



CS 115

Functional Programming

Lecture 8:

Type Classes, part 2



Functional Programming



Today

- Deriving type classes
- Instances with contexts
- Class "inheritance"
- Numeric type classes
- Floating-point literals
- Constructor classes: **Functor**
- Instances and the module system





Deriving type classes

- Last time saw that defining **Show** instance for **Color** datatype was boilerplate code:

```
data Color = Red | Green | Blue | Yellow  
instance Show Color where  
    show Red      = "Red"  
    show Green    = "Green"  
    show Blue     = "Blue"  
    show Yellow   = "Yellow"
```

- Tedious to write!





Deriving type classes

- Similarly, defining `Eq` instance for `Color` datatype is also boilerplate code:

```
instance Eq Color where
    Red    == Red    = True
    Green  == Green  = True
    Blue   == Blue   = True
    Yellow == Yellow = True
    _      == _      = False
```

- Really* tedious to write!





Deriving type classes

- For some type classes, Haskell compilers can automatically derive the instance definition
- Works for **Eq**, **Ord**, **Show** and a few other type classes
- Example:

```
data Color = Red | Green | Blue | Yellow  
deriving (Eq, Show)
```

- The **deriving** clause defines **Eq** and **Show** instances in the "standard" way





Deriving type classes

- **Ord** instances order by constructor location
- Example:

```
data Color = Red | Green | Blue | Yellow  
deriving (Eq, Ord, Show)
```

- Now we have **Red < Green < Blue < Yellow**
- Note: we never *have* to automatically derive type classes, but it's often very convenient





newtype deriving

- Since **newtype**-defined datatypes are basically just a wrapper around an existing datatype, with a **newtype** we can derive many more type classes than with **data**-defined types
- In principle, any type class that applies to a type can be derived for a **newtype** wrapper around that type
- This is not a standard Haskell feature
 - it's a GHC compiler extension called **GeneralizedNewtypeDeriving**





Aside: Enabling GHC extensions

- GHC has many extensions to "standard Haskell"
- Many of these are essential when writing real-world code
- However, different programs/modules need different extensions
- Let's see how to enable extensions





Aside: Enabling GHC extensions

- The way to do this is to use a "compiler pragma" in each file that uses extensions
- Such a construct will identify only those extensions used in that specific module
- Compiler uses it to enable named extensions
- Only relevant for code written in files
 - in **ghci**, need to instead type e.g.
:set -XGeneralizedNewtypeDeriving
- Let's say our module (file) needed to use **GeneralizedNewtypeDeriving**; how would we do it?





Aside: Enabling GHC extensions

- We write compiler pragmas as stylized multi-line comments at the top of the file
- Here, we would write this:

```
{ -# LANGUAGE
```

```
    GeneralizedNewtypeDeriving #- }
```

- We can also define multiple extensions:

```
{ -# LANGUAGE
```

```
    Extension1, Extension2, Extension3 #- }
```

- (replace **ExtensionN**s with the correct names)





Instances with contexts

- Sometimes need to add contexts to define instances of some type classes
- Example: `Eq` instance for `[a]`
- What is wrong with this?

```
instance Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```





Instances with contexts

```
instance Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```

- This won't work unless **a** is an **Eq** instance too
- Correct definition:

```
instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```





Type class "inheritance"

- Type classes can "inherit" method signatures from a parent type class
- Classic case in point: **Ord** "inheriting" from **Eq**
- Definition of **Ord**:

```
class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<) , (≤) , (>) , (≥) :: a -> a -> Bool
    max, min :: a -> a -> a
```





Type class "inheritance"

- Once again, this is not "inheritance" in the OOP sense
- Just a way to automatically include/require the method signatures from another class

```
class (Eq a) => Ord a where ...
```

- This means that any **Ord** instance must also be an instance of **Eq** (which defines `==` and `/=`)
- That way, **Ord** methods can use `==` and `/=` operators where appropriate





Type class "inheritance"

- Example: **Num** definition used to require this:

```
class (Eq a, Show a) => Num a where ...
```

- Here, **Num** must define `==`, `/=`, **show**
- Now **Num** doesn't require this anymore
 - **Show** constraint was always bogus
 - Can there be numbers that are not instances of **Eq**?
 - Maybe Church numerals (numbers encoded as functions)





Numeric type classes

- We've seen the **Num** type class
- **Num** represents the most fundamental operations we expect numbers to support
- Many other numeric classes are "subclasses" of **Num**
 - **Fractional**
 - **Real**
 - **Integral**
 - **RealFrac**
 - **Floating**
 - **(Enum)**





Fractional

- **Fractional** type class represents fractional numbers (e.g. rational, real numbers)
- Adds these methods to **Num**:
 - `(/) :: a -> a -> a` -- division
 - `recip :: a -> a` -- reciprocal
 - `fromRational :: Rational -> a`
- **Rational** is the type of rational numbers
 - technically, a type alias for **Ratio Integer**
 - form **Rational** numbers with the `%` operator (in module **Data.Ratio**)
 - `1 % 2 → 1/2`





Fractional

- **Fractional** is also the type of literal floating-point numbers:

```
Prelude> :t 1.23
```

```
1.23 :: Fractional a => a
```

- This means that e.g. **1.23** can represent a **Rational** number (also **Float**, **Double** etc.)

