3

Types and classes

In this chapter we introduce types and classes, two of the most fundamental concepts in Haskell. We start by explaining what types are and how they are used in Haskell, then present a number of basic types and ways to build larger types by combining smaller types, discuss function types in more detail, and conclude with the concepts of polymorphic types and type classes.

3.1 Basic concepts

A *type* is a collection of related values. For example, the type `Bool` contains the two logical values `False` and `True`, while the type `Bool -> Bool` contains all functions that map arguments from `Bool` to results from `Bool`, such as the logical negation function `not`. We use the notation `v :: T` to mean that `v` is a value in the type `T`, and say that `v` *has type* `T`. For example:

```
False :: Bool

True :: Bool

not :: Bool -> Bool
```

More generally, the symbol `::` can also be used with expressions that have not yet been evaluated, in which case the notation `e :: T` means that evaluation of the expression `e` will produce a value of type `T`. For example:

```
not False :: Bool

not True :: Bool

not (not False) :: Bool
```

In Haskell, every expression must have a type, which is calculated prior to evaluating the expression by a process called *type inference*. The key to this process is the following simple typing rule for function application, which states that if `f` is a function that maps arguments of type `A` to results of type `B`, and `e` is an expression of type `A`, then the application `f e` has type `B`:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{e :: A}}{\texttt{f e :: B}}$$

For example, the typing `not False :: Bool` can be inferred from this rule using the fact that `not :: Bool -> Bool` and `False :: Bool`. On the other hand, the expression `not 3` does not have a type under the above rule, because this would require that `3 :: Bool`, which is not valid because `3` is not a logical value. Expressions such as `not 3` that do not have a type are said to contain a *type error*, and are deemed to be invalid expressions.

Because type inference precedes evaluation, Haskell programs are *type safe*, in the sense that type errors can never occur during evaluation. In practice, type inference detects a very large class of program errors, and is one of the most useful features of Haskell. Note, however, that the use of type inference

does not eliminate the possibility that other kinds of error may occur during evaluation. For example, the expression `1 'div' 0` is well-typed, but produces an error when evaluated because the result of division by zero is undefined.

The downside of type safety is that some expressions that evaluate successfully will be rejected on type grounds. For example, the conditional expression `if True then 1 else False` evaluates to the number `1`, but contains a type error and is hence deemed invalid. In particular, the typing rule for a conditional expression requires that both possible results have the same type, whereas in this case the first such result, `1`, is a number and the second, `False`, is a logical value. In practice, however, programmers quickly learn how to work within the limits of the type system and avoid such problems.

In GHCi, the type of any expression can be displayed by preceding the expression by the command `:type`. For example:

```
> :type not
not :: Bool -> Bool

> :type False
False :: Bool

> :type not False
not False :: Bool
```

3.2 Basic types

Haskell provides a number of basic types that are built-in to the language, of which the most commonly used are described below.

`Bool` – logical values

This type contains the two logical values `False` and `True`.

`Char` – single characters

This type contains all single characters in the Unicode system, the international standard for representing text-based information. For example, it contains all characters on a normal English keyboard, such as `'a'`, `'A'`, `'3'` and `'_'`, as well as a number of control characters that have a special effect, such as `'\n'` (move to a new line) and `'\t'` (move to the next tab stop). As in most programming languages, single characters must be enclosed in single forward quotes `' '`.

`String` – strings of characters

This type contains all sequences of characters, such as `"abc"`, `"1+2=3"`, and the empty string `""`. Again, as is standard in most programming languages, strings of characters must be enclosed in double quotes `" "`.

`Int` – fixed-precision integers

This type contains integers such as `-100`, `0`, and `999`, with a fixed amount of memory being used for their storage. For example, the GHC system has values of type `Int` in the range -2^{63} to $2^{63} - 1$. Going outside this range can give unexpected results. For example, evaluating `2^63 :: Int` gives a negative number as the result, which is incorrect. (The use of `::` in this example forces the result to be an `Int` rather than some other numeric type.)

`Integer` – arbitrary-precision integers

This type contains all integers, with as much memory as necessary being used for their storage, thus avoiding the imposition of lower and upper limits on the range of numbers. For example, evaluating `2^63 :: Integer` using any Haskell system will produce the correct result.

