5 Type Classes and Overloading

There is one final feature of Haskell's type system that sets it apart from other programming languages. The kind of polymorphism that we have talked about so far is commonly called *parametric* polymorphism. There is another kind called *ad hoc* polymorphism, better known as *overloading*. Here are some examples of *ad hoc* polymorphism:

- The literals `1`, `2`, etc. are often used to represent both fixed and arbitrary precision integers.
- Numeric operators such as + are often defined to work on many different kinds of numbers.
- The equality operator (== in Haskell) usually works on numbers and many other (but not all) types.

Note that these overloaded behaviors are different for each type (in fact the behavior is sometimes undefined, or error), whereas in parametric polymorphism the type truly does not matter (`fringe`, for example, really doesn't care what kind of elements are found in the leaves of a tree). In Haskell, *type classes* provide a structured way to control *ad hoc* polymorphism, or overloading.

Let's start with a simple, but important, example: equality. There are many types for which we would like equality defined, but some for which we would not. For example, comparing the equality of functions is generally considered computationally intractable, whereas we often want to compare two lists for equality. (The kind of equality we are referring to here is "value equality," and opposed to the "pointer equality" found, for example, with Java's ==. Pointer equality is not referentially transparent, and thus does not sit well in a purely functional language.) To highlight the issue, consider this definition of the function `elem` which tests for membership in a list:

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

[For the stylistic reason we discussed in Section 3.1, we have chosen to define `elem` in infix form. == and || are the infix operators for equality and logical or, respectively.]

Intuitively speaking, the type of `elem` "ought" to be: `a->[a]->Bool`. But this would imply that == has type `a->a->Bool`, even though we just said that we don't expect == to be defined for all types.

Furthermore, as we have noted earlier, even if == were defined on all types, comparing two lists for equality is very different from comparing two integers. In this sense, we expect == to be *overloaded* to carry on these various tasks.

Type classes conveniently solve both of these problems. They allow us to declare which types are *instances* of which class, and to provide definitions of the overloaded *operations* associated with a class. For example, let's define a type class containing an equality operator:

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

Here `Eq` is the name of the class being defined, and == is the single operation in the class. This declaration may be read "a type `a` is an instance of the class `Eq` if there is an (overloaded) operation ==, of the

appropriate type, defined on it." (Note that == is only defined on pairs of objects of the same type.)

The constraint that a type `a` must be an instance of the class `Eq` is written `Eq a`. Thus `Eq a` is not a type expression, but rather it expresses a constraint on a type, and is called a *context*. Contexts are placed at the front of type expressions. For example, the effect of the above class declaration is to assign the following type to ==:

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

This should be read, "For every type `a` that is an instance of the class `Eq`, == has type `a->a->Bool`". This is the type that would be used for == in the `elem` example, and indeed the constraint imposed by the context propagates to the principal type for `elem`:

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

This is read, "For every type `a` that is an instance of the class `Eq`, `elem` has type `a->[a]->Bool`". This is just what we want---it expresses the fact that `elem` is not defined on all types, just those for which we know how to compare elements for equality.

So far so good. But how do we specify which types are instances of the class `Eq`, and the actual behavior of == on each of those types? This is done with an *instance declaration*. For example:

```
instance Eq Integer where
  x == y                =  x `integerEq` y
```

The definition of == is called a *method*. The function `integerEq` happens to be the primitive function that compares integers for equality, but in general any valid expression is allowed on the right-hand side, just as for any other function definition. The overall declaration is essentially saying: "The type `Integer` is an instance of the class `Eq`, and here is the definition of the method corresponding to the operation ==." Given this declaration, we can now compare fixed precision integers for equality using ==. Similarly:

```
instance Eq Float where
  x == y                =  x `floatEq` y
```

allows us to compare floating point numbers using ==.

