计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2022年09～12月

Adapted from Graham’s Lecture slides

第12章： Monads and More

主要知识点：

Functor、Applicative, Monad

两种提升代码抽象层次的方式

Level 1: Polymorphic Functions (over types)

```
length1 :: List a -> Int
```

Level 2: Generic Functions (over type constructors)

```
length2 :: t a -> Int
```

Functor / 函子

计算的抽象

```haskell
inc :: [Int] -> [Int]
inc []       = []
inc (n:ns) =  n+1 : inc ns
```

```haskell
sqr :: [Int] -> [Int]
sqr []       = []
sqr (n:ns) =  n^2 : sqr ns
```

抽象

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

```haskell
inc = map (+1)   sqr = map (^1)
```

Functor

```haskell
-- Exported by Prelude
class Functor f where

    fmap :: (a -> b) -> f a -> f b

    (<$) :: b -> f a -> f b
    (<$) = fmap . const
```

```haskell
(fmap . const) b fa
 fmap (const b) fa
```

```haskell
-- Exported by Prelude
const :: b -> a -> b
const x _ = x
```

Functor

```haskell
-- Exported by Prelude
class Functor f where

    fmap :: (a -> b) -> f a -> f b

    (<$) :: a -> f b -> f a
    (<$) = fmap . const
```

```
ghci> fmap (+1) [1,2,3]
[2,3,4]
ghci> fmap (^2) [1,2,3]
[1,4,9]
```

```haskell
-- Exported by Prelude
instance Functor [] where
    -- fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```

```haskell
data Maybe a = Nothing | Just a

instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing  = Nothing
    fmap g (Just x) = Just (g x)
```

```
ghci> fmap (+1) (Just 3)
Just 4
ghci> fmap (+1) Nothing
Nothing
ghci> fmap not (Just False)
Just True
```

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving (Show)

instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
    fmap g (Leaf x)   = Leaf $ g x
    fmap g (Node l r) = Node (fmap g l) (fmap g r)
```

```
ghci> fmap length (Leaf "abc")
Leaf 3
ghci> fmap even $ Node (Leaf 1) (Leaf 2)
Node (Leaf False) (Leaf True)
```

```haskell
instance Functor IO where
    -- fmap :: (a -> b) -> IO a -> IO b
    fmap g mx = do x <- mx
                   return $ g x
```

```
ghci> fmap show $ return True
"True"
```

Generic Function Definition

```haskell
inc :: Functor f => f Int -> f Int
inc = fmap (+1)
```

```
ghci> inc $ Just 1
Just 2
ghci> inc [1,2,3,4,5]
[2,3,4,5,6]
ghci> inc $ Node (Leaf 1) (Leaf 2)
Node (Leaf 2) (Leaf 3)
```

Functor Laws

①
```
fmap id = id
```

②
```
fmap (f . g) = fmap f . fmap g
```

- **For any parameterized type in Haskell, there is *at most one* function fmap that satisfies the required laws.**
  - That is, if it is possible to make a given parameterized type into a functor, there is only one way to achieve this.
  - Hence, the instances that we defined for lists, Maybe, Tree and IO were all uniquely determined.

<$> : An infix synonym for fmap

```haskell
-- Exported by Prelude
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

The name of this operator is an allusion to `$`. Note the similarities between their types:

```haskell
 ($)  ::                    (a -> b) ->   a ->   b
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Whereas `$` is function application, `<$>` is function application lifted over a `Functor`.

Applicative

Applicative Functor

如何定义一个一般性的fmap

```
fmap0 :: a -> f a
```

```
fmap1 :: (a -> b) -> f a -> f b
```

```
fmap2 :: (a -> b -> c) -> f a -> f b -> fc
```

```
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> fc -> f d
```

⋮

两个基本函数

```
pure :: a -> f a
```

```
(<*>) :: f (a -> b) -> f a -> f b
```

```
pure :: a -> f a
```

```
(<*>) :: f (a -> b) -> f a -> f b
```

```
fmap0 :: a -> f a
fmap0 = pure
```

```
fmap1 :: (a -> b) -> f a -> f b
fmap1 g x = pure g <*> x
fmap1 g x = fmap g x = g <$> x
```

```
fmap2 :: (a -> b -> c) -> f a -> f b -> fc
fmap2 g x y = pure g <*> x <*> y = g <$> x <*> y
```

```
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> fc -> f d
fmap3 g x y z = pure g <*> x <*> y <*> z = g <$> x <*> y <*> z
```

Applicative Functor

Applicative Functor: 一个简化版本

```haskell
class Functor f => Applicative f where
    -- Lift a value
    pure :: a -> f a

    -- Sequential application.
    (<*>) :: f (a -> b) -> f a -> f b
```

Applicative Functor: 一个简化版本

```haskell
class Functor f => Applicative f where
   -- Lift a value
    pure :: a -> f a
   -- Sequential application.
    (<*>) :: f (a -> b) -> f a -> f b
```

声明 Maybe为Applicative的一个实例

```haskell
instance Applicative Maybe where
   -- pure :: a -> Maybe a
   pure = Just

   -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
   Nothing  <*> _    = Nothing
   (Just g) <*> mx   = g <$> mx
```

Applicative Functor: –

```
class Functor f => Appli
   -- Lift a value
    pure :: a -> f a
   -- Sequential applica
    (<*>) :: f (a -> b)
```

```
ghci> pure (+1) <*> Just 1
Just 2
ghci> pure (+) <*> Just 1 <*> Just 2
Just 3
ghci> pure (+) <*> Nothing <*> Just 2
Nothing
ghci> Nothing <*> Just 1
Nothing
```

声明 Maybe为Applicative的一个实例

```
instance Applicative Maybe where
   -- pure :: a -> Maybe a
   pure = Just

   -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
   Nothing  <*> _    = Nothing
   (Just g) <*> mx   = g <$> mx
```

声明 [ ] 为Applicative的一个实例

```haskell
instance Applicative [] where
    -- pure :: a -> [a]
    pure x = [x]

    -- (<*>) :: [a -> b] -> [a] -> [b]
    gs <*> xs = [g x | g <- gs, x <- xs]
```

```
ghci> pure (+1) <*> [1,2,3]
[2,3,4]
ghci> pure (+) <*> [1] <*> [2]
[3]
ghci> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

声明 IO 为Applicative的一个实例

```haskell
instance Applicative IO where
  -- pure :: a -> IO a
  pure = return

  -- (<*>) :: IO (a -> b) -> IO a -> IO b
  mg <*> mx = do {g <- mg; x <- mx; return (g x)}
```

```haskell
getChars :: Int -> IO String
getChars 0 = return []
getChars n = pure (:) <*> getChar <*> getChars (n-1)
```

Generic Function Definition

```haskell
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = pure (:) <*> x <*> sequenceA xs
```

```
ghci> sequenceA [Just 1, Just 2, Just 3]
Just [1,2,3]
ghci> sequenceA [Just 1, Nothing, Just 3]
Nothing
ghci> sequenceA [[1,2,3], [4,5,6], [7,8,9]]
[[1,4,7],[1,4,8],[1,4,9],[1,5,7],[1,5,8],[1,5,9],[1,6,7],[1,6,8],[1,6,9],
[2,4,7],[2,4,8],[2,4,9],[2,5,7],[2,5,8],[2,5,9],[2,6,7],[2,6,8],[2,6,9],
[3,4,7],[3,4,8],[3,4,9],[3,5,7],[3,5,8],[3,5,9],[3,6,7],[3,6,8],[3,6,9]]
```

Applicative Laws

①
```haskell
pure id <*> x = x
```

②
```haskell
pure (g x) = pure g <*> pure x
```

③
```haskell
x <*> pure y = pure (\g -> g y) <*> x
```

④
```haskell
x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

Applicative Laws: 类型分析

```
pure id <*> x = x
```

```
a -> a        f a
```

```
pure (g x) = pure g <*> pure x
```

```
f b        a -> b        a
```

Applicative Laws: 类型分析

```
x <*> pure y = pure (\g -> g y) <*> x
```

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

```
x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

Applicative Laws: 类型分析

```
x <*> pure y = pure (\g -> g y) <*> x
```

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

```
f (b -> c)    f (a -> c)
```

```
x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

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

Monad

一个小问题：异常处理

```haskell
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Int
eval (Val n)     = n
eval (Div x y) = eval x `div` eval y
```

```
ghci> eval $ Div (Val 1) (Val 0)
*** Exception: divide by zero
```

