CS 457/557 Functional Programming

Lecture 12

Qualified Types and Type Classes

The Haskell Class System

- Think of a **qualified type** as a type with extra requirements.
- Types which meet those requirements have "extra" functionality.
- A **class** definition defines the function signatures for the "extra" functionality.
- An **instance** declaration defines the "extra" functionality for a particular type.

Why type classes?

- Consider the elem function for searching a list:

```
elem x [] = False
elem x (y:ys) | x == y = True
              | otherwise = x `elem` ys
```

- What should its type be? Something like

```
elem :: a -> [a] -> Bool
```

- But this is too general, since elem only makes sense on lists whose members can be compared for equality.
  - What things can't be? Functions, new data types for which == hasn't been written, ...
  - Want the type of elem to reflect this restriction.
- Solution: define **type classes** and use them to describe restrictions.

Prototypical type class: Eq

- Class definitions specify which methods (functions) must be defined on the type for it to be a member of the class.

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

- Qualified types include class constraints on type variables

```
elem :: Eq a => a -> [a] -> Bool
```

  and types like this will be automatically infered for function definitions that use `==`.

- Type qualifiers on functions propagate with use:

```
elemAll :: Eq a => a -> [[a]] -> Bool
elemAll x yss = all (elem x) yss
find :: (Eq a,Show a) => a -> [a] -> String
find x xs | x `elem` x = “Found “ ++ (show x)
          | otherwise = “Can't find “ ++ (show x)
```

Instance Declarations

– Instance declarations allow us to add new types to a class.

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

– Now we can use `==` on `Color` values.

```
(Blue == Red) == False

hasRed :: [Color] -> Bool
hasRed cs = Red `elem` cs
```

Fancier instance definitions

- What about parameterized types? Can they be instances? Yes, if properly qualified!

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

- Note that use of `==` in body is at type `a`, not `Maybe a`.
- So instance definition must be qualified too: we can only compare two `(Maybe a)` values if we can compare two `a` values.
- The same idea works for lists and other "container types."

Fancier Class declarations

- A class can specify multiple methods, and give **default** definitions for these methods.

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

- Now we can use `/=` on any values of a type belonging to class `Eq`. Instance declarations don't have to define `/=` assuming that the default implementation is ok.

```
allDifferent :: Eq a => a -> a -> a -> Bool
allDifferent x y z = x /= y && x /= z && y /= z
```

- In fact, `Eq` also contains a default method for ==

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

- Must define at least one of == or /= to avoid infinite loops!

Inheritance in Class Definitions

- We may want to use a class method when defining default operations for another class:

```
class Eq a => Ord a where
  (<),(<=),(>),(>=) :: a -> a -> Bool
  x <= y = (x < y) || (x == y)
  x >= y = (x > y) || (x == y)
```

- These definitions for `<=` and `>=` only make sense if `==` is defined on type `a`. This is what the `Eq a =>` qualifier means. We say `Ord a` **inherits** from `Eq a`.
- Think of classes as collections of types: since any type belonging to `Ord` must belong to `Eq` , we say that `Ord` is a **subclass** of `Eq` (or `Eq` is a **superclass** of `Ord`).
- Somewhat similar to object-oriented ideas, but not quite the same!

(Parts of) Some Prelude Classes

- Eq, Ord.
- Enumerable types:

```
class Enum a where
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFromTo :: a -> a -> [a]
```

  » `[a .. b]` is just syntactic sugar for `(enumFromTo a b)`

- Viewable types:

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

- Parseable types:

```
class Read a where
   read :: String -> a
```

  » Type inferencer must rely on context to determine result type `a`.

Predefined and Derived Instances

- The Prelude already has appropriate instance definitions for the types and classes defined there.
  - » Almost all types except IO, (->) belong to Eq, Ord, Show, Read
- To put newly defined types into standard classes requires an instance declaration. Writing these is typically straightforward, but tedious.
- For certain Prelude classes, Haskell allows us to **derive** the instance definition for a new type automatically.
  - `Eq, Ord, Enum, Show, Read, Bounded, Ix`
- Example Uses of deriving classes

```
data Color = Red | Green | Blue
  deriving Eq
data Exp = Int Int | Plus Exp Exp | Minus Exp Exp
  deriving (Eq,Show)
```

Some numeric classes (slightly wrong)

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger    :: Integer -> a
class Num a => Fractional a where
    (/) :: a -> a -> a
    recip :: a -> a
    fromRational   :: Rational -> a
class (Num a, Ord a) => Integral a where
    div, mod :: a -> a -> a
    toInteger :: a -> Integer
```

Numeric Types and Literals

- Classes form a hierarchy. Omitting some intermediate classes, we have:

`Int` (fixed-precision integers) belongs to `Integral`.

`Integer` (arbitrary-precision integers) belongs to `Integral`.

`Integral a => Ratio a` belongs to `Fractional`

```
Rational = Ratio Integer
```

`Float, Double` belong to `Fractional`

- Literals have very general types for maximum flexibility
  - `3` is syntactic sugar for `(fromInteger 3)`
  - `3.14` is syntactic sugar for `(fromRational (314/100))`

Defining your own Classes

- You can define your own classes and add new or existing types to them. E.g. , we had:

```
containsS :: Shape -> Point -> Bool
containsR :: Region -> Point -> Bool
```

- Can abstract:

```
class PC t where  -- point containment
  contains :: t -> Point -> Bool
instance PC Shape where
  contains = containsS
instance PC Region where
  contains = containsR
```

- Now can write:

```
Rectangle 2 3 `contains` p  -- uses containsS
(r1 `union` r2) `contains` p -- uses containsR
```

Implicit invariants of Type Classes

- When we define a type class (especially those with multiple methods) we often want some things to be true about the way the methods interact.
- In Haskell we can't make these invariants explicit

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

- Desirable invariants:

```
a == b  <=>  b == a
a == a
a == b && b == c => a == c
```

- But this is perfectly legal:

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