AN INTRODUCTION

TO THE

THEORY OF LISTS

by

Richard S. Bird

Technical Monograph PRG-56

October 1986

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

An Introduction to the Theory of Lists

Richard S. Bird

Programming Research Group,
University of Oxford,
11 Keble Rd.,
Oxford OX1 3QD,
United Kingdom.

Abstract

In these lectures we introduce a notation and a calculus for specifying and manipulating computable functions over lists. The calculus is used to derive efficient solutions for a number of problems, including problems in text processing. Although programming *per se* is not the main topic, we indicate briefly how these solutions can be implemented in a purely functional programming language.

Acknowledgements

Much of the theory presented in these lectures was developed in collaboration with L.G.L.T. Meertens of the Centrum voor Wiskunde en Informatica, Amsterdam. The influence of David Turner's work on the development of notation for functional programming has also been substantial. A particular debt of gratitude is owed to Phil Wadler who contributed a great deal in the way of ideas and examples. The work was supported by a grant from the Science and Engineering Research Council of Great Britain.

1. Elementary Operations

1.1 List Notation. A *list* is a linearly ordered collection of values of the same general nature; one can talk about the first element of a list, the second element, and so on. Lists are also called *sequences*, a term more commonly found in other branches of mathematics, but there is no difference between the concepts and we shall use the two words interchangeably.

A finite list is denoted using square brackets and commas. For example, $[1, 2, 3]$ is a list of three integers and $[[\text{'b'}, \text{'y'}, \text{'e'}], [\text{'b'}, \text{'y'}, \text{'e'}]]$ is a list of two elements, each element being a list of characters. The empty list is written as $[\,]$ and a singleton list, containing just one element a, is written as $[a]$. In particular, $[[\,]]$ is a singleton list containing the empty list as its only element. Lists can be infinite as well as finite, but in these lectures we shall consider only finite lists.

Unlike a set, a list may contain the same value more than once. For example, $[1, 1]$ is a list of two elements, both of which happen to be 1, and is distinct from the list $[1]$ which contains only one element.

The special form $[m \ldots n]$ will be used to denote the list of integers in increasing order from m to n inclusive. If $m > n$, then $[m \ldots n] = [\,]$.

It was stated at the outset that lists are collections of values of the same general nature. What this means is that we can have lists of numbers, lists of characters, even lists of functions; but we shall never mix two distinct kinds of value in the same list. Given this restriction, the kind (or *type*) of list under consideration can be described in a simple manner. A list of numbers will be assigned the type $[Num]$ (read as: *list of Num*); a list of characters will be assigned the type $[Char]$, and so on. For example, $[[Num]]$ describes the type of lists of lists of numbers, and $[A \to B]$ describes the type of lists of functions from A to B. It is useful to extend this notation and write $[A]^+$ to denote the *non-empty* lists whose elements are of type A.

In order that the above convention for naming types should work satisfactorily, it is necessary to allow type expressions to contain *type variables*. To illustrate why, consider the empty list $[\,]$. As the empty list is empty of all conceivable values, it possesses the type $[Num]$, $[Char]$, as well as infinitely many others. The resolution of this situation is to assign $[\,]$ the type $[\alpha]$, where α is a type variable. To assert that a value has type $[\alpha]$ is to say that it has type $[A]$ for every possible type A. In this sense, the concept of a type variable is just a convenient abstraction for describing universal quantification over types. The device is also useful for describing *generic* functions. For example, the function *id*, where $id\ x = x$, possesses the type

$\alpha \to \alpha$. Further examples will be seen below.

A general comment on our typographical conventions should be made at this point. We shall use letters a, b, c, ..., at the beginning of the alphabet to denote elements of lists, and letters x, y, z at the end of the alphabet to denote the lists themselves. On some occasions we shall want to emphasise that a particular list is, in fact, a list of lists. Compound symbols xs, ys and zs will be used to denote lists which contain lists as elements. The names of functions will be written in italics, while infix operators will be written using special symbols of various kinds.

Having covered most of the special notation, we shall now introduce a small number of useful functions and operators for manipulating lists. They will be described informally: precise definitions will be given later when the necessary machinery has been developed.

1.2 Length. The length of a finite list is the number of elements it contains. We denote this operation by the operator #. Thus,

$$\#[a_1, a_2, ..., a_n] = n.$$

In particular, $\#[\,] = 0$. For $m \leq n$ we have that

$$\#[m \,..\, n] = n - m + 1.$$

The type of # is given by

$$\# : [\alpha] \to Num.$$

The operator # takes a list, the nature of whose elements is irrelevant, and returns a (nonnegative, integer) number; hence the above type assignment.

1.3 Concatenation. Two lists can be concatenated together to form one longer list. This function is denoted by the operator $+\!\!+$ (pronounced "concatenate"). Thus,

$$[a_1, a_2, ..., a_n] +\!\!+ [b_1, b_2, ..., b_m] = [a_1, a_2, ..., a_n, b_1, b_2, ..., b_m].$$

In particular, we have

$$[\,] +\!\!+ x = x +\!\!+ [\,] = x$$

for all lists x, so the empty list is the *identity element* of the operator $+\!\!+$. Concatenation is also *associative*; we have

$$x +\!\!+ (y +\!\!+ z) = (x +\!\!+ y) +\!\!+ z$$

for all lists x, y and z.

A simple relationship between $\#$ and $+\!\!+$ is given by the equation

$$\#(x +\!\!+ y) = \#x + \#y$$

for all finite lists x and y.

Finally, the type of $+\!\!+$ is given by

$$+\!\!+ : [\alpha] \times [\alpha] \to [\alpha].$$

Concatenation takes a pair of lists, both of the same kind, and produces a third list, again of the same kind; hence the type assignment.

1.4 Map. The operator $*$ (pronounced "map") applies a function to each element of a list. We have

$$f * [a_1, a_2, \ldots, a_n] = [f\ a_1, f\ a_2, \ldots, f\ a_n].$$

In particular, $f\ [\,] = [\,]$. The type of $*$ is given by

$$* : (\alpha \to \beta) \times [\alpha] \to [\beta].$$

Hence, in the expression $f * x$, the first argument f is a function with type $\alpha \to \beta$, and the second argument x is a list with type $[\alpha]$. The result is a list of type $[\beta]$. These type variables can be instantiated to specific types. For example, if $even : Num \to Bool$ is the predicate which determines whether a number is even, then

$$even * [1 \ldots 4] = [false, true, false, true],$$

has type $[Bool]$. Here, *true* and *false* denote the two constants of type *Bool*.

As with other infix operators, the operator $*$ is allowed to appear in expressions accompanied by only one of its arguments. In particular, we can write $(f*)$ to denote the function of type $[\alpha] \to [\beta]$, where $f : \alpha \to \beta$, which takes a list and applies f to every element. By the same convention, $((f*)*)$ is a function which takes a list of lists and applies $(f*)$ to every element.

There are a number of important identities concerning $*$. First of all, $*$ distributes through $+\!\!+$; for all lists x and y we have

$$f * (x +\!\!+ y) = (f * x) +\!\!+ (f * y).$$

Second, $*$ distributes (backwards) through functional composition:

$$(f \cdot g)* = (f*) \cdot (g*).$$

We shall encounter many applications of these two identities in due course. Another rule is that if f is an injective function with inverse f^{-1}, then

$$(f*)^{-1} = (f^{-1}*).$$

1.5 Filter. The operator $\triangleleft$ (pronounced "filter") takes a predicate p and a list x and returns the list of elements of x which satisfy p. For example, we have

$$even \triangleleft [1 \ldots 10] = [2, 4, 6, 8, 10].$$

The type of $\triangleleft$ is given by

$$\triangleleft : (\alpha \to Bool) \times [\alpha] \to [\alpha].$$

Like $*$, the operator $\triangleleft$ distibutes through $+\!\!+$: for all lists x and y we have

$$p \triangleleft (x +\!\!+ y) = (p \triangleleft x) +\!\!+ (p \triangleleft y).$$

We also have the laws

$$\begin{array}{lcl} p \triangleleft q \triangleleft x & = & q \triangleleft p \triangleleft x \\ p \triangleleft p \triangleleft x & = & p \triangleleft x \\ p \triangleleft f * x & = & f * (p \cdot f) \triangleleft x, \end{array}$$

for all functions p, q and f and lists x. The first law (commutativity of filters) says that filtering a list with a (total) predicate q, and then filtering the result with a (total) predicate p, gives the same answer as first filtering with p and then with q. The second law says that $(p \triangleleft)$ is an idempotent operation. The third law (commutativity of map and filter) says that mapping with f followed by filtering with p gives the same result as first filtering with $p \cdot f$ and then mapping with f. We can also express these laws using functional composition:

$$\begin{array}{lcl} (p \triangleleft) \cdot (q \triangleleft) & = & (q \triangleleft) \cdot (p \triangleleft) \\ (p \triangleleft) \cdot (p \triangleleft) & = & (p \triangleleft) \\ (p \triangleleft) \cdot (f*) & = & (f*) \cdot ((p \cdot f) \triangleleft). \end{array}$$

1.6 Operator precedence. In addition to the above operators we have also encountered, without explicitly mentioning the fact, the operation of

functional application. Functional application is denoted by just a space in formulae, and when no confusion can arise the space is sometimes omitted. Thus $f\,a$ means f "applied to" a. Application associates to the left, so that $f\,a\,b$ means $(f\,a)\,b$ and not $f\,(a\,b)$.

It is normal in mathematical notation which deploys a number of infix operators to provide certain rules of precedence and association in order to reduce the number of brackets. We shall suppose that functional application is more binding than any other operator, so $f\,x +\!\!+\, y$ means $(f\,x) +\!\!+\, y$ and not $f(x +\!\!+\, y)$. It is also convenient to suppose that $+\!\!+$ has a low precedence, so $f * x +\!\!+\, g * y$ means $(f * x) +\!\!+\, (g * y)$ and not $f * (x +\!\!+\, g * y)$. For the other operators we shall put in brackets to clarify meaning. However, we shall assume that, in the absence of brackets, operators associate to the right in expressions. For example, $f * p \triangleleft x$ means $f * (p \triangleleft x)$ and not $(f * p) \triangleleft x$.

