More polymorphism and type classes

CIS 194 Week 5
11 February 2013

Haskell's particular brand of polymorphism is known as *parametric* polymorphism. Essentially, this means that polymorphic functions must work *uniformly* for any input type. This turns out to have some interesting implications for both programmers and users of polymorphic functions.

Parametricity

Consider the type

```
a -> a -> a
```

Remember that `a` is a *type variable* which can stand for any type. What sorts of functions have this type?

What about this:

```
f :: a -> a -> a
f x y = x && y
```

It turns out that this doesn't work. The syntax is valid, at least, but it does not type check. In particular we get this error message:

```
2012-02-09.lhs:37:16:
    Couldn't match type `a' with `Bool'
      `a' is a rigid type variable bound by
          the type signature for f :: a -> a -> a at 2012-02-09.lhs:37:3
    In the second argument of `(&&)', namely `y'
    In the expression: x && y
    In an equation for `f': f x y = x && y
```

The reason this doesn't work is that the *caller* of a polymorphic function gets to choose the type. Here we, the *implementors*, have tried to choose a specific type (namely, `Bool`), but we may be given `String`, or `Int`, or even some type defined by someone using `f`, which we can't possibly know about in advance. In other words, you can read the type

```
a -> a -> a
```

as a *promise* that a function with this type will work no matter what type the caller chooses.

Another implementation we could imagine is something like

```
f a1 a2 = case (typeOf a1) of
            Int  -> a1 + a2
            Bool -> a1 && a2
            _    -> a1
```

where `f` behaves in some specific ways for certain types. After all, we can certainly implement this in Java:

```
class AdHoc {

    public static Object f(Object a1, Object a2) {
        if (a1 instanceof Integer && a2 instanceof Integer) {
            return (Integer)a1 + (Integer)a2;
        } else if (a1 instanceof Boolean && a2 instanceof Boolean) {
            return (Boolean)a1 && (Boolean)a2;
        } else {
            return a1;
        }
```

```
    }

    public static void main (String[] args) {
        System.out.println(f(1,3));
        System.out.println(f(true, false));
        System.out.println(f("hello", "there"));
    }

}

[byorgey@LVN513-9:~/tmp]$ javac Adhoc.java && java AdHoc
4
false
hello
```

But it turns out there is no way to write this in Haskell. Haskell does not have anything like Java's `instanceof` operator: it is not possible to ask what type something is and decide what to do based on the answer. One reason for this is that Haskell types are *erased* by the compiler after being checked: at runtime, there is no type information around to query! However, as we will see, there are other good reasons too.

This style of polymorphism is known as *parametric polymorphism*. We say that a function like `f :: a -> a -> a` is *parametric* in the type `a`. Here "parametric" is just a fancy term for "works uniformly for any type chosen by the caller". In Java, this style of polymorphism is provided by *generics* (which, you guessed it, were inspired by Haskell: one of the original designers of Haskell, **Philip Wadler**, was later one of the key players in the development of Java generics).

So, what functions actually *could* have this type? Actually, there are only two!

```
f1 :: a -> a -> a
f1 x y = x

f2 :: a -> a -> a
f2 x y = y
```

So it turns out that the type `a -> a -> a` really tells us quite a lot.

Let's play the parametricity game! Consider each of the following polymorphic types. For each type, determine what behavior(s) a function of that type could possibly have.

- `a -> a`
- `a -> b`
- `a -> b -> a`
- `[a] -> [a]`
- `(b -> c) -> (a -> b) -> (a -> c)`
- `(a -> a) -> a -> a`

Two views on parametricity

As an *implementor* of polymorphic functions, especially if you are used to a language with a construct like Java's `instanceof`, you might find these restrictions annoying. "What do you mean, I'm not allowed to do X?"

However, there is a dual point of view. As a *user* of polymorphic functions, parametricity corresponds not to *restrictions* but to *guarantees*. In general, it is much easier to use and reason about tools when those tools give you strong guarantees as to how they will behave. Parametricity is part of the reason that just looking at the type of Haskell function can tell you so much about the function.

OK, fine, but sometimes it really is useful to be able to decide what to do based on types! For example, what about addition? We've already seen that addition is polymorphic (it works on `Int`, `Integer`, and `Double`, for example) but clearly it has to know what type of numbers it is adding to decide what to do: adding two `Integers` works in a completely different way than adding two `Doubles`. So how does it actually work? Is it just magical?

In fact, it isn't! And we *can* actually use Haskell to decide what to do based on types—just not in the way we were imagining before. Let's start by taking a look at the type of `(+)`:

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

Hmm, what's that `Num a =>` thingy doing there? In fact, `(+)` isn't the only standard function with a funny double-arrow thing in its type. Here are a few others:

```
(==) :: Eq a   => a -> a -> Bool
(<)  :: Ord a  => a -> a -> Bool
show :: Show a => a -> String
```

So what's going on here?

Type classes

`Num`, `Eq`, `Ord`, and `Show` are *type classes*, and we say that `(==)`, `(<)`, and `(+)` are "type-class polymorphic". Intuitively, type classes correspond to *sets of types* which have certain operations defined for them, and type class polymorphic functions work only for types which are instances of the type class(es) in question. 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 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`.

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 2013-03-14 14:40:00.391563