Algebras for Tree Algorithms

Algebras for Tree Algorithms

Jeremy Gibbons
Linacre College, Oxford

A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
at the University of Oxford
September 1991

Technical Monograph PRG-94
ISBN 0-902928-72-4

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

Author's current address:

Department of Computer Science
University of Auckland
Private Bag 92019
Auckland
New Zealand

Electronic mail: jeremy@cs.aukuni.ac.nz

Abstract

This thesis presents an investigation into the properties of various algebras of trees. In particular, we study the influence that the structure of a tree algebra has on the solution of algorithmic problems about trees in that algebra. The investigation is conducted within the framework provided by the Bird-Meertens formalism, a calculus for the construction of programs by equational reasoning from their specifications.

We present three different tree algebras: two kinds of binary tree and a kind of general tree. One of the binary tree algebras, called 'hip trees', is new. Instead of being built with a single ternary operator, hip trees are built with two binary operators which respectively add left and right children to trees which do not already have them; these operators enjoy a kind of associativity property.

Each of these algebras brings with it with a class of 'structure-respecting' functions called catamorphisms; the definition of a catamorphism and a number of its properties come for free from the definition of the algebra, because the algebra is chosen to be initial in a class of algebras induced by a (cocontinuous) functor. Each algebra also brings with it, but not for free, classes of 'structure-preserving' functions called accumulations. An accumulation is a function that preserves the shape of a structured object such as a tree, but replaces each element of that object with some catamorphism applied to some of the other elements. The two classes of accumulation that we study are the 'upwards' and 'downwards' accumulations, which pass information from the leaves of a tree towards the root and from the root towards the leaves, respectively.

Upwards and downwards accumulations turn out to be the key to the solution of many problems about trees. We derive accumulation-based algorithms for a number of problems; these include the parallel prefix algorithm for the prefix sums problem, algorithms for bracket matching and for drawing binary and general trees, and evaluators for decorating parse trees according to an attribute grammar.

Philosophy is written in this grand book—I mean the Universe—which stands continuously open to our gaze, but it cannot be understood unless one first learns to comprehend the language in which it is written. It is written in the language of mathematics, and its characters are triangles, circles and other geometrical figures, without which it is humanly impossible to understand a single word of it; without these, one is wandering about in a dark labyrinth.

Galileo, Il Saggiatore, 1623

Contents

1 Introduction

The concept of a tree is fundamental to all algebra, and hence to algorithm design too; this is because the terms of any recursively defined algebra are trees. Bourbaki's text on algebra (Bourbaki, 1942) has trees as the first definition on page one. Indeed, trees are fundamental, full stop: there is a 'beautiful combinatorial world' of tree-like branching patterns in fields ranging from molecular biology and neurophysiology to hydrogeology and astronomy (Viennot, 1990). Knuth (1968a) traces the history of trees back to the third day of creation, but the mathematical notion of a tree dates from Kirchhoff (1847), who was concerned with finding cycles in electronic circuits, and the name 'tree' in connection with this notion from Cayley (1857), from his series of papers on the structure of arithmetic expressions.

Trees are important in computing because they embody the idea of hierarchical structure. In the context of parallel execution, they permit fast collection and dissemination of information among their elements: the structure-respecting functions on trees can be computed in parallel in time proportional to the depth of their argument. In fact, it could be argued that all algorithms that take logarithmic time, whether sequentially or in parallel, do so because of an underlying tree structure.

The purpose of this thesis is to explore the algebraic properties of a number of species of tree, and in particular to investigate the influence that these properties have on the solution of algorithmic problems about trees. This work forms part of a larger objective, that of the formal calculation of computer programs from their specifications. We have been aware for more than twenty years that any attempt to construct programs by trial and error is doomed to failure; clearly a more systematic approach than this is required.

Program calculation

One methodology that offers some scope for making the construction of programs more mathematical is that of *transformational programming* (Gerhart, 1975; Wegbreit, 1976; Darlington and Burstall, 1976; Burstall and Darlington, 1977; Loveman, 1977; Feather, 1987). This methodology is described by Darlington (1981) as follows:

> *Using the transformational approach to programming, a programmer does not attempt to produce directly a program that is correct, understandable and efficient, rather he initially concentrates on producing a program which is as clear and understandable as possible ignoring any question of efficiency. Having satisfied himself that he has a correct program he successively transforms it to more and more efficient versions using methods guaranteed not to change the meaning of the program.*

In essence, the value of the approach is in its separation of the concerns of correctness and of efficiency and implementability.

There are two properties of a program notation that will greatly simplify the process of program construction by transformation. These properties are that the notation covers a wide spectrum, encompassing both inital 'specification' and final 'implementation', and that it is concise. The first is desirable because it is impossible to make purely local changes between stages of the development, if separate languages are used for different stages: each different stage entails a complete translation of the program from one language to the next (Bauer et al., 1979). The second is desirable because with the transformational approach, the program is rewritten many times with only small variations between successive versions; if the notation is verbose this is very clumsy, and moreover, a verbose notation will obscure the structure of a program, making it difficult to spot the applicability of transformations (Backhouse, 1989; Meertens, 1989a).

The clearest way of presenting a transformational development is to give a linear calculation, proceeding by equational reasoning, from the specification of a program to its implementation. A sequence of programs is presented, each differing from the previous one by the application of some relatively simple transformation of a subexpression. If each transformation preserves the meaning of the program, then all programs in the sequence—but in particular, the first and the last—are equivalent. Ideally, it should be possible to check the applicability of a transformation on a purely *syntactic* basis, without having to interpret the symbols, though of course it is likely that there will be some semantic intuition on the part of the designer in choosing which particular transformation to apply. Feijen's proof format (Dijkstra and Feijen, 1988) provides a clear method of laying out such a calculation: the calculation is displayed in the form

$$\begin{array}{lll}
 & \mathsf{P} & \\
= & & [\![\ \text{hint as to why } \mathsf{P} = \mathsf{Q}\ ]\!] \\
 & \mathsf{Q} & \\
= & & [\![\ \text{hint as to why } \mathsf{Q} = \mathsf{R}\ ]\!] \\
 & \mathsf{R} &
\end{array}$$

There is plenty of room for the hints, minimizing the amount of 'decoding' needed to understand the calculation, and it is clear that 'a step in the calculation is a very local affair' (van Gasteren, 1988), involving only two adjacent expressions and a hint.

Of course, a style of development by equational reasoning like this relies on having 'an *algebra of programs*, a rich collection of identities that hold between different representations of functions' (Backus et al., 1990). This collection of identities can be provided by exploiting the algebraic structure of the data concerned. In particular, there is a close correspondence between data structures (terms in an algebra) and control structures (homomorphisms over that algebra). This correspondence is the same as that between the manipulation of types and of functions by categorical *functors*. This close correspondence is also an aid to conciseness; common patterns of computation over data structures can be encapsulated as 'higher order operators', and the repetitive details elided.

We have discussed various aspects of program calculi in general; we turn now to a particular calculus, the Bird-Meertens formalism, which will form the framework for this thesis.

The remainder of this thesis is structured as follows. Chapter 2 provides a survey of the different species of tree that we will study. Chapters 3 and 4 form the main body of the thesis; in them we introduce the notions of *upwards* and *downwards accumulation* on trees, which embody the ideas of passing information towards the root and towards the leaves of a tree, respectively; it turns out that these accumulations are the basis of the solutions of many algorithmic problems on trees. Chapters 5 and 6 verify this observation by presenting two extended examples of the use of accumulations: Chapter 5 gives a derivation of the 'parallel prefix' algorithm and of two of its applications, running finite state machines and matching brackets, and Chapter 6 gives derivations of algorithms for drawing trees. Chapter 7 shows the connection between accumulations and attribute grammars, which also pass information towards the root and towards the leaves of a tree; we show that the evaluation of attribute grammars is naturally described using accumulations. Finally, in Chapter 8, we present the conclusions we have drawn, and discuss the relationship of the material presented here to other work.

The Bird-Meertens formalism

The *Bird-Meertens formalism* (Meertens, 1986; Bird, 1987, 1988; Backhouse, 1989) is a program calculus possessing all the desirable properties we have mentioned: it covers a broad range of levels of abstraction, it is concise, and it places a heavy emphasis on the algebraic properties of data types, resulting in a rich and powerful body of laws that encourage a calculational style of development based on equational reasoning.

Work within the Bird-Meertens formalism has covered a wide variety of programming paradigms. At one end of the spectrum of abstraction from physical computers, de Moor (1990) at Oxford and Backhouse et al. (1990, 1991) in the Netherlands have been investigating the theory behind a relational, as opposed to functional, approach. Such an approach provides the power of non-determinism, partiality and inverses; de Moor is using it to solve dynamic programming problems. At the other end of this spectrum, groups at Oxford and Glasgow (Luk, 1988; Jones, 1989; Jones and Sheeran, 1990a) have been using a similar formalism to synthesize circuit designs for hardware. Although the extra restrictions of locality and of minimizing wire crossings make hardware design more difficult than software design, the same methods can be used.

The formalism also covers the 'parallelism' axis well. Early work (Bird, 1987; Bird and Meertens, 1987) was based on a distinctly sequential intuition, but Skillicorn (1990) has shown that a language consisting of Bird's operators map, reduce, accumulation, filter and cross product forms a 'truly architecture-independent programming language', in the sense that these operations can be implemented with asymptotically optimal efficiency on any of the 'four major classes of parallel architecture: tightly coupled, SIMD, hypercuboid and constant valence topology multiprocessors'. Thus, the theory of lists is *universal* over these models of parallel computation; this subset of the Bird-Meertens formalism makes just as good a parallel programming language as it does a sequential one. For this reason, we will often talk about the amount of 'effort' or 'work', rather than 'time', that an algorithm requires; this is the product of processing power and execution time, assuming that all the processing power can effectively be used.

Similarly, the formalism is not tied to 'imperative' or to 'functional' implementations: a development will produce a program that can be implemented in either kind of language. The notation may well resemble that of a lean functional language like Backus' FP (Backus, 1978) or Turner's Miranda (Turner, 1985), but that is only because programs written in this sort of language are 'consistently an order of magnitude shorter than the equivalent programs in a conventional high level language' (Turner, 1982); the semantics may well resemble that of a lazy functional language such as Haskell (Hudak et al., 1990), but that is only because this gives the richest and most useful body of laws.

So much for the philosophical aspects of the Bird-Meertens formalism. The rest of this introduction will consist of a more detailed discussion of the notation that we will use.

Types and functions

We use the symbol $\in$ for type judgements, writing $a \in A$ for 'a has type A'. Uppercase letters near the beginning of the alphabet will usually be used for type variables. 'Primitive' types include the unit type $\mathbb{1}$ with unique element it, booleans $\mathbb{B}$ with elements $true$ and $false$, and naturals $\mathbb{N}$, including 0. The use of $\in$ for type membership is not intended to mean that types are sets; some types are too big to be sets, and besides, types are not just collections of elements (Morris, 1973). However, we do call on some set-theoretic properties later on—for example, that injective functions have post-inverses.

The function type former is written $\rightarrow$; the type of functions with source A and target B is $A \rightarrow B$. Function application is written with an infix dot: if $a \in A$ and $f \in A \rightarrow B$ then $f.a \in B$; application is the tightest binding of all operations. Functions are often curried, and application associates to the left, so $f.a.b$ parses as $(f.a).b$; because of this, the type former is *right* associative. Function composition is backwards, is the weakest binding of all operations, and is written with an infix $\circ$, so that $(\mathsf{f} \circ \mathsf{g}).\mathsf{a} = \mathsf{f}.(\mathsf{g}.\mathsf{a})$. The 'constant function' always returning a is written $!\mathsf{a}$ and satisfies $!\mathsf{a}.\mathsf{b} = \mathsf{a}$; the identity function is id.

One characteristic of the Bird-Meertens formalism is that reasoning is carried out at the function level rather than the object level, wherever practical. This makes expressions more concise, of course, but also tends to improve manipulability by reducing variables and simplifying pattern matching. However, we do not stick slavishly to the function level; Bird (1984a) says 'one can argue that the additional information provided by the presence of variables is very important for understanding the meaning of expressions'.

When we do have to resort to a pointwise argument, we often end up writing expressions like $\mathsf{h}.(\mathsf{g}.(\mathsf{f}.\mathsf{a}))$, or $(\mathsf{h} \circ \mathsf{g} \circ \mathsf{f}).\mathsf{a}$; the convention that application is left associative then becomes more of a hindrance than a help. In this situation, we take advantage of an idea of Morgan's (1989), of using a right associative application operator as well as the normal left associative one. We write this operator with an infix centred dot, $\cdot$, and we use it to write expressions like the above without using so many parentheses:

$$\mathsf{h}\cdot\mathsf{g}\cdot\mathsf{f}\cdot\mathsf{a} \quad = \quad \mathsf{h}.(\mathsf{g}.(\mathsf{f}.\mathsf{a}))$$

We give this right associative application the same high precedence as left associative application, instead of following Morgan and making it weakest-binding. Any expressions involving both left and right associative application will be disambiguated with parentheses.

Functors, the pair calculus and binary operators

For quite a while now, we have been used to the idea that data structure and program structure tend to follow the same pattern (Hoare, 1972). In categorical terms, this is the reasoning behind 'functors', data constructions that act on both types and functions: the image of a functor on a type is a data structure, and its image on a function over that type is a corresponding function over that data structure. We denote functor variables by uppercase letters F, G, ... and pre-apply them like we do functions: $F.A$ and $F.f$. Functors respect source and target:

$$f \in A \to B \quad \Rrightarrow \quad F.f \in F.A \to F.B$$

and they preserve identity and composition:

$$\begin{aligned} F.id &= id \\ F.(f \circ g) &= F.f \circ F.g \end{aligned}$$

The two kinds of primitive functor we will use are the identity functor Id and the constant functors $!B$ for various B; their actions on types and functions are given by

$$\begin{aligned} Id.A &= A \qquad & !B.A &= B \\ Id.f &= f \qquad & !B.f &= id \end{aligned}$$

We also use infix *bifunctors* cartesian product $\|$ and disjoint sum or coproduct $|$, acting on pairs of types and of functions; if $f \in A \to C$ and $g \in B \to D$ then $f \| g \in A \| B \to C \| D$ and $f \,|\, g \in A \,|\, B \to C \,|\, D$. Using $\|$ and $|$ instead of the more conventional $\times$ and $+$ leaves the latter free for arithmetic operations. If $a \in A$ and $b \in B$, we write the pair $(a, b) \in A \| B$: in effect, the infix comma is a synonym for the identity on products. We write $\ll \in A \| B \to A$ and $\gg \in A \| B \to B$ for the product destructors or projections, and $< \in A \to A \,|\, B$ and $> \in B \to A \,|\, B$ for the sum constructors or injections.

The product and sum morphisms are written $\curlywedge$ and $\curlyvee$, pronounced 'fork' and 'join'; if $f \in A \to B$ and $g \in A \to C$ then

$$f \curlywedge g \quad \in \quad A \to B \| C$$

and if $h \in B \to D$ and $j \in C \to D$ then

$$h \curlyvee j \quad \in \quad B \,|\, C \to D$$

These operations satisfy a number of laws, among which are

$$\begin{array}{rcl@{\qquad}rcl}
f \parallel g \circ h \parallel j & = & (f \circ h) \parallel (g \circ j) & f \mid g \circ h \mid j & = & (f \circ h) \mid (g \circ j) \\
f \parallel g \circ h \curlywedge j & = & (f \circ h) \curlywedge (g \circ j) & f \curlyvee g \circ h \mid j & = & (f \circ h) \curlyvee (g \circ j) \\
f \curlywedge g \circ h & = & (f \circ h) \curlywedge (g \circ h) & f \circ g \curlyvee h & = & (f \circ g) \curlyvee (f \circ h) \\
\ll \curlywedge \gg & = & id & < \curlyvee > & = & id \\
\ll \circ f \curlywedge g & = & f & f \curlyvee g \circ < & = & f \\
\gg \circ f \curlywedge g & = & g & f \curlyvee g \circ > & = & g
\end{array}$$

Recall that composition is weakest binding.

We write $\mathbf{I\!I}$ for the monofunctor that satisfies

$$\begin{array}{rcl}
\mathbf{I\!I}.A & = & A \parallel A \\
\mathbf{I\!I}.f & = & f \parallel f
\end{array}$$

We also sometimes write f^2 instead of $\mathbf{I\!I}.f$; note that f^2 is the product of functions $f \parallel f$, not their composition $f \circ f$.

Product and sum are not associative. To denote arbitrary products and sums we write $A_0 \parallel \cdots \parallel A_{n-1}$ and $A_0 \mid \cdots \mid A_{n-1}$, which are understood to be applications of n-ary operators; similarly, n-ary forks and joins are written $f_0 \curlywedge \cdots \curlywedge f_{n-1}$ and $f_0 \curlyvee \cdots \curlyvee f_{n-1}$. Because 'left' and 'right' have no simple analogy for arbitrary tuples, we write the projections π_i and injections ι_i on these types.

Another characteristic of the Bird-Meertens formalism is the frequent use of infix binary operators: if the function $\oplus$ has a binary product as its source type, it is written between its arguments. We have seen examples of this already: application, composition, apposition, product, sum, fork and join. Bird (1984a) says, 'Not only can such operators enhance the succinctness and, used sparingly, the readability of expressions, they also allow many transformations to be expressed as algebraic laws about their distributive and other properties.' Related to this, and to the desire to work at the function level wherever practicable to avoid redundant variable names, is the notion of *sectioning*, attributed by Wile (1973) to the mathematical literature (he cites a recursive function theory text).

Sectioning is a form of partial parameterization; a binary operator is given one of its arguments, and it turns into a function of the other argument.

$$\langle a\oplus\rangle.b \quad = \quad a \oplus b \quad = \quad \langle\oplus b\rangle.a$$

We will often omit the sectioning brackets, relying on spacing to make the sense clear. We also section an operator without giving it any arguments; this is just currying:

$$\langle\oplus\rangle.a.b \quad = \quad \oplus.(a,b) \quad = \quad a \oplus b$$

We make the convention that all other binary operators have the same precedence, between those of application and composition, and that most are right associative. The exceptions to this last rule are the few cases in which we define an operator of type $A \parallel B \to A$; repeated applications of such an operator are properly typed only when the operator is left associative.

Two further abbreviations involving binary operators will prove useful later. These are 'converse', $\tilde{\oplus}$, and 'lifting', $\hat{\oplus}$, of a binary operator $\oplus$, defined by

$$\begin{array}{rcl} \tilde{\oplus} & = & \oplus \circ \gg \curlywedge \ll \\ \hat{\oplus} & = & \langle\oplus\circ\rangle \circ \curlywedge \end{array}$$

That is,

$$\begin{array}{rcl} x \,\tilde{\oplus}\, y & = & y \oplus x \\ (f \,\hat{\oplus}\, g).x & = & f.x \oplus g.x \end{array}$$

Initial data types and catamorphisms

We base our notation for type definitions and for catamorphisms, their structure-respecting maps, on Malcolm's work (1990), who in turn bases it on that of Hagino (1987a, 1987b). We use a slightly different notation, but the mathematics is the same and we draw heavily on his results.

A type definition is written in the form

$$X \quad = \quad \tau_0.(F_0.X) \mid \cdots \mid \tau_{n-1}.(F_{n-1}.X)$$

where each F_i is a functor. In most cases, these functors will be *polynomial*, that is, constructed from the identity functor and constant functors using composition, product and sum, though we will see a type defined with a non-polynomial functor in Chapter 2. Informally, this definition says that if $a \in F_i.X$ then $\tau_i.a \in X$, that is, $\tau_i \in F_i.X \to X$. Implicit in this definition is that X is the 'least' type having this property. For example, the definition

$$\mathbb{N} \quad = \quad !0.\mathbb{1} \mid \mathsf{succ}.\mathbb{N}$$

says that $!0.\mathsf{it}$ is in $\mathbb{N}$, and if n is in $\mathbb{N}$ then so is $\mathsf{succ}.n$ (and that nothing else is in $\mathbb{N}$). The functors here are $!\mathbb{1}$ and Id, since $!\mathbb{1}.\mathbb{N} = \mathbb{1}$ and $\mathsf{Id}.\mathbb{N} = \mathbb{N}$.

We formalize this example below.

1. Definition An F-algebra is a pair (A, f) such that $f \in F.A \to A$. ◇

2. Definition A function h is (f, g) F-promotable iff

$$h \circ f \quad = \quad g \circ F.h$$

◇

3. Definition An F-homomorphism from an F-algebra (A, f) to an F-algebra (B, g) is a function in $A \to B$ which is (f, g) F-promotable. ◇

Clearly, F-homomorphisms are closed under composition.

Promotability is apparently a very important algebraic concept, judging by the frequency with which it crops up. For example, it generalizes both distributivity and associativity: a function distributes over an operator $\oplus$ iff it is $(\oplus, \oplus)$ $\mathbb{I}$-promotable, and an operator $\otimes$ with type $A \parallel A \to A$ is associative iff $a\otimes$ is $(\otimes, \otimes)$ $(\mathsf{Id}\ \hat{\parallel}\ !A)$-promotable for all a. (The 'lifted product' functor $\hat{\parallel}$ here is just an instance of binary operator lifting: it satisfies $(\mathsf{F} \mathbin{\hat{\|}} \mathsf{G}).\mathsf{A} = \mathsf{F.A} \| \mathsf{G.A}$ and $(\mathsf{F} \mathbin{\hat{\|}} \mathsf{G}).\mathsf{f} = \mathsf{F.f} \| \mathsf{G.f}$.) Promotability also generalizes conjugation: to say that g is the h-conjugate of f, $\mathsf{g} \circ \mathsf{h} = \mathsf{h} \circ \mathsf{f}$, is just to say that h is (f, g) Id-promotable. Conjugation is the property on which data refinement depends (Jones and Sheeran, 1990b): h is the abstraction function, and f and g the concrete and abstract operations.

The notions of F-algebra and F-homomorphism will only be used in this introduction, but F-promotability will be useful later; if the functor F is clear from the context, we will omit it, saying simply that h is (f, g) promotable.

Returning now to the type definition

$$\mathsf{X} \quad = \quad \tau_0.(\mathsf{F}_0.\mathsf{X}) \mid \cdots \mid \tau_{n-1}.(\mathsf{F}_{n-1}.\mathsf{X})$$

we define the 'collective constructor' τ by

$$\tau \quad = \quad \tau_0 \curlywedge \cdots \curlywedge \tau_{n-1}$$

and the 'collective functor' F by

$$\mathsf{F} \quad = \quad \mathsf{F}_0 \mathbin{\hat{|}} \cdots \mathbin{\hat{|}} \mathsf{F}_{n-1}$$

Thus, F is the lifted sum of the individual functors, and satisfies

$$\begin{aligned} \mathsf{F.A} \quad &= \quad \mathsf{F}_0.\mathsf{A} \mid \cdots \mid \mathsf{F}_{n-1}.\mathsf{A} \\ \mathsf{F.f} \quad &= \quad \mathsf{F}_0.\mathsf{f} \mid \cdots \mid \mathsf{F}_{n-1}.\mathsf{f} \end{aligned}$$

Now, we have $\tau \in \mathsf{F.X} \to \mathsf{X}$, and so (X, τ) is an F-algebra; we complete the definition of the type X up to isomorphism by defining (X, τ) to be an *initial* F-algebra, which is precisely to say that for every F-algebra (A, f) there is a unique F-homomorphism from (X, τ) to (A, f). We assume that we can fix some representative, so we can about *the* initial F-algebra. Initiality gives us the *unique extension property*:

4. Corollary Suppose (X, τ) is the initial F-algebra; then, for given f, there is a unique function that is (τ, f) F-promotable. ◇

The unique extension property turns out to be very important to us, because it eliminates the need for nearly all inductive proofs on initial data types: to show the equality of two functions with source X, where (X, τ) is the initial F-algebra, it suffices to show that they are both (τ, f) F-promotable for the same f. Intuitively, the unique extension property is a form of 'canned induction'; the demonstration that two functions have the same promotion properties is equivalent to that of the base case and the inductive step of an induction proof, and the invocation of the unique extension property corresponds to the ritual steps of the proof.

It is not immediately obvious that these initial data types actually exist. However, a standard result from category theory states that polynomial functors are *cocontinuous*—a 'distributivity through limits' property analogous to continuity elsewhere in mathematics—and Smyth and Plotkin (1982) showed that cocontinuous functors induce initial algebras. Thus, if we restrict ourselves to polynomial F_i, the types we define are guaranteed to exist.

There are actually three abuses of notation in the type definition

$$X \quad = \quad \tau_0.(F_0.X) \mid \cdots \mid \tau_{n-1}.(F_{n-1}.X)$$

The first abuse is that it really concerns an isomorphism, not an equality (Wraith, 1989); we make no excuse for this. The second abuse is that it is $F_0.X \mid \cdots \mid F_{n-1}.X$ that is isomorphic to X, and the constructors τ_i do not come into it; however, we need some way of introducing the constructors as well as the functors, and in the interests of brevity they should both be introduced with the same definition.

The third abuse is that it makes no sense to apply a function τ_i to a type $F_i.X$. Using application here, though, opens the way to a notational abbreviation: when $F_i.X$ is $A \parallel B$, the product of two types, we write the constructor τ_i in infix form, $A \, \tau_i \, B$, because it is a binary operator. For example, we would write the type snocnat of non-empty snoc lists of naturals (the name 'snoc' is the reverse of 'cons', the name of the LISP function that is used for prefixing an element to a list, among other things) as

$$\mathsf{snocnat} \quad = \quad \square.\mathbb{N} \mid \mathsf{snocnat} \mathbin{:\!>} \mathbb{N}$$

writing $\mathsf{snocnat} \mathbin{:\!>} \mathbb{N}$ instead of the clumsier $\mathbin{:\!>}.(\mathsf{snocnat} \parallel \mathbb{N})$. Informally, this definition says that if n is a natural number then the singleton list $\square.n$ is a $\mathsf{snocnat}$, and that if x is a $\mathsf{snocnat}$ and n is a natural number then $x \mathbin{:\!>} n$ is also a $\mathsf{snocnat}$. The functor yielding the source type of $\mathbin{:\!>}$ is $\mathsf{Id} \mathbin{\hat{\parallel}} !\mathbb{N}$, the lifted product of the identity functor and a constant functor.

Another abbreviation we make is that, if $\mathsf{F}_i.\mathsf{X}$ is $\mathbf{1}$, we will usually write τ_i instead of $\tau_i.\mathbf{1}$. For example, the defining equation for $\mathbb{N}$ becomes

$$\mathbb{N} \quad = \quad 0 \mid \mathsf{succ}.\mathbb{N}$$

writing 0 instead of $0.\mathbf{1}$.

