R11069

Lectures on Constructive Functional Programming

by

R.S. Bird

Technical Monograph PRG-69
ISBN 0-902928-51-1
September 1988
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

Lectures on Constructive Functional Programming

Richard S. Bird
Programming Research Group
11 Keble Rd., Oxford OX1 3QD, UK

Abstract

The subject of these lectures is a calculus of functions for deriving programs from their specifications. This calculus consists of a range of concepts and notations for defining functious over various data types (including lists, trees and arrays), together with their algebraic and other properties. Each lecture begins with a specific problem, and the theory necessary to solve it is theu developed. In this way we hope to show that a functional approach to the problem of systematically calculating programs from their specifications can take its place alongside other methodologies.

Acknowledgements

Much of the following material was developed in collaboration with Lambert Meertens of the CWI, Amsterdam, to whom I am grateful for both support and friendship.

I would also like to acknowledge the contributions of Roland Backhouse, Jeremy Gibbons, Geraint Jones, Andrew Kay, Ooge de Moor, Doaitse Swierstra, and Chris Wright, all of whom have suggested important ideas or improvements. Any failure of understanding or deficiencies in presentation remain, of course, my sole responsibility. Detailed references, including related work, are given at the end of each lecture.

1 Basic concepts

1.0 Problem

Consider the following simple identity:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 + 1 = ((1 \times a_1 + 1) \times a_2 + 1) \times a_3 + 1$$

This equation generalises in the obvious way to n values $a_1, a_2, \ldots, a_n$ and we will refer to it subsequently as **Horner's rule**. As we shall see, Horner's rule turns out to be quite useful in our calculus. The reason is that the interpretation of $\times$ and $+$ need not be confined to the usual multiplication and addition operations of arithmetic. The essential constraints are only that both operations are associative, $\times$ has identity element 1, and $\times$ distributes through $+$.

The problem we address in this lecture is to develop suitable notation for expressing Horner's rule concisely.

1.1 Functions

Except where otherwise stated all functions are assumed to be *total*. The fact that a function f has source type α and target type β will be denoted in the usual way by $f : \alpha \rightarrow \beta$. We shall say that f takes arguments in α and returns results in β.

Functional application is written without brackets; thus $f\ a$ means $f(a)$. Functions are curried and application associates to the left, so $f\,a\,b$ means $(f\ a)b$ and not $f(a\ b)$. Where convenient we will write $f_a\ b$ as an alternative to $f\ a\ b$. Functional application is more binding than any other operation, so $f\ a \oplus b$ means $(f\ a) \oplus b$ and not $f(a \oplus b)$.

Functional composition is denoted by a centralised dot $(\cdot)$. We have

$$(f \cdot g)a = f(g\ a)$$

Various symbols will be used to denote general binary operators, in particular, $\oplus$, $\otimes$, and $\circledast$ will be used frequently. No particular properties of an operator should be inferred from its shape. For example, depending on the context, $\oplus$ may or may not denote a commutative operation.

Binary operators can be *sectioned*. This means that $(\oplus)$, $(a\oplus)$ and $(\oplus a)$ all denote functions. The definitions are:

$$\begin{array}{rcl}(\oplus)\,a\,b & = & a \oplus b \\ (a\oplus)\,b & = & a \oplus b \\ (\oplus b)\,a & = & a \oplus b\end{array}$$

Thus, if $\oplus$ has type $\oplus : \alpha \times \beta \to \gamma$, then we have

$$\begin{array}{lcl} (\oplus) & : & \alpha \to \beta \to \gamma \\ (a\oplus) & : & \beta \to \gamma \\ (\oplus b) & : & \alpha \to \gamma \end{array}$$

for all a in α and b in β.

For example, one way of stating that functional composition is associative is to write

$$(f\,\cdot) \cdot (g\,\cdot) = ((f \cdot g)\cdot)$$

To improve readability we shall often write sections without their full complement of brackets. For example, the above equation can be written as

$$(f\,\cdot) \cdot (g\,\cdot) = (f \cdot g)\cdot$$

The identity element of $\oplus : \alpha \times \alpha \to \alpha$, if it exists, will be denoted by $id_\oplus$. Thus,

$$a \oplus id_\oplus = id_\oplus \oplus a = a$$

However, the identity element of functional composition (over functions of type $\alpha \to \alpha$) will be denoted by id_α. Thus

$$id_\alpha\, a = a$$

for all a in α.

The constant valued function $K : \alpha \to \beta \to \alpha$ is defined by the equation

$$K\, a\, b = a$$

for all a in α and b in β. We sometimes write K_a as an alternative to $K\,a$.

1.2 Lists

Lists are finite sequences of values of the same type. We use the notation $[\alpha]$ to describe the type of lists whose elements have type α. Lists will be denoted using square brackets; for example $[1,2,1]$ is a list of three numbers, and $[[1],[1,2],[1,2,1]]$ is a list of three lists of numbers. The function $[\cdot] : \alpha \to [\alpha]$ maps elements of α into singleton lists. Thus

$$[\cdot]a = [a]$$

The primitive operation on lists is concatenation, denoted by the sign $+\!\!+$. For example:

$$[1] +\!\!+ [2] +\!\!+ [1] = [1,2,1]$$

Concatenation is an associative operation, that is,

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

for all lists x, y and z in $[\alpha]$.

In the majority of situations (though not all) it is convenient to assume the existence of a special list, denoted by $[\,]$ and called the *empty* list, which acts as the identity element of concatenation. Thus,

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

for all x in $[\alpha]$. To distinguish this possibility, we shall let $[\alpha]$ denote the type of lists over α including $[\,]$, and $[\alpha]^+$ the type of lists over α excluding $[\,]$. Using algebraic terminology, $([\alpha], +\!\!+, [\,])$ is a *monoid*, while $([\alpha]^+, +\!\!+)$ is a *semigroup*.

In order to specify functions over lists we need one more assumption, namely that $([\alpha], +\!\!+, [\,])$ is the *free* monoid generated by α under the assignment $[\cdot] : \alpha \rightarrow [\alpha]$. This algebraic statement is equivalent to the assertion that for each function $f : \alpha \rightarrow \beta$ and associative operator $\oplus : \beta \times \beta \rightarrow \beta$, the three equations

$$\begin{array}{lcl} h[\,] & = & id_\oplus \\ h[a] & = & f\,a \\ h(x +\!\!+ y) & = & h\,x \oplus h\,y \end{array}$$

specify a unique function $h : [\alpha] \rightarrow \beta$. In the case that $id_\oplus$ is not defined, the last two equations by themselves determine a unique function $h : [\alpha]^+ \rightarrow \beta$.

Any function h satisfying the first and third equations above is, by definition, a *homomorphism* from the monoid $([\alpha], +\!\!+, [\,])$ to the monoid $(\beta, \oplus, id_\oplus)$. The statement that $([\alpha], +\!\!+, [\,])$ is free is equivalent to the statement that h is uniquely determined by its values on singletons. We shall discuss homomorphisms in greater detail in the next lecture.

One simple example of a homomorphism is provided by the function $\# : [\alpha] \rightarrow N$ which returns the *length* of a list. Here N denotes the natural numbers $\{0, 1, \ldots\}$. We have

$$\begin{array}{lcl} \#[\,] & = & 0 \\ \#[a] & = & 1 \\ \#(x +\!\!+ y) & = & \#x + \#y \end{array}$$

Observe that $+$ is associative with identity 0, so $(N, +, 0)$ is a monoid.

1.3 Bags and sets

By definition, a (finite) *bag* is a list in which the order of the elements is ignored. Bags are constructed by adding the rule that $+\!\!+$ is commutative as well as associative. Also by definition, a (finite) *set* is a bag in which repetitions of elements are ignored. Sets are constructed by adding the rule that $+\!\!+$ is idempotent as well as commutative and associative. As we shall see, much of the theory developed below holds for bags and sets as well as lists.

In the main we shall use the single operator $+\!\!+$ for all three structures, relying on context to resolve ambiguity. However, in order to distinguish different uses in one and the same expression, we shall sometimes use $\uplus$ (bag union) for the concatenation operator on bags, and $\cup$ (set union) for the same operator on sets. Singleton bags are denoted by $\wr a \int$ and singleton sets by $\{a\}$.

A similar algebraic statement about freeness holds for bags and sets as well as lists. We assume that $(\wr\alpha\int, \uplus, \wr\,\int)$ is the free commutative monoid generated by α under the assignment $\wr\cdot\int : \alpha \to \wr\alpha\int$. Similarly, $(\{\alpha\}, \cup, \{\,\})$ is the free commutative and idempotent monoid generated by α under the assignment $\{\cdot\} : \alpha \to \{\alpha\}$. In the case of bags this means that for each $f : \alpha \to \beta$ and associative and commutative operator $\oplus : \beta \times \beta \to \beta$, the equations

$$\begin{aligned} h\wr\,\int &= id_\oplus \\ h\wr a\int &= f\,a \\ h(x \uplus y) &= h\,x \oplus h\,y \end{aligned}$$

define a unique function $h : \wr\alpha\int \to \beta$. Similar remarks apply to sets, except that we also require $\oplus$ to be idempotent.

For example, the *size* of a bag is the number of elements it contains, counting repetitions. It can be defined by the equations

$$\begin{aligned} \#\wr\,\int &= 0 \\ \#\wr a\int &= 1 \\ \#(x \uplus y) &= \#x + \#y \end{aligned}$$

However, although $+$ is associative and commutative, it is not idempotent, so the same equations do not define the size function on sets.

1.4 Map

The operator $*$ (pronounced 'map') takes a function on its left and a list on its right. Informally, we have

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

Formally, we specify $f*$ by three equations

$$\begin{array}{lcl} f * [\,] & = & [\,] \\ f * [a] & = & [f\ a] \\ f * (x +\!\!+\, y) & = & (f * x) +\!\!+\, (f * y) \end{array}$$

Thus, for $f : \alpha \to \beta$ the function $f*$ is a homomorphism from $([\alpha], +\!\!+, [\,])$ to $([\beta], +\!\!+, [\,])$. This function is a homomorphism on bags and sets as well as lists.

An important property of $*$ is that it distributes (backwards) through functional composition:

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

This fact will be referred to as the ($*$ *distributivity*) rule. Its use in calculations is so frequent that we shall sometimes employ it without explicit mention.

1.5 Reduce

The operator $/$ (pronounced 'reduce') takes a binary operator on its left and a list on its right. Informally, we have

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

Formally, we specify $\oplus/$, where $\oplus$ is associative, by three equations

$$\begin{array}{lcll} \oplus/[\,] & = & id_\oplus & \quad \text{(if } id_\oplus \text{ exists)} \\ \oplus/[a] & = & a & \\ \oplus/(x +\!\!+\, y) & = & (\oplus/x) \oplus (\oplus/y) & \end{array}$$

If $\oplus$ is commutative as well as associative, then $\oplus/$ can be applied to bags; and if $\oplus$ is also idempotent, then $\oplus/$ can be applied to sets.

If $\oplus$ does not have an identity element, then we can regard $\oplus/$ as a function of type $\oplus/ : [\alpha]^+ \to \alpha$. Alternatively, we can invent an extra value and adjoin it to α. Provided a little care is taken, so-called 'fictitious' identity elements can always be added to a domain. For example, the minimum operator $\downarrow$ defined by

$$\begin{aligned} a \downarrow b &= a \quad \text{if } a \leqslant b \\ &= b \quad \text{otherwise} \end{aligned}$$

has no identity element in R (the domain of real numbers). However, the fictitious number ∞ can be adjoined to R. The only property attributed to ∞ is that

$$a \downarrow \infty = \infty \downarrow a = a$$

for all a in $R \cup \{\infty\}$. In this way, inconsistency is avoided.

Two useful functions where we choose not to invent identity elements are given by

$$\begin{aligned} \mathit{head} &= \ll / \\ \mathit{last} &= \gg / \end{aligned}$$

wbere the operators $\ll$ ('left') and $\gg$ ('right') are defined by

$$\begin{aligned} a \ll b &= a \\ a \gg b &= b \end{aligned}$$

The function *head* selects tbe first element of a non-empty list, while *last* selects the last element. Both $\ll$ and $\gg$ are associative but neither possesses an identity element. Both operators are idempotent, but neither is commutative, so *head* and *last* are not defined on bags and sets.

1.6 Promotion

The equations defining $f*$ and $\oplus/$ can be expressed as identities between functions. They come in three groups:

empty rules

$$\begin{aligned} f* \cdot K[\,] &= K[\,] \\ \oplus/ \cdot K[\,] &= id_{\oplus} \end{aligned}$$

one-point rules

$$\begin{aligned} f* \cdot [\cdot] &= [\cdot] \cdot f \\ \oplus/ \cdot [\cdot] &= id \end{aligned}$$

join rules

$$\begin{array}{rcl} f* \cdot +\!\!+/ & = & +\!\!+/ \cdot (f*)* \\ \oplus/ \cdot +\!\!+/ & = & \oplus/ \cdot (\oplus/)* \end{array}$$

The interesting rules here are the last two, since they are not simple transcriptions of the third equations for $f*$ and $\oplus/$. A rough and ready justification of the join rule for $f*$ is as follows:

$$\begin{array}{cl} & f * +\!\!+/[x_1, x_2, \ldots, x_n] \\ = & f * (x_1 +\!\!+ x_2 +\!\!+ \cdots +\!\!+ x_n) \\ = & (f * x_1) +\!\!+ (f * x_2) +\!\!+ \cdots +\!\!+ (f * x_n) \\ = & +\!\!+/[f * x_1, f * x_2, \ldots, f * x_n] \\ = & +\!\!+/(f*) * [x_1, x_2, \ldots, x_n] \end{array}$$

A similar justification can be given for the join rule for $\oplus/$. We shall refer to these two rules as *map promotion* and *reduce promotion* respectively. The nomenclature is historical and is intended to express the idea that an operation on a compound structure can be 'promoted' into its components.

The map and promotion rules are often applied in sequence. Consider the following little calculation:

$$\begin{array}{rcl} \oplus/ \cdot f* \cdot +\!\!+/ \cdot g* & = & \text{map promotion} \\ & & \oplus/ \cdot +\!\!+/ \cdot f ** \cdot g* \\ & = & \text{reduce promotion} \\ & & \oplus/ \cdot \oplus/* \cdot f ** \cdot g* \\ & = & * \text{ distributivity} \\ & & \oplus/ \cdot (\oplus/ \cdot f* \cdot g)* \end{array}$$

These three steps will, in future, be compressed into one under the name *map and reduce promotion*.

1.7 Directed reductions

We now introduce two more operators $\not\rightarrow$ (pronounced 'left-to-right reduce', or just 'left reduce') and $\not\leftarrow$ ('right-to-left reduce') which are closely related to $/$. Informally, we have

$$\begin{array}{rcl} \oplus \not\rightarrow_e [a_1, a_2, \ldots, a_n] & = & ((e \oplus a_1) \oplus a_2) \oplus \cdots \oplus a_n \\ \oplus \not\leftarrow_e [a_1, a_2, \ldots, a_n] & = & a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e)) \end{array}$$

In particular, we have

$$\begin{array}{lcl} \oplus \not\rightarrow_e [\,] & = & e \\ \oplus \not\leftarrow_e [\,] & = & e \end{array}$$

Thus, the parameter e (called the *starting value* or *seed*) defines the value of the reduction on empty lists.

Notice in a left-reduce that the brackets group from the left, and in a right-reduce they group from the right. Since an order of association is provided, the operator $\oplus$ of a left or right-reduction need not be associative. In fact, in a left-reduce we can have $\oplus : \beta \times \alpha \rightarrow \beta$ and in a right-reduce $\oplus : \alpha \times \beta \rightarrow \beta$. Thus, for a given seed e in β, the types of $\not\rightarrow_e$ and $\not\rightarrow_e$ are

$$\begin{array}{lcl} \not\rightarrow_e & : & (\beta \times \alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta \\ \not\leftarrow_e & : & (\alpha \times \beta \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta \end{array}$$

Formally, we can define $\oplus\not\rightarrow_e$ on lists by two equations:

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

These equations are different in form to previous definitions, reflecting the fact that $\oplus\not\rightarrow_e$ is *not* a homomorphism on lists. However, since every non-empty list can be expressed uniquely in the form $x +\!\!+ [a]$, these two equations determine $\oplus\not\rightarrow_e$ uniquely. (This point is dealt with in the third lecture). A similar definition holds for $\oplus\not\leftarrow_e$.