2. Reduction

2.1 The reduction operators. Most of the operations introduced in the first section transform lists into other lists. The reduction operators to be described in the present and following section are more general in that they can convert lists into other kinds of value as well.

The first reduction operator, written "/" and pronounced "reduce", takes an operator $\oplus$ on the left and a list x on the right. Its effect is to insert $\oplus$ between adjacent elements of x. Thus:

$$\oplus/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n.$$

For the right-hand side of this equation to be unambiguous in the absence of brackets, the operator $\oplus$ must be associative. In fact, the form $\oplus/x$ is only permitted when $\oplus$ is an associative operator, so the grouping of terms on the right is irrelevant.

In the case that the second argument of / is a singleton list $[a]$, we have from the informal description of $\oplus/$ that

$$\oplus/[a] = a.$$

Moreover, we also have

$$\oplus/(x +\!\!+\, y) = (\oplus/x) \oplus (\oplus/y)$$

whenever x and y are non-empty lists. These two equations (the definition of / on singletons and the distributive law) are important and will be used frequently in what follows.

The informal definition of / given above does not prescribe a meaning for the expression $\oplus/[\,]$. If $\oplus$ has an identity element e, then we suppose $\oplus/[\,] = e$; otherwise $\oplus/[\,]$ is not defined. The reason for this choice is to preserve the distributive law when either x or y is the empty list. Since

$$a = \oplus/[a] = \oplus/([a] +\!\!+ [\,]) = (\oplus/[a]) \oplus (\oplus/[\,]) = a \oplus e,$$

and also

$$a = \oplus/[a] = \oplus/([\,] +\!\!+ [a]) = (\oplus/[\,]) \oplus (\oplus/[a]) = e \oplus a,$$

it follows that, if defined, $\oplus/[\,]$ must be both a left and right identity element of $\oplus$. Hence, $\oplus/[\,]$ can only be the (unique) identity element of $\oplus$.

The type of / is given by

$$/ : (\alpha \times \alpha \to \alpha) \times [\alpha] \to \alpha.$$

Thus, in the combination $\oplus/x$, the operator $\oplus$ has a type of the form $\alpha \times \alpha \to \alpha$ and x has a type of the form $[\alpha]$. The combination will then have type α.

Some simple cases of reduction, indicative of the general utility of the operator, are given in the following definitions:

$$\begin{aligned}
\mathit{sum} &= +/ \\
\mathit{product} &= \times/ \\
\mathit{flatten} &= +\!\!+/ \\
\mathit{all}\ p &= (\wedge/) \cdot (p*) \\
\mathit{some}\ p &= (\vee/) \cdot (p*) \\
\mathit{min} &= \downarrow/ \\
\mathit{max} &= \uparrow/
\end{aligned}$$

All the operators involved in these definitions are associative and all, except the last two, have identity elements. The identity element for $+$ is 0, so $\mathit{sum}\,[\,] = 0$; for multiplication the identity element is 1, so $\mathit{product}\,[\,] = 1$, and so on. The expressions $+/$ and $\times/$ correspond to the special symbols $\sum$ and $\prod$ used in other branches of mathematics. For example,

$$\begin{aligned}
\textstyle\sum_{j=1}^{n} f_j &= +/f*[1\ldots n] \\
\textstyle\prod_{j=1}^{n} j &= \times/[1\ldots n].
\end{aligned}$$

The function *flatten* takes a list of lists and concatenates them to form a single list. Since $[\,]$ is the identity element of $+\!\!+$ we have $+\!\!+/[\,] = [\,]$.

The binary operators $\wedge$ and $\vee$ denote the operations of logical conjunction and disjunction respectively. Accordingly, the function *all* takes a predicate p and a list x and returns the value *true* if all the elements of x satisfy p, and *false* otherwise. For example,

$$all\ even\ [2,4,6] = true.$$

The function *some* takes similar arguments p and x and returns *true* if at least one of the elements of x satisfies p, and *false* otherwise. For example,

$$some\ (=1)\ [2,4,6] = false.$$

Since

$$\begin{array}{rcccl} a \wedge true & = & true \wedge a & = & a \\ a \vee false & = & false \vee a & = & a, \end{array}$$

it follows that $all\ p\ [\,] = true$ and $some\ p\ [\,] = false$ for all predicates p.

The operators $\uparrow$ and $\downarrow$ select the greater and lesser of their two (numerical) arguments respectively. Hence *max* selects the maximum of a list of numbers and *min* selects the minimum. These two operators are considered further below.

2.2 Fictitious values. Neither $\uparrow$ nor $\downarrow$ have identity elements in the domain of finite numbers, so both $max\ [\,]$ and $min\ [\,]$ are undefined. Despite this, it is often useful to be able to manipulate expressions involving terms of the form $\uparrow/x$ and $\downarrow/x$ without taking special precautions to ensure $x \neq [\,]$. The same holds for other operators without identity elements. Provided certain rules are observed, we can always invent a "fictitious" value to act as an identity element of a given operator. Suppose $\oplus$ is an associative operator, defined over some domain X but not possessing an identity element. Invent a new value e and adjoin it to X. Define $\oplus'$ by the rules

$$\begin{array}{rcll} a \oplus' b & = & b, & \text{if } a = e \\ & = & a, & \text{if } b = e \\ & = & a \oplus b, & \text{otherwise.} \end{array}$$

The new operator $\oplus'$ is associative, has identity element e, and agrees with $\oplus$ on arguments in X. For example, we can invent fictitious values $\uparrow'/[\,]$ and $\downarrow'/[\,]$ — calling them $-\infty$ and ∞ say — and adjoin them to the domain of finite numbers. As long as no other properties of these fictitious elements are assumed, we can continue to use the undecorated operators in expressions

and derivations. Care must be exercised to avoid imputing any additional laws to the new values. For example, the law

$$\downarrow/(a+) * x = a+ \downarrow/x$$

is only valid when restricted to the case $x \neq [\,]$.

To give another illustration of a useful operator which has no identity element, define $\ll$ by

$$a \ll b = a.$$

Since

$$(a \ll b) \ll c = a = a \ll (b \ll c),$$

the operator $\ll$ is associative, and we can form $\ll/x$. The value of this expression is the first element of the list x. This is a useful operation and we define

$$head\ x = \ll/x.$$

(A similarly useful operation is

$$last\ x = \gg/x,$$

where $a \gg b = b$.) The operator $\ll$ does not possess an identity element. If e were an identity element, then we would have $e \ll a = a$ for all a; but since $e \ll a = e$ by the definition of $\ll$, the conclusion would be that $e = a$ for all a. If necessary, we can invent a fictitious value e and define

$$\begin{aligned} a \ll' b &= b, \quad \text{if } a = e \\ &= a, \quad \text{otherwise.} \end{aligned}$$

The function $\ll'$ agrees with $\ll$ on non-fictitious arguments.

2.3 Homomorphisms. There is a close relationship between reductions and homomorphisms on sequences. By definition, a function h defined on finite lists is a homomorphism if there exists an associative operator $\oplus$ with identity element e such that $h\,[\,] = e$ and

$$h(x +\!\!+ y) = h\,x \oplus h\,y$$

for all lists x and y. If h is not defined on the empty list, then $\oplus$ is not required to possess an identity element and the above equation is asserted for non-empty lists only.

If h is a homomorphism, then h is uniquely determined by $\oplus$ and the values of h on singleton sequences. In other words, if we define f by the equation

$$f\,a = h\,[a],$$

then h is determined by $\oplus$ and f alone. The following lemma says that every homomorphism can be expressed as the composition of a reduction and a map, and every such composition is a homomorphism.

Lemma 1 [Homomorphism Lemma]. *A function h is a homomorphism with respect to $+\!\!+$ if and only if $h = (\oplus/) \cdot (f*)$ for some operator $\oplus$ and function f.*

Proof. First, suppose $h = (\oplus/) \cdot (f*)$. Then

$$\begin{aligned} h\,(x +\!\!+ y) &= \oplus/f * (x +\!\!+ y) \\ &= \oplus/((f * x) +\!\!+ (f * y)) \\ &= (\oplus/f * x) \oplus (\oplus/f * y) \\ &= h\,x \oplus h\,y, \end{aligned}$$

using the distributive laws for $*$ and $/$. Furthermore, if $\oplus$ has an identity element e, then

$$h\,[\,] = \oplus/f * [\,] = \oplus/[\,] = e.$$

Hence h is a homomorphism.

To prove the converse, suppose h is a homomorphism, so that

$$h\,(x +\!\!+ y) = h\,x \oplus h\,y$$

for some operator $\oplus$. Define f by the equation

$$f\,a = h\,[a].$$

We show $h = (\oplus/) \cdot (f*)$ by induction on the length of sequences.

If $h\,[\,]$ is defined, then it is the identity element of $\oplus$ and so

$$h\,[\,] = \oplus/[\,] = \oplus/f * [\,].$$

If $\#x = 1$, that is, $x = [a]$ for some a, then we have

$$h\,[a] = f\,a = \oplus/[f\,a] = \oplus/f * [a],$$

using the definition of $/$ on singletons and the definition of $*$. In the case $\#x = n$, where $n > 1$, we can set $x = y +\!\!+ z$, where $1 \leq \#y, \#z < n$. By

induction, we can suppose $h\ y = \oplus/f * y$ and $h\ z = \oplus/f * z$ and hence compute

$$\begin{aligned} h(y +\!\!+ z) &= h\ y \oplus h\ z \\ &= (\oplus/f * y) \oplus (\oplus/f * z) \\ &= \oplus/f * (y +\!\!+ z), \end{aligned}$$

using the distributive laws for $*$ and $/$ as before. This completes the proof.

2.4 Definition by homomorphisms. Many of the functions already introduced are homomorphisms. A reduction itself is a homomorphism and so is a map. We have

$$\oplus/ = (\oplus/) \cdot (id*)$$

where *id* is the identity function, and

$$f* = (+\!\!+/) \cdot (g*)$$

where g is the function defined by $g\ a = [f\ a]$.

A filter is also a homomorphism. We have

$$p \triangleleft = (+\!\!+/) \cdot (f_p*),$$

where the function f_p is defined by $f_p\ a = [a]$ if $p\ a$ and $f_p\ a = [\,]$ otherwise. This function replaces elements which satisfy p by singleton lists and others by the empty list. The filtered sequence can then be obtained by concatenating these lists together.

