 [1,2,3]
-- 1 + (foldr (+) 0 [2,3])
-- 1 + (2 + foldr (+) 0 [3])
-- 1 + (2 + (3 + foldr (+) 0 []))
-- 1 + ( 2 + (3 + 0))
--------------------------------------------------
max :: (Ord a) => a -> a -> a
max x y | x >= y    = x
        | otherwise = y

maximum :: (Ord a) => [a] -> a
maximum [] = error "empty list"
maximum (x:xs) = foldr max x xs
```

foldr

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + count xs

length = foldr (const (+1)) 0

const :: a -> b -> a
const :: (Int -> Int) -> b -> Int -> Int
const (+1) == \ x y -> y + 1
-----------------------------------------------------
all :: (a -> Bool) -> [a] -> Bool
all f = foldr ((&&) . f) True
all f xs = foldr (\ x acc -> f x && acc) True xs

all even [1..]
-- even 1 && (even 2 && (even 3 && ..
-- False
```

foldr

```
map f = foldr ((:) . f) []

map id [1,2,3]
-- foldr ((:) . id) []
-- foldr (:) [] [1,2,3]
-- 1 : (foldr (:) [] [2,3])
-- 1 : (2 : foldr (:) [] [3])
-- 1 : (2 : (3 : foldr (:) [] []))
-- 1 : ( 2 : (3 : []))

     (:)  --- foldr f z ---> f
     / \                    / \
    1  (:)                 1   f
       / \                    / \
      2  (:)                 2   f
         / \                    / \
        3  []                  3   z
```

foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (acc `f` x) xs

foldl (+) 0 [1,2,3]
-- foldl (+) (0 + 1) [2,3]
-- foldl (+) (0 + 1 + 2) [3]
-- foldl (+) (0 + 1 + 2 + 3) []
-- 0 + 1 + 2 + 3

     (:)  --- foldl f z --->    f
     / \                       / \
    1  (:)                    f   3
       / \                   / \
      2  (:)                f   2
         / \               / \
        3  []             z   1
```

thunk leak

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (acc `f` x) xs
```

```
    thunk--x
      |  \
      |   \
      f   thunk--x
            |  \
            |   \
            f   thunk--x
                |  \
                |   \
                f    ...
```

```
seq :: a -> b -> b -- 把对b的求值过程转化为对a和b的求值过程
```

foldl'

```
-- 每次迭代的同时对累计值求值
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f acc [] = acc
foldl' f acc (x:xs) =
    let acc' = acc `f` x
    in acc' `seq` foldl f acc' xs

foldl' (+) 0 [1,2,3]
-- foldl' (+) 1 [2,3]
-- foldl' (+) 3 [3]
-- foldl (+) 6 []
-- 6
-----------------------------------------------------
all' f = foldl' (\ acc x -> acc && f x) True
all' even [1..]
-- (...((True && even 1) && even 2) && even 3 ...
-- Never stop...
```

