CSc 372

Comparative Programming Languages

18 : Haskell — Type Classes

Department of Computer Science
University of Arizona

collberg@gmail.com

Type Classes

- Type classes allow us to specify that a particular type has certain operations defined for it.
- We've seen the `Eq` class already, it states that an instance of this class has to have == defined.

The Eq Class

- Consider this definition if Three Valued Logic:

```
data TVL = True' | False' | Unknown
```

- We would like to be able to compare values of TVL for equality, so we declare it as an instance of the Eq class:

```
instance Eq TVL where
    True' == True'    = True
    False' == False'  = True
    _ == _            = False
```

This requires us to define the behavior of == for all values of TVL.

The Eq Class...

- The `Eq` class is defined in the standard Prelude:

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

The Eq Class. . .

- Now that we know what it means for two values of type TVL to be equal, we can search in a list using the elem function:

```
> Unknown `elem` [False',True',Unknown,False']
False
> True' `elem` [False',True',Unknown,False']
True
```

- In ghci the :info command will show you definitions of a name:

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

The Show Class

- We often want to control the way the values of a particular type is displayed. To do this, create an instance of the `Show` class:

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

- Here we define how to print the values from the `TVL` class:

```
instance Show TVL where
   show True' = "T"
   show False' = "F"
   show Unknown = "?"
```

The Show Class...

- We only have to define the show function, the others have default implementations defined in terms of show.
- Now a list, for example, of TVL values will print the way we want to:

```
> [False',True',Unknown,False']
[F,T,?,F]
```

The Enum Class

- Class Enum defines operations on sequentially ordered types:

```haskell
class  Enum a  where
    succ, 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]
```

The Enum Class...

- We just have to define the `fromEnum` and `toEnum` operations:

```
instance Enum TVL where
    fromEnum True'    = 0
    fromEnum False'   = 1
    fromEnum Unknown = 2
    toEnum 0 = True'
    toEnum 1 = False'
    toEnum 2 = Unknown
```

- We now have access to all the functions in the class:

```
> [True' .. Unknown]
[T,F,?]
> succ True'
```

The Ord Class

- The Ord class is used for totally ordered datatypes:

```
data Ordering = LT | EQ | GT

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

We only need to define <=, the rest are added automatically.

The Ord Class...

- In our case, since we've already defined TVL as being an instance of Enum, declaring it an instance of Ord is easy, just define <= in terms of fromEnum:

```
instance Ord TVL where
  c <= c' = fromEnum c <= fromEnum c'
```

- Now we can sort, for example:

```
> sort [False',True',Unknown,False']
[T,F,F,?]
```

The Read Class...

- The Read class is approximately the opposite of the Show class: it converts a string to an element of a type.

```
instance Read TVL where
    readsPrec _ "T" = [(True',"")]
    readsPrec _ "F" = [(False',"")]
    readsPrec _ "?" = [(Unknown,"")]
```

- Examples:

```
> read "T" :: TVL
T
> read "?" :: TVL
?
```

Defining a Type Class

- There's nothing stopping us from creating our own type classes. Here's the `Shape` class that requires two functions `area` and `circumference` to be defined:

```
class Shape a where
   area :: a -> Float
   circumference :: a -> Float
```

- Here we define our own polygon data type:

```
data Poly = Triangle Float Float Float
          | Rectangle Float Float
           deriving Show
```

Defining a Type Class...

- And now we can make `Poly` an instance of the `Shape` class:

```
instance Shape Poly where
   area (Triangle a b c) =
         sqrt(p*(p-a)*(p-b)*(p-c))
            where p = (a+b+c)/2
   area (Rectangle a b)  = a*b
   circumference (Triangle a b c) = a+b+c
   circumference (Rectangle a b) = a+b
```

Exercise I

- Consider this definition of a tree:

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

- Here are some examples of trees:

```
t1 = Leaf 1
t2 = Leaf 2
t3 = Branch (Leaf 1) (Leaf 2)
t4 = Branch (Leaf 1) (Leaf 3)
t5 = Branch (Leaf 1)
       (Branch
          (Branch (Leaf 2) (Leaf 3))
          (Leaf 4))
```

Exercise I...

- Make `Tree` an instance of the `Eq` class so that we can compare trees for equality.
- Examples:

```
> t1==t1
True
> t1==t2
False
> t3==t4
False
> t5==t5
True
```

Exercise II

- Consider this data type for complex numbers:

```
data Complex = Complex Int Int deriving Show
```

- Instead of this ugly printing

```
> (Complex 4 5)
Complex 4 5
```

  we want complex numbers to be printed in standard notation:

```
> (Complex 4 5)
4+5i
```

- To accomplish this, make `Complex` an instance of the `Show` class,

```
instance Show Complex where
    show (Complex re im) = ...
```

Exercise II...

- We would like to be able to convert a string representation of a complex number into the `Complex` datatype:

```
>(read "4 + 5i")::Complex
4+5i
```

- For simplicity, we assume the + is always surrounded by whitespace.
- To allow this conversion, override the `readsPrec` function in the `Read` typeclass:

```
instance Read Complex where
    readsPrec _ s = [((Complex re im),"")]
              where
                  re = ...
                  im = ...
```

Exercise II...

- With `Complex` both an instance of the `Read` and the `Show` class, write a `main` program that prompts the user for a complex number, and then prints it out:

```
Enter a complex number: 4 + 5i
You entered: 4+5i
```

Acknowledgments

- http://www.haskell.org/tutorial/classes.html
- http://scienceblogs.com/goodmath/2007/01/16/haskell-the-basics-of-type-cla-1/