Not the Whole Story

There are some aspects of the typeclass system that haven't been discussed yet

- Some classes depend on other classes
- Some classes are themselves polymorphic
- Some classes are associated with type constructors

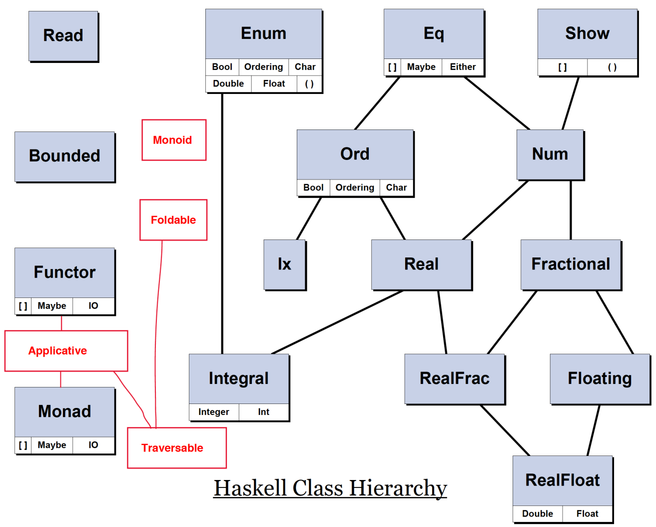
Classes based on other Classes

- Here is part of the class declaration for `Ord`:

```
class  (Eq a) => Ord a  where
compare              :: a -> a -> Ordering
(<), (<=), (>=), (>) :: a -> a -> Bool
max, min             :: a -> a -> a
compare x y
     | x == y    =  EQ
     | x <= y    =  LT
     | otherwise =  GT
```

- The notation `(Eq a) =>` is a *context*, stating that the `Ord` class depends on the `Eq` class (why?)
- In order to define `compare`, we have to use `==`
  - So, for a type to belong to `Ord`, it must belong to `Eq`
  - Think of it as a form of inheritance

Prelude Class Relationships



Saeed Jahed, 2009, updated by A. Butterfield, 2016 to include class additions/modification introduced in GHC 7.10

"Polymorphic" Type Classes (I)

How might we define an `Eq` instance for lists ?

- For `[Bool]`

```
instance Eq [Bool] where
  [] == []   =  True
  (b1:bs1) == (b2:bs2)  =  b1 == b2 && bs1 == bs2
  _ == _ = False
```

- For `[Int]`

```
instance Eq [Int] where
  [] == []   =  True
  (i1:is1) == (i2:is2)  =  i1 == i2 && is1 == is2
  _ == _ = False
```

- The red `==` above are where we use equality for `Bool` and `Int` respectively.
- Can't we do this polymorphically ?

“Polymorphic” Type Classes (II)

- We can !

```
instance (Eq a) => Eq [a] where
  [] == []   =  True
  (x1:xs1) == (x2:xs2)   =  x1 == x2 && xs1 == xs2
  _ == _ = False
```

- We can define equality on [a] provided we have equality set up for a
- Here we are defining equality for a type constructor ([] for lists) applied to a type a:
  - so the class refers to a type built with a constructor

Type-Constructor Classes

- Consider the class declaration for Functor

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

- Here we are associating a class with a *type-constructor* f
  - not with a type
  - See how in the type signature f is applied to type variables a and b.
  - So, f is something that takes a type as argument to produce a (different) type.

Type Constructor Examples

- The Maybe type-constructor

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

- The IO type-constructor

```
data IO a = ...
```

- The [] type-constructor
  The type we usually write as [a] can be written as [] a
  i.e. the application of list constructor [] to a type a.

Instances of Functor

- Maybe as a Functor

```
instance  Functor Maybe  where
  fmap f Nothing    =  Nothing
  fmap f (Just x)   =  Just (f x)
```

- [] as a Functor

```
instance Functor [] where
  fmap = map
```

- Both the above are straight from the Prelude.

Functor instance for Maybe, with annotations

In more detail, first a reminder of the class definition:

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

Next, the instance, with type annotations:

```
instance  Functor Maybe  where
  fmap (f :: a -> b) (Nothing :: Maybe a)
                                        =  Nothing :: Maybe b

  fmap f (Just (x :: a) :: Maybe a)
                                 = Just (f x :: b) :: Maybe b
```

Instances and Type Declarations

- A type can only have one instance of any given class. Why? Because each instance is a specific implementation. Which one shoud the compiler pick?
- A type synonym therefore cannot have its own instance declaration.
  `type MyType a = ...`
  It simply is a shorthand for an existing type
- A user-defined algebraic datatype can have instance declarations
  `data MyData a = ...`
  In general we need to do this for `Eq`, `Show` in any case
- A user-cloned (new) type can also have instance declarations
  `newtype MyNew a = ...`
  A key use of `newtype` is to allow instance declarations for existing types (now "re-badged").