Catamorphisms are the promotable functions mentioned in the statement of the unique extension property:

5. Definition Suppose (X, τ) is the initial F-algebra. Then, for given f , the unique function that is (τ, f) F-promotable is called a *catamorphism*, and written $(\!|\mathsf{X}\colon \mathsf{f}|\!)$. ◇

The identity function on any initial data type is a catamorphism, because id is (τ, τ) F-promotable for any F . Thus, the catamorphism built from the constructors of its source type is the identity:

$$(\!|\mathsf{X}\colon \tau|\!) \quad = \quad \mathsf{id}$$

A more interesting example is provided by the function $\#$ on $\mathsf{snocnat}$, which returns the length of a list of naturals. This satisfies

$$\begin{array}{lcll} \# \circ \square & = & !1 & \\ \# \circ \mathbin{:\!>} & = & \oplus \circ (\# \parallel \mathsf{id}) & \quad \text{where} \quad x \oplus a = x + 1 \end{array}$$

These two properties can be combined into a single one by joining the functions together:

$$\# \circ (\Box \curlyvee {:\!\cdot}) \quad = \quad (!1 \curlyvee \oplus) \circ (\mathsf{id} \mid (\# \parallel \mathsf{id}))$$

and so $\#$ is $(\Box \curlyvee {:\!\cdot},\, !1 \curlyvee \oplus)$ $(!\mathbb{N} \mathbin{\hat{|}} (\mathsf{id} \mathbin{\hat{\parallel}} !\mathbb{N}))$-promotable. Now, $\mathsf{snocnat}$ is the initial $(!\mathbb{N} \mathbin{\hat{|}} (\mathsf{id} \mathbin{\hat{\parallel}} !\mathbb{N}))$-algebra, so

$$\# \quad = \quad (\!|\, \mathsf{snocnat}\colon !1 \curlyvee \oplus \,|\!)$$

It is usually more convenient to look at the components of a catamorphism individually, as a function is $(\mathsf{f}_0 \curlyvee \cdots \curlyvee \mathsf{f}_{n-1}, \mathsf{g}_0 \curlyvee \cdots \curlyvee \mathsf{g}_{n-1})$ promotable iff it is $(\mathsf{f}_i, \mathsf{g}_i)$ promotable for each i. Thus, $\#$ is both $(\Box, !1)$ $!\mathbb{N}$-promotable and $({:\!\cdot}, \oplus)$ $(\mathsf{id} \mathbin{\hat{\parallel}} !\mathbb{N})$-promotable. This encourages us to write the components of the catamorphism separately, too; when we write $(\!|\, \mathsf{X}\colon \mathsf{f}_0, \ldots, \mathsf{f}_{n-1} \,|\!)$ we mean $(\!|\, \mathsf{X}\colon \mathsf{f}_0 \curlyvee \cdots \curlyvee \mathsf{f}_{n-1} \,|\!)$. We often omit the source type of the catamorphism if it is clear from context, so we might write

$$\# \quad = \quad (\!|\, !1, \oplus \,|\!)$$

for the length function; this rendition is shorter and more manipulable than the recursive definition given above.

It is often the case that an initial data type is a *polymorphic* type, parameterized by one or more type variables. In this case, some of the F_i will depend on these variables. For example, the type snoc_A of non-empty snoc lists over the type A is given by

$$\mathsf{snoc}_\mathsf{A} \quad = \quad \Box.\mathsf{A} \mid \mathsf{snoc}_\mathsf{A} {:\!\cdot}\, \mathsf{A}$$

The two functors involved here are $!\mathsf{A}$ and $\mathsf{Id} \mathbin{\hat{\parallel}} !\mathsf{A}$, both of which depend on the type variable A. We will now see how to define a functor snoc, which maps this type A to the type snoc_A.

We do this by defining snoc_A in terms of a *bifunctor*, one of whose arguments will be the parameter A. If $\otimes$ is a bifunctor, then the operation $(\mathsf{A}\otimes)$ on types and functions which satisfies

$$\begin{aligned}(\mathsf{A}\otimes).\mathsf{B} &= \mathsf{A}\otimes\mathsf{B}\\ (\mathsf{A}\otimes).\mathsf{f} &= \mathsf{id}\otimes\mathsf{f}\end{aligned}$$

is a functor. In the case of $\mathsf{snoc_A}$, this bifunctor $\otimes$ satisfies

$$\begin{aligned}\mathsf{A}\otimes\mathsf{B} &= \mathsf{A} \mid (\mathsf{B} \parallel \mathsf{A})\\ \mathsf{f}\otimes\mathsf{g} &= \mathsf{f} \mid (\mathsf{g} \parallel \mathsf{f})\end{aligned}$$

and $\mathsf{snoc_A}$ is the initial $(\mathsf{A}\otimes)$-algebra.

This gives us the type part of the functor snoc : it takes a parameter A and yields the type $\mathsf{snoc_A}$. What about its function part? Since snoc is to be a functor, its action on functions must satisfy

$$\mathsf{f}\in\mathsf{A}\rightarrow\mathsf{B} \quad\Rrightarrow\quad \mathsf{snoc.f}\in\mathsf{snoc.A}\rightarrow\mathsf{snoc.B}$$

and it should respect identity and composition. The *map* $\mathsf{f}*$, which applies the function f to every element of a snoc list leaving the structure unchanged, satisfies exactly these conditions; it is given by

$$\begin{aligned}\mathsf{f}* &= (\!|\mathsf{snoc_A}: \square\circ\mathsf{f},\ {:\!\cdot}\circ\mathsf{id}\parallel\mathsf{f}|\!)\\ &= (\!|\mathsf{snoc_A}: (\square \curlyvee {:\!\cdot})\circ(\mathsf{f}\otimes\mathsf{id})|\!)\end{aligned}$$

It turns out that the same procedure works for any parameterized type. If an algebra $(\mathsf{X_A},\tau)$, parameterized on A , is the initial F-algebra, then we define the bifunctor $\otimes$ such that $(\mathsf{A}\otimes) = \mathsf{F}$ (and such that A does not appear free in $\otimes$). Then the X that satisfies

$$\begin{aligned}\mathsf{X.A} &= \mathsf{X_A}\\ \mathsf{X.f} &= (\!|\mathsf{X_A}: \tau\circ\mathsf{f}\otimes\mathsf{id}|\!) \qquad \text{for } \mathsf{f}\in\mathsf{A}\rightarrow\mathsf{B}\end{aligned}$$

is a functor; $\mathsf{X.f}$ is written $\mathsf{f}*$. The proof that X respects identity and composition can be found in Malcolm's thesis, and is omitted here. This X is cocontinuous if $\otimes$ is.

For the sake of brevity, we often omit the parameter A from the type information of a catamorphism, writing $(\!|\mathsf{X}: \mathsf{f}|\!)$ instead of $(\!|\mathsf{X.A}: \mathsf{f}|\!)$.

The treatment we have given here assumes that the type has a single parameter, but the same procedure applies if it has a tuple of parameters, as for example do the types of trees with leaves and branches of different types that we introduce in the next chapter. In this case, the bifunctor generalizes to an $n+1$ -ary functor for tuples of length n , and the map takes an n -tuple of functions.

The types we have seen so far have all been 'free' types, that is, types where the constructors are injective, and so there is exactly one way to construct a given object. There are many interesting non-free types too, types where the constructors satisfy some laws and consequently where some objects can be constructed in more than one way. For example, the type of non-empty 'cat' (short for 'concatenate') lists is given by

$$\mathsf{cat.A} \quad = \quad \square\mathsf{.A} \mid \mathsf{cat.A} \mathbin{+\!\!+} \mathsf{cat.A}$$

modulo the law that $+\!\!+$ is associative. Strictly speaking, the singleton cat list constructor $\square$ should be distinguishable from the singleton snoc list constructor $\square$; the reader may imagine that they are printed in different colours.

Manes and Arbib (1986) say that a definition such as this defines cat.A to be 'the initial object of the category of all models of the specification', that is, the initial algebra among all those (!A î II)-algebras that have an associative binary operator. In essence, we can construct a congruence relation $\approx$ on terms in such an algebra, by taking the congruence closure of the symmetric relation $\sim$ on terms induced by the laws; we then take as objects of the type the congruence *classes* under $\approx$. In the case of cat lists, for example, the 'symmetric relation on terms induced by the laws' $\sim$ relates terms $(x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z$ and $x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z)$ which can be identified by a single application of the associativity property at the top level, and the congruence relation $\approx$ relates those pairs of terms that can be identified by any number of such applications.

The promotability property of catamorphisms can be seen as an evaluation rule, since

$$\llparenthesis f \rrparenthesis \circ \tau \quad = \quad f \circ F.\llparenthesis f \rrparenthesis$$

This reveals a subtlety when the source type is not free. Consider, for example, the cat list catamorphism $h = \llparenthesis \mathsf{cat}\colon f, \oplus \rrparenthesis$; we have

$$\begin{aligned} h.((x +\!\!\!+\, y) +\!\!\!+\, z) \quad &= \quad (h.x \oplus h.y) \oplus h.z \\ h.(x +\!\!\!+\, (y +\!\!\!+\, z)) \quad &= \quad h.x \oplus (h.y \oplus h.z) \end{aligned}$$

Now, the two arguments to the catamorphism on the left hand side are equal, because $+\!\!\!+$ is associative; therefore, the two right hand sides should also be equal, to retain substitutivity, that is, $\oplus$ should be associative (at least on the range of the catamorphism).

In view of this, we make the restriction that a function, and in particular the components of a catamorphism, should respect the laws that hold of the constructors of the source type:

6. Definition A function f respects a relation $\sim$ iff

$$x \sim y \quad \Rrightarrow \quad f.x = f.y$$

◇

7. Property A well-defined function respects the congruence relation generated by the laws on its source type. ◇

8. Corollary The components of a catamorphism respect the congruence relation generated by the laws on its source type. ◇

For example, a cat list catamorphism $\llparenthesis f, \oplus \rrparenthesis \in X \rightarrow A$ is 'proper' only when $\oplus$ is associative, for only then is $(A, f \triangledown \oplus)$ an object of 'the category of all models of the specification'.

We now present a few theorems about catamorphisms that will prove useful later on. None of them are new.

One important result is that any injective function is a catamorphism, because it has a post-inverse.

9. Theorem Suppose f has an initial data type as source; if there exists a g such that $g \circ f = id$, then f is a catamorphism. ◇

Proof If f has source type X with (X, τ) the initial F-algebra, and $g \circ f = id$, then f is $(\tau, f \circ \tau \circ F.g)$ F-promotable. ♡

In particular, any function can be written as the composition of a projection and a catamorphism, because the fork of any function with the identity is a catamorphism (Meertens, 1990):

10. Corollary For any f with an initial data type as source, there exists a catamorphism g such that $f = \ll \circ\, g$. ◇

Proof Just take $g = f \curlywedge id$; $\gg \circ\, g = id$ so g is a catamorphism. ♡

These last two results are usually of theoretical rather than practical interest: they give a method of computing $f{\cdot}\tau{\cdot}x$ in terms of $(F.f){\cdot}x$, but only by throwing away any intermediate result, reconstituting x, applying τ and starting from scratch.

Another important theorem concerning catamorphisms is the promotion theorem (Malcolm, 1990):

11. Theorem If h is (f, g) F-promotable, then

$$h \circ \llparenthesis f \rrparenthesis \quad = \quad \llparenthesis g \rrparenthesis$$

◇

Proof

$$\begin{array}{ll} & h \circ \llparenthesis f \rrparenthesis \circ \tau \\ = & \qquad \llbracket \text{ catamorphisms } \rrbracket \\ & h \circ f \circ F.\llparenthesis f \rrparenthesis \end{array}$$

$$
\begin{array}{ll}
= & [\![\text{ premise }]\!] \\
& g \circ F.h \circ F.\llparenthesis f \rrparenthesis \\
= & [\![\text{ functors }]\!] \\
& g \circ F.(h \circ \llparenthesis f \rrparenthesis)
\end{array}
$$

so $h \circ \llparenthesis f \rrparenthesis$ is (τ, g) promotable, and the result follows from the unique extension property. ♡

The promotion theorem gives the conditions under which a function h can be 'fused' with a catamorphism $\llparenthesis f \rrparenthesis$ to produce another catamorphism. In Chapters 3 to 6 we will be looking for catamorphic solutions to certain problems, and this is the tool we shall use.

A consequence of this theorem, which we will use in Chapter 5, is that a map can always be absorbed into a catamorphism:

12. Corollary Suppose that $(X.A, \tau)$ is the initial $(A\otimes)$-algebra, and $f \in B \otimes C \rightarrow C$ and $g \in A \rightarrow B$. Then

$$\llparenthesis X.B\colon f \rrparenthesis \circ g* \quad = \quad \llparenthesis X.A\colon f \circ (g \otimes \mathrm{id}) \rrparenthesis$$

◇

Proof Firstly, $g*$ is simply an abbreviation for $\llparenthesis X.A\colon \tau \circ (g \otimes \mathrm{id}) \rrparenthesis$. So,

$$
\begin{array}{ll}
& \llparenthesis f \rrparenthesis \circ \tau \circ (g \otimes \mathrm{id}) \\
= & [\![\text{ catamorphisms }]\!] \\
& f \circ (\mathrm{id} \otimes \llparenthesis f \rrparenthesis) \circ (g \otimes \mathrm{id}) \\
= & [\![\ \otimes \text{ is a bifunctor, so } \mathrm{id} \otimes h \text{ commutes with } g \otimes \mathrm{id} \ ]\!] \\
& f \circ (g \otimes \mathrm{id}) \circ (\mathrm{id} \otimes \llparenthesis f \rrparenthesis)
\end{array}
$$

that is, $\llparenthesis f \rrparenthesis$ is $(\tau \circ (g \otimes \mathrm{id}), f \circ (g \otimes \mathrm{id}))$ $(A\otimes)$-promotable. ♡

Finally, the fork of two catamorphisms is itself a catamorphism (Fokkinga, 1990).

13. Theorem Suppose (X, τ) is the initial F-algebra. Then

$$\llparenthesis X\colon f \rrparenthesis \curlywedge \llparenthesis X\colon g \rrparenthesis \quad = \quad \llparenthesis X\colon (f \circ F.\ll) \curlywedge (g \circ F.\gg) \rrparenthesis$$

◇

Proof

$$\begin{array}{ll}
 & (\llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis) \circ \tau \\
= & \qquad \llbracket \curlywedge \rrbracket \\
 & (\llparenthesis f \rrparenthesis \circ \tau) \curlywedge (\llparenthesis g \rrparenthesis \circ \tau) \\
= & \qquad \llbracket \text{ catamorphisms } \rrbracket \\
 & (f \circ F.\llparenthesis f \rrparenthesis) \curlywedge (g \circ F.\llparenthesis g \rrparenthesis) \\
= & \qquad \llbracket \curlywedge, \ll \text{ and } \gg\text{, reintroducing the original fork } \rrbracket \\
 & (f \circ F.(\ll \circ \llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis)) \curlywedge (g \circ F.(\gg \circ \llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis)) \\
= & \qquad \llbracket \text{ functors } \rrbracket \\
 & (f \circ F.\ll \circ F.(\llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis)) \curlywedge (g \circ F.\gg \circ F.(\llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis)) \\
= & \qquad \llbracket \curlywedge \rrbracket \\
 & ((f \circ F.\ll) \curlywedge (g \circ F.\gg)) \circ F.(\llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis)
\end{array}$$

and so $\llparenthesis f \rrparenthesis \curlywedge \llparenthesis g \rrparenthesis$ is $(\tau, (f \circ F.\ll) \curlywedge (g \circ F.\gg))$ F-promotable. ♡

We should pause to consider *why* our notation for type definitions ought to differ from the notations of those who have gone before (Hagino, 1987a; Malcolm, 1990; Verwer, 1990; Fokkinga and Meijer, 1991): it is because our needs our different. If one is concerned with the theory of these definitions in general, as these writers are, then one needs to talk about—and hence to name—the type and the functors, either individually or collectively, but one does not need to talk about the collective constructor much, nor about the individual constructors at all. If, however, one is concerned with the *application* of this theory to the derivation of algorithms, as we are, then the names of the functors are of less importance than the names of the individual constructors; that is why we need to name each τ_i.

The reader may also be wondering why the structure preserving maps over algebras are called homomorphisms, whereas those over initial data types are called catamorphisms. The answer is that homomorphisms and catamorphisms are subtly different. Consider the function constant, which holds of a snoc list whenever all its elements are the same:

$$\mathsf{constant}.x \quad = \quad \mathsf{all}.(=\mathsf{last}.x).x$$

where last and all.p are catamorphisms,

$$\begin{aligned} \mathsf{last} &\quad=\quad (\!|\mathsf{snoc}: \mathsf{id}, \gg|\!) \\ \mathsf{all}.p &\quad=\quad (\!|\mathsf{snoc}: p, \wedge \circ \mathsf{id} \parallel p|\!) \end{aligned}$$

Informally, last returns the last element of a list, and all.p holds of a list iff p holds of each of its elements.

Now, constant is not catamorphic, because it does not promote through $:\!\triangleright$; there is no $\oplus$ such that $\mathsf{constant}.(x :\!\triangleright a) = \mathsf{constant}.x \oplus a$. Intuitively, constant.x does not provide enough information about x to permit computation of $\mathsf{constant}.(x :\!\triangleright a)$.

However, consider now the function pairs, which tuples every element of a snoc list with the following element, and tuples the last element with the first. For example,

$$\mathsf{pairs}.[1, 2, 3] \quad = \quad [(1, 2), (2, 3), (3, 1)]$$

(We write $[a_0, \ldots, a_{n-1}]$ for the list $\square.a_0 :\!\triangleright \cdots :\!\triangleright a_{n-1}$.) This is an invertible function—it has post-inverse $\ll\!*$ —and so it is a catamorphism. In fact, $\mathsf{pairs} = (\!|\square \circ \mathsf{id} \curlywedge \mathsf{id}, \oplus|\!)$ where

$$\begin{aligned} \square.(b, b) \oplus a &\quad=\quad [(b, a), (a, b)] \\ (x :\!\triangleright (b, c)) \oplus a &\quad=\quad x :\!\triangleright (b, a) :\!\triangleright (a, c) \end{aligned}$$

Moreover,

$$\mathsf{all}.(=) \circ \mathsf{pairs} \quad = \quad \mathsf{constant}$$

We cannot prove this using the unique extension property, because $\mathsf{constant}$ is not catamorphic and so does not promote through $:\!\cdot$, but a simple inductive proof does work.

At this point, we should realize that something odd is going on. We have shown that $\mathsf{constant}$ is not a catamorphism, but that it *is* the composition of two catamorphisms, $\mathsf{all}.(=)$ and pairs. That is, catamorphisms are not closed under composition, in contrast to homomorphisms. The catch is that, for a given F, F-homomorphisms are closed under composition, whereas, for some other functor G, the composition of an F-homomorphism with a G-homomorphism need not yield another homomorphism. In our example, although pairs is a homomorphism, it is not a homomorphism to the algebra of snoc lists, formed by $\square$ and $:\!\cdot$; rather, it is a homomorphism to the strange algebra formed by $\square \circ \mathsf{id} \curlywedge \mathsf{id}$ and $\oplus$. Similarly, $\mathsf{all}.(=)$ is a homomorphism on snoc lists, but not a homomorphism on this strange algebra. For this reason, we choose to abandon the notion of 'homomorphisms' in favour of that of 'catamorphisms', being homomorphisms over initial data types.

Acknowledgements

No research can be performed in a vacuum, and none of the work presented in this thesis could have been done without the help and foundations provided by my colleagues. I thank Roland Backhouse, Oege de Moor, Maarten Fokkinga, Johan Jeuring, Geraint Jones, Wayne Luk, Lambert Meertens, David Skillicorn, Bernard Sufrin, Doaitse Swierstra, Jaap van der Woude, Netty van Gasteren, Chris Wright, and all the other squiggolists who have explained their work and mine to me; their suggestions have greatly improved the material in this thesis. I would particularly like to thank my examiners, Geraint and Lambert, for their many helpful suggestions. Most of all, I am indebted to my supervisor Richard Bird, for trying his best to remove my rough edges; the jagged bits that remain are of course solely my responsibility.

I am also grateful to the other denizens of the basement of the Programming Research Group for plying me with coffee, commiserating when the network goes down, and otherwise generally making life interesting. Greg, Jim, Dave, Alan, Quentin, Colin, Steve: thank you!

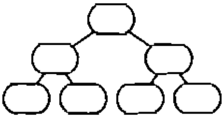
Last, but by no means least, my thanks go to my long-suffering housemate Simon Mercer for putting up with my ramblings and making me put up with his, and to my fellowship group for reminding me that there is more to life than writing theses.

This work was supported for three years by a grant from the Science and Engineering Research Council.

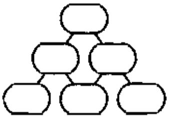
Jeremy Gibbons
Oxford, September 1991

2 A taxonomy of trees

In this thesis we will encounter three different species of tree, and numerous subspecies within these species; we give the details in this chapter. Recall the observation we made in the introduction, that trees are the foundation of algebra. This point is worth reiterating: for us, trees are just terms in some algebra; we do not think of trees as nested collections of sets, nor do we identify them with certain graphs. In particular, because of our algebraic viewpoint we have no way of saying anything about 'sharing of substructures': we cannot distinguish between the tree



and the directed acyclic graph



which results from 'sharing' the middle two leaves of the tree.

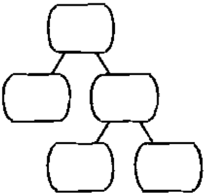
Moo trees

The simplest recursively defined term algebra of all is that of natural numbers, which we saw in the introduction. Of course, the naturals form a rather uninteresting species of tree: every 'parent' has exactly one 'child'. If we generalize the unary constructor succ , which takes a single 'child', to a binary constructor which takes two, we get the branching structure characteristic of trees. We use the operator $\pm$ for this constructor; it is a corruption—introduced for ease of writing—of the Chinese ideogram 木, pronounced 'moo' and meaning 'tree' or 'wood'.

This generalization gives us the type umtree of unlabelled moo trees:

$$\mathsf{umtree} \quad = \quad \triangle \mid \mathsf{umtree} \pm \mathsf{umtree}$$

That is, the empty tree $\triangle$ is an umtree, and if x and y are umtrees then so is $x \pm y$. For example, the expression $\triangle \pm (\triangle \pm \triangle)$ corresponds to the unlabelled tree



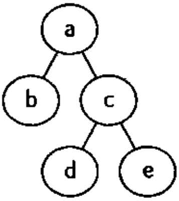
Naturally, unlabelled trees have no labels; the only information in a tree of type umtree is structural information. We can generalize further by labelling the leaves and the branches; this gives us the type mtree of (labelled) moo trees:

14. Definition

$$\mathsf{mtree}.(A, B) \quad = \quad \triangle.A \mid \mathsf{mtree}.(A, B) \pm_B \mathsf{mtree}.(A, B)$$

◇

Informally, if $a \in A$ then $\triangle.a$ is a tree of type $\mathsf{mtree}.(A, B)$, and if x and y are trees of type $\mathsf{mtree}.(A, B)$ and $b \in B$ then $x \pm_b y$ is another tree of type $\mathsf{mtree}.(A, B)$. For example, the expression $\triangle.b \pm_a (\triangle.d \pm_c \triangle.e)$ represents the mtree



To avoid confusion, we always write the type A of leaf labels before the type B of branch labels; similarly, when we come to define catamorphisms on trees, the leaf component will be given first.

The operator $\pm$ is now a *ternary* operator, and no longer simply a binary one; we write the extra argument as a subscript, for lack of anywhere better to put it. This can lead to some notational infelicities if the subscript is large, and for this reason, we extend the notion of sectioning to ternary operators: if ternary operator $\oplus$ has type $\mathsf{X} \parallel \mathsf{A} \parallel \mathsf{Y} \to \mathsf{B}$, then the sectioned operator $\langle\oplus\rangle$ is a function of type $\mathsf{A} \to \mathsf{X} \parallel \mathsf{Y} \to \mathsf{B}$, a function yielding a binary operator given the subscript argument. That is, $\langle\oplus\rangle.\mathsf{a}$ is the binary operator $\oplus_\mathsf{a}$ which, when applied to the pair (x,y), yields $\mathsf{x} \oplus_\mathsf{a} \mathsf{y}$. This means that, instead of writing '$\oplus$ where $\mathsf{u} \oplus_\mathsf{b} \mathsf{v} = \mathsf{u} \pm_{\mathsf{g.b}} \mathsf{v}$', as we would otherwise have had to do on page 42, we can write $\langle\pm\rangle \circ \mathsf{g}$.

That mtree *is* a generalization of umtree is clear from the fact that umtree is isomorphic to $\mathsf{mtree}.(\mathbf{1},\mathbf{1})$, the type of labelled moo trees where neither leaves nor branches carry any useful information. In fact, we will take this as the definition of umtree:

15. Definition

$$\mathsf{umtree} \quad = \quad \mathsf{mtree}.(\mathbf{1},\mathbf{1})$$

◇

It is tedious to have to write $\triangle.\mathsf{it}$ for the 'empty' tree and $\pm_{\mathsf{it}}$ for the binary moo operator, so we will abbreviate these to $\triangle$ and $\pm$, relying on context to dispel confusion. The advantage of making umtree a special case of mtree is of course that everything we say about mtree will hold automatically of umtree .

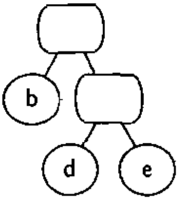
Two types that are intermediate in generality between mtree and umtree are the types of *leaf-labelled* and *branch-labelled* moo trees lmtree and bmtree , where the branch labels and the leaf labels, respectively, carry no information.

16. Definition

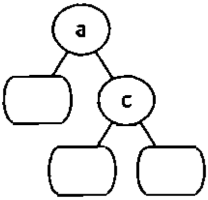
$$\begin{array}{rcl} \mathsf{lmtree}.A & = & \mathsf{mtree}.(A, \mathbb{1}) \\ \mathsf{bmtree}.B & = & \mathsf{mtree}.(\mathbb{1}, B) \end{array}$$

◇

We use the same abbreviations— ± with lmtree and △ with bmtree — as we do with unlabelled trees. Examples of these two tree types are the leaf-labelled tree expression $\triangle.b \pm (\triangle.d \pm \triangle.e)$, which corresponds to the tree



and the branch-labelled tree expression $\triangle \pm_a (\triangle \pm_c \triangle)$, which corresponds to



The last specialization of mtree that we will come across is that of *homogeneous* moo trees, where the leaf and branch labels have the same type.

17. Definition

$$\mathsf{hmtree}.A \quad = \quad \mathsf{mtree}.(A, A)$$

◇

If we have need of a name for the moo trees that are in none of the special cases, we will call them 'general (moo) trees'.

Moo tree catamorphisms

The definition of moo tree catamorphisms is completely determined by the definition of moo trees:

$$\begin{aligned} (\!|f, \oplus|\!) \circ \vartriangle &= f \\ (\!|f, \oplus|\!) \circ \underline{+} &= \oplus \circ ((\!|f, \oplus|\!) \parallel id \parallel (\!|f, \oplus|\!)) \end{aligned}$$

That is,

$$\begin{aligned} (\!|f, \oplus|\!).(\vartriangle.a) &= f.a \\ (\!|f, \oplus|\!).(x \underline{+}_b\, y) &= (\!|f, \oplus|\!).x \oplus_b (\!|f, \oplus|\!).y \end{aligned}$$

Of course, this definition holds for all specializations of mtree as well as for the general case, but for these there are some notational simplifications that we can make. For trees with no leaf labels, we have

$$(\!|f, \oplus|\!).(\vartriangle.it) \quad = \quad f.it$$

We have already said that we will write $\triangle$ instead of $\triangle.\mathsf{it}$; this gives us the more natural voicing

$$(\!|!\mathsf{e}, \oplus|\!).\triangle \quad = \quad \mathsf{e}$$

for the above equation, when we note that any function with source $\mathbb{1}$ is a constant function.

