7

Type Classes

A symbol is *overloaded* if it has two (or more) meanings, distinguished by type, that are resolved at compile time. For example, in Haskell, as in many other languages, the operator + has (at least) two distinct implementations associated with it, one of type `Int -> Int -> Int`, the other of type `Float -> Float -> Float`. The reason that both of these operations are given the name + is that both compute numeric addition. However, at the implementation level, the two operations are really very different. Because integers are represented in one way (as binary numbers) and floating point numbers in another (as exponent and mantissa, following scientific notation), the way that integer addition combines the bits of its arguments to produce the bits of its result is very different from the way this is done in floating point addition.

A characteristic of overloading is that overloading is *resolved* at compile time. If a function is overloaded, then the compiler must choose between the possible algorithms at compile time. The process of choosing one algorithm from among the possible algorithms associated with an overloaded function is called overload resolution. In many languages, if a function is overloaded, then only the function arguments are used to resolve overloading. For example, consider the following two expressions:

```
3 + 2       {- add two integers -   }
3.0 + 2.0   {- add two floating point numbers -     }
```

Here is how the compiler will produce code for evaluating each expression:

- `3 + 2`: The parsing phase of the compiler will build the parse tree of this expression, and the type-checking phase will compute a type for each symbol. Because the type-checking phase will determine that + must have type `Int -> Int -> Int`, the code-generation phase of the compiler will produce machine instructions that perform integer addition.
- `3.0 + 2.0` : The parsing phase of the compiler will build the parse tree of this expression, and the type-checking phase will compute a type for each symbol. Because the type-checking phase will determine that + must have type `Float -> Float ->`

`Float` , the code-generation phase of the compiler will produce machine instructions that perform floating point addition.

7.1 OVERLOADING VERSUS POLYMORPHISM

Parametric polymorphism can be contrasted with overloading. In an influential historical paper, Christopher Strachey referred to Haskell-style polymorphism as *parametric polymorphism* (although Haskell had not been invented yet) and overloading as *ad hoc polymorphism*. The key difference between parametric polymorphism and overloading is that parameteric polymorphic functions use one algorithm to operate on arguments of many different types, whereas overloaded functions may use a different algorithm for each type of argument.

Overloading is an important language feature because many useful functions are not polymorphic: they work only for types that support certain operations. For example, the `member` function

```
member :: [t] -> t -> Bool
```

does not work for all possible types `t`, but only for types `t` that support equality. This restriction arises from the fact that when applied to a list and an item, the `member` function has to check if the item is equal to any of the elements on the list. Similarly, the sort function `sort :: [t] -> [t]` works only for types `t` that have an ordering. Math computations such as `sumOfSquares :: [t] -> t` work only for types that support numeric operations.

7.2 CHALLENGES IN DESIGNING OVERLOADING MECHANISMS

Many overloading mechanisms focus primarily on overloading for simple mathematical operations and equality, probably because the alternative of having to provide separate symbols to denote equality, addition, subtraction, and other similar operations for different types of arguments seemed unpalatable. We will use these special cases to illustrate some of the challenges in designing a mechanism to support overloading.

Suppose we have a language that overloads addition `+` and multiplication `*`, providing versions that work over values of type `Int` and type `Float` . Now, consider the `double` function, written in terms of the overloaded addition operation:

```
double x = x + x
```

What does this definition mean? A naive interpretation would be to say that `double` is also overloaded, defining one function of type `Int -> Int -> Int` and a second of type `Float -> Float -> Float` . All seems fine, until we consider the function

```
doubles (x,y,z) = (double x, double y, double z)
```

Under the proposed scheme, this definition would give rise to eight different versions! This approach has not been widely used because of the exponential growth in the number of versions.

To avoid this blow-up, language designers have sometimes restricted the definition of overloaded functions. In this approach, which was adopted in Standard ML, basic operations can be overloaded, but not functions defined in terms of them. Instead, the language design specifies one of the possible versions as the meaning of the function. For example, Standard ML give preference to the type `int` over real, so the type (and implementation) of the function `double` would be `int -> int`. If the programmer wanted to define a `double` function over floating point numbers, she would have to explicitly write the type of the function in its definition and give the function a name distinct from the double function on integers. This approach is not particularly satisfying, because it violates a general principle of language design: giving the compiler the ability to define features that programmers cannot.

