# 函数式语言程序设计

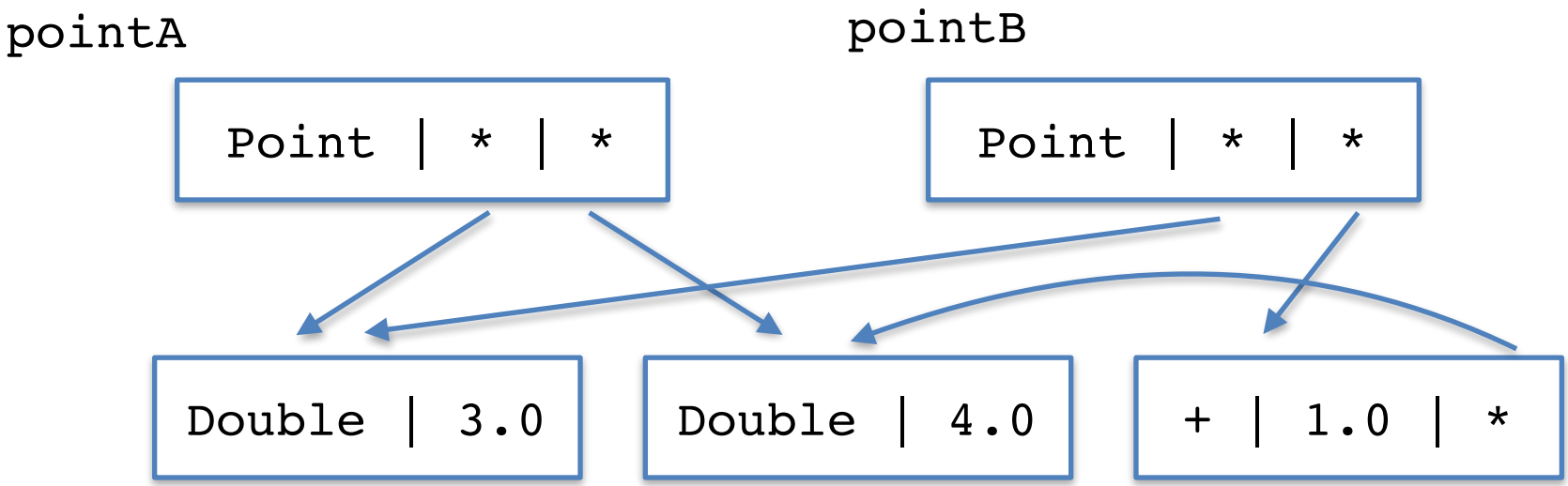
Lens

# 操作不可变数据

- 不可变数据的更新方式是基于原有的数据创建新的数据

```
data Point = Point { x :: Double, y :: Double }

pointA = Point 3.0 4.0
pointB = pointA{ y = y pointA + 1.0 }
```



# 问题?

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

lineA = Line pointA pointB

lineB = lineA{ start = s2 }
  where s1 = start lineA
        s2 = s1{ y = y s1 - 1.0 }

lineB = case lineA of
            Line (Point x y) e ->
                Line (Point x (y - 1.0)) e
```

在操作嵌套数据的时候，我们希望能拥有类似

`lineA . start . x = 0` 这样简洁的语法，How?

# 用函数封装更新操作

```
modifyX :: (Double -> Double) -> Point -> Point
modifyX f (Point x y) = Point (f x) y

modifyStart :: (Point -> Point) -> Line -> Line
modifyStart f (Line s e) = Line (f s) e

modifyStart (modifyX (+1)) $ lineA
```

# 复杂一些的更新操作?

```
multiX :: Double -> [Double]
multiX x = [x+1, x+2, x+3]

modifyXs :: (Double -> [Double]) -> Point -> [Point]
modifyXs f (Point x y) =
    map (\ x' -> Point x' y) (f x)
          Double -> Point    [Double]

maybeX :: Double -> Maybe Double
maybeX = ...
modifyXMaybe :: ???
modifyXMaybe = ???
```

# Functor to rescue!

```
fmodifyX :: Functor f
         => (Double -> f Double) -> Point -> f Point
fmodifyX f (Point x y) =
    fmap (\ x' -> Point x' y) (f x)
           ..........................................     ..........
             Double -> Point   f Double
```

我们利用 fmap 保持容器『形状』的特性，把 modifyXs 扩展到了任意的函子类型上。而不去关心容器具体的上下文含义。

# Functor to rescue!

```
fmodifyX :: Functor f
         => (Double -> f Double) -> Point -> f Point
fmodifyX f (Point x y) =
    fmap (\ x' -> Point x' y) (f x)
           ..........................................    ..........
           Double -> Point   f Double
```

```
newtype Identity a = Identity { runIdentity :: a }

modifyX :: (Double -> Double) -> Point -> Point
modifyX f = runIdentity . fmodifyX (Identity . f)
```

# Van Laarhoven Lens

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

xLens :: Lens Point Double
xLens f (Point x y) =
    fmap (\ x' -> Point x' y) (f x)

yLens = ...

startLens :: Lens Line Point
startLens f (Line s e) =
    fmap (\ s' -> Line s' e) (f s)

endLens = ...
```

# 如何使用 Van Laarhoven Lens?

```
over :: Lens b a -> (a -> a) -> (b -> b)
over l f = runIdentity . l (Identity . f)

(%~) = over
infixr 4 %~
```

```
                (Double -> f Double) -> Point -> f Point
```

```
lineA & startLens . xLens %~ (+1)
```

```
 (Point -> f Point) -> Line -> f Line
```

```
x & f = f x
infixl 1
```

# Functor Const is about the box!

```
-- Const a b 是一个只包含 a 类型数据的盒子
-- 这里 a 类型的数据是盒子的『形状』！
newtype Const a b = Const { getConst :: a }

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


view :: Lens b a -> b -> a
view l = getConst . (l Const)
```

a -> Const a b

Const a b -> a

(a -> f a) -> (b -> b)

# Const + Lens = Elegant getters

```
view l = getConst . (l Const)

view xlens (Point x y)
-- getConst (xlens Const (Point x y))
-- getConst (fmap (\ x' -> Point x' y) (Const x)
-- getConst (Const x)
-- x

(^.) :: b -> Lens b a -> a
b ^. lens = view lens b
infixl 8 ^.

lineA ^. startLens . xLens    -- get line's start's X
```