Similarly, for trees with no branch labels, we have

$$(\!|\mathsf{f}, \oplus|\!).(\mathsf{x} \pm_{\mathsf{it}} \mathsf{y}) \quad = \quad (\!|\mathsf{f}, \oplus|\!).\mathsf{x} \oplus_{\mathsf{it}} (\!|\mathsf{f}, \oplus|\!).\mathsf{y}$$

Again, we abbreviate $\mathsf{x} \pm_{\mathsf{it}} \mathsf{y}$ to $\mathsf{x} \pm \mathsf{y}$; in the same way, we write just $\oplus$ instead of $\oplus_{\mathsf{it}}$, and so this equation becomes

$$(\!|\mathsf{f}, \oplus|\!).(\mathsf{x} \pm \mathsf{y}) \quad = \quad (\!|\mathsf{f}, \oplus|\!).\mathsf{x} \oplus (\!|\mathsf{f}, \oplus|\!).\mathsf{y}$$

The *map* operation on moo trees takes a pair of functions, since in general moo trees have two base types; it applies one function to each leaf label and the other function to each branch label in the tree:

$$(\mathsf{f}, \mathsf{g})* \quad = \quad (\!|\triangle \circ \mathsf{f}, \langle\pm\rangle \circ \mathsf{g}|\!)$$

(Recall that $(\langle\pm\rangle \circ \mathsf{g}).\mathsf{b}.(\mathsf{u}, \mathsf{v}) = \mathsf{u} \pm_{\mathsf{g}.\mathsf{b}} \mathsf{v}$.) We make some more notational abbreviations for the special cases: for homogeneous (including unlabelled) trees, we write $\mathsf{f}*$ for $(\mathsf{f}, \mathsf{f})*$; for $\mathsf{lmtree.A}$ with A different from $\mathbb{1}$, we write $\mathsf{f}*$ for $(\mathsf{f}, !\mathsf{it})*$; for $\mathsf{bmtree.B}$ with B different from $\mathbb{1}$, we write $\mathsf{f}*$ for $(!\mathsf{it}, \mathsf{f})*$.

The identity catamorphism is, of course, the catamorphism built from the constructors:

$$\mathsf{id} \quad = \quad (\!|\triangle, \pm|\!)$$

The root of a homogeneous moo tree is given by the catamorphism

$$\mathsf{root} \quad = \quad (\!|\mathsf{id}, \pi_1|\!)$$

Here, π_1 is the projection returning the middle element of a triple.

The function depth, returning the number of elements on the longest path from the root to a leaf, is given by

$$\mathsf{depth} \quad = \quad (\![!1, \oplus]\!) \qquad \text{where} \quad u \oplus_a v = 1 + (u \uparrow v)$$

Note that depth gives the same result, 1, for both the 'empty' branch-labelled tree $\triangle$ and the 'singleton' leaf-labelled tree $\triangle.a$.

The analogue of the length function # on lists is a bit more difficult. Consider the two functions

$$\begin{aligned} \mathsf{leaves} \quad &= \quad (\![!1, \oplus]\!) \qquad \text{where} \quad u \oplus_a v = u + v \\ \mathsf{branches} \quad &= \quad (\![!0, \otimes]\!) \qquad \text{where} \quad u \otimes_a v = u + 1 + v \end{aligned}$$

returning the number of leaves and the number of branches in a tree, respectively. A popular undergraduate exercise in structural induction is to show that

$$\langle 1+ \rangle \circ \mathsf{branches} \quad = \quad \mathsf{leaves}$$

which, since $\langle 1+ \rangle$ is $(\otimes, \oplus)$ $(\mathsf{Id} \mathbin{\hat{\parallel}} !A \mathbin{\hat{\parallel}} \mathsf{Id})$-promotable, is an immediate consequence of the promotion theorem. Now, define the function elements by

$$\mathsf{elements} \quad = \quad \mathsf{leaves} \mathbin{\hat{+}} \mathsf{branches}$$

Again, thanks to the promotion theorem,

$$\mathsf{elements} \quad = \quad (\![!1, \otimes]\!)$$

The awkwardness in defining the size of a tree lies in our intuition concerning 'size'. Intuitively, we might expect the general tree $\triangle.b \pm_a (\triangle.d \pm_c \triangle.e)$ and the unlabelled tree $\triangle \pm (\triangle \pm \triangle)$ both to have size 5, but the leaf-labelled tree $\triangle.b \pm (\triangle.d \pm \triangle.e)$ to have size 3 and the branch-labelled tree $\triangle \pm_a (\triangle \pm_c \triangle)$ to have size 2. That is, 'size' ought to mean branches on branch-labelled (but not unlabelled) trees, leaves on leaf-labelled (but not unlabelled) trees, and elements everywhere else. Because of this awkwardness, we will stick with the three separate functions.

Our last example of moo tree catamorphisms will consist of the various traversals of trees. Similar complications arise here, due again to conflicting intuitions; we will present examples just for homogeneous trees, leaving the generalizations to the reader. For homogeneous trees, then, preorder, inorder and postorder traversals, each returning a cat list, are given by

$$\begin{array}{rcllc} \mathsf{preorder} & = & (\!|\square, \oplus|\!) & \text{where} & u \oplus_a v = \square.a \mathbin{+\!\!+} u \mathbin{+\!\!+} v \\ \mathsf{inorder} & = & (\!|\square, \oplus|\!) & \text{where} & u \oplus_a v = u \mathbin{+\!\!+} \square.a \mathbin{+\!\!+} v \\ \mathsf{postorder} & = & (\!|\square, \oplus|\!) & \text{where} & u \oplus_a v = u \mathbin{+\!\!+} v \mathbin{+\!\!+} \square.a \end{array}$$

We present a fourth kind of traversal, levelorder traversal, in Chapter 6.

Rose trees

We arrived at binary trees from natural numbers by generalizing the constructor succ , which takes one 'child', to the constructor $\underline{\times}$, which takes two. Another generalization that we might have made is to a constructor which takes a *list* of children; this generalization gives what Meertens (1988) calls 'rose trees'. Meertens allows his lists of children to be empty, so permitting parents with no children; to avoid this rather strange concept we use non-empty lists.

Unlabelled rose trees are given by the definition

$$\mathsf{urtree} \quad = \quad \triangle \mid \prec \cdot \mathsf{snoc} \cdot \mathsf{urtree}$$

The constructor $\prec$ could be pronounced 'tree' in this context. Generalizing unlabelled rose trees in the same way that we did unlabelled moo trees, we get general rose trees.

18. Definition

$$\mathsf{rtree}.(A, B) \quad = \quad \triangle.A \mid B \prec \mathsf{snoc} \cdot \mathsf{rtree} \cdot (A, B)$$

◇

for which the constructor $\prec$ is more naturally pronounced 'above'. As with lists, we make no excuse for using the constructor $\vartriangle$ for both moo and rose trees. We can define unlabelled, leaf-labelled, branch-labelled and homogeneous rose trees in terms of rtree, just as we did for moo trees.

19. Definition

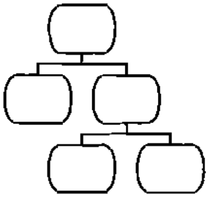
$$\begin{array}{rcl}
\mathsf{urtree} & = & \mathsf{rtree}.(\mathbf{1},\mathbf{1}) \\
\mathsf{lrtree}.\mathsf{A} & = & \mathsf{rtree}.(\mathsf{A},\mathbf{1}) \\
\mathsf{brtree}.\mathsf{B} & = & \mathsf{rtree}.(\mathbf{1},\mathsf{B}) \\
\mathsf{hrtree}.\mathsf{A} & = & \mathsf{rtree}.(\mathsf{A},\mathsf{A})
\end{array}$$

◇

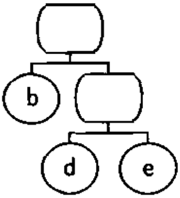
We use the same kind of abbreviations as for binary trees, writing $\vartriangle$ for $\vartriangle.\mathsf{it}$ and $\prec.\mathsf{x}$ for $\mathsf{it} \prec \mathsf{x}$.

Some example rose trees are:

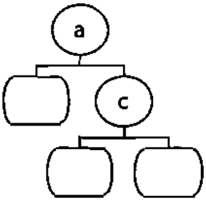
* the unlabelled rose tree $\prec.[\vartriangle, \prec.[\vartriangle, \vartriangle]]$, which might be drawn



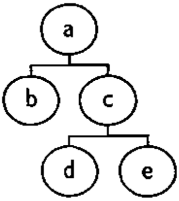
* the leaf-labelled rose tree $\prec.[\vartriangle.\mathsf{b}, \prec.[\vartriangle.\mathsf{d}, \vartriangle.\mathsf{e}]]$, which would be drawn



- the branch-labelled rose tree $a \prec [\triangle, c \prec [\triangle, \triangle]]$:



- the general rose tree $a \prec [\triangle.b, c \prec [\triangle.d, \triangle.e]]$:



There is a complication with rose trees, in that their defining functor is non-polynomial. The type rtree.(A, B) is the initial F-algebra, where

$$F.X \quad = \quad A \mid (B \parallel snoc.X)$$

The occurrence of the functor snoc in this expression makes F non-polynomial, that is, not built solely from identity and constant functors, sum, product and composition. This means that the standard results about polynomial functors being cocontinuous do not apply. However, for any A, the type $\mathsf{snoc}.A$ is itself the initial algebra formed by a cocontinuous functor; Malcolm (1990) showed that any functor like snoc satisfying this property is in turn cocontinuous: 'functors induced by parameterized Hagino types are [co]continuous if their defining functors are [co]continuous'.

Rose tree catamorphisms

Rose tree catamorphisms satisfy the equations

$$\begin{aligned} (\!| f, \oplus |\!) \circ \triangle &= f \\ (\!| f, \oplus |\!) \circ \prec &= \oplus \circ \mathrm{id} \parallel (\!| f, \oplus |\!)* \end{aligned}$$

That is,

$$\begin{aligned} (\!| f, \oplus |\!) \cdot \triangle \cdot a &= f.a \\ (\!| f, \oplus |\!) \cdot (b \prec x) &= b \oplus ((\!| f, \oplus |\!) * x) \end{aligned}$$

The $*$ here is a map over snoc lists.

As with moo trees, we make some notational abbreviations for the special cases. If the leaf type is $\mathbf{1}$, we write

$$(\!| !e, \oplus |\!).\triangle = e$$

and if the branch type is $\mathbf{1}$, we write as the second component the list function $\langle \mathrm{it}\oplus \rangle$ instead of the binary operator $\oplus$:

$$(\!| f, g |\!).(\prec .x) = g.((\!| f, g |\!) * x)$$

The rose tree map is given by

$$(f, g)* = (\!| \triangle \circ f, \prec \circ (g \parallel \mathrm{id}) |\!)$$

The only abbreviation that we will bother to make for map on rose trees is to write $f*$ instead of $(f, f)*$ on homogeneous trees.

We now give a few examples of rose tree catamorphisms. The identity catamorphism is, of course, given by building a catamorphism from the constructors:

$$\mathsf{id} \quad = \quad \llparenthesis \triangle, -\!\!\prec \rrparenthesis$$

The root of a homogeneous rose tree is given simply by

$$\mathsf{root} \quad = \quad \llparenthesis \mathsf{id}, \ll \rrparenthesis$$

The depth of a rose tree is given by

$$\mathsf{depth} \quad = \quad \llparenthesis !1, \langle 1+) \circ \llparenthesis \mathsf{id}, \uparrow \rrparenthesis \circ \gg \rrparenthesis$$

where $\uparrow$ returns the greater of its arguments. That is, the depth of a branch is one greater than the largest of the depths of its children.

The numbers of leaves, branches and elements in a rose tree are similar:

$$\begin{array}{rcl} \mathsf{leaves} & = & \llparenthesis !1, \llparenthesis \mathsf{id}, + \rrparenthesis \circ \gg \rrparenthesis \\ \mathsf{branches} & = & \llparenthesis !0, \langle 1+) \circ \llparenthesis \mathsf{id}, + \rrparenthesis \circ \gg \rrparenthesis \\ \mathsf{elements} & = & \llparenthesis !1, \langle 1+) \circ \llparenthesis \mathsf{id}, + \rrparenthesis \circ \gg \rrparenthesis \end{array}$$

Another example is provided by Dewey Decimal labelling (Knuth, 1968a), which returns a rose tree of *cons* lists; the latter are defined by

$$\mathsf{cons.A} \quad = \quad \square.\mathsf{A} \mid \mathsf{A} \mathbin{\cdot\!:} \mathsf{cons.A}$$

We introduce here the operator $\curlyvee$, pronounced 'zip', which is the post-inverse of $\ll\!* \curlywedge \gg\!*$ on pairs of snoc lists:

$$\curlyvee \circ (\ll\!* \curlywedge \gg\!*) \quad = \quad \mathsf{id}$$

That is, $\curlyvee$ satisfies

$$\begin{array}{rcl} \square.a \curlyvee \square.b & = & \square.(a, b) \\ (x \mathbin{:\!\cdot} a) \curlyvee (y \mathbin{:\!\cdot} b) & = & (x \curlyvee y) \mathbin{:\!\cdot} (a, b) \end{array}$$

on pairs of equal-length lists. We extend its definition to pairs of same-shaped trees in Chapter 5, and to pairs of different-length lists in Chapter 6. The idiom $(\oplus)* \circ \Upsilon$ crops up so often that we define an abbreviation for it:

$$\Upsilon_{\oplus} \;=\; (\oplus)* \circ \Upsilon$$

The function dewey is now given by

$$\mathsf{dewey} \;=\; \llparenthesis !(\square\cdot 0), (\prec) \circ (!(\square.0) \parallel (\mathsf{index}\ \hat{\Upsilon}_{\oplus}\ \mathsf{id})) \rrparenthesis$$

where

$$\mathsf{n} \oplus \mathsf{t} \;=\; \langle \mathsf{n}\cdot: \rangle * \mathsf{t}$$

and

$$\begin{aligned} \mathsf{index}\cdot\square\cdot\mathsf{t} &= \square\cdot 0 \\ \mathsf{index}.(\mathsf{x} \mathbin{:\!\cdot} \mathsf{t}) &= \mathsf{index}.\mathsf{x} \mathbin{:\!\cdot} \#.\mathsf{x} \end{aligned}$$

so that $\# \circ \mathsf{index} = \#$ and $\mathsf{index} = \llparenthesis \mathsf{snoc}: !(\square.0), \mathbin{:\!\cdot} \circ (\mathsf{id} \curlywedge \#) \circ \ll \rrparenthesis$. The function index simply replaces every element of a list with its position in the list, with the first position being 0.

In every example $\llparenthesis \mathsf{f}, \oplus \rrparenthesis$ of a rose tree catamorphism that we have seen so far, the snoc list function $\mathsf{b}\oplus$ has been a catamorphism. Although this appears to be true in nearly all 'natural' cases, it is not necessary, and in general $\mathsf{b}\oplus$ may simply be a list *function*. Consider, for example, the rather contrived predicate funny on leaf-labelled boolean rose trees: a leaf is funny iff its value is true, and a branch is funny iff all its children are funny, *or* all its children are unfunny:

$$\begin{aligned} \mathsf{funny}\cdot\triangle\cdot\mathsf{a} &= \mathsf{a} \\ \mathsf{funny}\cdot\prec\cdot\mathsf{x} &= \mathsf{constant}\cdot(\mathsf{funny} * \mathsf{x}) \end{aligned}$$

where $\mathsf{constant}$ is the predicate introduced at the end of Chapter 1, which holds of a list whenever all its elements are the same. Now, funny is a catamorphism

$$\mathsf{funny} \quad = \quad (\!|\mathsf{id}, \mathsf{constant}|\!)$$

but, as we saw in Chapter 1, constant is not a list catamorphism.

Hip trees

The third and last species of tree that we will encounter is that of *hip trees*, a name coined by Geraint Jones. Hip trees are a kind of homogeneous binary tree, but instead of being built from a single constructor $\star$ they use two constructors $\swarrow$ and $\searrow$, pronounced 'under' and 'over'. Intuitively, the tree $t \swarrow u$ is formed by adding t as a left child to u; similarly, $t \searrow u$ is formed by adding u as a right child to t. Thus, $\swarrow$ and $\searrow$ are a little like left and right hips, whence the name. These two operators satisfy the law

$$(t \swarrow u) \searrow v \quad = \quad t \swarrow (u \searrow v)$$

We say that '$\swarrow$ associates with $\searrow$'.

Formally, the type htree.A of hip trees with elements of type A is given by

20. Definition

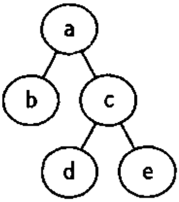
$$\mathsf{htree.A} \quad = \quad \triangle.\mathsf{A} \mid \mathsf{htree.A} \swarrow \mathsf{htree.A} \mid \mathsf{htree.A} \searrow \mathsf{htree.A}$$

modulo the law that $\swarrow$ associates with $\searrow$. ◇

For example, the hip tree expression

$$\triangle.\mathsf{b} \swarrow \triangle.\mathsf{a} \searrow (\triangle.\mathsf{d} \swarrow \triangle.\mathsf{c} \searrow \triangle.\mathsf{e})$$

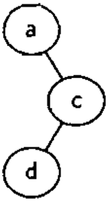
represents the by now familiar binary tree



whereas

$$\triangle.a \searrow (\triangle.d \swarrow \triangle.c)$$

represents the tall thin tree



Different presentations of hip trees can be found elsewhere (Gibbons, 1988; Bird, 1988).

Hip tree catamorphisms

The definition of hip tree catamorphisms is, of course, determined completely by the type definition:

$$\begin{aligned}
(\!|\text{htree}: f, \oplus, \otimes|\!) \circ \triangle &= f \\
(\!|\text{htree}: f, \oplus, \otimes|\!) \circ \swarrow &= \oplus \circ (\!|\text{htree}: f, \oplus, \otimes|\!)^2 \\
(\!|\text{htree}: f, \oplus, \otimes|\!) \circ \searrow &= \otimes \circ (\!|\text{htree}: f, \oplus, \otimes|\!)^2
\end{aligned}$$

Moreover, $\oplus$ must associate with $\otimes$.

One example hip tree catamorphism is the function elements, which returns the number of elements in a hip tree:

$$\mathsf{elements} \quad = \quad (\!|\,!1, +, +\,|\!)$$

Another is the function $\mathsf{inorder}$, returning a list consisting of the elements of its argument in the order given by an inorder traversal:

$$\mathsf{inorder} \quad = \quad (\!|\,\square, +\!\!+, +\!\!+\,|\!)$$

Yet another is the predicate $\mathsf{some.p}$, which holds of a tree iff some element of that tree satisfies p:

$$\mathsf{some.p} \quad = \quad (\!|\,\mathsf{p}, \vee, \vee\,|\!)$$

and the predicate $\mathit{all}.\mathsf{p}$, similarly:

$$\mathsf{all.p} \quad = \quad (\!|\,\mathsf{p}, \wedge, \wedge\,|\!)$$

In all these cases, the two binary operators are the same and are associative, that is, the first associates with the second. In fact, they are all special cases of the following theorem.

21. Theorem

$$(\!|\,\mathsf{cat}\colon \mathsf{f}, \oplus\,|\!) \circ \mathsf{inorder} \quad = \quad (\!|\,\mathsf{htree}\colon \mathsf{f}, \oplus, \oplus\,|\!)$$

if $\oplus$ is associative. ◇

Proof $(\!|\,\mathsf{cat}\colon \mathsf{f}, \oplus\,|\!)$ is $(+\!\!+, \oplus)$ promotable. ♡

Informally, the lifting of any cat list catamorphism to hip trees is still a catamorphism.

More interesting examples are the functions root and depth:

$$\begin{aligned} \mathsf{root} &= (\!|\,\mathsf{id}, \gg, \ll\,|\!) \\ \mathsf{depth} &= (\!|\,!1, \oplus, \tilde{\oplus}\,|\!) \qquad \text{where} \quad \mathsf{x} \oplus \mathsf{y} = (1 + \mathsf{x}) \uparrow \mathsf{y} \end{aligned}$$

Now, $\gg$ associates with $\ll$:

$$\begin{array}{ll}
 & \mathsf{x} \gg (\mathsf{y} \ll \mathsf{z}) \\
= & \qquad \llbracket\ \gg\ \rrbracket \\
 & \mathsf{y} \ll \mathsf{z} \\
= & \qquad \llbracket\ \gg\ \rrbracket \\
 & (\mathsf{x} \gg \mathsf{y}) \ll \mathsf{z}
\end{array}$$

In fact, we did not need any properties of $\ll$ at all, which means that $\gg$ associates with *anything*, and by symmetry anything associates with $\ll$. However, $\ll$ does not associate with $\gg$, because $\mathsf{x} \ll (\mathsf{y} \gg \mathsf{z})$ is x whereas $(\mathsf{x} \ll \mathsf{y}) \gg \mathsf{z}$ is z.

For depth, we have

$$\begin{array}{ll}
 & \mathsf{x} \oplus (\mathsf{y}\ \bar{\oplus}\ \mathsf{z}) \\
= & \qquad \llbracket\ \oplus\text{, twice}\ \rrbracket \\
 & (1 + \mathsf{x}) \uparrow ((1 + \mathsf{z}) \uparrow \mathsf{y}) \\
= & \qquad \llbracket\ \uparrow \text{ is associative and commutative}\ \rrbracket \\
 & (1 + \mathsf{z}) \uparrow ((1 + \mathsf{x}) \uparrow \mathsf{y}) \\
= & \qquad \llbracket\ \oplus\text{, twice}\ \rrbracket \\
 & (\mathsf{x} \oplus \mathsf{y})\ \bar{\oplus}\ \mathsf{z}
\end{array}$$

and again the catamorphism is proper.

For our last two examples, we consider conversions between moo and hip trees. One such conversion is the function hmh from homogeneous moo trees $\mathsf{hmtree.A}$ to hip trees $\mathsf{htree.A}$, which satisfies

$$\begin{array}{rcl}
\mathsf{hmh}\cdot\triangle\cdot\mathsf{a} & = & \triangle\cdot\mathsf{a} \\
\mathsf{hmh}\cdot(\mathsf{x} \pm_{\mathsf{a}} \mathsf{y}) & = & \mathsf{hmh}\cdot\mathsf{x} \swarrow \triangle\cdot\mathsf{a} \searrow \mathsf{hmh}\cdot\mathsf{y}
\end{array}$$

and so

$$\mathsf{hmh} \quad = \quad (\!|\mathsf{hmtree}\colon \triangle, \oplus|\!) \qquad \text{where} \quad \mathsf{t} \oplus_{\mathsf{a}} \mathsf{u} = \mathsf{t} \swarrow \triangle\cdot\mathsf{a} \searrow \mathsf{u}$$

Note that hmh is not surjective: hip trees in which some children are only children have no obvious counterpart as homogeneous moo trees.

A conversion in the opposite direction is provided by the function hbm from hip trees to branch-labelled moo trees,

$$\mathsf{hbm} \quad \in \quad \mathsf{htree.A} \rightarrow \mathsf{bmtree.A}$$

given by

$$\mathsf{hbm} \quad = \quad \llparenthesis \mathsf{htree}\colon \pm \circ (!\triangle \curlywedge \mathsf{id} \curlywedge !\triangle), \oplus, \otimes \rrparenthesis$$

where

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

Again, the conversion is not surjective, because there is no empty hip tree.

Consistency and partiality

As a kind of 'sanity check' on a type with laws, we will want to ensure that the laws we have chosen are *consistent* with the intended model, that is, that they do not equate terms that are intuitively 'different'; the quotient under the congruence relation $\approx$ will always exist, but too strong a collection of laws will make it collapse to fewer congruence classes than the intended model has objects.

One way of performing this check is to exhibit a free algebra that is 'obviously' isomorphic to the intended model, and to show that this algebra is also isomorphic to the quotient algebra; this guarantees that the congruence relation $\approx$ does not identify terms that are different in the intended model. It does not matter if this free algebra is clumsy to work with: its purpose is solely to show that the laws are not too strong, and after serving this purpose it can be forgotten.

For example, recall from Chapter 1 the type cat.A of non-empty cat lists over a type A. It is not difficult to show that $\mathsf{cat.A}$ is isomorphic to the free algebra $\mathsf{snoc.A}$ — the cat catamorphism $(\!|\square, \oplus|\!)$, where the associative operator $\oplus$ satisfies $\langle x\oplus\rangle = (\!|\langle x{:}\!\cdot\rangle, {:}\!\cdot|\!)$, and the snoc catamorphism $(\!|\square, (+\!\!\!+) \circ (\mathsf{id} \parallel \square)|\!)$ are each other's inverses—and so the associativity property is indeed consistent.

However, if we try to perform this check on hip trees, we find that it is not obvious what kind of tree they are isomorphic to—it is clear that, for example, $\triangle.a \swarrow \triangle.b \searrow \triangle.c$ represents the same tree as $\triangle.a \pm_b \triangle.c$, but what tree does the expression

$$\mathsf{odd} \quad = \quad (\triangle.a \swarrow \triangle.b \searrow \triangle.c) \swarrow (\triangle.d \swarrow \triangle.e \searrow \triangle.f)$$

represent? Evidently, some more laws are required in order to make the algebra an algebra of trees.

It seems that choosing these laws is not at all straightforward. If we add the two laws

$$\begin{aligned} t \swarrow (u \swarrow v) &= t \swarrow v \\ (t \searrow u) \searrow v &= t \searrow v \end{aligned}$$

which express the fact that adding a branch destroys any branch that was already present, then all sorts of functions cease to be catamorphisms—in fact, the only ones remaining out of those we have seen so far are root and hbm. Adding this law breaks the antisymmetry of the relation 'is a component of' on hip tree terms, and the process of structural induction (Burstall, 1969) relies on this ordering being well-founded.

A more sensible suggestion is to make $\swarrow$ and $\searrow$ associative, so that

$$\begin{aligned} t \swarrow (u \swarrow v) &= (t \swarrow u) \swarrow v \\ (t \searrow u) \searrow v &= t \searrow (u \searrow v) \end{aligned}$$

in which case $\swarrow$ and $\searrow$ add children at the 'bottom left corner' and 'bottom right corner' of the tree. This does give us an algebra corresponding to our intuition of trees, contrary to Meertens' (1989b) observation that 'associativity kills treehood', but it still means that depth is not a catamorphism: if $x \oplus y = (1 + x) \uparrow y$, then neither $\oplus$ nor $\tilde{\oplus}$ is associative, and indeed the depth of $t \swarrow u$ cannot be determined from just the depths of t and u.

The solution we would like is to say that the term odd 'does not correspond to a tree': that the operators $\swarrow$ and $\searrow$ are *partial*, and that $t \swarrow u$ is 'undefined' (whatever that may mean) if u already has a left branch, and $t \searrow u$ undefined if t already has a right branch. That is, define the predicate proper, which holds of a hip tree expression if it 'represents a proper tree', by

$$\begin{array}{rcl} \mathsf{proper}.(\triangle.a) & = & \mathsf{true} \\ \mathsf{proper}.(t \swarrow u) & = & \mathsf{proper}.t \wedge \mathsf{proper}.u \wedge \mathsf{le}.u \\ \mathsf{proper}.(t \searrow u) & = & \mathsf{proper}.t \wedge \mathsf{proper}.u \wedge \mathsf{re}.t \end{array}$$

where le and re (short for 'left empty' and 'right empty') hold of trees which have no left branch and no right branch, respectively:

$$\begin{array}{rcl} \mathsf{le} & = & \llparenthesis !\mathsf{true}, !\mathsf{false}, \ll \rrparenthesis \\ \mathsf{re} & = & \llparenthesis !\mathsf{true}, \gg, !\mathsf{false} \rrparenthesis \end{array}$$

Since proper is not a catamorphism, we should check from first principles that it respects the associativity of $\swarrow$ and $\searrow$; indeed,

$$\begin{array}{ll} & \mathsf{proper}.((t \swarrow u) \searrow v) \\ = & \qquad \llbracket\ \mathsf{proper}\ \rrbracket \\ & \mathsf{proper}.t \wedge \mathsf{le}.u \wedge \mathsf{proper}.u \wedge \mathsf{re}.u \wedge \mathsf{proper}.v \\ = & \qquad \llbracket\ \mathsf{proper}\ \rrbracket \\ & \mathsf{proper}.(t \swarrow (u \searrow v)) \end{array}$$

We would like to say that a hip tree is defined iff it satisfies proper.

Partial algebras, though, are notoriously difficult to work with. In particular, we would like to avoid any weakening of equality, because it plays such a crucial role in our calculational style. For instance, if we weaken equality so that it holds vacuously when either side is undefined, then it ceases to be transitive—but our Feijen-style calculations lean heavily on the transitivity of equality: if that goes, we must check separately the definedness of each step of each calculation. Alternatively, if we use an asymmetric 'refinement ordering' instead of equality, as Morris (1987) and Morgan (1990) do for their Refinement Calculus, then we can only apply some equations in one direction—we can no longer use the unfold-fold style of reasoning (Burstall and Darlington, 1977), on which we also rely.

So, we have a dilemma. On the one hand, we want to exhibit a free algebra isomorphic to htree, in order to demonstrate the consistency of the associativity property enjoyed by $\swarrow$ and $\searrow$; constructing such an algebra is made difficult by the fact that we have no intuition for the 'improper' hip trees. On the other hand, we do not want to eliminate these improper terms by making $\swarrow$ and $\searrow$ partial, because partial algebras introduce complications that we could do without.

The solution to this dilemma is that we need only exhibit a free algebra isomorphic to the proper subset. For, suppose that we have an algebra A with laws $\approx$, and a predicate p on A. Suppose also that we have a model B, an algebra isomorphic to the subset of terms of A that satisfy p, that is, we have $f \in A \rightarrow B$ and $g \in B \rightarrow A$ such that

$$f \circ g \quad = \quad id$$

and

$$p \circ g \quad = \quad !true$$

but only that

$$g \cdot f \cdot a \quad = \quad a \qquad \text{if} \quad p \cdot a$$

If $\approx$ relates two proper terms a and b, then $f \cdot a$ and $f \cdot b$ are equal, because f must respect $\approx$—that is, a and b correspond to the same object in B; the laws do not relate proper terms that correspond to different objects.

Returning to our dilemma, we need only show that the proper hip trees are isomorphic to some free algebra; this is given by the following theorem. This means that we can forget about making $\swarrow$ and $\searrow$ partial, and can instead remain firmly rooted in a calculus of total functions.

22. Theorem The proper hip trees are isomorphic to the free algebra

$$\mathsf{ftree.A} \quad = \quad \triangle.\mathsf{A} \mid \mathsf{ftree.A} \boxslash \mathsf{A} \mid \mathsf{A} \boxbslash \mathsf{ftree.A} \mid \mathsf{ftree.A} \boxtimes_{\mathsf{A}} \mathsf{ftree.A}$$

◇

Proof Define the functions $\mathsf{fh} \in \mathsf{ftree.A} \rightarrow \mathsf{htree.A}$ by

$$\mathsf{fh} \quad = \quad \llparenthesis \triangle, \swarrow \circ \mathsf{id} \parallel \triangle, \searrow \circ \triangle \parallel \mathsf{id}, \otimes \rrparenthesis \qquad \text{where} \quad \mathsf{t} \otimes_{\mathsf{a}} \mathsf{u} = \mathsf{t} \swarrow \triangle{\cdot}\mathsf{a} \searrow \mathsf{u}$$

and $\mathsf{hf} \in \mathsf{htree.A} \rightarrow \mathsf{ftree.A}$ by

$$\mathsf{hf} \quad = \quad \llparenthesis \triangle, \oslash, \obslash \rrparenthesis$$

where

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{x} \oslash \triangle{\cdot}\mathsf{a} & = & \mathsf{x} \boxslash \mathsf{a} & \triangle{\cdot}\mathsf{a} \obslash \mathsf{z} & = & \mathsf{a} \boxbslash \mathsf{z} \\
\mathsf{x} \oslash (\mathsf{y} \boxslash \mathsf{a}) & = & \mathsf{x} \boxslash \mathsf{a} & (\mathsf{x} \boxslash \mathsf{a}) \obslash \mathsf{z} & = & \mathsf{x} \boxtimes_{\mathsf{a}} \mathsf{z} \\
\mathsf{x} \oslash (\mathsf{a} \boxbslash \mathsf{z}) & = & \mathsf{x} \boxtimes_{\mathsf{a}} \mathsf{z} & (\mathsf{a} \boxbslash \mathsf{y}) \obslash \mathsf{z} & = & \mathsf{a} \boxbslash \mathsf{z} \\
\mathsf{x} \oslash (\mathsf{y} \boxtimes_{\mathsf{a}} \mathsf{z}) & = & \mathsf{x} \boxtimes_{\mathsf{a}} \mathsf{z} & (\mathsf{x} \boxtimes_{\mathsf{a}} \mathsf{y}) \obslash \mathsf{z} & = & \mathsf{x} \boxtimes_{\mathsf{a}} \mathsf{z}
\end{array}$$

Then

(i) $\mathsf{proper} \circ \mathsf{fh} = !\mathsf{true}$

(ii) $\mathsf{hf} \circ \mathsf{fh} = \mathsf{id}$

(iii) $(\mathsf{fh} \circ \mathsf{hf}) \mid \mathsf{id} \circ \mathsf{proper?} = \mathsf{proper?}$, where

$$\mathsf{p?a} \quad = \quad \begin{cases} <{\cdot}\mathsf{a} & \text{if} \quad \mathsf{p{\cdot}a} \\ >{\cdot}\mathsf{it} & \text{otherwise} \end{cases}$$

These results follow from the unique extension property; the calculations are straightforward but lengthy, so we omit them. ♡

Not all algebras that are intuitively partial are as well behaved as hip trees. For example, Wright (1988) has been doing work on two-dimensional 'arrays' or matrices similar to ours on trees. Arrays are constructed from an embedding and two associative binary operators $\ominus$ and $\obar$, pronounced 'above' and 'beside'. The intuitive model of arrays as rectangular matrices only holds up when all subterms of a term 'conform'—two arrays must have the same 'width' if they are to be placed one above the other, and the same 'height' if placed side by side. The operators $\ominus$ and $\obar$ enjoy a kind of distributivity property called the *abiding* law (Bird, 1988):

$$(w \obar x) \ominus (y \obar z) \quad = \quad (w \ominus y) \obar (x \ominus z)$$

provided that all four parenthesized terms conform. Jeffrey (1990) has shown that if the abiding law is strengthened to hold unconditionally, then the intuitive model breaks: this stronger law entails identities such as

$$\boxed{\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{matrix}} \quad = \quad \boxed{\begin{matrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}}$$

using the obvious graphic representation for rectangular arrays. Thus, the dilemma between consistency and manipulability is not so easily avoided: if the algebra is to be consistent with the model, either the constructors must be made partial—introducing the foundational complications we wish to avoid—or the laws must be weakened to their guarded forms—requiring that calculations be peppered with side conditions and checks of conformity—or both.

3 Upwards accumulation

We have now covered the background material on which we will build and can proceed with the main topic of this thesis, which is the study of *accumulations* on trees. Applying an accumulation to a structured object such as a list or a tree leaves the 'shape' of that object unchanged, but replaces every element of that object with some 'accumulated information' about—that is, a catamorphism applied to—other elements. Accumulations are a very common pattern of computation; the encapsulation of these patterns as higher order operators creates a powerful structuring tool, making programs clearer to read and easier to manipulate. Moreover, accumulations can provide an efficiency-improving transformation, if the naive computation of the accumulated information can be replaced by a more careful incremental computation; the naive computation corresponds to the declarative description of a result, and the incremental computation to a more efficient but less perspicuous way of achieving it.

The *upwards* and *downwards* accumulations that we discuss in this chapter and the next are two instances of this general scheme. Upwards accumulation replaces every element of a tree with some catamorphism applied to its *descendants*, and so corresponds to passing information up the tree from the leaves to the root; downwards accumulation consists of replacing every element with some catamorphism applied to its *ancestors*, and so passes information down the tree from the root towards the leaves. (Computing, as has been noted before, is one of the few areas in which trees have their leaves planted in the ground and their roots waving in the air.)

We start by reviewing accumulations on lists (Bird, 1987).

Accumulations on lists

Consider the function inits,

$$\mathsf{inits} \;\in\; \mathsf{snoc.A} \rightarrow \mathsf{snoc{\cdot}snoc{\cdot}A}$$

which returns the list of initial segments of a snoc list; for example,

$$\mathsf{inits.[a, b, c]} \;=\; \mathsf{[[a], [a, b], [a, b, c]]}$$

Intuitively, inits replaces every element of the list with the list of that element's predecessors. Formally, inits is characterized by the equations

$$\begin{aligned} \mathsf{inits}{\cdot}\Box{\cdot}\mathsf{a} &= \Box{\cdot}\Box{\cdot}\mathsf{a} \\ \mathsf{inits}{\cdot}(\mathsf{x} \mathbin{:\!>} \mathsf{a}) &= \mathsf{inits}{\cdot}\mathsf{x} \mathbin{:\!>} (\mathsf{x} \mathbin{:\!>} \mathsf{a}) \end{aligned}$$

Although it is not immediately obvious from these equations, inits is a snoc list catamorphism; for it to match the pattern for catamorphisms, the only occurrences of x on the right hand side of the second equation should be as part of the expression $\mathsf{inits.x}$. However, it is easy to see that

$$\mathsf{last} \circ \mathsf{inits} \;=\; \mathsf{id}$$

so, by Theorem 9, inits is catamorphic:

$$\mathsf{inits.(x} \mathbin{:\!>} \mathsf{a)} \;=\; \mathsf{inits.x} \mathbin{:\!>} (\mathsf{last{\cdot}inits{\cdot}x} \mathbin{:\!>} \mathsf{a})$$

and so

$$\mathsf{inits} \;=\; (\!|\Box \circ \Box, \oplus|\!) \qquad \text{where} \quad \mathsf{u} \oplus \mathsf{a} = \mathsf{u} \mathbin{:\!>} (\mathsf{last.u} \mathbin{:\!>} \mathsf{a})$$

An important property that holds of inits, and of the analogous functions on trees that we will introduce shortly, is given by the following theorem.

23. Theorem

$$\mathsf{inits} \circ \mathsf{f}* \;=\; \mathsf{f}{*}{*} \circ \mathsf{inits}$$

◇

Proof By the unique extension property, using the property that

$$\mathsf{f}{*}{*} \circ \oplus \quad = \quad \oplus \circ \mathsf{f}{*}{*} \parallel \mathsf{f}$$

♡

Categorically speaking, Theorem 23 is the statement that inits is a *natural transformation* from snoc to snoc ∘ snoc , the latter being the functor that maps A to snoc·snoc·A and f to f** . Intuitively, a polymorphic function like inits cannot 'examine' the elements of its argument, and so can do no more than blindly 'rearrange' these elements; mapping a function over the elements cannot change the way in which they are rearranged.

Functions that 'pass information from left to right' are characterized by the following definition.

24. Definition We call a function g *rightwards* if it can be written in the form

$$\mathsf{g} \quad = \quad \mathsf{h}{*} \circ \mathsf{inits}$$

for some h . ◇

Suppose g is rightwards, and can be written in the above form with an h that takes linear sequential time; this gives us a quadratic time algorithm for computing g . Now,

$$\begin{array}{ll}
 & \mathsf{g}\cdot(\mathsf{x} :\!\cdot\, \mathsf{a}) \\
= & \qquad [\![\text{ rewrite } \mathsf{g}]\!] \\
 & \mathsf{h} * \mathsf{inits}\cdot(\mathsf{x} :\!\cdot\, \mathsf{a}) \\
= & \qquad [\![\mathsf{inits}]\!] \\
 & \mathsf{h} * (\mathsf{inits}\cdot\mathsf{x} :\!\cdot\, (\mathsf{x} :\!\cdot\, \mathsf{a})) \\
= & \qquad [\![*]\!] \\
 & (\mathsf{h} * \mathsf{inits}\cdot\mathsf{x}) :\!\cdot\, \mathsf{h}\cdot(\mathsf{x} :\!\cdot\, \mathsf{a})
\end{array}$$

$$
\begin{array}{ll}
= & \llbracket \text{ fold } \mathsf{g} \text{ again } \rrbracket \\
 & \mathsf{g \cdot x} \mathbin{:\!>} \mathsf{h \cdot (x} \mathbin{:\!>} \mathsf{a)}
\end{array}
$$

This does not change the efficiency of computing g; if h takes linear time, for example, then this characterization of g will still take quadratic time. However, suppose h is a snoc list catamorphism:

$$\mathsf{h \cdot (x} \mathbin{:\!>} \mathsf{a)} \quad = \quad \mathsf{h \cdot x} \oplus \mathsf{a}$$

for some $\oplus$ taking constant time; then

$$\mathsf{g \cdot (x} \mathbin{:\!>} \mathsf{a)} \quad = \quad \mathsf{g \cdot x} \mathbin{:\!>} \mathsf{(h \cdot x} \oplus \mathsf{a)}$$

Moreover,

$$
\begin{array}{ll}
 & \mathsf{last} \circ \mathsf{g} \\
= & \llbracket \text{ unfold } \mathsf{g} \text{ } \rrbracket \\
 & \mathsf{last} \circ \mathsf{h}{*} \circ \mathsf{inits} \\
= & \llbracket \text{ promotion: } \mathsf{last} \circ \mathsf{h}{*} = \mathsf{h} \circ \mathsf{last} \text{ } \rrbracket \\
 & \mathsf{h} \circ \mathsf{last} \circ \mathsf{inits} \\
= & \llbracket \text{ inits has post-inverse last } \rrbracket \\
 & \mathsf{h}
\end{array}
$$

and so

$$\mathsf{g \cdot (x} \mathbin{:\!>} \mathsf{a)} \quad = \quad \mathsf{g \cdot x} \mathbin{:\!>} \mathsf{(last \cdot g \cdot x} \oplus \mathsf{a)}$$

and g can be computed in linear time, using only a linear number of applications of $\oplus$. What is more, g is a catamorphism:

$$\mathsf{g} \quad = \quad (\!|\, \square \circ \mathsf{h} \circ \square, \otimes \,|\!) \qquad \text{where} \quad \mathsf{u} \otimes \mathsf{a} = \mathsf{u} \mathbin{:\!>} \mathsf{(last \cdot u} \oplus \mathsf{a)}$$

Functions of this form are what we mean by accumulations on lists:

25. Definition Functions of the form $(\!|\mathsf{f}, \oplus|\!)* \circ \mathsf{inits}$ are called *rightwards accumulations*, and are written $(\mathsf{f}, \oplus)\mathbin{\#\!\!\!\!\rightarrow}$. ◇

Bird (1987) calls these functions 'left accumulations', a contraction of 'left-to-right accumulations', following functional programming convention, but this name is confusing in view of the rightwards-pointing arrow he uses to denote it.

Functions which are rightwards but not rightwards accumulations are awkward to deal with, since they are both inefficient and intractable; where possible, we try to find rightwards accumulations instead, which being catamorphic are both efficient and easy to manipulate.

Rightwards accumulation can be seen as a generalization of inits ; where inits replaces every element of a list with its predecessors, a rightwards accumulation replaces every element with *some catamorphism applied to* its predecessors. That it is a generalization and not just a variation follows from the fact that the identity is a catamorphism:

$$
\begin{aligned}
& \mathsf{inits} \\
= \quad & \qquad [\![\text{ identity }]\!] \\
& \mathsf{id}* \circ \mathsf{inits} \\
= \quad & \qquad [\![\text{ identity catamorphism }]\!] \\
& (\!|\square, \mathbin{:\!\cdot}|\!)* \circ \mathsf{inits} \\
= \quad & \qquad [\![\mathbin{\#\!\!\!\!\rightarrow}]\!] \\
& (\square, \mathbin{:\!\cdot})\mathbin{\#\!\!\!\!\rightarrow}
\end{aligned}
$$

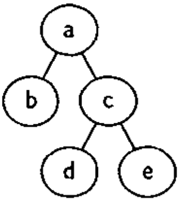
The equation

$$(\mathsf{f}, \oplus)\mathbin{\#\!\!\!\!\rightarrow} \quad = \quad (\!|\mathsf{f}, \oplus|\!)* \circ \mathsf{inits}$$

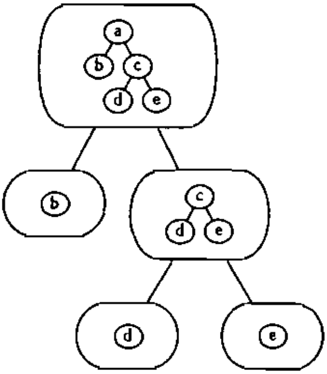
could be seen as an efficiency-improving transformation, when used from right to left. It can also, of course, be used from left to right, when it forms a 'manipulability-improving' transformation: we know many useful properties about inits, $*$ and catamorphisms, and so the right hand side may be more amenable to calculation than the left. Most of the time, however, we aim simply to express functions as accumulations—we know then that we can both implement them efficiently, by writing them in the form of the left hand side, and manipulate them readily, by writing them in the form of the right hand side.

Generalizing to moo trees

We now show how all this can be generalized to trees. The function inits on lists replaces every element of a list with its predecessors, that is, with the initial segment of the list that ends with that element. By analogy, the function subtrees on trees replaces every element of a tree with its *descendants*, that is, with the subtree rooted at that element. For example, applying subtrees to the tree



yields the tree of trees



The following definition characterizes subtrees .

26. Definition

$$\begin{aligned} \mathsf{subtrees}.(\triangle.a) &= \triangle{\cdot}\triangle{\cdot}a \\ \mathsf{subtrees}.(x \pm_b y) &= \mathsf{subtrees}.x \pm_{x \pm_b y} \mathsf{subtrees}.y \end{aligned}$$

◇

Again, since

$$\mathsf{root} \circ \mathsf{subtrees} = \mathsf{id}$$

we can calculate that

$$\mathsf{subtrees}.(x \pm_b y) = \mathsf{subtrees}.x \oplus_b \mathsf{subtrees}.y$$

where

$$u \oplus_b v = u \pm_z v \qquad \text{where} \quad z = \mathsf{root}.u \pm_b \mathsf{root}.v$$

and so

$$\mathsf{subtrees} = \llparenthesis \triangle \circ \triangle, \oplus \rrparenthesis$$

As with inits , we have the following theorem.

27. Theorem

$$\mathsf{subtrees} \circ f* \quad = \quad f** \circ \mathsf{subtrees}$$

◇

Proof By unique extension property, using the property that

$$f** \circ \oplus \quad = \quad \oplus \circ f** \parallel f \parallel f**$$

♡

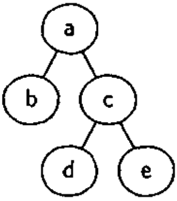
The analogue of a rightwards function on lists is an *upwards* function on trees:

28. Definition Functions of the form $h* \circ \mathsf{subtrees}$ are called *upwards* functions. ◇

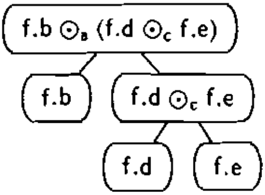
As with rightwards functions on lists, an upwards function can still be neither efficient nor catamorphic. If h is a tree catamorphism, though, then $h* \circ \mathsf{subtrees}$ is much more amenable.

29. Definition Functions of the form $(\!|\mathsf{mtree}: f, \odot|\!)* \circ \mathsf{subtrees}$ are called *upwards accumulations*, and are written $(f, \odot)\Uparrow$. ◇

An upwards accumulation replaces every element of a tree with some catamorphism applied to the descendants of that element. For example, applying the accumulation $(f, \odot)\Uparrow$ to the tree



produces the result



Because the catamorphism $\llparenthesis \triangle, \pm \rrparenthesis$ is the identity, subtrees is itself an upwards accumulation:

$$\mathsf{subtrees} \quad = \quad (\triangle, \pm)\Uparrow$$

Just as we did with left accumulation, we can calculate a catamorphic characterization of upwards accumulation. We observe first that

$$\begin{array}{ll}
 & \mathsf{root} \circ (\mathsf{f}, \odot)\Uparrow \\
= & \qquad [\![\; \Uparrow \;]\!] \\
 & \mathsf{root} \circ \llparenthesis \mathsf{f}, \odot \rrparenthesis * \circ \mathsf{subtrees} \\
= & \qquad [\![\; \mathsf{root} \circ \mathsf{f}* = \mathsf{f} \circ \mathsf{root} \;]\!] \\
 & \llparenthesis \mathsf{f}, \odot \rrparenthesis \circ \mathsf{root} \circ \mathsf{subtrees} \\
= & \qquad [\![\; \mathsf{root} \circ \mathsf{subtrees} = \mathsf{id} \;]\!] \\
 & \llparenthesis \mathsf{f}, \odot \rrparenthesis
\end{array}$$

Now let

$$\begin{array}{lllll}
\mathsf{r} & = & \llparenthesis \mathsf{f}, \odot \rrparenthesis.\mathsf{x} & = & \mathsf{root}.((\mathsf{f}, \odot) \Uparrow \mathsf{x}) \\
\mathsf{s} & = & \llparenthesis \mathsf{f}, \odot \rrparenthesis.\mathsf{y} & = & \mathsf{root}.((\mathsf{f}, \odot) \Uparrow \mathsf{y})
\end{array}$$

and so $\llparenthesis \mathsf{f}, \odot \rrparenthesis.(\mathsf{x} \pm_b \mathsf{y}) = \mathsf{r} \odot_b \mathsf{s}$. Then

$$\begin{array}{ll}
 & (\mathsf{f}, \odot) \Uparrow (\mathsf{x} \pm_b \mathsf{y}) \\
= & \qquad [\![\; \Uparrow \;]\!] \\
 & \llparenthesis \mathsf{f}, \odot \rrparenthesis * \mathsf{subtrees}.(\mathsf{x} \pm_b \mathsf{y})
\end{array}$$

$$
\begin{array}{ll}
= & \llbracket \text{ subtrees } \rrbracket \\
& (\!| f, \odot |\!) * (\text{subtrees}.x \pm_{x \pm_b y} \text{subtrees}.y) \\
= & \llbracket \ * \ \rrbracket \\
& ((\!| f, \odot |\!) * \text{subtrees}.x) \pm_{r \odot_b s} ((\!| f, \odot |\!) * \text{subtrees}.y) \\
= & \llbracket \ \Uparrow \ \rrbracket \\
& ((f, \odot) \Uparrow x) \pm_{r \odot_b s} ((f, \odot) \Uparrow y) \\
= & \llbracket \text{ let } \otimes \text{ satisfy } u \otimes_b v = \langle \pm \rangle.(\text{root}\cdot u \odot_b \text{root}\cdot v).(u, v) \rrbracket \\
& ((f, \odot) \Uparrow x) \otimes_b ((f, \odot) \Uparrow y)
\end{array}
$$

and so upwards accumulation is indeed a catamorphism:

$$(f, \odot)\Uparrow \quad = \quad (\!| \triangle \circ f, \otimes |\!)$$

Once more, we notice from this characterization that the upwards accumulation of $x \pm_b y$ can be computed from the accumulations of x and y using only *one* more application of $\odot$, giving us a linear 'algorithm' $(\!| \triangle \circ f, \otimes |\!)$ in place of the quadratic 'specification' $(\!| f, \odot |\!)* \circ \text{subtrees}$. In fact, if we have one processor per element of the tree, acting in parallel and connected in the same topology as the tree, then we can perform an upwards accumulation—as we can any tree catamorphism—in time proportional to the depth of the tree, assuming that the individual operations take constant time.

Examples of upwards accumulation

One example of an upwards accumulation is the function sizes, which replaces every element of a tree with the number of descendants it has:

$$
\begin{array}{ll}
& \text{sizes} \\
= & \llbracket \text{ definition } \rrbracket \\
& \text{elements}* \circ \text{subtrees}
\end{array}
$$

$$
\begin{array}{ll}
= & \quad [\![\text{ letting } \oplus \text{ satisfy } \mathsf{u} \oplus_{\mathsf{a}} \mathsf{v} = \mathsf{u} + 1 + \mathsf{v}]\!] \\
& \llparenthesis !1, \oplus \rrparenthesis * \circ \mathsf{subtrees} \\
= & \quad [\![\Uparrow]\!] \\
& (!1, \oplus) \Uparrow
\end{array}
$$

A variation on this function forms the first half of the logarithmic-time parallel list ranking algorithm that we will see in Chapter 4.

Another example would be that of the function minmax, which labels every element of a tree with the smallest and largest elements of the subtree rooted at that element:

$$\mathsf{minmax} \quad = \quad (\llparenthesis \mathsf{id}, \downarrow \rrparenthesis \curlywedge \llparenthesis \mathsf{id}, \uparrow \rrparenthesis) * \circ \mathsf{subtrees}$$

By Theorem 13, the fork of two catamorphisms is itself a catamorphism, and so minmax is also an upwards accumulation. When applied to a leaf-labelled binary search tree, that is, one for which an inorder traversal produces a sorted list of leaves, this function 'annotates' the tree with the information needed to enable the fast operations for which binary search trees are useful.

We give some more substantial examples of upwards accumulations in later chapters. In Chapter 5 we see that upwards accumulation forms the first half of the two-pass 'parallel prefix' algorithm. This algorithm passes information up towards the root of the tree and then back down to the leaves again—it is, in fact, a generalization of the list ranking algorithm that we mentioned in connection with sizes above. In Chapter 6 we discuss the problem of drawing a tree, which is also an upwards accumulation; the problem consists of labelling each branch of the tree with information about where to draw its children, and the label attached to an element is a catamorphism of the subtree rooted at that element. Finally, in Chapter 7, we see that there is a very close analogy between upwards accumulation and synthesized attributes in an attribute grammar.

