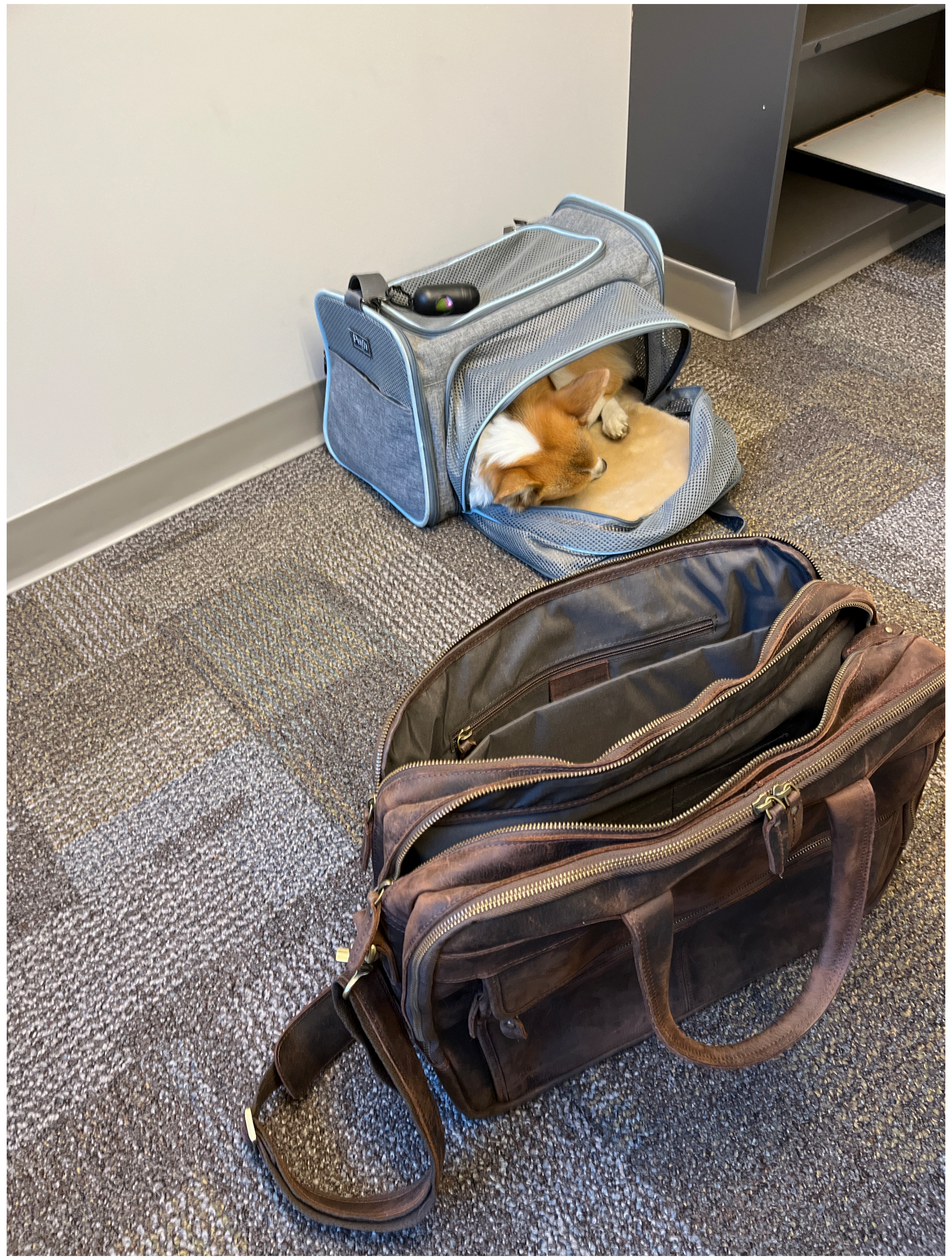
CMPT 383

Lecture 6: Type Classes



Anders Miltner

- Type Class Usage
- Type Class Definition
- Type Class Instantiation

What are Type Classes

- Closest Analogue — Object Interfaces?
- When you can assume a function exists for a specific type
- Haskell’s version of ad-hoc polymorphism / overloading

Motivation: OO

- Imagine you were in a world where your objects didn't have a .toString() method, or a .Equals() method
- Must pass toString() and .Equals() functions around the full system
- Without Type Classes, that's what Haskell is!
    - Many FP languages do not have type classes
    - Require either a different solution, or don't have a solution

Type Class Solution

- Give a name to a bundle of interrelated functions
- Can require that this bundle of functions exists for a instantiated types on (parametrically) polymorphic functions
- If you make this requirement, then you can use those functions within the function body
- (TypeClass a) => [a] -> a -> a

Example Type Class: Show

```
class Show a where
  show :: a -> String
```

```
showList :: (Show a) => [a] -> String
showList [] = “Nil”
showList (h:t) = show h ++ “:” ++ showList t
```

```
>>> :t show
(Show a) => a -> String

>>> :t showList
(Show a) => [a] -> String
```

Example Type Class: Eq

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

```haskell
ifEqShow :: (Eq a) => (Show a) => a -> a -> String
ifEqShow x y = if x == y then show x else “”
```

```
>>> :t (==)
(Eq a) => a -> a -> Bool

>>> :t ifEqShow
(Eq a) => (Show a) => a -> a -> String
```

So how do we instantiate these?

```
instance Eq Bool where
    (==) True True = True
    (==) False False = True
    (==) _ _ = False
```

```
instance Show Bool where
    show True = “True”
    show False = “False”
```

Eq should be reflexive, transitive, and symmetric

- Expected properties:
  - a == a
  - a == b /\ b == c ==> a == c
  - a == b ==> b == a
- Can we guarantee this?

Not in base Haskell!

- If we included such capabilities, type checking would become undecideable
- Exists in dependent Haskell: https://gitlab.haskell.org/ghc/ghc/-/wikis/dependent-haskell
- Exists in other languages like Coq: https://coq.inria.fr/, Agda: https://github.com/agda/agda, Lean: https://leanprover.github.io/, and more

Example Type Class: Ord

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

```
(>=) :: (Ord a) => a -> a -> Bool
(>=) x y = y <= x
```

```
(==) :: (Ord a) => a -> a -> Bool
(==) x y = x <= y && x >= y
```

```
(<) :: (Ord a) => a -> a -> Bool
(<) x y = x <= y && not (x == y)
```

Writing all these helper functions is a bit annoying

1. We can require an instantiated type class

- It doesn’t make sense to have an ordering without equality!

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

- Now, the only instances of Ord *must* also be instances of Eq

2. We can *immediately* provide extra functions

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

  {- Now I want (<), (>), (>=) -}

  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool

  (<) x y = x <= y && not (x == y)
  (>) x y = y < x
  (>=) x y = y <= x
```

Instances with Polymorphic Types

- What do we do about polymorphic types
- Say I want to instantiate List a as part of the “Eq” type class
  - This is tough, because I need to be able to compare the “a”s
- We can require the polymorphic variables to satisfy type class constraints

List Equality Example

```haskell
instance Eq a => Eq [a] where
  (==) :: [a] -> [a] -> Bool

  (==) [] [] = True
  (==) (h1:t1) (h2:t2) = h1 == h2 && t1 == t2
  (==) _ _ = False
```

But there's restrictions!

Must derive from smaller types (guarantees termination)
Must not have multiple ways of deriving instances (guarantees determinism)

With Haskell language extensions however, you can kind of do anything

Deriving Instances

- Some of these type classes can have very straightforward implementations
  - Ord, Eq, etc.
- We can *derive* these instances automatically
- These derived instances have default behavior, if you want different behavior you must handwrite
- You can write your “deriving” for your own type classes, but we won’t go into that in this class

Deriving for Tree

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

```
>>> :t sort
(Ord a) => [a] -> [a] -> [a]

>>> sort [Node(Leaf,1,Leaf), Leaf]
[Leaf,Node(Leaf,1,Leaf)]
```

Deriving Guarantees

- If there's no type variables:
  - derived (==) is syntactic equivalence
  - show will print the data structure in the same way you'd build it
  - <= will be transitive, reflexive, and total
    - Furthermore, a <= b && b <= a iff a == b

Can use to represent math objects

- A monoid is a set of values that has a binary associative operation *, and a designated element ident that is an identity element over *.
- Lets write monoid as a type class!

Monoid Class

```
class Monoid a where
  ident :: a
  (*) :: a -> a -> a
```

Monoid Use

```
combineList :: (Monoid a) => [a] -> a
combineList [] = ident
combineList (h:t) = h * (combineList t)
```

Note that having an ident is important here!
It doesn’t matter whether we combine from left or right (right now we’re doing right-to-left)

Monoid Instances

```
instance Monoid [a] where
  ident = “”
  (*) = (++)
```

```
instance Monoid Nat where
  ident = 0
  (*) = (+)
```

```
instance Monoid Nat where
  ident = 1
  (*) = Prelude.(*)
```