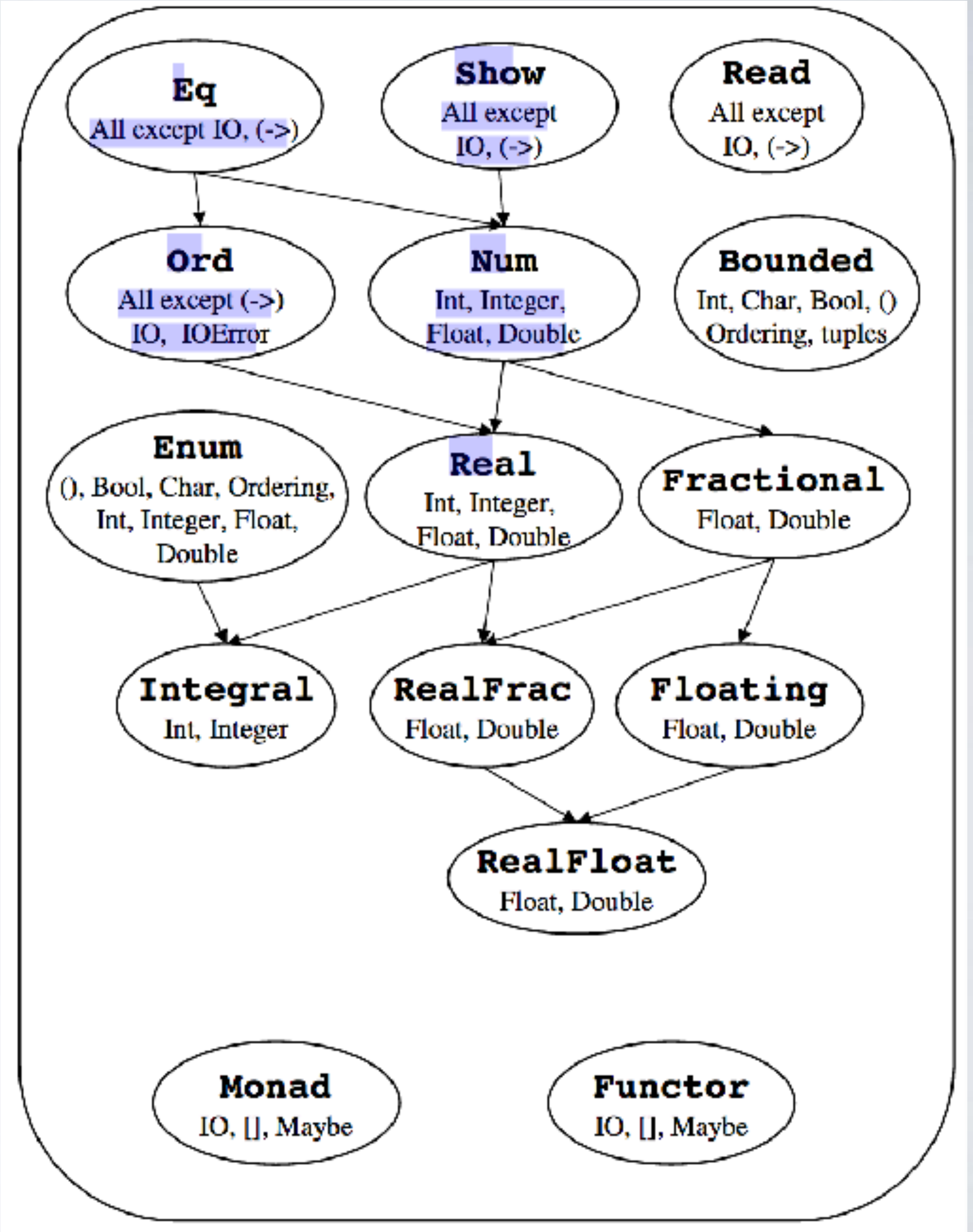
类型类语法

```haskell
class Eq a where
    (==), (/=) :: a -> a -> Bool

    x /= y         = not (x == y)  -- 默认实现
    x == y         = not (x /= y)

instance Eq Double where
    -- (==) :: Double -> Double -> Bool
    a == b = eqDouble# a b

instance Eq a => Eq (Maybe a) where
    (Just a) == (Just b)    =   a == b
    Nothing == Nothing      =   True
    _       == _            =   False
```

类型类语法

```
data Ordering = LT | EQ | GT -- 小于|等于|大于

class Eq a => Ord a where
    compare                :: a -> a -> Ordering
    (<), (<=), (>), (>=)   :: a -> a -> Bool
    max, min               :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger :: Integer -> a

class (Num a, Ord a) => Real a where
    toRational :: a -> Rational

class (Real a, Enum a) => Integral a
where
    quot, rem, div, mod :: a -> a -> a
    divMod, quotRem :: a -> a -> (a, a)
    toInteger :: a -> Integer

class Num a => Fractional a where
    (/) :: a -> a -> a
    recip :: a -> a
    fromRational :: Rational -> a

class Fractional a => Floating a where
    pi :: a
    exp, log, sqrt :: a -> a
    (**), logBase :: a -> a -> a
    sin, cos, tan :: a -> a
    asin, acos, atan :: a -> a
    sinh, cosh, tanh :: a -> a
    asinh, acosh, atanh :: a -> a

...
```



Monoid

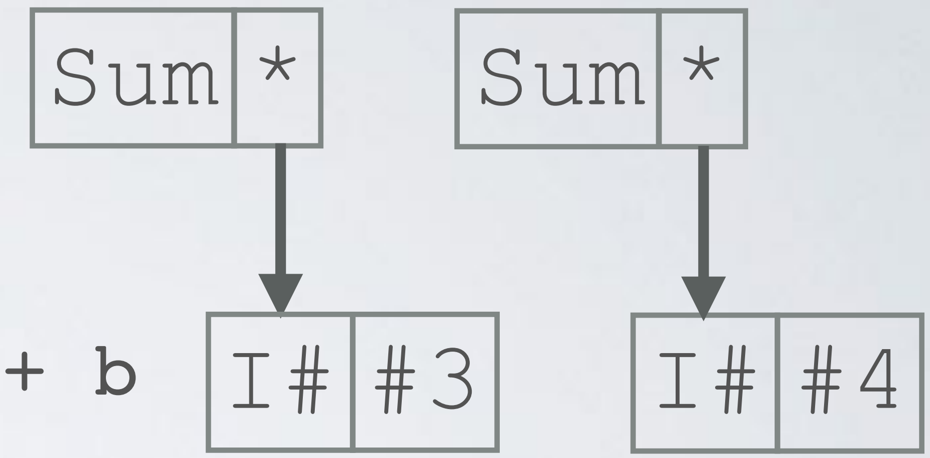
```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [a] -> a
    mconcat = foldr mappend mempty

(<>) = mappend
infixr 6 <>

-- Monoid的满足的定律
mempty <> x == x
x <> mempty == x
(x <> y) <> z == x <> (y <> z)
```

