Haskell

Type Classes

Monads

Algebraic Data Types (ADTs) (quick review)

Definition

An algebraic data type declares a new type, and provides one or more ways to create a value in the type.

- The datatype can have type parameters. (Example: Tree **a**)
  - these are parametric polymorphism, like Java's generics.
- Each shape of value has a constructor and 0+ arguments (listed by type)
  - A constructor is really a function, e.g. **RBG :: Int ⟶ Int ⟶ Int ⟶ Color**
  - instances provide the implementations of a type class for a specific type.

```
data Bool = True | False

data Coin = Quarter | Dime | Nickel | Penny
data Color = Green | Gold | RBG Int Int Int
data IntList  = ILCons Int IntList | EmptyIL
data MyList a = Cons a (MyList a)  | EmptyList
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

datatype usage

- We write functions over the new datatype.
- We use pattern matching to cover all expected shapes of values

```
data IntTree = Leaf | Br IntTree Int IntTree

sumIT :: IntTree ⟶ Int
sumIT Leaf = 0
sumIT (Br left v right) = (sumIT left) + v + (sumIT right)
```

See: adts in ClassCode2.hs

Type Classes

Definition

A type class declares a group of methods that can be provided at specific types.

instances provide the implementations of a type class for a specific type.

```
class Show a where
    show :: a ⟶ String

data IntPair = IP Int Int

instance Show IntPair where
    show (IP x y) = "("++(show x)++","++(show y)++")"
```

→ ***These are quite similar in purpose to Java's interfaces: both are ad-hoc polymorphism***

Example

Given a datatype that represents ordering, we can create a type class, Ord, that understands how to order anything given an instance:

```
data Ordering = LT | GT | EQ

class Ord a where
    compare :: a -> a -> Ordering

instance Ord Int where
    compare a b | a<b  = LT
                | a==b = EQ
                | otherwise = GT
```

The real Ord has many more methods, not just compare.

More Instances

We can provide instances for any specific types we want, whether it's a type we created or a type that is already available (as with `Int`). We can also rely upon other instances of the typeclass in the process.

```
data Color = RGB Int Int Int

instance Ord Color where
  compare (RGB a b c) (RGB x y z)
      | (a+b+c)<(x+y+z)  = LT
      | (a+b+c)==(x+y+z) = EQ
      | otherwise        = GT
```

Class Constraints

If you expect something to be usable via a typeclass instance as part of some other code, that can be added as a **class constraint** on a type variable (before the `=>`).

Example: pairs can be ordered by each successive element, but we need to know the elements are themselves orderable:

```
instance (Ord a, Ord b) => Ord (a,b) where
    compare (a,b) (c,d) = case compare a c of
                            EQ -> compare b d
                            LT -> LT
                            GT -> GT
```

Deriving Instances

Some built-in type classes have obvious instances that could be provided automatically.

- **Show** and **Eq** are two such candidates. But we can't "override" (redefine) an instance, so by default they aren't provided.
- To request the defaults, we add those deriving clauses:

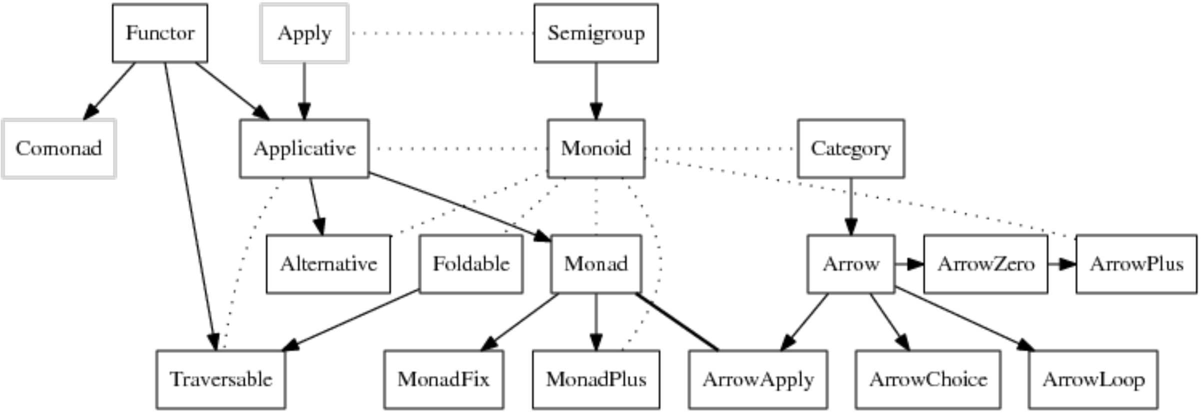
```
data IntPair = IP Int Int     deriving (Show, Eq)