Upwards accumulations on hip and rose trees

We do not need upwards accumulation on hip trees for this thesis, but it can be defined in much the same way as on moo trees. It is given by

$$(f, \oplus, \otimes)\Uparrow \;=\; \llparenthesis \mathsf{htree}\colon f, \oplus, \otimes \rrparenthesis * \circ \mathsf{subtrees}$$

where the function $\mathsf{subtrees}$ on hip trees—again, we make no apologies for reusing the name—satisfies

$$\begin{array}{rcl} \mathsf{subtrees}\cdot\triangle\cdot a & = & \triangle\cdot\triangle\cdot a \\ \mathsf{subtrees}\cdot(t \swarrow u) & = & t \oslash (\mathsf{subtrees}\cdot t \swarrow \mathsf{subtrees}\cdot u) \\ \mathsf{subtrees}\cdot(t \searrow u) & = & (\mathsf{subtrees}\cdot t \searrow \mathsf{subtrees}\cdot u) \obslash u \end{array}$$

where

$$\begin{array}{rclcrcl} t \oslash \triangle\cdot u & = & \triangle\cdot(t \swarrow u) & \quad & \triangle\cdot u \obslash t & = & \triangle\cdot(u \searrow t) \\ t \oslash (x \swarrow y) & = & x \swarrow (t \oslash y) & & (x \swarrow y) \obslash t & = & x \swarrow (y \obslash t) \\ t \oslash (x \searrow y) & = & (t \oslash x) \searrow y & & (x \searrow y) \obslash t & = & (x \obslash t) \searrow y \end{array}$$

We should check the consistency of these equations from first principles, since we have not phrased them as catamorphisms; we have that

$$\begin{array}{rclcl} t \oslash ((x \swarrow y) \searrow z) & = & x \swarrow (t \oslash y) \searrow z & = & t \oslash (x \swarrow (y \searrow z)) \\ ((x \swarrow y) \searrow z) \obslash u & = & x \swarrow (y \obslash u) \searrow z & = & (x \swarrow (y \searrow z)) \obslash u \end{array}$$

and so $\langle t\oslash\rangle$ and $\langle\obslash u\rangle$ are proper; they both affect only the root of a tree. Moreover, $\oslash$ associates with $\obslash$, and so

$$\mathsf{subtrees}\cdot(x \swarrow y \searrow z) \;=\; \mathsf{subtrees}\cdot x \swarrow (x \oslash \mathsf{subtrees}\cdot y \obslash z) \searrow \mathsf{subtrees}\cdot z$$

and so $\mathsf{subtrees}$ is itself proper. What is more, all three of these functions, $\mathsf{subtrees}$, $\langle t\oslash\rangle$ and $\langle\obslash u\rangle$, are injective and hence catamorphic.

Upwards accumulation on rose trees is a lot more straightforward than it is on hip trees, on account of the absence of laws on the algebra of rose trees. Again, the accumulation is just a catamorphism mapped over the subtrees,

$$(\mathsf{f}, \odot)\Uparrow \quad = \quad (\!|\mathsf{rtree}: \mathsf{f}, \odot|\!) * \circ \mathsf{subtrees}$$

where subtrees satisfies

$$\begin{aligned}\mathsf{subtrees}\cdot\triangle\cdot\mathsf{a} &= \triangle\cdot\triangle\cdot\mathsf{a}\\ \mathsf{subtrees}\cdot(\mathsf{a} \prec \mathsf{x}) &= (\mathsf{a} \prec \mathsf{x}) \prec (\mathsf{subtrees} * \mathsf{x})\end{aligned}$$

Once more, subtrees is injective:

$$\mathsf{root} \circ \mathsf{subtrees} \quad = \quad \mathsf{id}$$

and so is catamorphic:

$$\mathsf{subtrees} \quad = \quad (\!|\mathsf{rtree}: \triangle \circ \triangle, \oplus|\!) \qquad \text{where} \quad \mathsf{a} \oplus \mathsf{z} = (\mathsf{a} \prec \mathsf{root} * \mathsf{z}) \prec \mathsf{z}$$

—note that $\oslash_{\oplus}$ associates with $\obslash_{\otimes}$ for any $\oplus$ and $\otimes$—then we can define paths as follows.

30. Definition

$$\mathsf{paths} \quad = \quad \llparenthesis \mathsf{htree}: \triangle \circ \triangle, \oslash_{\swarrow}, \obslash_{\searrow} \rrparenthesis$$

◇

Thus,

$$\begin{aligned} \mathsf{paths}\cdot\triangle\cdot a &= \triangle\cdot\triangle\cdot a \\ \mathsf{paths}\cdot(t \swarrow u) &= \mathsf{paths}\cdot t \oslash_{\swarrow} \mathsf{paths}\cdot u \\ \mathsf{paths}\cdot(t \searrow u) &= \mathsf{paths}\cdot t \obslash_{\searrow} \mathsf{paths}\cdot u \end{aligned}$$

and in particular, we have

$$\mathsf{paths}\cdot(t \swarrow \triangle\cdot a \searrow u) \quad = \quad ((\swarrow\triangle\cdot a) * \mathsf{paths}\cdot t) \swarrow \triangle\cdot\triangle\cdot a \searrow ((\triangle\cdot a\searrow) * \mathsf{paths}\cdot u)$$

As before, paths is a natural transformation.

31. Theorem

$$\mathsf{paths} \circ f* \quad = \quad f** \circ \mathsf{paths}$$

◇

Proof By the unique extension property, using the fact that $\llparenthesis f, \oplus, \otimes \rrparenthesis *$ is $(\oslash_{\swarrow}, \oslash_{\oplus})$ and $(\obslash_{\searrow}, \obslash_{\otimes})$ promotable. ♡

Proceeding in the same way that we did in the previous chapter, we define *downwards* functions and accumulations.

32. Definition Functions of the form $h* \circ \mathsf{paths}$ are called *downwards functions*. ◇

33. Definition Functions of the form $\llparenthesis \mathsf{htree}: f, \oplus, \otimes \rrparenthesis * \circ \mathsf{paths}$ are called *downwards accumulations*, and are written $(f, \oplus, \otimes)\Downarrow$. ◇

The promotion properties used in the proof of Theorem 31 give us immediately that

$$\begin{array}{rcl}
(\mathsf{f}, \oplus, \otimes) \Downarrow \triangle \cdot \mathsf{a} & = & \triangle \cdot \mathsf{f} \cdot \mathsf{a} \\
(\mathsf{f}, \oplus, \otimes) \Downarrow (\mathsf{t} \swarrow \mathsf{u}) & = & (\mathsf{f}, \oplus, \otimes) \Downarrow \mathsf{t} \oslash_{\oplus} (\mathsf{f}, \oplus, \otimes) \Downarrow \mathsf{u} \\
(\mathsf{f}, \oplus, \otimes) \Downarrow (\mathsf{t} \searrow \mathsf{u}) & = & (\mathsf{f}, \oplus, \otimes) \Downarrow \mathsf{t} \obslash_{\otimes} (\mathsf{f}, \oplus, \otimes) \Downarrow \mathsf{u}
\end{array}$$

and so downwards accumulation is catamorphic:

$$(\mathsf{f}, \oplus, \otimes)\Downarrow \quad = \quad \llparenthesis \triangle \circ \mathsf{f}, \oslash_{\oplus}, \obslash_{\otimes} \rrparenthesis$$

However, something unexpected happens here. Consider the identity function; it is easy to see that $\oslash_{\ll} = \swarrow$ and $\obslash_{\gg} = \searrow$, so

$$\begin{array}{rcl}
\mathsf{id} \cdot \triangle \cdot \mathsf{a} & = & \triangle \cdot \mathsf{id} \cdot \mathsf{a} \\
\mathsf{id} \cdot (\mathsf{t} \swarrow \mathsf{u}) & = & \mathsf{id} \cdot \mathsf{t} \oslash_{\ll} \mathsf{id} \cdot \mathsf{u} \\
\mathsf{id} \cdot (\mathsf{t} \searrow \mathsf{u}) & = & \mathsf{id} \cdot \mathsf{t} \obslash_{\gg} \mathsf{id} \cdot \mathsf{u}
\end{array}$$

and so, by the unique extension property, we would expect that

$$\mathsf{id} \quad = \quad (\mathsf{id}, \ll, \gg)\Downarrow \quad = \quad \llparenthesis \mathsf{id}, \ll, \gg \rrparenthesis * \circ \mathsf{paths}$$

—but we noted in Chapter 2 that $\llparenthesis \mathsf{id}, \ll, \gg \rrparenthesis$ is not a hip tree catamorphism, because $\ll$ does not associate with $\gg$. The associativity property imposes conditions on all tree catamorphisms, even though not all hip trees can be paths; no element of any path of a tree has a sibling, and so the associativity property never comes into play for downwards accumulations.

We show next how to prevent these extra conditions on hip trees from interfering with downwards accumulations.

Threads

The solution to the dilemma mentioned above is to coin a new algebra, less general and more appropriate than hip trees, to represent paths; we call terms in this algebra *threads*. Every thread can be a path in some tree.

34. Definition

$$\mathsf{thread.A} \quad = \quad \diamond.\mathsf{A} \mid \mathsf{A} ↵ \mathsf{thread.A} \mid \mathsf{A} ↳ \mathsf{thread.A}$$

◇

The operators $↵$ and $↳$ might be pronounced 'left child' and 'right child'. Informally, threads are like hip trees in which every child is an only child; alternatively, they could be seen as a kind of non-empty cons list with two different cons operators. For example, the three-element path from page 76 corresponds to the thread $\mathsf{a} ↳ (\mathsf{c} ↵ \diamond.\mathsf{d})$.

With this new algebra, paths has type

$$\mathsf{paths} \quad \in \quad \mathsf{htree.A} \rightarrow \mathsf{htree{\cdot}thread{\cdot}A}$$

and is given by the next definition.

35. Definition (replacing Definition 30)

$$\mathsf{paths} \quad = \quad (\!| \triangle \circ \triangle, \oslash_{\boxslash}, \obslash_{\boxbslash} |\!) \qquad \text{where} \quad \begin{array}{rcl} \mathsf{x} \boxslash \diamond{\cdot}\mathsf{a} & = & \mathsf{a} ↵ \mathsf{x} \\ \diamond{\cdot}\mathsf{a} \boxbslash \mathsf{x} & = & \mathsf{a} ↳ \mathsf{x} \end{array}$$

◇

More lucidly,

$$\mathsf{paths}{\cdot}(\mathsf{t} \swarrow \triangle{\cdot}\mathsf{a} \searrow \mathsf{u}) \quad = \quad (\langle \mathsf{a}↵ \rangle * \mathsf{paths}{\cdot}\mathsf{t}) \swarrow \triangle{\cdot}\diamond{\cdot}\mathsf{a} \searrow (\langle \mathsf{a}↳ \rangle * \mathsf{paths}{\cdot}\mathsf{u})$$

The downwards functions remain the same as before: if we call the function in Definition 30 paths_0, in order to distinguish it from the new definition of paths in Definition 35, then

$$\mathsf{h}* \circ \mathsf{paths}_0 \quad = \quad (\mathsf{h} \circ \mathsf{th})* \circ \mathsf{paths}$$

where th is the injective but not surjective function converting threads to hip trees,

$$\mathsf{th} \quad = \quad (\!| \mathsf{thread}: \triangle, \bar{\swarrow} \circ \triangle \parallel \mathsf{id}, \searrow \circ \triangle \parallel \mathsf{id} |\!)$$

and

$$\mathsf{h}{*} \circ \mathsf{paths} \quad = \quad (\mathsf{h} \circ \mathsf{ht}){*} \circ \mathsf{paths}_0$$

where $\mathsf{ht} \in \mathsf{htree.A} \rightarrow \mathsf{thread.A}$ is the post-inverse of th.

The downwards accumulations, however, are different with this new definition of paths: a downwards accumulation is now a *thread* catamorphism mapped over the paths of a tree.

36. Definition (replacing Definition 33)

$$(\mathsf{f}, \oplus, \otimes)\Downarrow \quad = \quad \llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrparenthesis{*} \circ \mathsf{paths}$$

◇

Because threads form a free algebra, there are no laws to impose on the components of an accumulation, and the identity function is a downwards accumulation after all:

$$\mathsf{id} \quad = \quad (\mathsf{id}, \gg, \gg)\Downarrow$$

Now, unfortunately, something else goes wrong: $(\mathsf{f}, \oplus, \otimes)\Downarrow$ is no longer a catamorphism, because it promotes through neither $\oslash_{\boxslash}$ nor $\obslash_{\boxbslash}$. This can be seen for $\oslash_{\boxslash}$ by looking at the tree $\triangle\cdot\mathsf{a} \swarrow \triangle\cdot\mathsf{b}$:

$$\begin{array}{ll}
 & (\mathsf{f}, \oplus, \otimes) \Downarrow (\triangle\cdot\mathsf{a} \swarrow \triangle\cdot\mathsf{b}) \\
= & \qquad [\![\; \Downarrow \;]\!] \\
 & \llparenthesis \mathsf{f}, \oplus, \otimes \rrparenthesis * \mathsf{paths}\cdot(\triangle\cdot\mathsf{a} \swarrow \triangle\cdot\mathsf{b}) \\
= & \qquad [\![\; \mathsf{paths} \;]\!] \\
 & \llparenthesis \mathsf{f}, \oplus, \otimes \rrparenthesis * (\triangle\cdot(\mathsf{b} \hookleftarrow \circ\cdot\mathsf{a}) \swarrow \triangle\cdot\circ\cdot\mathsf{b}) \\
= & \qquad [\![\; *, \text{catamorphisms} \;]\!] \\
 & \triangle\cdot(\mathsf{b} \oplus \mathsf{f}\cdot\mathsf{a}) \swarrow \triangle\cdot\mathsf{f}\cdot\mathsf{b}
\end{array}$$

The accumulation depends on the root b of the tree, as well as the accumulations $\triangle\cdot\mathsf{f}\cdot\mathsf{a}$ and $\triangle\cdot\mathsf{f}\cdot\mathsf{b}$ of the components.

This can be rectified by defining paths and downwards accumulation on homogeneous moo trees instead of on hip trees; intuitively, the accumulations of the children happen 'at the same time', when the root is still available. Formally, we redefine paths once more to have type

$$\mathsf{paths} \quad \in \quad \mathsf{hmtree.A} \rightarrow \mathsf{hmtree{\cdot}thread{\cdot}A}$$

as follows.

37. Definition (replacing Definition 35)

$$\begin{aligned} \mathsf{paths} \quad &= \quad \llparenthesis \mathsf{hmtree}\colon \triangle \circ \diamond, \oplus \rrparenthesis \\ &\qquad \text{where} \quad u \oplus_a v = \langle a_{\hookleftarrow} \rangle * u \;\, \underline{\ast}_{\diamond \cdot a} \;\, \langle a_{\hookrightarrow} \rangle * v \end{aligned}$$

$\diamond$

It is still a natural transformation, and so satisfies Theorem 31.

With this definition, the downwards functions are now moo tree functions of the form

$$h* \circ \mathsf{paths} \quad \in \quad \mathsf{hmtree.A} \rightarrow \mathsf{hmtree.B}$$

Every downwards function in this sense corresponds to one in the sense of Definition 32, but the converse is not true because the correspondence between hmtree and htree is not surjective. The downwards accumulation

$$(f, \oplus, \otimes)\Downarrow \quad = \quad \llparenthesis \mathsf{thread}\colon f, \oplus, \otimes \rrparenthesis * \circ\, \mathsf{paths}$$

regains its catamorphic status, being now a catamorphism on moo trees:

$$\begin{aligned} (f, \oplus, \otimes) \Downarrow \triangle{\cdot}a \quad &= \quad \triangle{\cdot}f{\cdot}a \\ (f, \oplus, \otimes) \Downarrow (x \;\underline{\ast}_a\; y) \quad &= \quad (\langle a\oplus \rangle * (f, \oplus, \otimes) \Downarrow x) \;\underline{\ast}_{f \cdot a}\; (\langle a\otimes \rangle * (f, \oplus, \otimes) \Downarrow y) \end{aligned}$$

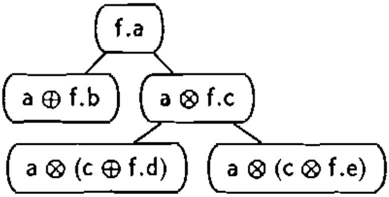
so

$$(f, \oplus, \otimes)\Downarrow \quad = \quad \llparenthesis \mathsf{hmtree}\colon \triangle \circ f, \circledast \rrparenthesis$$

where

$$u \circledast_a v \quad = \quad (\langle a\oplus \rangle * u) \;\, \underline{\ast}_{f \cdot a} \;\, (\langle a\otimes \rangle * v)$$

Applying the accumulation $(f, \oplus, \otimes){\Downarrow}$ to the five-element tree given earlier produces the tree



as a result.

Many useful functions on trees are downwards accumulations. We have already seen two examples, id and paths; another simple yet important accumulation is the function depths, which replaces every element of a tree with its depth in that tree:

$$\mathsf{depths} \quad = \quad (!1, \oplus, \oplus){\Downarrow} \qquad \text{where} \quad a \oplus x = 1 + x$$

The thread catamorphism $\llparenthesis !1, \oplus, \oplus \rrparenthesis$ involved here returns the length of a thread. We can write the weighted internal path length wipl of a tree in terms of depths:

$$\mathsf{wipl} \quad = \quad \llparenthesis \mathsf{id}, \oplus \rrparenthesis \circ \mathsf{depths} \qquad \text{where} \quad u \oplus_a v = u + a + v$$

A more substantial example is provided by backwards analysis of expressions (Hughes, 1990), which 'starts with information about the context of the entire expression and propagates it downwards through the syntax tree to the leaves to derive information about the contexts in which the subexpressions occur'—that is, it consists of a downwards accumulation applied to the parse tree of the expression.

As a final example, consider the function leftleaves, a variation on the function sizes of Chapter 3; this function replaces every branch of a leaf-labelled binary tree with the number of leaves in its left child, and replaces every leaf with 1:

$$\mathsf{leftleaves} \quad = \quad (\mathsf{leaves} \circ \mathsf{left}){*} \circ \mathsf{subtrees} \quad \in \quad \mathsf{lmtree.A} \longrightarrow \mathsf{hmtree.N}$$

where left satisfies

$$\begin{aligned} \mathsf{left}\cdot\triangle\cdot\mathsf{a} &= \triangle\cdot\mathsf{a} \\ \mathsf{left}\cdot(\mathsf{x} \pm \mathsf{y}) &= \mathsf{x} \end{aligned}$$

Now define the function rank by

$$\mathsf{rank} \quad = \quad (\mathsf{id}, !\mathsf{it}){*} \circ (\mathsf{id}, \gg, +){\Downarrow} \circ \mathsf{leftleaves} \quad \in \quad \mathsf{lmtree.A} \longrightarrow \mathsf{lmtree.N}$$

The function $(\mathsf{id}, !\mathsf{it}){*}$ turns a general binary tree into a leaf-labelled binary tree by throwing away the branch labels. A little manipulation shows that

$$\begin{aligned} \mathsf{rank}\cdot\triangle\cdot\mathsf{a} &= \triangle\cdot 1 \\ \mathsf{rank}\cdot(\mathsf{x} \pm \mathsf{y}) &= \mathsf{rank}\cdot\mathsf{x} \pm \langle \mathsf{leaves.x}{+} \rangle * \mathsf{rank}\cdot\mathsf{y} \end{aligned}$$

and so rank numbers the leaves of a tree from left to right.

Now, leftleaves is upwards, but it is not an upwards accumulation because $\mathsf{leaves} \circ \mathsf{left}$ is not catamorphic. Tupling produces the catamorphism $\mathsf{leaves} \curlywedge (\mathsf{leaves} \circ \mathsf{left})$:

$$\mathsf{leaves} \curlywedge (\mathsf{leaves} \circ \mathsf{left}) \quad = \quad \llparenthesis !(1,1), (+ \curlywedge \ll) \circ \ll^2 \rrparenthesis$$

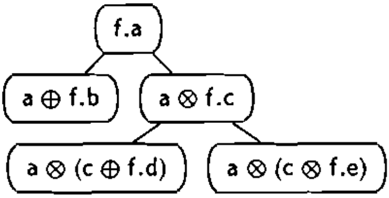
from which we can construct an upwards accumulation:

$$\begin{aligned} & \mathsf{leftleaves} \\ = \quad & \llbracket \text{ definition } \rrbracket \\ & (\mathsf{leaves} \circ \mathsf{left}){*} \circ \mathsf{subtrees} \\ = \quad & \llbracket \text{ pair calculus } \rrbracket \\ & \gg{*} \circ (\mathsf{leaves} \curlywedge (\mathsf{leaves} \circ \mathsf{left})){*} \circ \mathsf{subtrees} \\ = \quad & \llbracket \text{ tupling, upwards accumulation } \rrbracket \\ & \gg{*} \circ (!(1,1), (+ \curlywedge \ll) \circ \ll^2){\Uparrow} \end{aligned}$$

This gives rank as the composition of a map, a downwards accumulation, another map and an upwards accumulation. All but the downwards accumulation can be evaluated efficiently—with linear effort, and in parallel in time proportional to the depth of the tree. The downwards accumulation, though, is still quadratic. In the next section, we will see how a downwards accumulation can be evaluated efficiently—in particular, rank can be computed in parallel on a linear number of processors in time proportional to the depth of the tree. Thus, rank can be used as a logarithmic-time parallel function to label the elements of a list from left to right, by first building a balanced binary tree whose leaves are the elements of the list.

Efficient downwards accumulations

Look again at the result of applying the accumulation $(f, \oplus, \otimes)\Downarrow$ to the five-element tree on page 75:



This tree contains a quadratic number of applications of $\oplus$ and $\otimes$, and there are no common subexpressions; evaluating the accumulation will inevitably take quadratic effort.

We might ask, under what conditions can the accumulation be evaluated using only a linear number of $\oplus$ and $\otimes$ applications? That is, what properties are required of f, $\oplus$ and $\otimes$ in order that each parent in the result is a common subexpression of both its children?

Considering the two elements on the middle level of the tree we see that a necessary condition is that there exist operators $\boxplus$ and $\boxtimes$ such that

$$\begin{array}{rcl} a \oplus f.b & = & f.a \boxplus b \\ a \otimes f.c & = & f.a \boxtimes c \end{array}$$

for only then is the root of the tree a common subexpression of its two children. Given this property, a sufficient condition for the two elements on the bottom layer to have their parent as a common subexpression is that $\oplus$ and $\otimes$ each associate with both $\boxplus$ and $\boxtimes$ (that is, four associativity properties, not two); if this is the case then, for example,

$$\begin{array}{ll} & a \otimes (c \oplus f.d) \\ = & \qquad \llbracket\ \boxplus\ \rrbracket \\ & a \otimes (f.c \boxplus d) \\ = & \qquad \llbracket\ \text{associativity}\ \rrbracket \\ & (a \otimes f.c) \boxplus d \\ = & \qquad \llbracket\ \boxtimes\ \rrbracket \\ & (f.a \boxtimes c) \boxplus d \end{array}$$

which has $f.a \boxtimes c$, its parent, as a subexpression.

Let us give these various properties a simple name, so we can refer to them.

38. Definition Say $(f, \oplus, \otimes)$ *inverts to* $(f, \boxplus, \boxtimes)$ if

$$\begin{array}{rcl} a \oplus f.b & = & f.a \boxplus b \\ a \otimes f.c & = & f.a \boxtimes c \end{array}$$

and $\oplus$ and $\otimes$ each associate with both $\boxplus$ and $\boxtimes$. ◇

39. Definition Say the triple $(f, \oplus, \otimes)$ is *top-down* if there exist $\boxplus$ and $\boxtimes$ such that $(f, \oplus, \otimes)$ inverts to $(f, \boxplus, \boxtimes)$. ◇

These definitions give properties for threads analogous to those for lists under which a cons list catamorphism can be written as a snoc list catamorphism.

We sometimes extend these concepts in the natural way from components to catamorphisms—that is, we sometimes say 'the catamorphism $\llparenthesis \mathsf{f}, \oplus, \otimes \rrparenthesis$ is top-down' instead of the rather long-winded 'the components $(\mathsf{f}, \oplus, \otimes)$ of the catamorphism $\llparenthesis \mathsf{f}, \oplus, \otimes \rrparenthesis$ are top-down', and similarly for inversion.

As we shall see, if $(\mathsf{f}, \oplus, \otimes)$ is top-down then the accumulation $(\mathsf{f}, \oplus, \otimes)\Downarrow$ can be evaluated with only linear effort; moreover, it can be evaluated in parallel in time proportional to the depth of the tree on a number of processors linear in the number of elements of the tree.

Daerhts

Consider the type daerht, pronounced 'dirt' and defined as follows.

40. Definition

$$\mathsf{daerht.a} \quad = \quad \diamond.\mathsf{A} \mid \mathsf{daerht.A} \text{ ⬐ } \mathsf{A} \mid \mathsf{daerht.A} \text{ ⬎ } \mathsf{A}$$

◇

The two operators ⬐ and ⬎ could be pronounced 'left leaf' and 'right leaf', respectively. Informally, daerhts are 'inverted' threads; daerhts are constructed top down, whereas threads are constructed bottom up. Put another way, daerhts are to threads as snoc lists are to cons lists.

The correspondence between daerhts and threads is made formal by the function td of type $\mathsf{thread.A} \rightarrow \mathsf{daerht.A}$, which converts a thread to the corresponding daerht; this function is invertible (threads and daerhts are isomorphic), and so it is a catamorphism.

$$\begin{aligned} \mathsf{td} \quad = \quad \llparenthesis \mathsf{thread}\colon \diamond, \oslash, \text{⦸} \rrparenthesis \qquad \text{where} \quad \mathsf{a}\oslash &= \llparenthesis \mathsf{daerht}\colon \diamond.\mathsf{a}\text{⬐}, \text{⬐}, \text{⬎} \rrparenthesis \\ \mathsf{a}\text{⦸} &= \llparenthesis \mathsf{daerht}\colon \diamond.\mathsf{a}\text{⬎}, \text{⬐}, \text{⬎} \rrparenthesis \end{aligned}$$

For example, considering again the path from page 76,

$$\mathrm{td}.(a \mathbin{↳} (c \mathbin{↵} \diamond.d)) \quad = \quad (\diamond.a \mathbin{↴} c) \mathbin{⬐} d$$

Now, thread catamorphisms that are top-down are also daerht catamorphisms, modulo type conversion:

41. Theorem If $(f, \oplus, \otimes)$ inverts to $(f, \boxplus, \boxtimes)$, then

$$\llparenthesis \mathrm{thread}\colon f, \oplus, \otimes \rrparenthesis \quad = \quad \llparenthesis \mathrm{daerht}\colon f, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td}$$

◇

A significant proportion of the proof of this theorem will be used later, so we extract it as a lemma.