Overloading support for equality is even more critical than that for arithmetic operations. If it is unpleasant to be required to have two versions of double, `doubleInt` and `doubleFloat`, imagine how much more irritating it would be to be required to have a different symbol to denote equality for every type that supports the comparison! The first version of Standard ML treated equality in the same way it treated overloading for arithmetic operators: equality was overloaded but not functions defined in terms of equality. This design means that functions like `member` do not work in general:

```
member [] y = False
member (x:xs) y = (x == y) || member xs y

member [1,2,3] 2                -- okay if default is integer
member ['a', 'b', 'c'] 'b'   -- illegal
```

To avoid this limitation, the designers of Miranda adopted a different approach. They made equality fully polymorphic:

```
(==) :: t -> t -> Bool
```

With this design, the `member` function is also fully polymorphic, with type `[t] -> t -> Bool`. However, some types do not have a meaningful notion of equality, for example, functions (for which true equality is undecidable) and abstract types that do not provide a definition of equality. In Miranda, equality applied to a function produced a runtime error, while equality applied to a value with abstract type compared the underlying representation, which violates principles of abstraction.

A third approach, which is what Standard ML uses today, is to make equality polymorphic in a limited way. In this design, the type of equality is:

```
(==) :: ''t -> ''t -> Bool
```

The double ticks before the type variable `t` indicate that `t` is an *eqtype variable*, which means a type variable that ranges only over types that admit equality. With this notion, we can give a precise type to the `member` function:

```
member :: [''t] -> ''t -> Bool
```

Applying this `member` function to either integers and characters is legal since both types support equality, but applying `member` to a list of functions or abstract types produces a static error.

This discussion highlights some of the goals for a mechanism to support overloading, namely, the need to avoid a version explosion while allowing users to define new overloaded functions. In addition, a general principle of language design is to avoid special cases, so designs that treat equality and arithmetic operators specially are considered less good than designs that allow any kind of operation to be overloaded.

7.3 TYPE CLASSES

Type Classes are a language mechanism in Haskell designed to support general overloading in a principled way. They address each of the concerns raised above. They provide concise types to describe overloaded functions, so there is no exponential blow-up in the number of versions of an overloaded function. They allow users to define functions using overloaded operations, such as the `double` function we discussed earlier. They allow users to introduce new collections of overloaded functions, so equality and arithmetic operators are not privileged. A key idea in the design is to generalize Standard ML's eqtypes to user-defined collections of types, called a *type class*. Just as all eqtypes support equality, all members of a Haskell type class support some family of operations. We will describe the mechanism in more detail in the following sections. A final benefit of the type class design is that it fits smoothly in a type inference framework.

7.3.1 Intuition

To develop an intuition for how type classes work, consider a typical implementation of a simple quicksort function:

```
qsort :: (a -> a -> Bool) -> [a] -> [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs)
                   ++ [x] ++
                   qsort cmp (filter (not.cmp x) xs)
```

This function works by recursively sorting the elements less than `x`, recursively sorting the elements greater than x, and then concatenating the resulting lists with `x` in the middle. Notice that `qsort` is parameterized by the function `cmp` that indicates how to compare elements of the list. It is this extra argument to `qsort` that allows the function to be parametric rather than overloaded. To see this point, imagine `qsort` did not take `cmp` as an argument

but instead referred to an overloaded < operator. In this case, `qsort` would be overloaded rather than polymorphic. We can use this idea of passing functions as arguments to convert overloaded functions into parametric ones.

Consider the function `poly`, which is overloaded because of the overloaded arithmetic operators `*` and `+`.

```
poly x = x * (x + x)
```

To convert `poly` to a non-overloaded function, we can rewrite it to take the overloaded operators as arguments, just as `qsort` takes the `cmp` function as an argument:

```
poly1 (times, plus) x = times x (plus x x)
```

We can view this extra argument as a *dictionary* that provides the relevant implementations for overloaded operations. Of course, we now have to rewrite all the call sites to pass appropriate definitions for `times` and `plus`:

```
anInt = poly1 (int   _times, int  _plus) 10
aFloat = poly1 (float   _times, float  _plus) 2.71
```

To better document the fact that these extra arguments comprise a dictionary for numeric operations, we can introduce a type for the dictionary and accessor functions to extract the particular overloaded operations. For example:

```
-- Dictionary type
data NumDict a = MkNumDict (a->a->a) (a->a->a)

-- Accessor functions
get_times :: NumDict a -> (a->a->a)
get_times (MkNumDict times plus) = times

get_plus :: NumDict a -> (a->a->a)
get_plus (MkNumDict times plus) = plus
```

We can then rewrite the `poly` function to work with this dictionary type and use the accessor functions.

```
-- Dictionary-passing style
poly2 :: NumDict a -> a -> a
poly2 dict x = let times = get    _times dict
                   plus  = get _plus dict
               in times x (plus x x)
```

This code follows a very precise structure. It uses the accessor functions to create local bindings for the overloaded functions by extracting the corresponding bindings from the dictionary.