Apart from the different memory requirements and precision for numbers of type `Int` and `Integer`, the choice between these two types is also one of performance. In particular, most computers have built-in hardware for fixed-precision integers, whereas arbitrary-precision integers are usually processed using the slower medium of software, as sequences of digits.

`Float` – single-precision floating-point numbers

This type contains numbers with a decimal point, such as `-12.34`, `1.0`, and `3.1415927`, with a fixed amount of memory being used for their storage. The term *floating-point* comes from the fact that the number of digits permitted after the decimal point depends upon the size of the number. For example, evaluating `sqrt 2 :: Float` using GHCi gives the result `1.4142135` (the library function `sqrt` calculates the square root of a floating-point number), which has seven digits after the decimal point, whereas `sqrt 99999 :: Float` gives `316.2262`, which only has four digits after the point.

`Double` – double-precision floating-point numbers

This type is similar to `Float`, except that twice as much memory is used for storage of these numbers to increase their precision. For example, evaluating `sqrt 2 :: Double` gives `1.4142135623730951`. Using floating-point numbers is a specialist topic that requires a careful treatment of rounding errors, and we don't often use such numbers in this book.

We conclude this section by noting that a single number may have more than one numeric type. For example, the number `3` could have type `Int`, `Integer`, `Float` or `Double`. This raises the interesting question of what type such numbers should be assigned during the process of type inference, which will be answered later in this chapter when we consider type classes.

3.3 List types

A *list* is a sequence of *elements* of the same type, with the elements being enclosed in square parentheses and separated by commas. We write `[T]` for the type of all lists whose elements have type `T`. For

example:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]

["One","Two","Three"] :: [String]
```

The number of elements in a list is called its *length*. The list `[]` of length zero is called the empty list, while lists of length one, such as `[False]`, `['a']`, and `[[]]` are called singleton lists. Note that `[[]]` and `[]` are different lists, the former being a singleton list comprising the empty list as its only element, and the latter being simply the empty list that has no elements.

There are three further points to note about list types. First of all, the type of a list conveys no information about its length. For example, the lists `[False,True]` and `[False,True,False]` both have type `[Bool]`, even though they have different lengths. Secondly, there are no restrictions on the type of the elements of a list. At present we are limited in the range of examples that we can give because the only non-basic type that we have introduced at this point is list types, but we can have lists of lists, such as:

```
[['a','b'],['c','d','e']] :: [[Char]]
```

Finally, there is no restriction that a list must have a finite length. In particular, due to the use of lazy evaluation in Haskell, lists with an infinite length are both natural and practical, as we shall see in chapter 15.

3.4 Tuple types

A *tuple* is a finite sequence of *components* of possibly different types, with the components being enclosed in round parentheses and separated by commas. We write `(T1,T2,...,Tn)` for the type of all tuples whose `i`th components have type `Ti` for any `i` in the range `1` to `n`. For example:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)

("Yes",True,'a') :: (String,Bool,Char)
```

The number of components in a tuple is called its *arity*. The tuple `()` of arity zero is called the empty tuple, tuples of arity two are called pairs, tuples of arity three are called triples, and so on. Tuples of arity one, such as `(False)`, are not permitted because they would conflict with the use of parentheses to make the evaluation order explicit, such as in `(1+2)*3`.

In a similar manner to list types, there are three further points to note about tuple types. First of all, the type of a tuple conveys its arity. For example, the type `(Bool,Char)` contains all pairs comprising a first component of type `Bool` and a second component of type `Char`. Secondly, there are no restrictions on the types of the components of a tuple. For example, we can now have tuples of tuples, tuples of lists, and lists of tuples:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(['a','b'],[False,True]) :: ([Char],[Bool])

[('a',False),('b',True)] :: [(Char,Bool)]
```

Finally, note that tuples must have a finite arity, in order to ensure that tuple types can always be inferred prior to evaluation.

3.5 Function types

A *function* is a mapping from arguments of one type to results of another type. We write `T1 -> T2` for the type of all functions that map arguments of type `T1` to results of type `T2`. For example, we have:

