chapter 1

Introduction

In this chapter we set the stage for the rest of the book. We start by reviewing the notion of a function, then introduce the concept of functional programming, summarise the main features of Haskell and its history, and conclude with two small examples that give a taste of Haskell.

1.1 Functions

In Haskell, a *function* is a mapping that takes one or more arguments and produces a single result, and is defined using an equation that gives a name for the function, a name for each of its arguments, and a body that specifies how the result can be calculated in terms of the arguments.

For example, a function $double$ that takes a number x as its argument, and produces the result $x + x$, can be defined by the following equation:

$$double\ x \quad = \quad x + x$$

When a function is applied to actual arguments, the result is obtained by substituting these arguments into the body of the function in place of the argument names. This process may immediately produce a result that cannot be further simplified, such as a number. More commonly, however, the result will be an expression containing other function applications, which must then be processed in the same way to produce the final result.

For example, the result of the application $double\ 3$ of the function $double$ to the number 3 can be determined by the following calculation, in which each step is explained by a short comment in curly parentheses:

$$\begin{array}{ll} & double\ 3 \\ = & \quad \{ \text{ applying } double \text{ } \} \\ & 3 + 3 \\ = & \quad \{ \text{ applying } + \text{ } \} \\ & 6 \end{array}$$

Similarly, the result of the nested application $double\ (double\ 2)$ in which the function $double$ is applied twice can be calculated as follows:

$$
\begin{array}{ll}
 & double\ (double\ 2) \\
= & \quad \{ \text{ applying the inner } double \ \} \\
 & double\ (2+2) \\
= & \quad \{ \text{ applying } + \ \} \\
 & double\ 4 \\
= & \quad \{ \text{ applying } double \ \} \\
 & 4+4 \\
= & \quad \{ \text{ applying } + \ \} \\
 & 8
\end{array}
$$

Alternatively, the same result could also be calculated by starting with the outer application of the function $double$ rather than the inner:

$$
\begin{array}{ll}
 & double\ (double\ 2) \\
= & \quad \{ \text{ applying the outer } double \ \} \\
 & double\ 2 + double\ 2 \\
= & \quad \{ \text{ applying the first } double \ \} \\
 & (2+2) + double\ 2 \\
= & \quad \{ \text{ applying the first } + \ \} \\
 & 4 + double\ 2 \\
= & \quad \{ \text{ applying } double \ \} \\
 & 4 + (2+2) \\
= & \quad \{ \text{ applying the second } + \ \} \\
 & 4+4 \\
= & \quad \{ \text{ applying } + \ \} \\
 & 8
\end{array}
$$

However, this calculation requires two more steps than our original version, because the expression $double\ 2$ is duplicated in the first step and hence simplified twice. In general, the order in which functions are applied in a calculation does not affect the value of the final result, but it may affect the number of steps required, and may affect whether the calculation process terminates. These issues are explored in more detail in chapter 12.

1.2 Functional programming

What is functional programming? Opinions differ, and it is difficult to give a precise definition. Generally speaking, however, functional programming can be viewed as a *style* of programming in which the basic method of computation is the application of functions to arguments. In turn, a functional programming language is one that *supports* and *encourages* the functional style.

To illustrate these ideas, let us consider the task of computing the sum of the integers (whole numbers) between one and some larger number n. In most current programming languages, this would normally be achieved using two variables that store values that can be changed over time, one such variable used to count up to n, and the other used to accumulate the total.

For example, if we use the assignment symbol := to change the value of a variable, and the keywords **repeat** and **until** to repeatedly execute a sequence of instructions until a condition is satisfied, then the following sequence of instructions computes the required sum:

```
count := 0
total := 0
repeat
    count := count + 1
    total := total + count
until
    count = n
```

That is, we first initialise both the counter and the total to zero, and then repeatedly increment the counter and add this value to the total until the counter reaches n, at which point the computation stops.

In the above program, the basic method of computation is changing stored values, in the sense that executing the program results in a sequence of assignments. For example, the case of $n = 5$ gives the following sequence, in which the final value assigned to the variable $total$ is the required sum:

```
count := 0
total := 0
count := 1
total := 1
count := 2
total := 3
count := 3
total := 6
count := 4
total := 10
count := 5
total := 15
```

In general, programming languages in which the basic method of computation is changing stored values are called *imperative* languages, because programs in such languages are constructed from imperative instructions that specify precisely how the computation should proceed.