The length operator can be defined as the homomorphism

$$\# = (+/) \cdot (K_1*),$$

where $K_c\ a = c$ for all a. Every element of the list is therefore replaced by 1 and the result is summed to give the length.

The functions *head* and *last* are homomorphisms:

$$\begin{aligned} head &= (\ll/) \cdot (id*) \\ last &= (\gg/) \cdot (id*), \end{aligned}$$

where $a \ll b = a$ and $a \gg b = b$.

Of course, not all functions on lists are homomorphisms. One useful sufficient condition is that h is *injective*, i.e. $h(x) = h(y)$ if and only if $x = y$. If h is injective, then its inverse h^{-1} is well-defined on the range of h. Thus, if we define $\oplus$ by

$$u \oplus v = h(h^{-1}\ u +\!\!+ h^{-1}\ v),$$

then it follows that

$$\begin{aligned} h(x +\!\!+ y) &= h(h^{-1}(h\,x) +\!\!+ h^{-1}(h\,y)) \\ &= h\,x \oplus h\,y \end{aligned}$$

Hence h is the homomorphism $(\oplus/)\cdot(f*)$, where, as usual, $f\,a = h\,[a]$.

A simple application of this result is given by the function *reverse* which reverses the order of the elements in a list. Clearly, *reverse* is injective and is its own inverse. Hence

$$reverse = (\oplus/)\cdot(f*),$$

where

$$x \oplus y = reverse(reverse\ x +\!\!+ reverse\ y).$$

An informal argument, left to the reader, shows this last expression is equal to $(y +\!\!+ x)$. By convention, let $\oplus$ and $\tilde{\oplus}$ be related by the equation

$$x\,\tilde{\oplus}\,y = y \oplus x.$$

Also, let the special symbol $\Box$ denote the function which transforms values into singleton lists so that $\Box a = [a]$ for all a. Then we can write

$$reverse = (\widetilde{+\!\!+}/)\cdot(\Box*)$$

We turn now to a more advanced application of the same idea.

2.5 Example: processing text. Suppose we define a *text* to be a list of characters and a *line* to be a list of characters not containing the newline character NL. These classes can be introduced as new types:

$$\begin{aligned} Text &= [Char] \\ Line &= [Char \setminus \{\mathrm{NL}\}]. \end{aligned}$$

In this section we want to define a function *lines* which takes a text and returns the list of lines that make up the text. The function *lines* is an important component in many text-processing applications. It can be specified formally as the inverse of another function, *unlines* say, which inserts newline characters between adjacent lines and then concatenates the result. The definition of *unlines* is as a reduction:

$$\begin{aligned} unlines &= \oplus/ \\ x \oplus y &= x +\!\!+ [\mathrm{NL}] +\!\!+ y. \end{aligned}$$

The operator $\oplus$ does not have an identity element, so the value of *unlines* [ ] is not defined. We therefore assign *unlines* the type

$$unlines : [Line]^+ \rightarrow Text,$$

where $[X]^+$ denotes the non-empty members of $[X]$. It is easy to verify that *unlines* is injective. This means *lines* can be completely specified by the single equation

$$lines(\oplus/xs) = xs \tag{1}$$

for all non-empty sequences of lines xs.

Since *lines* itself is injective, we can look for a suitable homomorphism of the form

$$lines = (\otimes/) \cdot (f*). \tag{2}$$

If we succeed, then we shall have converted an implicit specification (1) into a constructive definition (2). The synthesis is by straightforward calculation.

First we determine f. By a standard argument we have

$$f\ a = lines\,[a]$$

If a is not the newline character, then

$$lines\,[a] = lines(\oplus/[[a]]) = [[a]]$$

using the definition of / on singletons and Equation (1). For $a = \mathrm{NL}$ we have

$$\begin{aligned} lines\,[\mathrm{NL}] &= lines([\,] +\!\!+ [\mathrm{NL}] +\!\!+ [\,]) \\ &= lines([\,] \oplus [\,]) \\ &= lines((\oplus/[[\,]]) \oplus (\oplus/[[\,]])) \\ &= lines(\oplus/[[\,],[\,]]) \\ &= [[\,],[\,]], \end{aligned}$$

using the definition of $\oplus$ and Equation (1).

Putting these results together,

$$\begin{aligned} f\ a &= [[\,],[\,]], \quad \text{if } a = \mathrm{NL}, \\ &= [[a]], \quad \text{otherwise.} \end{aligned}$$

Second, we determine $\otimes$. Since each argument of $\otimes$ is a non-empty list, we need only consider the definition of $(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys)$. Using Equations (1) and (2) and the distributive properties of /, we have that

$$\begin{aligned} (xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys) &= lines(\oplus/(xs +\!\!+ [x])) \otimes lines(\oplus/([y] +\!\!+ ys)) \\ &= lines((\oplus/(xs +\!\!+ [x])) +\!\!+ \oplus/([y] +\!\!+ ys)). \end{aligned}$$

Now,

$$\begin{aligned} \oplus/(xs +\!\!+ [x]) &= (\oplus/xs) \oplus (\oplus/[x]) \\ &= (\oplus/xs) +\!\!+ [\mathrm{NL}] +\!\!+ x, \end{aligned}$$

and similarly

$$\oplus/([y] +\!\!+ ys) = y +\!\!+ [\mathrm{NL}] +\!\!+ (\oplus/ys).$$

Their concatenation is therefore

$$\begin{aligned} & (\oplus/xs) +\!\!+ [\mathrm{NL}] +\!\!+ x +\!\!+ y +\!\!+ [\mathrm{NL}] +\!\!+ (\oplus/ys) \\ = \;& (\oplus/xs) +\!\!+ [\mathrm{NL}] +\!\!+ (\oplus/[x +\!\!+ y]) +\!\!+ [\mathrm{NL}] +\!\!+ (\oplus/ys) \\ = \;& (\oplus/xs) \oplus (\oplus/[x +\!\!+ y]) \oplus (\oplus/ys) \\ = \;& \oplus/(xs +\!\!+ [x +\!\!+ y] +\!\!+ ys). \end{aligned}$$

We conclude using Equation (1) that

$$\begin{aligned} (xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys) &= \mathit{lines}(\oplus/(xs +\!\!+ [x +\!\!+ y] +\!\!+ ys)) \\ &= xs +\!\!+ [x +\!\!+ y] +\!\!+ ys. \end{aligned}$$

Note that the above derivation actually juggles with some potentially fictitious values. No meaning has been assigned to $\oplus/[\,]$, yet terms of the form $\oplus/xs$ appear in a context where the case $xs = [\,]$ is not specifically excluded. No confusion can arise because, as we have seen in §2.2, a fictitious identity element of $\oplus$ can be added to the domain of values without inconsistency.

Notice also that, unlike $\oplus$, the operator $\otimes$ does have an identity element, namely $[[\,]]$. This follows from the fact that $\oplus/[[\,]] = [\,]$, since

$$\otimes/[\,] = \otimes/(\oplus/[[\,]]) = [[\,]].$$

It is instructive to develop this example a little further to show how other text processing functions can be synthesised. Define a *word* to be a non-empty sequence of characters not containing the newline or space characters. We can define the type *Word* by the equation

$$\mathit{Word} = [\mathit{Char} \setminus \{\mathrm{NL}, \mathrm{SP}\}]^+.$$

In a similar spirit to before, we can seek a constructive definition of a function *words* for breaking a line into words. The type of words is therefore

$$\mathit{words} : \mathit{Line} \to [\mathit{Word}].$$

The function $unwords : [Word]^+ \rightarrow Line$ defined by

$$\begin{aligned} unwords &= \oplus/ \\ x \oplus y &= x +\!\!+ [\mathrm{SP}] +\!\!+ y \end{aligned}$$

takes a sequence of words and concatenates them after inserting a space between adjacent words. The function *unwords* is injective, but not surjective. For example, none of the lines [ ], [SP], [SP, SP], . . . and so on, are in the range of *unwords*. However, if we temporarily admit the empty list as a possible word, then *unwords* becomes surjective on the augmented domain and we can define its inverse in an exactly similar way as we have done for *unlines*. Having done this, we can now define *words* by filtering out the empty sequences. Hence

$$words = ((\neq [\,])\triangleleft) \cdot (\otimes/) \cdot (f*)$$

where

$$\begin{aligned} f\,a &= [[\,],[\,]], \quad \text{if } a = \mathrm{SP} \\ &= [[a]], \quad \text{otherwise} \end{aligned}$$

and, as before,

$$(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys) = xs +\!\!+ [x +\!\!+ y] +\!\!+ ys.$$

Note that, although *words* · *unwords* is the identity function on non-empty sequences of words, the function *unwords* · *words* is not the identity function on lines. Redundant spaces are removed between words.

Finally, to complete a logical trio of functions, we can define a *paragraph* to be a non-empty sequence of non-empty lines and seek a definition of a function *paras* which breaks a sequence of lines into paragraphs. The type *Para* can be defined by the equation

$$Para = [Line^+]^+.$$

We require *paras* to have type $[Line] \rightarrow [Para]$. The function *unparas*, where

$$unparas : [Para]^+ \rightarrow [Line],$$

is defined by

$$\begin{aligned} unparas &= \oplus/ \\ xs \oplus ys &= xs +\!\!+ [[\,]] +\!\!+ ys, \end{aligned}$$

This function takes a sequence of paragraphs and converts it to a sequence of lines by inserting a single empty line between adjacent paragraphs and

concatenating the result. Like *unwords*, the function *unparas* is injective but not surjective. Again, by temporarily admitting the empty paragraph, we can make *unparas* surjective and define its inverse in the usual way. The empty sequences can then be filtered from the result.