Both kinds of reduction can be applied to bags and sets provided $\oplus$ satisfies additional conditions, designed to ensure that the result of a directed reduction is independent of any particular representation of the bag or set. For $\oplus\not\rightarrow_e$ to be defined on bags we require that

$$(b \oplus a_1) \oplus a_2 = (b \oplus a_2) \oplus a_1$$

for all b in β and a_1, a_2 in α. For sets we need the extra condition

$$(b \oplus a) \oplus a = b \oplus a$$

Similar conditions are required for right-reductions. These remarks are given for completeness for we shall have little occasion to apply directed reductions to bags or sets.

It is convenient for some purposes to define a more restricted form of directed reduction in which the seed is omitted. Informally, we have that

$$\begin{array}{lcl} \oplus \not\rightarrow [a_1, a_2, \ldots, a_n] & = & ((a_1 \oplus a_2) \oplus a_3) \oplus \cdots \oplus a_n \\ \oplus \not\leftarrow [a_1, a_2, \ldots, a_n] & = & a_1 \oplus (a_2 \oplus \cdots \oplus (a_{n-1} \oplus a_n)) \end{array}$$

Note that the type of $\oplus$ in this kind of directed reduce must be of the form $\oplus : \alpha \times \alpha \rightarrow \alpha$. The value of $\oplus \not\rightarrow []$ is not defined unless $\oplus$ possesses a unique *left-identity* element, i.e. a value e satisfying

$$e \oplus a = a$$

for all a. If such a value exists and is unique, we can set

$$\oplus \not\rightarrow [] = e$$

Similarly, $\oplus \not\leftarrow []$ is defined only if $\oplus$ has a unique right identity element.

There are a number of properties relating the various forms of directed reduction. For example, we have

$$\begin{array}{rcl} (\oplus\not\rightarrow) \cdot ([a] +\!\!+) & = & \oplus\not\rightarrow_a \\ (\oplus\not\leftarrow) \cdot (+\!\!+ [a]) & = & \oplus\not\leftarrow_a \end{array}$$

Other properties will be considered in a later lecture. For the present we give just one illustration of the use of left-reduce.

Recall from the first section that the right-hand side of Horner's rule reads

$$(((1 \times a_1 + 1) \times a_2 + 1) \times \cdots + 1) \times a_n + 1$$

This expression can be written using a left-reduce:

$$\circledast \not\rightarrow_1 [a_1, a_2, \ldots, a_n]$$

where the operator $\circledast$ is defined by

$$a \circledast b = (a \times b) + 1$$

It is interesting to compare this with another form of Horner's rule:

$$(a_1 \times a_2 \times a_3) + (a_2 \times a_3) + a_3 = ((a_1 \times a_2 + a_2) \times a_3 + a_3$$

Here, the general form of the right-hand side can be written as

$$\circledast \not\rightarrow [a_1, a_2, \ldots, a_n],$$

where, this time, $\circledast$ is defined by

$$a \circledast b = (a \times b) + b$$

1.8 Accumulations

With each form of directed reduction over lists there corresponds a form of computation called an *accumulation*. These forms are expressed with the operators $\mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}$ ('left-accumulate') and $\mathbin{\leftarrow\!\!\!\!\!\!/\!\!\!/}$ ('right-accumulate') and are defined informally by

$$\begin{array}{lcl} \oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e [a_1, a_2, \ldots, a_n] & = & [e, e \oplus a_1, \ldots, ((e \oplus a_1) \oplus a_2) \oplus \cdots \oplus a_n] \\ \oplus \mathbin{\leftarrow\!\!\!\!\!\!/\!\!\!/}_e [a_1, a_2, \ldots, a_n] & = & [a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e)), \ldots, a_n \oplus e, e] \end{array}$$

Formally, we have

$$\begin{array}{lcl} \oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e [\,] & = & [e] \\ \oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e ([a] \mathbin{+\!\!\!+} x) & = & [e] \mathbin{+\!\!\!+} (\oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_{e \oplus a}\, x) \end{array}$$

Alternatively, we can define a left-accumulation by

$$\begin{array}{lcl} \oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e [\,] & = & [e] \\ \oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e (x \mathbin{+\!\!\!+} [a]) & = & (\oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e\, x) \mathbin{+\!\!\!+} [b \oplus a] \\ & & \text{where } b = \mathit{last}(\oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e\, x) \end{array}$$

Yet a third way of defining $\oplus\mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e$ will be given in the next section. The definitions of the other accumulation operators are similiar and we omit details.

Observe from the above equations that $\oplus \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e\, x$ can be evaluated with n calculations of $\oplus$, where $n = \#x$. For example, the list $[0!, 1!, \ldots, n!]$ of the first $(n+1)$ factorial numbers can be computed by evaluating

$$\times \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_1 [1, 2, \ldots, n]$$

This requires $O(n)$ steps, whereas the alternative

$$\mathit{fact} * [0, 1, \ldots, n],$$

where $\mathit{fact}\; k = \times/[1, 2, \ldots, k]$, requires $O(n^2)$ steps. Also amusing is the fact that

$$\div \mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_1 [1, \ldots, n] = [\frac{1}{0!}, \ldots, \frac{1}{n!}]$$

Note that

$$\oplus\mathbin{\rightarrow\!\!\!\!\!\!/}_e = \mathit{last} \cdot \oplus\mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e$$

so left-reductions can be defined in terms of left-accumulations. On the other hand, we also have

$$\oplus\mathbin{/\!\!\!/\!\!\!\!\!\!\rightarrow}_e = \otimes\mathbin{\rightarrow\!\!\!\!\!\!/}_{[e]}$$

where

$$x \otimes a = x \mathbin{+\!\!+} [\mathit{last}\, x \oplus a]$$

Hence left-accumulations can be defined in terms of left-reductions.

1.9 Segments

A list y is a *segment* of x if there exist u and v such that $x = u \mathbin{+\!\!+} y \mathbin{+\!\!+} v$. If $u = [\,]$, then y is an *initial segment*, and if $v = [\,]$, then y is a *final segment*.

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, informally, we have

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

The functions inits^+ and tails^+ are similar, except that the empty list does not appear in the result.

These four functions can be defined by accumulations. For example,

$$\begin{array}{lcl} \mathit{inits} & = & (\mathbin{+\!\!+}\!\not\!\!\!/\!\!\!/_{[\,]}) \cdot [\cdot]* \\ \mathit{inits}^+ & = & (\mathbin{+\!\!+}\!\not\!\!\!/\!\!\!/) \cdot [\cdot]* \end{array}$$

Alternatively, we can define these functions by explicit recursion equations. For example:

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

The following result shows another way we can define accumulations.

Accumulation lemma

$$\begin{array}{lcl} (\oplus\!\not\!\!\!/\!\!\!/_e) & = & (\oplus\!\not\!\!\!/_e) * \cdot\, \mathit{inits} \\ (\oplus\!\not\!\!\!/\!\!\!/) & = & (\oplus\!\not\!\!\!/) * \cdot\, \mathit{inits}^+ \end{array}$$

There are similar equations for right-accumulations. The accumulation lemma is used frequently in the derivation of efficient algorithms for problems about segments. On lists of length n, evaluation of the left-hand side requires $O(n)$ computations involving $\oplus$, while the right-hand side requires $O(n^2)$ computations.

The functions *segs* returns a list of all segments of a list, and $segs^+$ returns a list of all non-empty segments. A convenient definition is

$$segs = +\!\!+/ \cdot tails * \cdot inits$$

For example,

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

Notice that the empty list $[\,]$ appears four times in $segs[1,2,3]$ (and not at all in $segs^+[1,2,3]$). The order in which the elements of $segs\,x$ appear is not important for our purposes and we shall make no use of it. In effect, we regard $segs\,x$ as a bag of lists. This identification can be made explicit by introducing a 'bagifying' function

$$bag = \uplus/ \cdot \lbag \cdot \rbag *$$

which converts a list into a bag of its elements (and is the identity function on bags). We can then define

$$bagsegs = bag \cdot segs$$

However, explicit use of *bag* can be avoided in many examples. Consider a function of the form

$$P = \oplus/ \cdot f * \cdot bagsegs$$

where we must have that $\oplus$ is commutative as well as associative. We can calculate

$$\begin{array}{lcl} P & = & \text{definition of } bagsegs \\ & & \oplus/ \cdot f * \cdot \uplus/ \cdot \lbag \cdot \rbag * \cdot segs \\ & = & \text{map and reduce promotion} \\ & & \oplus/ \cdot (\oplus/ \cdot f * \cdot \lbag \cdot \rbag) * \cdot segs \\ & = & \text{one-point rules} \\ & & \oplus/ \cdot f * \cdot segs \end{array}$$

and so *bagsegs* can be replaced by *segs*.

1.10 Horner's rule

Now let us return to the problem posed at the beginning. Horner's rule can be expressed as an equation

$$\oplus/ \cdot \otimes/ * \cdot tails = \circledast \not\rightarrow_e$$

where $e = id_{\otimes}$ and $a \circledast b = (a \otimes b) \oplus e$. This equation is valid provided $\otimes$ distributes (backwards) over $\oplus$, that is,

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

for all a, b and c of the appropriate type. This condition is equivalent to the assertion that the equation

$$(\otimes c) \cdot \oplus/ = \oplus/ \cdot (\otimes c)*$$

holds on all non-empty lists. If we also assume that $id_{\oplus}$ is a *left-zero* element of $\otimes$, i.e.,

$$id_{\oplus} \otimes c = id_{\oplus}$$

for all c, then the restriction to non-empty lists in the above assertion can be dropped.

Horner's rule is proved by induction. The idea is to show that

$$f = \oplus/ \cdot \otimes/* \cdot \mathit{tails}$$

satisfies the equations

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

The way to do this is to use the recursive characterisation of *tails* given in the previous section. We shall leave details to the reader.

Horner's rule can be generalised in a number of ways. We cite just two. First, we have

$$\oplus/ \cdot \otimes/* \cdot \mathit{tails}^{+} = \circledast \not\rightarrow$$

where $a \circledast b = (a \otimes b) \oplus b$. This formulation, which was hinted at in Section 1.8, does not require that $id_{\otimes}$ be defined. We also have

$$\oplus/ \cdot (\otimes/ \cdot f*)* \cdot \mathit{tails} = \circledast \not\rightarrow_{e}$$

where $e = id_{\otimes}$ and $a \circledast b = (a \otimes f\, b) \oplus e$. This particular form of Horner's rule will be used in the next lecture. There are also forms of the rule involving right-reductions and *inits*.

1.11 Application

Let us give one application of Horner's rule. There is a famous problem, called the *maximum segment sum* (*mss*) problem, which is to compute the maximum of the sums of all segments of a given sequence of numbers, positive, negative or zero. In symbols

$$mss = \uparrow/ \cdot +/* \cdot segs$$

Direct evaluation of the right-hand side of this equation requires $O(n^3)$ steps on a list of length n. There are $O(n^2)$ segments and each can be summed in $O(n)$ steps, giving $O(n^3)$ steps in all. Using Horner's rule it is easy to calculate an $O(n)$ algorithm:

$$\begin{array}{lll}
mss & = & \text{definition} \\
 & & \uparrow/ \cdot +/* \cdot segs \\
 & = & \text{definition of } segs \\
 & & \uparrow/ \cdot +/* \cdot +\!\!+/ \cdot tails * \cdot inits \\
 & = & \text{map and reduce promotion} \\
 & & \uparrow/ \cdot (\uparrow/ \cdot +/* \cdot tails) * \cdot inits \\
 & = & \text{Horner's rule with } a \circledast b = (a + b) \uparrow 0 \\
 & & \uparrow/ \cdot \circledast \not\rightarrow_0 * \cdot inits \\
 & = & \text{accumulation lemma} \\
 & & \uparrow/ \cdot \circledast \not\!\not\rightarrow_0
\end{array}$$

Horner's rule is applicable because $+$ distributes through $\uparrow$, and $0 = id_+$. The result is a linear time algorithm.

An interesting variation of the problem is not so well-known. It is to compute the maximum segment *product*. In symbols

$$msp = \uparrow/ \cdot \times/* \cdot segs$$

Since $\times$ does not distribute through $\uparrow$ for negative numbers, the previous derivation does not work. However, we do have

$$\begin{array}{lll}
(a \uparrow b) \times c & = & (a \times c) \uparrow (b \times c) \quad \text{if } c \geqslant 0 \\
(a \uparrow b) \times c & = & (a \times c) \downarrow (b \times c) \quad \text{if } c \leqslant 0
\end{array}$$

where $\downarrow$ takes the minimum of its two arguments. A similar pair of equations holds for $(a \downarrow b) \times c$. These facts are enough to ensure that, with suitable cunning, Horner's rule can be made to work. The idea is to define $\oplus$ by

$$(a_1, b_1) \oplus (a_2, b_2) = (a_1 \downarrow a_2, b_1 \uparrow b_2)$$

and $\otimes$ by

$$\begin{aligned}(a,b)\otimes c &= (a\times c, b\times c) \quad \text{if } c \geqslant 0\\ &= (b\times c, a\times c) \quad \text{otherwise}\end{aligned}$$

Then, using the observations about $\uparrow$ and $\downarrow$ given above, we have that

$$((a_1,b_1)\oplus(a_2,b_2))\otimes c = ((a_1,b_1)\otimes c)\oplus((a_2,b_2)\otimes c)$$

and so $\otimes$ distributes backwards through $\oplus$.

Now define

$$f\,x = (\downarrow/(\times/*\,segs\,x), \uparrow/(\times/*\,segs\,x))$$

A similar calculation to before shows that

$$f = \oplus/\cdot \circledast\!\!\not\!\!\!\rightarrow_e$$

where $e = (1,1)$, and

$$(a,b)\circledast c = ((a,b)\otimes c)\oplus(1,1)$$

Hence we have

$$msp = \pi_2\cdot\oplus/\cdot\circledast\!\!\not\!\!\!\rightarrow_e$$

where $\pi_2(a,b) = b$. Again this is a linear time algorithm.

1.12 Segment decomposition

The sequence of calculation steps given in the derivation of the *mss* problem arises frequently. Here is the essential idea expressed as a general theorem.

Theorem 1 (Segment decomposition) *Suppose S and T are defined by*

$$\begin{aligned}S &= \oplus/\cdot f*\cdot segs\\ T &= \oplus/\cdot f*\cdot tails\end{aligned}$$

If T can be expressed in the form $T = h\cdot\circledast\!\!\not\!\!\!\rightarrow_e$, then we have

$$S = \oplus/\cdot h*\cdot\circledast\!\!\not\!\!\!\!\!\rightarrow_e$$

Proof

We calculate

$$
\begin{array}{rcl}
S & = & \text{given} \\
 & & \oplus/ \cdot f* \cdot \mathit{segs} \\
 & = & \text{definition of } \mathit{segs} \\
 & & \oplus/ \cdot f* \cdot +\!\!\!+/ \cdot \mathit{tails}* \cdot \mathit{inits} \\
 & = & \text{map and reduce promotion} \\
 & & \oplus/ \cdot (\oplus/ \cdot f* \cdot \mathit{tails})* \cdot \mathit{inits} \\
 & = & \text{hypothesis on } T \\
 & & \oplus/ \cdot (h \cdot \circledast\!\not\rightarrow_e)* \cdot \mathit{inits} \\
 & = & * \text{ distributivity} \\
 & & \oplus/ \cdot h* \cdot \circledast\!\not\rightarrow_e * \cdot \mathit{inits} \\
 & = & \text{accumulation lemma} \\
 & & \oplus/ \cdot h* \cdot \circledast\, /\!\!\!\not\rightarrow_e
\end{array}
$$

□

1.13 References

Much of the foregoing theory is introduced in

[1] Bird, R.S. An introduction to the theory of lists. *Logic of Programming and Calculi of Discrete Design*, (edited by M. Broy), Springer-Verlag, (1987) 3-42.

An earlier account, using somewhat different syntactic conventions, is in

[2] 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.

An alternative treatment of some of the concepts, stressing the use of the map and reduction operators in functional programming, is in the recent textbook

[3] Bird, R.S. and Wadler, P.L. *Introduction to Functional Programming*. Prentice Hall International, Hemel Hempstead, UK, (1988)

The maximum segment sum problem is well-known and is discussed in, among other places,

[4] Bentley, J.L. *Programming Pearls* (Chapter 7) Addison-Wesley, Reading, Mass, USA (1986)

[5] Gries, D. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming* 2 (1982) 207-214

The maximum product problem appears as one of M. Rem's exercises in

[6] Rem, M. Small programming exercises. *Science of Computer Programming* (various issues)

2 Homomorphisms

2.0 Problem

Given is a sequence x and a predicate p. Required is an efficient algorithm for computing some longest segment of x, all of whose elements satisfy p.