Next, we create a dictionary for each type supporting numeric operations, for example, `Int` and `Float` :

```
-- Dictionary creation
intDict   = MkNumDict int   _times int _plus
floatDict = MkNumDict float   _times float _plus
```

Finally, we can rewrite each call to the overloaded `poly` function to take the appropriate dictionary as an argument:

```
-- Passing dictionaries
y = poly2 intDict 10
z = poly2 floatDict 2.71
```

The function `poly2` is a polymorphic function that provides the desired behavior of the overloaded function `poly` .

Of course, this series of transformations would be tedious for a programmer to carry out. To avoid this tedium, Haskell's type class mechanism automates the rewriting process, as we will see in the following sections.

7.3.2 Type Class Declarations

The Haskell type class mechanism is comprised of three components: *type class declarations*, *type class instance declarations*, and *qualified types*. We will describe each of these in turn. Type classes are quite different from classes in object-oriented languages, so don't be confused by the similarity in the name while reading the rest of this chapter!

A type class declaration defines a set of operations and their types and gives the set a name. For example, the following type class declaration introduces the type class `Eq`, which provides equality and inequality operations:

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

This declaration can be read: "Any type `a` that belongs to the `Eq` type class has two defined operators, `==` and `/=`, both of which return a `Bool` when applied to two values of type `a`." Intuitively, these two operations should define equality and inequality, respectively, but nothing in the language enforces this intuition.

As another example, the `poly` function from the previous section would use the `Num` type class, whose declaration looks like the following:

```
class Num a where
   (*) :: a -> a -> a
   (+) :: a -> a -> a
   negate :: a -> a
   ... <other numeric operations> ...
```

7.3.3 Instance Declarations

A type class instance declaration makes a type an instance of a type class by defining the operations of the type class for the given type. For example, the following instance declaration makes the type `Int` an instance of the `Eq` type class, assuming `int_eq` is the primitive equality operation for values of type `Int`.

```
instance Eq Int where
   i == j = int_eq i j
   i /= j = not (int_eq i j)
```

As another example, the instance declaration for `Num Int` looks like:

```
instance Num Int where
   (*) = int_times
   (+) = int_plus
   negate x = int_negate x
   ... <other numeric operations> ...
```

where `int_times`, `int_plus`, and `int_negate` are primitive arithmetic operations on values with type `Int`.

When processing an instance declaration, the compiler ensures that the types of the given definitions match the declared types in the associated type class after replacing the type variable in the class declaration with the type given in the instance declaration. For example, the compiler checks that the type of `int_times` is equal to the type `a -> a -> a` when `a` is replaced by `Int`.

7.3.4 Qualified Types

Qualified types concisely express the operations required to convert an overloaded function into a polymorphic one. For example, the type of the `member` function is a qualified type:

```
member :: Eq t => t -> [t] -> Bool
```

This declaration can be read "For all types `t` that belong to the `Eq` type class, the function `member` takes a value of type `t` and a list of `t` values, and returns a `Bool`. The "`Eq t =>` " prefix of this type is what makes it qualified. The qualification restricts the `member` function to arguments that belong to the `Eq` type class and hence support the == operation.

Similarly, the functions `double` and `poly` have qualified types:

```
double :: Num t => t -> t
poly   :: Num t => t -> t
```

indicating that they can be applied to a value of any type `t` as long as `t` belongs to the `Num` class.

Besides the `Eq` and `Num` type classes, other standard type classes include the `Ord` class for types that support ordering, the `Show` class for types that can be converted to strings, the `Read` class for types whose values can be constructed from strings, the `Arbitrary` class for types for which random values can be generated, the `Enum` class for types whose members can be enumerated, and the `Bounded` class for types that have lower and upper bounds. Every type class declaration introduces a new class name that can be used in qualified types. Examples that use these type classes are:

```
sort :: Ord t => [t] -> [t]
serialize :: Show t => t -> String
```

Each of these types precisely documents the operations that must be supported by the type variables. For `sort`, the type `t` must support comparison, while for `serialize`, the type `t` must support conversion to strings. If a function is not qualified, then it must be purely polymorphic and work for any type whatsoever.

Note that qualified types give a concise way to represent overloading. Consider the `doubles` function, which we saw earlier, that conceptually has eight different types. All eight of these types can be expressed with a single qualified type:

```
doubles :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
```

Note also that this qualified type shows that a single type can have multiple type variables constrained by type classes. In this case, all three are constrained by the `Num` type class, but in the general case, each type variable can be constrained by a different type class. In addition, a single type variable can be constrained by multiple type classes. For example, a function that serialized all values in a list that were greater than a particular value would have to belong to both the `Ord` and `Show` type classes:

```
serializeGreater :: (Ord a, Show a) => [a] -> a -> [String]
```

7.4 COMPILING TYPE CLASSES

Using the information provided by type class declarations, instance declarations, and qualified types, the Haskell compiler automates the rewriting process we went through by hand in 7.3.1. From a type class declaration, the Haskell compiler generates a new dictionary type and corresponding accessor functions. From a type class instance declaration, the Haskell compiler generates a dictionary value. For each function definition with a qualified type, the Haskell compiler rewrites the function definition to take a dictionary parameter following the same pattern as we saw in the definition of `poly2` . For each call to a function with a qualified type, the Haskell compiler inserts the appropriate dictionary as an extra argument.