To summarise: the types we have introduced are

$$\begin{array}{rcl} \mathit{Text} & = & [\mathit{Char}] \\ \mathit{Line} & = & [\mathit{Char} \setminus \{\mathrm{NL}\}] \\ \mathit{Word} & = & [\mathit{Char} \setminus \{\mathrm{NL}, \mathrm{SP}\}]^+ \\ \mathit{Para} & = & [\mathit{Line}^+]^+ \end{array}$$

The three "un-functions" are

$$\begin{array}{rcl} \mathit{unlines} & : & [\mathit{Line}]^+ \to \mathit{Text} \\ \mathit{unwords} & : & [\mathit{Word}]^+ \to \mathit{Line} \\ \mathit{unparas} & : & [\mathit{Para}]^+ \to [\mathit{Line}] \end{array} \qquad \begin{array}{rcl} \mathit{unlines} & = & \oplus_{\mathrm{NL}}/ \\ \mathit{unwords} & = & \oplus_{\mathrm{SP}}/ \\ \mathit{unparas} & = & \oplus_{[\,]}/ \end{array}$$

Here, we have

$$x \oplus_a y = x +\!\!+ [a] +\!\!+ y.$$

The three inverse functions are

$$\begin{array}{rcl} \mathit{lines} & = & (\otimes/) \cdot (f_{\mathrm{NL}}*) \\ \mathit{words} & = & ((\neq [\,])\triangleleft) \cdot (\otimes/) \cdot (f_{\mathrm{SP}}*) \\ \mathit{paras} & = & ((\neq [\,])\triangleleft) \cdot (\otimes/) \cdot (f_{[\,]}*), \end{array}$$

where

$$\begin{array}{rcll} f_b\, a & = & [[\,],[\,]], & \text{if } a = b \\ & = & [[a]], & \text{otherwise} \end{array}$$

and

$$(xs +\!\!+ [x]) \otimes ([y] +\!\!+ ys) = xs +\!\!+ [x +\!\!+ y] +\!\!+ ys.$$

These six functions have a variety of uses. We give just two. The number of lines, words and paragraphs in a text can be counted by

$$\begin{array}{rcl} \mathit{countlines} & = & \# \cdot \mathit{lines} \\ \mathit{countwords} & = & \# \cdot (+\!\!+/) \cdot (\mathit{words}*) \cdot \mathit{lines} \\ \mathit{countparas} & = & \# \cdot \mathit{paras} \cdot \mathit{lines}. \end{array}$$

Second, we can normalize a text by removing redundant empty lines between paragraphs and spaces between words. We have

$$\begin{array}{rcl} \mathit{normalize} & = & \mathit{unparse} \cdot \mathit{parse} \\ \mathit{parse} & = & ((\mathit{words}*)*) \cdot \mathit{paras} \cdot \mathit{lines} \\ \mathit{unparse} & = & \mathit{unlines} \cdot \mathit{unparas} \cdot ((\mathit{unwords}*)*). \end{array}$$

To *parse* a text here means to break it into lines, paragraphs and words. The type of *parse* is

$$parse : Text \rightarrow [[[Word]]]$$

For injective functions f and g we have

$$\begin{aligned}(f \cdot g)^{-1} &= g^{-1} \cdot f^{-1} \\ (f*)^{-1} &= (f^{-1}*),\end{aligned}$$

from which it follows that *parse* is injective and the definition of *unparse* is correct.

2.6 Promotion lemmas. As simple consequences of the Homomorphism Lemma of §2.3 we can derive the following useful identities. They generalise the distributive laws of $*$, $\triangleleft$ and $/$.

Lemma 2 [Promotion]. *For arbitrary function f, predicate p and associative operator $\oplus$ we have:*

$$\begin{aligned}(* \text{ promotion}) \quad (f*) \cdot (+\!\!+/) &= (+\!\!+/) \cdot ((f*)*) \\ (\triangleleft \text{ promotion}) \quad (p\triangleleft) \cdot (+\!\!+/) &= (+\!\!+/) \cdot ((p\triangleleft)*) \\ (/ \text{ promotion}) \quad (\oplus/) \cdot (+\!\!+/) &= (\oplus/) \cdot ((\oplus/)*).\end{aligned}$$

Proof. Set $h = (f*) \cdot (+\!\!+/)$. It is an easy calculation to show that

$$h\,[x] = f * x$$

and also

$$h(xs +\!\!+ ys) = h\ xs +\!\!+ h\ ys.$$

Hence h is the homomorphism $(+\!\!+/) \cdot ((f*)*)$. This establishes the $*$ - promotion law. Similar reasoning establishes $/$ - promotion. Finally, to prove $\triangleleft$ - promotion, recall that $(p\triangleleft)$ is a homomorphism of the form $(+\!\!+/) \cdot (f*)$ for a suitable function f, the definition of which is not relevant for the present proof. Using in turn, $*$ - promotion, $/$ - promotion and the distributivity of $*$ through composition, we have

$$\begin{aligned}(p\triangleleft) \cdot (+\!\!+/) &= (+\!\!+/) \cdot (f*) \cdot (+\!\!+/) \\ &= (+\!\!+/) \cdot (+\!\!+/) \cdot ((f*)*) \\ &= (+\!\!+/) \cdot ((+\!\!+/)*) \cdot ((f*)*) \\ &= (+\!\!+/) \cdot (((+\!\!+/) \cdot (f*))*) \\ &= (+\!\!+/) \cdot ((p\triangleleft)*)\end{aligned}$$

as required.

The term "promotion" is used to describe these results because they say that rather than mapping, reducing or filtering over one large sequence, one can divide the sequence into shorter ones, map, reduce or filter each of these (hence "promoting" the operation into the component sequences) and collect the outcomes. For example, consider the rule

$$(\downarrow/) \cdot (+\!\!+/) = (\downarrow/) \cdot ((\downarrow/)*).$$

In words this says the minimum of a flattened list of lists of numbers can be obtained by first minimising over each component list and then minimising over the results. If one of the component lists is empty, then its minimum will be the fictitious value ∞, but since $\infty \downarrow a = a \downarrow \infty = a$ the minimum of the minimums will only be ∞ if all the component lists are empty.

2.7 Selection and indeterminacy. We end the section with a discussion of two new operators which are mainly used with reductions.

Many problems in computation can be formulated as optimisation problems: find the cheapest, shortest, longest or perhaps the value of greatest profit in some given class of values. Such problems can be specified with the help of two new operators, $\downarrow_f$ and $\uparrow_f$. Just as $(a \downarrow b)$ selects the minimum of two numbers a and b, so $(a \downarrow_f b)$ selects either a or b according to which is smaller: $f\,a$ or $f\,b$. In the definition of $\downarrow_f$, function f has generic type $(\alpha \rightarrow \mathit{Num})$. The definition of $(a \uparrow_f b)$ is analogous: it selects a or b depending on which is greater: $f\,a$ or $f\,b$ (from now on we shall ignore $\uparrow_f$ as it is treated in an exactly similar manner to $\downarrow_f$). We have

$$\begin{aligned} a \downarrow_f b &= a, \quad \text{if } f\,a < f\,b \\ &= b, \quad \text{if } f\,a > f\,b. \end{aligned}$$

The lacuna in this definition occurs in the case $f\,a = f\,b$. If f is an injective function on the range of values of interest, then $f\,a = f\,b$ only if $a = b$ and we can assign $(a \downarrow_f b)$ their common value. For example, $\downarrow = \downarrow_{id}$. However, in the majority of practical cases the function f is not injective. To ask for the longest or shortest sequence in a class of sequences is really an abuse of language: there may be more than one such sequence. What is meant is *some* longest or shortest sequence.

In developing constructive solutions to problems of optimisation, the under specification permitted by $\downarrow_f$ can be very useful, especially as we are

not normally concerned with any other property of the result than that it minimises f. Accordingly, we shall allow expressions to contain occurrences of the operator $\downarrow_f$ when f is not an injective function. In these cases we interpret $\downarrow_f$ as standing for $\downarrow_{f'}$, where f' is some injective function — the precise nature of which we are not interested in — which respects the ordering given by f. That is, $f\,a < f\,b$ implies $f'\,a < f'\,b$. If $a \neq b$ but $f\,a = f\,b$, then either $f'\,a < f'\,b$ or $f'\,a > f'\,b$ and we do not care which. We suppose without proof that such an extension f' exists for any f (this assumption is related to the Axiom of Choice in Set theory).

When carrying out equational reasoning with $\downarrow_f$ we must be careful not to ascribe any properties to $\downarrow_f$ which are not implied by the foregoing convention. Only the following properties may be assumed:

(associativity)	$a \downarrow_f (b \downarrow_f c)$	$=$	$(a \downarrow_f b) \downarrow_f c$
(idempotence)	$a \downarrow_f a$	$=$	a
(commutativity)	$a \downarrow_f b$	$=$	$b \downarrow_f a$
(selectivity)	$a \downarrow_f b$	$=$	either a or b
(minimality)	$f(a \downarrow_f b)$	$=$	$f\,a \downarrow f\,b$.

At certain stages during the development of a constructive definition it may become appropriate to exercise a choice about the value of $(a \downarrow_f b)$ when $f\,a = f\,b$. Such a step is called a *choice* step and will be denoted by the sign $\rightsquigarrow$. For instance, if $f\,a = f\,b$ we can write

$$a \downarrow_f b \rightsquigarrow a$$

The sign $\rightsquigarrow$ can be read as "may be refined to". Taking a choice step is to be regarded as imposing a further property on the injective function f' of which f is the representative. This means that any choice step must be consistent with every previous choice step. For example, if $f(1) = f(2)$ and we decide, in some chain of reasoning, to impose the choice $1 \downarrow_f 2 \rightsquigarrow 1$, then it follows that

$$1 \downarrow_f 2 + 1 \downarrow_f 2 \rightsquigarrow 1 + 1 \downarrow_f 2 = 1 + 1 = 2.$$

However, the following reasoning is *not* valid:

$$1 \downarrow_f 2 + 1 \downarrow_f 2 \rightsquigarrow 1 + 1 \downarrow_f 2 \rightsquigarrow 1 + 2 = 3.$$

Having exercised a choice, the consequences must be followed consistently.

The major use of selection functions occurs in conjunction with reduction. For example, $\uparrow_{\#}/xs$ returns some longest sequence in the list of sequences xs. We shall see many examples in due course. As an extension to the minimality law we have

$$f \cdot (\downarrow_f /) = (\downarrow /) \cdot (f*).$$

For $f = \#$ this law expresses the formal equivalence of the English phrases "the length of the shortest" and "the minimum of the lengths".