2.1 Homomorphisms

By definition, a *homomorphism* from a monoid $(\alpha, \oplus, id_\oplus)$ to a monoid $(\beta, \otimes, id_\otimes)$ is a function h satisfying the two equations

$$\begin{array}{lcl} h\, id_\oplus & = & id_\otimes \\ h(x \oplus y) & = & h\,x \otimes h\,y \end{array}$$

Equivalently, using the map and reduction operators introduced in the first lecture, h is a homomorphism if

$$h \cdot \oplus/ = \otimes/ \cdot h*$$

We omit the proof (by induction) that these two definitions are equivalent.

Since the map function $f*$ is a homomorphism from monoid $([\alpha], +\!\!+, [\,])$ to monoid $([\beta], +\!\!+, [\,])$ whenever $f : \alpha \rightarrow \beta$, we have, as an immediate consequence of the second characterisation of homomorphisms, that

$$f* \cdot +\!\!+/ = +\!\!+/ \cdot f**$$

This is just the map promotion rule of the previous lecture. Likewise, the reduce promotion rule is an immediate consequence of the fact that $\oplus/$ is a homomorphism.

Our basic assumption, namely that $([\alpha], +\!\!+, [\,])$ is a *free* monoid, is equivalent to the condition that for each monoid $(\beta, \oplus, id_\oplus)$ there is a unique homomorphism h from $([\alpha], +\!\!+, [\,])$ to $(\beta, \oplus, id_\oplus)$. This homomorphism is determined by the values of h on singletons. That is, for each $f : \alpha \rightarrow \beta$, the additional equation

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

fixes h completely.

The following lemma gives a useful characterisation of homomorphisms on lists.

Lemma 2 *Every homomorphism on lists can be expressed as the composition of a reduction with a map, and every such combination is a homomorphism. More precisely, suppose*

$$\begin{array}{lcl} h[\,] & = & id_\oplus \\ h[a] & = & f\ a \\ h(x +\!\!+ y) & = & h\ x \oplus h\ y \end{array}$$

Then $h = \oplus/ \cdot f*$. *Conversely, if* h *has this form, then* h *is a homomorphism.*

Proof

The proof of the homomorphism lemma uses the following simple result (called the *identity lemma*), whose proof will not be given.

$$+\!\!+/ \cdot [\cdot]* = id_{[\alpha]}$$

We now calculate:

$$\begin{array}{lcl} h & = & \text{definition of } id \\ & & h \cdot id \\ & = & \text{identity lemma} \\ & & h \cdot +\!\!+/ \cdot [\cdot]* \\ & = & h \text{ is a homomorphism} \\ & & \oplus/ \cdot h* \cdot [\cdot]* \\ & = & * \text{ distributivity} \\ & & \oplus/ \cdot (h \cdot [\cdot])* \\ & = & \text{definition of } h \text{ on singletons} \\ & & \oplus/ \cdot f* \end{array}$$

Conversely, we reason that $h = \oplus/ \cdot f*$ is a homomorphism by calculating

$$\begin{array}{rcl} h \cdot +\!\!+/ & = & \text{given form for } h \\ & & \oplus/ \cdot f* \cdot +\!\!+/ \\ & = & \text{map and reduce promotion} \\ & & \oplus/ \cdot (\oplus/ \cdot f*)* \\ & = & \text{hypothesis} \\ & & \oplus/ \cdot h* \end{array}$$

□

Many functions on lists are homomorphisms and we shall see examples below. However, not all of the functions we can define in terms of homomorphisms will themselves be homomorphisms. We shall take up this point in the following lecture.

2.2 Examples

Let us consider some examples of homomorphisms on lists.

(1) First of all, the function # is a homomorphism:

$$\# = +/ \cdot K_1*$$

(2) Second, the function *reverse* which reverses the order of the elements in a list is a homomorphism:

$$reverse = \widetilde{+\!\!+}/ \cdot [\cdot]*$$

where $x \widetilde{+\!\!+} y = y +\!\!+ x$. (In general, we define $\widetilde{\oplus}$ by the equation $x \widetilde{\oplus} y = y \oplus x$.) Of course, on bags and sets, where $\widetilde{+\!\!+} = +\!\!+$, the function *reverse* is just the identity function.

(3) The function *sort* which reorders the elements of a list into ascending order is a homomorphism:

$$sort = \wedge\!\!\!\wedge/ \cdot [\cdot]*$$

Here, $\wedge\!\!\!\wedge$ (pronounced 'merge') is defined by the equations

$$\begin{array}{lcll} x \wedge\!\!\!\wedge [\,] & = & x & \\ [\,] \wedge\!\!\!\wedge y & = & y & \\ ([a] +\!\!+ x) \wedge\!\!\!\wedge ([b] +\!\!+ y) & = & [a] +\!\!+ (x \wedge\!\!\!\wedge ([b] +\!\!+ y)) & \text{if } a \leqslant b \\ & = & [b] +\!\!+ (([a] +\!\!+ x) \wedge\!\!\!\wedge y) & \text{otherwise} \end{array}$$

Thus, $x \wedge\!\!\wedge y$ is the result of merging two sorted lists x and y. Since $\wedge\!\!\wedge$ is both associative and commutative, the function *sort* can be applied to bags. By defining a variant of $\wedge\!\!\wedge$ that removes duplicates, so that the operation is also idempotent, we can sort sets.

(4) Two useful homomorphisms are *all* and *some*:

$$\begin{aligned} \mathit{all}\, p &= \wedge/ \cdot p* \\ \mathit{some}\, p &= \vee/ \cdot p* \end{aligned}$$

Here, $\wedge$ is logical conjunction and $\vee$ is logical disjunction. The function *all* p applied to a list x returns *True* if every element of x satisfies the predicate p, and *False* otherwise. The function *some* p applied to x returns *True* if at least one element of x satisfies p, and *False* otherwise. Since conjunction and disjunction are associative, commutative and idempotent operations, *all* and *some* can be applied to bags and sets as well as lists.

(5) The function $\mathit{split} : [\alpha]^+ \to [\alpha] \times \alpha$, which splits a non-empty list into its last element and the remainder, is a homomorphism:

$$\begin{aligned} \mathit{split}[a] &= ([\,], a) \\ \mathit{split}(x +\!\!+ y) &= \mathit{split}\, x \oplus \mathit{split}\, y \end{aligned}$$

where we define $\oplus$ by

$$(x, a) \oplus (y, b) = (x +\!\!+ [a] +\!\!+ y, b)$$

In particular, we can define

$$\mathit{init} = \pi_1 \cdot \mathit{split}$$

where $\pi_1(u, v) = u$. Unlike *last* (which is $\pi_2 \cdot \mathit{split}$), the function *init* is not a homomorphism. Note that the homomorphisms described in this example are homomorphisms on the *semigroup* $([\alpha]^+, +\!\!+)$.

Using *init* and *last*, we can define the function *tails* of the last lecture as a homomorphism

$$\mathit{tails} = \oplus/ \cdot f*$$

where

$$\begin{aligned} f\, a &= [[\,], [a]] \\ xs \oplus ys &= \mathit{init}\, xs +\!\!+ (\mathit{last}\, xs +\!\!+) * ys \end{aligned}$$

A simple calculation shows that $id_\oplus = [[\,]]$, so we have $\mathit{tails}[\,] = [[\,]]$, as expected.

2.3 All applied to

In order to be able to describe homomorphisms, such as *tails*, a little more concisely, it is useful to introduce an operator $^\circ$ (pronounced 'all applied to') defined by

$$\begin{array}{lcl} [\,]^\circ\, a & = & [\,] \\ [f]^\circ\, a & = & [f\ a] \\ (fs \mathbin{+\!\!\!+} gs)^\circ\, a & = & (fs^\circ\, a) \mathbin{+\!\!\!+} (gs^\circ\, a) \end{array}$$

Less formally, we have

$$[f, g, \ldots, h]^\circ\, a = [f\ a, g\ a, \ldots, h\ a]$$

Thus $^\circ$ takes a sequence of functions and a value and returns the result of applying each function to the value. Note that $(^\circ\, a)$ is a homomorphism. Note also that the notation $[\cdot]$ we have been using so far can be rewritten as $[id]^\circ$.

We can now write, for example,

$$tails = \oplus/ \cdot [[\,]^\circ, [id]^\circ]^\circ *$$

2.4 Conditional expressions

So far, we have been using the notation

$$\begin{array}{lcll} h\,x & = & f\,x & \text{if } p\,x \\ & = & g\,x & \text{otherwise} \end{array}$$

to describe functions defined by cases. From now on, we shall also use the McCarthy conditional form

$$h = (p \rightarrow f, g)$$

to describe the same function. Like the operator of the previous section, conditional forms can, in some situations, help to make the expression of homomorphisms and other functions more concise.

There are a number of well-known laws about conditional forms, the most important of which are:

$$\begin{array}{lcl} h \cdot (p \rightarrow f, g) & = & (p \rightarrow h \cdot f, h \cdot g) \\ (p \rightarrow f, g) \cdot h & = & (p \cdot h \rightarrow f \cdot h, g \cdot h) \\ (p \rightarrow f, f) & = & f \end{array}$$

(Remember, all functions are assumed to be *total*, so these laws need no qualifications about definedness.)

2.5 Filter

The operator $\triangleleft$ (pronounced 'filter') takes a predicate p and a list x and returns the sublist of x consisting, in order, of all those elements of x that satisfy p. Using the new notations just introduced, we can define $p\triangleleft$ as a homomorphism

$$p\triangleleft = +\!\!+/ \cdot (p \rightarrow [id]^\circ, []^\circ)*$$

In effect, $p \triangleleft x$ is obtained by replacing each element a of x by $[a]$ if $p\ a$ holds, and $[]$ otherwise, and then concatenating the resulting lists. Note that $p\triangleleft$ can be applied to bags and sets as well as lists.

An easy calculation shows that the following rule (which we will call *filter promotion*) holds:

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

Another rule, whose proof is also left to the reader, is the *map-filter swap* rule:

$$p\triangleleft \cdot f* = f* \cdot (p \cdot f)\triangleleft$$

2.6 Cross-product

If $\oplus$ is a binary operator, then $\mathsf{X}_\oplus$ is a binary operator that takes two lists x and y and returns a list of values of the form $a \oplus b$ for all a in x and b in y. For example:

$$[a, b]\,\mathsf{X}_\oplus\,[c, d, e] = [a \oplus c, b \oplus c, a \oplus d, b \oplus d, a \oplus e, b \oplus e]$$

Formally, we define $\mathsf{X}_\oplus$ by three equations:

$$\begin{array}{lcl} x\,\mathsf{X}_\oplus\,[] & = & [] \\ x\,\mathsf{X}_\oplus\,[a] & = & (\oplus a) * x \\ x\,\mathsf{X}_\oplus\,(y +\!\!+ z) & = & (x\,\mathsf{X}_\oplus\,y) +\!\!+ (x\,\mathsf{X}_\oplus\,z) \end{array}$$

Thus $(x\mathsf{X}_\oplus)$ is a homomorphism (on lists, bags or sets) for every x.

There are a number of useful properties of $\mathsf{X}_\oplus$. We shall state them without proof.

First of all, $\mathsf{X}_\oplus$ is associative if $\oplus$ is, and commutative if $\oplus$ is. It is not, in general, idempotent if $\oplus$ is.

Next, $[]$ is the zero element of $\mathsf{X}_\oplus$, that is,

$$[]\,\mathsf{X}_\oplus\,x = x\,\mathsf{X}_\oplus\,[] = []$$

for all x.
We also have the *cross promotion* rules:

$$\begin{aligned} f** \cdot \mathsf{X}_{+\!\!+}/ &= \mathsf{X}_{+\!\!+}/ \cdot f*** \\ \oplus/* \cdot \mathsf{X}_{+\!\!+}/ &= \mathsf{X}_{\oplus}/ \cdot \oplus/* \end{aligned}$$

Finally, we have that if $\otimes$ distributes through $\oplus$, then

$$\oplus/ \cdot \mathsf{X}_{\otimes}/ = \otimes/ \cdot \oplus/*$$

This result says that the sum of the products is the product of the sums. We shall call it the *cross-distributivity* rule.

The particular operator $\mathsf{X}_{+\!\!+}$ has many uses. For example, the *cartesian product* function $cp : [[\alpha]] \to [[\alpha]]$, defined by

$$cp = \mathsf{X}_{+\!\!+}/ \cdot [id]^{\circ}*$$

takes a list of lists and returns a list of lists of elements, one from each component. For example,

$$cp[[a,b],[c],[d,e]] = [[a,c,d],[b,c,d],[a,c,e],[b,c,e]]$$

Second, the list $subs\, x$ of all subsequences of x can be defined as the homomorphism

$$subs = \mathsf{X}_{+\!\!+}/ \cdot [[\,]^{\circ},[id]^{\circ}]^{\circ}*$$

For example

$$subs[a,b,c] = \mathsf{X}_{+\!\!+}/[[[\,],[a]],[[\,],[b]],[[\,],[c]]]$$

and the expression on the right simplifies to

$$[[\,],[a],[b],[a,b],[c],[a,c],[b,c],[a,b,c]]$$

Third, we have

$$(all\, p \to [id]^{\circ},[\,]^{\circ}) = \mathsf{X}_{+\!\!+}/ \cdot (p \to [[id]^{\circ}]^{\circ},[\,]^{\circ})*$$

This technical result means that we can write $all\, p\triangleleft$ in the form

$$all\, p\triangleleft = +\!\!+/ \cdot (\mathsf{X}_{+\!\!+}/ \cdot (p \to [[id]^{\circ}]^{\circ},[\,]^{\circ})*)*$$

This daunting expression will make another appearance in the next section but one.

2.7 Selection operators

Suppose f is a numeric valued function. We want to define an operator $\uparrow_f$ such that

1. $\uparrow_f$ is associative, commutative and idempotent;
2. $\uparrow_f$ is *selective* in that
$$x \uparrow_f y = x \quad \text{or} \quad x \uparrow_f y = y$$
3. $\uparrow_f$ is *maximising* in that
$$f(x \uparrow_f y) = f\ x \uparrow f\ y$$

If f is an injective function, then the above three conditions specify $\uparrow_f$ completely (actually, idempotence follows from selectivity). If, however, f is not injective, then the value of $x \uparrow_f y$ is not specified when $x \neq y$ but $f\ x = f\ y$. For example, the value of

$$[1,2] \uparrow_{\#} [3,4]$$

is not determined by the above conditions, beyond the fact that it must be one of $[1,2]$ or $[3,4]$.

There are two ways to resolve such under-specifications. One is to forgo commutativity, defining for instance a *left-biased* version of $\uparrow_f$:

$$\begin{aligned} x \uparrow_f y &= x \quad \text{if } f\ x \geqslant f\ y \\ &= y \quad \text{otherwise} \end{aligned}$$

This solution is not very satisfactory because the calculation of expressions such as

$$\uparrow_{\#}/ \cdot p \triangleleft \cdot \mathit{segs}$$

depends artificially on the precise order in which the function *segs* returns the list of segments of x (a feature which we said in the last lecture we would ignore).

The alternative is to let $\uparrow_f$ stand for $\uparrow_{f'}$, where f' is an injective function, the precise nature of which we are not interested in, that respects the ordering on values given by f, that is,

$$f\ x < f\ y \quad \text{implies} \quad f'\ x < f'\ y$$

If necessary to ease a calculation, we can always introduce *refinements* of f (i.e. a function that respects the ordering of f but may introduce further distinctions), provided such refinements are consistent with all previous ones.

One particular refinement of $\uparrow_{\#}$ is especially useful and we impose it at the outset. We shall assume that $+\!\!+$ distributes through $\uparrow_{\#}$, in other words:

$$\begin{array}{rcl} x +\!\!+ (y \uparrow_{\#} z) & = & (x +\!\!+ y) \uparrow_{\#} (x +\!\!+ z) \\ (x \uparrow_{\#} y) +\!\!+ z & = & (x +\!\!+ z) \uparrow_{\#} (y +\!\!+ z) \end{array}$$

Such a refinement arises if, for example, we always select the *lexicographically* least sequence as the value of $x \uparrow_{\#} y$ when $\#x = \#y$.

