chapter 6

Recursive functions

In this chapter we introduce recursion, the basic mechanism for looping in Haskell. We start with recursion on integers, then extend the idea to recursion on lists, consider multiple arguments, multiple recursion, and mutual recursion, and conclude with some advice on defining recursive functions.

6.1 Basic concepts

As we have seen in previous chapters, many functions can naturally be defined in terms of other functions. For example, a function that returns the *factorial* of a non-negative integer can be defined by using library functions to calculate the product of the integers between one and the number itself:

$$\begin{aligned} &\mathit{factorial} &&:: \mathit{Int} \rightarrow \mathit{Int} \\ &\mathit{factorial}\ n &&= \mathit{product}\ [1\,..\,n] \end{aligned}$$

In Haskell, it is also permissible to define functions in terms of themselves, in which case the functions are called *recursive*. For example, the *factorial* function can be defined in this manner as follows:

$$\begin{aligned} &\mathit{factorial}\ 0 &&= 1 \\ &\mathit{factorial}\ (n+1) &&= (n+1) * \mathit{factorial}\ n \end{aligned}$$

The first equation states that the factorial of zero is one, and is called a *base case*. The second equation states that the factorial of any strictly positive integer is the product of that number and the factorial of its predecessor, and is called a *recursive case*. For example, the following calculation shows how the factorial of three is computed using this definition:

$$\begin{aligned} &\quad \mathit{factorial}\ 3 \\ = &\quad \{\text{ applying } \mathit{factorial}\ \} \\ &\quad 3 * \mathit{factorial}\ 2 \\ = &\quad \{\text{ applying } \mathit{factorial}\ \} \\ &\quad 3 * (2 * \mathit{factorial}\ 1) \\ = &\quad \{\text{ applying } \mathit{factorial}\ \} \end{aligned}$$

$$\begin{aligned}
& 3 * (2 * (1 * \mathit{factorial}\ 0)) \\
= \quad & \{ \text{applying } \mathit{factorial} \} \\
& 3 * (2 * (1 * 1)) \\
= \quad & \{ \text{applying } * \} \\
& 6
\end{aligned}$$

Note that even though the *factorial* function is defined in terms of itself, it does not loop forever. In particular, each application of *factorial* reduces the integer argument by one, until it eventually reaches zero at which point the recursion stops and the multiplications are performed. Returning one as the factorial of zero is appropriate because one is the identity for multiplication. That is, $1 * x = x$ and $x * 1 = x$ for any integer x.

For the case of the *factorial* function, the original definition using library functions is simpler than the definition using recursion. However, as we shall see in the remainder of this book, many functions have a simple and natural definition using recursion. For example, many of the library functions in Haskell are defined in this way. Moreover, as we shall see in chapter 13, defining functions using recursion also allows properties of those functions to be proved using the powerful technique of induction.

As another example of recursion on integers, consider the multiplication operator $*$ used above. For efficiency reasons, this operator is provided as a primitive in Haskell. However, for non-negative integers it can also be defined using recursion on either of its two arguments, such as the second:

$$\begin{aligned}
(*) \quad & :: \quad \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
m * 0 \quad & = \quad 0 \\
m * (n + 1) \quad & = \quad m + (m * n)
\end{aligned}$$

For example:

$$\begin{aligned}
& 4 * 3 \\
= \quad & \{ \text{applying } * \} \\
& 4 + (4 * 2) \\
= \quad & \{ \text{applying } * \} \\
& 4 + (4 + (4 * 1)) \\
= \quad & \{ \text{applying } * \} \\
& 4 + (4 + (4 + (4 * 0))) \\
= \quad & \{ \text{applying } * \} \\
& 4 + (4 + (4 + 0)) \\
= \quad & \{ \text{applying } + \} \\
& 12
\end{aligned}$$

That is, the recursive definition for the $*$ operator formalises the idea that multiplication can be reduced to repeated addition.