```
not :: Bool -> Bool

even :: Int -> Bool
```

(The library function `even` decides if an integer is even.) Because there are no restrictions on the types of the arguments and results of a function, the simple notion of a function with a single argument and a single result is already sufficient to handle the case of multiple arguments and results, by packaging multiple values using lists or tuples. For example, we can define a function `add` that calculates the sum of a pair of integers, and a function `zeroto` that returns the list of integers from zero to a given limit, as follows:

```
add :: (Int,Int) -> Int
add (x,y) = x+y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

In these examples we have followed the Haskell convention of preceding function definitions by their types, which serves as useful documentation. Any such types provided manually by the user are checked for consistency with the types calculated automatically using type inference.

Note that there is no restriction that functions must be *total* on their argument type, in the sense that there may be some arguments for which the result is not defined. For example, the result of the library function `head` that selects the first element of a list is undefined if the list is empty:

```
> head []
*** Exception: Prelude.head: empty list
```

3.6 Curried functions

Functions with multiple arguments can also be handled in another, perhaps less obvious way, by exploiting the fact that functions are free to return functions as results. For example, consider the following definition:

```
add' :: Int -> (Int -> Int)
add' x y = x+y
```

The type states that `add'` is a function that takes an argument of type `Int`, and returns a result that is a function of type `Int -> Int`. The definition itself states that `add'` takes an integer `x` followed by an integer `y`, and returns the result `x+y`. More precisely, `add'` takes an integer `x` and returns a function, which in turn takes an integer `y` and returns the result `x+y`.

Note that the function `add'` produces the same final result as the function `add` from the previous section, but whereas `add` takes its two arguments at the same time packaged as a pair, `add'` takes its two

arguments one at a time, as reflected in the different types of the two functions:

```
add :: (Int,Int) -> Int

add' :: Int -> (Int -> Int)
```

Functions with more than two arguments can also be handled using the same technique, by returning functions that return functions, and so on. For example, a function `mult` that takes three integers `x`, `y` and `z`, one at a time, and returns their product, can be defined as follows:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

This definition states that `mult` takes an integer `x` and returns a function, which in turn takes an integer `y` and returns another function, which finally takes an integer `z` and returns the result `x*y*z`.

Functions such as `add'` and `mult` that take their arguments one at a time are called *curried functions*. As well as being interesting in their own right, curried functions are also more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function with less than its full complement of arguments. For example, a function that increments an integer can be given by the partial application `add' 1 :: Int -> Int` of the curried function `add'` with only one of its two arguments.

To avoid excess parentheses when working with curried functions, two simple conventions are adopted. First of all, the function arrow `->` in types is assumed to associate to the right. For example, the type

```
Int -> Int -> Int -> Int
```

means

```
Int -> (Int -> (Int -> Int))
```

Consequently, function application, which is denoted silently using spacing, is assumed to associate to the left. For example, the application

```
mult x y z
```

means

```
((mult x) y) z
```

Unless tupling is explicitly required, all functions in Haskell with multiple arguments are normally defined as curried functions, and the two conventions above are used to reduce the number of parentheses that are required. In chapter 4 we will see how the meaning of curried function definitions can be formalised in a simple manner using the notion of lambda expressions.

3.7 Polymorphic types

The library function `length` calculates the length of any list, irrespective of the type of the elements of the list. For example, it can be used to calculate the length of a list of integers, a list of strings, or even a list of functions:

```
> length [1,3,5,7]
4
```

```
> length ["Yes","No"]
2

> length [sin,cos,tan]
3
```

The idea that `length` can be applied to lists whose elements have any type is made precise in its type by the inclusion of a *type variable*. Type variables must begin with a lower-case letter, and are usually simply named `a`, `b`, `c`, and so on. For example, the type of `length` is as follows:

```
length :: [a] -> Int
```

That is, for any type `a`, the function `length` has type `[a] -> Int`. A type that contains one or more type variables is called *polymorphic* ("of many forms"), as is an expression with such a type. Hence, `[a] -> Int` is a polymorphic type and `length` is a polymorphic function. More generally, many of the functions provided in the standard prelude are polymorphic. For example:

```
fst :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

zip :: [a] -> [b] -> [(a,b)]

id :: a -> a
```

The type of a polymorphic function often gives a strong indication about the function's behaviour. For example, from the type `[a] -> [b] -> [(a,b)]` we can conclude that `zip` pairs up elements from two lists, although the type on its own doesn't capture the precise manner in which this is done.

