State Monad
(modified from lectures by Graham Hutton and Stephanie Weirich)

Schedule

	Haskell			
Tue Nov 5 / Fri Nov 8	The State Monad	Ch. 12	Quiz 4 on Fri Lecture_Week11	PS7 due Tuesday Lecture11.hs, Lecture11'.hs, State.hs
Tue Nov 12 / Fr Nov 15	Parsing Theory (a bit); Monadic Parsing	Ch. 13		PS8 **Checkpoint #1: attend office hours this week (or earlier)**
Tue Nov 19 / Fri Nov 22	Parsec		Quiz 5 on Fri	**5-8 min presentation in class on Friday**
Tue Nov 26	Property Testing; QuickCheck			PS8 due on Tuesday
Tue Dec 3 Fri Dec 6	TBD		Quiz 6 on Fri	**Checkpoint #2: attend office hours this week (or earlier)**
Tue Dec 10	Project presentations			**Project due** **5-8 min presentation in class**

Outline

- Back to monads
  - Maybe and List monads, brief review
  - Either monad
- State transformations
  - State and pure functional programming
  - Imperative state in Haskell
  - The state transformer
- A generic state transformer
- Exercises

Monad

- Monad is a higher-kinded type class:

```
class Monad m where
  -- | Sequentially compose two actions, passing any
value produced by the first action to the second
  (>>=)  :: m a -> (a -> m b) -> m b
                        f

  -- | Inject a value into a monad type
  return :: a -> m a
```

- What are some instances of Monad?

```
instance Monad Maybe where
  -- return :: a -> Maybe a
  return = Just    -- return x = Just x

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

"swallows" errors

(return sheep) >>= mother >>= father >>= ...

The List Monad

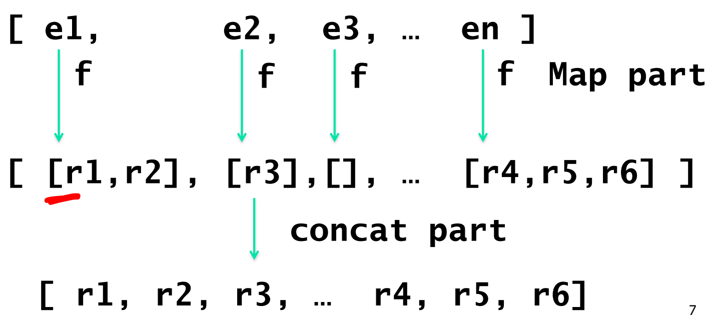
- List type constructor is an instance of the Monad type class:

```
instance Monad [] where
  -- return :: a -> [a]
  return x = [x]

  -- (>>=) :: [a] -> (a -> [b]) -> [b]
  li >>= f = concatMap f li
```

```
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]

> concatMap (return . product) [[1,2],[3,4],[5,6]]
```

List is Monadic

```
> concatMap f xs
```



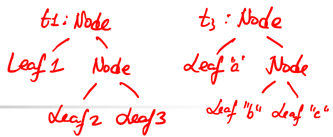
[e1, e2, e3, … en]

f f f f Map part

[[r1,r2], [r3],[], … [r4,r5,r6]]

concat part

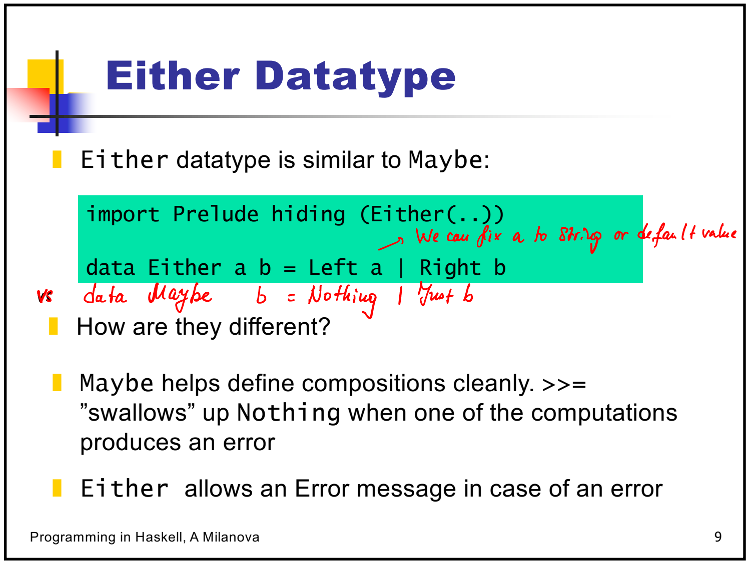
[r1, r2, r3, … r4, r5, r6]



- zipTree to zips two trees. If trees not isomorphic, return Nothing

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
      deriving (Eq, Show)
t1 = Node (Leaf 1) (Node (Leaf 2)(Leaf 3))
t2 = Node (Leaf 'a') (Node (Node (Leaf 'b') (Leaf 'c')) (Leaf 'd'))
t3 = Node (Leaf 'a') (Node (Leaf 'b')(Leaf 'c'))

zipTree :: (Tree a) -> (Tree b) -> Maybe (Tree (a,b))
zipTree (Leaf x) (Leaf y) = return (Leaf (a,b))
zipTree (Node l1 r1) (Node l2 r2) = do
    l' <- zipTree l1 l2
    r' <- zipTree r1 r2
    return (Node l' r')
zipTree _ _ = Nothing
```

```
(zipTree l1 l2) >>= (\l' ->
   (zipTree r1 r2) >>= (\r' ->
       return (Node l' r')))
```