6.2 Recursion on lists

Recursion is not restricted to functions on integers, but can also be used to define functions on lists. For example, the library function *product* used in the preceding section can be defined as follows:

$$\begin{array}{lcl} \mathit{product} & :: & \mathit{Num}\ a \Rightarrow [a] \rightarrow a \\ \mathit{product}\ [\,] & = & 1 \\ \mathit{product}\ (n : ns) & = & n * \mathit{product}\ ns \end{array}$$

The first equation states that the product of the empty list is one, which is appropriate because one is the identity for multiplication. The second equation states that the product of any non-empty list is given by multiplying the first number and the product of the remaining list of numbers. For example:

$$\begin{array}{cl} & \mathit{product}\ [2,3,4] \\ = & \quad \{\text{ applying } \mathit{product} \} \\ & 2 * \mathit{product}\ [3,4] \\ = & \quad \{\text{ applying } \mathit{product} \} \\ & 2 * (3 * \mathit{product}\ [4]) \\ = & \quad \{\text{ applying } \mathit{product} \} \\ & 2 * (3 * (4 * \mathit{product}\ [\,])) \\ = & \quad \{\text{ applying } \mathit{product} \} \\ & 2 * (3 * (4 * 1)) \\ = & \quad \{\text{ applying } * \} \\ & 24 \end{array}$$

Recall that lists in Haskell are actually constructed one element at a time using the cons operator. Hence, $[2, 3, 4]$ is just an abbreviation for $2 : (3 : (4 : [\,]))$. As another simple example of recursion on lists, the library function *length* can be defined using the same pattern of recursion as *product*:

$$\begin{array}{lcl} \mathit{length} & :: & [a] \rightarrow \mathit{Int} \\ \mathit{length}\ [\,] & = & 0 \\ \mathit{length}\ (\_ : xs) & = & 1 + \mathit{length}\ xs \end{array}$$

That is, the length of the empty list is zero, and the length of any non-empty list is the successor of the length of its tail. Note the use of the wildcard pattern _ in the recursive case, which reflects the fact that the length of a list does not depend upon the value of its elements.

Now let us consider the library function that reverses a list. This function can be defined using recursion as follows:

$$\begin{array}{lcl} \mathit{reverse} & :: & [a] \rightarrow [a] \\ \mathit{reverse}\ [\,] & = & [\,] \\ \mathit{reverse}\ (x : xs) & = & \mathit{reverse}\ xs +\!\!+ [x] \end{array}$$

That is, the reverse of the empty list is simply the empty list, and the reverse of any non-empty list is given by appending the reverse of its tail to a singleton list comprising the head of the list. For example:

$$\begin{array}{cl} & \mathit{reverse}\ [1,2,3] \\ = & \quad \{\text{ applying } \mathit{reverse} \} \\ & \mathit{reverse}\ [2,3] +\!\!+ [1] \\ = & \quad \{\text{ applying } \mathit{reverse} \} \\ & (\mathit{reverse}\ [3] +\!\!+ [2]) +\!\!+ [1] \\ = & \quad \{\text{ applying } \mathit{reverse} \} \\ & ((\mathit{reverse}\ [\,] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \end{array}$$

$$\begin{aligned} &= \quad \{\text{ applying } \mathit{reverse} \ \} \\ &\quad (([\,] +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1] \\ &= \quad \{\text{ applying } +\!\!+ \ \} \\ &\quad [3, 2, 1] \end{aligned}$$

In turn, the append operator ++ used in the above definition of *reverse* can itself be defined using recursion on its first argument:

$$\begin{array}{lcl} (+\!\!+) & :: & [a] \to [a] \to [a] \\ [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \end{array}$$

For example:

$$\begin{aligned} &\quad [1, 2, 3] +\!\!+ [4, 5] \\ &= \quad \{\text{ applying } +\!\!+ \ \} \\ &\quad 1 : ([2, 3] +\!\!+ [4, 5]) \\ &= \quad \{\text{ applying } +\!\!+ \ \} \\ &\quad 1 : (2 : ([3] +\!\!+ [4, 5])) \\ &= \quad \{\text{ applying } +\!\!+ \ \} \\ &\quad 1 : (2 : (3 : ([\,] +\!\!+ [4, 5]))) \\ &= \quad \{\text{ applying } +\!\!+ \ \} \\ &\quad 1 : (2 : (3 : [4, 5])) \\ &= \quad \{\text{ list notation } \} \\ &\quad [1, 2, 3, 4, 5] \end{aligned}$$

That is, the recursive definition for ++ formalises the idea that two lists can be appended by copying elements from the first list until it is exhausted, at which point the second list is joined on at the end.