data RoseTree = R Int | B Int [RoseTree]     deriving (Show, Eq)

data Option3 a b c = One a | Two b | Three c deriving (Show, Eq)
```

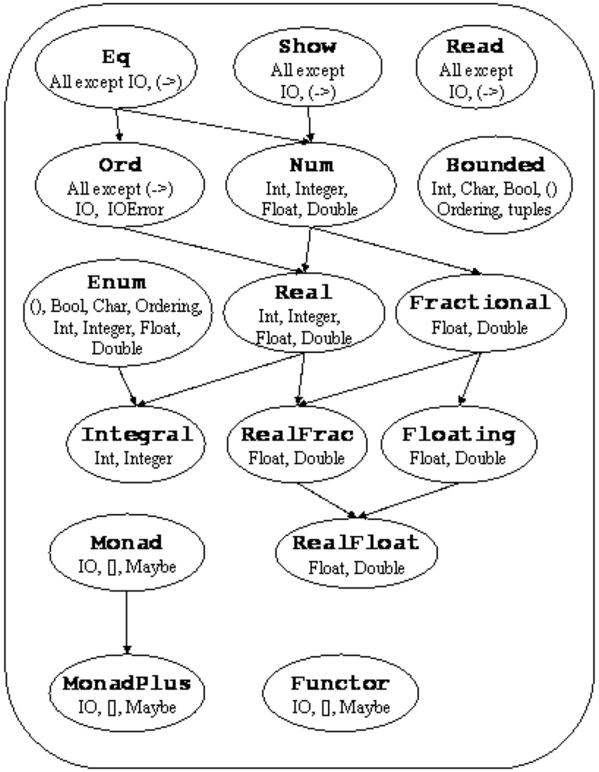
Available deriving classes

- **Eq**: provides (==) and (/=).
- **Ord**: comparisons. (<) (<=) (>) (>=) compare
- **Show**: convert things to strings via **show::a→String**
- **Read**: parses String to target type. (you need ascriptions to tell it what to expect). **read::String → a**
- **Enum, Bounded**: deals with enumerations and [a..b] syntax
- ***others:*** Sometimes you can add a language pragma (such as `{-# LANGUAGE DeriveDataTypable #-}`) in order to derive specific extra class instances, like `Data.Data` or `Data.Typeable`.
- **reach goal:** write your own! Explore `Data.Derive`, allowing you to replace what default instance is used or to add more self-defined patterns of default instances.

More Type Classes

- http://learnyouahaskell.com/types-and-typeclasses#typeclasses-101
- http://learnyouahaskell.com/making-our-own-types-and-typeclasses
- https://wiki.haskell.org/Typeclassopedia
- https://en.wikibooks.org/wiki/Haskell/Classes_and_types
- https://www.haskell.org/onlinereport/basic.html





More Prelude type classes

`Num`: represents numeric things. Definition:

```
class Num a where
    (+), (*), (-) :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger :: Integer -> a

class (Num a) => Fractional a where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod :: a -> a -> (a,a)
    toInteger :: a -> Integer
```

http://hackage.haskell.org/package/base-4.7.0.2/docs/Prelude.html#t:Num

Functor

the map function is specifically for lists:

map :: (a→b) → [a] → [b]

It navigates the list structure, applying the function to each spot.

Functors are any structure that has "spots" akin to the items in a list.
Some examples: values in tree structures; values in Maybe types.

```
class Functor f where
   fmap :: (a→b) → f a → f b
```

More typeclass examples:
typeclasses/TypeClasses*.hs

Monads

TODO:

- Read to/through the LYAHFGG materials on monads (chapters 12, 13)
- Read the RWH materials on monads (CH 7, 14, and more)
- Play with IO
- Play with Maybe

Idea (Not the Definition)

- **Monads** are used to *represent* a computation.
  - we compose pieces of represented computation together into larger computation
- We want to have a more direct way to simulate other kinds of computation. We will use monads to model extra computational features not directly implemented in Haskell.
  - Ultimately, uses `>>=` and `return` operations
  - but `do`-notation makes it prettier
- Example of representing a computational style: our lambda calculus implementations in Haskell provided a model of computation via eval.