Either Datatype

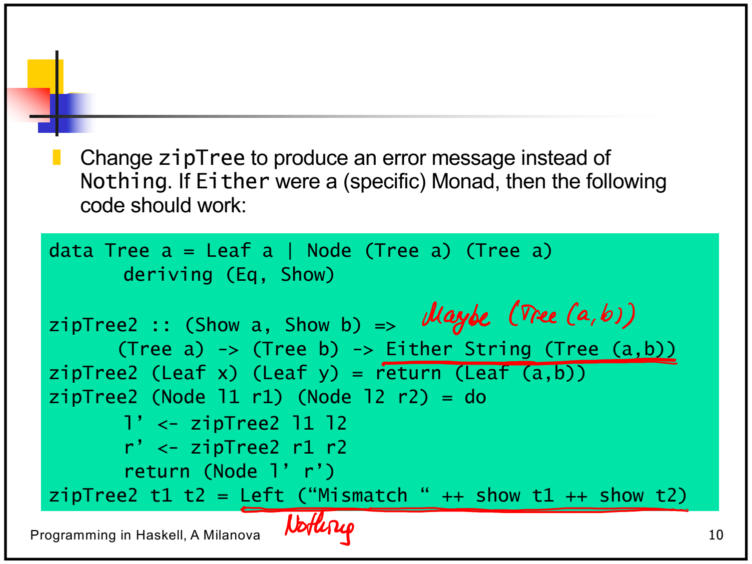
- `Either` datatype is similar to `Maybe`:

```
import Prelude hiding (Either(..))
                          → We can fix a to String or default value
data Either a b = Left a | Right b
```

vs data Maybe b = Nothing | Just b

- How are they different?
- `Maybe` helps define compositions cleanly. >>= "swallows" up `Nothing` when one of the computations produces an error
- `Either` allows an Error message in case of an error

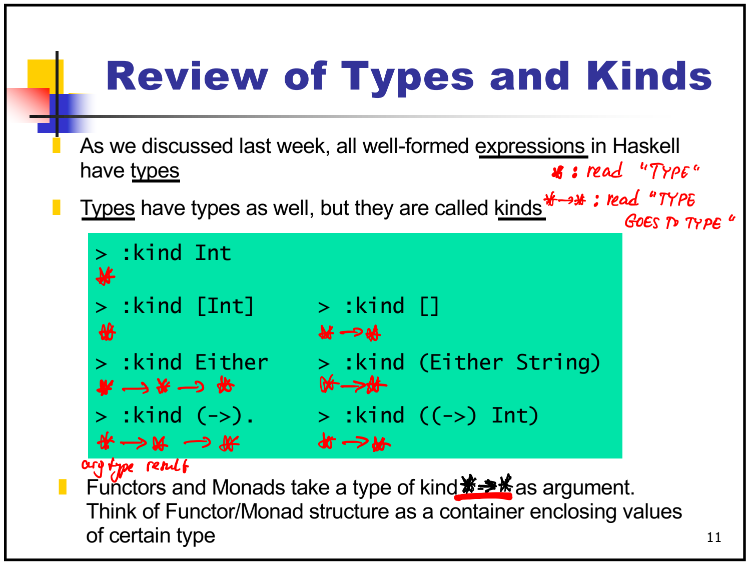
9



- Change `zipTree` to produce an error message instead of `Nothing`. If `Either` were a (specific) Monad, then the following code should work:

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

zipTree2 :: (Show a, Show b) =>    Maybe (Tree (a,b))
     (Tree a) -> (Tree b) -> Either String (Tree (a,b))
zipTree2 (Leaf x) (Leaf y) = return (Leaf (a,b))
zipTree2 (Node l1 r1) (Node l2 r2) = do
      l’ <- zipTree2 l1 l2
      r’ <- zipTree2 r1 r2
      return (Node l’ r’)
zipTree2 t1 t2 = Left (“Mismatch “ ++ show t1 ++ show t2)
                 Nothing
```

10



Review of Types and Kinds

- As we discussed last week, all well-formed expressions in Haskell have types
- Types have types as well, but they are called kinds

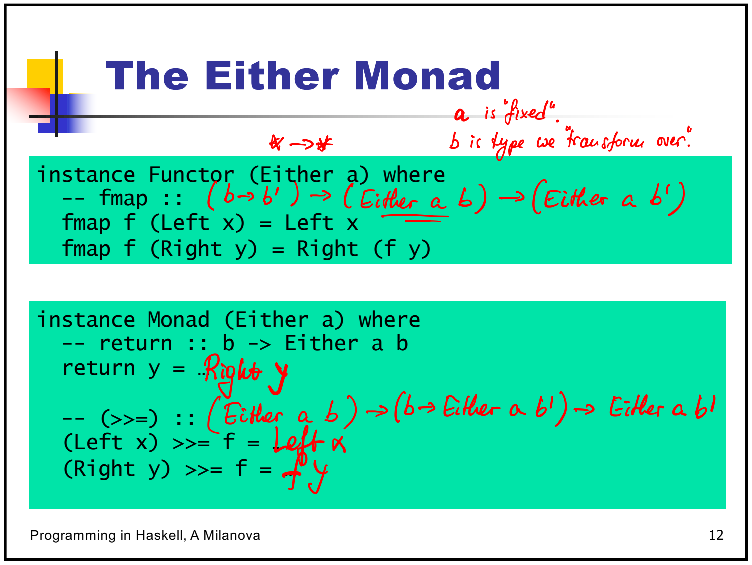
* : read "TYPE"

* → * : read "TYPE GOES TO TYPE"