We conclude this section with two examples of recursion on sorted lists. First of all, a function that inserts a new element of any ordered type into a sorted list to give another sorted list can be defined as follows:

$$\begin{array}{lcl} \mathit{insert} & :: & \mathit{Ord}\ a \Rightarrow a \to [a] \to [a] \\ \mathit{insert}\ x\ [\,] & = & [x] \\ \mathit{insert}\ x\ (y : ys) \mid x \leq y & = & x : y : ys \\ \qquad\qquad\qquad\quad \mid \mathit{otherwise} & = & y : \mathit{insert}\ x\ ys \end{array}$$

That is, inserting a new element into the empty list gives a singleton list, while for a non-empty list the result depends upon the ordering of the new element x and the head of the list y. In particular, if $x \leq y$, then the new element x is simply prepended to the start of the list, otherwise the head y becomes the first element of the resulting list, and we then proceed to insert the new element into the tail of the given list. For example:

$$\begin{aligned} &\quad \mathit{insert}\ 3\ [1, 2, 4, 5] \\ &= \quad \{\text{ applying } \mathit{insert}\ \} \\ &\quad 1 : \mathit{insert}\ 3\ [2, 4, 5] \\ &= \quad \{\text{ applying } \mathit{insert}\ \} \\ &\quad 1 : 2 : \mathit{insert}\ 3\ [4, 5] \\ &= \quad \{\text{ applying } \mathit{insert}\ \} \\ &\quad 1 : 2 : 3 : [4, 5] \end{aligned}$$

$$\begin{array}{ll} = & \{\text{ list notation }\} \\ & [1,2,3,4,5] \end{array}$$

Using *insert* we can now define a function that implements *insertion sort*, in which the empty list is already sorted, and any non-empty list is sorted by inserting its head into the list that results from sorting its tail:

$$\begin{array}{lcl} \mathit{isort} & :: & \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \\ \mathit{isort}\ [\,] & = & [\,] \\ \mathit{isort}\ (x : xs) & = & \mathit{insert}\ x\ (\mathit{isort}\ xs) \end{array}$$

For example:

$$\begin{array}{ll} & \mathit{isort}\ [3,2,1,4] \\ = & \{\text{ applying } \mathit{isort} \ \} \\ & \mathit{insert}\ 3\ (\mathit{insert}\ 2\ (\mathit{insert}\ 1\ (\mathit{insert}\ 4\ [\,]))) \\ = & \{\text{ applying } \mathit{insert} \ \} \\ & \mathit{insert}\ 3\ (\mathit{insert}\ 2\ (\mathit{insert}\ 1\ [4])) \\ = & \{\text{ applying } \mathit{insert} \ \} \\ & \mathit{insert}\ 3\ (\mathit{insert}\ 2\ [1,4]) \\ = & \{\text{ applying } \mathit{insert} \ \} \\ & \mathit{insert}\ 3\ [1,2,4] \\ = & \{\text{ applying } \mathit{insert} \ \} \\ & [1,2,3,4] \end{array}$$

6.3 Multiple arguments

Functions with multiple arguments can also be defined using recursion on more than one argument at the same time. For example, the library function *zip* that takes two lists and produces a list of pairs is defined as follows:

$$\begin{array}{lcl} \mathit{zip} & :: & [a] \rightarrow [b] \rightarrow [(a,b)] \\ \mathit{zip}\ [\,]\ \_ & = & [\,] \\ \mathit{zip}\ \_\ [\,] & = & [\,] \\ \mathit{zip}\ (x : xs)\ (y : ys) & = & (x,y) : \mathit{zip}\ xs\ ys \end{array}$$

