第十二章：Monads and More

Functors

Applicatives

Monads

Genericity/通用性

- Level 1: **Polymorphic** Functions (over types)

  length1 :: List **a** -> Int

- Level 2: **Generic** Functions (over type constructors)

  length2 :: **t** a -> Int

FUNCTORS （函子）

计算的抽象

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

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

抽象

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

```
inc = map (+1)
sqr = map (^2)
```

Abstraction over Parameterized Types

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

fmap takes a function of type a -> b and a structure of type f a whose elements have type a, and applies the function to each such element to give a structure of type f b whose elements now have type b.

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

```
Prelude> fmap (+1) [1,2,3]
[2,3,4]

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

```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)
```

```
Prelude> fmap (+1) (Just 3)
Just 4

Prelude> fmap (+1) Nothing
Nothing

Prelude> 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)
```

```
Prelude> fmap length (Leaf "abc")
Leaf 3

Prelude> 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)
```

```
Prelude> fmap show (return True)
"True”
```

Generic Function Definition

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

```
> inc (Just 1)
Just 2

> inc [1,2,3,4,5]
[2,3,4,5,6]

> inc (Node (Leaf 1) (Leaf 2))
Node (Leaf 2) (Leaf 3)
```

Functor Laws

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

They ensure that fmap does indeed perform a mapping operation.

- 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.

Applicative Functors

APPLICATIVES

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 -> f c

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

.
.
.
```

- Idea: 准备两个基本函数

```
pure :: a -> f a

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

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

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

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

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

Applicative Functor

```
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

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

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

```
> pure (+1) <*> Just 1
Just 2

> pure (+) <*> Just 1 <*> Just 2
> Just 3

> pure (+) <*> Nothing <*> Just 2
Nothing
```

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

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

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

> pure (+) <*> [1] <*> [2]
[3]

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

the applicative style for lists supports a form of non-deterministic programming

```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)
```

Effectful Programming/Generic Programming

Effectful Programming

Applicative functors can also be viewed as abstracting the idea of applying pure functions to effectful arguments, with the precise form of effects that are permitted depending on the nature of the underlying functor.

Generic Programming

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA []       =
sequenceA (x:xs) =
```

Effectful Programming/Generic Programming

Effectful Programming

Applicative functors can also be viewed as abstracting the idea of applying pure functions to effectful arguments, with the precise form of effects that are permitted depending on the nature of the underlying functor.

Generic Programming

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

Applicative Law

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

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

MONADS

异常处理

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

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

```
> 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
```

繁杂

北京大学 PEKING UNIVERSITY

解决方法2

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

类型不正确

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

问题：applicative functor 只允许纯函数作用在有副作用的参数上

- 引入新的操作 **bind**

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

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

- 引入**do**语法糖

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

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

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

Monads

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

   return = pure
```

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

Monads

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

   return = pure
```

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

例：The State Monad

- 问题：如何用函数描述状态的变换？

  - 状态：一个数据结构

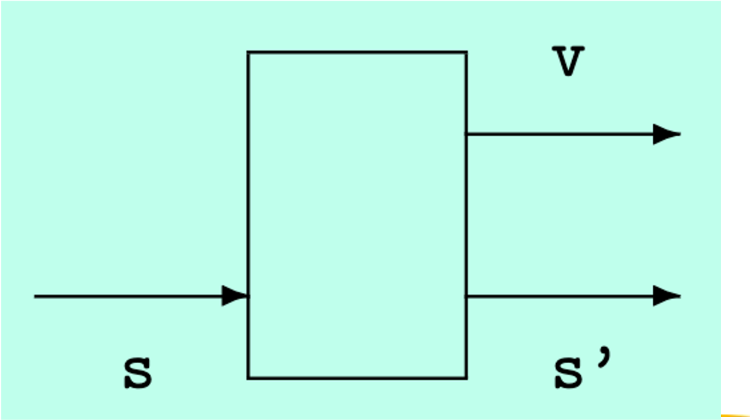
    `type State = Int` --仅仅是一个示例；需要根据具体问题确定状态的类型

  - 状态变换器

    `type ST = State -> State`

  - 带有结果的状态变换器

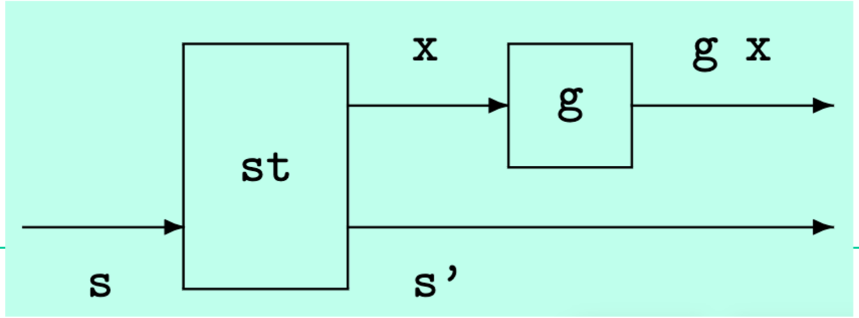
    `type ST a = State -> (a, State)`



例： The State Monad

- 用 newtype 定义 ST:

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

app :: ST a -> State -> (a,State)
app (S st) x = st x
```



