Defining and working with type classes

Lecture 5 of CSE 3100 Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

Lecture plan

- Type classes recap
- Defining type class instances
- Defining new type classes
- The classes `Functor` and `Applicative`

Recap: Working with Type Classes

Example: Generic `lookup` using `Eq`

```haskell
type Assoc k a = [(k,a)]

lookup :: Eq k =>
  k -> Assoc k a -> Maybe a
lookup k []            = Nothing
lookup k ((l,x):xs)
  | k == l             = Just x
  | otherwise          = lookup k xs
```

Recap: type classes

A type class defines a family of types that share a common interface.

- `Show`: types with `show`
- `Eq`: types with `(==)` and `(/=)`
- `Ord`: types with `(<)`, `(<=)`, ...
- `Num`: types with `(+)`, `(-)`, `(*)`, ...

Looking for common structure

A type class captures a family of types that share some structure.

Question. Why look for this common structure?

Looking for common structure

A type class captures a family of types that share some structure.

Question. Why look for this common structure?

- Avoid boilerplate code (DRY!)
- Enforce abstraction barriers
- Typeclass laws provide a sanity check for correctness
- Deeper understanding of your code

“Type classes are the design patterns of FP”

Definition of the `Eq` class

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

- First line declares a new type class `Eq` with one parameter `a`
- Body of class has one or more function declarations
- Each function optionally has a default implementation

Declaring new instances of `Eq`

```
data TrafficLight =
  Red | Yellow | Green
instance Eq TrafficLight where
  Red    == Red    = True
  Green  == Green  = True
  Yellow == Yellow = True
  _      == _      = False
```

- First line declares `TrafficLight` to be an instance of `Eq`
- Body gives implementation of class functions
- Skipped functions use default implementation

Using our type class instance

```
> Red == Red
True

> Red == Green
False

> let speeds = [(Green , 100),
                (Yellow, 50 ),
                (Red   , 0  )]
> lookup Yellow speeds
Just 50
```

Minimal complete definitions

An instance `Eq X` has to define either
`(==) :: X -> X -> Bool` or
`(/=) :: X -> X -> Bool`, or both.

In Haskell docs: *“Minimal complete definition:* `(==) | (/=)` *”*

Question. What happens if we define neither?

Question. What if `Eq` didn’t have default implementations?

Another example: the `Show` class

```
class Show a where
  show :: a -> String
  showList :: [a] -> ShowS
  showList = ...
  showsPrec :: Int -> a -> ShowS
  showsPrec = ...

instance Show TrafficLight where
  show Red    = "Red light"
  show Yellow = "Yellow light"
  show Green  = "Green light"
```

Automatically deriving classes

Instead of writing instances by hand, Haskell can automatically derive instances of built-in type classes such as `Eq` and `Show`:

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

> Red /= Red
False
> show Green
"Green"
```

The `Arbitrary` type class

QuickCheck uses the `Arbitrary` type class to generate random values and shrink them:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

You can make QuickCheck work with functions on your own data type by implementing an instance for the `Arbitrary` class.

Exercise: Define an instance of `Arbitrary` for the type `TrafficLight`.

Instances of parametrized datatypes

Question. How to define an `Eq` instance for `Maybe`?

First attempt:

```
instance Eq Maybe where ...
```

```
Expecting one more argument to
'Maybe' Expected a type, but
'Maybe' has kind '* -> *'
```

Question. How to define an `Eq` instance for `Maybe`?

Second attempt:

```
instance Eq (Maybe a) where
  Nothing == Nothing = True
  Just x  == Just y  = x == y
  _       == _       = False
```

```
No instance for (Eq a) arising from
a use of ‘==’
```

Question. How to define an `Eq` instance for `Maybe`?

Third attempt:

```
instance Eq a => Eq (Maybe a) where
  Nothing == Nothing = True
  Just x  == Just y  = x == y
  _       == _       = False

> Just 5 == Just 7
False
```

It works now!

Eq and Show instances for Tree

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

Eq and Show instances for Tree

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

instance Eq a => Eq (Tree a) where
  Leaf x   == Leaf y   =  x == y
  Node u v == Node w x =
    u == w && v == x
  _        == _        = False
```

Eq and Show instances for Tree

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

instance Eq a => Eq (Tree a) where
  Leaf x   == Leaf y   =  x == y
  Node u v == Node w x =
    u == w && v == x
  _        == _        = False

instance Show a => Show (Tree a) where
  show (Leaf x)   = "Leaf " ++ show x
  show (Node u v) =
      "Node " ++ showP u ++ " " ++ showP v
    where showP x = "(" ++ show x ++ ")"
```

`Eq` and `Show` instances for `Tree`

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
  deriving (Show, Eq)
```

`deriving` also works for parametrized datatypes!

Live coding: `Arbitrary (Tree a)`

```haskell
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

Goal: Define an instance of `Arbitrary` for the type `Tree a`.

Working with subclasses

Subclass example

Some type classes are a subclass of another class: each instance must also be an instance of the base class.

Example: `Ord` is a subclass of `Eq`:

```
class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  -- ...
```

Why use subclasses?

Two reasons to declare a class as a subclass of another:

- Shorter type signatures: we can write `Ord a => ...` instead of `(Eq a, Ord a) => ...`
- We can use functions from the base class in default implementations: `x <= y = x < y || x == y`

More examples of subclasses

```
class (Num a, Ord a) => Real a
  where ...
class (Real a, Enum a) => Integral a
  where ...
class (Num a) => Fractional a
  where ...
class (Fractional a) => Floating a
  where ...
```

Defining your own classes

A simple type class: `Reversible`

```
class Reversible a where
  rev :: a -> a

instance Reversible [a] where
  rev xs = reverse xs

instance Reversible (Tree a) where
  rev (Leaf x)   = Leaf x
  rev (Node l r) = Node (rev r) (rev
```

Example: Truthy and Falsy values in Haskell

In Python and other languages, values of many types are considered 'truthy' or 'falsy'

```
>>> if 5: print("hey whoah")
hey whoah
```

Let's simulate this behaviour in Haskell with a type class!