For example:

$$\begin{array}{ll} & \mathit{zip}\ [\texttt{'a'},\texttt{'b'},\texttt{'c'}]\ [1,2,3,4] \\ = & \{\text{ applying } \mathit{zip} \ \} \\ & (\texttt{'a'},1) : \mathit{zip}\ [\texttt{'b'},\texttt{'c'}]\ [2,3,4] \\ = & \{\text{ applying } \mathit{zip} \ \} \\ & (\texttt{'a'},1) : (\texttt{'b'},2) : \mathit{zip}\ [\texttt{'c'}]\ [3,4] \\ = & \{\text{ applying } \mathit{zip} \ \} \\ & (\texttt{'a'},1) : (\texttt{'b'},2) : (\texttt{'c'},3) : \mathit{zip}\ [\,]\ [4] \\ = & \{\text{ applying } \mathit{zip} \ \} \\ & (\texttt{'a'},1) : (\texttt{'b'},2) : (\texttt{'c'},3) : [\,] \\ = & \{\text{ list notation }\} \\ & [(\texttt{'a'},1),(\texttt{'b'},2),(\texttt{'c'},3)] \end{array}$$

Note that two base cases are required in the definition of *zip*, because either of the two argument lists may be empty. As another example of recursion on multiple arguments, the library function *drop* that removes a given number of elements from the start of a list is defined as follows:

$$\begin{array}{lcl} drop & :: & Int \rightarrow [\,a\,] \rightarrow [\,a\,] \\ drop\ 0\ xs & = & xs \\ drop\ (n+1)\ [\,] & = & [\,] \\ drop\ (n+1)\ (\_ : xs) & = & drop\ n\ xs \end{array}$$

Again, two base cases are required, one for removing zero elements, and one for attempting to remove one or more elements from the empty list.

6.4 Multiple recursion

Functions can also be defined using *multiple recursion*, in which a function is applied more than once in its own definition. For example, recall the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, …, in which the first two numbers are 0 and 1, and each subsequent number is given by adding the preceding two numbers in the sequence. In Haskell, a function that calculates the nth Fibonacci number for any integer $n \geq 0$ can be defined using double recursion as follows:

$$\begin{array}{lcl} fibonacci & :: & Int \rightarrow Int \\ fibonacci\ 0 & = & 0 \\ fibonacci\ 1 & = & 1 \\ fibonacci\ (n+2) & = & fibonacci\ n + fibonacci\ (n+1) \end{array}$$

As another example, in chapter 1 we showed how to implement another well-known method of sorting a list, called quicksort:

$$\begin{array}{lcl} qsort & :: & Ord\ a \Rightarrow [\,a\,] \rightarrow [\,a\,] \\ qsort\ [\,] & = & [\,] \\ qsort\ (x : xs) & = & qsort\ smaller +\!\!+ [\,x\,] +\!\!+ qsort\ larger \\ & & \quad \mathbf{where} \\ & & \qquad smaller = [\,a \mid a \leftarrow xs, a \leq x\,] \\ & & \qquad larger = [\,b \mid b \leftarrow xs, b > x\,] \end{array}$$

That is, the empty list is already sorted, and any non-empty list can be sorted by placing its head between the two lists that result from sorting those elements of its tail that are *smaller* and *larger* than the head.

6.5 Mutual recursion

Functions can also be defined using *mutual recursion*, in which two or more functions are all defined in terms of each other. For example, consider the library functions *even* and *odd*. For efficiency, these functions are normally defined using the remainder after dividing by two. However, for non-negative integers they can also be defined using mutual recursion:

```
even           ::  Int → Bool
even 0         =   True
even (n + 1)   =   odd n

odd            ::  Int → Bool
odd 0          =   False
odd (n + 1)    =   even n
```

That is, zero is even but not odd, and any strictly positive integer is even if its predecessor is odd, and odd if its predecessor is even. For example:

```
     even 4
=      { applying even }
     odd 3
=      { applying odd }
     even 2
=      { applying even }
     odd 1
=      { applying odd }
     even 0
=      { applying even }
     True
```

Similarly, functions that select the elements from a list at all even and odd positions (counting from zero) can be defined as follows:

```
evens           ::  [a] → [a]
evens []        =   []
evens (x : xs)  =   x : odds xs

odds            ::  [a] → [a]
odds []         =   []
odds (_ : xs)   =   evens xs
```

For example:

```
     evens "abcde"
=      { applying evens }
     'a' : odds "bcde"
=      { applying odds }
     'a' : evens "cde"
=      { applying evens }
     'a' : 'c' : odds "de"
=      { applying odds }
     'a' : 'c' : evens "e"
=      { applying evens }
     'a' : 'c' : 'e' : odds []
=      { applying odds }
     'a' : 'c' : 'e' : []
=      { string notation }
     "ace"
```

Recall that strings in Haskell are actually constructed as lists of characters. Hence, `"abcde"` is just an abbreviation for `['a','b','c','d','e']`.

