Use these laws, together with the definition of *unzip*, to prove the required result.

4.5 The fold functions

We have seen in the case of the datatype *Nat* that many recursive definitions can be expressed very succinctly using a suitable fold operator. Exactly the same is true of lists. Consider the following definition of a function h:

$$\begin{aligned} h\,[\,] &= e \\ h\,(x : xs) &= x \oplus h\,xs \end{aligned}$$

The function h works by taking a list, replacing [ ] by e and (:) by $\oplus$, and evaluating the result. For example, h converts the list

$$x_1 : (x_2 : (x_3 : (x_4 : [\,])))$$

to the value

$$x_1 \oplus (x_2 \oplus (x_3 \oplus (x_4 \oplus e)))$$

Since (:) associates to the right, there is no need to put in parentheses in the first expression. However, we do have to put in parentheses in the second expression because we do not assume that $\oplus$ associates to the right.

The pattern of definition given by h is captured in a function *foldr* (pronounced 'fold right') defined as follows:

$$\begin{aligned} \mathit{foldr} &:: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ \mathit{foldr}\ f\ e\,[\,] &= e \\ \mathit{foldr}\ f\ e\,(x : xs) &= f\ x\ (\mathit{foldr}\ f\ e\ xs) \end{aligned}$$

We can now write $h = \mathit{foldr}\,(\oplus)\,e$. The first argument of *foldr* is a binary operator that takes an α-value on its left and a β-value on its right, and delivers a β-value. The second argument of *foldr* is a β-value. The third argument is of type $[\alpha]$, and the result is a value of type β. In many cases, α and β will be instantiated to the same type, for instance when $\oplus$ denotes an associative operation.

The single function *foldr* can be used to define almost every function on lists that we have met so far. Here are just some examples:

$$\begin{aligned} \mathit{concat} &:: [[\alpha]] \to [\alpha] \\ \mathit{concat} &= \mathit{foldr}\,(+\!\!+)\,[\,] \end{aligned}$$

$$\begin{aligned} \mathit{reverse} &:: [\alpha] \to [\alpha] \\ \mathit{reverse} &= \mathit{foldr\ snoc}\ [\,] \\ &\ \textbf{where}\ \mathit{snoc}\ x\ \mathit{xs} = \mathit{xs} \mathbin{+\!\!+} [x] \end{aligned}$$

$$\begin{aligned} \mathit{length} &:: [\alpha] \to \mathit{Int} \\ \mathit{length} &= \mathit{foldr\ oneplus}\ 0 \\ &\ \textbf{where}\ \mathit{oneplus}\ x\ n = 1 + n \end{aligned}$$

$$\begin{aligned} \mathit{sum} &:: \mathit{Num}\ \alpha \Rightarrow [\alpha] \to \alpha \\ \mathit{sum} &= \mathit{foldr}\ (+)\ 0 \end{aligned}$$

$$\begin{aligned} \mathit{and} &:: [\mathit{Bool}] \to \mathit{Bool} \\ \mathit{and} &= \mathit{foldr}\ (\wedge)\ \mathit{True} \end{aligned}$$

$$\begin{aligned} \mathit{map} &:: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \mathit{map}\ f &= \mathit{foldr}\ (\mathit{cons} \cdot f)\ [\,] \\ &\ \textbf{where}\ \mathit{cons}\ x\ \mathit{xs} = x : \mathit{xs} \end{aligned}$$

$$\begin{aligned} \mathit{unzip} &:: [(\alpha, \beta)] \to ([\alpha], [\beta]) \\ \mathit{unzip} &= \mathit{foldr\ conss}\ ([\,], [\,]) \\ &\ \textbf{where}\ \mathit{conss}\ (x, y)\ (\mathit{xs}, \mathit{ys}) = (x : \mathit{xs}, y : \mathit{ys}) \end{aligned}$$

In some of these definitions the operator $\oplus$ that replaces $(:)$ is associative, and the constant c that replaces $[\,]$ is the unit of $\oplus$. For example, $(+\!\!+, [\,])$, $(+, 0)$, $(\wedge, \mathit{True})$, and so on. In other definitions, $\oplus$ is not associative; indeed, $\oplus$ may not even have a type appropriate for an associative operator, namely $\alpha \to \alpha \to \alpha$.

Not every function on lists can be defined as an instance of *foldr*. For example, *zip* cannot be so defined. Even for those that can, an alternative definition may be more efficient. To illustrate, suppose we want a function *decimal* that takes a list of digits and returns the corresponding decimal number; thus

$$\mathit{decimal}\,[x_0, x_1, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{(n-k)}$$

It is assumed that the most significant digit comes first in the list. One way to compute *decimal* efficiently is by a process of multiplying each digit by ten and adding in the following digit. For example,

$$\mathit{decimal}\,[x_0, x_1, x_2] = 10 \times (10 \times (10 \times 0 + x_0) + x_1) + x_2$$

This decomposition of a sum of powers is known as *Horner's rule*. Suppose we define $\oplus$ by $n \oplus x = 10 \times n + x$. Then we can rephrase the above equation as

$$\mathit{decimal}\,[x_0, x_1, x_2] \quad = \quad ((0 \oplus x_0) \oplus x_1) \oplus x_2$$

This is almost like an instance of *foldr*, except that the grouping is the other way round, and the starting value appears on the left, not on the right. In fact, the computation is dual: instead of processing from right to left, the computation processes from left to right.