3. Directed reduction and recursion

3.1 Left and right reduction We now introduce two more reduction operators: $\not\leftarrow$ (pronounced "right-reduce") and $\not\rightarrow$ (pronounced "left-reduce"). They are closely related to the reduction operator /, but each takes three arguments: an operator $\oplus$, a value e and a list x. They can be described by the equations

$$\begin{aligned}(\oplus \not\leftarrow e)[a_1, a_2, \ldots, a_n] &= a_1 \oplus (a_2 \oplus (\cdots (a_n \oplus e))) \\ (\oplus \not\rightarrow e)[a_1, a_2, \ldots, a_n] &= (((e \oplus a_1) \oplus a_2) \cdots \oplus a_n).\end{aligned}$$

In particular, we have

$$\begin{aligned}(\oplus \not\leftarrow e)\,[\,] &= e \\ (\oplus \not\rightarrow e)\,[\,] &= e.\end{aligned}$$

The operator $\oplus$ used in a left or right reduction need not be associative, so the brackets in the above equations are necessary. Indeed, the type of $\oplus$ may not even take the form $\alpha \times \alpha \rightarrow \alpha$. The types of $\not\leftarrow$ and $\not\rightarrow$ are given by

$$\begin{aligned}\not\leftarrow &: ((\alpha \times \beta \rightarrow \beta) \times \beta) \rightarrow [\alpha] \rightarrow \beta \\ \not\rightarrow &: ((\beta \times \alpha \rightarrow \beta) \times \beta) \rightarrow [\alpha] \rightarrow \beta.\end{aligned}$$

In the expression $(\oplus \not\leftarrow e)x$, the operator $\oplus$ has type $\alpha \times \beta \rightarrow \beta$, the value e has type β and x has type $[\alpha]$. The expression then has type β. Analogous reasoning applies to the combination $(\oplus \not\rightarrow e)x$. Note that $(\oplus \not\leftarrow e)$ and $(\oplus \not\rightarrow e)$ are both functions with type $[\alpha] \rightarrow \beta$.

Why do we need two more reduction operators? There are a number of answers to this question. First, the directed reductions can be regarded as "implementations" of the operator / in which the order of computation is completely specified. If $\oplus$ is associative with identity e, then certainly

$$\oplus/ = (\oplus \not\leftarrow e) = (\oplus \not\rightarrow e),$$

so undirected reductions can be expressed as directed reductions in two ways at least. In this sense, the directed reductions reflect a naive policy of sequential evaluation and can be translated directly into a suitable programming language. This point is amplified below.

The second, and more pragmatic answer is that many more functions can be described by directed reductions than by /. For example, the function $(f*)$ cannot be defined in terms of /, but we do have

$$\begin{aligned}(f*) &= (\oplus\!\!\not/[\,]) \\ &\quad \text{where } a \oplus x = [f\,a] +\!\!+ x.\end{aligned}$$

Furthermore, although every homomorphism can be expressed as a directed reduction (see §3.4), many functions which are not homomorphisms can be defined as directed reductions. For example, consider the function *prefix* which takes a predicate p and a list x as arguments and returns the longest initial segment of x all of whose elements satisy p (problems about segments will be discussed in §4). Thus,

$$\mathit{prefix}\ \mathit{even}\,[2,4,1,8] = [2,4].$$

The function *prefix* is not a homomorphism, but we do have

$$\mathit{prefix}\ p = (\oplus\!\!\not/[\,]),$$

where

$$\begin{aligned}a \oplus x &= [a] +\!\!+ x, \quad \text{if } p\,a \\ &= [\,], \qquad\quad \text{otherwise.}\end{aligned}$$

To illustrate this definition, consider

$$\begin{aligned}\mathit{prefix}\ \mathit{even}\,[2,4,1,8] &= 2 \oplus (4 \oplus (1 \oplus (8 \oplus [\,]))) \\ &= [2] +\!\!+ [4] +\!\!+ [\,] \\ &= [2,4]\end{aligned}$$

Further examples of directed reductions will be seen in due course.

3.2 Recursive characterisation. From the informal definition of $(\oplus\!\!\not/ e)$ we have

$$\begin{aligned}(\oplus\!\!\not/ e)[\,] &= e \\ (\oplus\!\!\not/ e)([a] +\!\!+ x) &= a \oplus (\oplus\!\!\not/ e)x\end{aligned}$$

for all elements a and lists x. Since every non-empty list can be expressed uniquely in the form $[a] +\!\!+ x$, these two equations characterise the behaviour

of $(\oplus\!\!\leftarrow e)$ completely. Putting it another way, we can regard $f = (\oplus\!\!\leftarrow e)$ as the *solution* of the recursive equations

$$\begin{array}{rcl} f\,[\,] & = & e \\ f\,([a] +\!\!+ x) & = & a \oplus f\,x \end{array}$$

In these equations the value of $f\,[\,]$ is specified directly, and the value $f\,([a] +\!\!+ x)$ is specified in terms of a and $f\,x$. Thus f is determined incrementally from "right to left". This explains the direction of the arrow in the sign $\leftarrow$. The progress of computation is essentially "recursive" (literally: to go backwards).

In the case of a left-reduction $(\oplus\!\!\rightarrow e)$, the informal description gives

$$\begin{array}{rcl} (\oplus\!\!\rightarrow e)\,[\,] & = & e \\ (\oplus\!\!\rightarrow e)\,(x +\!\!+ [a]) & = & (\oplus\!\!\rightarrow e)x \oplus a \end{array}$$

for all a and x. (Recall that functional application is more binding than any other operator and so the right hand side of the last equation is read as $((\oplus\!\!\rightarrow e)x) \oplus a$.) Hence $(\oplus\!\!\rightarrow e)$ processes lists from "left to right". The progress of computation is therefore essentially "iterative".

We shall now show that the function $(\oplus\!\!\rightarrow e)$ can also be characterised by the recursive equations

$$\begin{array}{rcl} (\oplus\!\!\rightarrow e)\,[\,] & = & e \\ (\oplus\!\!\rightarrow e)\,([a] +\!\!+ x) & = & (\oplus\!\!\rightarrow (e \oplus a))\,x. \end{array}$$

The first equation is immediate, so it is only necessary to show that the second one holds. We do this by induction on the length of x. For the empty list $[\,]$, we reason

$$(\oplus\!\!\rightarrow e)\,[a] = e \oplus a = (\oplus\!\!\rightarrow (e \oplus a))[\,].$$

For the case $x +\!\!+ [b]$, we reason inductively

$$\begin{array}{rcl} (\oplus\!\!\rightarrow e)([a] +\!\!+ (x +\!\!+ [b])) & = & (\oplus\!\!\rightarrow e)([a] +\!\!+ x +\!\!+ [b]) \\ & = & (\oplus\!\!\rightarrow e)([a] +\!\!+ x) \oplus b \\ & = & (\oplus\!\!\rightarrow (e \oplus a))\,x \oplus b \\ & = & (\oplus\!\!\rightarrow (e \oplus a))(x +\!\!+ [b]), \end{array}$$

using the associativity of $+\!\!+$ and the second defining equation for $(\oplus\!\!\rightarrow e)$.

It follows that $f = (\oplus/\!\!\!\!\leftarrow e)$ can be regarded as the solution of the recursive equations

$$\begin{aligned} f &= g\,e \\ g\,e\,[\,] &= e \\ g\,e\,([a] +\!\!+ x) &= g\,(e \oplus a)\,x \end{aligned}$$

We see therefore that both left and right reductions can be characterised by recursive equations of the same general form. In this form, a function f on lists is defined by (i) giving the value of $f[\,]$ directly; and (ii) specifying $f([a] +\!\!+ x)$ in terms of fx. Every list, of any type whatsoever, is either empty or of the form $[a] +\!\!+ x$ for *unique* values of a and x, so these two schemes are sufficient to characterise functions over (finite) lists. This style of recursive definition is a feature of functional programming languages (see [5] and [7]).

In functional programming the operation of concatenation is not provided as primitive. Instead, there is given a primitive operator ":" (pronounced "cons") which inserts a value into a list as a new first element. Thus, we have

$$a : x = [a] +\!\!+ x.$$

The type of ":" is given by

$$(:) : \alpha \times [\alpha] \rightarrow [\alpha].$$

Since

$$[a_1, a_2, \ldots, a_n] = a_1 : (a_2 : (\ldots (a_n : [\,]))),$$

every list, of any type whatsoever, can be constructed by inserting its elements successively into the empty list (hence the reason for the name "cons" which is an abbreviation for the word "construct").

We can define $+\!\!+$ in terms of cons by

$$x +\!\!+ y = (: \!\!\!\!\leftarrow y)x$$

for all lists x and y. From this equation the cost of evaluating $x +\!\!+ y$ is proportional to the length of x, assuming a ":" operation has unit cost.

There are a number of reasons why cons is taken as the primitive operation for lists in functional programming. One is that every non-empty list can be expressed in the form (or "pattern") $a : x$ in exactly one way, so that one can define an arbitrary recursive function by a scheme based on the patterns $[\,]$ and $a : x$. On the other hand, a non-empty list can be expressed in the form $x +\!\!+ y$ in many ways and this can lead to ambiguity in

a definition (unless, of course, the function is a homomorphism). Another reason concerns questions of efficiency to which we now briefly turn.

3.3 Efficiency considerations. With the introduction of the directed reduction operators we have begun to approach the question of what can reasonably be expected in the way of systematic computation by a machine. Although we shall not go into details of the underlying mechanisms, it is appropriate at this point to say something about the amount of time and space required to carry out the evaluation of a directed reduction. Most often, we shall solve a problem by a directed reduction and it is necessary to have some appreciation of the gains in efficiency thereby obtained.