Since we mainly do calculations at the function level, we would like to write the above distributive rules in the form

$$\begin{array}{rcl} (x+\!\!+) \cdot \uparrow_{\#}/ & = & \uparrow_{\#}/ \cdot (x+\!\!+)* \\ (+\!\!+x) \cdot \uparrow_{\#}/ & = & \uparrow_{\#}/ \cdot (+\!\!+x)* \end{array}$$

The missing piece which enables the two forms to be connected (without restricting ourselves to non-empty lists) concerns the fictitious value $\omega = \uparrow_{\#}/[\,]$. This is not the empty list, but a very short list satisfying $\#\omega = -\infty$. In other words, we want to suppose that ω is the zero element of $+\!\!+$. This decision can be couched in algebraic language: we suppose that $([\alpha], +\!\!+, \uparrow_{\#}, [\,], \omega)$ is a *semiring*. In general, a semiring $(S, \times, +, id_{\times}, id_{+})$ is a set S closed under two associative operations $\times$ and $+$, with $+$ also commutative, such that $\times$ distributes over $+$. Moreover, the identity element of $+$ is the zero element of $\times$.

Let us put these assumptions to work in a short calculation:

$$\begin{array}{cl} & \uparrow_{\#}/ \cdot \mathit{all}\, p\triangleleft \\ = & \text{daunting expression for } \mathit{all}\, p\triangleleft \text{ from Section 2.7} \\ & \uparrow_{\#}/ \cdot +\!\!+/ \cdot (\mathsf{X}_{+\!\!+}/ \cdot (p \rightarrow [[id]^{\circ}]^{\circ}, [\,]^{\circ})*)* \\ = & \text{reduce promotion} \\ & \uparrow_{\#}/ \cdot (\uparrow_{\#}/ \cdot \mathsf{X}_{+\!\!+}/ \cdot (p \rightarrow [[id]^{\circ}]^{\circ}, [\,]^{\circ})*)* \\ = & \text{semiring assumption and cross-distributivity} \\ & \uparrow_{\#}/ \cdot (+\!\!+/ \cdot \uparrow_{\#}/* \cdot (p \rightarrow [[id]^{\circ}]^{\circ}, [\,]^{\circ})*)* \\ = & * \text{ distributivity} \\ & \uparrow_{\#}/ \cdot (+\!\!+/ \cdot (\uparrow_{\#}/ \cdot (p \rightarrow [[id]^{\circ}]^{\circ}, [\,]^{\circ})*)* \\ = & \text{conditionals} \\ & \uparrow_{\#}/ \cdot (+\!\!+/ \cdot (p \rightarrow \uparrow_{\#}/ \cdot [[id]^{\circ}]^{\circ}, \uparrow_{\#}/ \cdot [\,]^{\circ})*)* \\ = & \text{empty and one-point rules} \\ & \uparrow_{\#}/ \cdot (+\!\!+/ \cdot (p \rightarrow [id]^{\circ}, K_{\omega})*)* \end{array}$$

We shall use this result in the next section.

2.8 Solution

The problem we started the lecture with was to compute the longest segment of a list, all of whose elements satisfied some given property p. In symbols, we want to compute f, where

$$f = \uparrow_{\#}/ \cdot \mathit{all}\, p \triangleleft \cdot \mathit{segs}$$

Let us calculate:

$$\begin{array}{cl}
 & \uparrow_{\#}/ \cdot \mathit{all}\, p \triangleleft \cdot \mathit{segs} \\
= & \text{segment decomposition} \\
 & \uparrow_{\#}/ \cdot (\uparrow_{\#}/ \cdot \mathit{all}\, p \triangleleft \cdot \mathit{tails})* \cdot \mathit{inits} \\
= & \text{result at end of last section} \\
 & \uparrow_{\#}/ \cdot (\uparrow_{\#}/ \cdot (+\!\!+/ \cdot (p \rightarrow [\mathit{id}]^{\circ}, K_{\omega})*)* \cdot \mathit{tails})* \cdot \mathit{inits} \\
= & \text{Horner's rule with } x \circledast a = (x +\!\!+ (p\, a \rightarrow [a], \omega)) \uparrow_{\#} [\,] \\
 & \uparrow_{\#}/ \cdot \circledast \not\rightarrow_{[\,]} * \cdot \mathit{inits} \\
= & \text{accumulation lemma} \\
 & \uparrow_{\#}/ \cdot \circledast \not\!\not\rightarrow_{[\,]}
\end{array}$$

Finally, we can simplify $x \circledast a$ to

$$x \circledast a = (p\, a \rightarrow x +\!\!+ [a], [\,])$$

This is a linear time algorithm (in the number of calculations of p).

The derivation of the above program might seem a little elaborate, bringing in cross-products, semirings, fictitious elements and so on, just to crack a small walnut. The central aspect, namely that

$$\uparrow_{\#}/ \cdot \mathit{all}\, p \triangleleft \cdot \mathit{tails}$$

can be expressed as a left-reduction, can be established quite quickly by an induction proof, one that avoids all talk of zero elements of concatenation. However, we have succeeded in avoiding induction, used only equational reasoning, brought in a second application of a useful rule, and introduced some more concepts and notations.

2.9 References

Further discussion of some of the operators introduced above is in:

[1] Bird, R.S. A calculus of functions for program derivation. *Proc. Institute of Declarative Programming*, University of Texas, USA, 1987. (Also available as a Programming Research Group Monograph PRG-64, Oxford, UK.)

An extensive discussion of homomorphisms on trees, lists, bags and sets is in:

[2] Backhouse, R. An exploration of the Bird-Meertens formalism. (*Unpublished draft*), Dept. of Computer Science, Groningen University, The Netherlands. (1988)

3 Left reductions

3.0 Problem

Given is a list of lists of numbers. Required is an efficient algorithm for computing the minimum of the maximum numbers in each list. More succinctly, we want to compute

$$minimax = \downarrow/ \cdot \uparrow/*$$

as efficiently as possible.

3.1 General equations

So far, we have mainly seen examples of homomorphisms. It is instructive to determine the conditions under which a general set of equations

$$\begin{array}{lcl} h[\,] & = & e \\ h[a] & = & f\ a \\ h(x \mathbin{+\!\!+} y) & = & H(x, y, f\ x, f\ y) \end{array}$$

determines a unique function h, not necessarily a homomorphism. After all, such sets of equations constitute the basic means for specifying functions on lists.

Consider the equations

$$\begin{array}{lcl} h'[\,] & = & ([\,], e) \\ h'[a] & = & ([a], f\ a) \\ h'(x \mathbin{+\!\!+} y) & = & h'\,x \oplus h'\,y \end{array}$$

where $\oplus$ is defined by

$$(x, u) \oplus (y, v) = (x +\!\!+\ y, H(x, y, u, v))$$

If h' is a well-defined function, then so is h. We have

$$h = \pi_2 \cdot h'$$

where $\pi_2(a, b) = b$.

In order to determine the conditions under which the above equations determine h', let β be the smallest set of values such that

1. $([\,], e)$ is in β;
2. $([a], f\ a)$ is in β for all a in α;
3. $(x, u) \oplus (y, v)$ is in β whenever (x, u) and (y, v) are.

Now, by our basic assumption, h' is uniquely determined if $(\beta, \oplus, ([\,], e))$ is a monoid. Translating the monoid conditions (associativity, and existence of an identity) into conditions on e and H, we must therefore have:

1. $H(x, [\,], u, e) = u$
2. $H([\,], x, e, u) = u$
3. $H(x +\!\!+\ y, z, H(x, y, u, v), w) = H(x, y +\!\!+\ z, u, H(y, z, v, w))$

These three conditions (the *consistency conditions*) determine the properties that H and e must satisfy in order for the equations for h to determine h completely.

Let us consider one example. Take $e = [\,]$ and

$$H(x, y, u, v) = (u = x \rightarrow u +\!\!+\ v, u)$$

Here we use the McCarthy conditional form on the right. We leave the verification of the first two conditions on e and H to the reader. For the third condition, the expression $H(x +\!\!+\ y, z, H(x, y, u, v), w)$ reduces to

$$\begin{aligned}
&(u = x \wedge v = y \rightarrow u +\!\!+\ v +\!\!+\ w,\\
&u = x \wedge v \neq y \rightarrow u +\!\!+\ v,\\
&u \neq x \wedge u = x +\!\!+\ y \rightarrow u +\!\!+\ w,\\
&u \neq x \wedge u \neq x +\!\!+\ y \rightarrow u)
\end{aligned}$$

On the other hand, the expression $H(x, y +\!\!+ z, u, H(y, z, v, w))$ reduces to

$$\begin{array}{l}(u = x \wedge y = y \rightarrow u +\!\!+ v +\!\!+ w, \\ u = x \wedge v \neq y \rightarrow u +\!\!+ v, \\ u \neq x \rightarrow u)\end{array}$$

These two expressions are not equal unless we have that

$$u = x \vee u \neq x +\!\!+ y$$

for all (x, u) in β. This condition is equivalent to $\#x \geqslant \#u$ for all (x, u) in β and is satisfied if

$$\#f\ a \leqslant 1$$

for all a. In particular, if we take

$$f = (p \rightarrow [\cdot], []^{\circ})$$

for an arbitrary p, then everything is all right. With this definition of f, the value of $h\ x$ is just the longest initial segment of x all of whose elements satisfy p. In symbols:

$$h = \uparrow_{\#}/ \cdot \mathit{all}\, p\triangleleft \cdot \mathit{inits}$$

As a last point, we show that h is not a homomorphism. For concreteness take $p = \mathit{even}$, the predicate that determines whether a number is even. Suppose

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

for some operator $\oplus$. Since $h[2, 1] = 2$,$h[4] = [4]$, and $h[2] = [2]$, we have

$$\begin{array}{rcl}h[2,1,4] & = & h[2,1] \oplus h[4] \\ & = & [2] \oplus [4] \\ & = & h[2] \oplus h[4] \\ & = & h[2,4]\end{array}$$

This is a contradiction, since $h[2, 1, 4] = [2]$ and $h[2, 4] = [2, 4]$.

3.2 Left reductions

We defined the directed reductions in the first lecture, but the pattern of the equations does not follow those laid down in the previous section. We really need to give an alternative definition in order to justify that, for example, $\oplus\!\not\!\rightarrow_e$ is a well-defined function.

In the monoid view of lists, the formal definition of $\oplus\not\to_e$ is

$$\begin{array}{lcl} \oplus \not\to_e [\,] & = & e \\ \oplus \not\to_e [a] & = & e \oplus a \\ \oplus \not\to_e (x \mathbin{+\!\!+} y) & = & \oplus \not\to_{e'} y \quad \text{where } e' = \oplus \not\to_e x \end{array}$$

Equivalently, setting $f\ e = \oplus\not\to_e$, we have

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

We leave to the reader to check that the above equations satisfy the consistency conditions of the previous section.

There is an instructive alternative way of seeing that $\oplus\not\to_e$ is well-defined. Define h by

$$\begin{array}{lcl} h\,[\,] & = & id \\ h\,[a] & = & (\oplus a) \\ h(x \mathbin{+\!\!+} y) & = & h\,y \cdot h\,x \end{array}$$

For $\oplus : \beta \times \alpha \to \beta$ we have that h is a homomorphism

$$h : ([\alpha], \mathbin{+\!\!+}, [\,]) \to (\beta \to \beta, \cdot, id_\beta)$$

Now we have

$$\oplus \not\to_e x = h\,x\,e$$

and so $\oplus\not\to_e$ is a well-defined function.

The above reasoning justifies the well-definedness of $\not\to_e$ but does not explain why the construct is important. The basic reason why left reductions are important (and similar remarks apply to right reductions) is as follows.

Consider a set of equations of the form

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

We claim that

$$f = \pi_2 \cdot \oplus\not\to_{e'}$$

where

$$\begin{array}{lcl} e' & = & ([\,], e) \\ (x, u) \oplus a & = & (x \mathbin{+\!\!+} [a], F(a, x, u)) \end{array}$$

In brief, every set of equations of the above form can be expressed in terms of a left reduction. Conversely, we have

$$\begin{array}{lcl} \oplus \not\to_e [\,] & = & e \\ \oplus \not\to_e (x \mathbin{+\!\!+} [a]) & = & (\oplus \not\to_e x) \oplus a \end{array}$$

so an arbitrary left reduction can be expressed in the same way.

The above discussion can be summarised in terms of the different ways we can view lists. The monoid view of lists is to say that every list is either (i) the empty list; (ii) a singleton list; or (iii) the concatenation of two (non-empty) lists. The primary mechanism for defining functions with this view is the homomorphism. Another view of lists is that every list is either (i) the empty list; or (ii) of the form $x \mathbin{+\!\!+} [a]$ for some list x and value a. The primary mechanism in this case is the left reduction.

Yet a third view of lists is that every list is either (i) the empty list; or (ii) of the form $[a] \mathbin{+\!\!+} x$ for some a and list x. The primary mechanism here is the right reduction. In the majority of functional programming languages, it is this third view that prevails. One of the reasons concerns the possibility of defining functions on *infinite* lists, a reason that we will not go into here. Fortnnately, we can define left reductions with this view as well. We have

$$\begin{array}{lcl} \oplus \not\to_e [\,] & = & e \\ \oplus \not\to_e ([a] \mathbin{+\!\!+} x) & = & \oplus \not\to_{e'} x \quad \text{where } e' = e \oplus a \end{array}$$

We leave the verification of this fact to the reader.

3.3 Loops

In the functional approach to program derivation, the final product of a calculation is an expression denoting a mathematical function. This expression still has to be translated into a specific programming language in order for it to be executable by computer. One obvious candidate is a functional programming language, such as ML or Miranda[1]. However, there is no reason why the final expression should not be translated into a conventional imperative language. For example, a left reduction can easily be translated into a loop. Using hopefully straightforward notation, the value $\oplus \not\to_e x$ is the result delivered by the following imperative program:

```
|[ var a; a := e;
```

[1]Miranda is a trademark of Rcsearch Software Ltd.

```
        for b in x
        do a := a oplus b;
        return a ]|
```

Here, the 'generator' `b in x` successively assigns to b the elements of x in order from left to right.

3.4 Left-zeros

Both the imperative and functional implementations of left reductions require that the argument list be traversed in its entirety. Such a traversal can be cut short if we recognize the possibility that an operator may have *left-zeros*. By definition, ω is a left-zero of $\oplus$ if

$$\omega \oplus a = \omega$$

for all a. An operator may have none, one or many different left-zeros. If ω is a left-zero of $\oplus$, then

$$\oplus \not\rightarrow_{\omega} x = \omega$$

for all x. Since

$$\oplus \not\rightarrow_{e} (x +\!\!+ y) \;=\; \oplus \not\rightarrow_{e'} y \quad \text{where } e' = \oplus \not\rightarrow_{e} x$$

it follows that

$$\oplus \not\rightarrow_{e} (x +\!\!+ y) = \oplus \not\rightarrow_{e} x$$

whenever the right-hand side is a left-zero of $\oplus$. In words, evaluation of a left-reduction can be terminated on encountering a left-zero.
Suppose $lzero_{\oplus}$ is a predicate that determines whether its argument is a left-zero of $\oplus$. Using hopefully equally straightforward notation as before, we then have that $\oplus \not\rightarrow_{e} x$ can be evaluated by the program

```
    |[ var a; a := e;
       for b in x while not lzero(a)
       do a := a oplus b;
       return a ]|
```

Before seeing an application of this idea we need a simple yet powerful result.

Lemma 3 (Specialisation) *Every homomorphism on lists can be expressed as a left (or also a right) reduction. More precisely,*

$$\oplus/ \cdot f* = \odot \not\to_e$$

where $e = id_\oplus$ *and*

$$a \odot b = a \oplus f\ b$$

We omit the simple proof.

3.5 Minimax

Let us return to the problem of computing

$$minimax = \downarrow/ \cdot \uparrow/*$$

efficiently. Using the specialisation lemma, we can write

$$minimax = \odot \not\to_\infty$$

where ∞ is the identity element of $\downarrow/$, and

$$a \odot x = a \downarrow (\uparrow/x)$$

Since $\downarrow$ distributes through $\uparrow$ we have

$$a \odot x = \uparrow/(a\downarrow) * x$$

Using the specialisation lemma a second time, we have

$$a \odot x = \oplus_a \not\to_{-\infty} x$$

where $-\infty$ is the identity element of $\uparrow$ and

$$b \oplus_a c = b \uparrow (a \downarrow c)$$

Now, a is a left-zero of $\oplus_a$ (and so, by the way, is ∞), and $-\infty$ is a left-zero of $\odot$. This means we can implement ($minimax\ xs$), where xs is a list of lists, by the loop

```
|[ var a; a:= infinity;
   for x in xs while a <> -infinity
   do a := a odot x;
   return a ]|
```

where the assignment `a := a odot x` can be implemented by the loop

```
|[ var b; b:= -infinity;
   for c in x while b <> a
   do b := b max (a min c);
   a := b ]|
```

3.6 The alpha-beta algorithm

We now generalise the minimax problem to trees. Consider the data-type

$$tree ::= Tip\ num | Fork\ [tree]$$

The syntax of this declaration is that of the functional language Miranda, and it is also employed in the notation of Bird and Wadler (reference [2] of Lecture 1). It says that $(Tip\ n)$ is a tree for each number n, and that $(Fork\ ts)$ is a tree whenever ts is a sequence of trees. The primitive functions *Tip* and *Fork* are called the *constructors* of the type *tree*.

