Exploring Languages with Interpreters and Functional Programming Chapter 22

H. Conrad Cunningham

5 November 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires use of a browser that supports the display of MathML. A good choice as of November 2018 is a recent version of Firefox from Mozilla.

22 Overloading and Type Classes

22.1 Chapter Introduction

Chapter 5 introduced the concept of overloading. Chapters 13 and 21 introduced the related concepts of type classes and instances.

The goal of this chapter and the next chapter is to explore these concepts in more detail.

The concept of type class was introduced into Haskell to handle the problem of comparisons, but it has had a broader and more profound impact upon the development of the language than its original purpose. This Haskell feature has also had a significant impact upon the design of subsequent languages (e.g. Scala and Rust) and libraries.

The source code for this chapter is in file `TypeClassMod.hs`.

22.2 Polymorphism in Haskell

Chapter 5 surveyed the different kinds of polymorphism. Haskell implements two of these kinds:

1. *Parametric polymorphism* (usually just called “polymorphism” in functional languages), in which a single function definition is used for all types of arguments and results.

   For example, consider the function `length :: [a] -> Int`, which returns the length of any finite list.

2. *Overloading*, in which the same name refers to different functions depending upon the type.

   For example, consider the `(+)` function, which can add any supported number.

Chapter 13 examined parametric polymorphism. This chapter explores the concept of overloading.

22.3 Why Overloading?

Consider testing for membership of a Boolean list, where `eqBool` is an equality-testing function for Boolean values.

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x []     = False
elemBool x (y:ys) = eqBool x y || elemBool x ys
```

We can define `eqBool` using pattern matching as follows:

```
eqBool :: Bool -> Bool -> Bool
eqBool True  False = False
eqBool False True  = False
eqBool _     _     = True
```

The above is not very general. It works for booleans, but what if we want to handle lists of integers? or of characters? or lists of lists of tuples?

The aspects of `elemBool` we need to generalize are the type of the input list and the function that does the comparison for equality.

Thus let's consider testing for membership of a general list, with the equality function as a parameter.

```
elemGen :: (a -> a -> Bool) -> a -> [a] -> Bool
elemGen eqFun x []     = False
elemGen eqFun x (y:ys) = eqFun x y || elemGen eqFun x ys
```

This allows us to define `elemBool` in terms of `elemGen` as follows:

```
elemBool :: Bool -> [Bool] -> Bool
elemBool = elemGen eqBool
```

But really the function `elemGen` is *too general* for the intended function. Parameter `eqFun` could be any

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

function, not just an equality comparison.

Another problem is that equality is a meaningless idea for some data types. For example, comparing functions for equality is a computationally intractable problem.

The alternative to the above to make `(==)` (i.e. equality) an overloaded function. We can then restrict the polymorphism in `elem`'s type signature to those types for which `(==)` is defined.

We introduce the concept of *type classes* to to be able to define the group of types for which an overloaded operator can apply.

We can then restrict the polymorphism of a type signature to a class by using a *context* constraint as `Eq a =>` is used below:

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

We used context constraints in previous chapters. Here we examine how to define the type classes and associate data types with those classes.

22.4 Defining an Equality Class and Its Instances

We can define class Eq to be the set of types for which we define the (==) (i.e. equality) operation.

For example, we might define the class as follows, giving the type *signature(s)* of the associated function(s) (also called the operations or *methods* of the class).

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

A type is made a member or *instance* of a class by defining the signature function(s) for the type. For example, we might define Bool as an instance of Eq as follows:

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

Other types, such as the primitive types Int and Char , can also be defined as instances of the class. Comparison of primitive data types will often be implemented as primitive operations built into the computer hardware.

An instance declaration can also be declared with a context constraint, such as in the equality of lists below. We define equality of a list type in terms of equality of the element type.

```
instance Eq a => Eq [a] where
    []     == []     =  True
    (x:xs) == (y:ys) =  x == y && xs == ys
     _     == _      =  False
```

Above, the == on the left sides of the equations is the operation being defined for lists. The x == y comparison on the right side is the previously defined operation on elements of the lists. The xs == ys on the right side is a recursive call of the equality operation for lists.

Within the class Eq , the (==) function is overloaded. The definition of (==) given for the types of its actual operands is used in evaluation.

In the Haskell standard prelude, the class definition for Eq includes both the equality and inequality functions. They may also have *default* definitions as follows:

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

In the case of class `Eq` , inequality is defined as the negation of equality and vice versa.

An instance declaration must *override* (i.e. redefine) at least one of these functions (in order to break the circular definition), but the other function may either be left with its default definition or overridden.

22.5 Type Class Laws

Of course, our expectation is that any operation `(==)` defined for an instance of `Eq` should implement an "equality" comparison. What does that mean?

In mathematics, we expect equality to be an *equivalence relation*. That is, equality comparisons should have the following properties *for all* values `x`, `y`, and `z` in the type's set.

- **Reflexivity**: `x == x` is `True`.
- **Symmetry**: `x == y` if and only if `y == x`.
- **Transitivity**: if `x == y` and `y == z`, then `x == z`.

In addition, `x /= y` is expected to be equivalent to `not (x == y)` as defined in the default method definition.

Thus class `Eq` has these *type class laws* that every instance of the class should satisfy. The developer of the instance should ensure that the laws hold.

As in many circumstances, the reality of computing may differ a bit from the mathematical ideal. Consider Reflexivity. If `x` is infinite, then it may be impossible to implement `x == x`. Also, this property might not hold for floating point number representations.

22.6 Another Example Class `Visible`

TODO: Replace this example (following Thompson, ed. 2) with a better one?

We can define another example class `Visible`, which might denote types whose values can be displayed as strings. Method `toString` represents an element of the type as a `String`. Method `size` yields the size of the argument as an `Int`.