Monoid

```
instance Monoid [a] where
        mempty  = []
        mappend = (++)

instance Monoid Int where
        mempty  = 0
        mappend = (+)

"hello" <> "world" -- "helloworld"
0 <> 1 <> 2 -- 3
-----------------------------------------------
-- 如何选择Int的两种Monoid实例?
instance Monoid Int where
    mempty  = 1
    mappend = (*)
0 <> 1 <> 2 -- 0
```

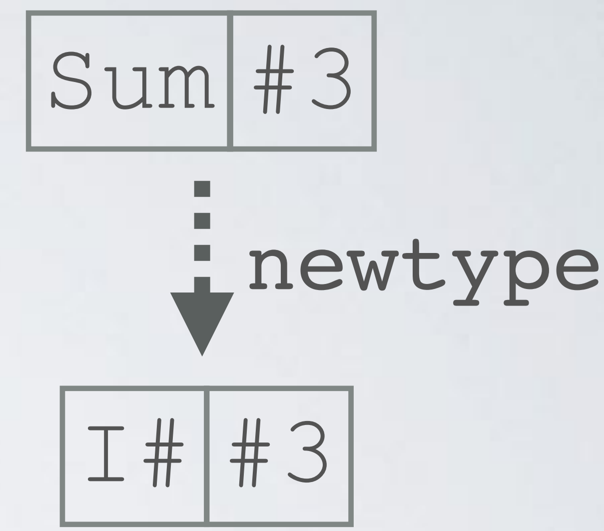
构造新的类型来选择实例

```
data Sum a = Sum {getSum :: a}

instance Num a => Monoid (Sum a)
    mempty = Sum  0
    Sum a `mappend` Sum b = Sum $ a + b

Sum 3 <> Sum 4  == Sum 7
getSum . mconcat . map Sum $ [1..100] == 5050
---------------------------------------------------
data Product a = Product {getProduct :: a}

instance Num a => Monoid (Product a)
    mempty = Product 1
    Product a `mappend` Product b = Product $ a * b

Product 3 <> Product 4  == Product 12
```



newtype

```
newtype Sum a = Sum {getSum :: a}

instance Num a => Monoid (Sum a)
    mempty = 0
    Sum a `mappend` Sum b = Sum $ a + b

Sum 3 <> Sum 4  == Sum 7
```



Sum只存在于编译阶段!

newtype的行为和data类似，但是省去了一层boxing。

因此速度快，而且没有额外的Laziness -- `Sum x == x `seq` Sum x`

newtype的使用条件:

- 新类型只有一个构造函数
- 构造函数只有一个参数

函子Functor

```
-- 函子提供了操作容器内部payload的能力
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    -- fmap :: (a -> b) -> [a] -> [b]
    fmap = map

instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap f (Just a) = Just (f a)
    fmap f Nothing = Nothing

fmap (+1) (Just 0) == Just 1
fmap (+1) Nothing == Nothing
fmap even [1,2,3] == [False, True, False]
```

两个极端的函子

```
-- 什么上下文都没有的Identity
newtype Identity a = Identity{runIdentity :: a}

instance Functor Identity where
    fmap f (Identity a) = Identity (f a)

fmap (+1) (Identity 0) == Identity 1

-- 只包含上下文，没有payload的Const
newtype Const a b = Const{getConst :: a}

instance Functor Const c where
    -- fmap :: (a -> b) -> Const c a -> Const c b
    fmap f (Const c) = Const c

fmap (+1) (Const 0) == Const 0
```

Lens

数据操作的艺术

```haskell
data Position =
    Position{posX :: Double, posY :: Double}

foo :: Position
foo = Position 1 2
```

```
       +---------------+
foo -->| Position      |
       +---------------+
       |   *   |   *   +-+
       +---+---+-------+ |
           |             |
           V             V
       +-----------+   +-----------+
       |  Double   |   |  Double   |
       +-----------+   +-----------+
       |     1     |   |     2     |
       +-----------+   +-----------+
```

bar = foo{posY = 5}

```
       +---------------+
foo -->|    Position   |
       +---------------+
       |   *   |   *   +-+
       +---+---+-------+ |
           |             |
           V             V
       +-----------+   +-----------+  +-----------+
       | ::Double  |   | ::Double  |  | ::Double  |
       +-----------+   +-----------+  +-----------+
       |     3     |   |     4     |  |     5     |
       +-----------+   +-----------+  +-----------+
           ^                                ^
           |                                |
       +---+---+-------+                    |
       |   *   |   *   +--------------------+
       +---------------+
 bar ->|    Position   |
       +---------------+
```