First, let us consider a right-reduction $(\oplus\!\!\not+ e)x$. If the list x has length n, then the definition of $\not+$ suggests that the evaluation of $(\oplus\!\!\not+ e)x$ requires n applications of the operator $\oplus$. However, not every computation with a right-reduction must necessarily begin at the right hand end of the list and traverse backwards to the head. To illustrate this point, consider the function *prefix*(<3) which selects the longest initial segment of a list of numbers, all of whose elements are less than 3. We have *prefix*$(<3) = (\oplus\!\!\not+ [\,])$, where

$$\begin{aligned} a \oplus x &= a : x, \quad \text{if } a < 3 \\ &= [\,], \quad \text{otherwise.} \end{aligned}$$

Here, $a : x$ is used in preference to $[a] +\!\!+ x$. Using the recursive characterisation of $\not+$ as the basis, we can "unfold" the computation of *prefix*$(<3)\,[1 \ldots 100]$ in the following way:

$$\begin{aligned} \textit{prefix}(<3)\,[1 \ldots 100] &= (\oplus\!\!\not+ [\,])[1 \ldots 100] \\ &= 1 \oplus (\oplus\!\!\not+ [\,])[2 \ldots 100] \\ &= 1 : (\oplus\!\!\not+ [\,])[2 \ldots 100] \\ &= 1 : (2 \oplus (\oplus\!\!\not+ [\,])[3 \ldots 100]) \\ &= 1 : 2 : (\oplus\!\!\not+ [\,])[3 \ldots 100] \\ &= 1 : 2 : (3 \oplus (\oplus\!\!\not+ [\,])[4 \ldots 100]) \\ &= 1 : 2 : [\,] \\ &= 1 : [2] \\ &= [1, 2] \end{aligned}$$

The length of this derivation is proportional to the length of the resulting list, *not* the length of the original list. In other words, the number of $\oplus$ operations actually carried out is 3 not 100. The crucial fact which enables the calculation to be shortened is that for $a \geq 3$ we have $a \oplus x = [\,]$ for

all lists x. Since the value of x is not required, it need not be calculated. This strategy of symbolic evaluation combined with the policy of performing only those calculations necessary to determine the result is known as *lazy evaluation*. Lazy evaluation can be programmed into a mechanical evaluator quite easily (see [5]), though we shall not go into details.

Here is another, more dramatic example. Consider $(\ll \not+ e)x$, where $a \ll b = a$. We have

$$\begin{aligned}(\ll \not+ e)[1 \ldots 100] &= 1 \ll (\ll \not+ e)[2 \ldots 100] \\ &= 1,\end{aligned}$$

so the computation terminates after only one step.

Now let us turn to left-reductions. The situation here is different from right-reduction in that, when processing lists from left to right, all the elements do have to be considered in order for the result to be returned. Consider, for instance, the symbolic evaluation of $(\ll \not+ 0)[1,2,3]$, where $a \ll b = a$. This evaluation is based on the recursive characterisation of $\not+$ by the equations

$$\begin{aligned}(\oplus \not+ e)[\,] &= e \\ (\oplus \not+ e)(a : x) &= (\oplus \not+ (e \oplus a))x\end{aligned}$$

For the specific example, we have:

$$\begin{aligned}(\ll \not+ 0)[1,2,3] &= (\ll \not+ (0 \ll 1))[2,3] \\ &= (\ll \not+ 0)[2,3] \\ &= (\ll \not+ (0 \ll 2))[3] \\ &= (\ll \not+ 0)[3] \\ &= (\ll \not+ (0 \ll 3))[\,] \\ &= (\ll \not+ 0)[\,] \\ &= 0.\end{aligned}$$

In this evaluation the complete list is traversed before the answer is returned.

To summarise these observations: right-reductions can be more *time* efficient than left-reductions; this happens when values of the operator concerned do not always depend on the full evaluation of the right-hand arguments. Such an operator is said to be *non-strict* (in its right argument).

The reverse situation can occur with space efficiency. Left-reduction can be more efficent in the amount of space required to carry out the computation. Compare the evaluations of $(+ \not+ 0)[1,2,3]$ and $(+ \not+ 0)[1,2,3]$. For the

former we have:

$$\begin{aligned}(+\not\leftarrow 0)[1,2,3] &= 1+(+\not\leftarrow 0)[2,3]\\ &= 1+(2+(+\not\leftarrow 0)[3])\\ &= 1+(2+(3+(+\not\leftarrow 0)[\,]))\\ &= 1+(2+(3+0))\\ &= 1+(2+3)\\ &= 1+5\\ &= 6.\end{aligned}$$

In this computation the sizes of the intermediate expressions grow in proportion to the length of the original list. This is an important measure because the sizes of the intermediate expressions reflect the amount of space which would have to be available to a mechanism in order to carry out the computation.

On the other hand, we can compute:

$$\begin{aligned}(+\not\leftarrow 0)[1,2,3] &= (+\not\leftarrow(0+1))[2,3]\\ &= (+\not\leftarrow 1)[2,3]\\ &= (+\not\leftarrow(1+2))[3]\\ &= (+\not\leftarrow 3)[3]\\ &= (+\not\leftarrow(3+3))[\,]\\ &= (+\not\leftarrow 6)[\,]\\ &= 6,\end{aligned}$$

and the size of the intermediate expressions never grows beyond a constant amount. The inner calculations are performed as they arise: this is safe because $+$ is a *strict* function, demanding complete evaluation of its arguments to determine the result.

In general, it is better to use right reduction when the operator concerned is non-strict and left reduction when it is strict. For example, when the operator is one of $+\!\!+$, $\wedge$, or $\vee$, we use right reduction; and when it is one of $+$, $\uparrow$, or $\downarrow$, we use left reduction.

This concludes a brief treatment of efficiency issues. In describing the symbolic evaluation of expressions, we have outlined the main method by which functional programming languages are implemented. For further details the reader should consult [5] or [7]. Since we wish to present problems, derivations and solutions at a higher level of abstraction than is provided by specific constructs in particular programming languages, it is left to informed readers to develop for themselves the connections between directed reductions and programs in conventional or functional languages.

3.4 Duality and specialisation. We state without proof two useful results concerning the relationship between the various forms of reduction.

Lemma 3 [Duality] *For all* $\oplus$ *and* e *we have*

$$(\oplus \not\leftarrow e) = (\tilde{\oplus} \not\rightarrow e) \cdot \mathit{reverse},$$

where $a \,\tilde{\oplus}\, b = b \oplus a$.

Lemma 4 [Specialisation] *Every homomorphism can be defined as either a left or a right reduction. More precisely,*

$$(\odot/) \cdot (f*) = (\oplus \not\leftarrow e) = (\otimes \not\rightarrow e),$$

where

$$\begin{aligned} a \oplus b &= f\, a \odot b \\ a \otimes b &= a \odot f\, b. \end{aligned}$$

We give just one illustration of the specialisation lemma. Consider the function

$$\mathit{lines} = (\otimes/) \cdot (f*)$$

of §2.5, where

$$\begin{aligned} f\, a &= [[\,],[\,]], \quad \text{if } a = \mathrm{NL} \\ &= [[a]], \quad \text{otherwise} \end{aligned}$$

and

$$(xs \mathbin{+\!\!+} [x]) \otimes ([y] \mathbin{+\!\!+} ys) = xs \mathbin{+\!\!+} [x \mathbin{+\!\!+} y] \mathbin{+\!\!+} ys.$$

Set $a \oplus xs = f\, a \otimes xs$. Since $\otimes/[\,] = [[\,]]$, we have by specialisation that

$$\mathit{lines} = (\oplus \not\leftarrow [[\,]]).$$

It remains to simplify the definition of $\oplus$. First, if $a = \mathrm{NL}$, then

$$\begin{aligned} a \oplus (x : xs) &= f\,\mathrm{NL} \otimes ([x] \mathbin{+\!\!+} xs) \\ &= [[\,],[\,]] \otimes ([x] \mathbin{+\!\!+} xs) \\ &= [[\,]] \mathbin{+\!\!+} [[\,] \mathbin{+\!\!+} x] \mathbin{+\!\!+} xs \\ &= [[\,]] \mathbin{+\!\!+} [x] \mathbin{+\!\!+} xs \\ &= [\,] : x : xs, \end{aligned}$$

using the relation $a : x = [a] \mathbin{+\!\!+} x$ and the definition of f and $\otimes$.

Second, if $a \neq \mathrm{NL}$, then

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

Hence we obtain

$$\begin{aligned} a \oplus (x : xs) &= [\,] : x : xs, \quad \text{if } a = \mathrm{NL} \\ &= (a : x) : xs, \quad \text{otherwise.} \end{aligned}$$

3.5 Accumulation. We end the discussion on directed reductions by introducing another operator $-\!\!\!/\!\!\!\!\rightarrow$ (pronounced "accumulate") which is closely related to left-reduction. Examples of its use will feature in the next section. Like $-\!\!/\!\!\rightarrow$ the operator $-\!\!\!/\!\!\!\!\rightarrow$ takes an operator $\oplus$, a value e and a list x as arguments. Its effect is described by the equation

$$(\oplus -\!\!\!/\!\!\!\!\rightarrow e)[a_1, a_2, \ldots, a_n] = [e, e \oplus a_1, (e \oplus a_1) \oplus a_2, \ldots, ((e \oplus a_1) \cdots \oplus a_n)].$$

The operator $-\!\!\!/\!\!\!\!\rightarrow$ encapsulates a common pattern of computation in which a sequence $c_0, c_1, \ldots, c_n$ is defined in terms of a given sequence $a_1, a_2, \ldots, a_n$ and a starting value e by a reccurrence relation of the form

$$\begin{aligned} c_0 &= e \\ c_{k+1} &= c_k \oplus a_{k+1} \quad (0 < k < n) \end{aligned}$$

We have

$$[c_0, c_1, \ldots, c_n] = (\oplus -\!\!\!/\!\!\!\!\rightarrow e)[a_1, \ldots, a_n].$$

For example, the list $0!, 1!, \ldots n!$ of factorial numbers can be defined by the expression

$$(\times -\!\!\!/\!\!\!\!\rightarrow 1)[1 \ldots n].$$

This expression can be evaluated more efficiently than the alternative

$$fac * [0 \ldots n],$$

where $fac\ k = \times/[1 \ldots k]$. The former requires just n multiplications to generate the list, while the latter requires $n(n-1)/2$ multiplications.

