Type classes

LYaH:

- Chapter 3 *Types and Type Classes*
- Chapter 8: *Making Our Own Types and Typeclasses*

Types: review

Base types: built-in atomic (no parts) types eg `Char`, `Int`, `Double`.

Ground types: built from basa types using type constructors e.g. (Char,Double) -> [Int]

Types: as above, but may include **type variables**, e.g. (a, Int) -> [b]

What do type variables mean? Polymorphism.

1. Any type, no restrictions: **universal polymorphism**.
   Eg `f :: a -> (a,a)`
   which means: for all types `a`, `f` has type `a -> (a,a)`
   ie: `f` has to work no matter what `a` is, without knowning anything about it

2. In Java, methods are polymorphic over subclasses: **subtype polymorphism**.
   Eg for `m` a method in class `C`, `m` polymorphic in `self`
   which means: for all `a ⊆ C` and `inst : a`, `inst.m` has the declared method type
   ie: `m` has to work for any subclass `a` of `C`.

3. In Haskell, we can constrain type variables: **constrained universal polymorphism**.
   Eg `f :: Eq a => a -> a -> Int`
   which means: for all types `a` that are in the type class `Eq`, `f` is in `a -> a -> Int`
   ie: the function can assume `==` is defined for type a `

In Haskell, constraints are *type classes*

A *simple type class* is defined by a set of "methods" (just functions, really).

```
class SomeTypeClass a where
    m1 :: T1  -- method m1 has type T1 (a type involving a)
    ...
    mn :: Tn  -- method mn has type Tn (a type involving a)
```

To declare a type `T` to be an *instance* of the type class:

```
instance SomeTypeClass T where
    m1 = ...  -- something of type T1 with a ≡ T
    ...
    mn = ...  -- something of type Tn with a ≡ T
```

Meaning of `SomeTypeClass`: the set of all of its instances.

Example: types with a null/default/zero value

```haskell
class Zero a where
    zero :: a

instance Zero Int where
    zero = 0

instance Zero [a] where
    zero = []

instance (Zero a, Zero b) => Zero (a,b) where
    zero = (zero, zero)

instance Zero Bool where
    zero = False
```

```haskell
instance Zero (Maybe a) where
    zero = Nothing

myLookup :: (Eq a, Zero b) => a -> [(a,b)] -> b
myLookup x l = case lookup x l of
    Just x -> x
    Nothing -> zero
```

Where's the code?

A polymorphic function can have different implementations.

How is the right one found? Could be found at runtime or compile time.

Java. Consider executing some method call `ob.m(17)`. What class declaring `m` is used?

Haskell. Consider evaluating `m ob 17`. What instance declaring `m` is used?

Finding the right instance in Haskell

```
class C a where
    op :: T

instance C [b] where
    op = ...

instance C (Maybe b) where
    op = ...
```

Consider a use of `op` in some typechecked program.

- It's context gives an expected type.
- There must some type `a = S` making `T` the same as the expected type.
- The instance to use is determined by the outermost constructor of `S`.
- If `S` is `[...]` then use the first instance; if it is `Maybe ...` then use the second.

```haskell
foo :: Int -> Bool
foo n =
  if n == zero
    then zero
    else null (n : zero)
```

Predefined ("built in") Eq class.

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    -- Minimal complete definition: either (==) or (/=)
    x /= y = not (x == y)
    x == y = not (x /= y)
```

Just the class is predefined. You can define your own instances.

E.g. for `data Set a = Set [a]`.

```haskell
data Set a = Set [a] deriving Eq  -- compiler can figure out a default implementation
                                  -- using list equality and assuming Eq a.
```

or

```haskell
data Set a = Set [a]

instance (Eq a) => Eq (Set a) where
  Set l0 == Set l1 = all (`elem` l1) l0 && all (`elem` l0) l1
```

Some handy type classes

All derivable in `data` definitions except as noted.

Type class	Operations (secondary)	Notes
Eq a	== (/=)	
Ord a	< (<=, ...)	Requires Eq
Show a	show (showList, ...)	
Read a	read (...)	(read "23") :: int ≡ 23
Enum a	succ, pred (toEnum, fromEnum)	for enum-like data types
Bounded a	maxBound::a, minBound::a	

Multiparameter type classes

A generalization of type classes.

Simple type classes	Multiparameter type classes
set of types	set of tuples of types
`class SomeClass a where`	`class SomeClass a b ... where`
available by default	requires language "pragma" in file

Example: converting one data representation to another

```haskell
class Convertible a b where
    safeConvert :: a -> ConvertResult b

type ConvertResult a = Either ConvertError a

convert :: Convertible a b => a -> b
convert x =
    case safeConvert x of
      Left e -> error (prettyConvertError e)
      Right x -> x
```

Instances of `Convertible`

```
instance Convertible a a where
    safeConvert x = Right x

instance Convertible a b => Convertible [a] [b] where
    safeConvert [] = Right []
    safeConvert (x:l) = do
        x' <- safeConvert x
        l' <- safeConvert l
        return $ x' : l'
```

Another generalization of classes

1. Done: classes as sets of types that have certain ops defined.
2. Done: classes as sets of *tuples* of types that have certain ops defined.
3. New: classes as sets of **type constructors**.

Notation: `*` stands for the *kind* of all types. Some Haskell versions use `type` for `*`.

```
Int           :: *
[Int]         :: *
[Int] -> Int  :: *
```

Notation: `* -> *` is the kind of all one-argument type constructors.

```
[]         :: * -> *    -- for any type a, [a] is a type, i.e. a ↦ [a]
Maybe      :: * -> *    -- a ↦ Maybe a
Tree       :: * -> *    -- where data Tree a = Leaf a | Node (Tree a) (Tree a)
((->) r)   :: * -> *    -- a  ↦  r->a
((,) r)    :: * -> *    -- a  ↦ (r,a)
```

The `Functor` class

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

instance Functor [] where
    fmap f l = map f l

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

data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node t0 t1) = Node (fmap t0) (fmap t1)
```

```haskell
class Functor ((,) r) where
    -- fmap :: (a -> b) -> (r,a) -> (r,b)
    fmap f (x,y) = (x, f y)

class Functor ((->) r) where
    -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = \x -> f (g x) -- = f . g
```

A puzzle for you. What does `fmap fmap fmap` do?

Monads

General definition of *monad*

```
class Monad m where
  (>>=) :: t a -> (a -> t b) -> t b  -- the "bind" operator -- ???
  (>>) :: t a -> t b -> t b          -- just a special case of the bind operator
  return :: a -> t a                 -- insert a value
```

What does this mean in general? Nothing!

We understand it through particular kinds of instances.

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

instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b   = ...
  return :: a -> IO a                    = ...
```

What `do` translates to in Haskell

```
do
    x1 <- e1                    e1 >>= (\x1 ->
    x2 <- e2                              e2 >>= (\x2 ->
    ...          ===>                                    ...
    xn <- en                                               en >>= (\xn ->
    e                                                                  e))...)
```

```haskell
instance Monad [] where
  -- (>>=) :: [a] -> (a -> [b]) -> [b]
  l >>= f = concat (map f l)
  -- return :: a -> [a]
  return x = [a]
```

A puzzle for you. What does the following return?

```haskell
do
  x <- [1,2,3]
  y <- [4,5,6]
  return (x,y)
```