```
eval :: Tm → Val
```

Monad Examples

- **IO** Monad: ***implements*** side effects: file I/O, user interaction, updatable variables, and more. *This one is special – you can't peek under the hood of this one!*
- **Maybe** Monad: simulates chains of steps that might not generate a value
- **Error** Monad: simulates failures (like explicit exception handling)
- **List** Monad: simulates non-determinism.
- **State** Monad: simulates having an updatable variable and sequential operations.
- **Reader** Monad: simulates having an environment of info (like Γ in $\lambda_{\rightarrow}$)
- **Writer** Monad: simulates having a "logger"/console to emit values to while evaluating.

Monad Implementations: a peek

- Generally will use datatype values to represent that monad's computations
  - Your code can use these values to build up expressions.
- Provides instance of class **Monad**
  - Allows chaining expressions together
  - Allows do-notation (convenient syntax)

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

- Probably provides a typeclass that embodies common operations for that monad. Often this is used more than the ADT directly!
- Needs a "run-" method that lets us take an expression representing a computation in that monad, plus any other needed starting info, and simulates the computation described.
  - Think of our **eval** function over Tm in our lambda calculus implementation.

We explore some example usage first; then we'll peek at some implementations.

Running Monads *(crib sheet)*

State Monad: simulate having an updatable variable and sequential operations.

```
runState  :: (State  s a) → s → (a,s)
```

- *Given a State computation and a starting state, run the computation based on that initial state. Give back the resulting answer and updated state.*

Reader Monad: simulate having an environment of info (like Γ in $\lambda_{\rightarrow}$)

```
runReader :: (Reader r a) → r → a
```

- *Given a Reader computation and a starting environment, what is the resulting value from the represented expression?*

Writer Monad: simulates having a "logger"/console to emit values to while evaluating.

```
runWriter :: (Writer w a) → (a,w)
```

- *Given a Writer computation, evaluate the answer as well as what , what is the resulting value from the represented expression?*

Maybe Monad: simulates chains of fail-possible steps.

(no runMaybe::(Maybe a)⟶ a . Maybes are already in the language!)

IO Monad: *implements* side effects: file I/O, user interaction, updatable variables, and more.

(no runIO::IO a → a. Why is that unsafe? We'll learn…)

Monad: the Idea

- We build programs out of pieces of computation, and selectively run the computation when the program is fully constructed.
  → it's like an embedded language
  → or like our lambda calculi, for extra features

- Example: statefulness (having variables to read/write)
  → e.g. access to a `[(String,Value)]` list throughout a function

- Multiple statements are often sequenced by semi-colon in imperative languages; we similarly chain operations together (via "bind", `>>=`)

Special Case: The IO Monad

- all side effects* are relegated to the IO monad – to operations such as:
  - `getLine :: IO String`
  - `putStr, putStrLn :: String -> IO ()`
  - `readFile :: FilePath -> IO String`

- You can tell, *just by the type*, which functions will lead to side-effects, because they'll have **IO** in the *return-type*.

Sample IO program

(preferred do-notation)

```
main :: IO ()
main = do
  putStrLn "what is your name?"
  name <- getLine
  putStrLn "how old are you?"
  ageStr <- getLine
  let age = read ageStr::Int
  putStr $ (map toUpper name)
  putStrLn $ ": you're nearly "++(show (age+1))
```

file: IOStuff.hs

Sample IO program

(using >>= directly, "normal" layout)

```
main :: IO ()
main =
  putStrLn "what is your name?"
  >>= (\ _ ->
      getLine
      >>= (\ name ->
        putStrLn "how old are you?"
        >>= (\ _ ->
          getLine
          >>= (\ ageStr ->
            let age = read ageStr::Int
            in putStr (map toUpper name)
                >>= (\ _ ->
                  putStrLn $ ": you're nearly "++(show (age+1))
      )   ) ) )      )
```

file: IOStuff.hs

Sample IO program

(using >>= directly – with layout mimicking the do-notation)