解决方法1

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = case eval x of
                   Nothing -> Nothing
                   Just n  -> case eval y of
                     Nothing -> Nothing
                     Just m  -> safediv n m
```

稍显繁杂

解决方法2

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n)     = pure n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

类型错误

type: Maybe (Maybe Int)

解决方法2

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = pure n
eval (Div x y) = case pure safediv <*> eval x <*> eval y of
                   Just r -> r
                   Nothing -> Nothing
```

Maybe (Maybe Int)

还是不够简洁

解决方法3：引入一个新的操作 bind

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x  -> f x
```

Maybe Int

```haskell
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Div x y)   = eval x >>= (\n -> (eval y >>= (\m -> safediv n m)))
```

Maybe Int　Int　Maybe Int　Int　Maybe Int

Maybe Int

解决方法3：引入一个新的操作 bind

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x  -> f x
```

```haskell
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Div x y)   = eval x >>= (\n -> (eval y >>= (\m -> safediv n m)))
```

‖ 先耍一点朝三暮四的小把戏

```haskell
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Div x y)   = eval x >>= \n ->
                   eval y >>= \m ->
                   safediv n m
```

= 再撒一点扑朔迷离的语法糖

```haskell
eval :: Expr -> Maybe Int
eval (Val n)     = Just n
eval (Div x y)   = do n <- eval x
                      m <- eval y
                      safediv n m
```

Monad

```haskell
{- The Monad class defines the basic operations over a monad,
   a concept from a branch of mathematics known as "category theory".
   From the perspective of a Haskell programmer, however,
   it is best to think of a monad as an abstract datatype of actions.
   The do expressions provide a convenient syntax for writing monadic expressions.-}
class Applicative m => Monad m where

    -- Inject a value into the monadic type.
    return :: a -> m a
    return = pure

    -- Sequentially compose two actions,
    -- passing any value produced by the first as an argument to the second.
    (>>=) :: m a -> (a -> m b) -> m b

    -- Sequentially compose two actions, discarding any value produced by the first,
    -- like sequencing operators (such as the semicolon) in imperative languages.
    (>>) :: m a -> m b -> m b
    m >> k = m >>= \_ -> k
```

```haskell
a >>= f  =  do v <- a
               f v
```

```haskell
a >> b  =  do a
              b
        =  do v <- a
              b
```

声明 Maybe 为Monad的一个实例

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    m >> k = m >>= \_ -> k
```

```haskell
instance Monad Maybe where
    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing  >>= _ = Nothing
    (Just x) >>= f = f x
```

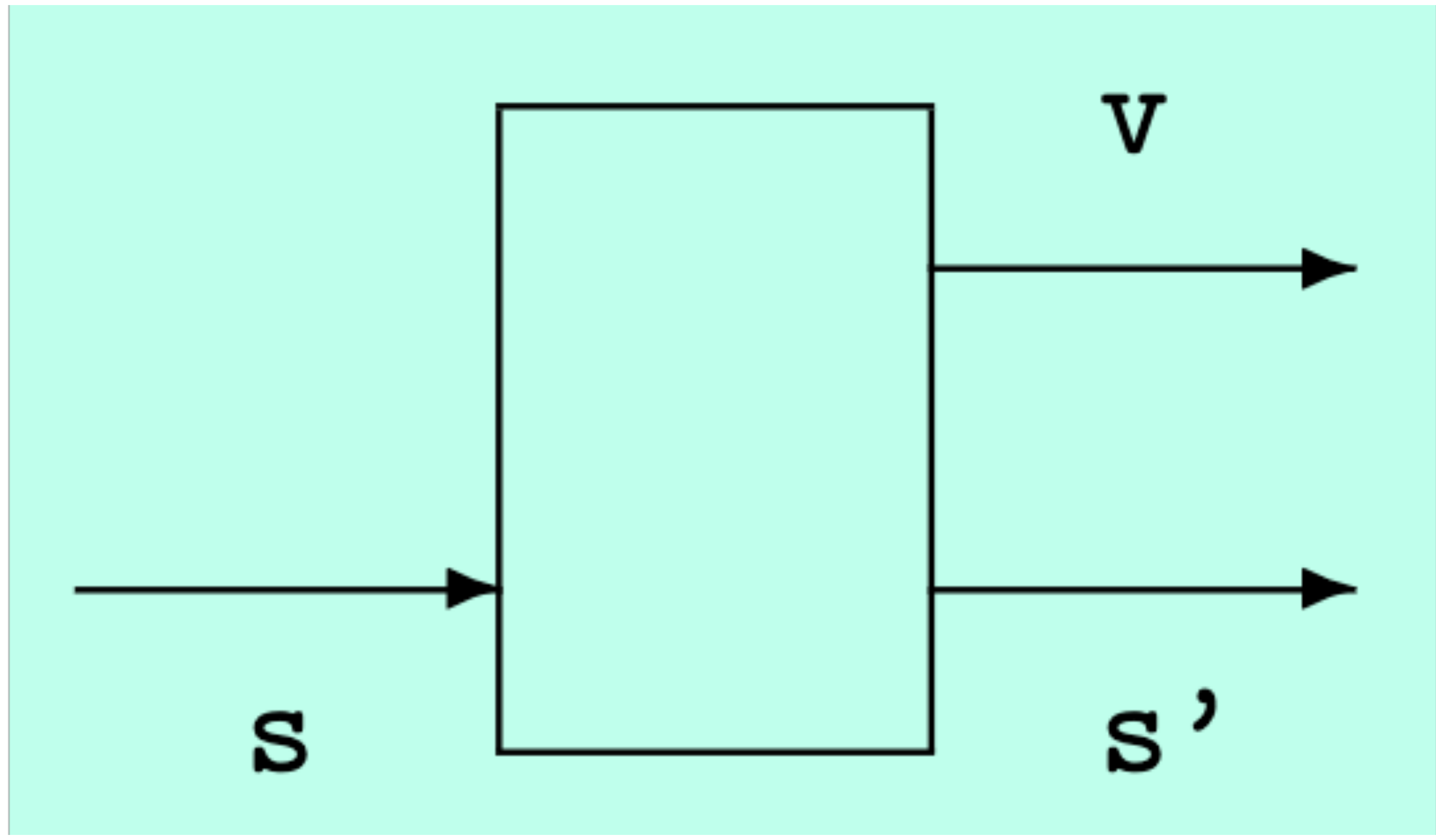
声明 [ ] 为Monad的一个实例

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    m >> k = m >>= \_ -> k
```

```haskell
instance Monad [] where
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = [y | x <- xs, y <- f x]
```

The State Monad

- 问题：如何用函数描述状态的变化
  - 状态：一种数据类型
    - type State = Int
    - 仅仅是一个示例；需根据具体问题确定状态的类型
  - 状态变换器
    - type ST = State -> State
  - 带有结果的状态变换器
    - type ST a = State -> (a, State)



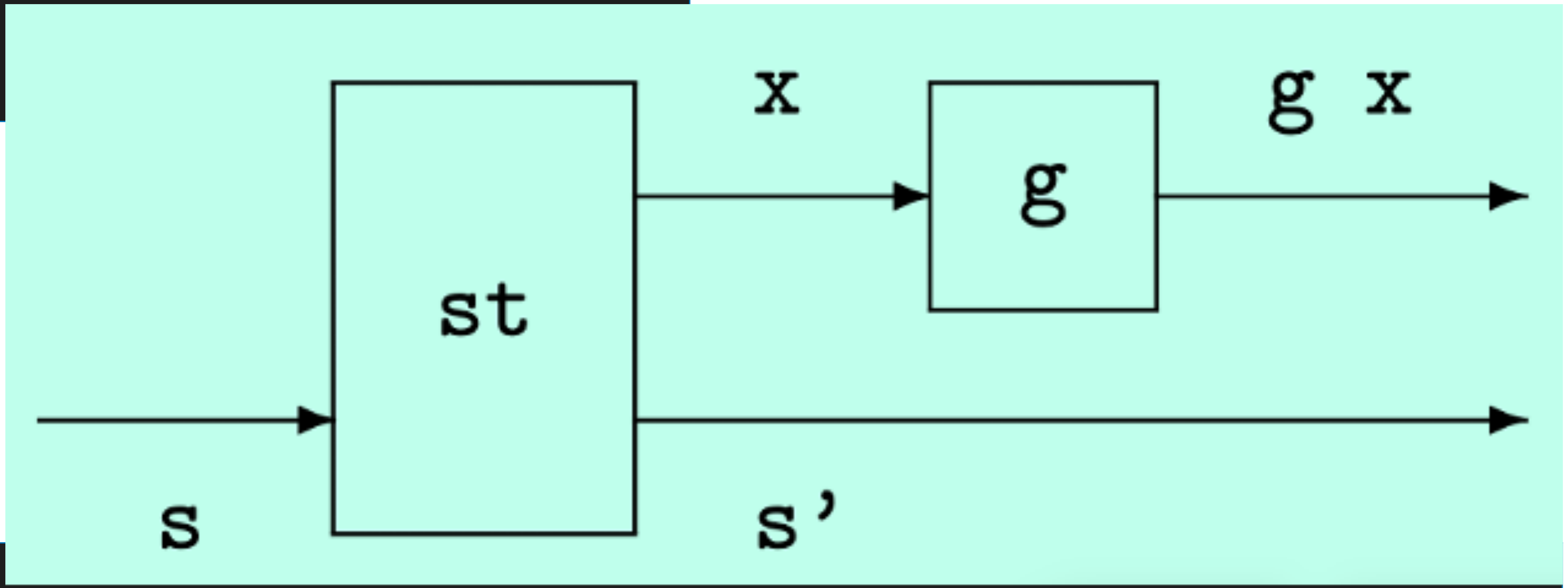
Haskell不支持将ST声明为Functor/Applicative/Monad的实例