```
> :kind Int
*

> :kind [Int]           > :kind []
*                       * -> *

> :kind Either          > :kind (Either String)
* -> * -> *             * -> *

> :kind (->).           > :kind ((->) Int)
* -> *  -> *            * -> *
arg type  result
```

- Functors and Monads take a type of kind * -> * as argument. Think of Functor/Monad structure as a container enclosing values of certain type

11



The Either Monad

a is "fixed".
b is type we "transform over".

* → *

```
instance Functor (Either a) where
  -- fmap ::  (b -> b’) -> (Either a b) -> (Either a b’)
  fmap f (Left x) = Left x
  fmap f (Right y) = Right (f y)
```

```
instance Monad (Either a) where
  -- return :: b -> Either a b
  return y = ..Right y

  -- (>>=) :: (Either a b) -> (b -> Either a b’) -> Either a b’
  (Left x) >>= f = Left x
  (Right y) >>= f = f y
```

12

Note:

- To make above code work, you'll need to define an instance of the Applicative functor for (`Either a`), Use the implementation from Control.Monad

---

Exercise

arg type a is "fixed"
result type b is one we transform into a b'

- What about the partially applied function type ((->) a)? Let's define Functor and Monad instances for it

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

```
instance Monad ((->) a) where
  -- return :: a -> b
  return b = \_ -> b
                fun                  f                  result of >>= composition
  -- (>>=) :: (a -> b) -> (b -> (a -> b')) -> (a -> b')
  fun >>= f = \x -> let arg = fun x in
               a
                      f arg   x
                      a->b'   a
                         b'
                     a -> b'        type, just what we need
```

State Transformations

- Now, even more Monads! Going back to Tree, how can we count number of leaves in a tree?

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
      deriving (Eq, Show)
t1 = Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
```

```
countF :: Tree a -> Integer
countF (Leaf _) = 1
countF (Node l r) = countF l + countF r
```

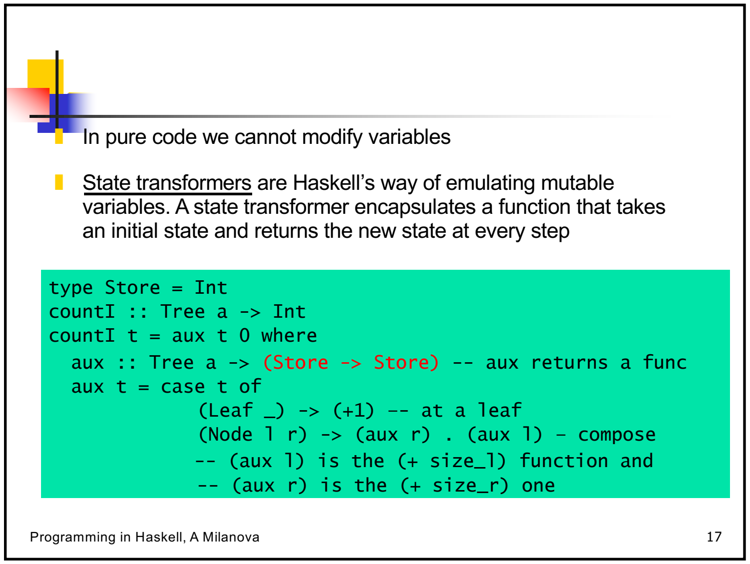
- Or we can just call length assuming Tree is Foldable (as they all should be)

- The count is mutable "state". In C or Java we might create a local variable and simply increment this variable as we walk over tree
- Interestingly, you can do this in Haskell (but you shouldn't). The `Data.IORef` module allows mutable variables

  a -> IO (IORef a)   IORef a -> IO a
- `newIORef` creates a new mutable variable, `readIORef` reads variable and `writeIORef` writes it, and `modifyIORef` updates it

  IORef a -> a -> IO ()

```
countIO :: Tree a -> IO Int
countIO t = do
    count <- IO.newIORef 0 -- create a mutable var
    let aux t = case t of
       (Leaf _) -> IO.modifyIORef count (+1)
       (Node l r) -> do
              aux l                 aux :: Tree a -> IO ()
              aux r
    aux t
    IO.readIORef count
```

