函数式语言程序设计

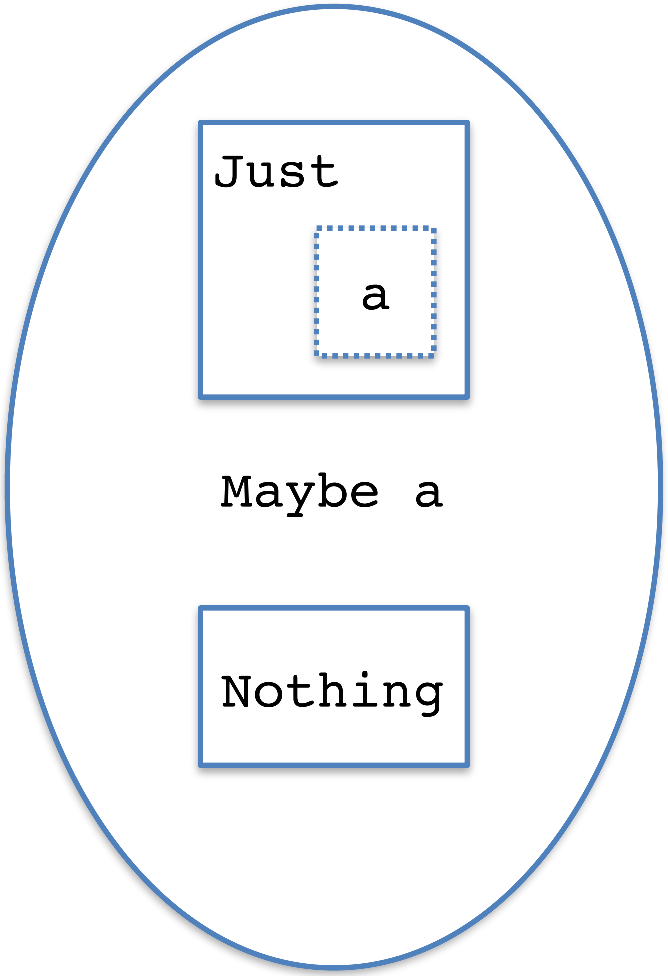
Fun With Functor!

盒子比喻

- Haskell 中数据是放在一个个盒子中的，构造函数负责打包，模式匹配负责解包。
- Haskell 中的类型，是对有哪些可能的盒子的一个保证。
- 实现起来一般是内存中一小块连续的区域。

各种各样的盒子...

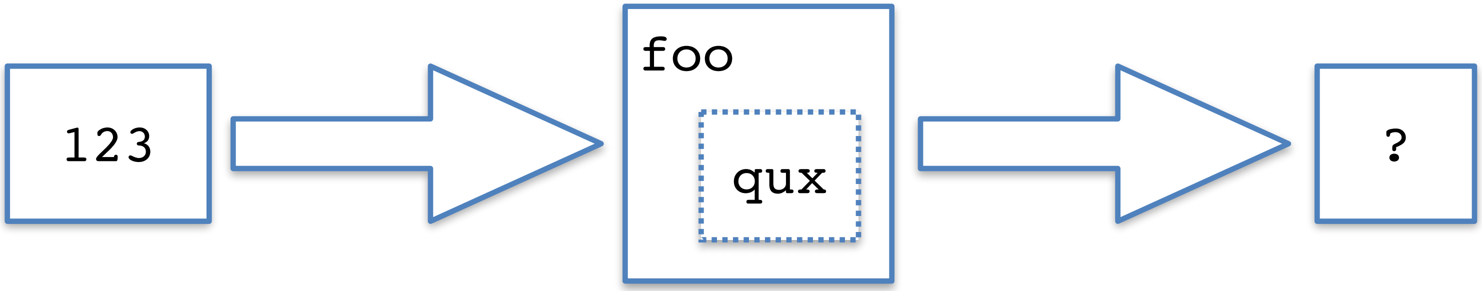
```
data Void
data Bool = True | False
newtype Identity a = Identity a
data Proxy a = Proxy
newtype Const a b = Const a
data Maybe a = Just a | Nothing
data [a] = a : [a] | []
data (a, b) = (a, b)
data Either a b = Left a | Right b
data Bin a = Nil
           | Node { val :: a
                  , left :: Bin a
                  , right :: Bin a}
```



函数盒子

我们也可以把函数也看做一类特除的盒子，他们通过运行代码构造，盒子里封装的是被函数捕获的自由变量，我们无法直接观察到这些自由变量，唯一的办法是传递参数进去，观察他们对返回值的影响。

```
main = do
   ...
   let foo bar = if qux then bar else bar+1
   ...
```



操作盒子里的数据

```haskell
mapIdentity :: (a -> b) -> Identity a -> Identity b
mapIdentity f (Identity x) = Identity (f x)

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a:as) = f a : map f as

mapBin :: (a -> b) -> Bin a -> Bin a
mapBin f Nil = Nil
mapBin f (Bin v l r) = Bin (f v) (mapBin f l)
                                 (mapBin f r)
```

操作盒子里的数据

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just a) = Just (f a)

mapProxy :: (a -> b) -> Proxy a -> Proxy b
mapProxy Proxy = Proxy

mapConst :: (a -> b) -> Const c a -> Const c b
mapConst (Const c) = Const c
```

函子抽象

```
class Functor f where
    fmap :: (a -> b) -- 应用在盒子里的数据上的函数
         -> f a      -- Identity a, [a], Maybe a...
         -> f b


instance Functor Identity where fmap = mapIdentity
instance Functor [] where fmap = map
instance Functor Maybe where fmap = mapMaybe
instance Functor (Const c) where fmap = mapConst
...
```

Functor laws

Functor 是关于『盒子』的抽象，在通过 fmap 操作盒子里面的数据前后，盒子本身的形状应保持不变。

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Reader Functor

让我们来考虑一个 x -> a 类型的函数，我们认为盒子里封装了一个可以根据 x 计算出 a 的计算过程。

```
instance Functor ((->) x) where
    -- fmap :: (a -> b) -> (x -> a) -> (x -> b)
    fmap f g = f . g

-- fmap id g === id . g === g === id g
-- (fmap x . fmap y) g === fmap x (fmap y g)
                       === fmap x (y . g)
                       === x . y . g
                       === fmap (x . y) g
```

Functor is ubiquitous

```
Functor 类型类的约束意味着你可以操作盒子里的数据
Functor f => ... f a ...

(<$>) = fmap
infixl 4 <$>

readEither :: Read a => String -> Either String a

(*1000) <$> readEither "123"
-- Right 123000
```