We will consider each of these translations in more detail using examples, starting with the translation for type class declarations. Given the `Num` type class declaration, the Haskell compiler will generate a data type declaration for the `Num` dictionary and a selector function for each operator in the dictionary. A value of type `Num t` is a dictionary of the numeric operations appropriate for values of type `t`.

```
data Num n = MkNum (n -> n -> n)
                   (n -> n -> n)
                   (n -> n)
                   ...types of other numeric operators...

(+) :: (Num n) -> n -> n -> n
(+) (MkNum  _ plus  _ ...) = plus
```

The bottom two lines are defining the + operator to extract the `plus` function from the `Num` dictionary.

From an instance declaration, the compiler generates a dictionary definition. For example, for the `Int` instance declaration for the `Num` class, the compiler generates

```
dNumInt :: Num Int
dNumInt = MkNum int  _times
                int _plus
                int _negate
                ... other numeric operators ...
```

The value `dNumInt` is the `Num` dictionary for the `Int` type.

For functions with qualified types, the compiler rewrites the function definition to replace the qualified type with a parameter having the type of the corresponding dictionary. For example, the compiler will replace the qualified `double` function

```
double :: Num t => t -> t
double x = x + x
```

with the corresponding `double` function written in dictionary-passing style:

```
double :: Num t -> t -> t
double d x = (+) d x x
```

Note that the qualified type `Num t =>` prefix is replaced by an extra argument `d` of type `Num t`. During the translation, the compiler also rewrote the call to `+` so that it uses prefix rather than infix syntax. The body of the rewritten `double` function applies the `(+)` accessor function to extract the appropriate definition of addition from the dictionary `d` and then applies this function to the argument `x` twice.

For calls to functions with qualified types, the compiler rewrites the call to insert the appropriate dictionary as an extra argument. The compiler uses the inferred *static* type of the call site to decide which dictionary to insert. This practice is different from dynamic dispatch in object-oriented languages, which uses the dynamic type of objects to determine which method to invoke. For example, given the call

```
let x :: Int = double 1
```

the compiler infers based on the static type of `x` that the rewritten `double` function should be called with the integer dictionary `dNumInt` rather than the other possible `Num` dictionaries.

Functions that are qualified over multiple type classes are rewritten to take multiple dictionaries as arguments. For example, the `doubles` function would be rewritten to take three `Num` dictionaries as arguments, while the `serializeGreater` function would take a `Show` dictionary and an `Ord` dictionary as arguments.

```
doubles :: (Num a, Num b, Num c) -> (a, b, c) -> (a, b, c)
doubles (da,db,dc) (x, y, z) =
                  (double da x, double db y, double dc z)

serializeGreater :: (Ord a, Show a) -> [a] -> a -> [String]
serializeGreater (da   _ord, da _show) items pivot = ...
```

7.5 COMPOSITIONALITY

Composability is a desirable feature in language design because it means programmers can build more complex structures out of simpler ones. Haskell's type class mechanism supports composability in two dimensions. First, it allows users to define overloaded functions from other overloaded functions, and second, it allows compound instances to be defined from simpler instances. We will discuss each of these in turn.

We saw at the beginning of this chapter that a hallmark of a good design for overloading is allowing users to define their own overloaded functions from existing ones. Haskell's type class mechanism supports overloaded functions calling other overloaded functions by threading the appropriate dictionaries through the calls. For example, consider the function

`prodDouble` that calls the overloaded function `double`

```
prodDouble :: Num t => t -> t -> t
prodDouble x y = (double x) * (double y)
```

The Haskell compiler will translate this code to

```
prodDouble :: Num t -> t -> t -> t
prodDouble d x y = (*) d (double d x) (double d y)
```

adding the `Num` dictionary `d` as a parameter and using the accessor `(*)` to extract the multiplication operation from this dictionary. It will also pass the dictionary on to the two calls to `double` , so each of these functions will be able to extract the necessary addition function.