6.6 Advice on recursion

Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it, may seem impossible when you first try to do it yourself, but becomes simple and natural with practice. In this section we offer some advice for defining functions in general, and recursive functions in particular, using a five-step process that we introduce by means of examples.

Example – *product*

As a simple first example, we show how the definition given earlier in this chapter for the library function that calculates the *product* of a list of numbers can be constructed in a stepwise manner.

Step 1: define the type

Thinking about types is very helpful when defining functions, so it is good practice to define the type of a function prior to starting to define the function itself. In this case, we begin with the type

$$product \;::\; [\,Int\,] \to Int$$

that states that *product* takes a list of integers and produces a single integer. As in this example, it is often useful to begin with a simple type, which can be refined or generalised later on as appropriate.

Step 2: enumerate the cases

For most types of argument, there are a number of standard cases to consider. For lists, the standard cases are the empty list and non-empty lists, so we can write down the following skeleton definition using pattern matching:

$$\begin{aligned} &product\;[\,] &&= \\ &product\;(n : ns) &&= \end{aligned}$$

For non-negative integers, the standard cases are 0 and $n + 1$, for logical values they are *False* and *True*, and so on. As with the type, we may need to refine the cases later on, but it is useful to begin with the standard cases.

Step 3: define the simple cases

By definition, the product of zero integers is one, because one is the identity for multiplication. Hence it is straightforward to define the empty list case:

$$\begin{aligned} &product\;[\,] &&= 1 \\ &product\;(n : ns) &&= \end{aligned}$$

As in this example, the simple cases often become base cases.

Step 4: define the other cases

How can we calculate the product of a non-empty list of integers? For this step, it is useful to first consider the ingredients that can be used, such as the function itself (*product*), the arguments (n and ns), and library functions of relevant types (+, −, ∗, and so on.) In this case, we simply multiply the first integer and the product of the remaining list of integers:

$$\begin{aligned} \mathit{product}\ [\,] &= 1 \\ \mathit{product}\ (n : ns) &= n * \mathit{product}\ ns \end{aligned}$$

As in this example, the other cases often become recursive cases.

Step 5: generalise and simplify

Once a function has been defined using the above process, it often becomes clear that it can be generalised and simplified. For example, the function *product* does not depend on the precise kind of numbers to which it is applied, so its type can be generalised from integers to any numeric type:

$$\mathit{product} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow a$$

In terms of simplification, we will see in chapter 7 that the pattern of recursion used in *product* is encapsulated by a library function called *foldr*, using which *product* can be redefined by a single equation:

$$\mathit{product} = \mathit{foldr}\ (*)\ 1$$

In conclusion, our final definition for *product* is as follows:

$$\begin{aligned} \mathit{product} &:: \mathit{Num}\ a \Rightarrow [a] \rightarrow a \\ \mathit{product} &= \mathit{foldr}\ (*)\ 1 \end{aligned}$$

This is precisely the definition from the standard prelude in appendix A, except that for efficiency reasons the use of *foldr* is replaced by the related library function *foldl*, which is also discussed in chapter 7.

Example – *drop*

As a more substantial example, we now show how the definition given earlier for the library function *drop* that removes a given number of elements from the start of a list can be constructed using the five-step process.

Step 1: define the type

Let us begin with a type that states that *drop* takes an integer and a list of values of some type a, and produces another list of such values:

$$\mathit{drop} :: \mathit{Int} \rightarrow [a] \rightarrow [a]$$

Note that we have made four decisions in defining this type: using integers rather than a more general numeric type, for simplicity; using currying rather than taking the arguments as a pair, for flexibility (see section 3.6); supplying the integer argument before the list argument, for readability (*drop* n xs can be read as "drop n elements from xs"); and, finally, making the function polymorphic in the type of the list elements, for generality.

Step 2: enumerate the cases

As there are two standard cases for the integer argument (0 and $n + 1$) and two for the list argument ([ ] and $x : xs$), writing down a skeleton definition using pattern matching requires four cases in total: