Algebraic Identities for Program Calculation

Article *in* The Computer Journal · February 1989

DOI: 10.1093/comjnl/32.2.122 · Source: DBLP

Richard Bird
University of Oxford

Algebraic Identities for Program Calculation

R. S. BIRD
Programming Research Group, University of Oxford, 11 Keble Road, Oxford OX1 3QD

To calculate a program means to derive it from a suitable specification by a process of equational reasoning. We describe a number of basic algebraic identities that turn out to be extremely useful in this task. These identities express relationships between the higher-order functions commonly encountered in functional programming. The idea of program calculation is illustrated with two non-trivial examples.

Received November 1988

1. INTRODUCTION

There is a style of functional programming (see, for example, Refs 1, 6 and 9), which focuses attention on a small collection of powerful higher-order functions that capture common patterns of computation. The idea is to try and express programs in terms of the composition of particular instances of these useful functions. Explicit use of recursion is thereby avoided, except as a last resort.

A similar, indeed dual, style can be advocated when it comes to program proof. The idea here is to try and conduct proofs through equational reasoning that exploits the algebraic properties of the collection of higher-order functions. Explicit use of induction is thereby avoided. Of course, functions in the repertoire may be defined recursively, and their properties may be established by induction, but once this has been done further use of recursion and induction is shunned. A proof based on algebraic reasoning is called *calculational*. One advantage of this approach is that, by starting with a suitable specification of the problem in hand, one can often derive an acceptably efficient program by straightforward calculation.

The purpose of this paper is to review some of the basic functions and their properties, and use them in one or two example calculations. The work can be regarded as an elaboration and extension of Ref. 9. Like Wadler,[9] we deal exclusively with functions over finite lists, a data structure about which a good deal is known. However, recent evidence shows that an analogous computational basis can be set up for other data structures, such as trees and arrays.[3]

In order to make the material as accessible as possible, we shall use the notation for functional programming described by Bird and Wadler.[6] This is very similar to that used in Miranda.*[8] (Our preferred notation (see Ref. 4) is rather different. For a start, it is more concise and mathematical. Moreover, the associated semantics is algebraic rather than domain-theoretic.)

2. MAP AND FOLD

The function map is well known. Informally, map applies a function to each element of a list; we have

$$\mathsf{map\,f}\,[x_1, x_2, \ldots, x_n] = [\mathsf{f}\,x_1, \mathsf{f}\,x_2, \ldots, \mathsf{f}\,x_n]$$

The formal recursive definition of map uses pattern matching with the primitive list construction operator (:), pronounced *cons*:

$$\mathsf{map\,f}\,[\,] = [\,]$$
$$\mathsf{map\,f}\,(\mathsf{x:xs}) = \mathsf{f\,x : map\,f\,xs}$$

* Miranda is a trademark of Research Software Ltd.

An alternative recursive characterisation, sufficient to define map for all finite lists, is as follows:

$$\mathsf{map\,f}[\,] = [\,] \qquad (1)$$

$$\mathsf{map\,f}\,(\mathsf{xs} +\!\!+ [\mathsf{x}]) = (\mathsf{map\,f\,xs}) +\!\!+ [\mathsf{f\,x}] \qquad (2)$$

In the last equation, the operator $+\!\!+$ denotes concatenation, and [x] denotes a singleton list, so xs $+\!\!+$ [x] is the list xs with x appended as a new last element. This second characterisation of map is used in a proof by induction in Section 7.

Probably the most useful law about map is the fact that it distributes over functional composition:

$$(\mathsf{map\,f}) \cdot (\mathsf{map\,g}) = \mathsf{map}\,(\mathsf{f} \cdot \mathsf{g}) \qquad (3)$$

This identity, which we shall call *map distributivity* (it is called the *map-map* rule in Ref. 9), says that if we apply g to every element of a list, and then apply f to every element of the result, the same effect is achieved by applying (f·g) to the original list. This particular law holds for infinite as well as finite lists. However, some other laws we shall describe hold only for finite lists. To avoid possible confusion on this point, we shall restrict our attention to finite lists only. Thus, in this paper all functional equations, such as (3), are asserted for finite lists only, whether or not they hold for the infinite case as well.

Since the expression on the left of equation (3) involves two traversals of the list, while the right-hand side involves only one, one could say that this rule is an example of *loop fusion*.[7] Below, we shall see further rules that can be interpreted as loop fusion rules.

Two more well-known functions are foldr and foldl. Informally, we have

$$\mathsf{foldr}\,(\oplus)\,\mathsf{a}\,[x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus \mathsf{a}) \cdots))$$

and

$$\mathsf{foldl}\,(\oplus)\,\mathsf{a}\,[x_1, x_2, \ldots, x_n] = (\cdots ((\mathsf{a} \oplus x_1) \oplus x_2) \cdots) \oplus x_n$$

An immediate consequence of these informal definitions is that if the operator $\oplus$ is associative with identity element a, then

$$\mathsf{foldr}\,(\oplus)\,\mathsf{a} = \mathsf{foldl}\,(\oplus)\,\mathsf{a} \qquad (4)$$

This law is cited in Ref. 6, where it is called the *first duality theorem*. It holds only for finite lists. Examples of the use of foldr and foldl include:

$$\mathsf{concat} = \mathsf{foldr}\,(+\!\!+)\,[\,]$$

$$\mathsf{sum} = \mathsf{foldl}(+)0$$

$$\text{product} = \text{foldl}(\times)1$$

$$\text{min} = \text{foldl}(\downarrow)\infty$$

$$\text{max} = \text{foldl}(\uparrow)(-\infty)$$

Recall, in the definition of concat, that ++ denotes the operation of list concatenation. The function concat takes a list of lists and concatenates them into one long list. The function sum sums a list of numbers, while product computes their numeric product. The function min computes the minimum element of a list of numbers (by definition, the operator ↓ selects the smaller of its two arguments), while max computes the maximum. We have cheated a little in the definition of min and max by allowing the appearance of two 'fictitious' values ∞ and $-\infty$; these are the identity elements of ↓ and ↑ respectively.

In each of the above examples the conditions of the first duality theorem hold, so it does not matter whether we use foldr or foldl. However, the theorem states equality only for finite lists, and the possibility of applying such functions to infinite lists does mean that the situation is not quite so simple. Since we are working exclusively with finite lists we will not elaborate this point, but a full discussion is given in Ref. 6.

To avoid saying everything twice, let us now concentrate on just one of the fold operators, foldl. There are two recursive characterisations of foldl that are useful in proofs by induction. First, there is the formal definition

$$\text{foldl}(\oplus)\,a[\,] = a$$

$$\text{foldl}(\oplus)\,a(x{:}xs) = \text{foldl}(\oplus)(a \oplus x)\,xs$$

which appears in Ref. 6. Secondly, there is the following characterisation. Suppose f is a function that satisfies the two equations

$$f[\,] = a \tag{5}$$

$$f(xs +\!\!+ [x]) = (f\,xs) \oplus x \tag{6}$$

for some value a and binary operator ⊕. Then

$$f = \text{foldl}\,(\oplus)\,a$$

In Section 7 we shall use this second characterisation in a proof of a useful identity concerning foldl.

Now let us turn to some of the laws. The first concerns the particular function concat and says that

$$\text{map}\,f\cdot\text{concat} = \text{concat}\cdot\text{map}\,(\text{map}\,f) \tag{7}$$

This law is called *map promotion*. In words, the result of concatenating a list of lists, and then applying f to each element, is the same as applying map f to each component list and then concatenating the outcomes.

The second law, called *fold promotion*, takes a basically similar form: if ⊕ is an associative operator with identity element a, then

$$\text{foldl}\,(\oplus)\,a\cdot\text{concat} = \text{foldl}\,(\oplus)\,a\cdot\text{map}(\text{foldl}(\oplus)\,a) \tag{8}$$

For example, since sum = foldl(+)0 we have

$$\text{sum}\cdot\text{concat} = \text{sum}\cdot\text{map}\,\text{sum}$$

In words, this says that to sum a list of lists of numbers we can either concatenate the lists and sum the results, or sum each component list and then sum the results.

The fold promotion law requires ⊕ to be associative with identity element a. If we forgo this condition, we get another law:

$$\text{foldl}(\oplus)a\cdot\text{concat} = \text{foldl}\,(\otimes)\,a \tag{9}$$

where ⊗ is defined by the equation

$$u \otimes x = \text{foldl}\,(\oplus)\,u\,x$$

For example, if we apply both sides of (9) to the list [xs, ys] and simplify, we get

$$\text{foldl}\,(\oplus)\,a\,(xs +\!\!+ ys) = \text{foldl}\,(\oplus)\,(\text{foldl}\,(\oplus)\,a\,xs)\,ys \tag{10}$$

This equation is used in the next section.

A final law, called *fold-map fusion* (it is called the *reduce-map* rule in Ref. 9), says that

$$\text{foldl}\,(\oplus)\,a\cdot\text{map}\,f = \text{foldl}\,(\odot)\,a \tag{11}$$

where ⊙ is defined by the equation

$$b \odot x = b \oplus f\,x$$

This is another example of a loop-fusion law. In words, rather than applying f to each element of a list and then folding with ⊕, we can fold directly with a modified operator ⊙, designed to take account of the fact that f must first be applied to its right argument.

We shall not prove the above identities. These can be done by inductive proofs similar to the one we shall give in Section 7.

3. LEFT-ZEROS

The evaluation of a foldl operation on a list requires that the list be traversed in its entirety. Such a traversal can be cut short if we recognise the possibility that the operator involved in the foldl may have *left-zeros*. By definition, z is a left-zero of ⊕ if

$$z \oplus x = z$$

for all x. An operator may have none, one or many left-zeros. If z is a left-zero of ⊕, then

$$\text{foldl}\,(\oplus)\,z\,xs = z$$

for all lists xs. It follows by equation (10) that

$$\text{foldl}\,(\oplus)\,a\,(xs +\!\!+ ys) = \text{foldl}\,(\oplus)\,a\,xs$$

whenever the right-hand side is a left-zero of ⊕. In words, we can stop traversing the list as soon as some left-zero has been computed.

This idea can be implemented as a modified version, fastfoldl, of foldl defined in the following way:

$$\begin{aligned}\text{fastfoldl}\,(\oplus)\,p\,a\,xs &= a, \quad \text{if } p\,a \vee xs = [\,]\\ &= \text{fastfoldl}\,(\oplus)\,p\,(a \oplus y)\,ys,\\ &\qquad \text{otherwise}\\ &\qquad \text{where } y{:}ys = xs\end{aligned}$$

In the new version of foldl a boolean-valued function p has been included as an extra argument. The intention is that fastfoldl (⊕) p should be called with p set to a test for whether its argument is a left-zero of ⊕.

Let us now see an application of this simple idea.

4. MINIMAX

Suppose we are given a non-empty list of non-empty lists of numbers. We want to compute the minimum of the maximum numbers in each list. In symbols, we want to evaluate minimax, where

$$\mathsf{minimax} = \mathsf{min} \cdot \mathsf{map\,max}$$

This is an executable definition, but not the best possible. In order to return the correct answer it is not always necessary to inspect every element of every list in the input.

Rather than think hard about a possible improvement and then try to justify it, let us just calculate. Recalling the definition

$$\mathsf{min} = \mathsf{foldl}\,(\downarrow)\,\infty$$

we see that the right-hand side of the definition of minimax matches the pattern of the fold-map fusion law, equation (11). We therefore get that

$$\mathsf{minimax} = \mathsf{foldl}\,(\odot)\,\infty$$

where

$$x \odot xs = x \downarrow (\mathsf{max}\,xs)$$

Recalling the definition $\mathsf{max} = \mathsf{foldl}(\uparrow)\,(-\infty)$, and using the fact that $\downarrow$ distributes through $\uparrow$, we can rewrite the definition of $\odot$ in the following way:

$$\begin{aligned} x \odot xs &= \mathsf{max}\,(\mathsf{map}\,(x\downarrow)\,xs) \\ &= (\mathsf{max} \cdot \mathsf{map}\,(x\downarrow))\,xs \end{aligned}$$