We wish to calculate an efficient algorithm for computing a function $eval : tree \rightarrow num$, where

$$\begin{array}{lcl} eval(Tip\ n) & = & n \\ eval(Fork\ ts) & = & \uparrow/(-eval) * ts \end{array}$$

Here we use the notation $-f$ for the function defined by $(-f)a = -(f\ a)$.

Using the specialisation lemma on the right-hand side of the second equation for *eval*, we obtain

$$eval(Fork\ ts) = \oplus \not\rightarrow_{-\infty}\ ts$$

where

$$a \oplus t = a \uparrow (-eval\ t)$$

We now expand this last equation by considering the two possible forms for a tree t:

$$\begin{array}{lcl} a \oplus (Tip\ n) & = & a \uparrow (-n) \\ a \oplus (Fork\ ts) & = & a \uparrow (-(\uparrow/(-eval) * ts)) \end{array}$$

The last equation can now be simplified using the laws

$$\begin{array}{lcl} -(a \uparrow b) & = & (-a) \downarrow (-b) \\ a \uparrow (b \downarrow c) & = & (a \uparrow b) \downarrow (a \uparrow c) \end{array}$$

We obtain

$$a \oplus (\mathit{Fork}\ ts) = \downarrow/(a\uparrow) * \mathit{eval} * ts$$

After using the $*$ distributivity law, the right-hand side of this equation is also a candidate for specialisation. We have

$$a \oplus (\mathit{Fork}\ ts) = \otimes_a \not\to_\infty ts$$

where

$$b \otimes_a t = b \downarrow (a \uparrow \mathit{eval}\ t)$$

Furthermore, since

$$\begin{aligned} \mathit{eval}\ t &= \infty \downarrow (-\infty \uparrow \mathit{eval}\ t) \\ &= \infty \otimes_{-\infty} t \end{aligned}$$

we have —without inventiveness — reduced the problem of calculating *eval t* to that of evaluating $b \otimes_a t$ for values of a and b.

Let us now expand the definition of $b \otimes_a t$ in a similar way as we did for $a \oplus t$. We obtain

$$\begin{aligned} b \otimes_a (\mathit{Tip}\ n) &= b \downarrow (a \uparrow n) \\ b \otimes_a (\mathit{Fork}\ ts) &= b \downarrow (a \uparrow (\uparrow/(-\mathit{eval}) * ts)) \end{aligned}$$

In oder to simplify the right-hand side of this last equation, we need the dual distributive law

$$b \downarrow (a \uparrow c) = (b \downarrow a) \uparrow (b \downarrow c)$$

and the fact that evaluation of $b \otimes_a t$ is required only for values of a and b satisfying $a = a \downarrow b$; in other words, for $a \leqslant b$. In such a case we have

$$b \downarrow (a \uparrow c) = a \uparrow (b \downarrow c)$$

by commutativity of $\downarrow$.

We then obtain

$$b \otimes_a (\mathit{Fork}\ ts) = a \uparrow (\uparrow/(b\downarrow) * (-\mathit{eval}) * ts)$$

Using specialisation yet a third time, we obtain

$$b \otimes_a (\mathit{Fork}\ ts) = \oplus_b \not\to_a ts$$

where

$$\alpha \oplus_b t = \alpha \uparrow (b \downarrow (-\mathit{eval}\, t))$$

At this point, the seemingly endless succession of expansion and specialisation steps can be stopped. A short calculation using the given properties of $-$, $\uparrow$ and $\downarrow$ yields

$$\alpha \oplus_\beta t = -(-\alpha) \otimes_{(-\beta)} t$$

Introducing

$$\mathit{bval}\, \alpha\, \beta\, t = \beta \otimes_\alpha t$$

and putting the resulting equations together, we obtain

$$\begin{array}{lcl}
\mathit{eval}\, t & = & \mathit{bval}\, (-\infty)\, \infty\, t \\
\mathit{bval}\, \alpha\, \beta\, (\mathit{Tip}\, n) & = & \beta \downarrow (\alpha \uparrow n) \\
\mathit{bval}\, \alpha\, \beta\, (\mathit{Fork}\, ts) & = & \oplus_\beta \not\to_\alpha ts \\
\alpha' \oplus_\beta t & = & -\mathit{bval}(-\beta)(-\alpha')t
\end{array}$$

Finally, we bring on the left-zeros. We only need to observe that β is a left-zero of $\oplus_\beta$. This follows from the definition of $\oplus_\beta$ and the absorbtive law

$$\beta \uparrow (\beta \downarrow \gamma) = \beta$$

Incorporating this optimisation yields the alpha-beta algorithm.

The various axioms concerning $(\uparrow, \downarrow, -, \infty, -\infty)$ used in the above derivation are precisely those of a Boolean algebra.

3.7 References

An alternative, and arguably less satisfactory, treatment of the alpha-beta algorithm occurs in

[1] Bird, R.S. and Hughes, R.J.M. The alpha-beta algorithm: an exercise in program transformation. *Information Processing Letters*, 24 (1987) 53-57.

The importance of the purely algebraic notion of left-zeros and its consequences for optimisation was discussed in

[2] Meertens, L. First steps towards the theory of Rose trees. *(Unpublished draft)*, CWI, Amsterdam, 1988.

This paper also contains another treatment of the alpha-beta algorithm.

4 Arrays

4.0 Problem

Given is an array x with elements in the set $\{0,1\}$. Required is an efficient algorithm for computing the area of the largest rectangle (i.e., contiguous subarray) of x, all of whose elements are 1.

4.1 Binoids

Suppose α is a set of values closed under two partial operations $+$ and $\times$ such that:

(i) $+$ and $\times$ are associative, in the sense that each of the equations

$$\begin{array}{rcl}(a+b)+c & = & a+(b+c)\\ (a\times b)\times c & = & a\times(b\times c)\end{array}$$

hold whenever both sides of the equation are defined;

(ii) $+$ and $\times$ satisfies the further equation

$$(a+b)\times(c+d)=(a\times c)+(b\times d)$$

whenever both sides are defined. (We shall refer to this property by saying that $+$ *abides* with $\times$. The reason for this choice of name will appear shortly.)

There is no standard terminology for such an algebraic structure so, for the sake of a name, we shall call it a *binoid*. Here are four examples of binoids.

Example 1. Let $\oplus:\alpha\times\alpha\to\alpha$ be associative and commutative. Then $\oplus$ abides with itself, and so $(\alpha,\oplus,\oplus)$ is a binoid.

Example 2. Recall that the operator $\ll$ is defined by $a\ll b=a$. Note that $\ll$ is associative but not commutative. Nevertheless, the structure $(\alpha,\ll,\ll)$ is a binoid. Since both sides of

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

reduce to a, we have that $\ll$ abides with itself. Similarly, $(\alpha,\gg,\gg)$ is a binoid, where $a\gg b=b$.

Example 3. The structure $(\alpha, \gg, \ll)$ is a binoid. We have that both sides of

$$(a \gg b) \ll (c \gg d) = (a \ll c) \gg (b \ll d)$$

reduce to b.

Example 4. Define the partial operator $\bullet$ by the equation

$$a \bullet b = a \quad \text{provided} \;\; a = b$$

The operator $\bullet$ is associative because

$$(a \bullet b) \bullet c = a \bullet (b \bullet c)$$

whenever both sides are defined, that is, when all three values are the same.

Let $\oplus : \alpha \times \alpha \rightarrow \alpha$ be some associative operator. We claim that $(\alpha, \oplus, \bullet)$ is a binoid. We have

$$(a \oplus b) \bullet (c \oplus d) = (a \bullet c) \oplus (b \bullet d)$$

whenever both sides are defined. The right-hand side is defined just in the case that $a = c$ and $b = d$. Its value is then $a \oplus b$. The left-hand side is defined just in the case that $a \oplus b = c \oplus d$ and its value is then $a \oplus b$. Notice that the left-hand side can be defined without the right-hand side being defined.

4.2 Arrays

The type of arrays with elements from α will be denoted by $|\alpha|$. This type is specified by a free algebra generated from elements of α under the assignment $|\cdot| : \alpha \rightarrow |\alpha|$ which maps elements of α to singleton arrays. The constituents of this algebra are:

(1) Two operators $\ominus$ (pronounced 'above') and ϕ (pronounced 'beside') such that $(|\alpha|, \ominus, \phi)$ forms a binoid. The abide property

$$(x \ominus y) \phi (u \ominus v) = (x \phi u) \ominus (c \phi d)$$

can be pictured as

$$\left(\frac{x}{y}\,\middle|\,\frac{u}{v}\right) = \left(\frac{x|u}{y|v}\right)$$

(The name a*bide* is an abbreviation of 'above-beside'.)

(2) Two functions *height* and *width* with types

$$height, width : |\alpha| \rightarrow N^{+}$$

where N^{+} denotes the positive integers. Informally, *height* returns the number of rows in the array, and *width* the number of columns. These functions are are such that $x \phi y$ is defined just in the case that $height\, x = height\, y$, and $x \ominus y$ is defined just in the case that $width\, x = width\, y$. Moreover, we have

$$\begin{aligned}
height\,|a| &= 1 \\
height(x \ominus y) &= height\, x + height\, y \\
height(x \phi y) &= height\, x \bullet height\, y \\
\\
width\,|a| &= 1 \\
width(x \ominus y) &= width\, x \bullet width\, y \\
width(x \phi y) &= width\, x + width\, y
\end{aligned}$$

where $\bullet$ is the operator described in Example 4 above.

In displayed examples we shall use round brackets to indicate arrays. For example, the array

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

has height and width 3. It is the array described by the formula

$$(|1| \phi |2| \phi |3|) \ominus (|4| \phi |5| \phi |6|) \ominus (|7| \phi |8| \phi |9|)$$

as well as many others.

By definition, a *row vector* is an array of unit height, and a *column vector* is an array of unit width.

Note, finally, that we choose not to define the concept of an empty array.

4.3 Map

We shall use the same symbol $*$ for mapping over arrays as for mapping over lists. Suppose $f : \alpha \rightarrow \beta$. Then $f* : |\alpha| \rightarrow |\beta|$ is the array homomorphism defined by the three equations:

$$\begin{aligned} f * |a| &= |f\, a| \\ f * (x \ominus y) &= (f * x) \ominus (f * y) \\ f * (x \phi y) &= (f * x) \phi (f * y) \end{aligned}$$

Note that

$$\begin{aligned} \mathit{height}(f * x) &= \mathit{height}\, x \\ \mathit{width}(f * x) &= \mathit{width}\, x \end{aligned}$$

The $*$ distributive law

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

is valid for arrays as well as lists.

4.4 Reduce

Given two operators $\oplus, \otimes : \alpha \times \alpha \to \alpha$, we can define a reduction operator $(\oplus, \otimes)/$ for arrays by three equations:

$$\begin{aligned} (\oplus,\otimes)/|a| &= a \\ (\oplus,\otimes)/(x \ominus y) &= ((\oplus,\otimes)/x) \oplus ((\oplus,\otimes)/y) \\ (\oplus,\otimes)/(x \phi y) &= ((\oplus,\otimes)/x) \otimes ((\oplus,\otimes)/y) \end{aligned}$$

For these equations to be consistent we require that $(\alpha, \oplus, \otimes)$ forms a binoid. Note that the operator $\oplus$, which replaces the 'above' operator $\ominus$, comes first in the reduction, and the operator $\otimes$, which replaces the 'beside' operator ϕ, comes second.

For example, $(+,+)/$ sums the elements in an array of numbers, and $(\vee,\vee)/$ determines whether there exists a true entry in an array of booleans. Both $+$ and $\vee$ are commutative and associative operators, so the binoid condition is satisfied.

As further examples of array reductions, we have

$$\begin{aligned} \mathit{height} &= (+,\bullet)/ \cdot K_1* \\ \mathit{width} &= (\bullet,+)/ \cdot K_1* \\ \mathit{area} &= (+,+)/ \cdot K_1* \end{aligned}$$

where $\bullet$ was defined in Example 4 above. We also have

$$\mathit{area}\, x = \mathit{height}\, x \times \mathit{width}\, x$$

The top-left element of an array is given by

$$topleft = (\ll, \ll)/$$

Similarly, we can define reductions that return each of the other corner elements. Examples 2 and 3 above show that the binoid condition is satisfied.

The identity function on arrays is given by

$$id = (\ominus, \phi)/ \cdot |\cdot|*$$

The companion function

$$tr = (\phi, \ominus)/ \cdot |\cdot|*$$

defines the operation of array transposition. A simple but important fact is that *tr* is its own inverse, that is,

$$tr \cdot tr = id$$

We shall prove this fact below.

Finally, notice that when reducing over a row or column vector one of the operators in a reduction is redundant. Accordingly, we shall write $\oplus/$ for reduction over vectors. The notation is the same as for reducing over lists, but the ambiguity is not harmful since row and column vectors can be regarded as lists.

4.5 Promotion

The one-point and join (or promotion) rules for lists have counterparts in the theory of arrays. (There is no analogue of the empty rule because we have not defined the concept of an empty array.) These rules are:

one-point rules

$$\begin{aligned} f* \cdot |\cdot| &= |\cdot| \cdot f \\ (\oplus, \otimes)/ \cdot |\cdot| &= id \end{aligned}$$

join rules

$$\begin{aligned} f* \cdot (\ominus, \phi)/ &= (\ominus, \phi)/ \cdot f** \\ (\oplus, \otimes)/ \cdot (\ominus, \phi)/ &= (\oplus, \otimes)/ \cdot (\oplus, \otimes)/* \end{aligned}$$

We also have two further join rules, called the *transpose* rules:

transpose rules

$$\begin{array}{lcl} f* \cdot (\phi,\ominus)/ & = & (\phi,\ominus)/ \cdot f** \\ (\oplus,\otimes)/ \cdot (\phi,\ominus)/ & = & (\otimes,\oplus)/ \cdot (\oplus,\otimes)/* \end{array}$$

Notice in the last rule that the order of the operators in the outer reduction is reversed. These six rules can be proved by induction, using the definitions of $*$ and $/$ over arrays.

To illustrate these rules, we calculate:

$$\begin{array}{lcl} tr \cdot tr & = & \text{definition of } tr \\ & & (\phi,\ominus)/ \cdot |\cdot|* \cdot (\phi,\ominus)/ \cdot |\cdot|* \\ & = & \text{map transpose rule} \\ & & (\phi,\ominus)/ \cdot (\phi,\ominus)/ \cdot |\cdot|** \cdot |\cdot|* \\ & = & \text{reduce transpose rule} \\ & & (\ominus,\phi)/ \cdot (\phi,\ominus)/* \cdot |\cdot|** \cdot |\cdot|* \\ & = & * \text{ distributivity} \\ & & (\ominus,\phi)/ \cdot ((\phi,\ominus)/ \cdot |\cdot|* \cdot |\cdot|)* \\ & = & \text{map one-point rule} \\ & & (\ominus,\phi)/ \cdot ((\phi,\ominus)/ \cdot |\cdot| \cdot |\cdot|)* \\ & = & \text{reduce one-point rule} \\ & & (\ominus,\phi)/ \cdot |\cdot|* \\ & = & \text{definition of } id \\ & & id \end{array}$$

4.6 Zip

Let us return for the moment to lists. For each binary operator $\oplus : \alpha \times \beta \rightarrow \gamma$ we define a partial operator $\curlyvee_\oplus$ (pronounced 'zip with $\oplus$') informally by the equation

$$[a_1, a_2, \ldots, a_n] \curlyvee_\oplus [b_1, b_2, \ldots, b_n] = [a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n]$$

Thus, $x \curlyvee_\oplus y$ is defined only in the case that $\#x = \#y$. The type of $\curlyvee_\oplus$ is given by

$$\curlyvee_\oplus : [\alpha] \times [\beta] \rightarrow [\gamma]$$

Among the many properties that $\curlyvee_\oplus$ enjoys is the fact that $\curlyvee_\oplus$ is associative if $\oplus$ is; similarly, for commutativity and idempotence.

Formally, we can specify $\curlyvee_\oplus$ by three equations:

$$\begin{array}{lcl} [\,] \curlyvee_\oplus [\,] & = & [\,] \\ [a] \curlyvee_\oplus [b] & = & [a \oplus b] \\ (x +\!\!+ y) \curlyvee_\oplus (u +\!\!+ v) & = & (x \curlyvee_\oplus u) +\!\!+ (y \curlyvee_\oplus v) \end{array}$$

The third equation is asserted only under the conditions that $\#x = \#u$ and $\#y = \#v$. It is, of course, just the condition that $\Upsilon_{\oplus}$ abides with $+\!\!\!+$.

The same operator can be defined on arrays (and, indeed, on many other data structures). We have

$$\begin{array}{lcl} |a|\,\Upsilon_{\oplus}\,|b| & = & |a \oplus b| \\ (x \ominus y)\,\Upsilon_{\oplus}\,(u \ominus v) & = & (x\,\Upsilon_{\oplus}\,u) \ominus (y\,\Upsilon_{\oplus}\,v) \\ (x \phi y)\,\Upsilon_{\oplus}\,(u \phi v) & = & (x\,\Upsilon_{\oplus}\,u) \phi (y\,\Upsilon_{\oplus}\,v) \end{array}$$

The last two equations require that x is the same shape as u, and y is the same shape as v. So, $\ominus$ abides with $\Upsilon_{\oplus}$, and so does ϕ.

Some useful examples of this operator are as follows. First, the function

$$rows = (\ominus, \Upsilon_{\phi})/ \cdot ||\cdot||*$$

converts an array into a column vector whose entries are row vectors, one for each row of the array. For example:

$$rows \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \\ \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \\ \begin{pmatrix} 7 & 8 & 9 \end{pmatrix} \end{pmatrix}$$

Similarly, the function

$$cols = (\Upsilon_{\ominus}, \phi)/ \cdot ||\cdot||*$$

converts an array into a row vector, each entry being a column vector.

The related functions

$$\begin{array}{lcl} listrows & = & (+\!\!\!+, \Upsilon_{+\!\!\!+})/ \cdot [[\cdot]]* \\ listcols & = & (\Upsilon_{+\!\!\!+}, +\!\!\!+)/ \cdot [[\cdot]]* \end{array}$$

each convert an array into a list of lists. The function *listrows* turns an array into a list of rows, each row being a list of entries from a row of the array. The function *listcols* turns the array into a list of columns. We have

$$\begin{array}{lcl} height & = & \# \cdot listrows \\ width & = & \# \cdot listcols \end{array}$$

Other pairs of identities one can establish are the following:

$$\begin{array}{lcl} listcols & = & listrows \cdot tr \\ listrows & = & listcols \cdot tr \\ \\ (\oplus, \otimes)/ & = & \oplus/ \cdot \otimes/* \cdot listrows \\ (\oplus, \otimes)/ & = & \otimes/ \cdot \oplus/* \cdot listcols \end{array}$$

4.7 Directed reductions

There are a number of different issues involved in setting up the idea of directed reductions on arrays. First of all, there is the question of how many operators should take part in a reduction: one or two. Second, there is the question of how many directions one might wish to identify as being useful.

We shall adopt the following solutions to these questions. Each directed reduction will involve a single operator, and we focus on two legitimate directions: top-down (*top* reductions), and left-right (*left* reductions). We shall use the notation $\oplus\!\not\downarrow$ for a top-reduction, and $\oplus\!\nrightarrow$ for a left reduction. (Since top reductions work downwards, we use a *downward* arrow, and since left reductions work rightwards we use a *rightward* arrow.)

The effect of a top reduction is illustrated by

$$\oplus\!\not\downarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} (1 \oplus 4) \oplus 7 & (2 \oplus 5) \oplus 8 & (3 \oplus 6) \oplus 9 \end{pmatrix}$$

Thus $\oplus \not\downarrow x$ produces a single row as its result. Similarly,

$$\oplus \nrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} (1 \oplus 2) \oplus 3 \\ (4 \oplus 5) \oplus 6 \\ (7 \oplus 8) \oplus 9 \end{pmatrix}$$

so $\oplus \nrightarrow x$ produces a single column.

Two basic examples of the directed rednctions are provided by the following alternative definitions of *rows* and *cols*:

$$\begin{array}{lcl} \mathit{rows} & = & (\phi\!\nrightarrow) \cdot |\cdot|* \\ \mathit{cols} & = & (\ominus\!\not\downarrow) \cdot |\cdot|* \end{array}$$

One consequence of the given forms of directed reductions is that if we compose

$$(\oplus\!\not\downarrow) \cdot (\otimes\!\nrightarrow)$$

in sequence, then the result is a singleton array. To extract this value, suppose we define $\mathit{the} : |\alpha| \to \alpha$ by

$$\mathit{the}\,|a| = a$$

Now we can state the fact that every general reduction can be expressed as the composition of two directed reductions:

$$\begin{aligned}(\oplus,\otimes)/ &= \mathit{the}\cdot(\oplus\dagger)\cdot(\otimes\nrightarrow)\\(\oplus,\otimes)/ &= \mathit{the}\cdot(\otimes\nrightarrow)\cdot(\oplus\dagger)\end{aligned}$$

We have not yet given the formal definitions of these two new reduction operators. We give the equations for a top reduction:

$$\begin{aligned}(\oplus\dagger)|a| &= |a|\\(\oplus\dagger)(x\ominus|a|) &= |b\oplus a| &&\text{where } b=\mathit{the}(\oplus\dagger x)\\(\oplus\dagger)(x\,\phi\,y) &= (\oplus\dagger x)\,\phi\,(\oplus\dagger y)\end{aligned}$$

4.8 Accumulations

Reasonably enough, corresponding to the top and left reduction operators are the top and left accumulation operators. These are denoted, respectively, by $\oplus\ddagger$ and $\oplus\not\!\nrightarrow$. Their effect is illustrated by the following examples:

$$(\oplus\ddagger)\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}=\begin{pmatrix}1&2&3\\1\oplus4&2\oplus5&3\oplus6\\(1\oplus4)\oplus7&(2\oplus5)\oplus8&(3\oplus6)\oplus9\end{pmatrix}$$

$$(\oplus\not\!\nrightarrow)\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}=\begin{pmatrix}1&1\oplus2&(1\oplus2)\oplus3\\4&4\oplus5&(4\oplus5)\oplus6\\7&7\oplus8&(7\oplus8)\oplus9\end{pmatrix}$$

Thus, if $\oplus:\alpha\times\alpha\rightarrow\alpha$, then

$$\oplus\ddagger,\oplus\not\!\nrightarrow:|\alpha|\rightarrow|\alpha|$$

Left accumulations on arrays are related to left accumulations on lists by the equation

$$\mathit{listrows}\cdot(\oplus\not\!\nrightarrow)=(\oplus\not\!\!/\!\!\!\rightarrow)*\cdot\mathit{listrows}$$

Similarly,

$$\mathit{listcols}\cdot(\oplus\ddagger)=(\oplus\not\!\!/\!\!\!\rightarrow)*\cdot\mathit{listcols}$$

The formal definition of $\oplus\ddagger$ is given by the equations

$$\begin{aligned}\oplus\ddagger|a| &= |a|\\\oplus\ddagger(x\ominus|a|) &= (\oplus\ddagger x)\ominus|b\oplus a| &&\text{where } b=\mathit{bottomleft}(\oplus\ddagger x)\\\oplus\ddagger(x\,\phi\,y) &= (\oplus\ddagger x)\,\phi\,(\oplus\ddagger y)\end{aligned}$$

The function *bottomleft* (which returns the bottom-left corner of an array) is defined by

$$\mathit{bottomleft} = (\gg, \ll)/$$

In order to relate array accumulations to array reductions, in the way that list accumulations were related to list reductions in the first lecture, we need to consider the array analogues of *inits* and *tails*. This we do next.

4.9 Tops and bottoms

There are four reasonable ways of dissecting an array: we shall call them *tops*, *bottoms*, *lefts* and *rights*. Two of them are illustrated by the following examples:

$$\mathit{lefts}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix} = \left(\begin{pmatrix}1\\4\\7\end{pmatrix}\begin{pmatrix}1&2\\4&5\\7&8\end{pmatrix}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}\right)$$

$$\mathit{tops}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix} = \left(\begin{matrix}\begin{pmatrix}1&2&3\end{pmatrix}\\\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}\\\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}\end{matrix}\right)$$

The essential points here are that *lefts* and *rights* each return a single row, while *tops* and *bottoms* each return a single column.

We can give the definition of *lefts* and *tops* in terms of accumulations:

$$\begin{array}{lcl}\mathit{lefts} & = & (\phi\!\not\Vert)\cdot \mathit{cols}\\ \mathit{tops} & = & (\ominus\ddagger)\cdot \mathit{rows}\end{array}$$

These equations should be compared to the definition

$$\mathit{inits}^{+} = (+\!\!\!+\!\not/\!\!/)\cdot[\cdot]*$$

of Lecture 1.

As one might expect, there is a multitude of relationships between the various forms of array reductions, array accumulations, and the four dissection functions introduced above. In particular, we have the

orthogonal reduction rules

$$\begin{array}{rcl}(\oplus⤈)* \cdot \mathit{lefts} & = & \mathit{lefts} \cdot (\oplus⤈) \\ (\oplus⇸)* \cdot \mathit{tops} & = & \mathit{tops} \cdot (\oplus⇸)\end{array}$$

— plus a further two equations obtained by replacing *lefts* by *rights* and *tops* by *bottoms*.

We also have another group of rules:

orthogonal accumulation rules

$$\begin{array}{rcl}(\oplus⇟)* \cdot \mathit{lefts} & = & \mathit{lefts} \cdot (\oplus⇟) \\ (\oplus⇻)* \cdot \mathit{tops} & = & \mathit{tops} \cdot (\oplus⇻)\end{array}$$

— plus a further two rules obtained in the same way as before.

Finally, we have the analogues of the accumulation lemma of Lecture 1.

Accumulation lemma

$$\begin{array}{rcl}(\oplus⤈)* \cdot \mathit{tops} & = & \mathit{rows} \cdot (\oplus⇟) \\ (\oplus⇸)* \cdot \mathit{lefts} & = & \mathit{cols} \cdot (\oplus⇻)\end{array}$$

4.10 Horner's rule

The alert reader might at this point be wondering if there is an analogue of Horner's rule that works for arrays. Since *tails* corresponds to *bottoms* (and also to *rights*), the expression to be simplified in the array version of Horner's rule is

$$(\oplus \cdot \oplus)/ \cdot (\otimes, \odot)/* \cdot \mathit{bottoms}$$

It turns out that the conditions we need are: (i) that $\otimes$ distriubtes (backwards) through $\oplus$; and (ii) $\oplus$ abides with $\odot$. The first condition is the same as in the case of Horner's rule for lists, but the second condition is new. If these two conditions are met, then

$$(\oplus, \oplus)/ \cdot (\otimes, \odot)/* \cdot \mathit{bottoms} = (\odot, \odot)/ \cdot \circledast⤈$$

where $\circledast$ is defined (as in the version of Horner's rule for non-empty lists) by

$$a \circledast b = (a \otimes b) \oplus b$$

We shall illustrate Horner's rule rather than give a formal proof. Consider the array

$$x = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

The left-hand side of Horner's rule applied to x gives

$$((1 \otimes 3 \otimes 5) \odot (2 \otimes 4 \otimes 6)) \oplus ((3 \otimes 5) \odot (4 \otimes 6)) \oplus (5 \odot 6)$$

Using the fact that $\oplus$ abides with $\odot$, we can write this in the form

$$((1 \otimes 3 \otimes 5) \oplus (3 \otimes 5) \oplus 5) \odot ((2 \otimes 4 \otimes 6) \oplus (4 \otimes 6) \oplus 6)$$

Now, since $\otimes$ distributes over $\oplus$, we can write the above expression in the form

$$((((1 \otimes 3) \oplus 3) \otimes 5) \oplus 5) \odot ((((2 \otimes 4) \oplus 4) \otimes 6) \oplus 6)$$

Using the definition of $\circledast$, this simplifies to

$$((1 \circledast 3) \circledast 5) \odot ((2 \circledast 4) \circledast 6)$$

and thus to

$$(\odot, \odot)/(\circledast \mathbin{\downarrow\!\!\!\!-} x)$$

where x is the original array. We shall see an application of this form of Horner's rule in the last section.

4.11 Rectangles

By definition, a *rectangle* of an array x is a contiguous subarray of x. Thus, rectangles are to arrays as segments are to lists. In this section we put the four dissection functions together in order to define the rectangles of an array.

First of all, we give some arrays of arrays that will prove useful. The *top-lefts* (*topls*) of an array is defined by

$$\mathit{topls} = (\ominus, \phi)/ \cdot \mathit{tops}* \cdot \mathit{lefts}$$

For example:

$$topls \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \left(\begin{array}{ccc} \begin{Bmatrix} 1 \end{Bmatrix} & \begin{Bmatrix} 1 & 2 \end{Bmatrix} & \begin{Bmatrix} 1 & 2 & 3 \end{Bmatrix} \\ \begin{Bmatrix} 1 \\ 4 \end{Bmatrix} & \begin{Bmatrix} 1 & 2 \\ 4 & 5 \end{Bmatrix} & \begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{Bmatrix} \\ \begin{Bmatrix} 1 \\ 4 \\ 7 \end{Bmatrix} & \begin{Bmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{Bmatrix} & \begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{Bmatrix} \end{array} \right)$$

Similarly, we have the *bottom-rights* (*botrs*) defined by

$$botrs = (\ominus, \phi)/ \cdot bottoms* \cdot rights$$

The *horizontal-segments* (*hsegs*) are defined by

$$hsegs = (\ominus, \phi)/ \cdot bottoms* \cdot tops$$

and the *vertical-segments* (*vsegs*) by

$$vsegs = (\ominus, \phi)/ \cdot rights* \cdot lefts$$

We can put these functions together to define the rectangles of an array:

$$rects = (\ominus, \phi)/ \cdot botrs* \cdot topls$$

Thus, *rects* returns an array whose elements are the rectangles of a given array. We shall see in the next section that there are various equivalent ways to define *rects*; in particular,

$$rects = (\ominus, \phi)/ \cdot hsegs* \cdot vsegs$$

defines exactly the same function.

4.12 BRTL rules

It is an inevitable consequence of the extra dimensionality provided by arrays that the number of possible algebraic identities goes up by a multiplicative factor. Too many identities are almost as much a problem in program calculation as too few. In this section we describe, in effect, no fewer than ten additional equations. Fortunately, they come in logical groupings.

We start with the fact that

$$(\ominus, \phi)/ \cdot tops* \cdot lefts = (\ominus, \phi)/ \cdot lefts* \cdot tops$$

There are three additional equations of this kind: we can replace *tops* by *bottoms* and *lefts* by *rights*.

From these four equations we can generate four more. Two of them are:

$$\begin{aligned}(\ominus,\phi)/\cdot \mathit{vsegs}{*}\cdot \mathit{tops} &= (\ominus,\phi)/\cdot \mathit{tops}{*}\cdot \mathit{vsegs}\\ (\ominus,\phi)/\cdot \mathit{hsegs}{*}\cdot \mathit{lefts} &= (\ominus,\phi)/\cdot \mathit{lefts}{*}\cdot \mathit{hsegs}\end{aligned}$$

The two additional ones are obtained by replacing *tops* by *bottoms* and *lefts* by *rights*.

Finally, to the above eight there can now be added:

$$\begin{aligned}(\ominus,\phi)/\cdot \mathit{botrs}{*}\cdot \mathit{topls} &= (\ominus,\phi)/\cdot \mathit{vsegs}{*}\cdot \mathit{hsegs}\\ (\ominus,\phi)/\cdot \mathit{vsegs}{*}\cdot \mathit{hsegs} &= (\ominus,\phi)/\cdot \mathit{hsegs}{*}\cdot \mathit{vsegs}\end{aligned}$$

That gives ten. A good way to remember these identities, particularly the last two, is to see that the relative order of *bottoms* (B) and *tops* (T), and the relative order of *rights* (R) and *lefts* (L) remain unchanged throughout all the equations. For example, the last two rules can be captured with the abbreviations

$$\begin{aligned}BR\cdot TL &= RL\cdot BT\\ RL\cdot BT &= BT\cdot RL\end{aligned}$$

Similar abbreviations hold for the other rules.

