Type classes

CIS 194 Week 5
25 September 2014

Before we get down to the business of the day, we need a little header information to get us going:

```
{-# LANGUAGE FlexibleInstances #-}
```

That's a so-called language pragma. GHC includes many features which are not part of the standardized Haskell language. To enable these features, we use language pragmas. There are *lots* of these language pragmas available — we'll see only a few over the course of the semester.

```
import Data.Char  ( isUpper, toUpper )
import Data.Maybe ( mapMaybe )
import Text.Read  ( readMaybe )
```

Putting the *fun* in function

We have seen now several cases of using a functional programming style. Here, we will look at several functions using a very functional style to help you get acclimated to this mode of programming.

Functional combinators

First, we need a few more combinators to get us going:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

The `(.)` operator, part of the Haskell Prelude, is just function composition. Say we want to take every element of a list and add 1 and then multiply by 4. Here is a good way to do it:

```
add1Mul4 :: [Int] -> [Int]
add1Mul4 x = map ((*4) . (+1)) x
```

While we're at it, we should also show the `($)` operator, which has a trivial-looking definition:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Why have such a thing? Because `($)` is parsed as an operator, and this is useful for avoiding parentheses. For example, if we wish to negate the number of even numbers in a list, we could say

```
negateNumEvens1 :: [Int] -> Int
negateNumEvens1 x = negate (length (filter even x))
```

or

```
negateNumEvens2 :: [Int] -> Int
negateNumEvens2 x = negate $ length $ filter even x
```

No more parentheses!

Point-free programming

Both examples above just apply a sequence of operations to a parameter. The parameter itself – `x` – isn't important at all. So, let's leave it out:

```
add1Mul4' :: [Int] -> [Int]
add1Mul4' = map ((*4) . (+1))

negateNumEvens3 :: [Int] -> Int
negateNumEvens3 = negate . length . filter even
```

Point-free notation is where the `(.)` operator really comes in: it's the way of combining functions when you want to perform one after another.

Lambda

It is sometimes necessary to create an anonymous function, or *lambda expression*. This is best explained by example. Say we want to duplicate every string in a list:

```
duplicate1 :: [String] -> [String]
duplicate1 = map dup
  where dup x = x ++ x
```

It's a tiny bit silly to name `dup`. Instead, we can make an anonymous function:

```
duplicate2 :: [String] -> [String]
duplicate2 = map (\x -> x ++ x)
```

The backslash binds the variables after it in the expression that follows the `->`. For anything but the shortest examples, it's better to use a named helper function, though.

Functional examples

Say we want to count the total number of uppercase letters in a list of strings. Here is a good way to do it:

```
numUppercase :: [String] -> Int
numUppercase = length . filter isUpper . concat
```

It may look like the implementation ends up building and then discarding large lists. But, Haskell style includes functions like these so much, GHC knows how to optimize these really well. The internal lists never get built!

Now, say we want to make all characters in a list of strings to be uppercase. Here we go:

```
listToUpper :: [String] -> [String]
listToUpper = map (map toUpper)
```

Say we want to find all the digits in a string and add them together. Here is one way:

```
sumStringDigits :: String -> Int
sumStringDigits = sum . mapMaybe read_digit
  where read_digit :: Char -> Maybe Int
        read_digit = readMaybe . listify

        listify :: a -> [a]
        listify x = [x]   -- also written (:[])  [[Think about why...]]
```

Further reading

Previous versions of CIS194 have focused more on the functional style of programming. In light of the fact that other Penn courses (specifically, CIS 120) do this quite well, I am choosing this lighter presentation. But, the old lecture notes are great. Find them **here**.

Type classes

Though you never knew it, there are two different forms of polymorphism. The polymorphism we have seen so far is parametric polymorphism, which we can also call *universal* polymorphism (though that isn't a standard term). A function like `length :: [a] -> Int` works for *any* type `a`. But, sometimes we don't want to be universal. Sometimes, we want a function to work for several types, but not every type.