```haskell
class Booly a where
  bool :: a -> Bool
```

Instances of `Booly` (1/2)

```haskell
instance Booly Bool where
  bool x = x

instance Booly Int where
  bool x = x /= 0

instance Booly Double where
  bool x = x /= 0.0
```

Instances of `Booly` (2/2)

```haskell
instance Booly (Maybe a) where
  bool Nothing  = False
  bool (Just x) = True

instance Booly [a] where
  bool []    = False
  bool (_:_) = True
```

An `if/then/else` for `Booly` values

```haskell
iffy :: Booly a => a -> b -> b -> b
iffy b x y =
  if bool b then x else y

> iffy [] "yes" "no"
"no"
> iffy False "yes" "no"
"no"
> iffy (Just False) "yes" "no"
"yes"
```

Functors

The `Functor` type class

The function `map` applies a function to *every element in a list.*

`Functor` generalizes this to other data structures for which we have a `map`-like function:

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

We can think of a functor as a container storing elements of type `a`.

Playing with `fmap`

What is the result of evaluating these expressions?

- `fmap (+1) [1,2,3]`
- `fmap (+1) (Just 1)`
- `fmap (+1) Nothing`
- `fmap (+1) (Right 2)`
- `fmap (+1) (Left 3)`
- `fmap (+1) (*2)`

Try it out in GHCi!

Examples of functors

```haskell
instance Functor [] where
  fmap f xs = map f xs

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

instance Functor Tree where
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Node l r) =
    Node (fmap f l) (fmap f r)
```

`Either a` is a functor

```
instance Functor (Either a)
  -- fmap :: (b -> c)
  --      -> Either a b
  --      -> Either a c
  fmap f (Left x)  = Left x
  fmap f (Right y) = Right (f y)
```

`(->) a` is a functor

Reminder: `(->) a b` is the same as `a -> b`.

```haskell
instance Functor ((->) a)
  -- fmap :: (b -> c)
  --         -> (a -> b)
  --         -> (a -> c)
  fmap f g = f . g
```

Question. Can you think of a type constructor that can *not* be made into an instance of `Functor`?

Question. Can you think of a type constructor that can not be made into an instance of `Functor`?

Answer. Here is an example:

```
newtype Endo a = Endo (a -> a)
instance Functor Endo where
  fmap f (Endo g) = Endo ???
```

More generally, if the type parameter `a` occurs *to the left of a function arrow*, the type cannot be made into a `Functor`.

Applicative functors

Reminder: subclasses in Haskell

A subclass is a family that extends the interface of another type class:

```haskell
-- Ord is a subclass of Eq
class Eq a => Ord a where
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  x <= y = (x < y) || (x == y)
```

Each instance of the subclass must already be an instance of the base class.

Applicative functors

`Applicative` is a subclass of `Functor` that adds two new operations `pure` and `(<*>)` (pronounced '*ap*' or '*zap*').

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`:
- For lists:
- For `Tree`:
- For `Either a`:
- For `(->) a`:

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists:
- For `Tree`:
- For `Either a`:
- For `(->) a`:

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`:
- For `Either a`:
- For `(->) a`:

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`:
- For `(->) a`:

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`:
- For `(->) a`:

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`: `pure x = Right x`
- For `(->) a`:

The function `pure`

The function `pure :: a -> f a` takes a value of type `a` and creates a container filled with one or more copies of this value:

- For `Maybe`: `pure x = Just x`
- For lists: `pure x = [x]`
- For `Tree`: `pure x = Leaf x`
- For `Either a`: `pure x = Right x`
- For `(->) a`: `pure x = const x`

The function `(<*>)`

The function

`(<*>) :: f (a -> b) -> f a -> f b`

combines two containers by applying functions in the first to values in the second one.

Example. For the `Maybe` functor, we have

```
Just f  <*> Just x  = Just (f x)
Nothing <*> _       = Nothing
_       <*> Nothing = Nothing
```

Applicative instance for lists

For lists, the function `(<*>)` iterates over all possible combinations of functions and values:

```
instance Applicative [] where
  pure x    = [x]
  fs <*> xs =
    [f x | f <- fs, x <- xs]
```

Example. `[(1+),(2*)] <*> [10,20]` = `[11,21,20,40]`

Remark. This is not the only way to make lists into an applicative functor (see Weblab).

Combining containers

We can combine two applicative containers by chaining `pure` and `(<*>)`:

```
zipA :: Applicative f
     => f a -> f b -> f (a,b)
zipA xs ys =
  pure (,) <*> xs <*> ys
```

Examples.

- `zipA (Just 1) (Just 2)` = `Just (1,2)`
- `zipA (Just 1) Nothing` = `Nothing`

Quiz question

Question. What is the result of
`zipA [1,2] ['a','b']`?

1. `([1,2], ['a','b'])`
2. `[(1,'a'),(2,'b')]`
3. `[(1,'a'),(1,'b'),(2,'a'),(2,'b')]`
4. `[(1,'a'),(2,'a'),(1,'b'),(2,'b')]`

Discussion question

Can every instance of the `Functor` type class be made into an instance of the `Applicative` type class?

Can a given functor be made into an `Applicative` functor in *different ways*?

What's next?

Next lecture: IO and Monads

To do:

- Read the book:
  - Today: 8.5, 12.1–12.2
  - Next lecture: 10.1–10.5, 12.3
- Start on week 3 exercises on Weblab