用 newtype 定义 ST

```haskell
newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S f) s = f s
```

将 ST 声明为 Functor 的实例

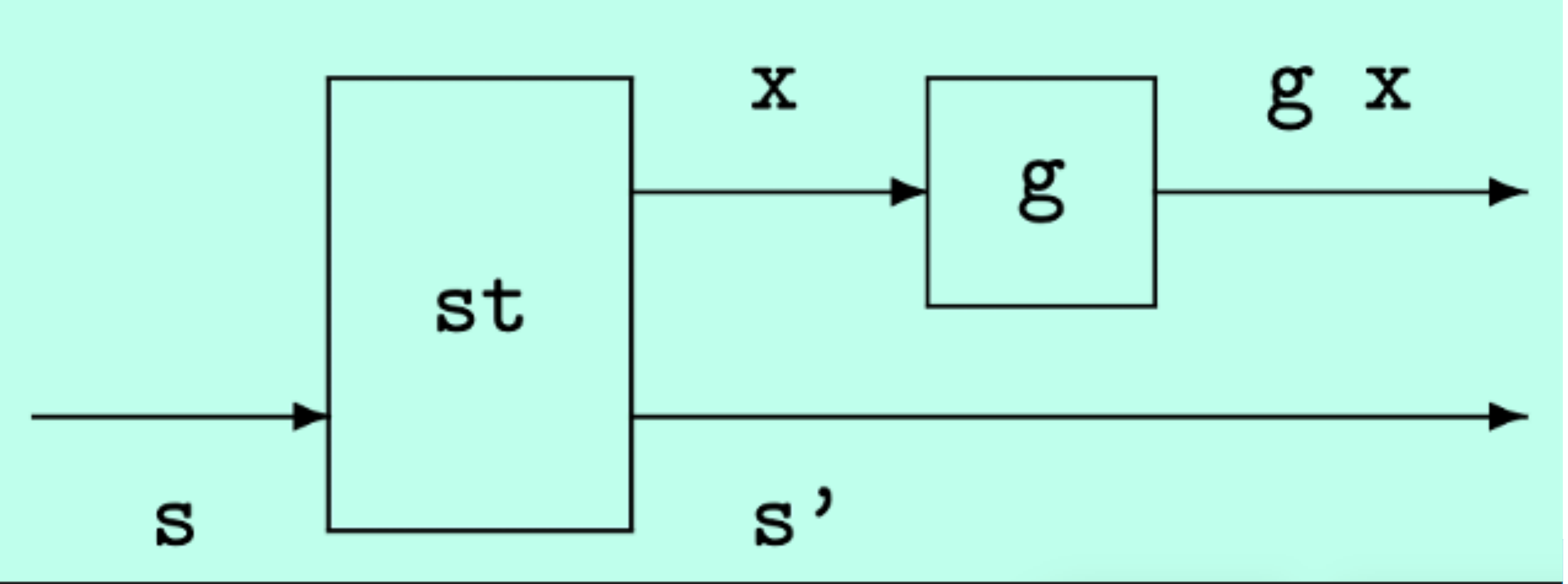
```haskell
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S f) s = f s
```




```haskell
instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  fmap g st = S
```

将 ST 声明为 Functor 的实例

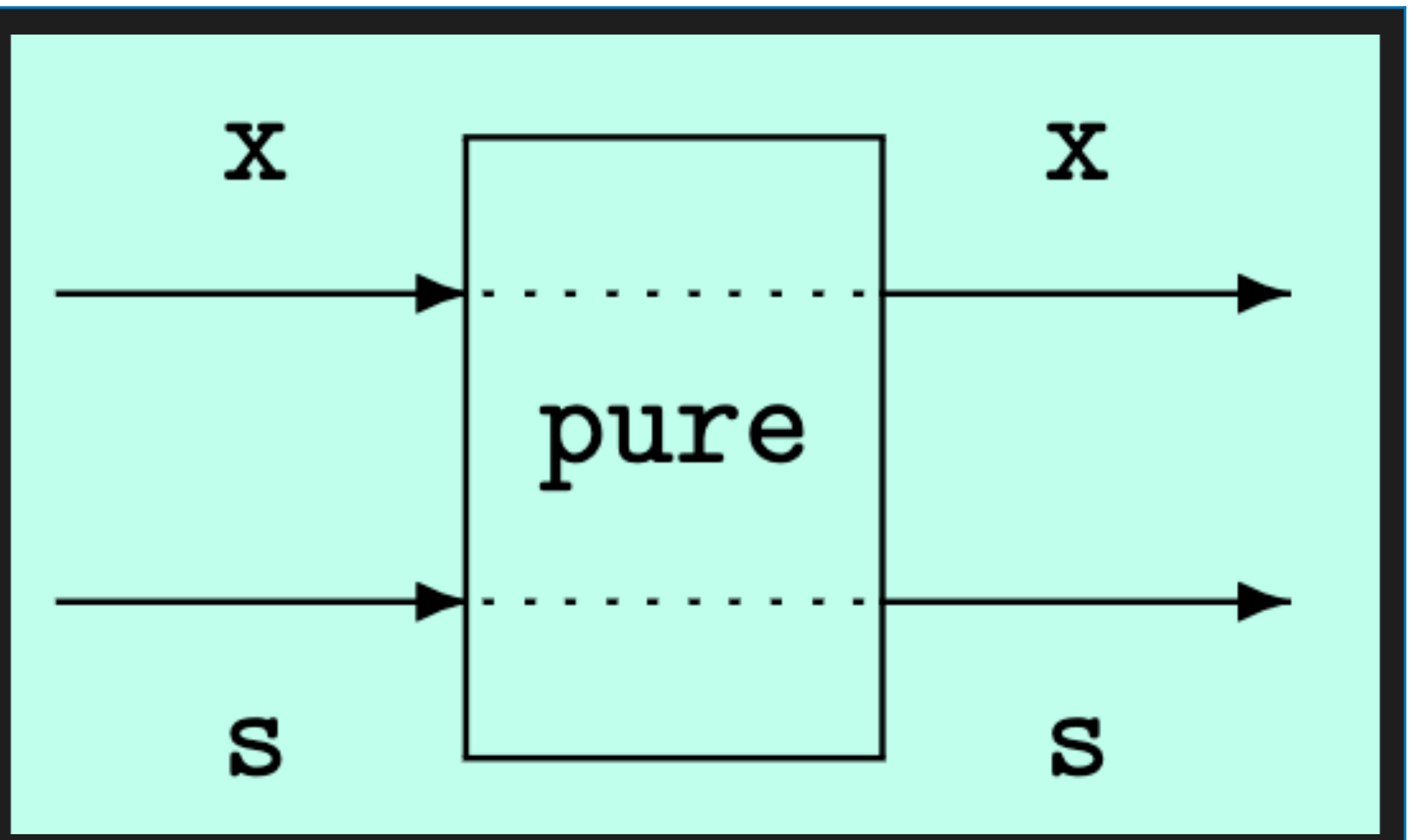
```haskell
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S f) s = f s
```




```haskell
instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  fmap g st = S $ \s -> let (x, s') = app st s in (g x, s')
```