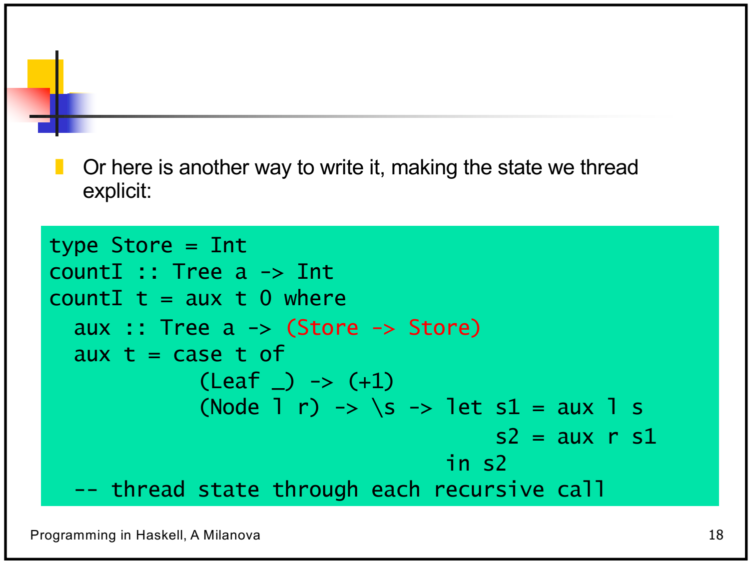


- In pure code we cannot modify variables
- State transformers are Haskell's way of emulating mutable variables. A state transformer encapsulates a function that takes an initial state and returns the new state at every step

```
type Store = Int
countI :: Tree a -> Int
countI t = aux t 0 where
  aux :: Tree a -> (Store -> Store) -- aux returns a func
  aux t = case t of
              (Leaf _) -> (+1) -- at a leaf
              (Node l r) -> (aux r) . (aux l) – compose
              -- (aux l) is the (+ size_l) function and
              -- (aux r) is the (+ size_r) one
```

Programming in Haskell, A Milanova 17

17

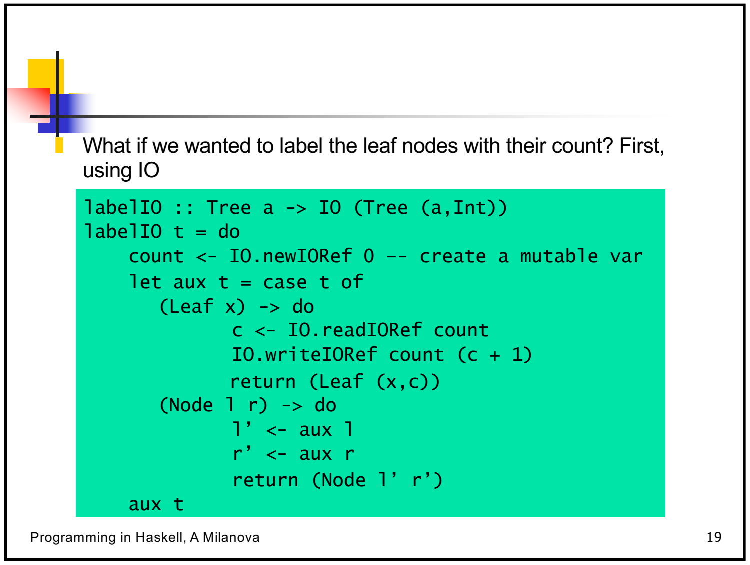


- Or here is another way to write it, making the state we thread explicit:

```
type Store = Int
countI :: Tree a -> Int
countI t = aux t 0 where
  aux :: Tree a -> (Store -> Store)
  aux t = case t of
            (Leaf _) -> (+1)
            (Node l r) -> \s -> let s1 = aux l s
                                    s2 = aux r s1
                                in s2
  -- thread state through each recursive call
```

Programming in Haskell, A Milanova 18

18

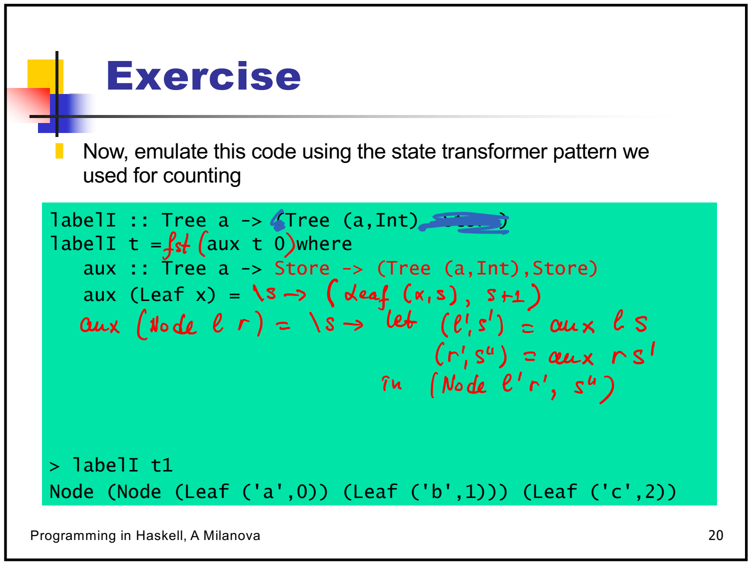


- What if we wanted to label the leaf nodes with their count? First, using IO

```
labelIO :: Tree a -> IO (Tree (a,Int))
labelIO t = do
    count <- IO.newIORef 0 -- create a mutable var
    let aux t = case t of
        (Leaf x) -> do
              c <- IO.readIORef count
              IO.writeIORef count (c + 1)
              return (Leaf (x,c))
        (Node l r) -> do
              l' <- aux l
              r' <- aux r
              return (Node l' r')
    aux t
```