A great example of this is `(+)`. We want to be able to add `Int`s and `Integer`s and `Double`s, but not `Maybe Char`s. This sort of polymorphism – where multiple types are allowed, but not every type – is called *ad-hoc* polymorphism. Haskell uses type classes to implement ad-hoc polymorphism.

A Haskell *type class* defines a set of operations. We can then choose several types that support those operations via *class instances*. (*Note:* These are **not** the same as object-oriented classes and instances!) Intuitively, type classes correspond to *sets of types* which have certain operations defined for them.

As an example, let's look in detail at the `Eq` type class.

```
class Eq a where
```

```
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

We can read this as follows: `Eq` is declared to be a type class with a single (type) parameter, `a`. Any type `a` which wants to be an *instance* of `Eq` must define two functions, `(==)` and `(/=)`, with the indicated type signatures. For example, to make `Int` an instance of `Eq` we would have to define `(==) :: Int -> Int -> Bool` and `(/=) :: Int -> Int -> Bool`. (Of course, there's no need, since the standard Prelude already defines an `Int` instance of `Eq` for us.)

Let's look at the type of `(==)` again:

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

The `Eq a` that comes before the `=>` is a *type class constraint*. We can read this as saying that for any type `a`, *as long as `a` is an instance of `Eq`*, `(==)` can take two values of type `a` and return a `Bool`. It is a type error to call the function `(==)` on some type which is not an instance of `Eq`. If a normal polymorphic type is a promise that the function will work for whatever type the caller chooses, a type class polymorphic function is a *restricted* promise that the function will work for any type the caller chooses, *as long as* the chosen type is an instance of the required type class(es).

The important thing to note is that when `(==)` (or any type class method) is used, the compiler uses type inference to figure out *which implementation of `(==)` should be chosen*, based on the inferred types of its arguments. In other words, it is something like using an overloaded method in a language like Java.

To get a better handle on how this works in practice, let's make our own type and declare an instance of `Eq` for it.

```
data Foo = F Int | G Char

instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False

  foo1 /= foo2 = not (foo1 == foo2)
```

It's a bit annoying that we have to define both `(==)` and `(/=)`. In fact, type classes can give *default implementations* of methods in terms of other methods, which should be used whenever an instance does not override the default definition with its own. So we could imagine declaring `Eq` like this:

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

Now anyone declaring an instance of `Eq` only has to specify an implementation of `(==)`, and they will get `(/=)` for free. But if for some reason they want to override the default implementation of `(/=)` with their own,

they can do that as well.

In fact, the `Eq` class is actually declared like this:

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

This means that when we make an instance of `Eq`, we can define *either* `(==)` or `(/=)`, whichever is more convenient; the other one will be automatically defined in terms of the one we specify. (However, we have to be careful: if we don't specify either one, we get infinite recursion!)

As it turns out, `Eq` (along with a few other standard type classes) is special: GHC is able to automatically generate instances of `Eq` for us. Like so:

```
data Foo' = F' Int | G' Char
  deriving (Eq, Ord, Show)
```

This tells GHC to automatically derive instances of the `Eq`, `Ord`, and `Show` type classes for our data type `Foo`. This `deriving` mechanism is baked into Haskell – you can't make your own class and tell GHC how to derive instances. The full list of derivable classes is `Eq`, `Ord`, `Enum`, `Ix`, `Bounded`, `Show`, and `Read`. Not each of this is applicable to each datatype, though. GHC does provide extensions that allow other classes to be derived; see the GHC manual for details.

Type classes and Java interfaces

Type classes are quite similar to Java interfaces. Both define a set of types/classes which implement a specified list of operations. However, there are a couple of important ways in which type classes are more general than Java interfaces:

1. When a Java class is defined, any interfaces it implements must be declared. Type class instances, on the other hand, are declared separately from the declaration of the corresponding types, and can even be put in a separate module.

2. The types that can be specified for type class methods are more general and flexible than the signatures that can be given for Java interface methods, especially when *multi-parameter type classes* enter the picture. For example, consider a hypothetical type class

   ```
   class Blerg a b where
     blerg :: a -> b -> Bool
   ```

   Using `blerg` amounts to doing *multiple dispatch*: which implementation of `blerg` the compiler should choose depends on *both* the types `a` and `b`. There is no easy way to do this in Java.

   Haskell type classes can also easily handle binary (or ternary, or …) methods, as in

   ```
   class Num a where
   ```