Recursive types such as `Tree` defined earlier can also be handled:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a         == Leaf b          =  a == b
  (Branch l1 r1) == (Branch l2 r2)  =  (l1==l2) && (r1==r2)
  _              == _               =  False
```

Note the context `Eq a` in the first line---this is necessary because the elements in the leaves (of type `a`) are compared for equality in the second line. The additional constraint is essentially saying that we can compare trees of `a`'s for equality as long as we know how to compare `a`'s for equality. If the context were omitted from the instance declaration, a static type error would result.

The Haskell Report, especially the Prelude, contains a wealth of useful examples of type classes. Indeed, a class `Eq` is defined that is slightly larger than the one defined earlier:

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

This is an example of a class with two operations, one for equality, the other for inequality. It also demonstrates the use of a *default method*, in this case for the inequality operation `/=`. If a method for a particular operation is omitted in an instance declaration, then the default one defined in the class declaration, if it exists, is used instead. For example, the three instances of `Eq` defined earlier will work perfectly well with the above class declaration, yielding just the right definition of inequality that we want: the logical negation of equality.

Haskell also supports a notion of *class extension*. For example, we may wish to define a class `Ord` which *inherits* all of the operations in `Eq`, but in addition has a set of comparison operations and minimum and maximum functions:

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

Note the context in the `class` declaration. We say that `Eq` is a *superclass* of `Ord` (conversely, `Ord` is a *subclass* of `Eq`), and any type which is an instance of `Ord` must also be an instance of `Eq`. (In the next Section we give a fuller definition of `Ord` taken from the Prelude.)

One benefit of such class inclusions is shorter contexts: a type expression for a function that uses operations from both the `Eq` and `Ord` classes can use the context `(Ord a)`, rather than `(Eq a, Ord a)`, since `Ord` "implies" `Eq`. More importantly, methods for subclass operations can assume the existence of methods for superclass operations. For example, the `Ord` declaration in the Standard Prelude contains this default method for `(<)`:

```
    x < y               =  x <= y && x /= y
```

As an example of the use of `Ord`, the principal typing of `quicksort` defined in Section 2.4.1 is:

```
quicksort               ::  (Ord a) => [a] -> [a]
```

In other words, `quicksort` only operates on lists of values of ordered types. This typing for `quicksort` arises because of the use of the comparison operators < and >= in its definition.

Haskell also permits *multiple inheritance*, since classes may have more than one superclass. For example, the declaration

```
class (Eq a, Show a) => C a where ...
```

creates a class `C` which inherits operations from both `Eq` and `Show`.

Class methods are treated as top level declarations in Haskell. They share the same namespace as ordinary variables; a name cannot be used to denote both a class method and a variable or methods in different classes.

Contexts are also allowed in `data` declarations; see §4.2.1.

Class methods may have additional class constraints on any type variable except the one defining the current class. For example, in this class:

```
class C a where
  m                     :: Show b => a -> b
```

the method `m` requires that type `b` is in class `Show`. However, the method `m` could not place any additional

class constraints on type `a`. These would instead have to be part of the context in the class declaration.

So far, we have been using "first-order" types. For example, the type constructor `Tree` has so far always been paired with an argument, as in `Tree Integer` (a tree containing `Integer` values) or `Tree a` (representing the family of trees containing `a` values). But `Tree` by itself is a type constructor, and as such takes a type as an argument and returns a type as a result. There are no values in Haskell that have this type, but such "higher-order" types can be used in `class` declarations.

To begin, consider the following `Functor` class (taken from the Prelude):

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

The `fmap` function generalizes the `map` function used previously. Note that the type variable `f` is applied to other types in `f a` and `f b`. Thus we would expect it to be bound to a type such as `Tree` which can be applied to an argument. An instance of `Functor` for type `Tree` would be:

```
instance Functor Tree where
  fmap f (Leaf x)       = Leaf   (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

This instance declaration declares that `Tree`, rather than `Tree a`, is an instance of `Functor`. This capability is quite useful, and here demonstrates the ability to describe generic "container" types, allowing functions such as `fmap` to work uniformly over arbitrary trees, lists, and other data types.

[Type applications are written in the same manner as function applications. The type `T a b` is parsed as `(T a) b`. Types such as tuples which use special syntax can be written in an alternative style which allows currying. For functions, `(->)` is a type constructor; the types `f -> g` and `(->) f g` are the same. Similarly, the types `[a]` and `[] a` are the same. For tuples, the type constructors (as well as the data constructors) are `(,)`, `(,,)`, and so on.]