The right-hand side of this equation is also a candidate for the fold-map fusion law. We therefore obtain

$$x \odot xs = \mathsf{foldl}\,(\oplus_x)\,(-\infty)\,xs$$

where

$$u \oplus_x v = u \uparrow (x \downarrow v)$$

The hard work has now been done. It only remains to observe that x is a left-zero of $\oplus_x$, that is,

$$x \uparrow (x \downarrow v) = x$$

for all v. This result means we can implement minimax by the following program:

$$\begin{aligned} \mathsf{minimax} &= \mathsf{foldl}\,(\odot)\,\infty \\ x \odot xs &= \mathsf{fastfoldl}\,(\oplus)\,(=x)\,(-\infty)\,xs \\ &\quad \text{where } u \oplus v = u \uparrow (x \downarrow v) \end{aligned}$$

The new definition of minimax captures the essential idea behind the alphabeta algorithm as derived in Ref. 5.

5. SCAN

Corresponding to foldr and foldl are two further higher-order functions, which we shall call scanr and scanl. Since the theory of foldr and scanr is dual to that of foldl and scanl we shall concentrate only on the latter. Informally, we have

$$\mathsf{scanl}\,(\oplus)\,a[x_1\,x_2,\ldots,x_n] = [a, a \oplus x_1, (a \oplus x_1) \oplus x_2, \ldots]$$

Thus scanl applies a foldl operation to every initial segment of a list and produces a list of results. In particular, the last element is just the value of the foldl operation on the whole list, so we have

$$\mathsf{foldl}\,(\oplus)\,a = \mathsf{last} \cdot \mathsf{scanl}\,(\oplus)\,a$$

Formally, we can define

$$\begin{aligned} \mathsf{scanl}\,(\oplus)\,a[\,] &= [a] \\ \mathsf{scanl}\,(\oplus)\,a(x{:}xs) &= [a] \mathbin{+\!\!+} \mathsf{scanl}\,(\oplus)\,(a \oplus x)\,xs \end{aligned}$$

The important point here is that each element in a scanl can be computed in terms of the preceding element using just one extra $\oplus$ operation. Thus it requires just n evaluations of $\oplus$ to compute the value of scanl on a list of length n.

The most important identity concerning the function scanl is called the *scan lemma*. Let inits be the function that returns the list of initial segments of a list, in increasing order of length. Thus, we have

$$\mathsf{inits}\,[x_1, x_2, \ldots, x_n] = [[\,], [x_1], [[x_1, x_2], \ldots]$$

The scan lemma says that

$$\mathsf{map}\,(\mathsf{foldl}\,(\oplus)\,a) \cdot \mathsf{inits} = \mathsf{scanl}\,(\oplus)\,a \qquad (12)$$

The usefulness of this lemma lies not in the relationship it expresses (which is obvious given the informal definition of scanl), but in the relative efficiency of the two sides of the equation. Evaluating the expression on the left for a list of length n requires $O(n^2)$ operations involving $\oplus$, while the expression on the right requires only $O(n)$ such operations. Therefore, when applied from left to right, the scan lemma ensures an order-of-magnitude increase in efficiency.

There are a number of other identities concerning scanl of which we cite just one. The *fold-scan fusion* law says that

$$\mathsf{foldl}\,(\oplus)\,a \cdot \mathsf{scanl}\,(\otimes)\,b = \mathsf{fst} \cdot \mathsf{foldl}\,(\odot)\,(a \oplus b, b) \qquad (13)$$

where fst is the function $\mathsf{fst}\,(a, b) = a$ and the operation $\odot$ is defined by the equation

$$(u, v) \odot x = (u \oplus w, w) \quad \text{where} \quad w = v \otimes x$$

This is another example of a loop-fusion law that shows when two traversals of a list can be combined into one. We shall give an example of the fold-scan fusion law in Section 8.

6. SEGMENTS

The function inits, which returns the initial segments of a list, was introduced above. The function tails, which returns the final segments, is similar:

$$\mathsf{tails}\,[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_n], \ldots, [x_n], [\,]]$$

Note that tails returns the final segments in decreasing order of length.