The second form of composability allows programmers to declare complex instance declarations from simpler ones. To see the utility of such a capability, consider defining an equality operation on lists. If a type `t` supports equality, we can define an equality for type `[t]` without knowing the details of the type `t`. Such a function would take two lists of type `[t]` and compare the elements using the definition of equality for values of type `t`. Given this fact, it would be annoying to have to define equality for lists of `Int` s and then again for lists of `Float` s, and so on! Haskell's instance declarations support a form that allows such generic definitions. The following instance declaration defines list equality in terms of an equality function for the list element type:

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

This instance declaration says that if `t` belongs to the `Eq` class ("`Eq t =>` "), then the type `[t]` also belongs to the `Eq` class. Just like the instance declarations we saw before, this instance declaration defines the methods of the class. In this case, the instance declaration defines the == operation using pattern matching. The first clause says that two empty lists are always equal. The second clause says that two lists are equal if the heads of the lists are equal (`x` and `y`) and if the tails are equal (`xs` and `ys`). Note that the heads have type `t` and so it is `t`'s version of equality that is used to compare these elements, while the tails have type `[t]` , and so it is the operation we are currently defining that is being used recursively to determine if the tails are equal. Finally, if neither of the first two patterns match, then the lists cannot be equal.

Instance declarations translate to dictionary value declarations, so this instance declaration for `Eq [t]` must also generate a dictionary declaration. However, to construct a list dictionary, we need the dictionary for the element type so we can compare head elements. Hence the instance declaration for lists translates to a function from the dictionary for the element type to the dictionary for the list. Given an element `Eq` dictionary, we can construct

a list `Eq` dictionary:

```
dEqList :: Eq a -> Eq [a]            -- List Dictionary
dEqList d = MkEq eqlist
   where
     eqlist [] [] = True
     eqlist (x:xs) (y:ys) = (==) d x y && eqlist xs ys
     eqlist  _ _ = False

data Eq = MkEq (a->a->Bool)          -- Dictionary type
(==) (MkEq eq) = eq                  -- Accessor function
```

In this code, if `d` is the dictionary for the element type, then `dEqList d` is the list dictionary. The function `dEqList` applies the `Eq` dictionary constructor `MkEq` to the value `eqlist`, which is the raw dictionary for lists, constructed from the operator definitions in the list instance declaration. Notice how the `d` dictionary is used to find the equality function with which to compare the list head elements while the list dictionary `eqlist` is used to compare the tails.

7.6 FLESHING OUT THE DESIGN

So far, we have looked at the core elements of Haskell's type class mechanism. In this section, we will look at various additional features of the design that make using it more convenient for programmers.

7.6.1 Subclasses

Sometimes membership in one type class logically implies membership in another. For example, we could treat the `Eq` and `Num` type classes separately, listing each in the qualified type if we need operations from either:

```
mem_double :: (Eq t, Num t) => t -> [t] -> Bool
mem_double x xs = member (double x) xs
```

However, any type providing the operations of the `Num` class would also want to provide the operations in the `Eq` type class. The *subclass declaration* form expresses this relationship.

```
class Eq t => Num t where
   (+) :: t -> t -> t
   (*) :: t -> t -> t
   ...
```

This declaration says that a type `t` that belongs to the `Eq` type class "`Eq t =>` " can also

belong to the `Num` type class if it defines the numeric operations `(+)`, `(*)`, and so on. In other words, a type `t` can belong to the `Num` type class only if it also belongs to the `Eq` type class.

Given this subclass declaration for the `Num` type class, we do not have to list both the `Eq` and `Num` type classes in the type of the `mem_double` function:

```
mem_double :: (Num t) => t -> [t] -> Bool
mem_double x xs = member (double x) xs
```

Instead, the code can simply list the `Num` class.

Just as for simple class declarations, the Haskell compiler generates a dictionary and a collection of accessor functions from subclass declarations. The compiler adds the methods of the superclass to the dictionary for the new class and generates corresponding accessor functions.

7.6.2 Default Methods

Sometimes in defining a type class, there is a sensible default version of some (or all) of the methods in the class, often in terms of the other methods of the class. For example, the `Eq` class contains two methods, equality `(==)` and inequality `(/=)`. Once one of these methods is defined, it is trivial to define the second in terms of the first. The type class mechanism supports this scenario by allowing programmers to declare *default methods* in type class declarations. Such defaults are overridden for a type `T` if there is type class instance declaration for `T` that gives a more specific implementation.

For example, the `Eq` class has the following default methods:

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

This code specifies that two values `x` and `y` are equal if they are not unequal, and two values are unequal if they are not equal. In defining an instance of the `Eq` class, a programmer need only define one of the two methods, the default will suffice for the second. Note that the programmer must define at least one of the methods; otherwise invoking either equal or unequal will cause an infinite loop. This expectation is recorded in the comment above the `Eq` class declaration, but the compiler does not check that the program satisfies the expectation.

7.6.3 Deriving

When defining a new datatype, it is often desirable to make the new type an instance of a number of type classes, such as the `Eq` or `Show` type classes. Obvious implementations for

the required methods of these type classes exist, but it can be tedious for the programmer to write them down. To avoid this tedium, the Haskell compiler supports automatic deriving for certain type classes, including the `Read`, `Show`, `Bounded`, `Enum`, `Eq`, and `Enum` classes. For example, given the following datatype declaration for the type `Color`,

```
data Color = Red | Green | Blue
     deriving (Show, Read, Eq, Ord)
```

the compiler generates code to serialize colors to strings (`Show`), to convert such serialized strings back to values of type `Color` (`Read`), to compare colors for equality (`Eq`), and to order colors, using the order that the colors were written in the type declaration (`Ord`).