As we know, the type system detects typing errors in expressions. But what about errors due to malformed type expressions? The expression `(+) 1 2 3` results in a type error since `(+)` takes only two arguments. Similarly, the type `Tree Int Int` should produce some sort of an error since the `Tree` type takes only a single argument. So, how does Haskell detect malformed type expressions? The answer is a second type system which ensures the correctness of types! Each type has an associated *kind* which ensures that the type is used correctly.

Type expressions are classified into different *kinds* which take one of two possible forms:

- The symbol * represents the kind of type associated with concrete data objects. That is, if the value *v* has type *t*, the kind of *v* must be *.
- If k_1 and k_2 are kinds, then k_1->k_2 is the kind of types that take a type of kind k_1 and return a type of kind k_2.

The type constructor `Tree` has the kind *->*; the type `Tree Int` has the kind *. Members of the `Functor` class must all have the kind *->*; a kinding error would result from an declaration such as

```
instance Functor Integer where ...
```

since `Integer` has the kind *.

Kinds do not appear directly in Haskell programs. The compiler infers kinds before doing type checking without any need for `kind declarations'. Kinds stay in the background of a Haskell program except when an erroneous type signature leads to a kind error. Kinds are simple enough that compilers should be able to provide descriptive error messages when kind conflicts occur. See §4.1.2 and §4.6 for more information about kinds.

A Different Perspective.

Before going on to further examples of the use of type classes, it is worth pointing out two other views of Haskell's type classes. The first is by analogy with object-oriented programming (OOP). In the following general statement about OOP, simply substituting *type class* for *class*, and *type* for *object*, yields a valid summary of Haskell's type class mechanism:

"*Classes* capture common sets of operations. A particular *object* may be an instance of a *class*, and will have a method corresponding to each operation. *Classes* may be arranged hierarchically, forming notions of super*classes* and sub *classes*, and permitting inheritance of operations/methods. A default method may also be associated with an operation."

In contrast to OOP, it should be clear that types are not objects, and in particular there is no notion of an object's or type's internal mutable state. An advantage over some OOP languages is that methods in Haskell are completely type-safe: any attempt to apply a method to a value whose type is not in the required class will be detected at compile time instead of at runtime. In other words, methods are not "looked up" at runtime but are simply passed as higher-order functions.

A different perspective can be gotten by considering the relationship between parametric and *ad hoc* polymorphism. We have shown how parametric polymorphism is useful in defining families of types by universally quantifying over all types. Sometimes, however, that universal quantification is too broad---we wish to quantify over some smaller set of types, such as those types whose elements can be compared for equality. Type classes can be seen as providing a structured way to do just this. Indeed, we can think of parametric polymorphism as a kind of overloading too! It's just that the overloading occurs implicitly over all types instead of a constrained set of types (i.e. a type class).

Comparison to Other Languages.

The classes used by Haskell are similar to those used in other object-oriented languages such as C++ and Java. However, there are some significant differences:

- Haskell separates the definition of a type from the definition of the methods associated with that type. A class in C++ or Java usually defines both a data structure (the member variables) and the functions associated with the structure (the methods). In Haskell, these definitions are separated.
- The class methods defined by a Haskell class correspond to virtual functions in a C++ class. Each instance of a class provides its own definition for each method; class defaults correspond to default definitions for a virtual function in the base class.
- Haskell classes are roughly similar to a Java interface. Like an interface declaration, a Haskell class declaration defines a protocol for using an object rather than defining an object itself.
- Haskell does not support the C++ overloading style in which functions with different types share a common name.
- The type of a Haskell object cannot be implicitly coerced; there is no universal base class such as `Object` which values can be projected into or out of.
- C++ and Java attach identifying information (such as a VTable) to the runtime representation of an

object. In Haskell, such information is attached logically instead of physically to values, through the type system.
- There is no access control (such as public or private class constituents) built into the Haskell class system. Instead, the module system must be used to hide or reveal components of a class.

A Gentle Introduction to Haskell, Version 98
back next top