Programming in Haskell, A Milanova 19

19



Exercise

- Now, emulate this code using the state transformer pattern we used for counting

```
labelI :: Tree a -> ~~(~~Tree (a,Int)~~,Store)~~
labelI t = fst (aux t 0)where
   aux :: Tree a -> Store -> (Tree (a,Int),Store)
   aux (Leaf x) = \s -> (Leaf (x,s), s+1)
   aux (Node l r) = \s -> let (l',s') = aux l s
                              (r',s") = aux r s'
                          in (Node l' r', s")


> labelI t1
Node (Node (Leaf ('a',0)) (Leaf ('b',1))) (Leaf ('c',2))
```

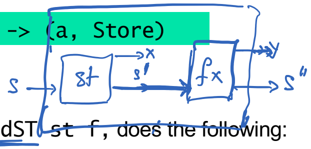
Programming in Haskell, A Milanova 20

20

State Transformer ST

- A state transformer is a function that takes the current store and returns a tuple of a value and a store (the effects of the function)

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



- This looks like a monad!
- bind operation, denoted by `bindST st f`, does the following:

1. Applies state transformer `st` on the incoming store `s` producing value and a new store `(x,s')`
2. Applies `f` on the result value, giving a new state transformer `(f x)`
3. Applies `(f x)` on intermediate state `s'` producing a new value and a new store `(y,s'')`

- At the end, result of `bindST` is a new state transformer which is the composition of `st` and the transformation of `f`

- Now download **Lecture11.hs** from course website and make sure you code as we move on through lecture

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

- ST is a monad. So let us define return and bind

```
returnST :: a -> ST a
-- takes "a" and returns a function Store -> (a, Store)
returnST x = \s -> (x, s)    -- returnST x s = (x,s)
```

In homework we have inferTypes exp index returning (subst, type, index'). Index should be handled with a state transformer!

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

- ST is a monad. So let us define return and bind

st
f

```
bindST :: ST a -> (a -> ST b) -> ST b
-- (Store -> (a, Store)) -> (a -> (Store -> (b, Store)) -> (Store -> (b, Store))
-- takes "Store -> (a, Store)" function (i.e., the ST a monad),
-- takes "(a -> (Store -> (b, Store)))" function (i.e., the f)
-- returns "Store -> (b, Store)" function (i.e., the new ST b monad)
-- Importantly, second transformer takes into account result of first one
bindST st f = \s -> let (x, s') = st s in
                    (f x) s'
```

Exercise

- Rewrite labeling function using `returnST` and `bindST`
- Rewrite only the Node clause, leave the Leaf clause as is

```
label2 :: Tree a -> Tree (a, Int)
label2 t = fst (aux t 0) where
  aux :: Tree a -> ST (Tree (a,Int))
  aux (Leaf x) = \s -> (Leaf (x,s), s+1)
  aux (Node t1 t2) =
```

~~`\s -> let (t1', s') = aux t1 s`~~
~~`(t2', s'') = aux t2 s'`~~
~~`in (Node t1' t2', s'')`~~

```
bindST (aux t1) (\l' ->
  bindST (aux t2) (\r' ->
    returnST (Node l' r')))
```

- Towards defining a Monad instance

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

- Can we define the monad instance like this?

```
instance Monad ST where
  -- return :: a -> ST a
  return = returnST

  -- >>= :: ST a -> (a -> ST b) -> ST b
  st >>= f = bindST st f
```

- No. Types defined using `type` cannot be made into instances of classes. We need to redefine ST using `data` or `newtype` and a dummy constructor

Monad ST2

```
newtype ST2 a = S (Store -> (a, Store))

runState :: ST2 a -> (Store -> (a, Store))
runState (S f) = f
```

```
instance Monad ST2 where
  -- return :: a -> ST2 a
  return x = S (\s -> (x, s))

  -- >>= :: ST2 a -> (a -> ST2 b) -> ST2 b
  st >>= f = S (\s -> let (x,s') = runState st s in
                      runState (f x) s' )
```

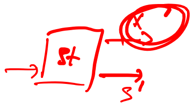
- In Haskell, an instance of Monad must also be an instance of Functor and Applicative (we'll cover Applicatives later)
- Since we don't need Applicatives, simply use definitions from Control.Monad:

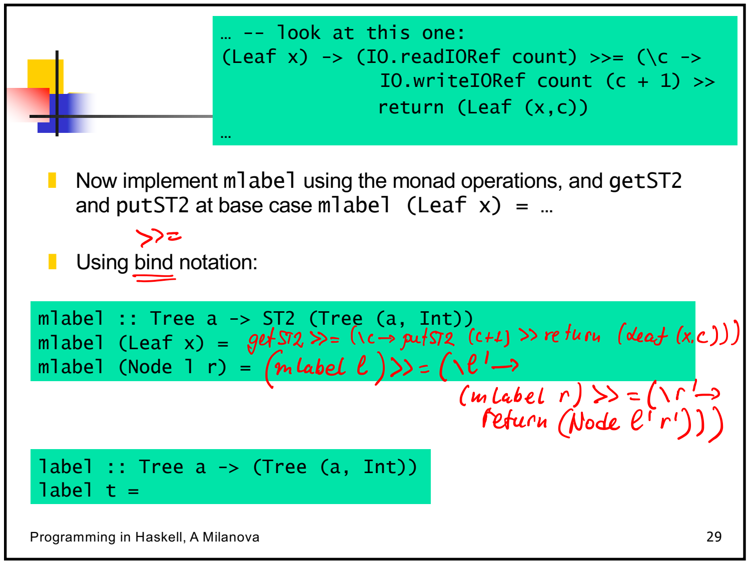
```
instance Functor ST2 where
  -- fmap :: (a -> b) -> ST2 a -> ST2 b
  fmap = liftM -- from Control.Monad
```

```
instance Applicative ST2 where
  -- pure :: a -> ST2 a
  pure = return

  -- (<*>) :: ST2 (a -> b) -> ST2 a -> ST2 b
  (<*>) = ap -- from Control.Monad
```