Introducing "Monads"

- IO in Haskell uses the `IO` type constructor along with
  - Primitive I/O operations that return an "action value" of type `IO t`
  - I/O combinators "bind" `(>>=)`, "seq" `(>>)` and `return`.
- Haskell goes further, though. It uses Haskell's class system to leverage the key concepts.
- Type `IO t` is an instance of the so-called `Monad` class.
  - The term "monad" comes originally from Greek philosophy, more recent material from Leibniz, and even more recently from Category Theory.
- We shall see that the monad concept goes beyond I/O and has much wider utility.

Monads in Haskell

Monads in Haskell are represented by a type class:

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

Since `>>` can be defined in terms of `>>=` we usually only need to provide instances for `return` and `>>=`.

The fourth member of the class, `fail`, is an error handling operation which takes an error message and causes the chain of functions to fail, perhaps by using `error` to halt the program

Monads in Haskell, 7.10 onwards

In fact the declaration of the Monad class in Haskell now has the form:

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

- The annotation `(m :: * -> *)` simply says that `m` is a type-constructor, not a type (View `*` as representing a type argument or result).
- We shall ignore this for this course, as it has no effect on what is to come.

do is *syntactic sugar*

There is a mechanical translation from the do-notation form to the combinator form, which we can summarize:

```
do { a1 ; a2 ; .. ; an }
  ⇝ a1 >> do {a2 ; .. ; an }

do { x <- a1 ; a2 ; .. ; an }
  ⇝ a1 >>= \ x -> do {a2 ; .. ; an }

do a            ⇝       a
```

Note that above we show the full Haskell syntax for do-notation with explicit }, ; and }, rather than relying on the offside-rule.

The Monad laws

In order to retain the semantics that we want, any implementation of a monad is required to follow these rules:

```
(return v >>= f)    ==  f v

f >>= return        ==  f

(x >>= f) >>= g     ==  x >>= (\ a -> f a >>= g)
```

These laws are not checked by the compiler.

Any monad?

Any implementation of a monad?

Yes, monads represent something fundamental in computation, the idea of connecting two computations by sequencing them.

There are more monads than just `IO a`.

For example, another monad which we have already seen is `Maybe`!

Using the Maybe monad

Imagine a function:

```
f dict = case (lookup "foo" dict) of
           Nothing  ->  Nothing
           Just x   ->  case (lookup "bar" dict) of
                          Nothing  ->  Nothing
                          Just y   ->  Just (x,y)
```

We can clean this up because "Maybe" is a monad!

```
f dict
 = do x <- lookup "foo" dict
      y <- lookup "bar" dict
      return (x,y)
```

Let's think about how we can define `>>=` and `return` so that this code behaves like the code above.

The relevant definitions?

- We make the `Maybe` Type constructor an instance of the `Monad` class:
  `instance Monad Maybe where`
- To return a result we wrap it with `Just`:
  `return x = Just x`
- In bind, if a previous function returns `Nothing` we simply propagate this:
  `Nothing >>= f = Nothing`
- If a previous function returned `Just` something we apply the (monadic) function to it:
  `(Just x) >>= f = f x`
- If we want to report an error (fail), we produce `Nothing`:
  `fail s = Nothing`
- All of this is in the standard prelude

Maybe forms a monad?

- It represents the type of computations that may succeed or fail.
- More specifically, it combines actions by trying the first, and applying the second if the first succeeded (produced a `Just` result).
- `Maybe a` is the type of short-circuiting computations which can produce an `a`.
- There are no "side-effects" here — so monads are not just a way to hide those.

What's actually happening?

Let's *desugar* the monadic program and translate it into ordinary functions.

```
f dict
 = do x <- lookup "foo" dict
      y <- lookup "bar" dict
      return (x,y)
```

```
f dict = lookup "foo" dict >>= (\ x ->
         lookup "bar" dict >>= (\ y ->
         Just (x, y) ) )
```

What's actually happening?

What's going to happen if the first lookup fails?

```
f dict = Nothing >>= (\ x ->
         lookup "bar" dict >>= (\ y ->
         Just (x, y) ) )
```

```
f dict = Nothing
```

How about the second?

```
f dict = lookup "foo" dict >>= (\ x ->
         Nothing >>= (\ y ->
         Just (x, y) ) )
```

```
f dict = lookup "foo" dict >>= (\ x -> Nothing )
```

```
f dict = Nothing
```