This example motivates the introduction of a second fold operator, called *foldl* (pronounced 'fold left'). Informally,

$$\mathit{foldl}\,(\oplus)\,e\,[x_0, x_1, \ldots, x_{n-1}] \quad = \quad (\cdots((e \oplus x_0) \oplus x_1) \cdots) \oplus x_{n-1}$$

The parentheses group from the left, which is the reason for the name. The full definition of *foldl* is

$$\begin{array}{lcl} \mathit{foldl} & :: & (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\ \mathit{foldl}\,f\,e\,[\,] & = & e \\ \mathit{foldl}\,f\,e\,(x : xs) & = & \mathit{foldl}\,f\,(f\,e\,x)\,xs \end{array}$$

For example,

$$\begin{array}{cl} & \mathit{foldl}\,(\oplus)\,e\,[x_0, x_1, x_2] \\ = & \mathit{foldl}\,(\oplus)\,(e \oplus x_0)\,[x_1, x_2] \\ = & \mathit{foldl}\,(\oplus)\,((e \oplus x_0) \oplus x_1)\,[x_2] \\ = & \mathit{foldl}\,(\oplus)\,(((e \oplus x_0) \oplus x_1) \oplus x_2)\,[\,] \\ = & ((e \oplus x_0) \oplus x_1) \oplus x_2 \end{array}$$

If $\oplus$ is associative with unit e, then *foldr* $(\oplus)$ e and *foldl* $(\oplus)$ e define the same function on finite lists, as we will see in the following section.

As another example of the use of *foldl*, consider the following definition:

$$\begin{array}{lcl} \mathit{reverse}' & :: & [\alpha] \to [\alpha] \\ \mathit{reverse}' & = & \mathit{foldl}\;\mathit{cons}\;[\,] \quad \textbf{where}\; \mathit{cons}\;xs\;x = x : xs \end{array}$$

Note the order of the arguments to *cons*; we have $\mathit{cons} = \mathit{flip}\,(:)$, where the standard function *flip* is defined by $\mathit{flip}\,f\,x\,y = f\,y\,x$. The function *reverse'* reverses a finite list. For example,

$$\begin{array}{cl} & \mathit{reverse}'\,[x_0, x_1, x_2] \\ = & \mathit{cons}\,(\mathit{cons}\,(\mathit{cons}\,[\,]\,x_0)\,x_1)\,x_2 \\ = & \mathit{cons}\,(\mathit{cons}\,[x_0]\,x_1)\,x_2 \\ = & \mathit{cons}\,[x_1, x_0]\,x_2 \\ = & [x_2, x_1, x_0] \end{array}$$

One can prove that *reverse′* = *reverse* by induction, or as an instance of a more general result to be described in the following section. Of greater importance than the mere fact that *reverse* can be defined in a different way, is that *reverse′* gives a much more efficient program: *reverse′* takes time proportional to n on a list of length n, while *reverse* takes time proportional to n^2.

There are a number of important laws concerning *foldr* and its relationship to *foldl*, but we will postpone discussion to the following section.

4.5.1 Fold over nonempty lists

Say we wish to find the maximum element in a finite list of elements drawn from an ordered type. We would like to do this by defining

$$\begin{aligned} \mathit{maxlist} &:: (\mathit{Ord}\ \alpha) \Rightarrow [\alpha] \rightarrow \alpha \\ \mathit{maxlist} &= \mathit{foldr}\ (\mathbf{max})\ e \end{aligned}$$

where $(x\ \mathbf{max}\ y)$ returns the greater of x and y. But what should we choose as the value of e? In other words, what is *maxlist* [ ]? The choice is not arbitrary because we naturally want *maxlist* $[x] = x$, and the definition gives $(x\ \mathbf{max}\ e)$ instead. So e has to be smaller than any other value in the type α. Such a value may not exist. In Haskell there is a type class *Bounded* whose instances are types possessing maximum and minimum elements:

$$\begin{aligned} &\mathbf{class}\ \mathit{Bounded}\ \alpha\ \mathbf{where} \\ &\quad \mathit{minBound}, \mathit{maxBound} :: \alpha \end{aligned}$$

Hence one solution is to redefine *maxlist* to

$$\begin{aligned} \mathit{maxlist} &:: (\mathit{Ord}\ \alpha, \mathit{Bounded}\ \alpha) \Rightarrow [\alpha] \rightarrow \alpha \\ \mathit{maxlist} &= \mathit{foldr}\ (\mathbf{max})\ \mathit{minBound} \end{aligned}$$

An alternative solution that does not make use of *Bounded* is to introduce a new fold function, one that works on nonempty lists. In fact, there are two such functions, *foldr1* and *foldl1*. The definitions are

$$\begin{aligned} \mathit{foldr1} &:: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha \\ \mathit{foldr1}\ f\ (x : xs) &= \mathbf{if}\ \mathit{null}\ xs\ \mathbf{then}\ x\ \mathbf{else}\ f\ x\ (\mathit{foldr1}\ f\ xs) \end{aligned}$$

$$\begin{aligned} \mathit{foldl1} &:: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha \\ \mathit{foldl1}\ f\ (x : xs) &= \mathit{foldl}\ f\ x\ xs \end{aligned}$$

For example,

$$\begin{aligned} \mathit{foldr1}\ (\oplus)\ [x_0, x_1, x_2, x_3] &= x_0 \oplus (x_1 \oplus (x_2 \oplus x_3)) \\ \mathit{foldl1}\ (\oplus)\ [x_0, x_1, x_2, x_3] &= ((x_0 \oplus x_1) \oplus x_2) \oplus x_3 \end{aligned}$$

Now we can solve the problem by defining

$$maxlist \quad = \quad foldr1\,(\mathbf{max})$$

Of course, since **max** is associative, we could equally well have defined *maxlist* in terms of *foldl1*.

4.5.2 Scan left

Sometimes it is convenient to apply a *foldl* operation to every initial segment of a list. This is done by a function *scanl*, pronounced 'scan left'. For example,

$$scanl\,(\oplus)\;e\,[x_0, x_1, x_2] \quad = \quad [e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]$$

In particular, *scanl* $(+)$ 0 computes the list of accumulated sums of a list of numbers, and *scanl* $(\times)\,1\,[1\,..\,n]$ computes a list of the first n factorial numbers. More substantial examples of the use of *scanl* are given in Sections 5.2 and 5.3. We will give two programs for *scanl*; the first is the clearest, while the second is more efficient.

For the first program we will need the function *inits* that returns the list of all initial segments of a list. For example,

$$inits\,[x_0, x_1, x_2] \quad = \quad [[\,], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

The empty list has only one segment, namely the empty list itself; a list $(x : xs)$ has the empty list as its shortest initial segment, and all the other initial segments begin with x and are followed by an initial segment of xs. Hence

$$\begin{array}{lcl} inits & :: & [\alpha] \rightarrow [[\alpha]] \\ inits\,[\,] & = & [[\,]] \\ inits\,(x : xs) & = & [\,] : map\,(x :)\,(inits\,xs) \end{array}$$

The function *inits* can be defined more succinctly as an instance of *foldr*:

$$inits \quad = \quad foldr\,f\,[[\,]] \quad \mathbf{where}\; f\,x\,xss = [\,] : map\,(x :)\,xss$$

Now we define

$$\begin{array}{lcl} scanl & :: & (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\alpha] \\ scanl\,f\,e & = & map\,(foldl\,f\,e) \cdot inits \end{array}$$

This is the clearest definition of *scanl* but it leads to an inefficient program. The function f is applied k times in the evaluation of *foldl* f e on a list of length k and, since the initial segments of a list of length n are lists with lengths $0, 1, \ldots, n$, the function f is applied about $n^2/2$ times in total.

Let us now synthesise a more efficient program. The synthesis is by an induction argument on xs so we lay out the calculation in the same way.

Case ([ ]). Here we get $scanl\ f\ e\ [\,] = [e]$. The details are routine and are left as an exercise.

Case $(x : xs)$. We calculate

$$\begin{array}{ll}
 & scanl\ f\ e\ (x : xs) \\
= & \quad \{\text{definition of } scanl\} \\
 & map\ (foldl\ f\ e)\ (inits\ (x : xs)) \\
= & \quad \{\text{definition of } inits\} \\
 & map\ (foldl\ f\ e)\ ([\,] : map\ (x :)\ (inits\ xs)) \\
= & \quad \{\text{second equation for } map\} \\
 & foldl\ f\ e\ [\,] : map\ (foldl\ f\ e)\ (map\ (x :)\ (inits\ xs)) \\
= & \quad \{\text{first equation for } foldl\} \\
 & e : map\ (foldl\ f\ e)\ (map\ (x :)\ (inits\ xs)) \\
= & \quad \{\text{since } map \text{ is a functor}\} \\
 & e : map\ (foldl\ f\ e \cdot (x :))\ (inits\ xs) \\
= & \quad \{\text{claim: } foldl\ f\ e \cdot (x :) = foldl\ f\ (f\ e\ x)\} \\
 & e : map\ (foldl\ f\ (f\ e\ x))\ (inits\ xs) \\
= & \quad \{\text{definition of } scanl\} \\
 & e : scanl\ f\ (f\ e\ x)\ xs
\end{array}$$

The claim is left as a short exercise. In summary, we have derived

$$\begin{array}{lcl}
scanl\ f\ e\ [\,] & = & [e] \\
scanl\ f\ e\ (x : xs) & = & e : scanl\ f\ (f\ e\ x)\ xs
\end{array}$$

This program is more efficient in that the function f is applied exactly n times on a list of length n.

4.5.3 Scan right

The dual computation is given by $scanr$, defined by

$$scanr\ f\ a \quad = \quad map\ (foldr\ f\ a) \cdot tails$$

The function *tails* returns the tail segments of a list. For example,

$$tails\,[x_0, x_1, x_2] \;=\; [[x_0, x_1, x_2], [x_1, x_2], [x_2], [\,]]$$

Note that while *inits* produces a list of initial segments in increasing order of length, *tails* produces the tail segments in decreasing order of length. We can define *tails* by

$$\begin{aligned} tails\,[\,] &= [[\,]] \\ tails\,(x : xs) &= (x : xs) : tails\;xs \end{aligned}$$

The corresponding efficient program for *scanr* is given by

$$\begin{aligned} scanr\;f\;a\,[\,] &= [a] \\ scanr\;f\;a\,(x : xs) &= f\;x\,(head\;ys) : ys \\ &\;\; \textbf{where}\; ys = scanr\;f\;a\;xs \end{aligned}$$

We will leave the synthesis of the more efficient version as an exercise.

4.5.4 Scans over nonempty lists

We can also define two further scan functions, *scanl1* and *scanr1*. We have

$$\begin{aligned} scanl1\;f &= map\,(foldl1\;f) \cdot inits1 \\ scanr1\;f &= map\,(foldr1\;f) \cdot tails1 \end{aligned}$$

where *inits1* and *tails1* return the nonempty initial and final segments of a nonempty list. For example,

$$inits1\,(x : xs) \;=\; map\,(x :)\,(inits\;xs)$$

Using $foldl1\;f \cdot (x :) = foldl\;f\;x$, we obtain

$$\begin{aligned} scanl1 &:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\ scanl1\;f\,(x : xs) &= scanl\;f\;x\;xs \end{aligned}$$

The efficient version of *scanr1* is left as an exercise.

Exercises

4.5.1 Define *filter p* as an instance of *foldr*.

4.5.2 Consider the two functions *takeWhile* and *dropWhile*, which were discussed in Exercise 4.3.7. Define *takeWhile* as an instance of *foldr*. Can *dropWhile* be defined as an instance of a fold function?

4.5.3 Which, if either, of the following equations is true?

$$\begin{aligned} \mathit{foldl}\ (-)\ x\ xs &= x - \mathit{sum}\ xs \\ \mathit{foldr}\ (-)\ x\ xs &= x - \mathit{sum}\ xs \end{aligned}$$

4.5.4 Consider the following definition of a function *insert*:

$$\mathit{insert}\ x\ xs \;=\; \mathit{takeWhile}\ (\leq x)\ xs \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{dropWhile}\ (\leq x)\ xs$$

Show that if *xs* is in nondecreasing order, then so is *insert x xs* for all *x*. Use *insert* to define a function *isort* for sorting a list into nondecreasing order. Use *foldr*.

4.5.5 The function *remdups* removes adjacent duplicates from a list. For example, *remdups* $[1,2,2,3,3,3,1,1] = [1,2,3,1]$. Define *remdups* using either *foldl* or *foldr*.

4.5.6 Given a list $xs = [x_1, x_2, \ldots, x_n]$ of numbers, the sequence of successive maxima *ssm xs* is the longest subsequence $[x_{j_1}, x_{j_2}, \ldots, x_{j_m}]$ such that $j_1 = 1$ and $x_j < x_{j_k}$ for $j < j_k$. For example, the sequence of successive maxima of $[3,1,3,4,9,2,10,7]$ is $[3,4,9,10]$. Define *ssm* in terms of *foldl*.

4.5.7 Prove that *scanl f e* $[\,] = [e]$.

4.5.8 Justify the efficient program for *scanr*.

4.5.9 What list does *scanl* $(/)\ [1\,..\,n]$ produce?

4.5.10 The mathematical constant *e* is defined by $e = \sum_{n=0}^{\infty} 1/n!$. Write down an expression that can be used to evaluate *e* to some reasonable measure of accuracy.

4.5.11 Consider the datatype *Liste* α defined in Exercise 4.1.4. The fold function for this datatype is defined by

$$\begin{aligned} &\mathit{folde} && :: && (\beta \to \alpha \to \beta) \to \beta \to \mathit{Liste}\ \alpha \to \beta \\ &\mathit{folde}\ f\ e\ \mathit{Nil} && = && e \\ &\mathit{folde}\ f\ e\ (\mathit{Snoc}\ xs\ x) && = && f\ (\mathit{folde}\ f\ e\ xs)\ x \end{aligned}$$

The type assigned to *folde* is very nearly the same type as that assigned to *foldl*, the only difference being that $[\alpha]$ is replaced by *Liste* α. In fact,

$$\mathit{folde}\ f\ e \;=\; \mathit{foldl}\ f\ e \cdot \mathit{convert}$$

where *convert* :: $[\alpha] \to$ *Liste* α. Define the function *convert* and prove by induction that the above equation holds.

4.5.12 Derive an efficient program for *scanr1*.

4.6 Laws of fold

There are a number of important laws concerning *foldr* and its relationship with *foldl*. As we saw in Section 3.3, instead of having to prove a property of a recursively defined function over a recursive datatype by writing down an explicit induction proof, one can often phrase the property as an instance of one of the laws of the fold operator for the datatype.

4.6.1 Duality theorems

The first three laws are called *duality theorems* and concern the relationship between *foldr* and *foldl*. The *first duality theorem* is as follows:

First duality theorem. Suppose $\oplus$ is associative with unit e. Then

$$\mathit{foldr}\ (\oplus)\ e\ \mathit{xs} \quad = \quad \mathit{foldl}\ (\oplus)\ e\ \mathit{xs}$$

for all finite lists xs.

For example, we could have defined

$$\begin{array}{lcl} \mathit{sum} & = & \mathit{foldl}\ (+)\ 0 \\ \mathit{and} & = & \mathit{foldl}\ (\wedge)\ \mathit{True} \\ \mathit{concat} & = & \mathit{foldl}\ (+\!\!+)\ [\,] \end{array}$$

However, as we will elaborate in Chapter 7, it is sometimes more efficient to implement a function using *foldl*, and sometimes more efficient to use *foldr*.

The *second duality theorem* is a generalisation of the first.

Second duality theorem. Suppose $\oplus$, $\otimes$, and e are such that for all x, y, and z we have

$$\begin{array}{lcl} x \oplus (y \otimes z) & = & (x \oplus y) \otimes z \\ x \oplus e & = & e \otimes x \end{array}$$

In other words, $\oplus$ and $\otimes$ associate with each other, and e on the right of $\oplus$ is equivalent to e on the left of $\otimes$. Then

$$\mathit{foldr}\ (\oplus)\ e\ \mathit{xs} \quad = \quad \mathit{foldl}\ (\otimes)\ e\ \mathit{xs}$$

for all finite lists xs.

Proof. The proof of the second duality theorem is by induction on xs.

Case ([ ]). Both sides simplify to e.

