CAMBRIDGE

Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information

# PART ONE

# BASICS

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information



What makes a good algorithm? There are as many answers to this question as there are to the question of what makes a good cookbook recipe. Is the recipe clear and easy to follow? Does the recipe use standard and well-understood techniques? Does it use widely available ingredients? Is the preparation time reasonably short? Does it involve many pots and pans and a lot of kitchen space? And so on and so on. Some people when asked this question say that what is most important about a recipe is whether the dish is attractive or not, a point we will try to bear in mind when expressing our functional algorithms.

In the first three chapters we review the ingredients we need for designing good recipes for attractive algorithms in a functional kitchen, and describe the tools we need for analysing their efficiency. Our functional language of choice is Haskell, and the ingredients are Haskell functions. These ingredients and the techniques for combining them are reviewed in the first chapter. Be aware that the chapter is *not* an introduction to Haskell; its main purpose is to outline what should be familiar territory to the reader, or at least territory that the reader should feel comfortable travelling in.

The second chapter concerns efficiency, specifically the running time of algorithms. We will ignore completely the question of space efficiency, for the plain fact of the matter is that executing a functional program can take up quite a lot of kitchen space. There are methods for controlling the space used in evaluating a functional expression, but we refer the reader to other books for their elaboration. That chapter reviews asymptotic notation for stating running times, and explores how recurrence relations, which are essentially recursive functions for determining the running times of recursive functions, can be solved to give asymptotic estimates. The chapter also introduces, albeit fairly briefly, the notion of amortised running times because it will be needed later in the book.

The final chapter in this part introduces a small number of basic data structures that will be needed at one or two places in the rest of the book. These are symmetric lists, random-access lists, and purely functional arrays. Mostly we postpone discussion of any data structure required to make an algorithm efficient until the algorithm itself is introduced, but these three form a coherent group that can be discussed without having specific applications in mind.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information

# Chapter 1

# Functional programming

Haskell is a large and powerful language, brimming with clever ideas about how to structure programs and possessing many bells and whistles. But in this book we will use only a small subset of the host of available features. So, no Monads, no Applicatives, no Foldables, and no Traversables. In this chapter we will spell out what we do need to construct effective algorithms. Some of the material will be revisited when particular problems are put under the microscope, so you should regard the chapter primarily as a way to check your understanding of the basic ideas of Haskell.

## 1.1 Basic types and functions

We will use only simple types, such as Booleans, characters, strings, numbers of various kinds, and lists. Most of the functions we use can be found in Haskell's Standard Prelude (the *Prelude* library), or in the library *Data.List*. Be warned that the definitions we give of some of these functions may not be exactly the definitions given in these libraries: the library definitions are tuned for optimal performance and ours for clarity. We will use type synonyms to improve readability, and **data** declarations of new types, especially trees of various kinds. When necessary we make use of simple type classes such as *Eq*, *Ord*, and *Num*, but we will not introduce new ones. Haskell provides many kinds of number, including two kinds of integer, *Int* and *Integer*, and two kinds of floating-point number, *Float* and *Double*. Elements of *Int* are restricted in range, usually $[-2^{63}, 2^{63})$ on 64-bit computers, though Haskell compilers are only required to cover the range $[-2^{29}, 2^{29})$. Elements of *Integer* are unrestricted. We will rarely use the floating-point numbers provided by *Float* and *Double*. In one or two places we will use *Rational* arithmetic, where a *Rational* number is the ratio of two *Integer* values. Haskell does not have a type of natural

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information



numbers,[1] though the library *Numeric.Natural* does provide arbitrary-precision ones. Instead, we will sometimes use the type synonym

$$\textbf{type}\ Nat = Int$$

Haskell cannot enforce the constraint that elements of *Nat* be natural numbers, and we use the synonym purely to document intention. For example, we can assert that $length :: [a] \to Nat$ because the length of a list, as defined in the Prelude, is a nonnegative element of *Int*. Haskell also provides unsigned numbers in the *Data.Word* library. Elements of *Word* are unsigned numbers and can represent natural numbers $n$ in the range $0 \leqslant n < 2^{64}$ on 64-bit machines. However, defining **type** $Nat = Word$ would be inconvenient simply because we could not then assert that $length :: [a] \to Nat$.