- 定义 functor

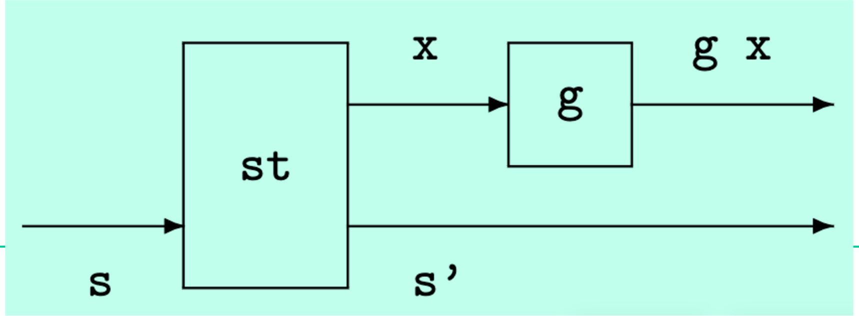
```
instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  fmap g st = S
```

例：The State Monad

- 用 newtype 定义 ST:

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

app :: ST a -> State -> (a,State)
app (S st) x = st x
```

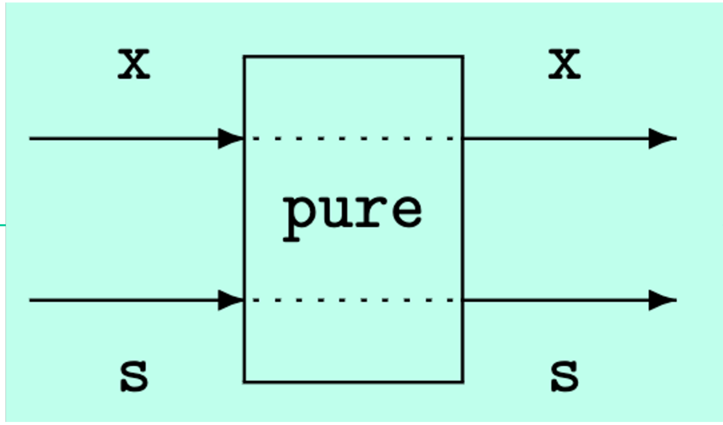
- 定义 functor