```
class Visible a where
    toString :: a -> String
    size     :: a -> Int
```

We can make various data types instances of this class:

```
instance Visible Char where
    toString ch  = [ch]
    size _       = 1
```

```haskell
instance Visible Bool where
    toString True  = "True"
    toString False = "False"
    size _         = 1

instance Visible a => Visible [a] where
    toString = concat . map toString
    size     = foldr (+) 1 . map size
```

What type class laws should hold for `Visible`?

There are no constraints on the conversion to strings. However, `size` must return an `Int`, so the "size" of the input argument must be finite and bounded by the largest value in type `Int`.

22.7 Class Extension (Inheritance)

Haskell supports the concept of *class extension*. That is, a new class can be defined that *inherits* all the operations of another class and adds additional operations.

For example, we can derive an ordering class `Ord` from the class `Eq`, perhaps as follows. (The definition in the Prelude may differ from the following.)

```haskell
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a
    -- Minimal complete definition: (<) or (>)
    x <= y                  = x < y || x == y
    x <  y                  = y > x
    x >= y                  = x > y || x == y
    x >  y                  = y < x
    max x y   | x >= y      = x
              | otherwise   = y
    min x y   | x <= y      = x
              | otherwise   = y
```

With the above, we define `Ord` as a *subclass* of `Eq`; `Eq` is a *superclass* of `Ord`.

The above default method definitions are circular: `<` is defined in terms of `>` and vice versa. So a complete definition of `Ord` requires that at least one of these be given an appropriate definition for the type. Method `==` must, of course, also be defined appropriately for superclass `Eq`.

What type class laws should apply to instances of `Ord`?

Mathematically, we expect an instance of class `Ord` to implement a *total order* on its type set. That is, given the comparison operator (i.e. binary relation) `<=`, then the following properties hold *for all* values `x`, `y`, and `z` in the type's set.

- **Reflexivity**: `x <= x` is `True`.
- **Antisymmetry**: `x <= y` and `y <= x`, then `x == y`.
- **Transitivity**: if `x <= y` and `y <= z`, then `x <= z`.
- **Trichotomy** (comparability, totality): `x <= y` or `y <= x`.

A relation that satisfied the first three properties above is a *partial order*. The fourth property requires that all values in the type's set can be compared by `<=`.

In addition to the above laws, we expect `==` (and `/=`) to satisfy the `Eq` type class laws and `<`, `>`, `>=`, `max`, and `min` to satisfy the properties (i.e. default method definitions) given in the class `Ord` declaration.

As an example, consider the function `isort'` (insertion sort), defined in a previous chapter. It uses class `Ord` to constrain the list argument to ordered data items.

```
isort' :: Ord a => [a] -> [a]
isort' []     = []
isort' (x:xs) = insert' x (isort' xs)

insert' :: Ord a => a -> [a] -> [a]
insert' x []    = [x]
insert' x (y:ys)
    | x <= y    = x:y:ys
    | otherwise = y : insert' x ys
```

22.8 Multiple Constraints

Haskell also permits classes to be constrained by two or more other classes.

Consider the problem of sorting a list and then displaying the results as a string:

```
vSort :: (Ord a,Visible a) => [a] -> String
vSort = toString . isort'
```

To sort the elements, they need to be from an ordered type. To convert the results to a string, we need them to be from a `Visible` type.

The multiple contraints can be over two different parts of the signature of a function. Consider a program that displays the second components of tuples if the first component is equal to a given value:

```
vLookupFirst :: (Eq a,Visible b) => [(a,b)] -> a -> String
vLookupFirst xs x = toString (lookupFirst xs x)

lookupFirst :: Eq a => [ (a,b) ] -> a -> [b]
lookupFirst ws x  = [ z | (y,z) <- ws, y == x ]
```

Multiple constraints can occur in an instance declaration, such as might be used in extending equality to cover pairs:

```
instance (Eq a,Eq b) => Eq (a,b) where
    (x,y) == (z,w) =  x == z && y == w
```

Multiple constraints can also occur in the definition of a class, as might be the case in definition of an ordered visible class.