Most important for our purposes are the basic functions that manipulate lists. Of these the most useful are *map*, *filter*, and folds of various kinds. Here is the definition of *map*:

$$\begin{aligned} &map :: (a \to b) \to [a] \to [b] \\ &map\ f\ [\,] \quad\ = [\,] \\ &map\ f\ (x : xs) = f\ x : map\ f\ xs \end{aligned}$$

The function *map* applies its first argument, a function, to every element of its second argument, a list. The function *filter* is defined as follows:

$$\begin{aligned} &filter :: (a \to Bool) \to [a] \to [a] \\ &filter\ p\ [\,] \quad\ = [\,] \\ &filter\ p\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ x : filter\ p\ xs\ \textbf{else}\ filter\ p\ xs \end{aligned}$$

The function *filter* filters a list, retaining only those elements that satisfy the given test. There are various fold functions on lists, most of which will be explained in due course. Two of the important ones are *foldr* and *foldl*. The former is defined as follows:

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ f\ e\ [\,] \quad\ = e \\ &foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs) \end{aligned}$$

The function *foldr* folds a list from right to left, starting with a value $e$ and using a binary operator $\oplus$ to reduce the list to a single value. For example,

$$foldr\ (\oplus)\ e\ [x, y, z] = x \oplus (y \oplus (z \oplus e))$$

In particular, $foldr\ (:)\ [\,]\ xs = xs$ for *all* lists $xs$, including infinite lists. However, we will not make much use of infinite lists in what follows, except for idioms such as

[1] In the documentation for the GHC libraries, there is the statement "It would be very natural to add a type *Natural* providing an unbounded size unsigned integer, just as *Prelude.Integer* provides unbounded size signed integers. We do not do that yet since there is no demand for it." Maybe this book will create such a demand.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information



$$
\begin{aligned}
&label :: [a] \to [(Nat, a)] \\
&label\ xs = zip\ [0..]\ xs
\end{aligned}
$$

As another example, we can write

$$
\begin{aligned}
&length :: [a] \to Nat \\
&length = foldr\ succ\ 0\ \ \textbf{where}\ succ\ x\ n = n + 1
\end{aligned}
$$

The second main function, *foldl*, folds a list from left to right:

$$
\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b \\
&foldl\ f\ e\ [\,] \qquad = e \\
&foldl\ f\ e\ (x : xs) = foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$

Thus

$$foldl\ (\oplus)\ e\ [x, y, z] = ((e \oplus x) \oplus y) \oplus z$$

For example, we could also write

$$
\begin{aligned}
&length :: [a] \to Nat \\
&length = foldl\ succ\ 0\ \ \textbf{where}\ succ\ n\ x = n + 1
\end{aligned}
$$

Note that *foldl* returns a well-defined value only on finite lists; evaluation of *foldl* on an infinite list will never terminate. There is an alternative definition of *foldl*, namely

$$foldl\ f\ e = foldr\ (flip\ f)\ e \cdot reverse$$

where *flip* is a useful prelude function defined by

$$
\begin{aligned}
&flip :: (a \to b \to c) \to b \to a \to c \\
&flip\ f\ x\ y = f\ y\ x
\end{aligned}
$$

Since one can reverse a list in linear time, this definition is asymptotically as fast as the former. However, it involves two traversals of the input, one to reverse it and the second to fold it.