```
main :: IO ()
main =
  putStrLn "what is your name?" >>= (\ _      ->
  getLine                       >>= (\ name   ->
  putStrLn "how old are you?"   >>= (\ _      ->
  getLine                       >>= (\ ageStr ->
  let age = read ageStr::Int in
  putStr (map toUpper name)     >>= (\ _      ->
  putStrLn $ ": you're nearly "++(show (age+1))
 )))))
```

file: IOStuff.hs

Performing IO in Haskell

- Write any pure functions as before, without the IO monad.
  - try to ***write as much of the program as you can in pure style***

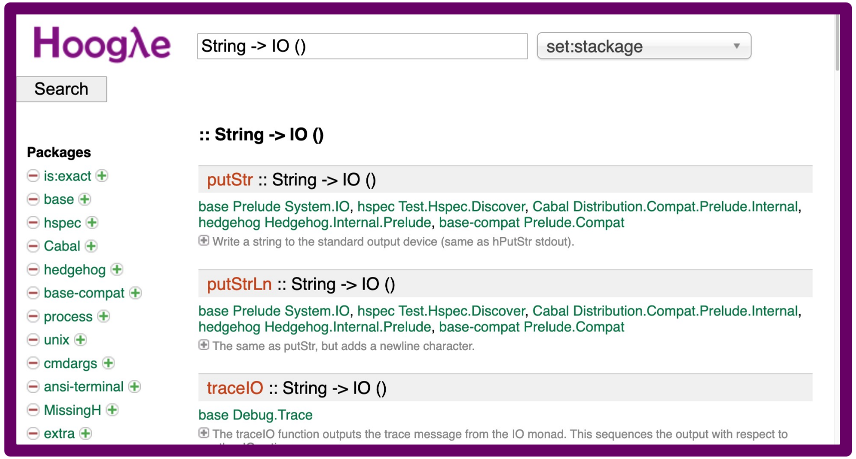
- write one **main::IO()** function (plus helpers) that deals with the user/files/world.
  - sequence your pure calculations with IO actions
  - imperative code often has the same pattern – objects hide their implementations internally, and there's probably only one main() method that interacts with the outside world

- **Any side-effectful function will have IO in its (return) type.**

Finding Definitions

- Looking for specific functions? Hoogle is your friend: https://hoogle.haskell.org/

- Guess the type.
- search for example "**String -> IO ()**"
- we found the `putStrLn` function!
- We also realize we'll need to `import System.IO`



Monads Behind the Scenes

- chain multiple operations together to compose more complex operations.
  - with "bind", `>>=`, or preferably with do-notation
- The chosen monad defines what chain operations mean.
- the implementation of `>>=` is the crucial bit that defines the special features present. It accepts two arguments to build a larger computation:
  - an initial computation (`:: m a`)
  - function from one value (result of "running" that initial computation) to define a new computation that could be run (`a -> m b`)
- A bind of both arguments is just some computation (`:: m b`)
- `do`-syntax and syntactic sugar hides most `>>=` operators (nice!)

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

Impetus for the Maybe Monad:

multiple failworthy actions

- Consecutive calculations that each may fail can require multiple case-exprs over a Maybe value.
- helper functions might not be sufficient/desirable to avoid this linear indentation.

writing this code (without the fail-response notion) is annoying

```
smallerMaxCasey :: ([Int],[Int]) -> Maybe Int
smallerMaxCasey (xs,ys) =
    case maybeMax xs of
        Nothing     -> Nothing
        Just xsMax -> case maybeMax ys of
                          Nothing     -> Nothing
                          Just ysMax -> Just (min xsMax ysMax)
```

file: MaybeMonad.hs

Maybe, as a Monad

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

hiding in the Prelude somewhere…
https://hackage.haskell.org/package/base-4.19.0.0/docs/Data-Maybe.html#t:Maybe

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

- have the value we want to return? **Just** return it.
- want to chain two operations together?
  - if the first one gave us **Nothing**, we don't care what the second operation was – the whole process failed, and the answer is **Nothing**.
  - if the first one gave us **Just** the value **x**, we can feed it to the second operation to find out the overall answer.

- chances are, the "second operation" is itself a long chain of operations.
  - As long as it results in some **Maybe a** type, it'll work.

MaybeMonad.hs

Using >>= versus using return

(equivalent definitions, using different syntax styles)

```
smallerMaxBind :: ([Int],[Int]) -> Maybe Int
smallerMaxBind (xs,ys) =
  maybeMax xs >>= (\ maxX ->
                    maybeMax ys >>= (\ maxY ->
                                       Just (min maxX maxY)
                                     )
                  )

smallerMax :: ([Int],[Int]) -> Maybe Int
smallerMax (xs,ys) = do
    xsMax  <- maybeMax xs
    ysMax  <- maybeMax ys
    return  (min xsMax ysMax)
```

file: MaybeMonad.hs

Same functionality

- The cases, `>>=`, and do-notation versions all performed the same calculations.
- we explicitly indicate how to handle failures and continued calculations with cases/`>>=`, but the do-notation separates "how to perform chaining" code from the steps we're chaining.

→ see MaybeMonad.hs for more examples from the Maybe Monad.

⟶ here is a nice longform discussion on this "failure path" mentality:

https://fsharpforfunandprofit.com/rop/

see the slides or video on "Railway Oriented Programming"

Motivation:

The State Monad

Writing a (helper) function that threads through some background "state", which is sometimes used, sometimes updated for further calls, is a common pattern (see sumIter, maxH). This can be annoying.

this version has to manually thread through its 'state' (extra parameters)