For a sample proof we shall tackle the fifth equation. For convenience, we introduce the additional abbreviations: θ for $(\ominus,\phi)/$ and V for *vsegs*. The following chain of equational reasoning is given without comments.

$$\begin{aligned}\theta\cdot V{*}\cdot T &= \theta\cdot(\theta\cdot R{*}\cdot L){*}\cdot T\\ &= \theta\cdot\theta{*}\cdot R{*}{*}\cdot L{*}\cdot T\\ &= \theta\cdot\theta\cdot R{*}{*}\cdot L{*}\cdot T\\ &= \theta\cdot R{*}\cdot\theta\cdot L{*}\cdot T\\ &= \theta\cdot R{*}\cdot\theta\cdot T{*}\cdot L\\ &= \theta\cdot\theta\cdot R{*}{*}\cdot T{*}\cdot L\\ &= \theta\cdot\theta{*}\cdot R{*}{*}\cdot T{*}\cdot L\\ &= \theta\cdot(\theta\cdot R{*}\cdot T){*}\cdot L\\ &= \theta\cdot(\theta\cdot T{*}\cdot R){*}\cdot L\\ &= \theta\cdot\theta{*}\cdot T{*}{*}\cdot R{*}\cdot L\\ &= \theta\cdot\theta\cdot T{*}{*}\cdot R{*}\cdot L\\ &= \theta\cdot T{*}\cdot\theta\cdot R{*}\cdot L\\ &= \theta\cdot T{*}\cdot V\end{aligned}$$

4.13 Rectangle decomposition

Let us now turn to the problem of computing R, where

$$R = (\oplus, \oplus)/ \cdot f* \cdot rects$$

We want to show that if

$$B = (\oplus, \oplus)/ \cdot f* \cdot bottoms$$

can be expressed in the form

$$B = g \cdot (\circledast \dagger)$$

for suitable g and $\circledast$, then R can be expressed in the form

$$R = (\oplus, \oplus)/ \cdot h* \cdot rows \cdot (\circledast \ddagger)$$

where h is defined by

$$h = (\oplus, \oplus)/ \cdot g* \cdot vsegs$$

The advantage of this expression for R is that if h (which is essentially a problem about the segments of a list) can be computed in linear time, and if values of $\circledast$ can be computed in constant time, then R can be computed in time proportional to the number of elements in the given array.

We shall need the following observation about *rects*:

$$\begin{array}{rcl}
rects & = & \text{one definition of } rects \\
 & & (\ominus, \phi)/ \cdot vsegs* \cdot hsegs \\
 & = & \text{definition of } hsegs \\
 & & (\ominus, \phi)/ \cdot vsegs* \cdot (\ominus, \phi)/ \cdot bottoms* \cdot tops \\
 & = & \text{map and reduce promotion} \\
 & & (\ominus, \phi)/ \cdot ((\ominus, \phi)/ \cdot vsegs* \cdot bottoms)* \cdot tops \\
 & = & BRTL \text{ rule} \\
 & & (\ominus, \phi)/ \cdot ((\ominus, \phi)/ \cdot bottoms* \cdot vsegs)* \cdot tops
\end{array}$$

Now we argue:

$$
\begin{array}{rcl}
R & = & \text{observation} \\
 & & (\oplus,\oplus)/ \cdot f* \cdot (\ominus,\phi)/ \cdot ((\ominus,\phi)/ \cdot \mathit{bottoms}* \cdot \mathit{vsegs})* \cdot \mathit{tops} \\
 & = & \text{map and reduce promotion} \\
 & & (\oplus,\oplus)/ \cdot ((\oplus,\oplus)/ \cdot f* \cdot (\ominus,\phi)/ \cdot \mathit{bottoms}* \cdot \mathit{vsegs})* \cdot \mathit{tops} \\
 & = & \text{promotion rules} \\
 & & (\oplus,\oplus)/ \cdot ((\oplus,\oplus)/ \cdot ((\oplus,\oplus)/ \cdot f* \cdot \mathit{bottoms})* \cdot \mathit{vsegs})* \cdot \mathit{tops} \\
 & = & \text{assumption on } B \\
 & & (\oplus,\oplus)/ \cdot ((\oplus,\oplus)/ \cdot (g \cdot (\circledast\dagger))* \cdot \mathit{vsegs})* \cdot \mathit{tops} \\
 & = & * \text{ distributivity} \\
 & & (\oplus,\oplus)/ \cdot ((\oplus,\oplus)/ \cdot g* \cdot (\circledast\dagger)* \cdot \mathit{vsegs})* \cdot \mathit{tops} \\
 & = & \text{orthogonal reduction rule} \\
 & & (\oplus,\oplus)/ \cdot ((\oplus,\oplus)/ \cdot g* \cdot \mathit{vsegs} \cdot (\circledast\dagger))* \cdot \mathit{tops} \\
 & = & * \text{ distributivity; } h = (\oplus,\oplus)/ \cdot g* \cdot \mathit{vsegs} \\
 & & (\oplus,\oplus)/ \cdot h* \cdot (\circledast\dagger)* \cdot \mathit{tops} \\
 & = & \text{accumulation lemma} \\
 & & (\oplus,\oplus)/ \cdot h* \cdot \mathit{rows} \cdot (\circledast\ddagger)
\end{array}
$$

4.14 Application

At long last we are in a position to make progress on the problem posed at the beginning of this lecture. The problem is to compute R, where

$$R = \uparrow/ \cdot \mathit{area}* \cdot \mathit{filled}\triangleleft \cdot \mathit{bag} \cdot \mathit{rects}$$

given that R is applied to arrays whose elements are 1 or 0.

The function *bag* is defined by

$$\mathit{bag} = (\uplus,\uplus)/ \cdot \lbag\cdot\rbag*$$

Since we have not defined the filter operation on arrays (and cannot reasonably do so if the result is to be an array), we must first make a bag out of the array of rectangles in order to filter out those rectangles we do not want.

The predicate *filled* can be defined by

$$\mathit{filled} = (\wedge,\wedge)/ \cdot (=1)*$$

The area of a rectangle can be defined by

$$\mathit{area} = (+,+)/ \cdot K_1*$$

In fact, we can make a convenient simplification and eliminate the filter from the specification of R. Define a modified addition operator $\oplus$ by the equation

$$\begin{array}{rcll} a \oplus b & = & 0 & \text{if } a = 0 \vee b = 0 \\ & = & a + b & \text{otherwise} \end{array}$$

This operator is both associative and commutative. If we define

$$area' = (\oplus, \oplus)/$$

then it is easy to show, provided x is an array over $\{0, 1\}$, that

$$\begin{array}{rcll} area'\, x & = & area\, x, & \text{if } \mathit{filled}\, x \\ & = & 0 & \text{otherwise} \end{array}$$

Thus we can eliminate the filter, and reformulate the problem as one of computing R, where

$$R = (\uparrow, \uparrow)/ \cdot (\oplus, \oplus)/* \cdot \mathit{rects}$$

For an array filled with zeros, the new version of R returns zero (while the previous version returned $-\infty$); otherwise the functions are the same.

The new form of R is such that we can try to apply the rectangle decomposition theorem of the previous section. This means that we have to express

$$B = (\uparrow, \uparrow)/ \cdot (\oplus, \oplus)/* \cdot \mathit{bottoms}$$

in the form

$$B = g \cdot (\circledast \not\downarrow)$$

for suitable g and $\circledast$.

A good place to start is to try and use Horner's rule. Fortunately, the first condition of Horner's rule, namely that $\oplus$ distributes through $\uparrow$, is easily seen to be satisfied. Unfortunately, the second condition, which is that $\oplus$ abides with $\uparrow$, is not satisfied.

All is not lost, because we have not yet used the fact that we are only considering arrays over $\{0, 1\}$. The first crucial observation is that, *provided* x is an array over $\{0, 1\}$, we have

$$(\oplus, \oplus)/x = \mathit{width}\, x \times (\oplus, \downarrow)/x$$

The restriction on x is necessary to ensure that the reduction $(\oplus, \downarrow)/x$ is well-defined. The abide condition, namely that

$$(a \oplus b) \downarrow (c \oplus d) = (a \downarrow c) \oplus (b \downarrow d)$$

is only valid when at least one of a, b, c, d is zero, or all are non-zero and equal. If this condition is true of the elements of x (i.e. all non-zero elements are equal in value), then the abide condition is satisfied. For such an x we therefore have

$$B(x) = \mathit{width}\, x \times (\uparrow, \uparrow)/(\oplus, \downarrow)/ * \mathit{bottoms}\, x$$

Furthermore — and this is the crux — under the same assumption about the elements of x, we have that $\downarrow$ abides with $\uparrow$. This gives us Horner's rule.

We can summarise the above observations in the following way. Define

$$g\, x = \mathit{width}\, x \times \downarrow/x$$

and

$$a \circledast b = (a \oplus b) \uparrow b$$

Equivalently, we can define $\circledast$ by

$$\begin{array}{rcll} a \circledast b & = & 0 & \text{if } b = 0 \\ & = & a + b & \text{otherwise} \end{array}$$

Then, provided x is an array over $\{0, 1\}$, we have

$$(\uparrow, \uparrow)/(\oplus, \oplus)/ * \mathit{bottoms}\, x = g(\circledast \not\downarrow x)$$

A formal proof of this result can be given by induction and is left to the reader.

Now, using rectangle decomposition, we obtain

$$R = \uparrow/ \cdot h* \cdot \mathit{rows} \cdot (\circledast \ddagger)$$

where

$$h = \uparrow/ \cdot g* \cdot \mathit{vsegs}$$

We can reformulate h as a function on lists. Replace *rows* in the expression for R by *listrows*, and *vsegs* in the definition of h by *segs*. We then have

$$h = \uparrow/ \cdot f* \cdot \mathit{segs}$$

where

$$f\, x = \#x \times \downarrow/x$$

The new definition of h has a simple intuitive interpretation. Think of a sequence x of nonnegative numbers as representing a *histogram*. The function h then computes the largest rectangular area under the histogram x. In the final lecture we shall show how to compute h in linear time. Since $\circledast$ is computable in constant time, it then follows that R can be computed in time proportional to the number of entries in the array.

4.15 References

The largest filled rectangle problem was invented as a generalisation of a similar problem about largest filled *squares*. This largest filled square problem was posed and solved in

[1] Gries, D. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming* 2 (1982) 207-214

After solving the largest filled rectangle problem, the author learnt of another solution (expressed as a five-pass algorithm) described in

[2] Dean Brock, J. Finding the largest empty rectangle on a grated surface. *Proc. 4th Annual Symp. on Theoretical Aspects of Computer Science*, Passau, W. Germany — in LNCS, No. 247, Springer-Verlag, Berlin, (1987) 66-75.

5 Trees

(*Note:* The material of this section is the result of joint work with Jeremy Gibbons of the PRG, Oxford. For reasons of space, most of the ideas are sketched only briefly. A fuller account will appear elsewhere.)

5.0 Problem

Define a *heap* to be a labelled binary tree t with the property that for each subtree t' of t, the label of t' is a number which is no greater than the labels of all subtrees of t'. Required is an efficient algorithm for converting a sequence of numbers x into a heap whose inorder traversal is x.

5.1 Trees

The type of labelled binary trees (henceforth, just called *trees*) with labels from α will be denoted by $\langle\alpha\rangle$. This type is specified by a free algebra generated from elements of α under the assignment $\langle\cdot\rangle : \alpha \to \langle\alpha\rangle$ which maps elements of α to singleton trees. The constituents of this algebra are:

(1) Two partial operations $\swarrow$ (pronounced 'under') and $\searrow$ (pronounced 'over') such that $\swarrow$ *associates* with $\searrow$, in the sense that

$$(x \swarrow y) \searrow z = x \swarrow (y \searrow z)$$

whenever both sides are defined.

(2) Two predicates *noleft* and *noright* such that $x \swarrow y$ is defined just in the case that *noleft* y holds, and $x \searrow y$ is defined just in the case that *noright* x holds. Moreover, we suppose that

$$\begin{array}{lcl} \textit{noleft}\,\langle a\rangle & = & \textit{True} \\ \textit{noleft}(x \swarrow y) & = & \textit{False} \\ \textit{noleft}(x \searrow y) & = & \textit{noleft}\,x \\ \\ \textit{noright}\,\langle a\rangle & = & \textit{True} \\ \textit{noright}(x \swarrow y) & = & \textit{noright}\,y \\ \textit{noright}(x \searrow y) & = & \textit{False} \end{array}$$

Another way of putting these conditions is to say that *noleft* x holds just in the cases that x is a singleton tree, or x is of the form $\langle a\rangle \searrow y$. Similarly, *noright* x holds just in the cases that x is a singleton tree, or x is of the form $y \swarrow \langle a\rangle$. It follows that the expression

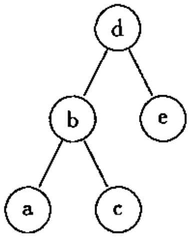
$$x \swarrow y \searrow z$$

is well-defined just in the case that y is a singleton tree.

Let us relate these operations on trees to the usual pictures. For example, the tree

$$(\langle a\rangle \swarrow \langle b\rangle \searrow \langle c\rangle) \swarrow \langle d\rangle \searrow \langle e\rangle$$

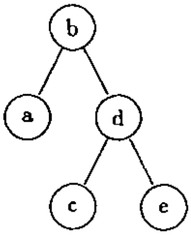
can be drawn in the following way:



On the other hand, the tree

$$\langle a\rangle \swarrow \langle b\rangle \searrow (\langle c\rangle \swarrow \langle d\rangle \searrow \langle e\rangle)$$

can be pictured as follows:



We have not yet defined the notion of an empty tree. If we wish to have such a tree, then we can denote it by $\langle\,\rangle$ and suppose that $\langle\,\rangle$ is the unique left identity of $\swarrow$ and the unique right identity of $\searrow$. Thus

$$\begin{aligned}\langle\,\rangle \swarrow x &= x \\ x \searrow \langle\,\rangle &= x\end{aligned}$$

for all non-empty trees x. We also suppose that $x \swarrow \langle\,\rangle$ and $\langle\,\rangle \searrow x$ are undefined, for otherwise

$$x \swarrow (\langle\,\rangle \searrow y) \neq (x \swarrow \langle\,\rangle) \searrow y$$

and so $\swarrow$ and $\searrow$ no longer associate.

5.2 An alternative view

An alternative, and more common, view of labelled binary trees is in terms of a ternary constructor *Bin*. This view is captured in the type declaration

$$\mathit{tree}\,\alpha ::= \mathit{Nil} \mid \mathit{Bin}\,(\mathit{tree}\,\alpha)\,\alpha\,(\mathit{tree}\,\alpha)$$

(The syntax used here is that of the functional language Miranda; it is also used in the text [3] cited in Lecture 1.) This declaration defines a tree to be either the empty tree *Nil*, or a tree $\mathit{Bin}\,x\,a\,y$ with left subtree x, label a, and right subtree y.

We can move from the ternary view to the binary view by using the equivalences

$$\begin{aligned}\mathit{Nil} &\equiv \langle\,\rangle \\ \mathit{Bin}\,x\,a\,y &\equiv x \swarrow \langle a\rangle \searrow y\end{aligned}$$

A slightly different view of trees excludes the empty tree. This view recognises trees as being of one of the following forms:

$$\begin{gathered}\langle a\rangle \\ x \swarrow \langle a\rangle \\ \langle a\rangle \searrow y \\ x \swarrow \langle a\rangle \searrow y\end{gathered}$$

The advantage of viewing labelled binary trees in terms of two partial binary operators, rather than a single total ternary operator, is primarily that we can set up the notion of reduction on trees in a simple manner. However, each view of trees has its advantages and disadvantages, so we shall employ whichever is the more convenient in a given situation.

5.3 Map and reduce

The next move, which should be familiar by now, is to define the map and reduction operators for trees. The definition of $*$ is given by three equations:

$$\begin{aligned} f * \langle a\rangle &= \langle f\, a\rangle \\ f * (x \swarrow y) &= (f * x) \swarrow (f * y) \\ f * (x \searrow y) &= (f * x) \searrow (f * y)\end{aligned}$$

For reduction we need two operators $\oplus$ and $\otimes$ such that $\oplus$ associates with $\otimes$. We then define

$$\begin{aligned} (\oplus,\otimes)/\langle a\rangle &= a \\ (\oplus,\otimes)/(x \swarrow y) &= ((\oplus,\otimes)/x) \oplus ((\oplus,\otimes)/y) \\ (\oplus,\otimes)/(x \searrow y) &= ((\oplus,\otimes)/x) \otimes ((\oplus,\otimes)/y)\end{aligned}$$

These equations define reduction over non-empty trees. If, in addition, $\oplus$ has a unique left identity element e, and e is also the unique right identity element of $\otimes$, then we can set

$$(\oplus,\otimes)/\langle\,\rangle = e$$

Let us now consider some examples of reduction.