In general,

$$(\oplus -\!\!/\!\!\rightarrow e) = last \cdot (\oplus -\!\!\!/\!\!\!\!\rightarrow e),$$

so every left-reduction can be defined in terms of an accumulation. More interesting is the fact that an accumulation can be defined as a left-reduction. We have

$$(\oplus \mathbin{-\!\!\!\not\!\!\rightarrow} e) = (\otimes \mathbin{\not\rightarrow} [e]),$$

where

$$x \otimes a = x +\!\!+ [last\ x \oplus a].$$

This result shows why the number of $\oplus$ operations can be reduced from $O(n^2)$ to $O(n)$, where n is the length of the argument list.

Alternatively, we can characterize $\mathbin{-\!\!\!\not\!\!\rightarrow}$ by two recursive equations:

$$\begin{array}{rcl} (\oplus \mathbin{-\!\!\!\not\!\!\rightarrow} e)[\,] & = & [e] \\ (\oplus \mathbin{-\!\!\!\not\!\!\rightarrow} e)(a : x) & = & [e] +\!\!+ (\oplus \mathbin{-\!\!\!\not\!\!\rightarrow} (e \oplus a))x. \end{array}$$

From the point of view of efficiency, this definition is superior to the definition as a left-reduction. Under a strategy of lazy evaluation using the recursive definition as a basis, elements of the result list can be produced before the argument list is completely traversed.

4. Segments and Partitions

4.1 Definitions. The object of the present section is to derive computationally efficient solutions for a number of problems about segments. A list y is said to be a *segment* of x if there exist u and v such that $x = u +\!\!+ y +\!\!+ v$. A list y is an *initial segment* of x if there exists a v such that $x = y +\!\!+ v$, and a *final segment* if there exists a u such that $x = u +\!\!+ y$.

The function *inits* returns the list of initial segments of a list, in increasing order of length. The function *tails* returns the list of final segments of a list, in decreasing order of length. Thus

$$\begin{array}{rcl} inits[a_1, a_2, \ldots, a_n] & = & [[\,], [a_1], [a_1, a_2], \ldots, [a_1, a_2, \ldots, a_n]] \\ tails[a_1, a_2, \ldots, a_n] & = & [[a_1, a_2, \ldots, a_n], [a_2, a_3, \ldots, a_n], \ldots, [\,]] \end{array}$$

Since both functions are injective, they can be defined formally as homomorphisms; using the specialisation lemma, they can therefore be defined as directed reductions. We shall do this directly. Since $inits([a] +\!\!+ x)$ consists, in order, of $[\,]$ and the list of initial segments of x in which each element is prefixed by a, we have

$$\begin{array}{rcl} inits\,[\,] & = & [[\,]] \\ inits([a] +\!\!+ x) & = & [[\,]] +\!\!+ ([a] +\!\!+) * inits\ x. \end{array}$$

Solving this recursion gives

$$inits = (\oplus \leftarrow [[\,]]),$$

where

$$a \oplus xs = [[\,]] +\!\!+ ([a] +\!\!+) * xs.$$

Analogous reasoning gives

$$\begin{array}{rcl} tails\,[\,] & = & [[\,]] \\ tails(x +\!\!+ [a]) & = & (+\!\!+ [a]) * tails\, x +\!\!+ [[\,]], \end{array}$$

and so

$$tails = (\oplus \leftarrow [[\,]]),$$

where

$$xs \oplus a = (+\!\!+ [a]) * xs +\!\!+ [[\,]].$$

We shall make use of the recursive characterisation of *tails* below.

The following simple result, whose proof is omitted, relates the function $\twoheadleftarrow$ to $\leftarrow$.

Lemma 5 *For all* $\oplus$, e *and lists* x *we have*

$$(\oplus \twoheadleftarrow e)x = (\oplus \leftarrow e) * inits\, x.$$

The function *segs* returns a list of all segments of a given list. We shall define

$$segs\, x = +\!\!+ / tails * inits\, x.$$

For example,

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

The order in which the segments appear in this list is not important for our purposes and we shall make no use of it. Notice that the empty list occurs more than once in the result.

4.2 Segment decomposition The following theorem can be used as the starting point in the derivation of efficient solutions to a number of problems about segments.

Theorem 1 [Segment Decompostion] *Suppose S and T are defined by*

$$\begin{aligned} S\,x &= \oplus/f * p \triangleleft segs\,x \\ T\,x &= \oplus/f * p \triangleleft tails\,x. \end{aligned}$$

Then $S\,x = \oplus/T * inits\,x$.

Proof. The proof is by straightforward calculation. We have

$$\begin{aligned} S\,x &= \oplus/f * p \triangleleft segs\,x \\ &= \oplus/f * p \triangleleft (+\!\!+/tails * inits\,x) && \text{(defn.}\,segs\text{)} \\ &= \oplus/f * (+\!\!+/(p \triangleleft) * tails * inits\,x) && (\triangleleft \text{ promotion}) \\ &= \oplus/(+\!\!+/(f*) * (p \triangleleft) * tails * inits\,x) && (* \text{ promotion}) \\ &= \oplus/(\oplus/) * (f*) * (p \triangleleft) * tails * inits\,x && (/ \text{ promotion}) \\ &= \oplus/((\oplus/) \cdot (f*) \cdot (p \triangleleft) \cdot tails) * inits\,x && (*, \cdot \text{ distrib.}) \\ &= \oplus/T * inits\,x && (\text{defn. } T) \end{aligned}$$

Corollary 1 *Suppose $T = (\otimes\!\!\not\!\!\rightarrow e)$ for some operator $\otimes$ and value e. Then*

$$S = (\oplus/) \cdot (\otimes\!\!\not\!\!\!\rightarrow e).$$

Proof. Immediate, using the above relationship between $\not\!\!\!\rightarrow$ and $\not\!\!\rightarrow$.

It follows from this corollary that if $\oplus$ and $\otimes$ have constant cost, then $S\,x$ can be computed in $O(n)$ steps, where $n = \#x$.

The following lemma gives a sufficient condition for T to be expressible as a left reduction.

Lemma 6 *Suppose $T\,x = \oplus/f * tails\,x$, where $f = (\otimes\!\!\not\!\!\rightarrow e)$. If $\otimes$ distributes through $\oplus$, i.e.*

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c),$$

then $T = (\odot\!\!\not\!\!\rightarrow e)$, where

$$a \odot b = (a \otimes b) \oplus e.$$

Proof. It is easy to show $T\,[\,] = e$. We prove $T(x +\!\!+ [a]) = (T\,x \otimes a) \oplus e$. Solving these equations gives the required result. To establish the equation, observe that if $f = (\otimes\!\!\not\!\!\rightarrow e)$, then

$$f * (+\!\!+ [a]) * xs = (\otimes a) * f * xs$$

for all lists (of lists) xs. Furthermore, if $\otimes$ distributes through $\oplus$, then

$$\oplus/(\otimes a) * xs = (\otimes a)(\oplus/xs).$$

Using these results, together with the recursive characterisation of *tails*, we can therefore compute

$$\begin{aligned} T(x +\!\!+ [a]) &= \oplus/f * \mathit{tails}(x +\!\!+ [a]) \\ &= \oplus/f * ((+\!\!+ [a]) * \mathit{tails}\ x +\!\!+ [[\,]]) \\ &= \oplus/((\otimes a) * f * \mathit{tails}\ x +\!\!+ [e]) \\ &= (\oplus/(\otimes a) * f * \mathit{tails}\ x) \oplus e \\ &= (\otimes a)(\oplus/f * \mathit{tails}\ x) \oplus e \\ &= (T\ x \otimes a) \oplus e. \end{aligned}$$

This completes the proof.

This result can be illustrated by solving a problem of Gries. The problem is to compute the minimum of the sums of all segments of a given list of positive and negative numbers: in symbols,

$$\mathit{minsum}\ x = \downarrow/(+/) * \mathit{segs}\ x.$$

Direct calculation from this expression requires $O(n^3)$ steps, where $n = \#x$. There are $O(n^2)$ segments of x and each can be summed in $O(n)$ steps. As the minimum of the sums can be computed in $O(n^2)$ steps, there are $O(n^3)$ steps in total. However, it is easy to derive a linear time algorithm. Since $(+/) = (+\!\!\!\!/\,0)$ and $+$ distributes through $\downarrow$, we have from the work above that

$$\mathit{minsum} = (\downarrow/) \cdot (\odot\!\!\!\!/\,0),$$

where $a \odot b = (a + b) \downarrow 0$.

4.3 Extremal problems. A common problem is to find some longest segment of a list satisfying a given property p. In text processing, for example, we may want to take the longest initial segment of a list of words which will fit on a line of given width. By repeating this process with the remaining words on subsequent lines, it is possible to solve the problem of formatting text. This problem will be discussed in more detail later on.

Let us consider the problem of computing the functions

$$\begin{aligned} S\ x &= \uparrow_{\#}/p \triangleleft \mathit{segs}\ x \\ I\ x &= \uparrow_{\#}/p \triangleleft \mathit{inits}\ x \\ T\ x &= \uparrow_{\#}/p \triangleleft \mathit{tails}\ x \end{aligned}$$

It will be assumed throughout that p holds for the empty list at least, so there is always a well-defined (and, indeed, unique in the case of I and T) solution for any given list x. Assuming $O(n^k)$ steps are required to determine whether p holds for a list of length n, the time required to compute $S\ x$ is $O(n^{k+2})$ steps, where $n = \#x$. Our purpose is to examine useful conditions which can be imposed on p to reduce this estimate.

We mention three such conditions. A predicate p on lists will be called *prefix-closed* if

$$p(x +\!\!+ y) \Rightarrow p\ x$$

for all x and y (here, $\Rightarrow$ denotes logical implication). Similarly, p is called *suffix-closed* if

$$p(x +\!\!+ y) \Rightarrow p\ y.$$

Finally, p is *segment-closed* if it is both prefix and suffix-closed; that is,

$$p(x +\!\!+ y) \Rightarrow p\ x \wedge p\ y.$$

for all x and y. The terminology is appropriate since it is easy to show that a segment-closed predicate holds for all segments of x whenever it holds for x.

Each of the three classes of predicates is closed under the operations of conjunction and disjunction. One can also show that p is prefix-closed if and only if

$$p \triangleleft \mathit{inits}\ x = \mathit{inits}(\uparrow_{\#} / p \triangleleft \mathit{inits}\ x).$$

A similar characterisation holds for suffix-closed predicates.

We now state without proof two results concerning these properties.

Lemma 7 *If p is prefix-closed, then $T = (\oplus\!\!\not\rightarrow[\,])$, where*

$$x \oplus a = \uparrow_{\#} / p \triangleleft \mathit{tails}(x +\!\!+ [a])$$

Consequently, $S = (\uparrow_{\#}/) \cdot (\oplus\!\!\not\rightarrow[\,])$.

The second part of the lemma follows from the Corollary to the Segment Decomposition Theorem. To see what this result buys in the way of increased efficiency, suppose

$$(\oplus\!\!\not\rightarrow[\,])x = [x_0, x_1, \ldots, x_n],$$

where $n = \#x$. Let $l_j = \#x_j$. To compute x_{j+1} from x_j requires p to be applied in succession to lists of lengths $l_j + 1, l_j, \ldots l_{j+1}$. If p requires $O(n^k)$ steps for a list of length n, then the jth step requires

$$\sum_{i=l_{j+1}}^{l_j+1} i^k$$

steps. Summing over j leads to the result that $S\ x$ can be computed in $O(n^{k+1})$ steps.

We give one illustration. Let *nodups* x denote the property that list x contains no repeated elements. If the only available comparison test is the test for equality, the computation of *nodups* requires $O(n^2)$ steps on a list of length n. Direct calculation of

$$\uparrow_{\#} / \mathit{nodups} \triangleleft \mathit{segs}\ x$$

therefore requires $O(n^4)$ steps. However, *nodups* is prefix-closed, so using the algorithm implicit in the above result we can bring the time down to $O(n^3)$ steps.

The next lemma shows how to decrease the time still further.

Lemma 8 *Suppose p is segment-closed, holds for all singleton sequences, and satisfies*

$$p(x +\!\!+ [a]) = p\ x \wedge q\ a\ x$$

for some suitable predicate q. Then $T = (\oplus \!\!\rightarrow [\,])$, where

$$x \oplus a = (\uparrow_{\#} / q\ a \triangleleft \mathit{tails}\ x) +\!\!+ [a]$$

Consequently, $S = (\uparrow_{\#} /) \cdot (\oplus \!\!\rightarrow [\,])$.

It can be shown that if q is computable in $O(n^k)$ steps, then $S\ x$ can be computed in $O(n^{k+1})$ steps.

To illustrate this result, consider the *nodups* problem again. The predicate *nodups* is segment-closed and holds for all singleton sequences. Moreover,

$$\mathit{nodups}(x +\!\!+ [a]) = \mathit{nodups}\ x \wedge \mathit{all}(\neq a)x$$

Since $\mathit{all}(\neq a)x$ can be computed in $O(n)$ steps, where $n = \#x$, we have that $\uparrow_{\#} / \mathit{nodups} \triangleleft \mathit{segs}\ x$ can be computed in $O(n^2)$ steps.

4.4 Partitions. A *partition* of a list x is a decomposition of x into non-empty segments. In symbols, xs is a partition of x if

$$+\!\!+/xs = x \wedge all\,(\neq [\,])\,xs.$$

The function *parts* returns a list of all possible partitions of x. In this subsection we state without proof an important theorem for solving problems of the form

$$\downarrow_f / all\ p \triangleleft parts\ x$$

for suitable f and p. We first give two illustrations of why this problem is important in practice.

Text-formatting. Suppose x is a list of words (see §2.5 for the relevant definitions used in this example). An important problem in text processing is to format text into lines of given width m, ensuring as many words as possible are on each line (adjacent words being separated by at least one space). A list x will fill a line of width m just in the case that $m \geq \#unwords\,x$. Define

$$fits\ m\ x = m \geq \#unwords\ x.$$

The problem of formatting text can be described as an optimisation problem

$$format\ x = \downarrow_{waste} / all(fits\ m) \triangleleft parts\ x,$$

where *waste* is a suitable measure of the badness of a given way of breaking text into lines. This problem was considered in [1], where the following definitions of waste were examined:

$$\begin{aligned} waste1 &= (\uparrow/) \cdot (whitespace\ m*) \\ waste2 &= (+/) \cdot (whitespace\ m*), \end{aligned}$$

where $whitespace\ m\ x = m - \#unwords\ x$.

Sorting by merging. A list of numbers can be sorted by first partitioning the list into ordered segments (called **runs**) and then merging the runs. The function

$$runs\ x = \downarrow_{\#} / all\ ordered \triangleleft parts\ x$$

determines the optimal way to partition the sequence prior to merging. In fact, if $x \otimes y$ denotes the ordered list which results when x and y are merged, then

$$sort\ x = \otimes / runs\ x$$

specifies the complete sorting procedure.

For suitably restricted f and p, the problem of computing

$$\downarrow_f / \mathit{all}\ p \lhd \mathit{parts}\ x$$

can be solved by a "greedy" algorithm which computes the solution incrementally by taking as much of x at each stage as it can. We define

$$\begin{array}{lcll} \mathit{greedy}\ p\ x & = & [\,], & \text{if } x = [\,] \\ & = & [x'] +\!\!+ \mathit{greedy}\ p\ (x \setminus x'), & \text{otherwise} \\ & & \text{where } x' = \uparrow_{\#} / p \lhd \mathit{inits}\ x & \end{array}$$

For an initial segment x' of x, the value of $x \setminus x'$ is the final segment which remains when x' is removed from x. For suitable p we shall see how to compute x' quickly, so the greedy algorithm can be very efficient.

We need two conditions on f to relate the greedy algorithm to the partition problem. Say a function $f : [[\alpha]] \to \mathit{Num}$ is *stable* if

$$f\ xs \leq f\ ys \Rightarrow f([x] +\!\!+ xs) \leq f([x] +\!\!+ ys)$$

for all lists xs, ys and x.

Furthermore, say f is *greedy* if

$$f([x +\!\!+ y +\!\!+ z] +\!\!+ xs) \leq f([x +\!\!+ y] +\!\!+ [z] +\!\!+ xs) \leq f([x] +\!\!+ [y +\!\!+ z] +\!\!+ xs)$$

for all x, y, z and xs.

One proof of the following result can be found in [1].

Theorem 2 [The Greedy Theorem for Partitions] *Suppose p is segment-closed and holds for all singletons. Suppose f is stable and greedy. Then*

$$\downarrow_f / \mathit{all}\ p \lhd \mathit{parts}\ x \rightsquigarrow \mathit{greedy}\ p\ x.$$

Notice the refinement step $\rightsquigarrow$. The theorem does not state that the greedy algorithm gives the only optimal way of partitioning x, but just one optimal way. Moreover, the conditions on f are such that knowledge of f is not required for execution of the algorithm.

Two obvious applications of the theorem are to the problems of formatting text and sorting by merging. For the first, the predicate *fits m* is segment-closed and *waste2* can be shown to satisfy the hypothesis on f. For the second, the predicate *ordered* is segment-closed and the function #

is stable and greedy. Both problems can therefore be solved by a greedy algorithm.

There remains the problem of computing

$$\uparrow_{\#} / p \triangleleft \mathit{inits}\ x$$

quickly, since this is crucial to the success of the greedy algorithm. We state without proof a final lemma which addresses this problem. In the statement we use the function *prefix* mentioned in §3 and a related function *take* which selects initial segments of a list with given length: $\mathit{take}\ n\ x$ takes the initial segment of x of length n.

Lemma 9 *If p is prefix-closed and $p = q \cdot (\oplus \!\!\not\!\!\rightarrow e)$, then*

$$\uparrow_{\#} / p \triangleleft \mathit{inits}\ x = \mathit{take}(\#y - 1)x$$

where $y = \mathit{prefix}\ q((\oplus \!\!\not\!\!\!\!\rightarrow e)x)$.

To illustrate this result, recall that

$$\mathit{fits}\ m\ x = m \geq \#\mathit{unwords}\ x$$

From §2.5 we have $\# \cdot \mathit{unwords} = (\oplus/) \cdot (\#*)$, where $n \oplus m = n + m + 1$. By the specialisation lemma, we have $(\oplus/) \cdot (\#*) = (\otimes \!\!\not\!\!\rightarrow e)$, where e is the identity element of $\oplus$, so $e = -1$, and $n \otimes w = n + \#w + 1$. It follows that

$$\mathit{fits}\ m = (m \geq) \cdot (\otimes \!\!\not\!\!\rightarrow e).$$

As $(\mathit{fits}\ m)$ is prefix-closed, the lemma is applicable and reduces the cost of calculating

$$\uparrow_{\#} / \mathit{fits}\ m \triangleleft \mathit{inits}\ x$$

to $O(m)$ steps.

4.5 Conclusions. We hope we have shown enough of the theory of lists to convince the student of its mathematical depth and elegance, as well as its usefulness in deriving solutions to practical problems. There are many subjects we have not touched on: the theory of subsequences, permutations, arrays (lists of lists), and infinite lists to name just a few. There are other kinds of greedy theorem for other kinds of problems about lists. Even having disposed of lists, there remains trees, bags and sets for which a similar generic theory is appropriate.

References

1. Bird, R.S. Transformational programming and the paragraph problem. *Science of Computer Programming* 6 (1986) 159-189.

2. Bird, R.S. The promotion and accumulation strategies in transformational programming. *ACM. Trans. on Prog. Lang. and Systems* 6 (1984) 487-450. Addendum *Ibid* 7 (1985).

3. Bird, R.S. and Hughes, R.J.M. The alpha-beta algorithm: an exercise in program transformation. *Inf. Proc. Letters* (to appear 1986).

4. Bird, R.S. and Meertens L.G.L.T Two exercises found in a book on algorithmics. *Proc. TC2 Conference on Program Specification and Transformation*, Bad Tolz, W. Germany 1986 (to appear Springer LNCS 1986).

5. Bird, R.S. and Wadler, P. *An Introduction to Functional Programming* Prentice-Hall (to be published 1987).

6. Meertens, L.G.L.T Algorithmics - towards programming as a mathematical activity. *Proc. CWI Symp. on Mathematics and Computer Science*, CWI Monographs, North-Holland, 1 (1986) 289-334.

7. Turner, D. Recursion equations as a programming language. *Functional Programming and its Applicatons*, Cambridge University Press, Cambridge, U.K. 1982.