- Two useful functions, analogous to `readIORef` and `writeIORef`

```
getST2 :: ST2 Store
getST2 = S (\s -> (s, s))

putST2 :: Store -> ST2 ()
putST2 s = S (\_ -> ((), s))
```





```
… -- look at this one:
(Leaf x) -> (IO.readIORef count) >>= (\c ->
                     IO.writeIORef count (c + 1) >>
                     return (Leaf (x,c))
…
```

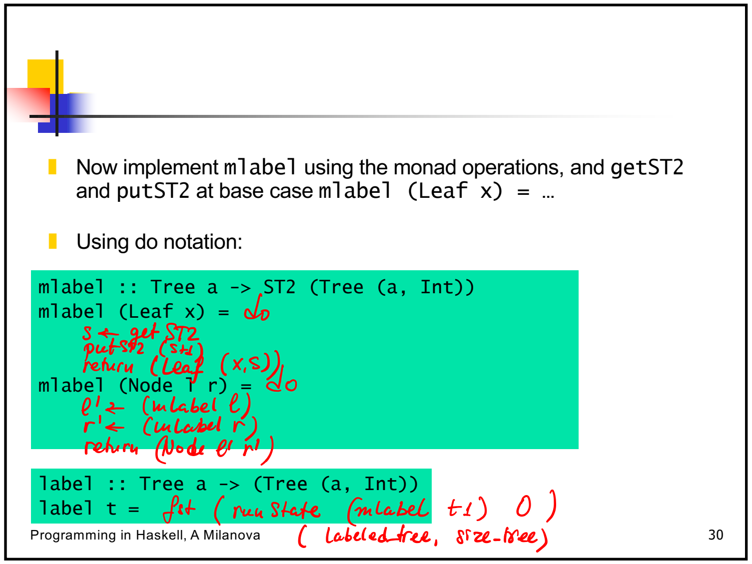
- Now implement `mlabel` using the monad operations, and `getST2` and `putST2` at base case `mlabel (Leaf x) = …`

>>=

- Using bind notation:

```
mlabel :: Tree a -> ST2 (Tree (a, Int))
mlabel (Leaf x) = getST2 >>= (\c -> putST2 (c+1) >> return (Leaf (x,c)))
mlabel (Node l r) = (mlabel l )>>= (\l' ->
                           (mlabel r) >>= (\r' ->
                            return (Node l' r')))
```

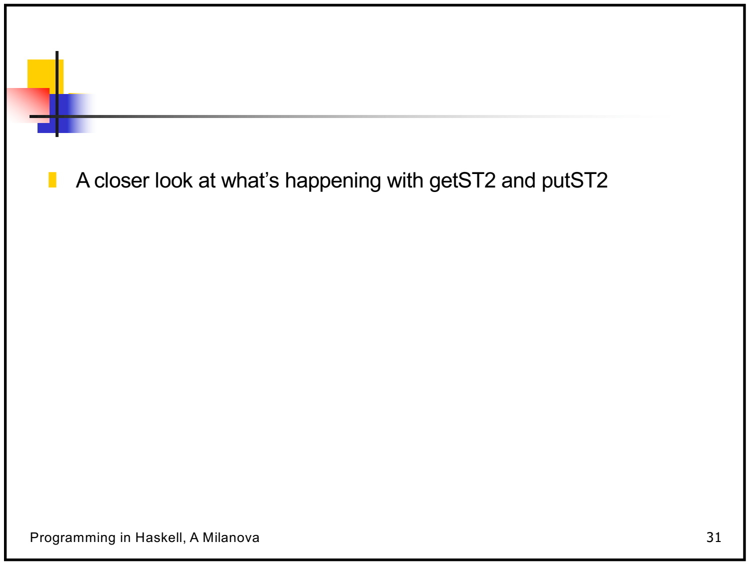
```
label :: Tree a -> (Tree (a, Int))
label t =
```