```
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'))
```

例： The State Monad

- Applicative

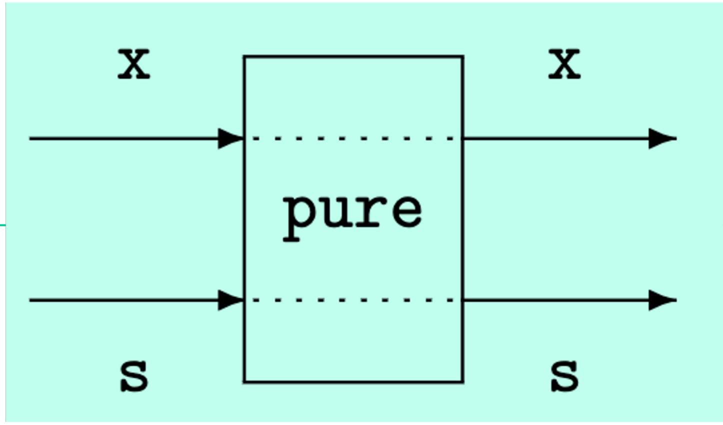


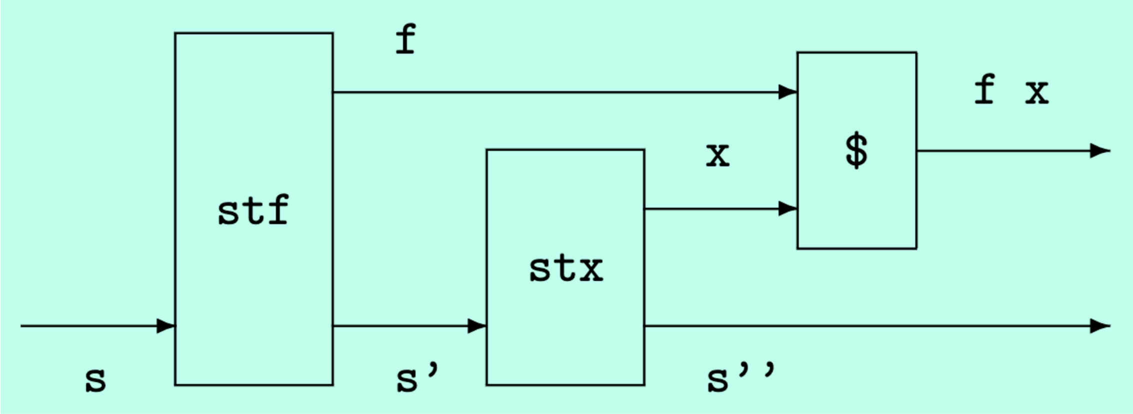

```
instance Applicative ST where
   -- pure :: a -> ST a
   pure x =

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

例：The State Monad

- Applicative

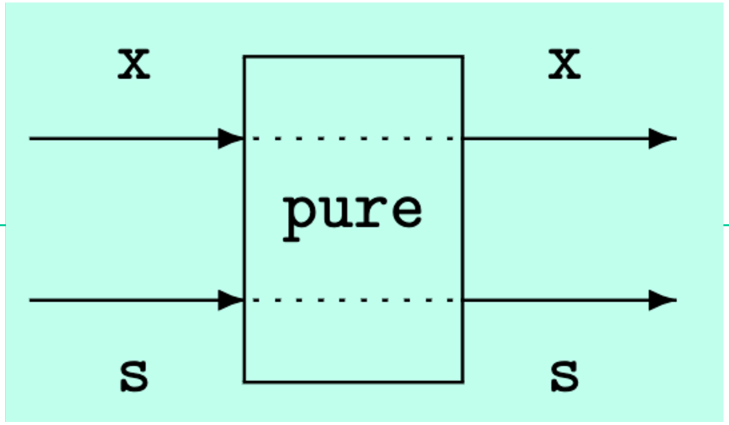
```
instance Applicative ST where
   -- pure :: a -> ST a
   pure x = S (\s -> (x,s))

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



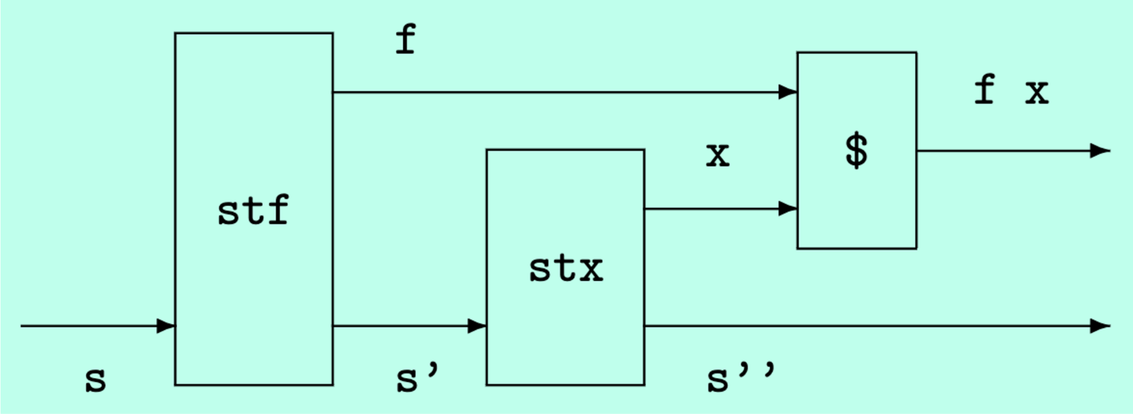


例： The State Monad

- Applicative



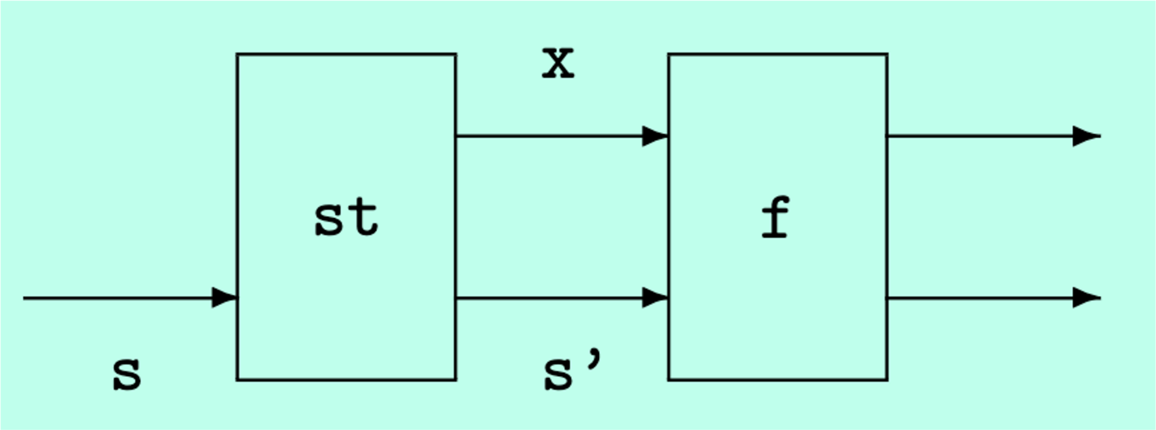