## 1.2 Processing lists

The difference between *foldr* and *foldl* prompts a general observation. When a programmer brought up in the imperative programming tradition meets functional programming for the first time, they are likely to feel that many computations seem to be carried out in the wrong order. Recursion has been described as the curious process of reaching one's goal by walking backwards towards it. Specifically, lists often seem to be processed from right to left when the natural way surely appears to be from left to right. Appeals to naturalness are often suspicious, and appearances can be deceptive. We normally read an English sentence from left to right, but when we encounter a phrase such as "a lovely little old French silver butter knife" the adjectives have to be applied from right to left. If the knife was made of French

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information



silver, but not necessarily made in France, we have to write "a lovely little old French-silver butter knife" to avoid ambiguity. Mathematical expressions too are usually understood from right to left, certainly those involving a chain of functional compositions. As to deceptiveness, the definition

$$head = foldr\ (\ll)\ \bot\ \ \textbf{where}\ x \ll y = x$$

though a little strange is certainly correct and takes constant time. The evaluation of $foldr\ (\ll)$, conceptually from right to left, is abandoned after the first element is encountered. Thus

$$\begin{aligned} head\ (x:xs) &= foldr\ (\ll)\ \bot\ (x:xs) \\ &= x \ll foldr\ (\ll)\ \bot\ xs \\ &= x \end{aligned}$$

The last step follows from the fact that Haskell is a *lazy* language in which evaluations are performed only when needed, so evaluation of $\ll$ does not require evaluation of its second argument.

Sometimes the direction of travel *is* important. For example, consider the following two definitions of *concat*:

$$\begin{aligned} &concat_1, concat_2 :: [[a]] \to [a] \\ &concat_1 = foldr\ (+\!\!+)\ [\,] \\ &concat_2 = foldl\ (+\!\!+)\ [\,] \end{aligned}$$

We have $concat_1\ xss = concat_2\ xss$ for all finite lists $xss$ (see Exercise 1.10), but which definition is better? We will look at the precise running times of the two functions in the following chapter, but here is one way to view the problem. Imagine a long table on which there are a number of piles of documents. You have to assemble these documents into one big pile ensuring that the correct order is maintained, so the second pile (numbering from left to right) has to go under the first pile, the third pile under the second pile, and so on. You could start from left to right, picking up the first pile, putting it on top of the second pile, picking the combined pile up and putting it on top of the third pile, and so on. Or you could start at the other end, placing the penultimate pile on the last pile, the antepenultimate pile on top of that, and so on (even English words are direction-biased: the words 'first', 'second', and 'third' are simple, but 'penultimate' and 'antepenultimate' are not). The left to right solution involves some heavy lifting, particularly at the last step when a big pile of documents has to be lifted up and placed on the last pile, but the right to left solution involves picking up only one pile at each step. So $concat_1$ is potentially a much more efficient way to concatenate a list of lists than $concat_2$.

Here is another example. Consider the problem of breaking a list of words into a list of lines, ensuring that the width of each line is at most some given bound. This problem is known as the *paragraph problem*, and there is a section devoted

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information



to it in Chapter 12. It seems natural to process the input from left to right, adding successive words to the end of the current line until no more words will fit, in which case a new line is started. This particular algorithm is a greedy one. There are also non-greedy algorithms for the paragraph problem that process words from right to left. Part Three of the book is devoted to the study of greedy algorithms. Nevertheless, these two examples apart, the direction of travel is often unimportant.

The direction of travel is also related to another concept in algorithm design, the notion of an *online* algorithm. An online algorithm is one that processes a list without having the entire list available from the start. Instead, the list is regarded as a potentially infinite *stream* of values. Consequently, any online algorithm for solving a problem for a given stream also has to solve the problem for every prefix of the stream. And that means the stream has to be processed from left to right. In contrast, an *offline* algorithm is one that is given the complete list to start with, and can process the list in any order it wants. Online algorithms can usually be defined in terms of another basic Haskell function *scanl*, whose definition is as follows:

$$\begin{aligned} &scanl :: (b \to a \to b) \to b \to [a] \to [b] \\ &scanl\ f\ e\ [\,] \quad\ = [e] \\ &scanl\ f\ e\ (x : xs) = e : scanl\ f\ (f\ e\ x)\ xs \end{aligned}$$

For example,

$$scanl\ (\oplus)\ e\ [x, y, z, \ldots] = [e, e \oplus x, (e \oplus x) \oplus y, ((e \oplus x) \oplus y) \oplus z, \ldots]$$

In particular, *scanl* can be applied to an infinite list, producing an infinite list as result.