将 ST 声明为 Applicative 的实例

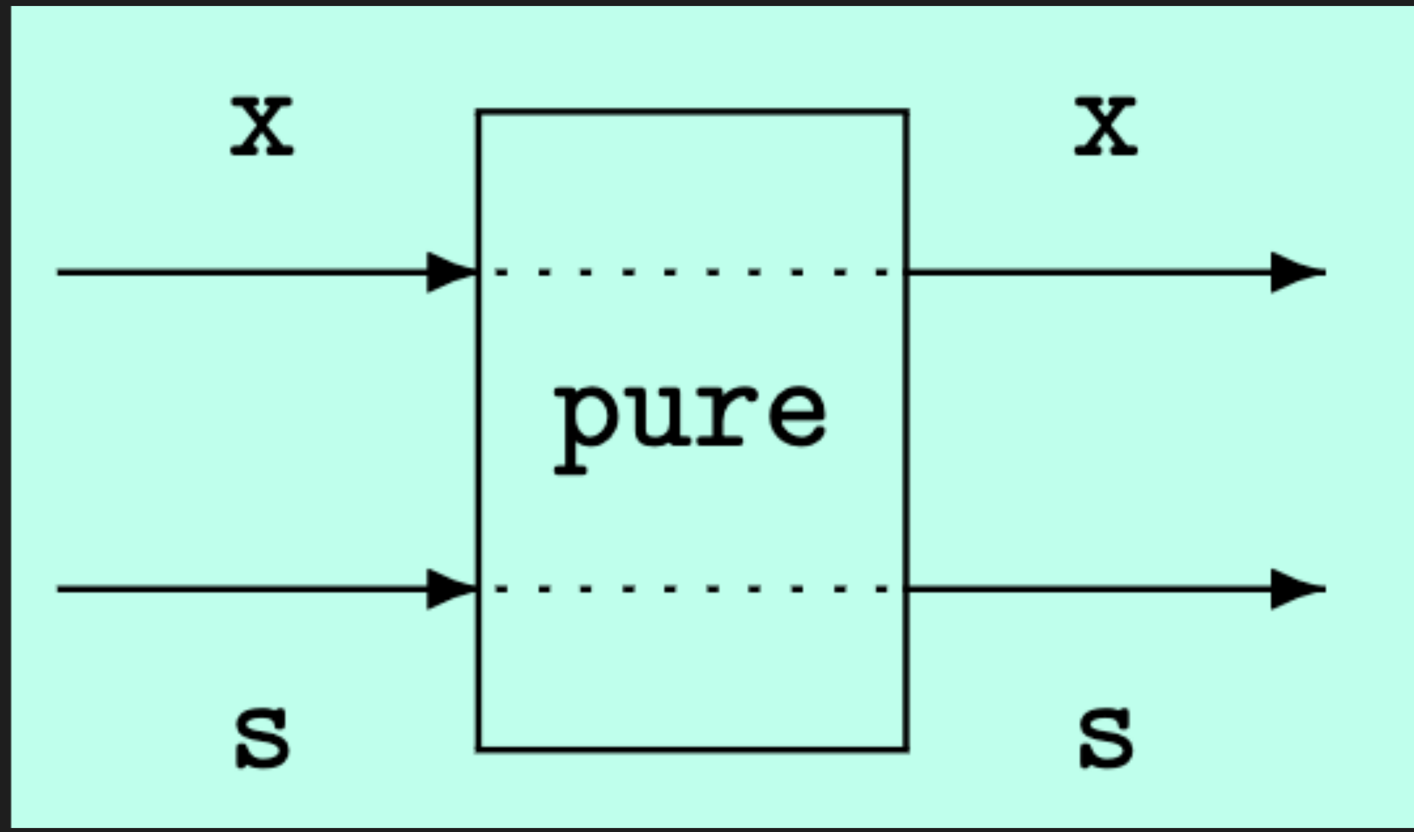
```haskell
instance Applicative ST where
    -- pure :: a -> ST a
    pure x = S [redacted]

    -- (<*>) :: ST (a -> b) -> ST a -> ST b
    stf <*> stx = S [redacted]
```



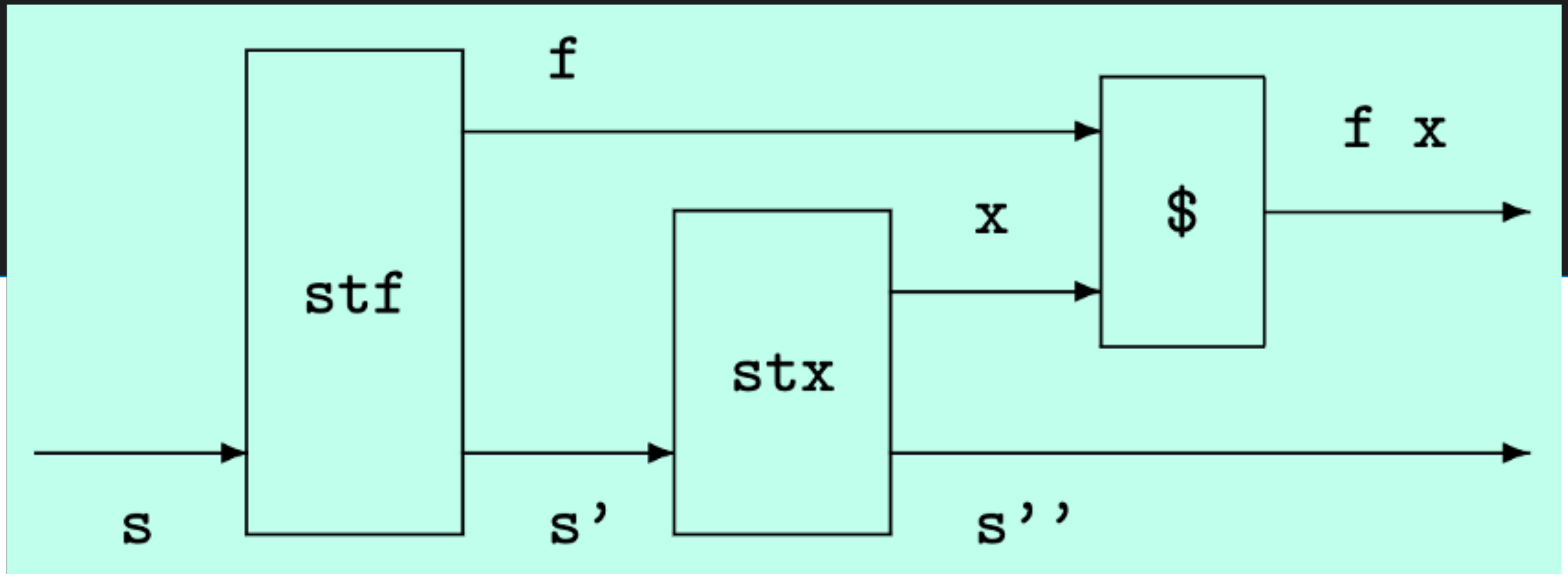

```haskell
newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S f) s = f s
```

将 ST 声明为 Applicative 的实例

```haskell
instance Applicative ST where
    -- pure :: a -> ST a
    pure x = S $ \s -> (x,s)
```



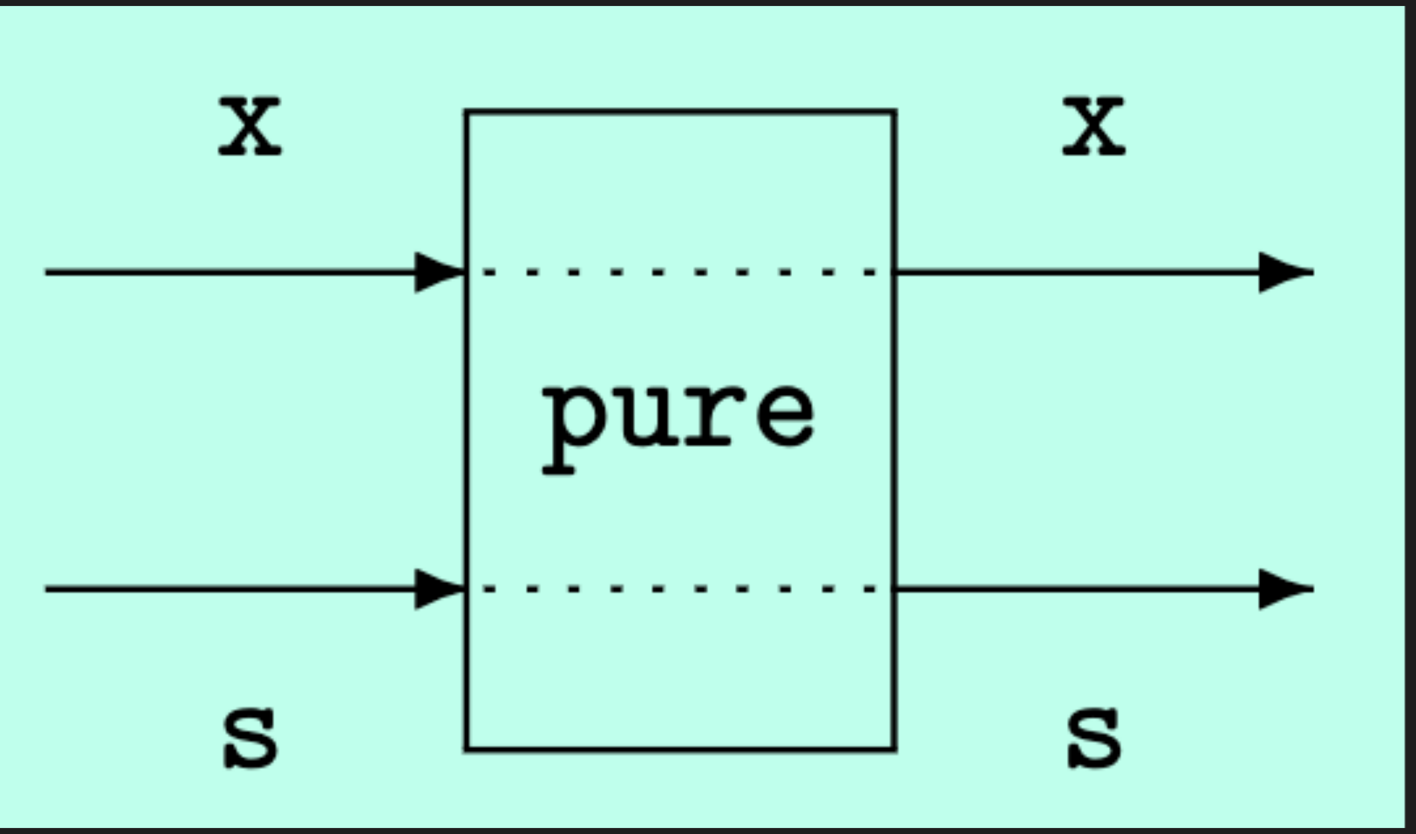

```haskell
    -- (<*>) :: ST (a -> b) -> ST a -> ST b
    stf <*> stx = S
```

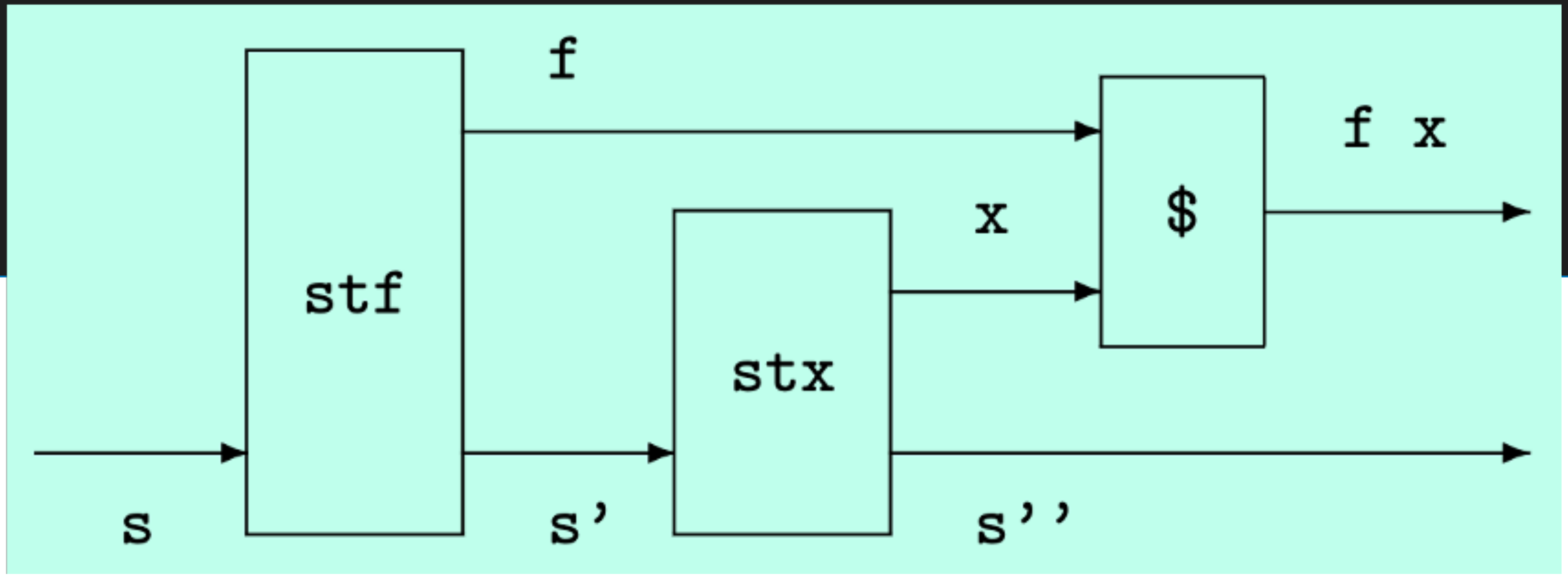



```haskell
newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S f) s = f s
```

将 ST 声明为 Applicative 的实例

```haskell
instance Applicative ST where
    -- pure :: a -> ST a
    pure x = S $ \s -> (x,s)

    -- (<*>) :: ST (a -> b) -> ST a -> ST b
    stf <*> stx = S $ \s -> let (f, s' ) = app stf s
                                (x, s'') = app stx s'
                            in  (f x, s''))
```

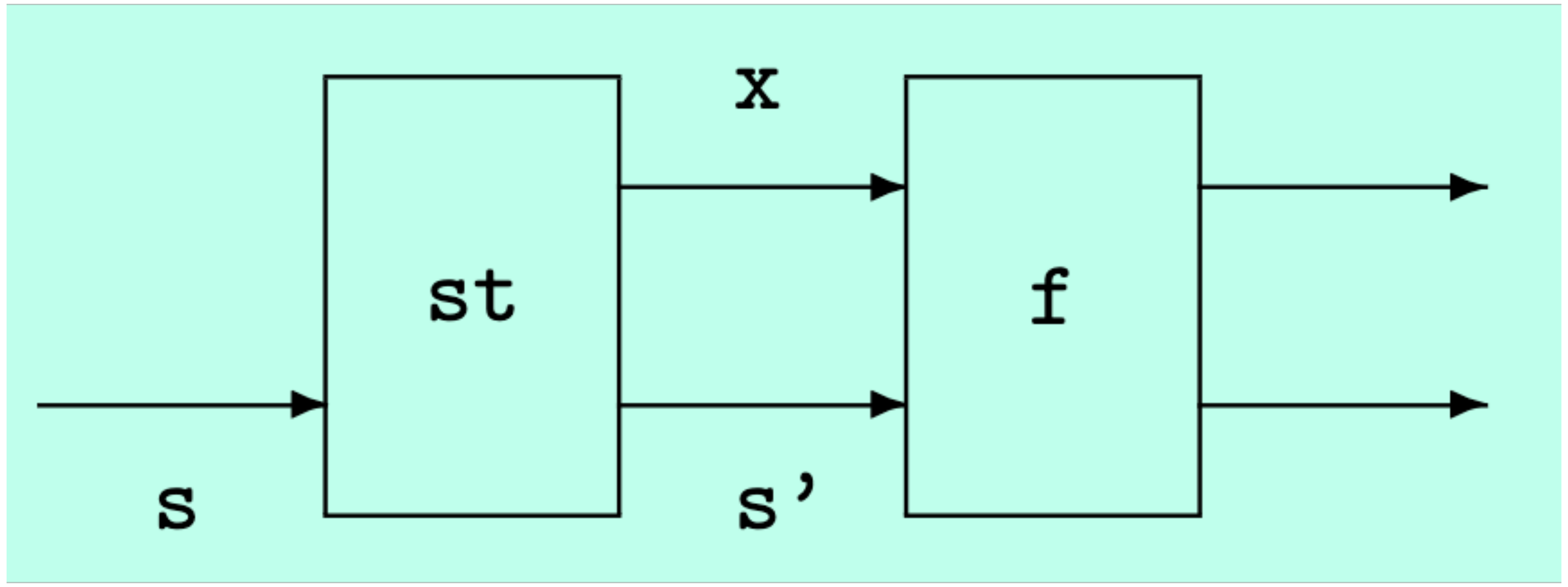





```haskell
newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S f) s = f s
```