42. Lemma

$$\begin{aligned}
\llparenthesis f, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td} \circ \langle a{↵} \rangle \quad &= \quad \llparenthesis f.a\boxplus, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td} \\
\llparenthesis f, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td} \circ \langle a{↳} \rangle \quad &= \quad \llparenthesis f.a\boxtimes, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td}
\end{aligned}$$

◇

Proof We prove only the first part; the second is symmetric.

$$\begin{aligned}
& \llparenthesis f, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td} \circ \langle a{↵} \rangle \\
= \quad & \qquad \llbracket \ \mathrm{td} \ \rrbracket \\
& \llparenthesis f, \boxplus, \boxtimes \rrparenthesis \circ \langle a\oslash \rangle \circ \mathrm{td} \\
= \quad & \qquad \llbracket \ \oslash \ \rrbracket \\
& \llparenthesis f, \boxplus, \boxtimes \rrparenthesis \circ \llparenthesis \diamond.a{⬐}, {⬐}, {↴} \rrparenthesis \circ \mathrm{td} \\
= \quad & \qquad \llbracket \ \text{promotion} \ \rrbracket \\
& \llparenthesis \llparenthesis f, \boxplus, \boxtimes \rrparenthesis \circ \langle \diamond.a{⬐} \rangle, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td} \\
= \quad & \qquad \llbracket \ \text{catamorphisms} \ \rrbracket \\
& \llparenthesis \langle f.a\boxplus \rangle, \boxplus, \boxtimes \rrparenthesis \circ \mathrm{td}
\end{aligned}$$

♡

Proof (of Theorem 41) The proof follows directly from the unique extension property. For singleton threads we have

$$\begin{array}{ll}
 & \llparenthesis \mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td} \circ \diamond \\
= & \qquad [\![\ \mathsf{td} \ ]\!] \\
 & \llparenthesis \mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes \rrparenthesis \circ \diamond \\
= & \qquad [\![\ \text{catamorphisms} \ ]\!] \\
 & \mathsf{f} \\
= & \qquad [\![\ \text{catamorphisms} \ ]\!] \\
 & \llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrparenthesis \circ \diamond
\end{array}$$

For longer threads, we only show the case for $\hookleftarrow$; the argument for $\hookrightarrow$ is symmetric. We have as premise that $(\mathsf{f}, \oplus, \otimes)$ inverts to $(\mathsf{f}, \boxplus, \boxtimes)$.

$$\begin{array}{ll}
 & \llparenthesis \mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td} \circ \langle \mathsf{a}\hookleftarrow \rangle \\
= & \qquad [\![\ \text{Lemma 42} \ ]\!] \\
 & \llparenthesis \mathsf{daerht}\colon \langle \mathsf{f.a}\boxplus \rangle, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td} \\
= & \qquad [\![\ \text{premise: } \langle \mathsf{f.a}\boxplus \rangle = \langle \mathsf{a}\oplus \rangle \circ \mathsf{f} \ ]\!] \\
 & \llparenthesis \mathsf{daerht}\colon \langle \mathsf{a}\oplus \rangle \circ \mathsf{f}, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td} \\
= & \qquad [\![\ \text{premise: } \oplus \text{ associates with } \boxplus \text{ and } \boxtimes\text{; promotion} \ ]\!] \\
 & \langle \mathsf{a}\oplus \rangle \circ \llparenthesis \mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td}
\end{array}$$

and of course, $\llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrparenthesis$ follows the same recursive pattern:

$$\llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrparenthesis \circ \langle \mathsf{a}\hookleftarrow \rangle \quad = \quad \langle \mathsf{a}\oplus \rangle \circ \llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrparenthesis$$

Invoking the unique extension property completes the proof. ♡

Thus, any top-down downwards accumulation can be expressed in terms of daerht catamorphisms:

43. Corollary If $(\mathsf{f}, \oplus, \otimes)$ inverts to $(\mathsf{f}, \boxplus, \boxtimes)$ then

$$(\mathsf{f}, \oplus, \otimes)\Downarrow \quad = \quad \llparenthesis \mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes \rrparenthesis * \circ \mathsf{td} * \circ \mathsf{paths}$$

◇

Proof

$$
\begin{aligned}
& (\mathsf{f}, \oplus, \otimes) \Downarrow \\
= \quad & \qquad [\![\ \Downarrow \ ]\!] \\
& (\!|\mathsf{thread} : \mathsf{f}, \oplus, \otimes|\!) * \circ \mathsf{paths} \\
= \quad & \qquad [\![\ \text{Theorem 41} \ ]\!] \\
& (\!|\mathsf{daerht} : \mathsf{f}, \boxplus, \boxtimes|\!) * \circ \mathsf{td} * \circ \mathsf{paths}
\end{aligned}
$$

♡

Let us define the function htaps, pronounced 'taps', to return the paths of a tree as daerhts rather than as threads:

44. Definition

$$\mathsf{htaps} \quad = \quad \mathsf{td} * \circ \mathsf{paths}$$

◇

Again, we have a theorem about htaps promoting through a map.

45. Theorem

$$\mathsf{htaps} \circ \mathsf{f} * \quad = \quad \mathsf{f} ** \circ \mathsf{htaps}$$

◇

Proof

$$
\begin{aligned}
& \mathsf{htaps} \circ \mathsf{f} * \\
= \quad & \qquad [\![\ \mathsf{htaps} \ ]\!] \\
& \mathsf{td} * \circ \mathsf{paths} \circ \mathsf{f} * \\
= \quad & \qquad [\![\ \text{Theorem 31} \ ]\!] \\
& \mathsf{td} * \circ \mathsf{f} ** \circ \mathsf{paths} \\
= \quad & \qquad [\![\ \text{promotion: } \mathsf{td} \text{ commutes with } \mathsf{f} * \ ]\!] \\
& \mathsf{f} ** \circ \mathsf{td} * \circ \mathsf{paths}
\end{aligned}
$$

$$= \qquad [\![\ \mathsf{htaps} \ ]\!]$$

$$\mathsf{f}{*}{*} \circ \mathsf{htaps}$$

♡

The downwards functions in terms of htaps are the same as the downwards functions in terms of paths, since

$$\mathsf{h}{*} \circ \mathsf{htaps} \quad = \quad (\mathsf{h} \circ \mathsf{td}){*} \circ \mathsf{paths}$$

and

$$\mathsf{h}{*} \circ \mathsf{paths} \quad = \quad (\mathsf{h} \circ \mathsf{dt}){*} \circ \mathsf{htaps}$$

where dt is the inverse of td. Downwards accumulations in terms of htaps are given by the following definition.

46. Definition

$$(\mathsf{f}, \boxplus, \boxtimes)\text{⅄} \quad = \quad (\!|\mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes|\!){*} \circ \mathsf{htaps}$$

◇

We might call this 'htaps accumulation', to distinguish it from 'paths accumulation'. Almost by definition, paths accumulations that are top-down are htaps accumulations.

47. Theorem If $(\mathsf{f}, \oplus, \otimes)$ inverts to $(\mathsf{f}, \boxplus, \boxtimes)$ then

$$(\mathsf{f}, \oplus, \otimes)\Downarrow \quad = \quad (\mathsf{f}, \boxplus, \boxtimes)\text{⅄}$$

◇

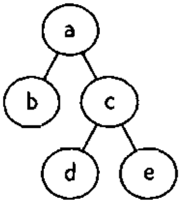
Proof By definition of ⅄, using Corollary 43. ♡

As we would expect—since this was the reason for which we defined it—htaps accumulation can be evaluated with a linear number of applications of its component operators. Let $\mathsf{h} = (\!|\mathsf{daerht}\colon \mathsf{f}, \boxplus, \boxtimes|\!)$; then

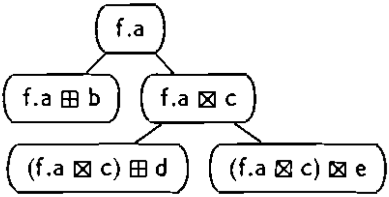
$$
\begin{aligned}
& (\mathsf{f}, \boxplus, \boxtimes)\ \text{⅄}\ (\mathsf{x} \overset{\ast}{-}_{\mathsf{a}} \mathsf{y}) \\
= \quad & \qquad \llbracket\ \text{⅄};\ \mathsf{htaps}\ \rrbracket \\
& \mathsf{h} * \mathsf{td} * \mathsf{paths}.(\mathsf{x} \overset{\ast}{-}_{\mathsf{a}} \mathsf{y}) \\
= \quad & \qquad \llbracket\ \mathsf{paths}\ \rrbracket \\
& \mathsf{h} * \mathsf{td} * (\langle \mathsf{a}_{\hookleftarrow} \rangle * \mathsf{paths}.\mathsf{x}\ \ \overset{\ast}{-}_{\circ,\mathsf{a}}\ \ \langle \mathsf{a}\hookrightarrow \rangle * \mathsf{paths}.\mathsf{y}) \\
= \quad & \qquad \llbracket\ *, \text{td and catamorphisms}\ \rrbracket \\
& (\mathsf{h} \circ \mathsf{td} \circ \langle \mathsf{a}_{\hookleftarrow} \rangle) * \mathsf{paths}.\mathsf{x}\ \ \overset{\ast}{-}_{\mathsf{f},\mathsf{a}}\ \ (\mathsf{h} \circ \mathsf{td} \circ \langle \mathsf{a}\hookrightarrow \rangle) * \mathsf{paths}.\mathsf{y} \\
= \quad & \qquad \llbracket\ \text{Lemma 42}\ \rrbracket \\
& \llparenthesis \mathsf{f}.\mathsf{a}\boxplus, \boxplus, \boxtimes \rrparenthesis * \mathsf{td} * \mathsf{paths}.\mathsf{x}\ \ \overset{\ast}{-}_{\mathsf{f},\mathsf{a}}\ \ \llparenthesis \mathsf{f}.\mathsf{a}\boxtimes, \boxplus, \boxtimes \rrparenthesis * \mathsf{td} * \mathsf{paths}.\mathsf{y} \\
= \quad & \qquad \llbracket\ \text{⅄}\ \rrbracket \\
& (\mathsf{f}.\mathsf{a}\boxplus, \boxplus, \boxtimes)\ \text{⅄}\ \mathsf{x}\ \ \overset{\ast}{-}_{\mathsf{f}.\mathsf{a}}\ \ (\mathsf{f}.\mathsf{a}\boxtimes, \boxplus, \boxtimes)\ \text{⅄}\ \mathsf{y}
\end{aligned}
$$

and the last line contains no expensive maps. Moreover, assuming that $\boxplus$ and $\boxtimes$ take constant time, the accumulation can be evaluated in parallel in time proportional to the depth of the tree, given as many processors as there are leaves.

For example, applying the htaps accumulation $(\mathsf{f}, \boxplus, \boxtimes)\text{⅄}$ to the standard five-element tree



produces the result



which has only a linear number of different subexpressions.

Note that htaps accumulation is not in general catamorphic: the accumulation $(f, \boxplus, \boxtimes)\curlywedge$ of the parent $x \mathbin{\triangle_a} y$ depends on *different* accumulations $(f.a\boxplus, \boxplus, \boxtimes)\curlywedge$ and $(f.a\boxtimes, \boxplus, \boxtimes)\curlywedge$ of its children.

We have already seen one example of htaps accumulation, the function htaps itself:

$$\mathsf{htaps} \quad = \quad (\diamond, \swarrow, \searrow)\curlywedge$$

because the daerht catamorphism $(\!|\diamond, \swarrow, \searrow|\!)$ is the identity. Another example is the identity function:

$$\mathsf{id} \quad = \quad (\mathsf{id}, \gg, \gg)\curlywedge$$

The daerht catamorphism $(\!|\mathsf{id}, \gg, \gg|\!)$ returns the last (bottom) element of a daerht.

Most natural downwards accumulations are both paths- and htaps accumulations. For example, recall the function depths from page 82:

$$\mathsf{depths} \quad = \quad (!1, \oplus, \oplus)\Downarrow \qquad \text{where} \quad a \oplus x = 1 + x$$

Now, $(!1, \oplus, \oplus)$ inverts to $(!1, \tilde{\oplus}, \tilde{\oplus})$—because $a \oplus !1.b = !1.a \mathbin{\tilde{\oplus}} b$ and $\oplus$ associates with $\tilde{\oplus}$—and so

$$\mathsf{depths} \quad = \quad (!1, \tilde{\oplus}, \tilde{\oplus})\curlywedge$$

as well. The evaluation of this latter formulation of depths takes effort linear, rather than quadratic, in the size of the tree.

However, there are natural paths accumulations that are not htaps accumulations, and vice versa. An example of the former is the paths accumulation $(\mathsf{id}, \gg, +)\Downarrow$ that formed part of the definition of rank on page 83; there are no operators $\boxplus$ and $\boxtimes$ for which the functions $\llparenthesis \mathsf{thread}\colon \mathsf{id}, \gg, + \rrparenthesis$ and $\llparenthesis \mathsf{daerht}\colon \mathsf{id}, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td}$ are equal. To see this, define, for brevity, two functions

$$\begin{array}{lcl} \mathsf{f} & = & \llparenthesis \mathsf{thread}\colon \mathsf{id}, \gg, + \rrparenthesis \\ \mathsf{g} & = & \llparenthesis \mathsf{daerht}\colon \mathsf{id}, \boxplus, \boxtimes \rrparenthesis \circ \mathsf{td} \end{array}$$

for some given $\boxplus$ and $\boxtimes$, and consider the three threads of numbers

$$\begin{array}{lcl} \mathsf{x} & = & \mathsf{a} \hookrightarrow \diamond.\mathsf{b} \\ \mathsf{y} & = & \mathsf{a} \hookrightarrow (\mathsf{b} \hookleftarrow \diamond.\mathsf{c}) \\ \mathsf{z} & = & (\mathsf{a} + \mathsf{b}) \hookleftarrow \diamond.\mathsf{c} \end{array}$$

with a being non-zero. We will show that f and g must disagree on at least one of these threads.

We have

$$\begin{array}{lclclcl} \mathsf{f}.\mathsf{x} & = & \mathsf{a} + \mathsf{b} & \quad & \mathsf{g}.\mathsf{x} & = & \mathsf{a} \boxtimes \mathsf{b} \\ \mathsf{f}.\mathsf{y} & = & \mathsf{a} + \mathsf{c} & & \mathsf{g}.\mathsf{y} & = & (\mathsf{a} \boxtimes \mathsf{b}) \boxplus \mathsf{c} \\ \mathsf{f}.\mathsf{z} & = & \mathsf{c} & & \mathsf{g}.\mathsf{z} & = & (\mathsf{a} + \mathsf{b}) \boxplus \mathsf{c} \end{array}$$

Comparing the values on x, we see that $\mathsf{a} \boxtimes \mathsf{b} = \mathsf{a} + \mathsf{b}$, in which case g returns the same values for y and z—but f returns different values. Therefore, for no $\boxplus$ and $\boxtimes$ does g equal f; hence, the paths accumulation $(\mathsf{id}, \gg, +)\Downarrow$ is not a htaps accumulation as well.

When we first introduced the function rank, we promised that we would show how it can be efficiently evaluated; we have just shown, though, that the downwards accumulation component itself is *not* an efficient (htaps) accumulation. The solution is to tuple the downwards accumulation so that it can be inverted to form an easily computed htaps accumulation. We just present the tupling here as an example, but in the next chapter we show how to calculate it.

Consider the daerht catamorphism $\mathsf{h.a}$ defined by

$$\mathsf{h.a} \quad = \quad \llparenthesis \mathsf{daerht}: !\mathsf{a} \curlywedge \mathsf{a}+, \oplus, \otimes \rrparenthesis \qquad \text{where} \quad \begin{array}{rcl} (\mathsf{u},\mathsf{v}) \oplus \mathsf{a} & = & (\mathsf{u}, \mathsf{u}+\mathsf{a}) \\ (\mathsf{u},\mathsf{v}) \otimes \mathsf{a} & = & (\mathsf{v}, \mathsf{v}+\mathsf{a}) \end{array}$$

It is easy to show that

$$\begin{array}{rcl} \gg \circ\, \mathsf{h.0} \circ \mathsf{td} \circ \diamond & = & \mathsf{id} \\ \gg \circ\, \mathsf{h.0} \circ \mathsf{td} \circ \langle \mathsf{a}\hookleftarrow \rangle & = & \gg \circ\, \mathsf{h.0} \circ \mathsf{td} \\ \gg \circ\, \mathsf{h.0} \circ \mathsf{td} \circ \langle \mathsf{a}\hookrightarrow \rangle & = & \langle \mathsf{a}+ \rangle \circ \gg \circ\, \mathsf{h.0} \circ \mathsf{td} \end{array}$$

(for example, by first showing that $\mathsf{h.a} = (\mathsf{a}+ \parallel \mathsf{a}+) \circ \mathsf{h.0}$, a simple consequence of the promotion theorem), and so

$$\gg \circ \llparenthesis !0 \curlywedge 0+, \oplus, \otimes \rrparenthesis \circ \mathsf{td} \quad = \quad \llparenthesis \mathsf{id}, \gg, + \rrparenthesis$$

whence

$$\gg\!* \circ (!0 \curlywedge \mathsf{id}, \oplus, \otimes)\curlywedge_{\downarrow} \quad = \quad (\mathsf{id}, \gg, +)\Downarrow$$

Thus,

$$\begin{array}{ll} & \mathsf{rank} \\ = & \qquad [\![\text{ definition, page 83 }]\!] \\ & (\mathsf{id}, \mathsf{it})* \circ (\mathsf{id}, \gg, +)\Downarrow \circ \gg\!* \circ (!(1,1), (+ \curlywedge \ll) \circ \ll^2)\Uparrow \\ = & \qquad [\![\text{ above }]\!] \\ & (\mathsf{id}, \mathsf{it})* \circ \gg\!* \circ (!0 \curlywedge \mathsf{id}, \oplus, \otimes)\curlywedge_{\downarrow} \circ \gg\!* \circ (!(1,1), (+ \curlywedge \ll) \circ \ll^2)\Uparrow \end{array}$$

which is the algorithm running in time proportional to the depth of the tree that was promised earlier.

We have just seen that not all paths accumulations are htaps accumulations; we now see a counterexample to the inclusion in the other direction. Consider the function $\mathsf{tp.a}$, which tuples every element of a tree with its parent, tupling the root with a. It is defined by

$$\mathsf{tp.a} \quad = \quad (!\mathsf{a} \curlywedge \mathsf{id}, \oplus, \oplus)\curlywedge_{\downarrow} \qquad \text{where} \quad (\mathsf{a},\mathsf{b})\oplus \;=\; !\mathsf{b} \curlywedge \mathsf{id}$$

For the sake of brevity, as before, define the functions

$$\begin{array}{lcl} \mathsf{f} & = & \llparenthesis \mathsf{daerht}\colon !\mathsf{a} \curlywedge \mathsf{id}, \oplus, \oplus \rrparenthesis \circ \mathsf{td} \\ \mathsf{g} & = & \llparenthesis \mathsf{thread}\colon !\mathsf{a} \curlywedge \mathsf{id}, \otimes, \otimes \rrparenthesis \end{array}$$

for some $\otimes$. We show that f and g must differ on some arguments; hence, there is no operator $\otimes$ such that f and g are equal.

Define the three threads

$$\begin{array}{lcl} \mathsf{x} & = & \mathsf{a} \hookrightarrow \diamond.\mathsf{c} \\ \mathsf{y} & = & \mathsf{b} \hookrightarrow (\mathsf{a} \hookrightarrow \diamond.\mathsf{c}) \\ \mathsf{z} & = & \mathsf{b} \hookrightarrow \diamond.\mathsf{c} \end{array}$$

with a and b different. We have

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{f.x} & = & (\mathsf{a},\mathsf{c}) & \mathsf{g.x} & = & \mathsf{a} \otimes (\mathsf{a},\mathsf{c}) \\ \mathsf{f.y} & = & (\mathsf{a},\mathsf{c}) & \mathsf{g.y} & = & \mathsf{b} \otimes (\mathsf{a} \otimes (\mathsf{a},\mathsf{c})) \\ \mathsf{f.z} & = & (\mathsf{b},\mathsf{c}) & \mathsf{g.z} & = & \mathsf{b} \otimes (\mathsf{a},\mathsf{c}) \end{array}$$

If f and g are to be equal, then certainly $\mathsf{a} \otimes (\mathsf{a},\mathsf{c}) = (\mathsf{a},\mathsf{c})$ —but then f and g cannot agree on both y and z. Therefore, there is no thread catamorphism equal to f, and no paths accumulation equal to $\mathsf{tp.a}$.

Most natural downwards accumulations, though, are both paths- and htaps accumulations, and so are both catamorphic and linear—that is, both easily manipulated and efficiently implemented. In the next chapter, for example, we look for a function that can be written as a paths accumulation, but that has the associativity properties that allow it to be inverted to produce a htaps accumulation; we can calculate using the tractable catamorphic form, but know all the time that we can evaluate it quickly in the efficient form.

Downwards accumulation on rose trees

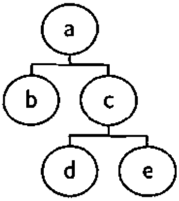
Now that we have worked out all the details for moo trees, the definition of downwards accumulations on rose trees causes no great surprises; the only aspect that needs a little thought is the definition of *rose threads.*

48. Definition

$$\mathsf{rthread.A} \quad = \quad \diamond.\mathsf{A} \mid \mathsf{A} \Downarrow_{\mathbb{N}} \mathsf{rthread.A}$$

◇

Thus, rose threads are built from a unary and a ternary constructor. Informally, the ternary constructor could be seen as a countable infinity of binary constructors, corresponding to the two binary constructors $\hookleftarrow$ and $\hookrightarrow$ of binary threads. The intention is that, for example, the rose thread to the element d in the homogeneous rose tree



is $\mathsf{a} \Downarrow_1 (\mathsf{c} \Downarrow_0 \diamond\cdot\mathsf{d})$; the numbers identify which branch to take on each level, with 0 signifying the first branch.

Rose thread catamorphisms satisfy the equations

$$\begin{aligned} (\!|\mathsf{rthread}\colon \mathsf{f}, \oplus|\!)\cdot\triangle\cdot\mathsf{a} \quad &= \quad \mathsf{f}\cdot\mathsf{a} \\ (\!|\mathsf{rthread}\colon \mathsf{f}, \oplus|\!)\cdot(\mathsf{a} \Downarrow_{\mathsf{i}} \mathsf{x}) \quad &= \quad \mathsf{a} \oplus_{\mathsf{i}} (\!|\mathsf{rthread}\colon \mathsf{f}, \oplus|\!)\cdot\mathsf{x} \end{aligned}$$

dictated by the above definition.

The function paths on rose trees—again, we trust to context to resolve any ambiguities between rose and moo tree accumulations—replaces every element of a rose tree with the path to that element, and is given by

$$\begin{aligned} \mathsf{paths}\cdot\triangle\cdot\mathsf{a} \quad &= \quad \triangle\cdot\diamond\cdot\mathsf{a} \\ \mathsf{paths}\cdot(\mathsf{a} \prec \mathsf{x}) \quad &= \quad \diamond\cdot\mathsf{a} \prec (\mathsf{index}\cdot\mathsf{x} \curlyvee_{\oplus}, (\mathsf{paths} * \mathsf{x})) \end{aligned}$$

where

$$\mathsf{i} \oplus_{\mathsf{a}} \mathsf{r} \quad = \quad \langle \mathsf{a}_{\leftrightarrow_{\mathsf{i}}} \rangle * \mathsf{r}$$

and where the function index is as defined in Chapter 2:

$$\mathsf{index} \quad = \quad (\langle -1 \rangle \circ \#) * \circ\, \mathsf{inits}$$

It is clear that $\mathsf{index} = \mathsf{index} \circ \mathsf{paths}*$, because a map does not change the length of a list, and so

$$\mathsf{paths} \cdot (\mathsf{a} \prec \mathsf{x}) \quad = \quad \diamond \cdot \mathsf{a} \prec (\mathsf{index} \cdot (\mathsf{paths} * \mathsf{x})\ \Upsilon_{\oplus_{\mathsf{a}}}\ (\mathsf{paths} * \mathsf{x}))$$

and paths is catamorphic:

$$\mathsf{paths} \quad = \quad (\!|\mathsf{rtree}\colon \vartriangle \circ \diamond, \otimes|\!)$$

where

$$\mathsf{a} \otimes \mathsf{ps} \quad = \quad \diamond \cdot \mathsf{a} \prec (\mathsf{index} \cdot \mathsf{ps}\ \Upsilon_{\oplus_{\mathsf{a}}}\ \mathsf{ps})$$

A paths accumulation over rose trees is just a rose thread catamorphism mapped over paths:

49. Definition

$$(\mathsf{f}, \oplus)\Downarrow \quad = \quad (\!|\mathsf{rthread}\colon \mathsf{f}, \oplus|\!) * \circ\, \mathsf{paths}$$

◇

For example, the function dewey from Chapter 2 is given more succinctly by

$$\mathsf{dewey} \quad = \quad (!(\square \cdot 0), \oplus)\Downarrow \qquad \text{where} \quad \mathsf{a} \oplus_{\mathsf{i}} \mathsf{r} = \mathsf{i} :\!< \mathsf{r}$$

Paths accumulations on rose trees are inefficient, just as are paths accumulations on moo trees; the above characterization of dewey, for example, like the characterization given in Chapter 2, takes quadratic effort in the worst case. With this in mind, we define rose *daerhts*.

50. Definition

$$\mathsf{rdaerht}.A \quad = \quad \diamond.A \mid \mathsf{rdaerht}.A \hookleftarrow_N A$$

$$\diamond$$

The obvious correspondence between rose daerhts and rose threads is made formal by the function $\mathsf{rtd} \in \mathsf{rthread}.A \longrightarrow \mathsf{rdaerht}.A$:

$$\mathsf{rtd} \quad = \quad (\!|\mathsf{rthread}\colon \diamond, \oplus|\!) \qquad \text{where} \quad \langle a \oplus_i \rangle = (\!|\mathsf{rdaerht}\colon \diamond \cdot a \hookleftarrow_i, \hookleftarrow|\!)$$

For example, we have

$$\mathsf{rtd} \cdot (a \hookrightarrow_1 (c \hookrightarrow_0 \diamond \cdot d)) \quad = \quad (\diamond \cdot a \hookleftarrow_1 c) \hookleftarrow_0 d$$

We can compute the htaps of a rose tree by converting the paths into daerhts, just as we did for moo trees:

$$\mathsf{htaps} \quad = \quad \mathsf{rtd}{*} \circ \mathsf{paths}$$

And, finally, we can express htaps accumulation, an efficient downwards accumulation on rose trees, by

51. Definition

$$(f, \oplus)\curlywedge \quad = \quad (\!|\mathsf{rdaerht}\colon f, \oplus|\!){*} \circ \mathsf{htaps}$$

$$\diamond$$

This accumulation satisfies the equation

$$(f, \oplus) \curlywedge (a \prec x) \quad = \quad f \cdot a \prec (\mathsf{index} \cdot x \curlyvee_{\odot_a} x)$$

where

$$i \odot_a t \quad = \quad (\langle f \cdot a \oplus_i \rangle, \oplus) \curlywedge t$$

For example, a linear effort characterization of Dewey Decimal indexing, yewed , returning a homogeneous rose tree of snoc lists, is given by

$$\mathsf{yewed} \quad = \quad (!(\square \cdot 0), \oplus)\curlywedge \qquad \text{where} \quad r \oplus_i a = r \mathbin{:\!\cdot} i$$

This concludes the definitions of accumulations on trees; in the next three chapters we shall see them at work on some example problems.