The compiler supports the deriving mechanism for only a fixed collection of type classes, because it is only these classes that it knows how to generalize to any datatype declaration. It is not obvious, for example, how to add two values of an arbitrary datatype, so the compiler has no way to make an arbitrary datatype an instance of the `Num` type class.

This design is not fully satisfactory from a programming-language design point of view. Why should deriving be restricted to a fixed collection of type classes? Why shouldn't the programmer be able to declare new type classes and have the compiler derive instances for these type classes? Furthermore, why shouldn't programmers be able to specify what code the compiler should generate for new datatypes and new type classes? Providing better support for deriving is an active area of research.

7.6.4 Numeric Literals

One challenge for any programming language is how to handle numeric literals. In mathematical notation, the symbol `1` can denote an integer, a real number, a fraction, a complex number, and so on. In a programming language, however, each of these different kinds of things is represented differently, and so the compiler needs to be able to distinguish which kind of thing the programmer meant when writing down the number `1`. This need leads many programming languages to require that the various usages be syntactically distinct, for example, writing `1` only for the integer one and `1.0` for the real version.

Haskell leverages the type class mechanism to allow the programmer to use the numeric literal `1` as any kind of numeric type. To enable this flexibility, the `Num` type class has an additional member, `fromInteger`, that converts any value of type `Integer` to the numeric type `t`:

```
class Eq t => Num t where
   (+) :: t -> t -> t
   (*) :: t -> t -> t
   fromInteger :: Integer -> t
```

Note that the type of `fromInteger` is `Num t => Integer -> t`. Consequently, following the normal procedure for compiling overloaded functions, the compiler will convert each occurrence of `fromInteger` in the program text to code that looks up the type-specific

definition of `fromInteger` in the appropriate numeric dictionary. The compiler uses the static type of the context in which the `fromInteger` function is used to determine which dictionary is "appropriate."

With this mechanism in place, all that is needed to support overloaded numeric literals is to have the parser convert occurrences of integer literals in the program to calls to the `fromInteger` overloaded function. For example, if the literal `1` appears in the program, then the parser returns a parse tree for the code `fromInteger 1`, making it look to the rest of the compiler like the programmer had written the more verbose form. Thus if a literal `1` appears in a context expecting a `Float`, then the normal type class compilation process results in calling the `fromInteger` function defined in the `Float` instance of the `Num` type class, yielding a value of type `Float`. If it appears in a context expecting an `Integer`, then the integer version of `fromInteger` is called, yielding a value of type `Integer`, and so on.

This ability to overload numeric literals extends to user-defined types. For example, we can define a type `Cpx` of complex numbers. Making `Cpx` an instance of the `Num` type class allows us to write `1` to denote a complex number as well as an `Integer` or a `Float`.

```
data Cpx a = Cpx a a
    deriving (Eq, Show)

instance Num a => Num (Cpx a) where
    (Cpx r1 i1) + (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
    fromInteger n = Cpx (fromInteger n) 0
    ...
```

The definition of the `fromInteger` function in this instance declaration uses the `fromInteger` function defined for the type of `n` to produce a representation for the real portion of the complex number. It sets the imaginary portion to be the value `0`. With this instance declaration, we can use values of type `Cpx` in any context requiring a type from the `Num` class.

```
c1 = 1 :: Cpx Int
c2 = 2 :: Cpx Int
c3 = c1 + c2

c4 = prodDouble c3
```

In this example, `c1` is bound to the complex number denoted by the literal `1` and `c2` is bound to the complex number denoted by the literal `2`. Such complex numbers can be added just like integers and can be passed to user-defined functions such as `prodDouble`, which work for any type belonging to the `Num` class.

7.7 TYPE INFERENCE

The type inference algorithm described in the previous chapter can be extended to include the overloaded types introduced using type classes. In this richer setting, the type inference algorithm infers a qualified type, of the form `Q => T`. The type `T` is a purely polymorphic type of the kind from Chapter 6, inferred using the same procedure as before. The constraint `Q` is a set of type class predicates that the types mentioned in `T` must belong to. For example, consider the function `example` :

```
example z xs =
   case xs of
     [] -> False
     (y:ys) -> y > z || (y==z && ys == [z])
```

Running simple type inference on this example produces a polymorphic type `t -> [t] -> Bool`. Examining the body of the `example` function for uses of overloaded functions, we can collect the required constraints. In particular, we get the constraint `Ord t` from the comparison `y > z`, the constraint `Eq t` from the equality check `y == z`, and the constraint `Eq [t]` from the equality check `ys == [z]`. Combining this information results in the inferred type `(Ord t, Eq t, Eq [t]) => t -> [t] -> Bool` for the `example` function. However, as we will see, the constraint `(Ord t, Eq t, Eq [t])` is more complicated than necessary.