将 ST 声明为 Monad 的实例

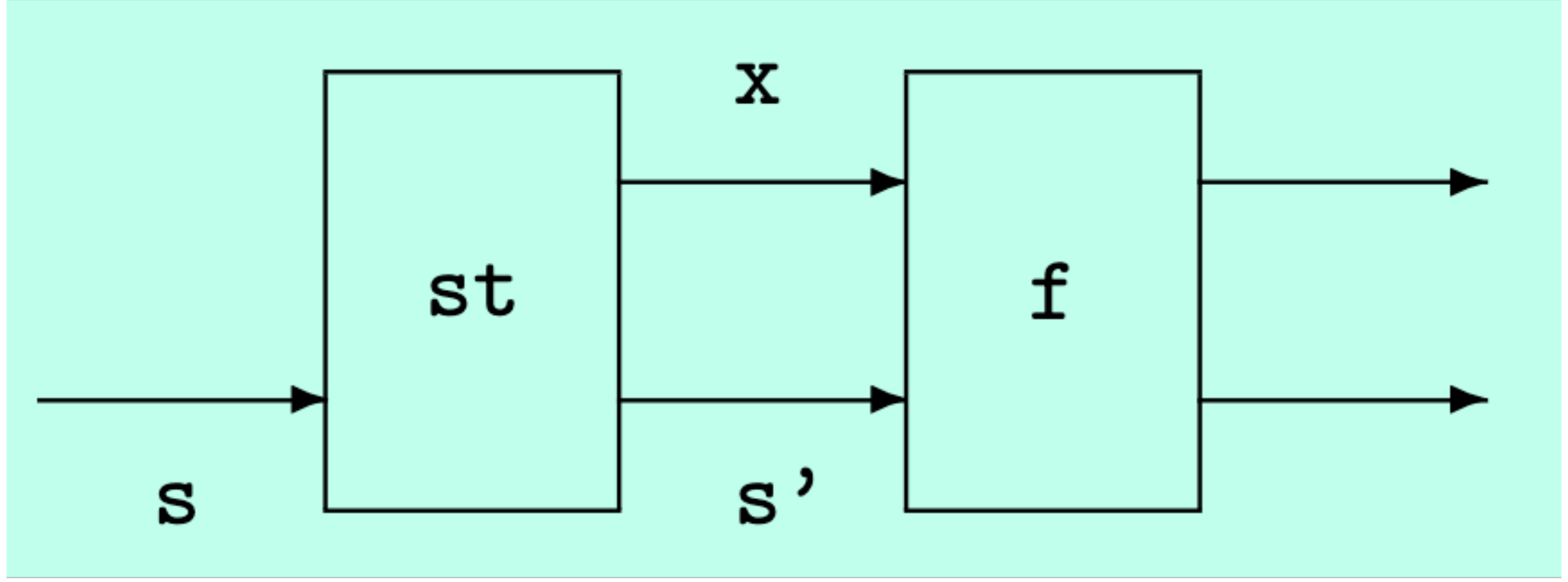
```
instance Monad ST where
    -- (>>=) :: ST a -> (a -> ST b) -> ST b
    st >>= f = S
```




```
newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S f) s = f s
```

将 ST 声明为 Monad 的实例

```haskell
instance Monad ST where
    -- (>>=) :: ST a -> (a -> ST b) -> ST b
    st >>= f = S $ \s -> let (x,s') = app st s
                         in app (f x) s')
```




```haskell
newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S f) s = f s
```

The State Monad

这几张幻灯片讲的挺好的

下次不要再讲了

感觉讲了一些无用的废话

在我第一次看到State Monad时
内心的想法其实也和你们差不多

The State Monad 之 应用示例：树的重新标注

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

- Consider the problem of defining a function that relabels each leaf in such a tree with a unique or fresh integer.

```
ghci> relabel tree
Node (Node (Leaf 0) (Leaf 1)) (Leaf 2)
```

树的重新标注 之 方法一：朴实无华~隐入尘烟

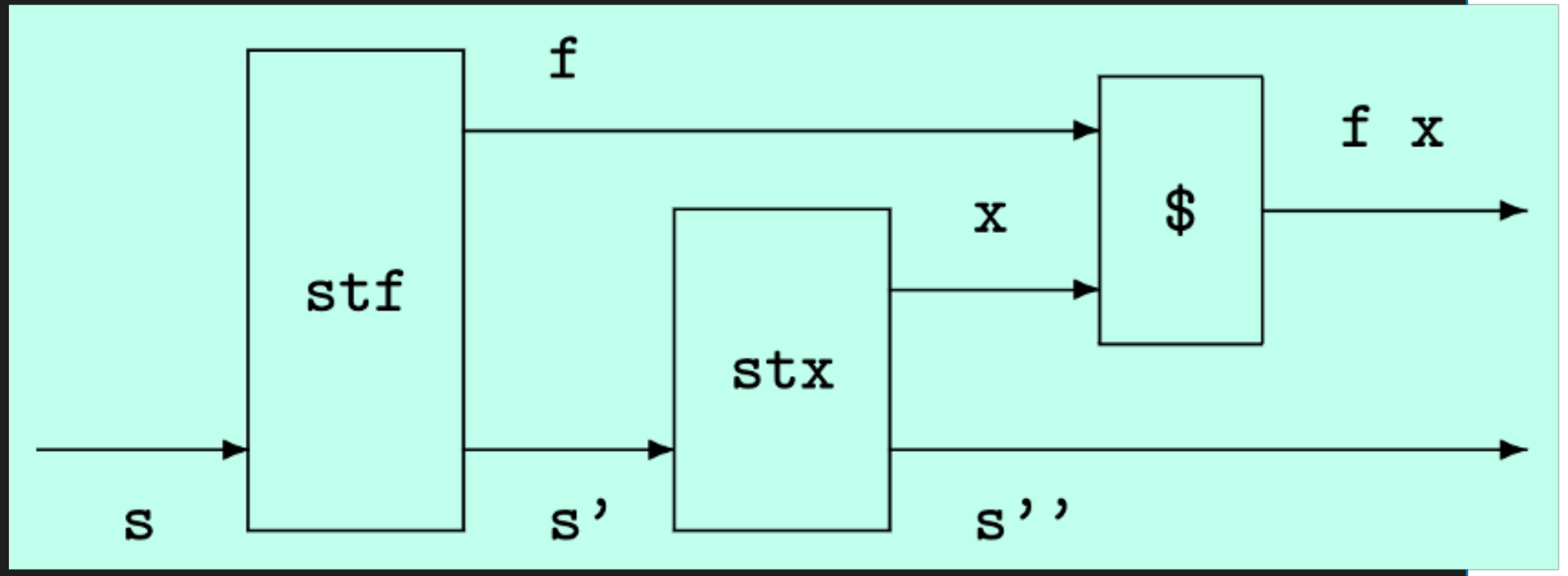
```haskell
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _   ) n = (Leaf n, n+1)
rlabel (Node l r) n = (Node l' r', n'')
                where (l', n' ) = rlabel l n
                      (r', n'') = rlabel r n'

relabel :: Tree a -> Tree Int
relabel t = fst (rlabel t 0)
```