```
class (Ord a,Visible a) => OrdVis a

vSort :: OrdVis a => [a] -> String
```

The case where a class extends two or more classes, as above for `OrdVis` is called *multiple inheritance*.

Instances of class `OrdVis` must satisfy the type class laws for classes `Ord` and `Visible`.

22.9 Built-In Haskell Classes

See Section 6.3 of the Haskell 2010 Language [Marlow 2011] for discussion of the various classes in the Haskell Prelude library.

22.10 Comparison to Other Languages

Let's compare Haskell concept of type class with the class concept in familiar object-oriented languages such as Java and C++.

- In Haskell, a class is a collection of types. In Java and C++, class and type are similar concepts.

  For example, Java's static type system treats the collection of objects defined with a **`class`** construct as a (nominal) type. A **`class`** can be used to implement a type. However, it is possible to implement classes whose instances can behave in ways outside the discipline of the type (i.e. not satisfy the Liskov Substitution Principle).

- Haskell classes are similar in concept to Java and C++ abstract classes except that Haskell classes have no data fields. (There is no multiple inheritance from classes in Java, of course.)

- Haskell classes are similar in concept to Java interfaces. Haskell classes can give default method definitions, a feature that was only added in Java 8 and beyond.

- Instances of Haskell classes are types, not objects. They are somewhat like concrete Java or C++ classes that extend abstract classes or concrete Java classes that implement Java interfaces.

- Haskell separates the definition of a type from the definition of the methods associated with that type. A class in Java or C++ 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 methods defined by a Haskell class correspond to the instance methods in Java or 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. Of course, Haskell class instances do not have implicit receiver object or mutable data fields.

- Methods of Haskell classes are bound statically at compile time, not dynamically bound at runtime as in Java.

- C++ and Java attach identifying information to the runtime representation of an object. In Haskell, such information is attached logically instead of physically to values through the type system.

- 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 Java's `Object` which values can be projected into or out of.

- 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. (In that sense, it is similar to the object-oriented languages Component Pascal and to the systems programming language Rust.)

Type classes first appeared in Haskell, but similar concepts have been implemented in more recently designed languages.

- The imperative systems programming language Rust supports traits, a limited form of type classes.

- The object-functional hybrid language Scala has implicit classes and parameters, which enable a type enrichment programming idiom similar to type classes.

- The functional language PureScript supports Haskell-like type classes.

- The dependently typed functional language Idris supports interfaces, which are, in some ways, a generalization of Haskell's type classes.

- Functional JavaScript libraries such as Ramda have type class-like features

22.11 What Next?

This chapter motivated and explored the concepts of overloading, type classes, and instances in Haskell and compared them to features in other languages.

The next chapter further explores the profound impact of type classes on Haskell.

22.12 Exercises

TODO

22.13 Acknowledgements

In Spring 2017, I adapted and revised this chapter from my previous notes on this topic. I based the previous notes, in part, on the presentations in:

- Chapter 12 of Simon Thompson's textbook *Haskell: The Craft of Functional Programming* (Second Edition) [Thompson 1999].
- Section 5 of *A Gentle Introduction to Haskell Version 98* [Hudak 1999].

For new content on Haskell type class laws, I read the discussions of type class laws on the websites:

- Typeclassopedia [Yorgey 2011]
- StackOverflow
- Reddit

I also reviewed the mathematical definitions of equality, equivalence relations, and total orders on such sites as:

- Wolfram MathWorld
- Wikipedia

In Summer and Fall 2017, I continued to develop this work as Chapter 9, Overloading and Type Classes, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the Overloading and Type Classes chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Most of the existing content became Chapter 22, Overloading and Type Classes (this chapter). I moved the planned content on advanced type class topics (functors, monads) to a planned new chapter 23.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

22.14 References

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[**Hudak 1999**]**:** Paul Hudak, John Peterson, and Joseph H. Fasel. *A Gentle Introduction to Haskell ACM*, https://www.haskell.org/tutorial/haskell-98-tutorial.pdf, 1999.

[**Marlow 2011**]**:** Simon Marlow, editor. *Haskell 2010 Language Report*, <http://www.haskell org/> May 2011.

[**Thompson 1999**]**:** Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

[**Yorgey 2011**]**:** Brent Yorgey. The typeclassopedia, https://wiki.haskell.org/Typeclassopedia, 2011. Originally published in *The Monad. Reader*, Vol. 12, No. 13, pp. 17-68, March 2009.

22.15 Terms and Concepts

Polymorphism in Haskell (parametric polymorphism, overloading); Haskell type system concepts (type classes, overloading, instances, signatures, methods, default definitions, context constraints, class extension, inheritance, subclass, superclass, overriding, multiple inheritance, class laws) versus related Java/C++ type system concepts (abstract and concrete classes, objects, inheritance, interfaces); mathematical concepts (equivalence relation, reflexivity, symmetry, antisymmetry, transitivity, trichotomy, total and partial orders).