Case $(x : xs)$. For the left-hand side we reason

$$\begin{aligned}
& \mathit{foldr}\ (\oplus)\ e\ (x : xs) \\
= \quad & \{\text{definition of } \mathit{foldr}\} \\
& x \oplus \mathit{foldr}\ (\oplus)\ e\ xs \\
= \quad & \{\text{induction hypothesis}\} \\
& x \oplus \mathit{foldl}\ (\otimes)\ e\ xs
\end{aligned}$$

For the right-hand side we reason

$$\begin{aligned}
& \mathit{foldl}\ (\otimes)\ e\ (x : xs) \\
= \quad & \{\text{definition of } \mathit{foldl}\} \\
& \mathit{foldl}\ (\otimes)\ (e \otimes x)\ xs \\
= \quad & \{\text{assumption}\} \\
& \mathit{foldl}\ (\otimes)\ (x \oplus e)\ xs
\end{aligned}$$

The two sides have simplified to two different expressions. To show that they are equal we have to resort to a second induction proof. However, if one takes as the second induction hypothesis the obvious assertion that

$$x \oplus \mathit{foldl}\ (\otimes)\ e\ xs \quad = \quad \mathit{foldl}\ (\otimes)\ (x \oplus e)\ xs$$

for all x and finite lists xs, then the induction doesn't work. The only way to appreciate this fact is to try it, a task we leave as an exercise. What is needed is a *stronger* hypothesis, namely

$$x \oplus \mathit{foldl}\ (\otimes)\ y\ xs \quad = \quad \mathit{foldl}\ (\otimes)\ (x \oplus y)\ xs$$

for all x, y, and finite lists xs. As we saw in Section 4.2 in the discussion of *reverse*, making an induction hypothesis stronger can often simplify a proof.

Case ([ ]). Both sides simplify to $x \oplus a$.

Case $(z : xs)$. (Note the use of z to avoid confusion with the variable x already present in the assertion.) For the left-hand side we reason

$$x \oplus \mathit{foldl}\ (\otimes)\ y\ (z;\ xs)$$

$$
\begin{aligned}
&= \quad \{\text{definition}\}\\
&\quad x \oplus \mathit{foldl}\ (\otimes)\ (y \otimes z)\ \mathit{xs}\\
&= \quad \{\text{induction hypothesis}\}\\
&\quad \mathit{foldl}\ (\otimes)\ (x \oplus (y \otimes z))\ \mathit{xs}\\
&= \quad \{\text{assumption}\}\\
&\quad \mathit{foldl}\ (\otimes)\ ((x \oplus y) \otimes z)\ \mathit{xs}
\end{aligned}
$$

For the right-hand side we reason

$$
\begin{aligned}
&\quad \mathit{foldl}\ (\otimes)\ (x \oplus y)\ (z : \mathit{xs})\\
&= \quad \{\text{definition}\}\\
&\quad \mathit{foldl}\ (\otimes)\ ((x \oplus y) \otimes z)\ \mathit{xs}
\end{aligned}
$$

□

The second duality theorem has the first duality theorem as a special case, namely when $(\oplus) = (\otimes)$.

To illustrate the second duality theorem, consider the following definitions of *length* and *reverse*:

$$
\begin{aligned}
\mathit{length} &= \mathit{foldr}\ \mathit{oneplus}\ 0, && \text{where } \mathit{oneplus}\ x\ n = 1 + n\\
\mathit{length} &= \mathit{foldl}\ \mathit{plusone}\ 0, && \text{where } \mathit{plusone}\ n\ x = n + 1\\[1ex]
\mathit{reverse} &= \mathit{foldr}\ \mathit{snoc}\ [\,], && \text{where } \mathit{snoc}\ x\ \mathit{xs} = \mathit{xs} +\!\!+ [x]\\
\mathit{reverse} &= \mathit{foldl}\ \mathit{cons}\ [\,], && \text{where } \mathit{cons}\ \mathit{xs}\ x = x : \mathit{xs}
\end{aligned}
$$

The functions *oneplus*, *plusone*, and 0 meet the conditions of the second duality theorem, as do *snoc*, *cons*, and [ ]. We leave the verification as an exercise. Hence the two definitions of *length* and *reverse* are equivalent on all finite lists. It is not obvious whether there is any practical difference between the two definitions of *length*, but the second program for *reverse* is the more efficient of the two,

The proof of the *third duality theorem* is left as an exercise.

Third duality theorem. For all finite lists *xs*,

$$\mathit{foldr}\ f\ e\ \mathit{xs} \quad = \quad \mathit{foldl}\ (\mathit{flip}\ f)\ e\ (\mathit{reverse}\ \mathit{xs})$$

where $\mathit{flip}\ f\ x\ y = f\ y\ x$.

To illustrate the third duality theorem, consider

$$\mathit{foldr}\ (:)\ [\,]\ \mathit{xs} \quad = \quad \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ (\mathit{reverse}\ \mathit{xs})$$

Since $foldr\ (:)\ [\,] = id$ and $foldl\ (flip\ (:))\ [\,] = reverse$, we obtain

$$xs \quad = \quad reverse\ (reverse\ xs)$$

for all finite lists xs, a result we have already proved directly.

4.6.2 Fusion

The duality theorems deal with the relationship between the two kinds of fold operator on lists; the fusion theorems deal with one kind of fold operator only. The fusion theorems for *foldr* and *foldl* are as follows.