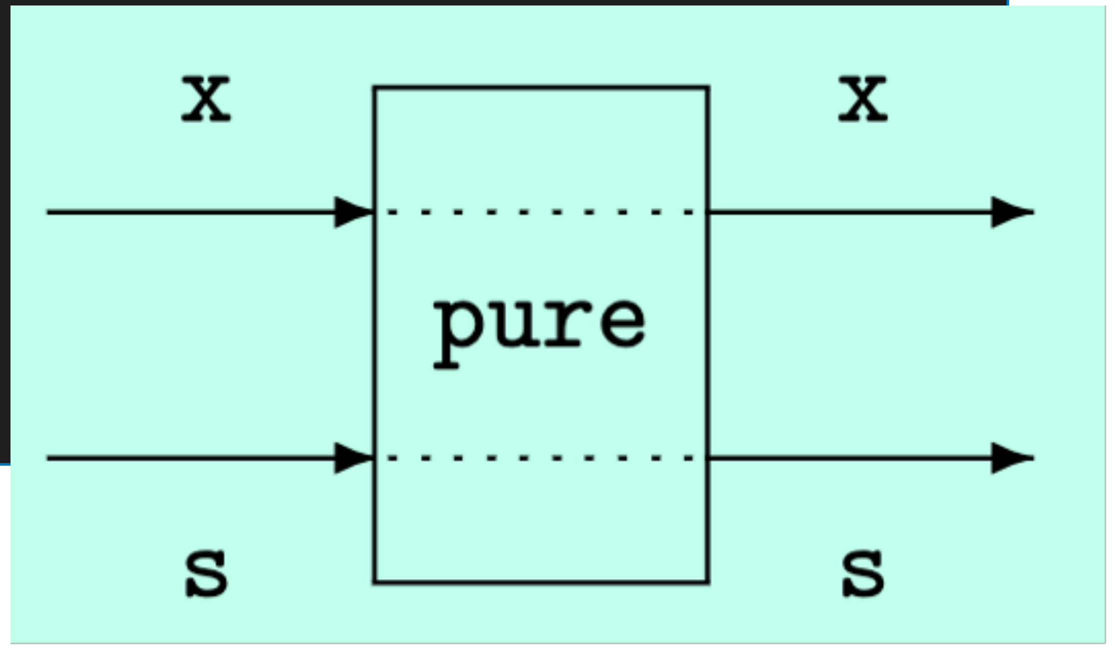
缺点：rlabel 的定义中需要显式维护中间状态

```
ghci> relabel tree
Node (Node (Leaf 0) (Leaf 1)) (Leaf 2)
```

树的重新标注 之 方法二：Applicative

```haskell
fresh :: ST Int
fresh = S $ \n -> (n, n+1)

alabel :: Tree a -> ST (Tree Int)
alabel (Leaf _)   = Leaf <$> fresh
alabel (Node l r) = Node <$> alabel l <*> alabel r

relabel' :: Tree a -> Tree Int
relabel' t = fst $ app (alabel t) 0
```



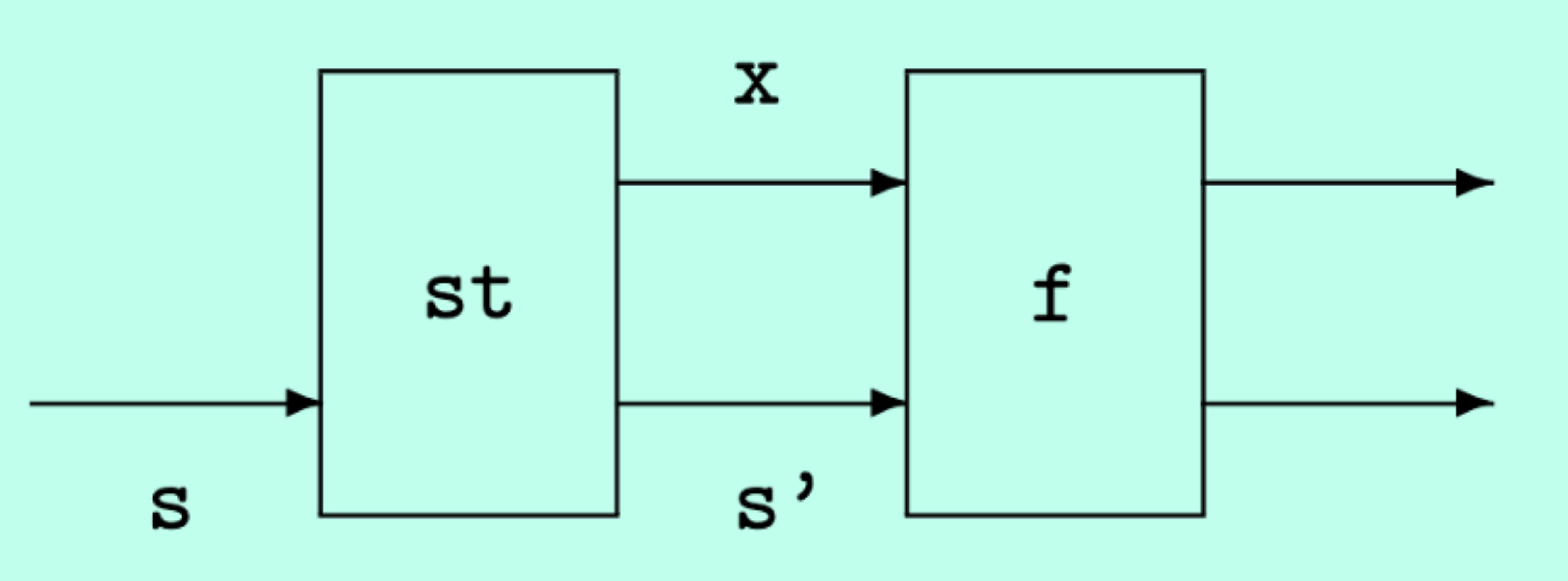


<$> = `fmap`
or
g <$> x = pure g <*> x

我时常在想，这些东西是永恒的吗？
如果是，它们栖身何处，以至可以被人类发现并表达

树的重新标注 之 方法三：Monad

```haskell
mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _)    = fresh >>= \n -> return $ Leaf n
mlabel (Node l r) = mlabel l >>= \l' ->
                    mlabel r >>= \r' -> return $ Node l' r'

relabel'' :: Tree a -> Tree Int
relabel''  t = fst $ app (mlabel t) 0
```




```haskell
mlabel (Leaf _   ) = do n <- fresh
                        return (Leaf n)
mlabel (Node l r) = do l' <- mlabel l
                       r' <- mlabel r
                       return $ Node l' r'
```

使用 do 改写 mlabel

Monad Laws

Left identity	`return a`	`>>=`	`h`	`=`	`h a`
Right identity	`mx`	`>>=`	`return`	`=`	`mx`
Associativity	`(mx >>= g)`	`>>=`	`h`	`=`	`mx >>= (\x -> g x >>= h)`
	`(mx >>= \x -> g x) >>= h`			`=`	`mx >>= (\x -> g x >>= h)`

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    m >> k = m >>= \_ -> k
```

Monad Laws: Another Form

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure
    (>>=) :: m a -> (a -> m b) -> m b
    ...
```

```haskell
-- The monad-composition operator
-- defined in Control.Monad
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g       = \x -> f x >>= g
```

Left identity	`return a >>= h`			=	`h a`
	`(\x -> return x >>= h) a`			=	`h a`
	`( return >=> h) a`			=	`h a`
	`return >=> h`			=	`h`

看！是不是 Left identity

Monad Laws: Another Form

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure
    (>>=) :: m a -> (a -> m b) -> m b
    ...
```

```haskell
-- The monad-composition operator
-- defined in Control.Monad
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g      = \x -> f x >>= g
```

Right identity	mb	>>=	return	=	mb
	f a	>>=	return	=	f a
	(\x -> f x	>>=	return) a	=	f a
	(f	>=>	return) a	=	f a
	f	>=>	return	=	f

我时常在想，“朝三暮四”是个贬义词吗？

Monad Laws: Another Form

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    return = pure
    (>>=) :: m a -> (a -> m b) -> m b
    ...
```

```haskell
-- The monad-composition operator
-- defined in Control.Monad
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g      = \x -> f x >>= g
```

Assoc	`( mb >>= g) >>= h = mb >>= (\x -> g x >>= h)`
	`( f a >>= g) >>= h = f a >>= (\x -> g x >>= h)`
	`( f a >>= g) >>= h = f a >>= ( g >=> h)`
	`(\x -> f x >>= g) a >>= h = f a >>= ( g >=> h)`
	`( f >=> g) a >>= h = f a >>= ( g >=> h)`
	`(\x -> ( f >=> g) x >>= h) a = (\x -> f x >>= ( g >=> h)) a`
	`( ( f >=> g) >=> h) a = ( f >=> ( g >=> h)) a`
	`( f >=> g) >=> h = f >=> ( g >=> h)`