There are a number of ways to simplify constraint sets. First, we can eliminate duplicate constraints. If the constraint set is {`Eq t, Eq t`}, we can simplify the set to be just {`Eq t`}. Second, we can use information from compound instance declarations. For example, the instance declaration for `Eq [t]` describes how to construct an instance of `Eq [t]` from an instance of `Eq t`. Hence given `Eq t`, we know that `Eq [t]` holds as well. In other words, given that we know `Eq t`, we also know `Eq [t]`. Hence we can simplify the constraint set {`Eq [t], Eq t`} to be just {`Eq t`}. Third, we can use information from a subclass declaration. For example, the class declaration for the `Ord` class has the form:

```
class Eq t => Ord t where
   (<) :: t -> t -> Bool
   ...
```

This declaration indicates that any instance of the class `Ord` must also be an instance of the class `Eq`. Hence, we may simplify the constraint {`Ord t, Eq t`} to {`Ord t`}. Applying these rules to the constraint set `(Ord t, Eq t, Eq [t])` produces the simpler set {`Ord t`}. The inferred type for the `example` function is thus

```
example :: (Ord t) => t -> [t] -> Bool
```

In this setting, type errors are reported when a predicate that is required to hold for a

particular type is known not to do so. For example, the type checker knows that the type `Char` does not belong to the `Num` type class, and so attempting to add `1` to the character `'a'` produces an error message:

```
Prelude> 'a' + 1
    No instance for (Num Char)
    arising from a use of '+' at <interactive>:1:0-6
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 'a' + 1
    In the definition of 'it': it = 'a' + 1
```

The error message explains exactly this point. To add one to `'a'`, the type `Char` must be an instance of class `Num`. Because it is not, the type checker reports a type error.

7.8 CONSTRUCTOR CLASSES

So far, we have seen how to make types instances of type classes. Haskell also supports making *type constructors* instances of type classes, where a type constructor is a function at the type level. A type constructor takes a type as an argument and returns a type as a result. For example, `[-]` is a type constructor in Haskell. If `T` is a type, then `[T]` is the corresponding list type.

We will first motivate type constructor classes by showing a family of functions involving different type constructors that all share the same structure. In particular, we will focus on various data structures for which it is useful to have a `map` function. First, consider the `map` function for lists, which we will write as `mapList` for now to distinguish it from other versions of the `map` function.

```
mapList:: (a -> b) -> [a] -> [b]
mapList f [] = []
mapList f (x:xs) = f x : mapList f xs

result = mapList (  \x->x+1) [1,2,4]
```

The `mapList` function takes a function `f` as an argument and a list and returns the result of applying `f` to every element in the input list.

Now consider a tree data structure, `Tree`, and a `mapTree` function

```
Data Tree a = Leaf a | Node(Tree a, Tree a)
    deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node(l,r)) = Node (mapTree f l, mapTree f r)
```

```
t1 = Node(Node(Leaf 3, Leaf 4), Leaf 5)
result = mapTree (  \x->x+1) t1
```

The `mapTree` function takes a function `f` as an argument and a tree and returns the result of applying `f` to every element in the input tree.

Finally, consider the `Opt` data structure and the corresponding `mapOpt` function:

```
Data Opt a = Some a | None
   deriving Show
mapOpt :: (a -> b) -> Opt a -> Opt b
mapOpt f None = None
mapOpt f (Some x) = Some (f x)

o1 = Some 10
result = mapOpt (  \x->x+1) o1
```

The `mapOpt` function takes a function `f` as an argument and an option and returns the result of applying `f` to every element in the input option.

Note that all of these map functions share the same structure:

```
mapList :: (a -> b) -> [a]     -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
mapOpt  :: (a -> b) -> Opt a  -> Opt b
```

They can all be written as

```
map :: (a -> b) -> g a -> g b
```

where `g` is `[-]` for lists, `Tree` for trees, and `Opt` for options. Note that `g`, `[-]`, `Tree`, and `Opt` are all examples of type constructors, functions from types to types.

We can capture this pattern in a *constructor class*, which is a type class where the predicate ranges over type constructors instead of over simple types:

```
class HasMap g where
   map :: (a -> b) -> g a -> g b
```

This declaration introduces the constructor class `HasMap`. For a type constructor `g` to belong to the class `HasMap`, there must be a function `map` which takes a function `a -> b` and a value of type `g a` and returns a value of type `g b`.