Now let us consider computing the sum of the numbers between one and n using Haskell. This would normally be achieved using two library functions, one called [. .] used to produce the list of numbers between one and n, and the other called sum used to produce the sum of this list:

```
sum [1..n]
```

In this program, the basic method of computation is applying functions to arguments, in the sense that executing the program results in a sequence of applications. For example, the case of $n = 5$ gives the following sequence, in which the final result is the required sum:

```
  sum [1..5]
=       { applying [ . . ] }
```

$$
\begin{aligned}
& sum\ [1,2,3,4,5] \\
= & \quad \{ \text{applying } sum \} \\
& 1+2+3+4+5 \\
= & \quad \{ \text{applying } + \} \\
& 15
\end{aligned}
$$

Most imperative languages support some form of programming with functions, so the Haskell program $sum\ [1\,..\,n]$ could be translated into such languages. However, most imperative languages do not *encourage* programming in the functional style. For example, many languages discourage or prohibit functions from being stored in data structures such as lists, from constructing intermediate structures such as the list of numbers in the above example, from taking functions as arguments or producing functions as results, or from being defined in terms of themselves. In contrast, Haskell imposes no such restrictions on how functions can be used, and provides a range of features to make programming with functions both simple and powerful.

1.3 Features of Haskell

For reference, the main features of Haskell are listed below, along with the particular chapters of this book that give further details.

- **Concise programs** (chapters 2 and 4)
  Due to the high-level nature of the functional style, programs written in Haskell are often much more *concise* than in other languages, as illustrated by the example in the previous section. Moreover, the syntax of Haskell has been designed with concise programs in mind, in particular by having few keywords, and by allowing indentation to be used to indicate the structure of programs. Although it is difficult to make an objective comparison, Haskell programs are often between two and ten times shorter than programs written in other current languages.

- **Powerful type system** (chapters 3 and 10)
  Most modern programming languages include some form of *type system* to detect incompatibility errors, such as attempting to add a number and a character. Haskell has a type system that requires little type information from the programmer, but allows a large class of incompatibility errors in programs to be automatically detected prior to their execution, using a sophisticated process called type inference. The Haskell type system is also more powerful than most current languages, by allowing functions to be "polymorphic" and "overloaded".

- **List comprehensions** (chapter 5)
  One of the most common ways to structure and manipulate data in computing is using lists. To this end, Haskell provides lists as a basic concept in the language, together with a simple but powerful *comprehension* notation that constructs new lists by selecting and filtering elements from one or more existing lists. Using the comprehension notation allows many common

functions on lists to be defined in a clear and concise manner, without the need for explicit recursion.

- **Recursive functions** (chapter 6)
  Most non-trivial programs involve some form of repetition or looping. In Haskell, the basic mechanism by which looping is achieved is by using *recursive* functions that are defined in terms of themselves. Many computations have a simple and natural definition in terms of recursive functions, particularly when "pattern matching" and "guards" are used to separate different cases into different equations.

- **Higher-order functions** (chapter 7)
  Haskell is a *higher-order* functional language, which means that functions can freely take functions as arguments and produce functions as results. Using higher-order functions allows common programming patterns, such as composing two functions, to be defined as functions within the language itself. More generally, higher-order functions can be used to define "domain-specific languages" within Haskell, such as for list processing, parsing, and interactive programming.

- **Monadic effects** (chapters 8 and 9)
  Functions in Haskell are pure functions that take all their input as arguments and produce all their output as results. However, many programs require some form of *side effect* that would appear to be at odds with purity, such as reading input from the keyboard, or writing output to the screen, while the program is running. Haskell provides a uniform framework for handling effects without compromising the purity of functions, based upon the mathematical notion of a *monad*.

- **Lazy evaluation** (chapter 12)
  Haskell programs are executed using a technique called *lazy evaluation*, which is based upon the idea that no computation should be performed until its result is actually required. As well as avoiding unnecessary computation, lazy evaluation ensures that programs terminate whenever possible, encourages programming in a modular style using intermediate data structures, and even allows data structures with an infinite number of elements, such as an infinite list of numbers.

- **Reasoning about programs** (chapter 13)
  Because programs in Haskell are pure functions, simple *equational reasoning* can be used to execute programs, to transform programs, to prove properties of programs, and even to derive programs directly from specifications of their behaviour. Equational reasoning is particularly powerful when combined with the use of "induction" to reason about functions that are defined using recursion.