Fusion theorem for *foldr*. If f is strict, $f\ a = b$, and $f\ (g\ x\ y) = h\ x\ (f\ y)$ for all x and y, then

$$f \cdot foldr\ g\ a \quad = \quad foldr\ h\ b$$

Fusion theorem for *foldl*. If f is strict, $f\ a = b$, and $f\ (g\ x\ y) = h\ (f\ x)\ y$ for all x and y, then

$$f \cdot foldl\ g\ a \quad = \quad foldl\ h\ b$$

The proof of the fusion theorem for *foldr* is very similar to the one in the previous chapter for *foldn*, and is left as an exercise. The proof of the fusion theorem for *foldl* is trickier, and is explored in Exercise 4.6.3.

The fusion law is very general, and there are useful special cases. Four of these are considered next.

4.6.3 Fold-map fusion

Recall that $map\ g = foldr\ (cons \cdot g)\ [\,]$, where $cons\ x\ xs = x : xs$. Consider what conditions ensure that

$$foldr\ f\ a \cdot map\ g \quad = \quad foldr\ h\ b$$

Looking at the conditions of the fusion theorem for *foldr*, we have that $foldr\ f\ a$ is a strict function, and the second condition is satisfied by taking $a = b$. The third condition expands to

$$\begin{array}{ll} & foldr\ f\ a\ ((cons \cdot g)\ x\ xs) = h\ x\ (foldr\ f\ a\ xs) \\ \equiv & \quad \{\text{definition of } cons\} \\ & foldr\ f\ a\ (g\ x : xs) = h\ x\ (foldr\ f\ a\ xs) \\ \equiv & \quad \{\text{definition of } foldr\} \end{array}$$

$$\begin{aligned} & f\ (g\ x)\ (\mathit{foldr}\ f\ a\ \mathit{xs}) = h\ x\ (\mathit{foldr}\ f\ a\ \mathit{xs}) \\ \Leftarrow\quad & \{\text{generalisation}\} \\ & f\ (g\ x)\ y = h\ x\ y \end{aligned}$$

The symbol $\Leftarrow$ is pronounced 'follows from'. Since $h\ x\ y = f\ (g\ x)\ y$ if and only if $h\ x = f\ (g\ x)$ if and only if $h = f \cdot g$, we get the result that

$$\mathit{foldr}\ f\ a \cdot \mathit{map}\ g \quad = \quad \mathit{foldr}\ (f \cdot g)\ a$$

In words, a fold after a map can always be expressed as a single fold. We will call this law the *fold-map fusion* law.

4.6.4 Fold-concat fusion

As a second example, recall that $\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]$. Consider what conditions ensure that

$$\mathit{foldr}\ f\ a \cdot \mathit{concat} \quad = \quad \mathit{foldr}\ h\ b$$

Again, looking at the conditions for fusion, we have that $\mathit{foldr}\ f\ a$ is strict, and the second condition is satisfied by taking $a = b$. The third condition expands to

$$\begin{aligned} & \mathit{foldr}\ f\ a\ (\mathit{xs} +\!\!+ \mathit{ys}) = h\ \mathit{xs}\ (\mathit{foldr}\ f\ a\ \mathit{ys}) \\ \equiv\quad & \{\text{claim: } \mathit{foldr}\ f\ a\ (\mathit{xs} +\!\!+ \mathit{ys}) = \mathit{foldr}\ f\ (\mathit{foldr}\ f\ a\ \mathit{ys})\ \mathit{xs}\} \\ & \mathit{foldr}\ f\ (\mathit{foldr}\ f\ a\ \mathit{ys})\ \mathit{xs} = h\ \mathit{xs}\ (\mathit{foldr}\ f\ a\ \mathit{ys}) \end{aligned}$$

Hence we can take $h\ \mathit{xs}\ y = \mathit{foldr}\ f\ y\ \mathit{xs}$ or, more shortly, $h = \mathit{flip}\ (\mathit{foldr}\ f)$. The proof of the claim is left as an exercise. Hence we have shown that

$$\mathit{foldr}\ f\ a \cdot \mathit{concat} \quad = \quad \mathit{foldr}\ (\mathit{flip}\ (\mathit{foldr}\ f))\ a$$

We will call this the *fold-concat fusion* law.

4.6.5 Bookkeeping law

As a third example, consider what conditions on f and a ensure that

$$\mathit{foldr}\ f\ a \cdot \mathit{concat} \quad = \quad \mathit{foldr}\ f\ a \cdot \mathit{map}\ (\mathit{foldr}\ f\ a)$$

We will call this particular equation the *bookkeeping* law. The special instance $\mathit{sum} \cdot \mathit{concat} = \mathit{sum} \cdot \mathit{map}\,\mathit{sum}$ is exploited by every bookkeeper who sums each individual day's accounts, and then gets the year's total by summing the sums.

We can use the fold-concat and fold-map fusion laws to rewrite the bookkeeping law in the form

$$\mathit{foldr}\ (\mathit{flip}\ (\mathit{foldr}\ f))\ a \quad = \quad \mathit{foldr}\ (f \cdot \mathit{foldr}\ f\ a)\ a$$

This equation holds if $\mathit{flip}\ (\mathit{foldr}\ f) = f \cdot \mathit{foldr}\ f\ a$. Applying both sides to xs and then y, we get

$$\mathit{foldr}\ f\ y\ xs \quad = \quad f\ (\mathit{foldr}\ f\ a\ xs)\ y$$

This equation holds if f is associative with unit a, a proof we will leave as an exercise.