```
Prelude> 1.23 :: Rational
```

```
123 % 100
```

```
Prelude> 1.23 :: Float
```

```
1.23
```





Real

- **Real** adds to **Num** the ability to convert the number to a **Rational**
- `toRational :: a -> Rational`
- **Real** is not used much by itself
- It's used as a superclass of other type classes e.g. **RealFrac**
- Not the best name choice in my opinion





Integral

- **Integral** represents integral numbers and their operations:
 - **quot** :: $a \rightarrow a \rightarrow a$
 - **rem** :: $a \rightarrow a \rightarrow a$
 - **div** :: $a \rightarrow a \rightarrow a$
 - **mod** :: $a \rightarrow a \rightarrow a$
 - **quotRem** :: $a \rightarrow a \rightarrow (a, a)$
 - **divMod** :: $a \rightarrow a \rightarrow (a, a)$
 - **toInteger** :: $a \rightarrow Integer$





Integral

- **quot** is integer division, truncating towards zero
- **rem** is remainder after **quot**
- **div** is integer division, truncating towards negative infinity
- **mod** is remainder after **div**
- **quotRem** is pair of (**quot**, **rem**) results
- **divMod** is pair of (**div**, **mod**) results





RealFrac

- **RealFrac** (another poor name choice?) contains methods to convert between **Integral** and **Fractional** types:
 - `properFraction :: Integral b => a -> (b, a)`
 - `truncate :: Integral b => a -> b`
 - `round :: Integral b => a -> b`
 - `ceiling :: Integral b => a -> b`
 - `floor :: Integral b => a -> b`





Floating

- **Floating** contains methods for standard floating-point operations:
 - `pi`
 - `exp`, `log`, `logBase`
 - `sqrt`
 - `(**)`
 - `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
 - `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- Seems somewhat arbitrary to me





Note

- Numeric type classes are somewhat controversial
- Alternative "Numeric Preludes" exist
- Not a language problem per se
- Hard to know what the best way is to factor numerical functionality among type classes
 - or the appropriate granularity





Enum

- **Enum** type class is not a subclass of **Num**
 - But sufficiently important that it's worth presenting here
- **Enum** methods:
 - `succ :: a -> a`
 - `pred :: a -> a`
 - `toEnum :: Int -> a`
 - `fromEnum :: a -> Int`
 - `enumFrom :: a -> [a]`
 - `enumFromThen :: a -> a -> [a]`
 - `enumFromTo :: a -> a -> [a]`
 - `enumFromThenTo :: a -> a -> a -> [a]`





Enum

- Some unusual classes are instances of **Enum**
 - **Bool**, **Char**

```
Prelude> enumFromTo 'a' 'z'
```

```
"abcdefghijklmnopqrstuvwxyz"
```

```
Prelude> ['a'..'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
```

```
Prelude> enumFromTo False True
```

```
[False, True]
```

```
Prelude> [False .. True] -- need spaces!
```

```
[False, True]
```





Constructor classes

- So far, all instances are concrete types
 - e.g. concrete type **Int** is an instance of **Enum**
- More general kinds (pun!) of instances are possible
- Consider the **Functor** type class
- Represents the notion of "something that can be mapped over"
 - like a generalization of list mapping





Functor

- Definition of **Functor** type class:

```
class Functor f where
```

```
fmap :: (a -> b) -> f a -> f b
```

- Anything seem strange here?
- **fmap** is a generalization of the **map** function
- Instead of **[a]** and **[b]**, we have **f a** and **f b**
- **f** is a *type constructor* (kind: *** -> ***)
- **Functor** is thus a *constructor class*
 - i.e. instances are not types, but type constructors





Functor

- **Functor** instances also have to obey the "functor laws":
 - `fmap id == id` -- `id` = identity function
 - `fmap (f . g) = fmap f . fmap g`
- Haskell cannot enforce this!
- Responsibility of programmer to make sure that instances obey functor laws
 - or code using **Functor** will not behave in a "reasonable" way





Functor

- The name "**Functor**" comes from category theory (CT)
 - *Extremely* abstract branch of mathematics
 - Like an abstract "algebra of functions"
- Other languages use the word "functor" to mean essentially arbitrary things
 - e.g. C++ : "function objects", OCaml: "functions on modules"
- Haskell's functors are essentially the same concept as category theoretic functors
- CT concepts will come up again: monads!





Functor

- **Functor** instance for lists:

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

- **Functor** instance for **Maybe** type constructor:

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





Functor

- **Tree** data type:

```
data Tree a =
```

```
    Leaf
```

```
    | Node a (Tree a) (Tree a)
```

- What is the **Functor** instance?
- (*n.b.* leaving off **Ord** constraint from last lecture, which is probably just as well since it's a deprecated feature)





Functor

```
instance Functor Tree where
    fmap _ Leaf = Leaf
    fmap f (Node x left right) =
        Node (f x) (fmap f left) (fmap f right)
```

- Advantage of **Functor**:
- Can define functions that work generically over any "mappable" data type





Type classes and modules

- Type classes can be explicitly exported/imported like any other names
 - ditto for specific methods in a type class
- To export type class **Foo** along with all its methods, write

```
module XXX(Foo(..), ...)
```

- To export type class **Foo** with only some methods, write

```
module XXX(Foo(method1, method2), ...)
```





Type classes and modules

- To export type class `Foo` without its methods, write
`module XXX(Foo, ...)`
- *Instances* of type classes are "global"
- All instances are exported from a module without having to declare them
 - can't restrict this!
- All instances in a module are imported when importing the module





Type classes and modules

- To import only instances from a module `Foo`, write:

```
import Foo()
```

- This will not import any names from `Foo`, just the instance declarations





Next time

- We've covered "basic Haskell programming"
- Next time, we start part 2 of the class:
- Introduction to monads!