(1) The *label* of a tree is given by

$$\mathit{label} = (\gg,\ll)/$$

The operator $\gg$ associates with $\ll$, for both sides of

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

simplify to b. Note, however, that $\ll$ does not associate with $\gg$, that is,

$$a \ll (b \gg c) \neq (a \ll b) \gg c$$

Here, the left-hand side reduces to a, but the right-hand side reduces to c.

(2) The inorder traversal of a tree is defined by

$$inorder = (+\!\!+, +\!\!+)/ \cdot [\cdot]*$$

(3) The size of a tree is defined by

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

(4) The function *member* x, which determines whether x appears as a label in a given tree, is defined by

$$member\, x = (\vee, \vee)/ \cdot (= x)*$$

In the examples above there is only one operator in the reduction, and this operator is associative. The reduction is well-defined because an associative operator associates with itself. Moreover, each operator possesses an identity element (which is therefore the unique left and right identity element of the operator), and so each of the above functions is defined on the empty tree.

Here are some examples where the operators in a reduction are not the same.

(5) The *depth* of a tree is defined by

$$depth = (\oplus, \tilde{\oplus})/ \cdot K_1*$$

where the operator $\oplus$ is defined by

$$a \oplus b = (1 + a) \uparrow b$$

and, as usual, $a \tilde{\oplus} b = b \oplus a$. Neither $\oplus$ nor $\tilde{\oplus}$ is associative, but nevertheless $\oplus$ associates with $\tilde{\oplus}$. The proof of this fact is left to the reader.

(6) A similar function is *heaporder*, defiued by

$$heaporder = (\oplus, \tilde{\oplus})/ \cdot [\cdot]*$$

where $\oplus$ is defined by

$$x \oplus ([a] \mathbin{+\!\!+} y) = [a] \mathbin{+\!\!+} (x \curlywedge y)$$

We leave to the reader the proof that $\oplus$ associates with $\tilde{\oplus}$, and that $[\,]$ is the unique left identity of $\oplus$ (and therefore the unique right identity of $\tilde{\oplus}$). Thus, we have

$$heaporder\langle\,\rangle = [\,]$$

An alternative definition of *heaporder* can be based on the ternary view of trees:

$$\begin{array}{lcl} heaporder\,\langle\,\rangle & = & [\,] \\ heaporder\,(x \swarrow \langle a\rangle \searrow y) & = & [a] \mathbin{+\!\!+} (heaporder\, x \curlywedge heaporder\, y) \end{array}$$

By definition, a tree x is a heap if $heaptree\, x$ holds, where

$$heaptree = nondec \cdot heaporder$$

Here, *nondec* is a predicate that determines whether a sequence is in non-decreasing order. An alternative definition of *heaptree* is given by the equations:

$$\begin{array}{lcl} heaptree\langle a\rangle & = & True \\ heaptree(x \swarrow y) & = & heaptree\, x \wedge heaptree\, y \wedge label\, x \leqslant label\, y \\ heaptree(x \searrow y) & = & heaptree\, x \wedge heaptree\, y \wedge label\, y \leqslant label\, x \end{array}$$

5.4 Accumulations

Let us now very briefly consider accumulations on trees. There are two kinds: upwards (or towards the root) and downwards (or towards the leaves). For reasons of space we consider only the first kind.

An upwards accumulation (on non-empty trees) is denoted by $\curlywedge\!\!\!\lambda$, and takes a pair of operators $(\oplus, \otimes)$ as its left argument, and a tree x as its right argument. The result is a tree of the same shape as x. The function $up = (\oplus, \otimes)\curlywedge\!\!\!\lambda$ is defined by the equations:

$$\begin{array}{lcl} up\langle a\rangle & = & \langle a\rangle \\ up(x \swarrow \langle a\rangle) & = & up\, x \swarrow \langle b \oplus a\rangle \\ up(\langle a\rangle \searrow y) & = & \langle a \otimes c\rangle \searrow up\, y \\ up(x \swarrow \langle a\rangle \searrow y) & = & up\, x \swarrow \langle b \oplus a \otimes c\rangle \searrow up\, y \\ & & \text{where } b = label(up\, x) \text{ and } c = label(up\, y) \end{array}$$

For example, the function *subtrees* with type

$$subtrees : \langle\alpha\rangle \to \langle\langle\alpha\rangle\rangle$$

can be defined by

$$subtrees = (\swarrow, \searrow) \curlywedge \cdot \langle\cdot\rangle *$$

The expression $subtrees\, x$ evaluates to a tree of exactly the same shape as x but whose labels are the subtrees of x. In particular,

$$label * subtrees\, x = x$$

We shall use this result below.

The accumulation lemma for trees states that

$$(\oplus, \otimes)\curlywedge = (\oplus, \otimes)/* \cdot subtrees$$

The expression on the left can be computed in time linear in the size of the tree, whereas the expression on the right takes quadratic time in the worst case.

For example, the function

$$treesizes = size* \cdot subtrees$$

can be re-expressed as

$$treesizes = (+, +) \curlywedge \cdot K_1 *$$

and therefore can be computed in linear time. (We leave the reader to formulate and prove the necessary subsidiary identity that enable the second definition of *treesizes* to be calculated from the first.)

5.5 Building a heap

The problem stated at the beginning was to build a heap whose inorder traversal was a given sequence. We are required, therefore, to construct a function *heap* satisfying the equations

$$\begin{aligned} inorder(heap\, x) &= x \\ heaptree(heap\, x) &= True \end{aligned}$$

for all lists x. In words, the first equation says that *heap* is a right-inverse of *inorder*, while the second says that *heap* must return a tree satisfying the heap condition. Note that *inorder* is surjective, but not injective. For example, both the trees

$$\langle 1\rangle \searrow (\langle 1\rangle \searrow \langle 2\rangle) \quad \text{and} \quad \langle 1\rangle \swarrow \langle 1\rangle \searrow \langle 2\rangle$$

produce the inorder traversal $[1,1,2]$. Moreover, both trees are heaps, so the specification of *heap* does not determine the function uniquely. Indeterminacy arises because there is a choice as to the relative placement of equal values.

To determine a constructive definition of *heap*, we can look for a solution of the form

$$heap = \oplus/ \cdot f*$$

In other words, we can look for a definition as a homomorphism over lists. Using the specification of *heap*, it is possible to calculate the values of f and $\oplus$. We obtain that $f = \langle\cdot\rangle$ and that $\oplus$ is determined by the equation

$$\begin{array}{lcll} (x \swarrow \langle a\rangle \searrow y) \oplus (u \swarrow \langle b\rangle \searrow v) & = & x \swarrow \langle a\rangle \searrow (y \oplus (u \swarrow \langle b\rangle \searrow v)) & \text{if } a < b \\ & = & ((x \swarrow \langle a\rangle \searrow y) \oplus u) \swarrow \langle b\rangle \searrow v & \text{otherwise} \end{array}$$

together with the condition that $\langle\,\rangle = id_{\oplus}$. We leave to the reader the proof that this definition of *heap* meets its specification.

In this solution, indeterminacy is resolved by placing equal values to the left. More precisely, suppose we define the *right-spine* of a tree by the equation

$$rspine = (\gg, +\!\!+)/ \cdot [\cdot]*$$

Informally, the right-spine of a tree is the sequence of labels obtained by starting at the root and proceeding along the right branches to the rightmost tip. Then our definition of *heap* is such that $rspine(heap\, x)$ is a sequence in strictly increasing order.

5.6 Solution as a left reduction

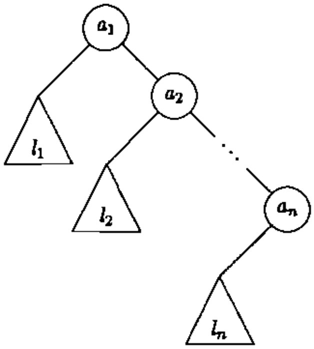
The definition of *heap* as a homomorphism does not prescribe an order of computation. To obtain a sequential algorithm we can specialise the definition to a left reduction:

$$heap = \otimes\!\not\rightarrow_{\langle\,\rangle}$$

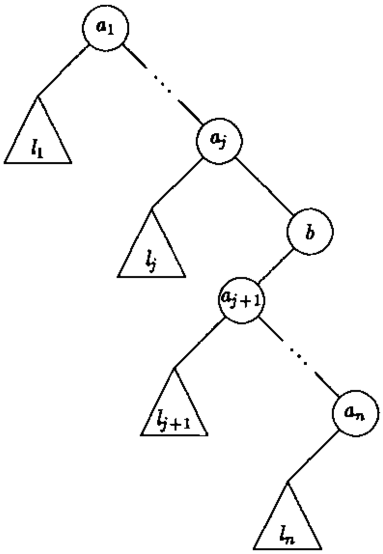
where $x \otimes b = x \oplus f\ b$. Simplification yields:

$$\begin{array}{lcll}
\langle\,\rangle \otimes b & = & \langle b \rangle & \\
(x \swarrow \langle a \rangle \searrow y) \otimes b & = & x \swarrow \langle a \rangle \searrow (y \otimes b) & \text{if } a < b \\
 & = & ((x \swarrow \langle a \rangle \searrow y) \swarrow \langle b \rangle & \text{otherwise}
\end{array}$$

A pictorial interpretation of $\otimes$ is as follows. If x is the heap



then $x \otimes b$ will be the heap



where j is defined by the condition $a_j < b \leqslant a_{j+1}$.

The running time of this algorithm is $O(N^2)$, where N is the length of the argument. Each element b may be compared with every label in the right spine, and the right spine can increase by one in length at each step. A more efficient algorithm can be obtained by comparing b with labels in the right spine, starting with the rightmost label and proceeding to the root. The amount of processing done at each step is then proportional to the change in length of the right spine. This gives a linear time algorithm for building the heap. To implement the idea we need a change in representation.

Consider the function *cut* defined informally by the equation

$$cut(x_1 \searrow (x_2 \searrow (\ldots \searrow x_n))) = [x_1, x_2, \ldots, x_n]$$

where each x_j is such that $noright\, x_j$ holds. Thus, *cut* takes an arbitrary tree and returns a sequence of trees obtained by removing every right branch along the right spine. It is easy to check that *cut* is a bijective function. If we define

$$paste = \searrow\!\not\leftarrow_{\langle\,\rangle}$$

(a definition involving a truly horrendous juxtaposition of arrows) then *cut* and *paste* are inverse functions, that is,

$$\begin{aligned} cut \cdot paste &= id_{[\langle\alpha\rangle]} \\ paste \cdot cut &= id_{\langle\alpha\rangle} \end{aligned}$$

To implement the change in representation, we can modify the definition of *heap* by writing

$$heap = paste \cdot \odot\!\not\rightarrow_{[\,]}$$

where $\odot$ is specified by the equation

$$(cut\, x) \odot b = cut(x \otimes b)$$

Putting it another way, if $\odot$ and $\otimes$ are related by the above equation, then

$$cut \cdot \otimes\!\not\rightarrow_{\langle\,\rangle} = \odot\!\not\rightarrow_{cut\,\langle\,\rangle}$$

and so, by applying *paste* to both sides, we get the new equation for *heap*.

To complete the change in representation, it remains to synthesise a constructive definition of $\odot$ from its specification. Omitting details, we can calculate that

$$\begin{array}{lcll} [\,] \odot b & = & [\langle b \rangle] & \\ ([x \swarrow \langle a \rangle] \mathbin{+\!\!+} xs) \odot b & = & [x \swarrow \langle a \rangle] \mathbin{+\!\!+} (xs \odot b) & \text{if } a < b \\ & = & [paste([x \swarrow \langle a \rangle] \mathbin{+\!\!+} xs) \swarrow \langle b \rangle] & \text{otherwise} \end{array}$$

This effects the change in representation but, in order to achieve the desired increase in efficiency, we still need to change the order in which the elements of the left-hand argument of $\odot$ are processed.

5.7 Prefix and suffix

Let us introduce four new operators on lists. They are $\lrcorner$ ('take prefix'), $\urcorner$ ('drop prefix'), $\llcorner$ ('take suffix') and $\ulcorner$ ('drop suffix'). Each operator takes a predicate on the left and a list on the right. The definitions of $p \lrcorner x$ and $p \llcorner x$ are:

$$\begin{array}{lcl} p \lrcorner x & = & \uparrow_{\#}/ \mathit{all}\, p \triangleleft \mathit{inits}\, x \\ p \llcorner x & = & \uparrow_{\#}/ \mathit{all}\, p \triangleleft \mathit{tails}\, x \end{array}$$

Both operations can be implemented efficiently so that the number of calculations of p equals one more than the number of elements in the result.

The remaining two operators are defined by the equations:

$$\begin{array}{lcl} (p \lrcorner x) \mathbin{+\!\!+} (p \urcorner x) & = & x \\ (p \ulcorner x) \mathbin{+\!\!+} (p \llcorner x) & = & x \end{array}$$

Thus $p \urcorner x$ is what remains when $p \lrcorner x$ is removed from x. A similar statement holds for $p \ulcorner x$.

We state without proof the following lemma.

Lemma 4 *Let x be a sequence and p a predicate such that $p * x$ is non-increasing (taking False < True). Then*

$$p \lrcorner x = \bar{p} \ulcorner x$$

where $\bar{p}$ is the negation of p.

5.8 A linear algorithm

We can now use the newly introduced operators in the construction of a linear time algorithm for our problem about building a heap. Recall that, currently, we have

$$heap = paste \cdot \odot\!\not\rightarrow_{[\,]}$$

where

$$paste = \swarrow\!\!\not{/}_{\langle\,\rangle}$$

and

$$\begin{array}{lcll} [\,] \odot b & = & [\langle b\rangle] & \\ ([x \swarrow \langle a\rangle] \mathbin{+\!\!\!+} xs) \odot b & = & [x \swarrow \langle a\rangle] \mathbin{+\!\!\!+} (xs \odot b) & \text{if } a < b \\ & = & [paste([x \swarrow \langle a\rangle] \mathbin{+\!\!\!+} xs) \swarrow \langle b\rangle] & \text{otherwise} \end{array}$$

Using the operators $\lrcorner$ and $\urcorner$, we can rewrite the definition of $\odot$ in the form

$$xs \odot b = (p_b \lrcorner xs) \mathbin{+\!\!\!+} [paste(p_b \urcorner xs) \swarrow \langle b\rangle]$$

where the predicate p_b is defined by

$$p_b(x \swarrow \langle a\rangle) = (a < b)$$

Since, by the heap assumption, $p_b * xs$ is non-increasing, we have that

$$xs \odot b \;\; = \;\; (\bar{p}_b \ulcorner xs) \mathbin{+\!\!\!+} [paste(\bar{p}_b \llcorner xs) \swarrow \langle b\rangle]$$

With the new definition of $\odot$, the function *heap* can be computed in linear time.

5.9 Application

The representation of sequences by heaps has a number of uses, of which we give just one brief illustration. The problem that arose in the last lecture, namely to compute the area of the largest rectangle under a histogram, can be formulated as a function

$$mra = \uparrow/ \cdot area* \cdot segs$$

where

$$area\, x = \#x \times \downarrow/x$$

We claim without proof that the function *mra* can be written in the form

$$mra = (\uparrow, \uparrow)/ \cdot area'* \cdot subtrees \cdot heap$$

where

$$area'\, x = size\, x \times label\, x$$

Now, suppose we define

$$areas = area'{*} \cdot subtrees$$

Using the results about *subtrees* cited above, we can then calculate

$$\begin{array}{rcl} areas\,x & = & size * subtrees\,x \curlyvee_{\times} label * subtrees\,x \\ & = & ((+,+) \curlywedge K_1 * x) \curlyvee_{\times} x \end{array}$$

Thus *areas* can be computed in linear time. It follows that

$$mra = (\uparrow,\uparrow)/ \cdot areas \cdot heap$$

can also be computed in linear time.

5.10 References

The idea of using heaps to solve certain problems about segments can be found in

[1] De Moor, O. and Swierstra, D. The low segment problem. Presentation at WG2.1, Rome, March 1988.

The largest rectangle under a histogram is a generalisation of Problem 40 in

[2] Rem, M. Small programming exercises. *Science of Computer Programming*. 1987.

Rem's problem is to compute the size of the largest square under a histogram; in symbols,

$$\uparrow/ \cdot \#{*} \cdot p{\triangleleft} \cdot segs,$$

where $p\,x = (\downarrow/x \geqslant \#x.)$