5 Prefix sums

The *prefix sums* problem consists of evaluating all the 'running totals' $(\!|\mathsf{cat}\colon \mathsf{f}, \odot|\!) * \circ \mathsf{inits}$ of a list. The operator $\odot$ must be associative for the catamorphism to be proper; we also assume that $\odot$ has a unit, e. For example, applied to the list $[\mathsf{a}_0, \ldots, \mathsf{a}_{n-1}]$, the problem is to evaluate

$$[\mathsf{f.a}_0, \mathsf{f.a}_0 \odot \mathsf{f.a}_1, \ldots, \mathsf{f.a}_0 \odot \mathsf{f.a}_1 \odot \cdots \odot \mathsf{f.a}_{n-1}]$$

This problem encapsulates a very common pattern of computation on lists; it has applications in, among other places, the evaluation of polynomials, compiler design, and numerous graph problems including minimum spanning tree and connected components (Akl, 1989).

It would appear from the above example that the problem inherently takes linear time to solve; the structure of the result seems to preclude any faster solution. However, Ladner and Fischer (1980), reworking earlier results by Kogge and Stone (1973) and Estrin (1960), prove the rather unexpected result that the evaluation can be performed in *logarithmic* time on a linear number of processors acting in parallel. We will derive their 'parallel prefix' algorithm in this chapter, but we will express it in higher level terms than they do—in fact, in terms of upwards and downwards accumulations.

Recall the function rank from the previous chapter. It satisfies the equation

$$\mathsf{iol} \circ \mathsf{rank} \quad = \quad \# * \circ \mathsf{inits} \circ \mathsf{iol}$$

where iol, short for 'inorder leaves', returns the list of leaves of a tree, in left-to-right order:

$$\mathsf{iol} \quad = \quad (\!|\square, \oplus|\!) \qquad \text{where} \quad \mathsf{u} \oplus_{\mathsf{a}} \mathsf{v} = \mathsf{u} +\!\!+ \mathsf{v}$$

We defined inits in Chapter 3 to have snoc lists as source and target; here we lift it in the obvious way to cat lists.

Suppose we have a pre-inverse loi of iol, that is, a function satisfying $\mathsf{iol} \circ \mathsf{loi} = \mathsf{id}$; such a function takes a list and returns a tree whose leaves in inorder traversal produce this list. Now, loi allows us to use rank to compute $\#* \circ \mathsf{inits}$:

$$\#* \circ \mathsf{inits} \quad = \quad \mathsf{iol} \circ \mathsf{rank} \circ \mathsf{loi}$$

Ignoring the time taken to construct and destroy the tree—hopefully this cost can be spread over several operations—the right hand side will take parallel time proportional to the depth of the tree; if loi constructs a balanced tree then the right hand side will take logarithmic parallel time.

The left hand side of this equation, $\#* \circ \mathsf{inits}$, is an instance of the prefix sums problem, since $\#$ is a catamorphism $\llparenthesis \mathsf{cat}{:}\ !1, + \rrparenthesis$; we might ask whether we can find a function corresponding to rank for *any* instance of the problem. That is, for given f and $\odot$ such that $\odot$ is associative and has a unit, we would like to find a function pps satisfying the implicit specification

$$\mathsf{iol} \circ \mathsf{pps} \quad = \quad \llparenthesis \mathsf{f}, \odot \rrparenthesis * \circ \mathsf{inits} \circ \mathsf{iol}$$

The need for $\odot$ to have a unit is not obvious from the specification, but becomes clear as we proceed to calculate. The domain and range of $\odot$ can always be augmented with such a unit, if none already exists.

Calculating the parallel prefix algorithm

We can calculate immediately the result of applying pps to a leaf, since

$$\begin{array}{ll} & \mathsf{iol} \circ \mathsf{pps} \circ \triangle \\ = & \qquad \left[\!\left[\text{ specification of } \mathsf{pps} \right]\!\right] \\ & \llparenthesis \mathsf{f}, \odot \rrparenthesis * \circ \mathsf{inits} \circ \mathsf{iol} \circ \triangle \end{array}$$

$$
\begin{aligned}
= \quad & [\![\ \mathsf{iol}\ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \circ \mathsf{inits} \circ \Box \\
= \quad & [\![\ \mathsf{inits}\ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \circ \Box \circ \Box \\
= \quad & [\![\ * \ ]\!] \\
& \Box \circ (\![\mathsf{f},\odot]\!) \circ \Box \\
= \quad & [\![\ \text{catamorphisms}\ ]\!] \\
& \Box \circ \mathsf{f}
\end{aligned}
$$

and hence

$$\mathsf{pps} \circ \triangle \quad = \quad \triangle \circ \mathsf{f}$$

since iol is injective on leaves. Letting $\mathsf{s} = (\![\mathsf{f},\odot]\!) \circ \mathsf{iol}$, we get on branches

$$
\begin{aligned}
& \mathsf{iol}\cdot\mathsf{pps}\cdot(\mathsf{x} \pm \mathsf{y}) \\
= \quad & [\![\ \text{specification of pps}\ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \cdot\mathsf{inits}\cdot\mathsf{iol}\cdot(\mathsf{x} \pm \mathsf{y}) \\
= \quad & [\![\ \mathsf{iol}\ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \cdot\mathsf{inits}\cdot(\mathsf{iol}\cdot\mathsf{x} \mathbin{+\!\!\!+} \mathsf{iol}\cdot\mathsf{y}) \\
= \quad & [\![\ \mathsf{inits}\ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \cdot(\mathsf{inits}\cdot\mathsf{iol}\cdot\mathsf{x} \mathbin{+\!\!\!+} \langle \mathsf{iol}\cdot\mathsf{x} \mathbin{+\!\!\!+} \rangle * \mathsf{inits}\cdot\mathsf{iol}\cdot\mathsf{y}) \\
= \quad & [\![\ * \ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \mathsf{inits}\cdot\mathsf{iol}\cdot\mathsf{x} \ \mathbin{+\!\!\!+}\ (\![\mathsf{f},\odot]\!) * \langle \mathsf{iol}\cdot\mathsf{x} \mathbin{+\!\!\!+} \rangle * \mathsf{inits}\cdot\mathsf{iol}\cdot\mathsf{y} \\
= \quad & [\![\ \text{catamorphisms}\ ]\!] \\
& (\![\mathsf{f},\odot]\!) * \mathsf{inits}\cdot\mathsf{iol}\cdot\mathsf{x} \ \mathbin{+\!\!\!+}\ \langle \mathsf{s}\cdot\mathsf{x}\odot \rangle * (\![\mathsf{f},\odot]\!) * \mathsf{inits}\cdot\mathsf{iol}\cdot\mathsf{y}
\end{aligned}
$$

$$\begin{aligned} = \quad & [\![\text{ specification of pps }]\!] \\ & \mathsf{iol}\cdot\mathsf{pps}\cdot x \mathbin{+\!\!\!+} \langle s\cdot x\odot\rangle * \mathsf{iol}\cdot\mathsf{pps}\cdot y \end{aligned}$$

This does not completely determine pps on branches, since iol is not injective on branches, but it is 'sweetly reasonable' to suppose that

$$\mathsf{pps}\cdot(x \mathbin{\underline{+}} y) \quad = \quad \mathsf{pps}\cdot x \mathbin{\underline{+}}_a \langle s\cdot x\odot\rangle * \mathsf{pps}\cdot y$$

for some a ; certainly, this supposition satisfies the implicit specification of pps. The calculation can tell us nothing about the value of a , the root of $\mathsf{pps}\cdot(x \mathbin{\underline{+}} y)$, because iol throws branch labels away.

This gives us an explicit—that is, executable—specification of pps :

$$\begin{aligned} \mathsf{pps}\cdot\triangle\cdot a \quad &= \quad \triangle\cdot f\cdot a \\ \mathsf{pps}\cdot(x \mathbin{\underline{+}} y) \quad &= \quad \mathsf{pps}\cdot x \mathbin{\underline{+}}_a \langle s\cdot x\odot\rangle * \mathsf{pps}\cdot y \end{aligned}$$

for some a . Executing this specification requires parallel time quadratic in the depth of the tree in the worst case; we show next how to improve this to linear parallel time.

Suppose that $a = s\cdot x$, that is, that

$$\mathsf{pps}\cdot(x \mathbin{\underline{+}} y) \quad = \quad \mathsf{pps}\cdot x \mathbin{\underline{+}}_{s\cdot x} \langle s\cdot x\odot\rangle * \mathsf{pps}\cdot y$$

Intuitively, this allows the computation of $\mathsf{pps}\cdot(x \mathbin{\underline{+}} y)$ to be split into two parts, the first bringing $s\cdot x$ to the root of the tree and the second mapping $\langle s\cdot x\odot\rangle$ over the right child. More formally, suppose that

$$\begin{aligned} \mathsf{up}\cdot(x \mathbin{\underline{+}} y) \quad &= \quad \mathsf{up}\cdot x \mathbin{\underline{+}}_{s\cdot x} \mathsf{up}\cdot y \\ \mathsf{down}\cdot(u \mathbin{\underline{+}}_b v) \quad &= \quad \mathsf{down}\cdot u \mathbin{\underline{+}}_b \langle b\odot\rangle * \mathsf{down}\cdot v \end{aligned}$$

whence

$$\mathsf{down}\cdot\mathsf{up}\cdot(x \mathbin{\underline{+}} y) \quad = \quad \mathsf{down}\cdot\mathsf{up}\cdot x \mathbin{\underline{+}}_{s\cdot x} \langle s\cdot x\odot\rangle * \mathsf{down}\cdot\mathsf{up}\cdot y$$

so $\mathsf{down} \circ \mathsf{up}$ follows the same pattern as pps . Provided that

$$\mathsf{down} \circ \mathsf{up} \circ \triangle \quad = \quad \mathsf{pps} \circ \triangle$$

which holds if

$$\begin{aligned}\mathsf{up}\cdot\triangle\cdot a &= \triangle\cdot f\cdot a\\ \mathsf{down}\cdot\triangle\cdot b &= \triangle\cdot b\end{aligned}$$

an inductive proof shows that

$$\mathsf{pps} = \mathsf{down}\circ\mathsf{up}$$

We cannot use the unique extension property because pps is not expressed as a catamorphism.

We have not yet improved the efficiency; up and down both take parallel time quadratic in the depth of the tree. However, as the names suggest, up is an upwards function and down a downwards accumulation, and we know something about making such functions efficient.

Let $\mathsf{sl} = \mathsf{root}\circ\mathsf{up}$, so

$$\begin{aligned}\mathsf{sl}\cdot\triangle\cdot a &= f\cdot a\\ \mathsf{sl}\cdot(x \pm y) &= s\cdot x\end{aligned}$$

Now,

$$\mathsf{up} = \mathsf{sl}{*}\circ\mathsf{subtrees}$$

so up is an upwards function—vindicating the choice of name. It is not an accumulation, though, because sl is not a catamorphism; however, $s \curlywedge \mathsf{sl}$ *is* a catamorphism,

$$s \curlywedge \mathsf{sl} = (\!|\, f \curlywedge f, (\odot \curlywedge \ll)\circ \ll^2 \,|\!)$$

as a little calculation shows. This means that

$$\begin{aligned}&\mathsf{up}\\ =\quad & [\![\ \text{above}\ ]\!]\\ &\mathsf{sl}{*}\circ\mathsf{subtrees}\end{aligned}$$

$$
\begin{array}{ll}
= & \llbracket \text{ pairs } \rrbracket \\
 & \gg\!\ast \circ (\mathsf{s} \curlywedge \mathsf{sl})\ast \circ \mathsf{subtrees} \\
= & \llbracket \ \mathsf{s} \curlywedge \mathsf{sl} \text{ is a catamorphism } \rrbracket \\
 & \gg\!\ast \circ \llparenthesis \mathsf{f} \curlywedge \mathsf{f}, (\odot \curlywedge \ll) \circ \ll^2 \rrparenthesis\ast \circ \mathsf{subtrees} \\
= & \llbracket \Uparrow \rrbracket \\
 & \gg\!\ast \circ (\mathsf{f} \curlywedge \mathsf{f}, (\odot \curlywedge \ll) \circ \ll^2)\Uparrow
\end{array}
$$

which—assuming that $\odot$ takes constant time—can be evaluated in time proportional to the depth of the tree in parallel.

So much for up ; what about down ? We have

$$
\begin{array}{rcl}
\mathsf{down} \cdot \triangle \cdot \mathsf{a} & = & \triangle \cdot \mathsf{a} \\
\mathsf{down} \cdot (\mathsf{u} \pm_{\mathsf{b}} \mathsf{v}) & = & \mathsf{down} \cdot \mathsf{u} \pm_{\mathsf{b}} \langle \mathsf{b}\odot \rangle \ast \mathsf{down} \cdot \mathsf{v}
\end{array}
$$

and so down is already a paths accumulation,

$$\mathsf{down} \quad = \quad (\mathsf{id}, \gg, \odot)\Downarrow$$

However, $(\mathsf{id}, \gg, \odot)$ is not top-down, as we showed on page 93 for the special case where $\odot$ is $+$. Consider, though, the fork of thread catamorphisms

$$\llparenthesis !\mathsf{e}, \gg, \odot \rrparenthesis \curlywedge \llparenthesis \mathsf{id}, \gg, \odot \rrparenthesis$$

By Theorem 13, this is itself a catamorphism:

$$\llparenthesis !\mathsf{e}, \gg, \odot \rrparenthesis \curlywedge \llparenthesis \mathsf{id}, \gg, \odot \rrparenthesis \quad = \quad \llparenthesis !\mathsf{e} \curlywedge \mathsf{id}, \gg, \boxtimes \rrparenthesis$$

where

$$\mathsf{a} \boxtimes (\mathsf{b}, \mathsf{c}) \quad = \quad (\mathsf{a} \odot \mathsf{b}, \mathsf{a} \odot \mathsf{c})$$

Moreover, it is top-down—it inverts to $\llparenthesis !\mathsf{e} \curlywedge \mathsf{id}, \oplus, \otimes \rrparenthesis$ where

$$
\begin{array}{rcl}
(\mathsf{b}, \mathsf{c}) \oplus \mathsf{d} & = & (\mathsf{b}, \mathsf{b} \odot \mathsf{d}) \\
(\mathsf{b}, \mathsf{c}) \otimes \mathsf{d} & = & (\mathsf{c}, \mathsf{c} \odot \mathsf{d})
\end{array}
$$

Thus,

$$\begin{array}{ll}
 & \mathsf{down} \\
= & \qquad [\![\ \mathsf{down}\ ]\!] \\
 & (\mathsf{id}, \gg, \odot) \Downarrow \\
= & \qquad [\![\ \Downarrow\ ]\!] \\
 & (\!|\mathsf{id}, \gg, \odot|\!) * \circ \mathsf{paths} \\
= & \qquad [\![\ \text{pairs}\ ]\!] \\
 & \gg * \circ (\!|\,!\mathsf{e} \curlywedge \mathsf{id}, \gg, \boxtimes|\!) * \circ \mathsf{paths} \\
= & \qquad [\![\ \text{Theorem 47}\ ]\!] \\
 & \gg * \circ (!\mathsf{e} \curlywedge \mathsf{id}, \oplus, \otimes) \curlywedge\!\!\!\searrow
\end{array}$$

This gives us the promised efficient algorithm for pps :

$$\mathsf{pps} \quad = \quad \gg * \circ (!\mathsf{e} \curlywedge \mathsf{id}, \oplus, \otimes) \curlywedge\!\!\!\searrow \circ \gg * \circ (\mathsf{f} \curlywedge \mathsf{f}, (\odot \curlywedge \ll) \circ \ll^2) \Uparrow$$

Note that the map between the two accumulations can be absorbed into the downwards accumulation, by Theorem 45 and Corollary 12, though this does not change the asymptotic efficiency of the algorithm.

O'Donnell (1990) has presented a similar derivation to this, but he only went as far as producing a catamorphic characterization of pps ,

$$\mathsf{pps}.(\mathsf{x} \mathbin{+\!\!\!\ast} \mathsf{y}) \quad = \quad \mathsf{g}.(\mathsf{pps}.\mathsf{x} ⦶ \mathsf{pps}.\mathsf{y})$$

and then making it efficient by making it 'top-down'; he did not separate out the ⦶ and g to get an upwards accumulation followed by a downwards accumulation. The result is a very operational description of a 'sweep' operation consisting of a tree processes, each of which 'sends information in both directions on each data path'. The advantage of identifying the two accumulations is that it becomes clear that the algorithm operates in two distinct phases; the intermediate results, and even the fact that the algorithm terminates in the first place, are more evident.

Prefix sums for a non-associative operator

One surprising property of the prefix sums problem is that it can even be applied to snoc list catamorphisms, where the binary operator need not be associative. Imagine, for example, that we have a finite state machine with initial state e and with state transition function $\oplus$, which produces a new state $s \oplus i$ given the old state s and input i. This machine is modelled by the catamorphism $(\![\mathsf{snoc}\colon e\oplus, \oplus]\!)$ on non-empty snoc lists; the operator $\oplus$ certainly is not associative, because its left and right arguments are not of the same type. Running this finite state machine on the input list x produces the list of states $(\![\mathsf{snoc}\colon e\oplus, \oplus]\!) * \mathsf{inits}.x$, which looks very much like an instance of the prefix sums problem.

Imagine also that we know the list of inputs that the machine will be given, but that we do not know the initial state; this is the case, for example, when we want to run the machine concurrently on two halves of an input list, since the initial state for the second half is the final state for the first. Can we rewrite $(\![e\oplus, \oplus]\!)$ in a form that allows us to do some 'precomputation' using the inputs? That is, can we extract the e from the catamorphism?

We have

$$
\begin{array}{ll}
 & u \oplus a \\
= & \qquad [\![\text{ sectioning, } \bar{\cdot} \ ]\!] \\
 & u \mathbin{\bar{\cdot}} \langle \oplus a \rangle \\
= & \qquad [\![\ \bar{\cdot} \text{ and sectioning again }]\!] \\
 & u \mathbin{\bar{\cdot}} (\langle \bar{\oplus} \rangle \cdot a)
\end{array}
$$

so

$$\oplus \quad = \quad \bar{\cdot} \circ \langle \mathsf{id} \| \langle \bar{\oplus} \rangle \rangle$$

and in particular,

$$
\begin{array}{ll}
 & e \oplus a \\
= & \qquad \llbracket \text{ above } \rrbracket \\
 & e \mathbin{\bar{\cdot}} (\langle \bar{\oplus} \rangle \cdot a) \\
= & \qquad \llbracket \text{ sectioning } \rrbracket \\
 & \langle e \bar{\cdot} \rangle \cdot (\langle \bar{\oplus} \rangle \cdot a) \\
= & \qquad \llbracket \circ \rrbracket \\
 & (\langle e \bar{\cdot} \rangle \circ \langle \bar{\oplus} \rangle) \cdot a
\end{array}
$$

so

$$e\oplus \quad = \quad \langle e \bar{\cdot} \rangle \circ \langle \bar{\oplus} \rangle$$

This gives us

$$\llparenthesis e\oplus, \oplus \rrparenthesis \quad = \quad \llparenthesis \langle e \bar{\cdot} \rangle \circ \langle \bar{\oplus} \rangle, \bar{\cdot} \circ \mathrm{id} \| \langle \bar{\oplus} \rangle \rrparenthesis$$

Recall Theorem 11, the promotion theorem, from the introduction; instantiated for snoc lists, it states that

$$f.(x \oplus a) = f.x \otimes a \quad \Rightarrow \quad f \circ \llparenthesis g, \oplus \rrparenthesis = \llparenthesis f \circ g, \otimes \rrparenthesis$$

Now,

$$
\begin{array}{ll}
 & (\bar{\cdot} \circ \mathrm{id} \| \langle \bar{\oplus} \rangle) \cdot (e \mathbin{\bar{\cdot}} x, a) \\
= & \qquad \llbracket \circ \rrbracket \\
 & (e \mathbin{\bar{\cdot}} x) \mathbin{\bar{\cdot}} (\langle \bar{\oplus} \rangle \cdot a) \\
= & \qquad \llbracket \ (u \mathbin{\bar{\cdot}} f) \mathbin{\bar{\cdot}} g = u \mathbin{\bar{\cdot}} (f \mathbin{\bar{\circ}} g) \ \rrbracket \\
 & e \mathbin{\bar{\cdot}} (x \mathbin{\bar{\circ}} \langle \bar{\oplus} \rangle \cdot a) \\
= & \qquad \llbracket \circ \rrbracket \\
 & e \mathbin{\bar{\cdot}} ((\bar{\circ} \circ \mathrm{id} \| \langle \bar{\oplus} \rangle) \cdot (x, a))
\end{array}
$$

so by the promotion theorem, with f , g , $\oplus$ and $\otimes$ instantiated to $\langle e\bar{\cdot}\rangle$, $\langle\bar{\oplus}\rangle$, $\bar{\circ} \circ \mathsf{id}\|\langle\bar{\oplus}\rangle$ and $\bar{\cdot} \circ \mathsf{id}\|\langle\bar{\oplus}\rangle$, we have

$$(\!|\langle e\bar{\cdot}\rangle \circ \langle\bar{\oplus}\rangle, \bar{\cdot} \circ \mathsf{id}\|\langle\bar{\oplus}\rangle|\!) \quad = \quad \langle e\bar{\cdot}\rangle \circ (\!|\langle\bar{\oplus}\rangle, \bar{\circ} \circ \mathsf{id}\|\langle\bar{\oplus}\rangle|\!)$$

If a binary operator $\boxplus$ is associative, then the snoc list catamorphism $(\!|\mathsf{snoc}\colon f, \boxplus \circ \mathsf{id}\|f|\!)$ and the cat list catamorphism $(\!|\mathsf{cat}\colon f, \boxplus|\!)$ are equal, modulo input type conversions; that is, if cs is the function that converts a cat list to the corresponding snoc list,

$$cs \quad = \quad (\!|\mathsf{cat}\colon \square, \oplus|\!) \qquad \text{where} \quad \langle x\oplus\rangle = (\!|\mathsf{snoc}\colon x{:}\!\cdot, {:}\!\cdot|\!)$$

then

$$(\!|\mathsf{snoc}\colon f, \boxplus \circ \mathsf{id}\|f|\!) \circ cs \quad = \quad (\!|\mathsf{cat}\colon f, \boxplus|\!)$$

The operator in this case, $\bar{\circ}$, is indeed associative, so we have

$$(\!|\mathsf{snoc}\colon e\oplus, \oplus|\!) \circ cs \quad = \quad \langle e\bar{\cdot}\rangle \circ (\!|\mathsf{cat}\colon \langle\bar{\oplus}\rangle, \bar{\circ}|\!)$$

In effect, we have shown that, although $(\!|\mathsf{snoc}\colon e\oplus, \oplus|\!) \circ cs$ is not a cat list catamorphism, the function f satisfying

$$f.x.e \quad = \quad (\!|\mathsf{snoc}\colon e\oplus, \oplus|\!) \cdot cs \cdot x$$

is a catamorphism.

In the case of a general snoc list catamorphism, this does not produce any gain in efficiency; the 'sum' $(\!|\mathsf{cat}\colon \langle\bar{\oplus}\rangle, \bar{\circ}|\!).x$ is a composition of functions which takes order $\#.x$ steps to apply to an argument. However, if the left domain and range of $\oplus$ is finite—for example, if $\oplus$ is the state transition function for a finite state machine—then each function $\langle\bar{\oplus}\rangle.x_i$ in this composition of functions is a finite mapping, and the whole composition can be precomputed by composing these finite mappings; the precomputed composition *can* then be applied in constant time.

This means that we can use the parallel prefix algorithm to run a finite state machine on n inputs in $\log.n$ time in parallel. This method is commonly used to produce fast carry-lookahead circuits (Ladner and

Fischer, 1980) and to parallelize the 'lexing' stage of compilation (Schell, 1979).

Suffix sums

We have just derived the parallel prefix algorithm for the prefix sums problem $\mathsf{ps} = \llparenthesis \mathsf{f}, \odot \rrparenthesis * \circ \mathsf{inits}$. A closely related problem is that of finding the *suffix* sums $\mathsf{ss} = \llparenthesis \mathsf{f}, \odot \rrparenthesis * \circ \mathsf{tails}$ of a list, where the injective function tails returns the list of tail segments of a list in descending order of length:

$$\begin{aligned} \mathsf{tails}\cdot\square\cdot\mathsf{a} &= \square\cdot\square\cdot\mathsf{a} \\ \mathsf{tails}\cdot(\mathsf{x} +\!\!+ \mathsf{y}) &= ((+\!\!+\mathsf{y}) * \mathsf{tails}\cdot\mathsf{x}) +\!\!+ \mathsf{tails}\cdot\mathsf{y} \end{aligned}$$

By following exactly the same reasoning as for prefix sums, we can calculate a 'parallel suffix' algorithm

$$\mathsf{pss} \;=\; \gg * \circ (!\mathsf{e} \curlywedge \mathsf{id}, \oplus, \otimes)_{\curlywedge} \circ \gg * \circ (\mathsf{f} \curlywedge \mathsf{f}, (\odot \curlywedge \gg) \circ \ll^2) \Uparrow$$

where

$$\begin{aligned} (\mathsf{b}, \mathsf{c}) \oplus \mathsf{d} &= (\mathsf{c}, \mathsf{d} \odot \mathsf{c}) \\ (\mathsf{b}, \mathsf{c}) \otimes \mathsf{d} &= (\mathsf{b}, \mathsf{d} \odot \mathsf{b}) \end{aligned}$$

This parallel suffix algorithm satisfies

$$\mathsf{iol} \circ \mathsf{pss} \;=\; \mathsf{ss} \circ \mathsf{iol}$$

Indeed, we can find prefix and suffix sums *together*, with an algorithm of the same form. That is, suppose we have to replace every element a of a list $\mathsf{x} +\!\!+ \square.\mathsf{a} +\!\!+ \mathsf{y}$ with the value

$$\llparenthesis \mathsf{f}, \oplus \rrparenthesis.(\mathsf{x} +\!\!+ \square.\mathsf{a}) \;\odot\; \llparenthesis \mathsf{g}, \otimes \rrparenthesis.(\square.\mathsf{a} +\!\!+ \mathsf{y})$$

for some fixed f, $\oplus$, g, $\otimes$ and $\odot$; the task of making this replacement for every element of a list is performed by the function

$$(\llparenthesis \mathsf{f}, \oplus \rrparenthesis * \circ \mathsf{inits}) \;\hat{Y}_{\odot}\; (\llparenthesis \mathsf{g}, \otimes \rrparenthesis * \circ \mathsf{tails})$$

where we have lifted the operator $\curlyvee$, the post-inverse of $\ll\!*\ \curlywedge \gg\!*$, to act on pairs of equal-length cat lists. Call this function fs, for 'fix sums', since it involves both prefix and suffix sums. The claim is that there is a 'parallel fix' algorithm pfs, of the same form as the parallel prefix algorithm, which satisfies

$$\mathsf{iol} \circ \mathsf{pfs} \quad = \quad \mathsf{fs} \circ \mathsf{iol}$$

To show this, we first note that the fix sums can be found by zipping the parallel prefix and suffix trees, then taking the leaves; once more, we lift the zip operator, this time to pairs of same-shaped trees.

$$
\begin{array}{ll}
 & \mathsf{fs} \circ \mathsf{iol} \\
= & \qquad \llbracket\ \mathsf{fs}\ \rrbracket \\
 & (\mathsf{ps}\ \hat{\curlyvee}_{\odot}\ \mathsf{ss}) \circ \mathsf{iol} \\
= & \qquad \llbracket\ \text{pairs}\ \rrbracket \\
 & \curlyvee_{\odot} \circ (\mathsf{ps} \circ \mathsf{iol}) \curlywedge (\mathsf{ss} \circ \mathsf{iol}) \\
= & \qquad \llbracket\ \mathsf{pps},\ \mathsf{pss}\ \rrbracket \\
 & \curlyvee_{\odot} \circ (\mathsf{iol} \circ \mathsf{pps}) \curlywedge (\mathsf{iol} \circ \mathsf{pss}) \\
= & \qquad \llbracket\ \text{wish: } \curlyvee_{\odot} \circ \mathsf{iol}^2 = \mathsf{iol} \circ \curlyvee_{\odot}\ \rrbracket \\
 & \mathsf{iol} \circ \curlyvee_{\odot} \circ (\mathsf{pps} \curlywedge \mathsf{pss}) \\
= & \qquad \llbracket\ \text{pairs}\ \rrbracket \\
 & \mathsf{iol} \circ (\mathsf{pps}\ \hat{\curlyvee}_{\odot}\ \mathsf{pss})
\end{array}
$$

So if we can fulfill the wish that $\curlyvee_{\odot} \circ \mathsf{iol}^2 = \mathsf{iol} \circ \curlyvee_{\odot}$, then $\mathsf{pps}\ \hat{\curlyvee}_{\odot}\ \mathsf{pss}$ satisfies the requirements for pfs; we have then only to show that this 'zip of two parallel fix algorithms' is another parallel fix algorithm.