The functions inits and tails can be characterised formally in a number of ways, one of which is by a scan operation. For example, we have

$$\mathsf{inits} = \mathsf{scanl}\,(\oplus)\,[\,]$$

where

$$xs \oplus x = xs \mathbin{+\!\!+} [x]$$

A recursive characterisation of tails, useful in proofs by induction, is:

$$\mathsf{tails}\,[\,] = [[\,]] \qquad (14)$$

$$\mathsf{tails}\,(xs \mathbin{+\!\!+} [x]) = \mathsf{map}\,(\mathbin{+\!\!+}\,[x])\,(\mathsf{tails}\,xs) \mathbin{+\!\!+} [[\,]] \qquad (15)$$

Here the function $\mathsf{map}\,(\mathbin{+\!\!+}\,[x])$ appends x to the end of every element in a list of lists.

The function segs returns a list of *all* segments of a given list. One formal definition of segs is

$$\mathsf{segs} = \mathsf{concat}\cdot\mathsf{map\,tails}\cdot\mathsf{inits}$$

This definition expresses the process of taking all the final segments of each initial segment, and concatenating the results. For example,

$$\begin{aligned}&\mathsf{segs}\,[1,2,3]\\&\quad=[[\,],[1],[\,],[1,2],[2],[\,],[1,2,3],[2,3],[3],[\,]]\end{aligned}$$

The empty segment appears four times in this list, since there are four initial segments of [1, 2, 3].

There are other definitions of segs, but the above is a particularly convenient one to work with. We shall see an example of its use in Section 8.

7. HORNER'S RULE

Consider the following algebraic identity:

$$\begin{aligned}&(x_1\times x_2\times x_3)+(x_2\times x_3)+x_3+1\\&\qquad=((1\times x_1+1)\times x_2+1)\times x_3+1\end{aligned}$$

The equation generalises to an arbitrary list $[x_1, x_2, \ldots, x_n]$ of numbers and we will refer to it as *Horner's rule*. Expressed in terms of the functions we have introduced above, we can write Horner's rule in the form

$$\mathsf{sum}\cdot\mathsf{map\,product}\cdot\mathsf{tails} = \mathsf{foldl}\,(\odot)\,1$$

where $\odot$ is defined by

$$x\odot y = (x\times y)+1$$

The usefulness of Horner's rule turns on the fact that it can be generalised. The general formulation of Horner's rule is as follows.

Lemma 1
Suppose that $\oplus$ and $\otimes$ are two operators such that $\otimes$ distributes (backwards) through $\oplus$, that is,

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

Suppose further that a is a left-identity of $\oplus$. Then

$$\mathsf{foldl}\,(\oplus)\,a\cdot\mathsf{map}\,(\mathsf{foldl}\,(\otimes)\,b)\cdot\mathsf{tails} = \mathsf{foldl}\,(\odot)\,b \qquad (16)$$

where $\odot$ is defined by the equation

$$x\odot y = (x\otimes y)\oplus b$$

Proof
Abbreviating the left-hand side of (16) by lhs, we shall show that

$$\begin{aligned}\mathsf{lhs}\,[\,] &= b\\ \mathsf{lhs}\,(xs \mathbin{+\!\!+} [x]) &= (\mathsf{lhs}\,xs)\odot x\end{aligned}$$

from which we can conclude that

$$\mathsf{lhs} = \mathsf{foldl}\,(\odot)\,b$$

by the characterisation of foldl discussed in Section 2.

To shorten the proof, it is convenient to introduce two abbreviations:

$$\mathsf{sum} = \mathsf{foldl}\,(\oplus)\,a$$

$$\mathsf{prod} = \mathsf{foldl}\,(\otimes)\,b$$

In particular, it follows from the definition of prod and equation (6) that:

$$\mathsf{prod}\cdot(\mathbin{+\!\!+}\,[x]) = (\otimes\,x)\cdot\mathsf{prod} \qquad (17)$$

This fact is used below.

We will spell out the proof in some detail, with justification for each step (given before the step itself).

