# LECTURE 08

Theory and Design of PL (CS 538)

February 27, 2020

# NEWS

# HW1 GRADING

- Grading is in progress, back in a few days
- Style definitely matters
  - Don’t repeat yourself
  - Don’t use a ton of nested ifs
  - If you’re not sure, hlint/ask us
- No points off for style this time
  - May deduct style points starting HW2

*Read our comments on your HW!*

# LAST TIME: TYPECLASSES

1. Declare class with required functions
2. Implement class for your type
3. Fns can use typeclass constraints

# EXAMPLE: ORD

```haskell
class Eq a => Ord a where
  (<)  :: a -> a -> Bool
  -- ... more stuff ...

data Nat = Zero | Succ Nat

instance Ord Nat where
  Zero   < Zero   = False
  Succ _ < Zero   = False
  Zero   < Succ _ = True
  Succ n < Succ m = n < m
  -- ... more stuff

sort :: Ord a => [a] -> [a]
sort list = -- ... < ...
```

# A PEEK UNDER THE HOOD

# ENCODE TYPECLASS INFO

- Given *class* declaration…

```
class Ord a where
  (<)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
```

- Compiler makes dictionary *type*…

```
data OrdDict a = MkOrdDict { (<) :: a -> a -> Bool
                           , (<=) :: a -> a -> Bool }
```

# ENCODE INSTANCE INFO

- Given *instance* declaration for type…

```
instance Ord Nat where
  n < n'  = natLessThan n n'
  n <= n' = natLeqThan n n'
```

- Compiler makes *dictionary*…

```
NatOrdDict :: OrdDict Nat
NatOrdDict = MkOrdDict { (<)  = natLessThan
                       , (<=) = natLeqThan }
```

# THREAD THE DICTIONARY

- Say we have function to find the bigger tuple element

```
max :: Ord a => a -> a -> a
max x y
    | x < y      = y
    | otherwise = x
```

- Compiler replaces constraint with dictionary
- Gets method instances from the dictionary

```
max' :: OrdDict a -> a -> a -> a
max' dict x y
    | ((<) dict) x y      = y
    | otherwise           = x
```

# ADJUST FUNCTION CALLS

- Say we call the `max` function

```
bigger = max Zero (Succ Zero)
```

- Compiler adds in the dictionary for `Nat`

```
bigger = max' NatOrdDict Zero (Succ Zero)
```

- Voilà! No more typeclasses, just plain functions

# MAX ON OTHER TYPES

- Say we call the `max` function on `Char`

```
bigger = max 'a' 'b'
```

- Compiler adds in the dictionary for `Char`

```
bigger = max' CharOrdDict 'a' 'b'
```

# TODAY: FUNCTOR

# GOING UP A LEVEL

- So far: typeclass instances for types
- Many things in Haskell are not types:
  - `Maybe`
  - `[]`
- They need a type argument to become a type:
  - `Maybe Int`
  - `[Int]`

> *Define typeclasses for these things!*

# MAPPABLE

- We can map over many things: `Maybe`, lists, trees, …
- Factor this into a type class:

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

- Think: a container `f` is “mappable” if it has a `fmap`
  - Note: `f` doesn’t always need to be a “container”

# EXAMPLES OF FUNCTOR

# WARMUP: LISTS

- We already know a mapping function for lists:

```haskell
instance Functor ([]) where
    fmap = map
    -- infix: foo <$> bar === fmap foo bar

    -- What's the type?
    -- fmap :: (a -> b) -> [a] -> [b]
```

# MAYBE

- Would like to map over a Maybe:

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

    -- What's the type?
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
```

# “READER”

- Previous examples: containers
- This example: type of “reader” functions
  - Conversions from type `r` to something else

```haskell
instance Functor ((->) r)
  -- What's the heck is this type??
  -- fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
  -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
  -- Solution is now clear:

  fmap fab fra = \r -> fab (fra r)
```

# FUNCTOR LAWS

# A BROKEN FMAP

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

-- What's the type?
-- fmap :: (a -> b) -> Maybe a -> Maybe b
```

- Type is OK, but it doesn't seem to "map"…

# FOLLOW THE LAWS

- Many Haskell typeclasses come with “laws”
  - Expected equations that should hold
- You should check the laws hold
  - Compiler won’t check these laws for you
  - Breaking laws is almost always a bug

# FUNCTOR LAW: IDENTITY

```
-- Identity function id
-- id :: a -> a

fmap id === id
```

- Mapping a do-nothing function should do nothing

# A BROKEN FMAP

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

    -- What's the type?
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
```

- Breaks law: `fmap id (Just 42) === Nothing`

# FUNCTOR LAW: COMPOSITION

```haskell
-- Suppose: f :: a -> b, g :: b -> c
fmap (g . f) === fmap g . fmap f
```

- Map `f` then map `g` is same as map `g . f`