1.4 Historical background

Many of the features of Haskell are not new, but were first introduced by other languages. To help place Haskell in context, some of the main historical developments related to the language are briefly summarised below:

- In the 1930s, Alonzo Church developed the lambda calculus, a simple but powerful mathematical theory of functions.
- In the 1950s, John McCarthy developed Lisp ("LISt Processor"), generally regarded as being the first functional programming language. Lisp had some influences from the lambda calculus, but still adopted variable assignments as a central feature of the language.
- In the 1960s, Peter Landin developed ISWIM ("If you See What I Mean"), the first pure functional programming language, based strongly on the lambda calculus and having no variable assignments.
- In the 1970s, John Backus developed FP ("Functional Programming"), a functional programming language that particularly emphasised the idea of higher-order functions and reasoning about programs.
- Also in the 1970s, Robin Milner and others developed ML ("Meta-Language"), the first of the modern functional programming languages, which introduced the idea of polymorphic types and type inference.
- In the 1970s and 1980s, David Turner developed a number of lazy functional programming languages, culminating in the commercially produced language Miranda (meaning "admirable").
- In 1987, an international committee of researchers initiated the development of Haskell (named after the logician Haskell Curry), a standard lazy functional programming language.
- In 2003, the committee published the Haskell Report, which defines a long-awaited stable version of Haskell, and is the culmination of fifteen years of work on the language by its designers.

It is worthy of note that three of the above researchers — McCarthy, Backus, and Milner — have each received the ACM Turing Award, which is generally regarded as being the computing equivalent of a Nobel prize.

1.5 A taste of Haskell

We conclude this chapter with two small examples that give a taste of programming in Haskell. First of all, recall the function *sum* used earlier in this chapter, which produces the sum of a list of numbers. In Haskell, this function can be defined using the following two equations:

$$\begin{array}{lcl} sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

The first equation states that the sum of the empty list is zero, while the second states that the sum of any non-empty list comprising a first number x and a remaining list of numbers xs is given by adding x and the sum of xs. For example, the result of $sum\ [1, 2, 3]$ can be calculated as follows:

$$\begin{array}{ll} & sum\ [1, 2, 3] \\ = & \quad \{ \text{applying } sum \} \\ & 1 + sum\ [2, 3] \\ = & \quad \{ \text{applying } sum \} \\ & 1 + (2 + sum\ [3]) \\ = & \quad \{ \text{applying } sum \} \\ & 1 + (2 + (3 + sum\ [\,])) \\ = & \quad \{ \text{applying } sum \} \\ & 1 + (2 + (3 + 0)) \\ = & \quad \{ \text{applying } + \} \\ & 6 \end{array}$$

Note that even though the function sum is defined in terms of itself and is hence recursive, it does not loop forever. In particular, each application of sum reduces the length of the argument list by one, until the list eventually becomes empty, at which point the recursion stops. Returning zero as the sum of the empty list is appropriate because zero is the *identity* for addition. That is, $0 + x = x$ and $x + 0 = x$ for any number x.

In Haskell, every function has a *type* that specifies the nature of its arguments and results, which is automatically inferred from the definition of the function. For example, the function sum has the following type:

$$Num\ a \Rightarrow [a] \rightarrow a$$

This type states that for any type a of numbers, sum is a function that maps a list of such numbers to a single such number. Haskell supports many different types of numbers, including integers such as 123, and "floating-point" numbers such as 3.14159. Hence, for example, sum could be applied to a list of integers, as in the calculation above, or to a list of floating-point numbers.

Types provide useful information about the nature of functions, but, more importantly, their use allows many errors in programs to be automatically detected prior to executing the programs themselves. In particular, for every function application in a program, a check is made that the type of the actual arguments is compatible with the type of the function itself. For example, attempting to apply the function sum to a list of characters would be reported as an error, because characters are not a type of numbers.