Monad Laws in practice

Left identity
`do { x' <- return x; f x' }`
`return x >>= \x' -> f x'`
`return x >>=          f`
`f x`
`do { f x }`

Monad Laws in practice

Right identity
`do {      x <- mx; return x }`
`     mx >>= \x -> return x`
`     mx >>=       return`
`     mx`
`do { mx }`

Monad Laws in practice

Associativity
`do { y <- do { x <- mx; f x }; g y }`
`do { x <- mx; f x } >>= \y -> g y`
`( mx >>= \x -> f x ) >>= \y -> g y`
`( mx >>=       f   ) >>=       g`
`  mx >>=(\x -> f x     >>=       g)`
`do { x <- mx; do { y <- f x; g y} }`
`do { x <- mx;      y <- f x; g y  }`

Monad Laws in practice

```
skip_and_get = do unused <- getLine
                  line   <- getLine
                  return line
```

|| Right identity

```
skip_and_get = do unused <- getLine
                  getLine
```

Monad Laws in practice

```
main = do answer <- skip_and_get
          putStrLn answer
```

|| inlining

```
main = do answer <- do { unused <- getLine;
                         getLine }
          putStrLn answer
```

|| Associativity

```
main = do unused <- getLine
          answer <- getLine
          putStrLn answer
```

这些law根本不是什么约束
而是天然就应该存在的

Monads as computation

- Monadic computations have results.
  - This is reflected in the types. Given a monad M, a value of type M t is a computation resulting in a value of type t.
- For any value, there is a computation which "does nothing", and produces that result.
  - `return :: (Monad m) => a -> m a`

Monads as computation

- Given a pair of computations x and y, one can form the computation x >> y, which intuitively "runs" the computation x, throws away its result, then runs y returning its result.
  - `(>>) :: (Monad m) => m a -> m b -> m b`
- Further, we're allowed to use the result of the first computation to decide "<u>what to do next</u>", rather than just throwing it away.
  - `(>>=) :: (Monad m) => m a -> (a -> m b) -> m b`
  - `x >>= f`: a computation which runs x, then applies f to its result, getting a computation which it then runs.

Monads as computation

```haskell
main :: IO ()
main = getLine >>= putStrLn
```

```haskell
main :: IO ()
main = putStrLn "Enter a line of text:"
          >> getLine >>= \x -> putStrLn (reverse x)
```

- Because computations are typically going to be built up from long chains of >> and >>=, in Haskell, we have some syntax-sugar, called do-notation

```haskell
main = do putStrLn "Enter a line of text:"
          x <- getLine
          putStrLn (reverse x)
```

Monads as computation

- The basic mechanical translation for the do-notation:

```
do { x } = x

do { x ; <stmts> }
  = x >> do { <stmts> }

do { v <- x ; <stmts> }
  = x >>= \v -> do { <stmts> }

do { let <decls> ; <stmts> }
  = let <decls> in do { <stmts> }
```

Monads as computation

- This gives monadic computations a bit of an imperative feel.
- But it's important to remember that the monad in question gets to decide what the combination means, and so some unusual forms of control flow might actually occur.
- In some monads (like parsers, or the list monad), "backtracking" may occur, and in others, even more exotic forms of control might show up.

Monads as computation
Some examples from Control.Monad

- A function which takes a list of computations of the same type, and builds from them a computation which will run each in turn and produce a list of the results.

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []      = return []
sequence (x:xs) = x >>= \v -> sequence xs >>= \vs -> return (v:vs)
```

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []      = return []
sequence (x:xs) = do v <- x
                     vs <- sequence xs
                     return (v:vs)
```

```
main = sequence [getLine, getLine] >>= print
```

Monads as computation
Some examples from Control.Monad

```haskell
forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
forM xs f = sequence (map f xs)
```

```haskell
main = forM [1..10] $ \x -> do
         putStr "Looping: "
         print x
```

- There are variants of sequence and forM, called sequence_ and forM_, which simply throw the results away as they run each of the actions.

```haskell
sequence_ :: (Monad m) => [m a] -> m ()
sequence_ []       = return ()
sequence_ (x:xs) = x >> sequence_ xs

forM_ :: (Monad m) => [a] -> (a -> m b) -> m ()
forM_ xs f = sequence_ (map f xs)
```

Monads as computation
Some examples from Control.Monad

- Sometimes we only want a computation to happen when a given condition is true.

```
when :: (Monad m) => Bool -> m () -> m ()
when p x = if p then x else return ()
```

课堂练习 1

- Define an instance of the Functor class for the following type of binary trees that have data in their nodes:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving (Show)
```

```haskell
instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
```

课堂练习 1

- Define an instance of the Functor class for the following type of binary trees that have data in their nodes:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving (Show)
```

```haskell
instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
    fmap g Leaf = Leaf
    fmap g (Node l x r) = Node (fmap g l) (g x) (fmap g r)
```

课堂练习 2

- Complete the following instance declaration to make the partially-applied function type `(->) a` into a functor:

```haskell
instance Functor ((->) a) where
  -- fmap :: (a -> b) -> f a -> f b
```

课堂练习 2

- Complete the following instance declaration to make the partially-applied function type `(->) a` into a functor:

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

课堂练习 2

- Complete the following instance declaration to make the partially-applied function type `(->) a` into a functor:

```haskell
instance Functor ((->) a) where
  -- fmap :: (a -> b) -> f a -> f b
  -- fmap :: (b -> c) -> f b -> f c
  -- fmap :: (b -> c) -> (->) a b -> (->) a c
```

课堂练习 2

- Complete the following instance declaration to make the partially-applied function type `(->) a` into a functor:

```haskell
instance Functor ((->) a) where
  -- fmap :: (a -> b) -> f a -> f b
  -- fmap :: (b -> c) -> f b -> f c
  -- fmap :: (b -> c) -> (->) a b -> (->) a c
  -- fmap :: (b -> c) -> (a -> b) -> (a -> c)
```

如果一个东西可以被定义为Functor的实例，那么，只有一种fmap的定义方式

课堂练习 2

- Complete the following instance declaration to make the partially-applied function type `(->) a` into a functor:

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

如果一个东西可以被定义为Functor的实例，那么，只有一种fmap的定义方式

课堂练习 3

- Define an instance of the Applicative class for the type `(->) a`

```haskell
instance Applicative ((->) a) where
  -- pure :: a -> f a



  -- (<*>) :: f (a -> b) -> f a -> f b



```

课堂练习 3

- Define an instance of the Applicative class for the type `((->) a)`

```
instance Applicative ((->) a) where
  -- pure :: a -> f a
  -- pure :: b -> f b



  -- (<*>) :: f (a -> b) -> f a -> f b
  -- (<*>) :: f (b -> c) -> f b -> f c


```

课堂练习 3

✤ Define an instance of the Applicative class for the type `((->) a)`

```
instance Applicative ((->) a) where
  -- pure :: a -> f a
  -- pure :: b -> f b
  -- pure :: b -> a -> b


  -- (<*>) :: f (a -> b) -> f a -> f b
  -- (<*>) :: f (b -> c) -> f b -> f c
  -- (<*>) :: (a -> b -> c) -> (a -> b) -> (a -> c)

```

课堂练习 3

- Define an instance of the Applicative class for the type `(->) a`

```haskell
instance Applicative ((->) a) where
  -- pure :: a -> f a
  -- pure :: b -> f b
  -- pure :: b -> a -> b
  pure = const

  -- (<*>) :: f (a -> b) -> f a -> f b
  -- (<*>) :: f (b -> c) -> f b -> f c
  -- (<*>) :: (a -> b -> c) -> (a -> b) -> (a -> c)
```

课堂练习 3

- Define an instance of the Applicative class for the type `(->) a`

```
instance Applicative ((->) a) where
  -- pure :: a -> f a
  -- pure :: b -> f b
  -- pure :: b -> a -> b
  pure = const

  -- (<*>) :: f (a -> b) -> f a -> f b
  -- (<*>) :: f (b -> c) -> f b -> f c
  -- (<*>) :: (a -> b -> c) -> (a -> b) -> (a -> c)
  g <*> h = \x -> g x $ h x
```

作业

12-1 Define an instance of the Monad class for the type (->) a.

12-2 Given the following type of expressions

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
              deriving Show
```

that contain variables of some type a, show how to make this type into instances of the Functor, Applicative and Monad classes. With the aid of an example, explain what the >>= operator for this type does.

第10章： Monads and More

就到这里吧