问题?

```
data Line = Line{start :: Position, end :: Position}

lineA = Line (Position 0 0) (Position 3 4)

-- move end’s posY to 5

lineB =
    let lineA’sEnd = end lineA
    in lineA{
        end = lineAs’End{ posY = 5 }
    }

-- 深层次嵌套的不可变数据操作如此麻烦?
-- 我们希望能够使用类似lineA.end.posY = 5的语法
-- 同时保持数据不可变和共享的特性
```

meet lens

```
type Lens b a =
    (Functor f) => (a -> f a) -> b -> f b

posXLens :: Lens Postion Double
posXLens :: (Functor f) =>
  (Double -> f Double) -> Position -> f Position

posXLens f p =
    let x = (posX p)  -- x :: Double
        x′ = f x      -- x′ :: f Double
        — setter :: Double -> Position
        setter = \x -> p{posX = x}
    in
        fmap setter x′    -- f Position
```

lens与functor

```
posXLens :: Lens Postion Double
posXLens :: (Functor f) =>
  (Double -> f Double) -> Position -> f Position

mirror x :: Double -> [Double]
mirror x = [-x, x]

foo = Position 3 4

posXLens mirror foo
== [Position -3 4, Position 3 4]
/= Position [-3,3] 4
```

使用lens获得modifier

```
over :: Lens b a -> (a -> a) -> b -> b
over lens f b =
  let
    idF = Identity . f    -- idF :: a -> Identity a
    idB = lens idF        -- idB :: b -> Identity b
    fb = idB b            -- db :: Identity b
  in
    runIdentity fb        -- b

over lens f = runIdentity . lens (Identity . f)

foo = Position 3 4
over posXLens (+1) foo == Position 4 4
```

使用lens获得setter

```
set :: Lens b a -> a -> b -> b
set lens a b =
  let
    idF = Identity . (\_ -> a)
    idB = lens idF
    fb = idB b
  in runIdentity fb

const :: a -> b -> a
const a _ = a

set lens a
    = runIdentity . lens (Identity . const a)
set posXLens 0 foo == Position 0 4
```

使用lens获得getter

```
view :: Lens b a -> b -> a
Lens b a :: Functor f => (a -> f a) -> b -> f b
```

从f b类型得到a类型? ? `view lens b = ???`

```
view lens b =
  let
    -- Const :: a -> Const a a
    cb = lens Const       -- cb :: b -> Const a b
    ca = cb b             -- ca :: Const a b
  in getConst ca          -- a

view lens = getConst . (lens Const)
view posXLens foo == 3
```

(.)

The power of dot

组合lens

```
posXLens :: Lens Position Double
posXLens :: Functor f =>
  (Double -> f Double) -> Position -> f Position
posXLens f p =
    fmap (\x -> p{posX = x}) (f $posX p)

startLens :: Lens Line Position
startLens :: Functor f =>
  (Position -> f Position) -> Line -> f Line
startLens f l =
    fmap (\s -> l{start = s}) (f $ start l)

(startLens . posXLens) :: Functor f =>
    (Double -> f Double) -> Line -> f Line
(startLens . posXLens) :: Lens Line Double
```

组合lens

```
lineA = Line (Position 0 0) (Position 3 4)

-- move start’s posX to 5
set (startLens . posXLens) lineA 5
-- Line (Position 5 0) (Position 3 4)
```

实际上你并不需要书写每一个Lens，真正的使用大约类似这样：

```
data Profile = Profile {
    _name :: String
,   _address :: Address
,   …
}

mkLenses ‘’Profile
-- now you have name/address lenses
```

中缀函数DSL

```
(%~) = over
-- lens (%~) f = over lens f
infixr 4 %~
lineA & start. posX %~ (+1)

(.~) = set
infixr 4 .~
lineA & start. posX .~ 0

value ^. lens = view lens value
-- (^.) = flip view

infixl 8 ^.
lineA ^. start . posX
```

being first class !

```
triAngleA :: [Line]
triAngleA = [ Line (Position 0 0) (Position 0 4)
            , Line (Position 0 4) (Position 3 0)
            , Line (Position 3 0) (Position 0 0)
            ]

map (^. start) triAngleA
-- [(Position 0 0),(Position 0 4),(Position 3 0)]

foldl' (\ acc x -> acc + x ^. start . posX) 0 triAngleA
-- 3

map (start . posX %~ (+5)) triAngleA
-- [ Line (Position 5 0) (Position 0 4)
  , Line (Position 5 4) (Position 3 0)
  , Line (Position 8 0) (Position 0 0)
  ]
```