```
sum xs = sumIter 0 xs

sumIter :: Int -> [Int] -> Int
sumIter n []      = n
sumIter n (x:xs) = sumIter (n+x) xs


maxL :: [Int] -> Maybe Int
maxL [] = Nothing
maxL (x:xs) = Just (maxH xs x)

maxH :: [Int] -> Int -> Int
maxH []      m = m
maxH (x:xs) m = if x > m  then (maxH xs x)  else (maxH xs m)
```

State, as a Monad

(don't get caught up on this – using State is more important than fully understanding its implementation for now)

```
data State s a = State (s -> (a,s))

# to run something that needs a state input, give it its input.
runState :: (State s a) -> s -> (a,s)
runState (State f) s = f s



instance Monad (State s) where
  # when given a state, just pass it through.
  return a = State (\s -> (a,s))
  # feed state to the first part; get the output state and feed to second part.
  (State f) >>= gm = State $ \s -> case (f s) of
                                    (val, s') -> runState (gm val) s'
```

Goals:

- identify how the input-state is abstracted out, and applied later.
- look for the chaining of multiple 'stateful' actions

Adding "non-proper morphisms"

Some data/state/value is being threaded through our calc.

- we want to **get** the current state (read it)
- we want to **put** a new current state (assign to it/replace it)
- already possible with bind, but we want convenient shorthand methods. We use a typeclass.

```
class MonadState s m | m -> s  where
  get :: m s
  put :: s -> m ()


instance MonadState s (State s) where
  get   = State $ \s -> (s,s)
  put s = State $ \_ -> ((),s)
```

m -> s : This is a __functional dependency__. "knowing type m dictates the type s."

Common Usage: State

- Using get and put, in do-notation, describe some computations that you'd like to do that are stateful. They'll have types like this:

```
compM :: arg1 ⟶ … ⟶ argN ⟶ State s a
```

- "run" the simulation at the top level (like a driver function), via runState, evalState, or execState, e.g.:

```
go :: args ⟶ a
go args = evalState (compM args) s_initial

go2 :: args ⟶ a
go2  args = case runState (compM args) s_initial of
              (lastval,laststate) -> lastval
```

Using get/put to create state-simulations

```
fibM :: Int -> State (Int,Int) Int
fibM 0 = do
    (a,b) <- get        # get current state (it's a pair of ints)
    return a            # answer is a.
fibM n = do
    (a, b) <- get       # get current state pair
    put (b, a+b)        # change stored state pair
    fibM (n-1)          # recurse, knowing state changed


fib :: Int -> Int
fib n = case runState (fibM n) (1,1) of
          # ignore our (a,b) state since we're done
          (ans, (a,b)) -> ans
```

Larger Example:
Stack Machine

- See **`StateExamples.hs`** for an example that builds a stack machine
  - first, without using State
  - we'll have many functions that will mirror aspects of the actual state monad
  - finally, we'll draw the parallels between the two representations, and view our work in the state monad.

  - if you squint, you could view it as something familiar:

>>= version	do-notation	imperative analog
expr >>= \n → …	name ← expr	name = expr;
$expr_1$ >> $expr_2$	expr expr	$stmt_1$; $stmt_2$

List Monad

trying all possibilities (non-determinism)

List comprehensions can be written in do-notation.

- generators are separate binding lines
- guards (filtering out unworthy values) use `guard :: Bool -> [()]`
- return value is the piece-wise result

list comprehension version

```
rightTri n =
   [(a,b,c)
   | a <- [1..n]
   , b <- [a..n]
   , c <- [b..n],
   , a*a+b*b==c*c
   ]
```

do-notation version

```
rightTri_do n = do
   a <- [1..n]
   b <- [a..n]
   c <- [b..n]
   guard $ a*a+b*b==c*c
   return (a,b,c)
```

More Monads!

- Reader – for maintaining an environment (like Γ)
- Writer – for sending values (messages) to a log
- Error – for representing error cases
- ...

- Multiple monads at once: you need monad transformers or other entirely different approaches (beyond the scope of this presentation…)