CIS 194: **Home** | **Lectures & Assignments** | **Policies** | **Resources** | **Final Project** | **Older version**

Type Classes

CIS 194 Week 4
11 February 2015

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 )
```

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. The specific instance that is chosen is always known statically. This is very important since, as we saw with parametric polymorphism, types are erased at runtime.

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 these is applicable to any 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 may seem 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. Type classes often come with a set of mathematical laws that *should* be followed by all instances. Examples of this include associativity and commutativity of addition in the Num type class. We will see more of this shortly when we examine type classes stemming ideas in Category Theory.

2. 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 (these are called orphan instances).

3. 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. Furthermore, Haskell has the concept of *Functional Dependencies*. Say we want a type class for extracting elements from containers:

```
class Extract a b | a -> b where
  extract :: a -> b
```

Here, we have introduced a Functional Dependency stating that the type a uniquely determines b. We can now define an instance of this class that extracts the first element from a tuple:

```
instance Extract (a, b) a where
  extract (x, y) = x
```

However, because of the functional dependency, we cannot create the instance:

```
instance Extract (a, b) b where...
```

because the type (a,b) uniquely determines a.

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 Ints, Integers, Doubles, 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 Strings. This is what GHCi uses to display values.
- **Read** is the dual of Show.
- **Integral** represents whole number types such as Int and Integer.

Monoids

Consider some type m and an operation (<>) :: m -> m -> m. The type and operation form a *monoid* when

1. there exists a particular element mempty :: m such that x <> mempty == x and mempty <> x == x; and

2. the operation (<>) is associative. That is, (a <> b) <> c == a <> (b <> c).

Monoids actually come from a field of abstract mathematic called Category Theory, but they are ubiquitous in programming. This is true in all languages, but we make their presence in Haskell much more explicit, through the use of a type class:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m

  mconcat :: [m] -> m     -- this can be omitted from Monoid instances
  mconcat = foldr mappend mempty

(<>) :: Monoid m => m -> m -> m    -- infix operator for convenience
(<>) = mappend
```

There are a great many Monoid instances available. Perhaps the easiest one is for lists:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Monoids are useful whenever an operation has to combine results, but there may be, in general, multiple different types of results and multiple different ways of combining the results. For example, say we are interested in the positive integers less than 100 that are divisible by 5 or 7, but not both. We can write a function that accumulates these in a monoid:

```
-- this is not the most efficient!
intInts :: Monoid m => (Integer -> m) -> m   -- interesting ints!
intInts mk_m = go [1..100]   -- [1..100] is the list of numbers from 1 to 100
  where go [] = mempty
        go (n:ns)
          | let div_by_5 = n `mod` 5 == 0
                div_by_7 = n `mod` 7 == 0
          , (div_by_5 || div_by_7) && (not (div_by_5 && div_by_7))
          = mk_m n <> go ns
          | otherwise
          = go ns
```

The mk_m parameter converts an Integer into whatever monoid the caller wants. The recursive go function then combines all the results according to the monoid operation.

Here, we can get these results as a list:

```
intIntsList :: [Integer]
intIntsList = intInts (:[])
```

The (:[]) is just a section, applying the cons operator : to the empty list. It is the same as (₩x -> [x]). (:[]) is sometimes pronounced “robot”.

Suppose we want to combine the numbers as a product, instead of as a list. You might be tempted to say

```
intIntsProduct :: Integer
intIntsProduct = intInts id
```

(Recall that id :: a -> a.) That doesn't work, because there is no Monoid instance for Integer, and for good reason. There are *several* ways one might want to combine numbers monoidically. Instead of choosing one of these ways to be the Monoid instance, Haskell defines no Monoid instance. Instead, the Data.Monoid module exports two "wrappers" for numbers, with appropriate Monoid instances. Here is one:

```
data Product a = Product a
instance Num a => Monoid (Product a) where
  mempty                          = Product 1
  mappend (Product x) (Product y) = Product (x * y)

getProduct :: Product a -> a
getProduct (Product x) = x
```

Now, we can take the product of the interesting integers:

```
intIntsProduct :: Integer
intIntsProduct = getProduct $ intInts Product
```

We still do have to explicit wrap (with Product) and unwrap (with getProduct).

The idiom we see with Product is quite common when working with type classes. Because you can define only one instance of a class per type, we use this trick to effectively differentiate among instances.

Check out the documentation for the Data.Monoid module to see more of these wrappers.

Functor

There is one last type class you should learn about, Functor:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

It may be helpful to see some instances before we pick the definition apart:

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Note that the type argument to Functor is not quite a type: it's a *type constructor*. (Or, equivalently, f has kind * -> *.) That's why we make instances for [] (the list type) and Maybe, not, say, for [Int] or Maybe Bool. fmap takes a normal function and "lifts" it into the Functor type. For lists, this is just the map operation; for Maybe, the function affects the Just constructor but leaves Nothing well enough alone.

You can think of functors as being containers, where it is possible to twiddle the contained bits. The fmap operation allows you access to the contained bits, *without* affecting the container. One of the key properties of fmap is that fmap id == id. That is, if you don't change the elements of the container (id does nothing, recall), then you haven't changed anything. For example, a binary tree might have a Functor instance. You can fmap to change the data in the tree, but the tree shape itself would stay the same.

(Note that you wouldn't want to do this with a binary *search* tree, because fmaping might change the ordering relationship among elements, and your tree would no longer satisfy the binary search tree invariants.)

Instead of writing out `fmap`, many people prefer to use the operator version `(<$>)`. When dealing with containers that you know nothing about, a `Functor` instance is often all you need to make progress!

Generated 2015-02-11 09:25:50.93789