In summary, the bookkeeping law is valid whenever the arguments to *foldr* are an associative operation and its unit. For example, since $\mathit{sum} = \mathit{foldr}\ (+)\ 0$, the bookkeeper's accounting method is justified. As another example,

$$\mathit{concat} \cdot \mathit{concat} \quad = \quad \mathit{concat} \cdot \mathit{map}\ \mathit{concat}$$

This equation says that flattening a list of lists of lists into one long list can be done by concatenating outside-in or inside-out.

4.6.6 Fold-scan fusion

Consider the equation

$$1 + x_0 + x_0 \times x_1 + x_0 \times x_1 \times x_2 \quad = \quad 1 + x_0 \times (1 + x_1 \times (1 + x_2))$$

The equation generalises to an arbitrary list of numbers and can be expressed in the form

$$\mathit{foldr1}\ (+) \cdot \mathit{scanl}\ (\times)\ 1 \quad = \quad \mathit{foldr}\ (\odot)\ 1$$

where $x \odot y = 1 + x \times y$. This leads to a more general question: when does the law

$$\mathit{foldr1}\ (\oplus) \cdot \mathit{scanl}\ (\otimes)\ e \quad = \quad \mathit{foldr}\ (\odot)\ e$$

hold, where $x \odot y = e \oplus (x \otimes y)$?

Under the assumption that $\otimes$ is associative and e is the unit of $\otimes$, we can express *scanl* as an instance of *foldr*:

$$\mathit{scanl}\ (\otimes)\ e \quad = \quad \mathit{foldr}\ g\ [e] \quad \text{where } g\ x\ xs = e : \mathit{map}\ (x\otimes)\ xs$$

The proof is left as an exercise. Having rewritten *scanl* as an instance of *foldr*, we can see that the required law is yet another instance of the fusion law for *foldr*. The first two conditions are immediate, and the third expands to

$$\begin{array}{ll} & \mathit{foldr1}\ (\oplus)\ (g\ x\ xs) = x \odot (\mathit{foldr1}\ (\oplus)\ xs) \\ \equiv & \quad \{\text{definition of } g\} \\ & \mathit{foldr1}\ (\oplus)\ (e : \mathit{map}\ (x\otimes)\ xs) = x \odot (\mathit{foldr1}\ (\oplus)\ xs) \\ \equiv & \quad \{\text{definition of } \mathit{foldr1} \text{ and } \odot\} \\ & e \oplus \mathit{foldr1}\ (\oplus)\ (\mathit{map}\ (x\otimes)\ xs) = e \oplus (x \otimes \mathit{foldr1}\ (\oplus)\ xs) \\ \Leftarrow & \quad \{\text{substitution}\} \\ & \mathit{foldr1}\ (\oplus) \cdot \mathit{map}\ (x\otimes) = (x\otimes) \cdot \mathit{foldr1}\ (\oplus) \end{array}$$

The last equation holds if $\otimes$ distributes over $\oplus$, that is, if

$$x \otimes (y \oplus z) \quad = \quad (x \otimes y) \oplus (x \otimes z)$$

for all x, y, and z. The proof is left as an exercise.

In summary, if $\otimes$ is associative with unit e, and $\otimes$ distributes over $\oplus$, then $\mathit{foldr1}\ (\oplus) \cdot \mathit{scanl}\ (\otimes)\ e = \mathit{foldr}\ (\odot)\ e$ where $x \odot y = e \oplus (x \otimes y)$. We will call this the *fold-scan* fusion law.

In the remainder of this section we will give an example of how these laws can be used in the synthesis of efficient programs.

4.6.7 Example: the maximum segment sum

The maximum segment sum problem is a famous one and its history is described in J. Bentley's *Programming Pearls* (1987). Given a sequence of numbers it is required to compute the maximum of the sums of all segments in the sequence. For example, the sequence

$$[-1, 2, -3, 5, -2, 1, 3, -2, -2, -3, 6]$$

has maximum sum 7, the sum of the segment $[5, -2, 1, 3]$. On the other hand, the sequence $[-1, -2, -3]$ has a maximum segment sum of zero, since the empty sequence $[\,]$ is a segment of every list and its sum is zero. It follows that the maximum segment sum is always nonnegative. (The exercises deal with a variant problem in which the empty sequence is excluded, so the maximum segment sum can be negative.)

The problem can be specified formally by

$$\begin{array}{lcl} \mathit{mss} & :: & [\mathit{Int}] \rightarrow \mathit{Int} \\ \mathit{mss} & = & \mathit{maxlist} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{segs} \end{array}$$

where *segs* returns a list of all segments of a list. This function can be defined in a number of ways, including

$$\mathit{segs} \quad = \quad \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails}$$

This definition corresponds to the process of taking all the initial segments of all the tail segments. For example,

$$\mathit{segs}\,\text{“abc”} \quad = \quad [\text{“”}, \text{“a”}, \text{“ab”}, \text{“abc”}, \text{“”}, \text{“b”}, \text{“bc”}, \text{“”}, \text{“c”}, \text{“”}]$$

The empty sequence appears four times in this list, once for every tail segment of “abc”.

Direct evaluation of *mss* will take a number of steps proportional to n^3 on a list of length n. There are about n^2 segments, and summing them all takes about n^3 steps. It is not obvious that we can do better than cubic time for this problem.

