Type Classes

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

- Describe the concept of *polymorphism*.
- Show how to declare instances of a type class.
- Understand the `Eq`, `Ord`, `Show`, and `Read` type classes.

Polymorphism

- We often want to use the *same operation* on things of *different type*.
- How can we do that?
  - Overloading — C++ - like languages
  - Inheritance — Object oriented languages
  - Parameterized Types — Hindley Milner typed languages (Haskell, SML, etc.); C++ (templates), Java (generics)
  - Type Classes — Haskell

Overloading

```
int inc(int i) {
    return i + 1;
}
double inc(double i) {
    return i + 1.0;
}
```

Inheritance

```
public class Shape {
    public int loc_x,loc_y;
}

public class Square extends Shape {
    public int width,height;
}
```

Parametric Polymorphism

```
public class List<E> {
    public E data;
    public List<E> next;
}
```

```
data Cons a = Cons a (Cons a)
            | Nil
```

The Eq Type Class

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

   -- Minimal complete definition:
   --       (==) or (/=)

   x /= y      =  not (x == y)
   x == y      =  not (x /= y)
```

Using Eq

```
data Foo = Foo Int

x = Foo 10
y = Foo 10
```

- If you try to compare these...

```
*Main> x == y

<interactive>:1:3:
    No instance for (Eq Foo)
      arising from a use of ‘==’
    Possible fix: add an instance declaration for (Eq Foo)
    In the expression: x == y
    In an equation for ‘it’: it = x == y
```

Use an Instance

```
instance Eq Foo where
   (==) (Foo i) (Foo j) = i == j
```

- Now if you try to compare these...

```
*Main> let x = Foo 10
*Main> let y = Foo 10
*Main> x == y
True
```

tl;dc

- Too long! Didn't Code!
- Let Haskell do the work.

```
data Foo = Foo Int
  deriving Eq
```

The Ord Typeclass

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

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

The Show Typeclass

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

instance Show Foo where

data Foo = Foo Int
-- one way...
  deriving (Show,Eq)

-- other way...
instance Show Foo where
  show (Foo i) = "Foo " ++ show i
```

The Read Typeclass

```
{-# LANGUAGE ViewPatterns #-}
import Data.List

instance Read Foo where
   read (stripPrefix "Foo " -> Just i) = Foo (read i)
```

- Sample run...

```
*Main> let x = "Foo 10"
*Main> read it :: Foo
Foo 10
```