Now let us consider a more interesting function concerning lists, which illustrates a number of other aspects of Haskell. Suppose that we define a function called $qsort$ by the following two equations:

```
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                 smaller = [a | a ← xs, a ≤ x]
                 larger  = [b | b ← xs, b > x]
```

In this definition, $+\!\!+$ is an operator that appends two lists; for example, $[1,2,3] +\!\!+ [4,5] = [1,2,3,4,5]$. In turn, **where** is a keyword that introduces local definitions, in this case a list *smaller* that consists of all elements a from the list xs that are less than or equal to x, together with a list *larger* that consists of all elements b from xs that are greater than x. For example, if $x = 3$ and $xs = [5,1,4,2]$, then $smaller = [1,2]$ and $larger = [5,4]$.

What does *qsort* actually do? First of all, we show that it has no effect on lists with a single element, in the sense that $qsort\ [x] = [x]$ for any x:

```
    qsort [x]
=       { applying qsort }
    qsort [] ++ [x] ++ qsort []
=       { applying qsort }
    [] ++ [x] ++ []
=       { applying ++ }
    [x]
```

In turn, we now work through the application of *qsort* to an example list, using the above property to simplify the calculation:

```
    qsort [3,5,1,4,2]
=       { applying qsort }
    qsort [1,2] ++ [3] ++ qsort [5,4]
=       { applying qsort }
    (qsort [] ++ [1] ++ qsort [2]) ++ [3]
    ++ (qsort [4] ++ [5] ++ qsort [])
=       { applying qsort, above property }
    ([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
=       { applying ++ }
    [1,2] ++ [3] ++ [4,5]
=       { applying ++ }
    [1,2,3,4,5]
```

In summary, *qsort* has sorted the example list into numerical order. More generally, this function produces a sorted version of any list of numbers. The first equation for *qsort* states that the empty list is already sorted, while the second states that any non-empty list can be sorted by inserting the first number between the two lists that result from sorting the remaining numbers that are *smaller* and *larger* than this number. This method of sorting is called *quicksort*, and is one of the best such methods known.

The above implementation of quicksort is an excellent example of the power of Haskell, being both clear and concise. Moreover, the function *qsort* is also more general than might be expected, being applicable not just with numbers, but with any type of ordered values. More precisely, the type

$$qsort \;::\; Ord\ a \Rightarrow [a] \rightarrow [a]$$

states that, for any type a of ordered values, $qsort$ is a function that maps between lists of such values. Haskell supports many different types of ordered values, including numbers, single characters such as `'a'`, and strings of characters such as `"abcde"`. Hence, for example, the function $qsort$ could also be used to sort a list of characters, or a list of strings.

1.6 Chapter remarks

The Haskell Report is freely available on the web from the Haskell home page, *www.haskell.org*, and has also been published as a book (25). A more detailed historical account of the development of functional programming languages is given in Hudak's survey article (11).

1.7 Exercises

1. Give another possible calculation for the result of $double\ (double\ 2)$.

2. Show that $sum\ [x] = x$ for any number x.

3. Define a function $product$ that produces the product of a list of numbers, and show using your definition that $product\ [2, 3, 4] = 24$.

4. How should the definition of the function $qsort$ be modified so that it produces a *reverse* sorted version of a list?

5. What would be the effect of replacing $\leq$ by $<$ in the definition of $qsort$? Hint: consider the example $qsort\ [2, 2, 3, 1, 1]$.

chapter 2

First steps

In this chapter we take our first proper steps with Haskell. We start by introducing the Hugs system and the standard prelude, then explain the notation for function application, develop our first Haskell script, and conclude by discussing a number of syntactic conventions concerning scripts.

2.1 The Hugs system

As we saw in the previous chapter, small Haskell programs can be executed by hand. In practice, however, we usually require a system that can execute programs automatically. In this book we use an interactive system called *Hugs*, which is the most widely used implementation of Haskell.

The interactive nature of Hugs makes it well suited for teaching and prototyping, and its performance is sufficient for most applications. However, if greater performance or a stand-alone executable version of a program is required, a number of compilers for Haskell are also available, of which the most widely used is the Glasgow Haskell Compiler. This compiler also has an interactive version that operates in a similar manner to Hugs, and can readily be used in its place for the purposes of this book.

2.2 The standard prelude

When the Hugs system is started it first loads a library file called $Prelude.hs$, and then displays a $>$ prompt to indicate that the system is waiting for the user to enter an expression to be evaluated. For example, the library file defines many familiar functions that operate on integers, including the five main arithmetic operations of addition, subtraction, multiplication, division, and exponentiation, as illustrated below:

```
> 2 + 3
5
```