```
instance Applicative ST where
   -- pure :: a -> ST a
   pure x = S (\s -> (x,s))

   -- (<*>) :: 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”))
```



例： The State Monad

- Monad

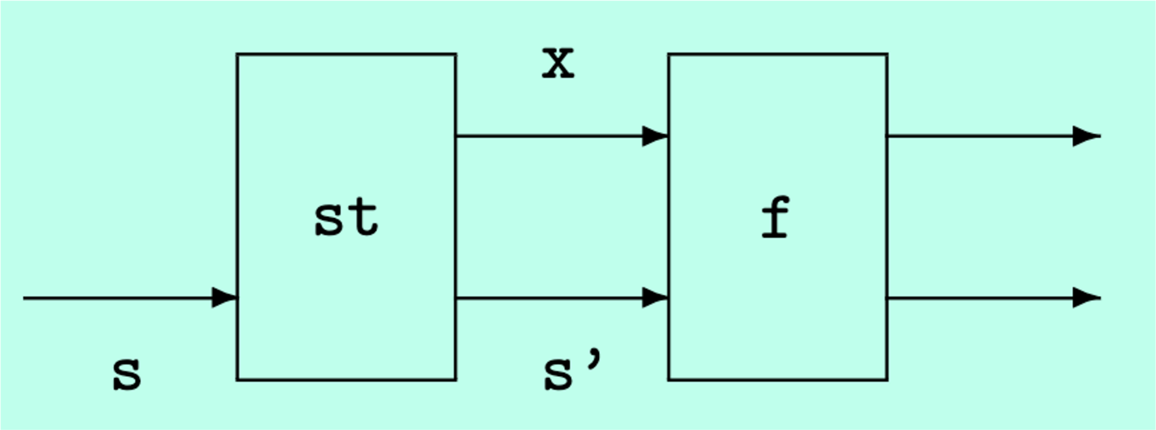
```
instance  Monad ST where
   -- (>>=) :: ST a -> (a -> ST b) -> ST b
   st >>= f =
```



例： The State Monad

- Monad

```
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’)
```



应用：树的重新标注

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

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

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

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

解法1

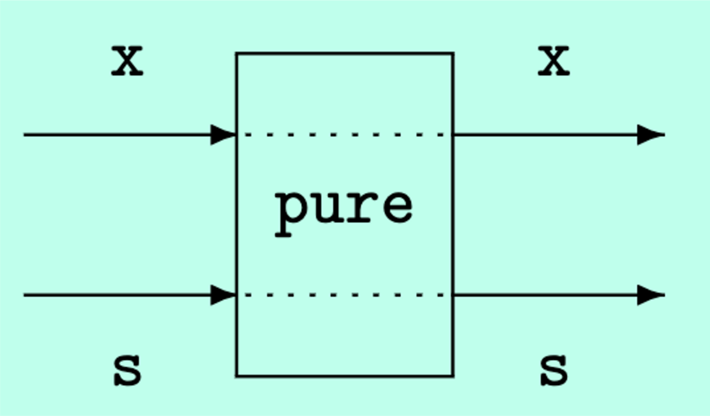
```
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _)   n  = (Leaf n, n+1)
rlabel (Node l r) n  = (Node l’ r’, n’’)
                       where  (l’,n’)   = rlabel l n
                              (r’,n’’)  = rlabel r n’