We can then make lists, trees, and options instances of this type constructor class

```
instance HasMap [] where
   map = mapList

instance HasMap Tree where
   map = mapTree

instance HasMap Opt where
   map = mapOpt
```

With these declarations in place, we can use the overloaded symbol `map` to map over all three kinds of data structures:

```
Main> map ( \x->x+1) [1,2,3]
  [2,3,4]
  it :: [Integer]
Main> map ( \x->x+1) (Node(Leaf 1, Leaf 2))
  Node (Leaf 2,Leaf 3)
  it :: Tree Integer
Main> map ( \x->x+1) (Some 1)
  Some 2
  it :: Opt Integer
```

7.9 TYPE CLASSES VS. OBJECT-ORIENTED CLASSES

Given the terminology associated with type classes, it is natural to ask how type classes are related to object-oriented classes. There are a number of similarities. A type class defines a collection of method names and associated types. In this way, type classes are similar to interfaces in Java. A type class can define default implementations for some methods, rather like abstract classes in C++. Instance declarations specify the implementations of the methods of a type class, rather like an object-oriented class that is defined to implement an interface or abstract class. With type classes, the collection of methods is gathered into a dictionary structure, which is similar to a method table in an object-oriented language. When a method is invoked, the type class code looks up the appropriate method in a dictionary, while the object-oriented program looks up the method in a method suite.

Despite these similarities, however, there are a number of differences. First, the algorithm that is used to resolve the method invocation in the two cases is different. With type classes, the *static* type of the arguments and the expected return type of the method is used to select the appropriate dictionary *at compile time*. With object-oriented programs, the method suite is associated with objects at run-time, and the appropriate method definition is selected based on the *dynamic type* of the receiver object. Hence code selection with type classes is fundamentally static, while code selection in object-oriented languages is fundamentally dynamic.

Second, in type classes, the appropriate method to invoke is determined by all the arguments to the method as well as its result type. For example, the `fromInteger` function that

we discussed in conjunction with overloaded numeric literals has type `Num t => Integer -> t`. Hence it is the *result type* of this function that determines which dictionary to use. In contrast, main-stream object-oriented languages use only the receiver object to select the method body to run.

Third, existing types can be made instances of new type classes by adding new instance declarations. For example, if we define a new type class `PrettyPrint`, we can make any previously defined type an instance of `PrettyPrint` by adding the appropriate instance declaration. In contrast, object-oriented languages typically require a class to specify the interfaces it implements and the abstract superclasses from which it inherits when the class is declared.

Finally, type classes are based on parametric polymorphism and do not require subtyping, while object-oriented languages make heavy use of subtyping but have only recently started to incorporate parametric polymorphism.

7.10 CHAPTER SUMMARY

Overloading is a form of polymorphism in which the same program identifier is allowed to have multiple implementations. Which implementation a given occurrence of the identifier refers to is determined by the static type of the identifier. Overloading is particularly important for equality operators and numeric operations. Early designs for overloading mechanisms restricted the collection of symbols that could be overloaded and prevented users from defining their own overloaded functions.

Type classes in Haskell are a principled language feature designed to support overloading. A type class declaration allows the programmer to introduce a named collection of overloaded function names and specifies the types of these functions. An instance declaration allows the programmer to specify how a particular type belongs to a named type class by giving definitions for the functions declared in the type class. Qualified types described overloaded functions. The qualifier concisely describes the type classes that are used in the function. Type classes are translated away during compilation. Each type class declaration is converted into the type of method dictionary associated with the class. Each instance declaration is translated to a dictionary. Functions with qualified types are converted to purely polymorphic functions that take an extra dictionary parameter as an argument. References to overloaded functions are converted to code that looks up the appropriate implementation within the method dictionary.

Type classes solve a variety of problems related to overloading. They provide concise types to describe overloaded functions, so there is no exponential blow-up in the number of versions of an overloaded function. They allow users to define functions using overloaded operations, such as the `double` function. They allow users to introduce new collections of overloaded functions, so equality and arithmetic operators are not privileged. A key idea in the design is to generalize Standard MLs eqtypes to user-defined collections of types. Just as all eqtypes support equality, all members of a Haskell type class support some family of operations. Additional benefits of the type class design is that it fits smoothly in a type inference framework and can be generalized to allow overloading on type constructors as well as types.

The difference between parametric polymorphism and overloading is that parametric polymorphism allows one algorithm to be given many types, whereas overloading involves different algorithms. For example, the function `+` is overloaded in many languages. In

an expression adding two integers, the integer addition algorithm is used. In adding two floating-point numbers, a completely different algorithm is used for computing the sum.

Type classes are differ from classes in object-oriented languages despite the similarity in terminology. A key difference is in resolving the code to which a method name refers. In type classes, the code to run is determined statically, while in object-oriented languages, the code is determined dynamically.