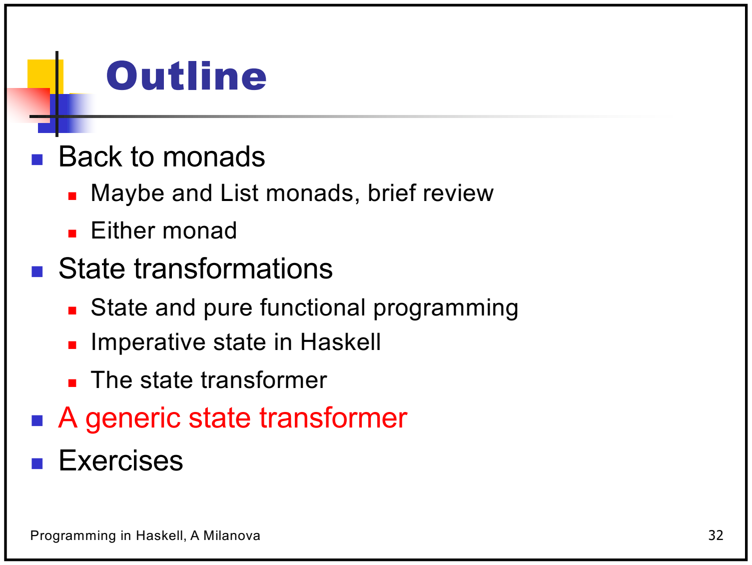
- Now implement `mlabel` using the monad operations, and `getST2` and `putST2` at base case `mlabel (Leaf x) = …`
- Using do notation:

```
mlabel :: Tree a -> ST2 (Tree (a, Int))
mlabel (Leaf x) = do
      s <- getST2
      putST2 (s+1)
      return (Leaf (x,s))
mlabel (Node l r) = do
      l' <- (mlabel l)
      r' <- (mlabel r)
      return (Node l' r')
```

```
label :: Tree a -> (Tree (a, Int))
label t = fst ( runState (mlabel t1) 0 )
                      ( labeled-tree, size-tree)
```



- A closer look at what's happening with getST2 and putST2



Outline

- Back to monads
  - Maybe and List monads, brief review
  - Either monad
- State transformations
  - State and pure functional programming
  - Imperative state in Haskell
  - The state transformer
- A generic state transformer
- Exercises

- In our examples we worked with an Int store
- In reality of course, the store is more complex, e.g., we may have more than one “mutable variables” whose values we need to update
- E.g., we may have

```
type Store = (Int, Int)
```

- Therefore, we’ll introduce a generic store and a (more) generic state transformer

Generic State Transformer

- Now download **State.hs**, the generic state transformer and **Lecture11’.hs** and again make sure you code as we move on

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

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

- And we’ll define the Monad with return and bind operations and the helper functions

```
                      *              * -> * -> *
                  State s a   or   State
instance Monad (State s) where
  -- return :: a -> (State s a)

  return x = S (\s -> (x, s))

  -- >>= :: (State s a) -> (a -> (State s b)) -> (State s b)

  st >>= f = …
```

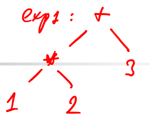
- `get` (modeled after `readIORef`, to retrieve value of state), `put` (modeled after `writeIORef`, to “write” value of state), and `modify`

```
get :: State s s
get = S (\s -> (s, s))

put :: s -> State s ()
put s = S (\_ -> ((), s))

modify :: (s -> s) -> State s ()
modify f = do
  s <- get
  put (f s)
-- or get >>= (\s -> put (f s))
```

- Let's rewrite labeling function in terms of the generic transformer

```
mlabelS :: Tree a -> S.State Int (Tree (a, Int))
mlabelS (Leaf x) = do
      c ← S.get
      S.put (c+1)
      return (Leaf (x, c))
mlabelS (Node l r) = do
    l' ← (mlabelS l)
    r' ← (mlabelS r)
    return (Node l' r')
> S.runState (mlabelS t1) 0
…
> S.runState (mlabelS t2) 100
… (…      , 104)
```

Exercise



- Simple state

```
data Exp = Lit Int | Add Exp Exp | Mul Exp Exp deriving (Show,Eq)
exp1 = Add (Mul (Lit 1) (Lit 2)) (Lit 3)
exp2 = Mul exp1 exp1

evalExp :: Exp -> S.State Int Exp
-- store is Int and value is Exp
evalExp ~~= undefined~~ (Lit i) = do
   S.put i
   return (Lit i)

evalExp (Add l r) = do
    l' ← (evalExp l)
    v1 ← S.get
    r' ← (evalExp r)
    v2 ← S.get
    S.put  v1+v2
    return (Add l' r')

> S.runState (evalExp exp1) 0
(Add (Mul (Lit 1) (Lit 2)) (Lit 3),5)
> S.runState (evalExp exp2) 0
(Mul …, 25)
```

Exercise

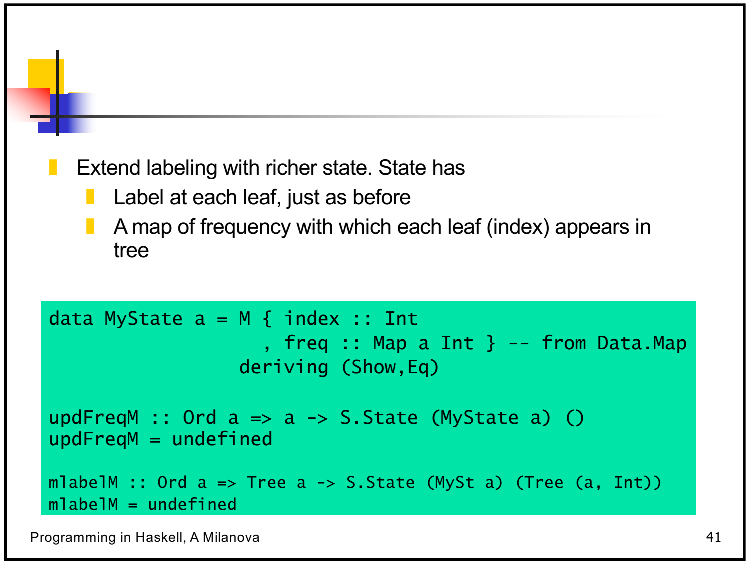
- Simple state

```
lenS :: [a] -> S.State Int Int
-- store is Int and value is Exp
lenS [] = …




> S.runState (len [1,2,3]) 0
(3,3)
```