```
(+) :: a -> a -> a
...
```

There is no nice way to do this in Java: for one thing, one of the two arguments would have to be the "privileged" one which is actually getting the `(+)` method invoked on it, and this asymmetry is awkward. Furthermore, because of Java's subtyping, getting two arguments of a certain interface type does *not* guarantee that they are actually the same type, which makes implementing binary operators such as `(+)` awkward (usually requiring some runtime type checks).

Standard type classes

Here are some other standard type classes you should know about:

- **Ord** is for types whose elements can be *totally ordered*, that is, where any two elements can be compared to see which is less than the other. It provides comparison operations like `(<)` and `(<=)`, and also the `compare` function.
- **Num** is for "numeric" types, which support things like addition, subtraction, and multipication. One very important thing to note is that integer literals are actually type class polymorphic:

  ```
  Prelude> :t 5
  5 :: Num a => a
  ```

  This means that literals like `5` can be used as `Int`s, `Integer`s, `Double`s, or any other type which is an instance of `Num` (`Rational`, `Complex Double`, or even a type you define...)
- **Show** defines the method `show`, which is used to convert values into `String`s.
- **Read** is the dual of `Show`.
- **Integral** represents whole number types such as `Int` and `Integer`.

A type class example

As an example of making our own type class, consider the following:

```
class Listable a where
  toList :: a -> [Int]
```

We can think of `Listable` as the class of things which can be converted to a list of `Int`s. Look at the type of `toList`:

```
toList :: Listable a => a -> [Int]
```

Let's make some instances for `Listable`. First, an `Int` can be converted to an `[Int]` just by creating a singleton list, and `Bool` can be converted similarly, say, by translating `True` to `1` and `False` to `0`:

```
instance Listable Int where
```

```
  -- toList :: Int -> [Int]
  toList x = [x]

instance Listable Bool where
  toList True  = [1]
  toList False = [0]
```

We don't need to do any work to convert a list of `Int` to a list of `Int`:

```
instance Listable [Int] where
  toList = id
```

Finally, here's a binary tree type which we can convert to a list by flattening:

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

instance Listable (Tree Int) where
  toList Empty        = []
  toList (Node x l r) = toList l ++ [x] ++ toList r
```

If we implement other functions in terms of `toList`, they also get a `Listable` constraint. For example:

```
-- to compute sumL, first convert to a list of Ints, then sum
sumL x = sum (toList x)
```

`ghci` informs us that type type of `sumL` is

```
sumL :: Listable a => a -> Int
```

which makes sense: `sumL` will work only for types which are instances of `Listable`, since it uses `toList`. What about this one?

```
foo x y = sum (toList x) == sum (toList y) || x < y
```

`ghci` informs us that the type of `foo` is

```
foo :: (Listable a, Ord a) => a -> a -> Bool
```

That is, `foo` works over types which are instances of *both* `Listable` and `Ord`, since it uses both `toList` and comparison on the arguments.

As a final, and more complex, example, consider this instance:

```
instance (Listable a, Listable b) => Listable (a,b) where
  toList (x,y) = toList x ++ toList y
```

Notice how we can put type class constraints on an instance as well as on a function type. This says that a pair type `(a,b)` is an instance of `Listable` as long as `a` and `b` both are. Then we get to use `toList` on values of

types `a` and `b` in our definition of `toList` for a pair. Note that this definition is *not* recursive! The version of `toList` that we are defining is calling *other* versions of `toList`, not itself.

`Generated 2014-12-04 13:37:42.737682`