## 1.3 Inductive and recursive definitions

While most functions make use of recursion, the nature of the recursion is different in different functions. The functions *map*, *filter*, and *foldr* all make use of *structural* recursion. That is, the recursion follows the structure of lists built from the empty list $[\,]$ and the cons constructor $(:)$. There is one clause for the empty list and another, recursive clause for $x : xs$ in terms of the value of the function for $xs$. We will call such definitions *inductive* definitions. Most inductive definitions can be expressed as instances of *foldr*. For example, both *map* and *filter* can be so expressed (see the exercises).

Here is another example, an inductive definition of the function *perms* that returns a list of all the permutations of a list (we call it $perms_1$ because later on we will meet another definition, $perms_2$):

$$\begin{aligned} &perms_1\ [\,] \quad\ = [[\,]] \\ &perms_1\ (x : xs) = [zs \mid ys \leftarrow perms_1\ xs, zs \leftarrow inserts\ x\ ys] \end{aligned}$$

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE
Cambridge University Press
978-1-108-49161-7 — Algorithm Design with Haskell
Richard Bird , Jeremy Gibbons
Excerpt
More Information



The permutations of a nonempty list are obtained by taking each permutation of the tail of the list and returning all the ways the first element can be inserted. The function *inserts* is defined by

$$\begin{aligned}&inserts :: a \to [a] \to [[a]]\\&inserts\ x\ [\,] \quad\ = [[x]]\\&inserts\ x\ (y : ys) = (x : y : ys) : map\ (y:)\ (inserts\ x\ ys)\end{aligned}$$

For example,

$$inserts\ 1\ [2,3] = [[1,2,3],[2,1,3],[2,3,1]]$$

The definition of $perms_1$ uses explicit recursion and a list comprehension, but another way is to use a *foldr*:

$$perms_1 = foldr\ step\ [[\,]]\ \textbf{where}\ step\ x\ xss = concatMap\ (inserts\ x)\ xss$$

The useful function *concatMap* is defined by

$$\begin{aligned}&concatMap :: (a \to [b]) \to [a] \to [b]\\&concatMap\ f = concat \cdot map\ f\end{aligned}$$

Observe that since

$$step\ x\ xss = (concatMap \cdot inserts)\ x\ xss$$

the definition of $perms_1$ can be expressed even more briefly as

$$perms_1 = foldr\ (concatMap \cdot inserts)\ [[\,]]$$

The idiom $foldr\ (concatMap \cdot steps)\ e$ will be used frequently in later chapters for various definitions of *steps* and $e$, so keep the abbreviation in mind.

Here is another way of generating permutations, one that is recursive rather than inductive:

$$\begin{aligned}&perms_2\ [\,] = [[\,]]\\&perms_2\ xs = [x : zs \mid (x, ys) \leftarrow picks\ xs, zs \leftarrow perms_2\ ys]\\&picks :: [a] \to [(a,[a])]\\&picks\ [\,] \quad\ = [\,]\\&picks\ (x : xs) = (x,xs) : [(y, x : ys) \mid (y,ys) \leftarrow picks\ xs]\end{aligned}$$

The function *picks* picks an arbitrary element from a list in all possible ways, returning both the element and what remains. The function $perms_2$ computes a permutation by picking an arbitrary element of a nonempty list as a first element, and following it with a permutation of the rest of the list.

The function $perms_2$ uses a list comprehension, but an equivalent way is to write

$$\begin{aligned}&perms_2\ [\,] = [[\,]]\\&perms_2\ xs = concatMap\ subperms\ (picks\ xs)\\&\qquad\qquad \textbf{where}\ subperms\ (x,ys) = map\ (x:)\ (perms_2\ ys)\end{aligned}$$

© in this web service Cambridge University Press www.cambridge.org