3.8 Overloaded types

The arithmetic operator `+` calculates the sum of any two numbers of the same numeric type. For example, it can be used to calculate the sum of two integers, or the sum of two floating-point numbers:

```
> 1 + 2
3

> 1.0 + 2.0
3.0
```

The idea that `+` can be applied to numbers of any numeric type is made precise in its type by the inclusion of a *class constraint*. Class constraints are written in the form `C a`, where `C` is the name of a class and `a` is a type variable. For example, the type of the addition operator `+` is as follows:

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

That is, for any type `a` that is an *instance* of the class `Num` of numeric types, the function `(+)` has type `a -> a -> a`. (Parenthesising an operator converts it into a curried function, as we shall see in chapter 4.)

A type that contains one or more class constraints is called *overloaded*, as is an expression with such a

type. Hence, `Num a => a -> a -> a` is an overloaded type and `(+)` is an overloaded function. More generally, most of the numeric functions provided in the prelude are overloaded. For example:

```
(*) :: Num a => a -> a -> a

negate :: Num a => a -> a

abs :: Num a => a -> a
```

Numbers themselves are also overloaded. For example, `3 :: Num a => a` means that for any numeric type `a`, the value `3` has type `a`. In this manner, the value `3` could be an integer, a floating-point number, or more generally a value of any numeric type, depending on the context in which it is used.

3.9 Basic classes

Recall that a type is a collection of related values. Building upon this notion, a *class* is a collection of types that support certain overloaded operations called *methods*. Haskell provides a number of basic classes that are built-in to the language, of which the most commonly used are described below. (More advanced built-in classes are considered in part II of the book.)

`Eq` – equality types

This class contains types whose values can be compared for equality and inequality using the following two methods:

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

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

All the basic types `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double` are instances of the `Eq` class, as are list and tuple types, provided that their element and component types are instances. For example:

```
> False == False
True

> 'a' == 'b'
False

> "abc" == "abc"
True

> [1,2] == [1,2,3]
False

> ('a',False) == ('a',False)
True
```

Note that function types are not in general instances of the `Eq` class, because it is not feasible in general to compare two functions for equality.

`Ord` – ordered types

This class contains types that are instances of the equality class `Eq`, but in addition whose values are totally (linearly) ordered, and as such can be compared and processed using the following six methods:

```
(<) :: a -> a -> Bool

(<=) :: a -> a -> Bool

(>) :: a -> a -> Bool

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

min :: a -> a -> a

max :: a -> a -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double` are instances of the `Ord` class, as are list types and tuple types, provided that their element and component types are instances. For example:

```
> False < True
True

> min 'a' 'b'
'a'

> "elegant" < "elephant"
True

> [1,2,3] < [1,2]
False

> ('a',2) < ('b',1)
True

> ('a',2) < ('a',1)
False
```

Note that strings, lists and tuples are ordered *lexicographically*; that is, in the same way as words in a dictionary. For example, two pairs of the same type are in order if their first components are in order, in which case their second components are not considered, or if their first components are equal, in which case their second components must be in order.

`Show` – showable types

This class contains types whose values can be converted into strings of characters using the following method:

```
show :: a -> String
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double` are instances of the `Show` class, as are list types and tuple types, provided that their element and component types are instances. For

example:

```
> show False
"False"

> show 'a'
"'a'"

> show 123
"123"

> show [1,2,3]
"[1,2,3]"

> show ('a',False)
"('a',False)"
```

`Read` – readable types

This class is dual to `Show`, and contains types whose values can be converted from strings of characters using the following method:

```
read :: String -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double` are instances of the `Read` class, as are list types and tuple types, provided that their element and component types are instances. For example:

```
> read "False" :: Bool
False

> read "'a'" :: Char
'a'

> read "123" :: Int
123

> read "[1,2,3]" :: [Int]
[1,2,3]