Case [ ]

$$\begin{aligned}\mathsf{lhs}\,[\,] &= \text{definition of composition}\\ &\quad\ \mathsf{sum}\,(\mathsf{map\,prod}\,(\mathsf{tails}\,[\,]))\\ &= \text{equation (14)}\\ &\quad\ \mathsf{sum}\,(\mathsf{map\,prod}\,[[\,]])\\ &= \text{equations (1) and (2)}\\ &\quad\ \mathsf{sum}\,[\mathsf{prod}\,[\,]]\\ &= \text{definition of prod and equation (5)}\\ &\quad\ \mathsf{sum}\,[b]\\ &= \text{definition of sum and equations (5) and (6)}\\ &\quad\ a\oplus b\\ &= \text{assumption that a is a left-identity of }\oplus\\ &\quad\ b\end{aligned}$$

completing the case.

Case $xs \mathbin{+\!\!+} [x]$

$$\begin{aligned}&\mathsf{lhs}\,(xs \mathbin{+\!\!+} [x])\\ &= \text{definition of composition}\\ &\quad\ \mathsf{sum}\,(\mathsf{map\,prod}\,(\mathsf{tails}\,(xs \mathbin{+\!\!+} [x])))\\ &= \text{equation (15)}\\ &\quad\ \mathsf{sum}\,(\mathsf{map\,prod}\,(\mathsf{map}\,(\mathbin{+\!\!+}\,[x])\,(\mathsf{tails}\,xs) \mathbin{+\!\!+} [[\,]]))\\ &= \text{equation (2)}\\ &\quad\ \mathsf{sum}\,((\mathsf{map\,prod}\,(\mathsf{map}\,(\mathbin{+\!\!+}\,[x])\,(\mathsf{tails}\,xs)))\\ &\quad\ \mathbin{+\!\!+} [\mathsf{prod}\,[\,]])\\ &= \text{equation (5)}\\ &\quad\ \mathsf{sum}\,((\mathsf{map\,prod}\,(\mathsf{map}\,(\mathbin{+\!\!+}\,[x])\,(\mathsf{tails}\,xs)) \mathbin{+\!\!+} [b])\\ &= \text{definition of sum and equation (6)}\\ &\quad\ (\mathsf{sum}\,(\mathsf{map\,prod}\,(\mathsf{map}\,(\mathbin{+\!\!+}\,[x])\,(\mathsf{tails}\,xs))))\oplus b\\ &= \text{map distribution law (3)}\\ &\quad\ (\mathsf{sum}\,(\mathsf{map}\,(\mathsf{prod}\cdot(\mathbin{+\!\!+}\,[x]))\,(\mathsf{tails}\,xs)))\oplus b\\ &= \text{equation (17)}\\ &\quad\ (\mathsf{sum}\,(\mathsf{map}\,((\otimes\,x)\cdot\mathsf{prod})\,(\mathsf{tails}\,xs)))\oplus b\\ &= \text{map distribution law (3)}\\ &\quad\ (\mathsf{sum}\,(\mathsf{map}\,(\otimes x)\,(\mathsf{map\,prod}\,(\mathsf{tails}\,xs))))\oplus b\\ &= \text{lemma; see below}\\ &\quad\ ((\mathsf{sum}\,(\mathsf{map\,prod}\,(\mathsf{tails}\,xs)))\otimes x)\oplus b\\ &= \text{definition of lhs}\\ &\quad\ ((\mathsf{lhs}\,xs)\otimes x)\oplus b\\ &= \text{definition of }\odot\\ &\quad\ (\mathsf{lhs}\,xs)\odot x\end{aligned}$$

completing the case

The subsidiary lemma used in this calculation is as follows:

$$\mathsf{sum}\,(\mathsf{map}\,(\otimes x)\,ys) = (\mathsf{sum}\,ys)\otimes x$$

for all *nonempty* finite lists ys. Again, the proof is by induction.

Case [y]

$$\begin{aligned}\mathsf{sum}\,(\mathsf{map}\,(\otimes\,x)\,[y]) &= \text{equations (1) and (2)}\\ &\quad\ \mathsf{sum}\,[y\otimes x]\\ &= \text{definition of sum}\\ &\quad\ a\oplus(y\otimes x)\\ &= \text{a is a left-identity of }\oplus\\ &\quad\ (a\oplus y)\otimes x\\ &= \text{definition of sum}\\ &\quad\ (\mathsf{sum}\,[y])\otimes x\end{aligned}$$

completing the case.