However, let us see what we can achieve with some calculation. Installing the definition of *segs* in *mss*, we have to compute

$$\mathit{maxlist} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails}$$

The only thing we can do with this expression (apart from going backwards) is to use the law $\mathit{map}\,f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\,(\mathit{map}\,f)$ to replace the second and third terms, obtaining

$$\mathit{maxlist} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum}) \cdot \mathit{map}\ \mathit{inits} \cdot \mathit{tails}$$

There are two things we can do now: we can either apply the bookkeeping law to the first two terms, or apply the law $\mathit{map}\,(f \cdot g) = \mathit{map}\,f \cdot \mathit{map}\,g$ to the third and fourth terms. Of course, we can also do both these steps at once, in which case the result will be

$$\mathit{maxlist} \cdot \mathit{map}\ \mathit{maxlist} \cdot \mathit{map}\ (\mathit{map}\ \mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails}$$

For the next step we can apply the *map* law once more and obtain

$$\mathit{maxlist} \cdot \mathit{map}\ (\mathit{maxlist} \cdot \mathit{map}\ \mathit{sum} \cdot \mathit{inits}) \cdot \mathit{tails}$$

The term $\mathit{map}\ \mathit{sum} \cdot \mathit{inits}$ can be replaced by a *scanl*, a substitution that could have been done one step earlier:

$$\mathit{maxlist} \cdot \mathit{map}\ (\mathit{maxlist} \cdot \mathit{scanl}\ (+)\ 0) \cdot \mathit{tails}$$

Since $\mathit{maxlist} = \mathit{foldr1}\ (\mathbf{max})$ and $(+)$ distributes over $(\mathbf{max})$, the fold-scan fusion law now gives

$$\mathit{maxlist} \cdot \mathit{map}\ (\mathit{foldr}\ (\odot)\ 0) \cdot \mathit{tails}$$

where $x \odot y = 0 \,\mathbf{max}\, (x + y)$. Finally, the last two terms can be replaced by a *scanr*, and we obtain

$$mss \quad = \quad maxlist \cdot scanr\ (\odot)\ 0$$

The result is a linear-time program for the maximum segment sum.

Exercises

4.6.1 Prove the third duality theorem.

4.6.2 Prove the fusion theorem for *foldr*.

4.6.3 The fusion theorem for *foldl* states that $f \cdot foldl\ g\ a = foldl\ h\ b$ provided f is strict, $f\ a = b$, and $f\ (g\ x\ y) = h\ (f\ x)\ y$ for all x and y.
In the induction proof the case $(x : xs)$ requires us to show that

$$f\ (foldl\ g\ (g\ a\ x)\ xs) \quad = \quad foldl\ h\ (h\ (f\ a)\ x)\ xs$$

To do this we need a second induction hypothesis, namely that

$$f\ (foldl\ g\ (g\ x\ y)\ zs) \quad = \quad foldl\ h\ (h\ (f\ x)\ y)\ zs$$

for all x and y and for all lists zs. Prove this equation and complete the proof of the fusion law.

4.6.4 Prove the equations

$$\begin{aligned} foldr\ f\ a\ (xs +\!\!+\ ys) \quad &= \quad foldr\ f\ (foldr\ f\ a\ ys)\ xs \\ foldl\ f\ a\ (xs +\!\!+\ ys) \quad &= \quad foldl\ f\ (foldl\ f\ a\ xs)\ ys \end{aligned}$$

What restrictions on xs and ys are necessary, if any?

4.6.5 Prove that $scanl\ (\otimes)\ e$ can be written as a *foldr* if $\otimes$ is associative with unit e. Can $scanl\ (\otimes)\ e$ be written as a *foldl* without any assumptions on $\otimes$ and e?

4.6.6 Prove that if $\otimes$ distributes over $\oplus$, then

$$foldr1\ (\oplus) \cdot map\ (x\otimes) \quad = \quad (x\otimes) \cdot foldr1\ (\oplus)$$

4.6.7 It is tempting to try to push the calculation of the maximum segment sum one step further and use the fold-scan fusion law to replace $maxlist \cdot scanr\ (\odot)\ 0$ by a *foldl*. Why doesn't this work? (See Exercise 4.6.10 for a version of fold-scan fusion that does work.)

4.6.8 Give a fusion law for *foldr1* and use it to give conditions on f, g, and h so that the law

$$\mathit{foldr1}\ f \cdot \mathit{scanl1}\ g \quad = \quad \mathit{foldr1}\ h$$

is valid.

4.6.9 Define *segs1* by *segs1* = *concat* · *map inits1* · *tails1*, where *inits1* returns the nonempty initial segments of a list, and *tails1* the nonempty tail segments. Use this version to state and solve a version of the maximum segment sum problem that leaves out empty segments.

4.6.10 There is a version of the fold-scan fusion law that does not depend on any properties of the operators involved. The law is

$$\mathit{foldl1}\ (\oplus) \cdot \mathit{scanl}\ (\otimes)\ e \quad = \quad \mathit{fst} \cdot \mathit{foldl}\ (\odot)\ (a, a)$$

with $(x, y) \odot z = (x \oplus t, t)$, where $t = y \otimes z$. Prove this version of fold-scan fusion.

4.7 Chapter notes

Haskell provides many more standard functions on lists than we have described in this chapter. For full details, consult the Haskell standard prelude and the library of list functions.

A great deal of work has been done on automatic or machine-aided generation of induction proofs; see, for example, Boyer and Moore (1979), Gordon, Milner, and Wadsworth (1979), Paulson (1983), Martin and Nipkow (1990).

The relationship between polymorphic functions and naturality conditions is explored in Wadler (1989); see also Bird and de Moor (1997).

The fusion laws for fold operators were systematised in Malcolm (1990), Fokkinga (1992), Jeuring (1993), and Meijer (1992).

The maximum segment sum problem is described in Bentley (1987). The derivation recorded in this chapter was first given in Bird (1989).