> read "('a',False)" :: (Char,Bool)
('a',False)
```

The use of `::` in these examples resolves the type of the result, which would otherwise not be able to be inferred by GHCi. In practice, however, the necessary type information can usually be inferred automatically from the context. For example, the expression `not (read "False")` requires no explicit type information, because the application of the logical negation function `not` implies that `read "False"` must have type `Bool`.

Note that the result of `read` is undefined if its argument is not syntactically valid. For example, the expression `not (read "abc")` produces an error when evaluated, because `"abc"` cannot be read as a logical value:

```
> not (read "abc")
```

```
*** Exception: Prelude.read: no parse
```

`Num` – numeric types

This class contains types whose values are numeric, and as such can be processed using the following six methods:

```
(+) :: a -> a -> a

(-) :: a -> a -> a

(*) :: a -> a -> a

negate :: a -> a

abs :: a -> a signum :: a -> a
```

(The method `negate` returns the negation of a number, `abs` returns the absolute value, while `signum` returns the sign.) The basic types `Int`, `Integer`, `Float`, and `Double` are instances of the `Num` class. For example:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> negate 3.0
-3.0

> abs (-3)
3

> signum (-3)
-1
```

As illustrated above, negative numbers must be parenthesised when used as arguments to functions, to ensure the correct interpretation of the minus sign. For example, `abs -3` without parentheses means `abs - 3`, which is both the incorrect meaning here and an ill-typed expression.

Note that the `Num` class does not provide a division method, but as we shall now see, division is handled separately using two special classes, one for integral numbers and one for fractional numbers.

`Integral` – integral types

This class contains types that are instances of the numeric class `Num`, but in addition whose values are integers, and as such support the methods of integer division and integer remainder:

```
div :: a -> a -> a

mod :: a -> a -> a
```

In practice, these two methods are often written between their two arguments by enclosing their names in single back quotes. The basic types `Int` and `Integer` are instances of the `Integral` class. For example:

```
> 7 `div` 2
3

> 7 `mod` 2
1
```

For efficiency reasons, a number of prelude functions that involve both lists and integers (such as `take` and `drop`) are restricted to the type `Int` of finite-precision integers, rather than being applicable to any instance of the `Integral` class. If required, however, such generic versions of these functions are provided as part of an additional library file called `Data.List`.

`Fractional` – fractional types

This class contains types that are instances of the numeric class `Num`, but in addition whose values are non-integral, and as such support the methods of fractional division and fractional reciprocation:

```
(/) :: a -> a -> a

recip :: a -> a
```

The basic types `Float` and `Double` are instances. For example:

```
> 7.0 / 2.0
3.5

> recip 2.0
0.5
```

3.10 Chapter remarks

The term `Bool` for the type of logical values celebrates the pioneering work of George Boole on symbolic logic, while the term *curried* for functions that take their arguments one at a time celebrates the work of Haskell Curry (after whom the language Haskell itself is named) on such functions. The relationship between the type of a polymorphic function and its behaviour is formalised in [3]. A more detailed account of the type system is given in the Haskell Report [4], and a formal description of the type system can be found in [5].

3.11 Exercises

1. What are the types of the following values?

   ```
   ['a','b','c']

   ('a','b','c')
   ```

```
[(False,'O'),(True,'1')]

([False,True],['0','1'])

[tail, init, reverse]
```

2. Write down definitions that have the following types; it does not matter what the definitions actually do as long as they are type correct.

```
bools :: [Bool]

nums :: [[Int]]

add :: Int -> Int -> Int -> Int

copy :: a -> (a,a)

apply :: (a -> b) -> a -> b
```

3. What are the types of the following functions?

```
second xs = head (tail xs)

swap (x,y) = (y,x)

pair x y = (x,y)

double x = x*2

palindrome xs = reverse xs == xs

twice f x = f (f x)
```

   Hint: take care to include the necessary class constraints in the types if the functions are defined using overloaded operators.

4. Check your answers to the preceding three questions using GHCi.

5. Why is it not feasible in general for function types to be instances of the `Eq` class? When is it feasible? Hint: two functions of the same type are equal if they always return equal results for equal arguments.

Solutions to exercises 1 and 2 are given in appendix A.