Exercise

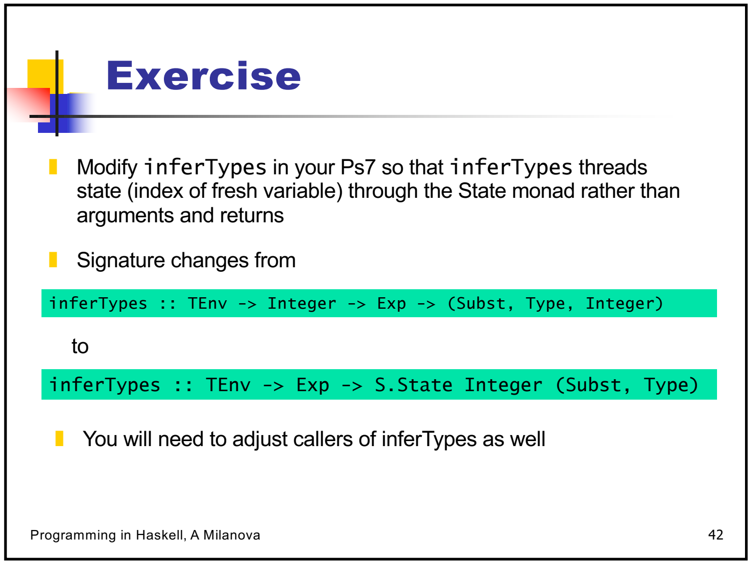
- Extend labeling with richer state. State has
  - Label at each leaf, just as before
  - A map of frequency with which each leaf (index) appears in tree

```
data MyState a = M { index :: Int
                   , freq :: Map a Int } -- from Data.Map
                 deriving (Show,Eq)

updIndexM :: S.State (MyState a) Int
updIndexM = do
  m <- S.get
  let i = index m
  S.put (m{index = i + 1})
  -- create a new record like m, but index as given
  return i
```



- Extend labeling with richer state. State has
  - Label at each leaf, just as before
  - A map of frequency with which each leaf (index) appears in tree

```
data MyState a = M { index :: Int
                   , freq :: Map a Int } -- from Data.Map
                 deriving (Show,Eq)

updFreqM :: Ord a => a -> S.State (MyState a) ()
updFreqM = undefined

mlabelM :: Ord a => Tree a -> S.State (MySt a) (Tree (a, Int))
mlabelM = undefined
```



Exercise

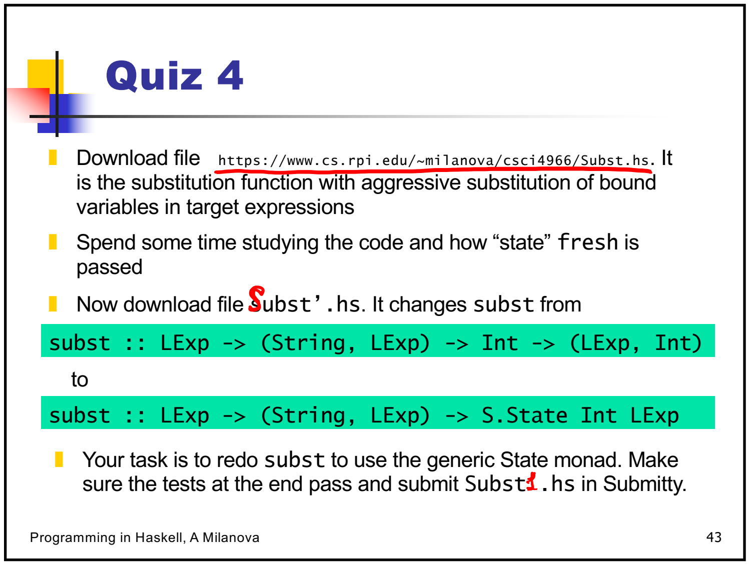
- Modify `inferTypes` in your Ps7 so that `inferTypes` threads state (index of fresh variable) through the State monad rather than arguments and returns
- Signature changes from

```
inferTypes :: TEnv -> Integer -> Exp -> (Subst, Type, Integer)
```

to

```
inferTypes :: TEnv -> Exp -> S.State Integer (Subst, Type)
```

- You will need to adjust callers of inferTypes as well



Quiz 4

- Download file `https://www.cs.rpi.edu/~milanova/csci4966/Subst.hs`. It is the substitution function with aggressive substitution of bound variables in target expressions
- Spend some time studying the code and how "state" `fresh` is passed
- Now download file `Subst'.hs`. It changes `subst` from

```
subst :: LExp -> (String, LExp) -> Int -> (LExp, Int)
```

to

```
subst :: LExp -> (String, LExp) -> S.State Int LExp
```

- Your task is to redo `subst` to use the generic State monad. Make sure the tests at the end pass and submit `Subst1.hs` in Submitty.