relabel t = fst (rlabel t 0)
```

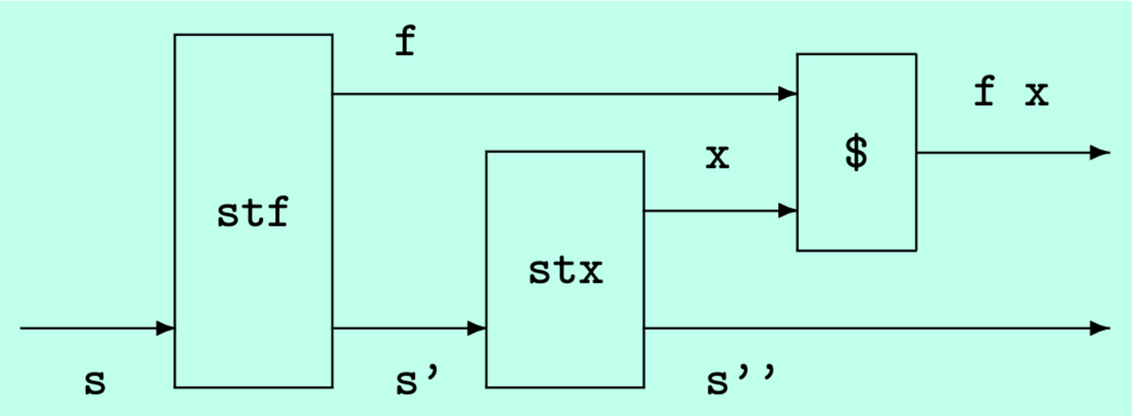
Note: This definition for rlabel is complicated by the need to explicitly thread an integer state through the computation.

解法2: 用Applicative

```
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  t = fst (app (alabel tree) 0)
```

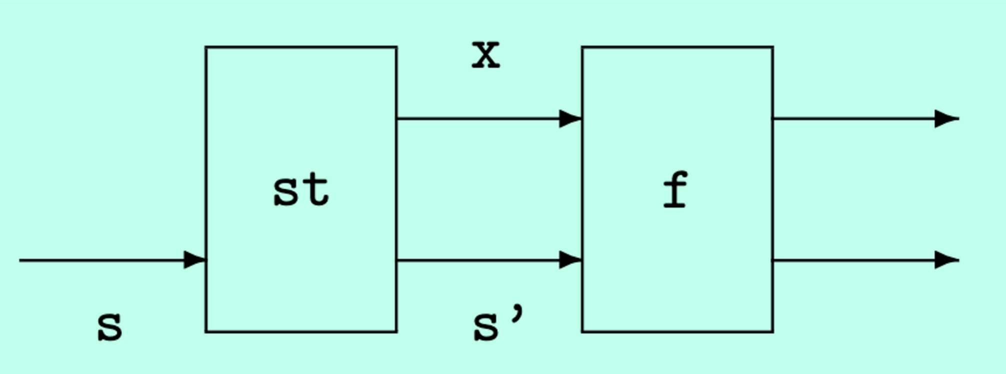
<$> = \`fmap\`
or
g <$> x = pure g <*> x





解法3: 用Monad

```
mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _)    = do n <- fresh
                        return (Leaf n)
mlabel (Node l r) = do l’ <- mlabel l
                        r’ <- mlabel r
                       return (Node l’ r’)

relabel  t = fst (app (mlabel t) 0)
```




```
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  t = fst (app (mlabel t) 0)
```

Monad Laws

```
 return x  >>=  f    = f x
 mx >>= return       = mx
(mx >>= f) >>= g     = mx >>= (\x -> (f x >>= g))
```

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

  return = pure
```

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:

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

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

```
instance Functor ((->) a) where
    ...
```

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

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.