Case ys ++ [y], for non-empty ys

```
    sum (map (⊗x) (ys ++ [y]))
  = equation (2)
    sum ((map (⊗ x) ys) ++ [y ⊗ x])
  = definition of sum and equation (6)
    (sum (map (⊗ x) ys)) ⊕ (y ⊗ x)
  = induction hypothesis
    ((sum ys) ⊗ x) ⊕ (y ⊗ x)
  = hypothesis that ⊗ distributes through ⊕
    ((sum ys) ⊕ y) ⊗ x
  = definition of sum and equation (6)
    (sum (ys ++ [y]) ⊗ x
```

completing the case, and the proof of Horner's rule.

The above proof is fairly unattractive. Not only is there a need to introduce abbreviations to make expressions more readable, a need that would not arise if we used a more succinct notation, there are also many more brackets than one can reasonably handle. The reason is that proofs by induction require us to formulate expressions in which functions are actually *applied* to arguments. A style of expression that relies on functional composition (·) rather than functional application, is more sparing in its use of brackets and therefore preferable.

Let us now proceed to an application of Horner's rule.

8. MAXIMUM SEGMENT SUM

The maximum segment sum problem is a famous one and its history is described in J. Bentley's *Programming Pearls*.[2] The problem is to compute the maximum of the sums of all segments of a list of arbitrary numbers. In symbols, the problem is to compute

mss = max · map sum · segs

Direct evaluation of mss from this definition requires $O(n^3)$ steps for a list of length n. There are $O(n^2)$ segments, and each can be summed in a further $O(n)$ steps, giving $O(n^3)$ steps in total. Using Horner's rule it is easy to calculate an $O(n)$ algorithm:

```
mss = definition
        max · map sum · segs
    = definition of segs
        max · map sum · concat · map tails · inits
    = map promotion (7)
        max · concat · map(map sum) · map tails · inits
    = definition of max and fold promotion (8)
        max · map max · map (map sum) · map tails · inits
    = map distributivity (3)
        max · map (max · map sum · tails) · inits
    = Horner's rule, with x ⊗ y = (x+y)↑0
        max · map (foldl (⊗) 0) · inits
    = scan lemma (12)
        max · scanl (⊗) 0
    = fold-scan fusion (13)
        fst · foldl (⊙) (0, 0)
```

where ⊙ is defined by

(u, v) ⊙ x = (u↑w, w)   where   w = (v+x)↑0

Horner's rule is applicable since + distributes over ↑. The result of the above calculation is a linear time algorithm for mss.

REFERENCES

1. J. Backus. Can programming be liberated from the Von Neuman style? A functional style and its algebra of programs. *Commun. ACM* **21** (8), 613–641 (1978)
2. J. L. Bentley, *Programming Pearls*. Addison-Wesley, Reading, Mass. (1986).
3. R. S. Bird, Lectures on constructive functional programming. *NATO Advanced Study Institute on Constructive Methods in Computing Science*, Marktoberdorf, F.R.G., August 1988.
4. R. S. Bird, Introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, edited M. Broy. NATO ASI Series F, vol. 36. Springer-Verlag, (1987).
5. R. S. Bird and R. J. M. Hughes, The alpha-beta algorithm: an exercise in program transformation. *Information Processing Letters* **24**, 53–57 (1987).
6. R. S. Bird and P. L. Wadler, *Introduction to Functional Programming*. Prentice-Hall International, Hemel Hempstead (1988).
7. J. Darlington, Program transformation. In *Functional Programming and its Applications*, edited J. Darlington, P. Henderson and D. A. Turner. Cambridge University Press (1982).
8. D. Turner, An overview of Miranda. *SIGPLAN Notices* (December 1986).
9. P. Wadler, Applicative style programming, program transformation, and list operators. *Report of ACM Conference on Functional Programming Languages and Computer Architecture, Wentworth, NH, USA (1981)*, pp. 25–32.