In fact, the wish does not hold in general; it only holds in this case because the function $\mathsf{pps} \curlywedge \mathsf{pss}$ which precedes it returns a pair of trees that are the same 'shape'. We will prove the following lemma:

52. Lemma

$$\curlyvee \circ \mathsf{iol}^2 \circ (\ll\!* \curlywedge \gg\!*) \quad = \quad \mathsf{iol} \circ \curlyvee \circ (\ll\!* \curlywedge \gg\!*)$$

$$\Diamond$$

The more general case for $\curlyvee_{\odot}$ follows from this, since $\curlyvee_{\odot} = \odot\!* \circ \curlyvee$ and a map commutes with iol.

Proof

$$\begin{aligned}
& \curlyvee \circ \mathsf{iol}^2 \circ (\ll\!* \curlywedge \gg\!*) \\
= \quad & \qquad [\![\text{ pairs }]\!] \\
& \curlyvee \circ ((\mathsf{iol} \circ \ll\!*) \curlywedge (\mathsf{iol} \circ \gg\!*)) \\
= \quad & \qquad [\![\text{ map commutes with iol; pairs }]\!] \\
& \curlyvee \circ (\ll\!* \curlywedge \gg\!*) \circ \mathsf{iol} \\
= \quad & \qquad [\![\curlyvee \circ \ll\!* \curlywedge \gg\!* = \mathsf{id}, \text{ twice }]\!] \\
& \mathsf{iol} \circ \curlyvee \circ (\ll\!* \curlywedge \gg\!*)
\end{aligned}$$

$$\heartsuit$$

So we have shown that the equation

$$\mathsf{iol} \circ (\mathsf{pps}\ \hat{\curlyvee}_{\odot}\ \mathsf{pss}) \quad = \quad \mathsf{fs} \circ \mathsf{iol}$$

holds. We have now to show that $\mathsf{pps}\ \hat{\curlyvee}_{\odot}\ \mathsf{pss}$ is another 'parallel fix algorithm', that is, the composition of a map, a downwards accumulation, another map and an upwards accumulation. In fact, we prove the following theorem:

53. Theorem

$$\begin{aligned}
&(f\!* \circ (g, \oplus, \otimes)\!\Downarrow \circ\, h\!* \circ (k, \circledast)\!\Uparrow)\ \hat{\curlyvee}_{\odot}\ (p\!* \circ (q, \boxplus, \boxtimes)\!\Downarrow \circ\, r\!* \circ (s, \boxast)\!\Uparrow) \\
&\qquad = \quad (\odot \circ f \parallel p)\!* \circ (g \parallel q, \oplus\!\bowtie\!\boxplus, \otimes\!\bowtie\!\boxtimes)\!\Downarrow \circ\, (h \parallel r)\!* \circ (k \curlywedge s, \circledast\!\bowtie\!\boxast)\!\Uparrow
\end{aligned}$$

$$\Diamond$$

The operator $\bowtie$ satisfies

$$(\oplus) \bowtie (\boxplus) \quad = \quad (\oplus \parallel \boxplus) \circ (\ll^2 \curlywedge \gg^2)$$

That is,

$$(a, b) \oplus\!\bowtie\!\boxplus (c, d) \quad = \quad (a \oplus c, b \boxplus d)$$

It is a generalization of the 'centre-swap' operator $\ll^2 \curlywedge \gg^2$ used by Meertens and van der Woude (1991).

In order to prove Theorem 53, we must call on a number of lemmas. The first two of these are really part of the pair calculus.

54. Lemma

$$f* \curlywedge g* \quad = \quad (\ll* \curlywedge \gg*) \circ (f \curlywedge g)*$$

$\diamondsuit$

Proof

$$\begin{array}{ll}
 & (\ll* \curlywedge \gg*) \circ (f \curlywedge g)* \\
= & \qquad \llbracket \text{ pairs } \rrbracket \\
 & (\ll* \circ (f \curlywedge g)*) \curlywedge (\gg* \circ (f \curlywedge g)*) \\
= & \qquad \llbracket \text{ map distributivity; pairs } \rrbracket \\
 & f* \curlywedge g*
\end{array}$$

$\heartsuit$

55. Corollary

$$(f* \parallel g*) \circ (\ll* \curlywedge \gg*) \quad = \quad (\ll* \curlywedge \gg*) \circ (f \parallel g)*$$

$\diamondsuit$

Proof Corollary to Lemma 54, since

$$f \parallel g \quad = \quad (f \circ \ll) \curlywedge (g \circ \gg)$$

$\heartsuit$

The next two results are corollaries of general-purpose theorems about catamorphisms that we gave in Chapter 1. Before presenting them, we introduce some notation that will shorten subsequent calculations.

56. Definition Define the operators $\overset{\rightarrow}{\ }$ and $\overset{\leftarrow}{\ }$ by

$$\begin{aligned} \overset{\leftarrow}{\mathsf{f}} &= \mathsf{f} \parallel \mathsf{id} \\ \overset{\rightarrow}{\mathsf{f}} &= \mathsf{id} \parallel \mathsf{f} \end{aligned}$$

◇

Note that

$$\overset{\leftarrow}{\mathsf{f}} \circ \overset{\rightarrow}{\mathsf{g}} \quad = \quad \mathsf{f} \parallel \mathsf{g} \quad = \quad \overset{\rightarrow}{\mathsf{g}} \circ \overset{\leftarrow}{\mathsf{f}}$$

57. Definition Define the operator $\boxdot$ by

$$\oplus\boxdot\boxplus \quad = \quad (\oplus \circ \overset{\rightarrow}{\ll}) \curlywedge (\boxplus \circ \overset{\rightarrow}{\gg})$$

◇

In particular, we have

$$(\oplus \circ \overset{\leftarrow}{\ll})\boxdot(\boxplus \circ \overset{\leftarrow}{\gg}) \quad = \quad \oplus\bowtie\boxplus$$

We return now to the corollaries. Firstly, a map can be brought inside a thread catamorphism:

58. Corollary (to Corollary 12)

$$\llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrbracket \circ \mathsf{g}* \quad = \quad \llparenthesis \mathsf{thread}\colon \mathsf{f} \circ \mathsf{g}, \oplus \circ \overset{\leftarrow}{\mathsf{g}}, \otimes \circ \overset{\leftarrow}{\mathsf{g}} \rrparenthesis$$

◇

And secondly, the fork of two catamorphisms is a catamorphism:

59. Corollary (to Theorem 13)

$$\begin{aligned} \llparenthesis \mathsf{thread}\colon \mathsf{f}, \oplus, \otimes \rrparenthesis \curlywedge \llparenthesis \mathsf{thread}\colon \mathsf{g}, \boxplus, \boxtimes \rrparenthesis &= \llparenthesis \mathsf{thread}\colon \mathsf{f} \curlywedge \mathsf{g}, \oplus\boxdot\boxplus, \otimes\boxdot\boxtimes \rrparenthesis \\ \llparenthesis \mathsf{mtree}\colon \mathsf{f}, \oplus \rrparenthesis \curlywedge \llparenthesis \mathsf{mtree}\colon \mathsf{g}, \boxplus \rrparenthesis &= \llparenthesis \mathsf{mtree}\colon \mathsf{f} \curlywedge \mathsf{g}, \oplus\bowtie\boxplus \rrparenthesis \end{aligned}$$

◇

The last two lemmas we need form significant parts of the proof of the theorem; we separate them out in order to divide the otherwise rather large calculation into more manageable pieces. Informally, they state that two accumulations running 'in parallel' on same-shaped trees can be combined to form a single accumulation.

60. Lemma

$$(\mathsf{k}, \circledast)\Uparrow \Yup (\mathsf{s}, \boxast)\Uparrow \quad = \quad (\ll\!* \Yup \gg\!*) \circ (\mathsf{k} \Yup \mathsf{s}, \circledast\!\bowtie\!\boxast)\Uparrow$$

◇

Proof

$$\begin{aligned}
& (\mathsf{k}, \circledast)\Uparrow \Yup (\mathsf{s}, \boxast)\Uparrow \\
= & \qquad \llbracket\ \Uparrow\text{; pairs}\ \rrbracket \\
& ((\!|\mathsf{k}, \circledast|\!)\!* \Yup (\!|\mathsf{s}, \boxast|\!)\!*) \circ \mathsf{subtrees} \\
= & \qquad \llbracket\ \text{Lemma 54}\ \rrbracket \\
& (\ll\!* \Yup \gg\!*) \circ ((\!|\mathsf{k}, \circledast|\!) \Yup (\!|\mathsf{s}, \boxast|\!))\!* \circ \mathsf{subtrees} \\
= & \qquad \llbracket\ \text{Corollary 59}\ \rrbracket \\
& (\ll\!* \Yup \gg\!*) \circ (\!|\mathsf{k} \Yup \mathsf{s}, \circledast\!\bowtie\!\boxast|\!)\!* \circ \mathsf{subtrees} \\
= & \qquad \llbracket\ \Uparrow\ \rrbracket \\
& (\ll\!* \Yup \gg\!*) \circ (\mathsf{k} \Yup \mathsf{s}, \circledast\!\bowtie\!\boxast)\Uparrow
\end{aligned}$$

♡

61. Lemma

$$((\mathsf{f}, \oplus, \otimes)\Downarrow \parallel (\mathsf{g}, \boxplus, \boxtimes)\Downarrow) \circ (\ll\!*\Yup\gg\!*) \quad = \quad (\ll\!*\Yup\gg\!*) \circ (\mathsf{f} \parallel \mathsf{g}, \oplus\!\bowtie\!\boxplus, \otimes\!\bowtie\!\boxtimes)\Downarrow$$

◇

Proof

$$
\begin{array}{ll}
 & ((\mathsf{f},\oplus,\otimes)\Downarrow \parallel (\mathsf{g},\boxplus,\boxtimes)\Downarrow)\circ(\ll\!*\curlywedge\gg\!*) \\
= & \qquad \llbracket\ \Downarrow;\ \text{pairs}\ \rrbracket \\
 & (\llparenthesis \mathsf{f},\oplus,\otimes\rrparenthesis * \parallel \llparenthesis \mathsf{g},\boxplus,\boxtimes\rrparenthesis *)\circ \mathsf{paths}^2 \circ(\ll\!*\curlywedge\gg\!*) \\
= & \qquad \llbracket\ \text{pairs; Theorem 31; pairs}\ \rrbracket \\
 & (\llparenthesis \mathsf{f},\oplus,\otimes\rrparenthesis * \parallel \llparenthesis \mathsf{g},\boxplus,\boxtimes\rrparenthesis *)\circ(\ll\!**\curlywedge\gg\!**)\circ \mathsf{paths} \\
= & \qquad \llbracket\ \text{Lemma 54}\ \rrbracket \\
 & (\llparenthesis \mathsf{f},\oplus,\otimes\rrparenthesis * \parallel \llparenthesis \mathsf{g},\boxplus,\boxtimes\rrparenthesis *)\circ(\ll\!*\curlywedge\gg\!*)\circ(\ll\!*\curlywedge\gg\!*)*\circ \mathsf{paths} \\
= & \qquad \llbracket\ \text{Corollary 55}\ \rrbracket \\
 & (\ll\!*\curlywedge\gg\!*)\circ(\llparenthesis \mathsf{f},\oplus,\otimes\rrparenthesis \parallel \llparenthesis \mathsf{g},\boxplus,\boxtimes\rrparenthesis)*\circ(\ll\!*\curlywedge\gg\!*)*\circ \mathsf{paths} \\
= & \qquad \llbracket\ \text{pairs; Corollary 58}\ \rrbracket \\
 & (\ll\!*\curlywedge\gg\!*)\circ(\llparenthesis \mathsf{f}\circ\ll,\ \oplus\circ\overset{\leftarrow}{\ll},\ \otimes\circ\overset{\leftarrow}{\ll}\rrparenthesis \curlywedge \llparenthesis \mathsf{g}\circ\gg,\ \boxplus\circ\overset{\leftarrow}{\gg},\ \boxtimes\circ\overset{\leftarrow}{\gg}\rrparenthesis)*\circ \mathsf{paths} \\
= & \qquad \llbracket\ \text{Corollary 59}\ \rrbracket \\
 & (\ll\!*\curlywedge\gg\!*)\circ\llparenthesis \mathsf{f}\parallel \mathsf{g},\ (\oplus\circ\overset{\leftarrow}{\ll})\boxdot(\boxplus\circ\overset{\leftarrow}{\gg}),\ (\otimes\circ\overset{\leftarrow}{\ll})\boxdot(\boxtimes\circ\overset{\leftarrow}{\gg})\rrparenthesis *\circ \mathsf{paths} \\
= & \qquad \llbracket\ \text{observation concerning } \boxdot \text{ and } \bowtie\ \rrbracket \\
 & (\ll\!*\curlywedge\gg\!*)\circ\llparenthesis \mathsf{f}\parallel \mathsf{g},\ \oplus\bowtie\boxplus,\ \otimes\bowtie\boxtimes\rrparenthesis *\circ \mathsf{paths} \\
= & \qquad \llbracket\ \Downarrow\ \rrbracket \\
 & (\ll\!*\curlywedge\gg\!*)\circ(\mathsf{f}\parallel \mathsf{g},\ \oplus\bowtie\boxplus,\ \otimes\bowtie\boxtimes)\Downarrow
\end{array}
$$

♡

Finally, we can prove the original theorem.

Proof (of Theorem 53) We introduce some abbreviations for the various subexpressions involved:

$$
\begin{array}{lll}
\mathsf{u}_0 & = & (\mathsf{k},\circledast)\Uparrow \\
\mathsf{u}_1 & = & (\mathsf{s},\boxast)\Uparrow
\end{array}
$$

$$
\begin{aligned}
\mathsf{u} &= (\mathsf{k} \curlywedge \mathsf{s}, \circledast\!\bowtie\!\boxast)\Uparrow \\
\mathsf{d}_0 &= (\mathsf{g}, \oplus, \otimes)\Downarrow \\
\mathsf{d}_1 &= (\mathsf{q}, \boxplus, \boxtimes)\Downarrow \\
\mathsf{d} &= (\mathsf{g} \parallel \mathsf{q}, \oplus\!\bowtie\!\boxplus, \otimes\!\bowtie\!\boxtimes)\Downarrow
\end{aligned}
$$

Now,

$$
\begin{aligned}
& (\mathsf{f}{*} \circ \mathsf{d}_0 \circ \mathsf{h}{*} \circ \mathsf{u}_0)\ \hat{\curlyvee}_{\odot}\ (\mathsf{p}{*} \circ \mathsf{d}_1 \circ \mathsf{r}{*} \circ \mathsf{u}_1) \\
= \quad & \llbracket\ \hat{\ };\curlyvee\ \rrbracket \\
& \odot{*} \circ \curlyvee \circ (\mathsf{f}{*} \circ \mathsf{d}_0 \circ \mathsf{h}{*} \circ \mathsf{u}_0) \curlywedge (\mathsf{p}{*} \circ \mathsf{d}_1 \circ \mathsf{r}{*} \circ \mathsf{u}_1) \\
= \quad & \llbracket\ \text{pairs}\ \rrbracket \\
& \odot{*} \circ \curlyvee \circ (\mathsf{f}{*} \parallel \mathsf{p}{*}) \circ (\mathsf{d}_0 \parallel \mathsf{d}_1) \circ (\mathsf{h}{*} \parallel \mathsf{r}{*}) \circ (\mathsf{u}_0 \curlywedge \mathsf{u}_1) \\
= \quad & \llbracket\ \text{Lemma 60}\ \rrbracket \\
& \odot{*} \circ \curlyvee \circ (\mathsf{f}{*} \parallel \mathsf{p}{*}) \circ (\mathsf{d}_0 \parallel \mathsf{d}_1) \circ (\mathsf{h}{*} \parallel \mathsf{r}{*}) \circ ({\ll}{*} \curlywedge {\gg}{*}) \circ \mathsf{u} \\
= \quad & \llbracket\ \text{Corollary 55}\ \rrbracket \\
& \odot{*} \circ \curlyvee \circ (\mathsf{f}{*} \parallel \mathsf{p}{*}) \circ (\mathsf{d}_0 \parallel \mathsf{d}_1) \circ ({\ll}{*} \curlywedge {\gg}{*}) \circ (\mathsf{h} \parallel \mathsf{r}){*} \circ \mathsf{u} \\
= \quad & \llbracket\ \text{Lemma 61}\ \rrbracket \\
& \odot{*} \circ \curlyvee \circ (\mathsf{f}{*} \parallel \mathsf{p}{*}) \circ ({\ll}{*} \curlywedge {\gg}{*}) \circ \mathsf{d} \circ (\mathsf{h} \parallel \mathsf{r}){*} \circ \mathsf{u} \\
= \quad & \llbracket\ \text{Corollary 55}\ \rrbracket \\
& \odot{*} \circ \curlyvee \circ ({\ll}{*} \curlywedge {\gg}{*}) \circ (\mathsf{f} \parallel \mathsf{p}){*} \circ \mathsf{d} \circ (\mathsf{h} \parallel \mathsf{r}){*} \circ \mathsf{u} \\
= \quad & \llbracket\ \curlyvee \circ ({\ll}{*} \curlywedge {\gg}{*}) = \mathsf{id}\ \rrbracket \\
& \odot{*} \circ (\mathsf{f} \parallel \mathsf{p}){*} \circ \mathsf{d} \circ (\mathsf{h} \parallel \mathsf{r}){*} \circ \mathsf{u}
\end{aligned}
$$

♡

Moreover, if the two original fixes were efficient, then so is the new one: if $(\mathsf{g}, \oplus, \otimes)$ inverts to $(\mathsf{g}, \oslash, \obslash)$ and $(\mathsf{q}, \boxplus, \boxtimes)$ inverts to $(\mathsf{q}, \boxslash, \boxbslash)$, then $(\mathsf{g} \parallel \mathsf{q}, \ominus\!\bowtie\!\boxplus, \otimes\!\bowtie\!\boxtimes)$ inverts to $(\mathsf{g} \parallel \mathsf{q}, \oslash\!\bowtie\!\boxslash, \obslash\!\bowtie\!\boxbslash)$.

Bracket matching

An example of the application of Theorem 53 is the so-called *bracket matching* problem: given a 'balanced' string of brackets, the function mbs (for 'match brackets') replaces every bracket with the length of the 'phrase' in that string of which it forms an endpoint. For example

$$\mathsf{mbs}.\text{“\{\{\}\{\}\}\{\}”} \quad = \quad [6,2,2,2,2,6,2,2]$$

A balanced string is one that can be reduced to the empty string by repeatedly erasing from it all occurrences of the substring “{}” ; the language of balanced bracket strings is what a formal language theorist would call a 'Dyck language' (Illingworth et al., 1990). An arbitrary string that has been reduced in this way will consist of a sequence of closing brackets followed by a sequence of opening brackets, and so is completely determined by the pair of numbers giving the lengths of these two sequences. Thus, we define bracket reduction to be the catamorphism $(\!|\mathsf{cat}\colon \mathsf{f}, \odot|\!)$, where

$$\begin{aligned} \mathsf{f}.\text{‘\{’} &= (0,1) \\ \mathsf{f}.\text{‘\}’} &= (1,0) \\ (a,b) \odot (c,d) &= (a + (c \mathbin{\dot{-}} b), (b \mathbin{\dot{-}} c) + d) \end{aligned}$$

where $\dot{-}$, pronounced 'monus', is subtraction bounded below by 0 . The predicate bal , which holds precisely of balanced strings, can now be defined by

$$\mathsf{bal} \quad = \quad \langle (0,0) = \rangle \circ (\!|\mathsf{f}, \odot|\!)$$

If the bracket a in the string $x \mathbin{+\!\!+} \Box.a \mathbin{+\!\!+} y$ is an opening bracket, then 'the phrase of which it forms an endpoint' is the shortest balanced non-empty initial segment of $\Box.a \mathbin{+\!\!+} y$; because the input $x \mathbin{+\!\!+} \Box.a \mathbin{+\!\!+} y$ to the problem is itself balanced, this phrase exists (this is a property of Dyck languages). Similarly, if a is a closing bracket then the corresponding phrase is the shortest balanced non-empty tail segment of

$x \mathbin{+\!\!+} \Box.a$. Denote the functions that return these two phrases by mo and mc respectively; we have

$$\begin{array}{lcl} \mathsf{mo} & = & \downarrow_{\#}/ \circ \mathsf{bal}\triangleleft \circ \mathsf{inits} \\ \mathsf{mc} & = & \downarrow_{\#}/ \circ \mathsf{bal}\triangleleft \circ \mathsf{tails} \end{array}$$

For our purposes, the function $p\triangleleft$ returns a *bag* consisting of those elements of its argument that satisfy the predicate p . Bags are given by the type definition

$$\mathsf{bag}.A \quad = \quad \varnothing \mid \text{⟅}.A \mid \mathsf{bag}.A \uplus \mathsf{bag}.A$$

modulo the laws that $\uplus$ is associative and commutative and has unit $\varnothing$. The function $p\triangleleft$ is then given by

$$p\triangleleft \quad = \quad (\!|\mathsf{cat}\colon p?, \uplus|\!) \qquad \text{where} \quad p?a = \begin{cases} \text{⟅}.a & \text{if} \quad p.a \\ \varnothing & \text{otherwise} \end{cases}$$

If the operator $\Box$ is associative and commutative and has unit e , we write the bag catamorphism $(\!|\mathsf{bag}\colon e, \mathsf{id}, \Box|\!)$ as $\Box/$ for brevity. In this case, the associative and commutative operator $\downarrow_{\#}$ returns the shorter of its two arguments. Since it is always presented with different-length arguments here, the complication of choosing between equal-length arguments can be avoided; however, what of its unit? Let us augment the range and domain of $\downarrow_{\#}$ with a 'fictional element' ω , and make it the unit of $\downarrow_{\#}$. This is the value returned by mo (respectively, mc) if its argument has no balanced initial (respectively, tail) segment, which is the case—for our example $x \mathbin{+\!\!+} \Box.a \mathbin{+\!\!+} y$ —if it is applied to $\Box.a \mathbin{+\!\!+} y$ when a is a '}' (respectively, to $x \mathbin{+\!\!+} \Box.a$ when a is a '{'). With this knowledge, we can define the function mb which matches one bracket:

$$\mathsf{mb}.(x \mathbin{+\!\!+} \Box.a, \Box.a \mathbin{+\!\!+} y) \quad = \quad \mathsf{mc}.(x \mathbin{+\!\!+} \Box.a) \downarrow_{\#} \mathsf{mo}.(\Box.a \mathbin{+\!\!+} y)$$

One of the arguments to $\downarrow_{\#}$ here will be ω , and the result will be the other argument. The original problem is then

$$\begin{array}{lcl} \mathsf{mbs} & = & (\# \circ \mathsf{mb})* \circ (\mathsf{inits}\ \hat{\curlyvee}\ \mathsf{tails}) \\ & = & \#* \circ (\mathsf{mc}* \circ \mathsf{inits}\ \hat{\curlyvee}_{\downarrow_{\#}}\ \mathsf{mo}* \circ \mathsf{tails}) \end{array}$$

If we can express mc and mo as cat list catamorphisms, then mbs is an instance of the fix sums problem and we can solve it in logarithmic parallel time with the parallel fix algorithm.

Let us focus on mo for a while. As it stands, it is not a catamorphism: it returns “{}” for both the inputs “{}” and “{}{”, yet should return different results for the inputs “{}” $+\!\!+$ “}” and “{}{” $+\!\!+$ “}”. Therefore, we introduce the function mor, for ‘match open bracket with remainder’, defined by:

$$\mathsf{mor.x} \quad = \quad (\mathsf{mo.x}, \mathsf{mo.x} \neg \mathsf{x})$$

The operator $\neg$, pronounced ‘drop prefix’, satisfies $\mathsf{x} \neg (\mathsf{x} +\!\!+ \mathsf{y}) = \mathsf{y}$ if $\mathsf{x} \neq \omega$. We make the convention that ω is a left unit of $\neg$, so that $\mathsf{mor.x} = (\omega, \mathsf{x})$ when $\mathsf{mo.x} = \omega$. Now, mor is injective, and so certainly is a catamorphism; moreover,

$$\mathsf{mo} \quad = \quad \ll \circ\ \mathsf{mor}$$

If we manipulate $\mathsf{mo.(x} +\!\!+ \mathsf{y)}$, we see that

$$\begin{array}{ll} & \mathsf{mo.(x} +\!\!+ \mathsf{y)} \\ = & \qquad \llbracket\ \mathsf{mo}\ \rrbracket \\ & \downarrow_{\#}/ \cdot \mathsf{bal}\lhd \cdot \mathsf{inits} \cdot (\mathsf{x} +\!\!+ \mathsf{y}) \\ = & \qquad \llbracket\ \mathsf{inits}\ \rrbracket \\ & \downarrow_{\#}/ \cdot \mathsf{bal}\lhd \cdot (\mathsf{inits} \cdot \mathsf{x} +\!\!+ \langle \mathsf{x} +\!\!+ \rangle * \mathsf{inits} \cdot \mathsf{y}) \\ = & \qquad \llbracket\ \text{promotion}\ \rrbracket \\ & \downarrow_{\#}/ \cdot \mathsf{bal}\lhd \cdot \mathsf{inits} \cdot \mathsf{x} \ \downarrow_{\#}\ \downarrow_{\#}/ \cdot \mathsf{bal}\lhd \cdot \langle \mathsf{x} +\!\!+ \rangle * \cdot \mathsf{inits} \cdot \mathsf{y} \\ = & \qquad \llbracket\ \mathsf{mo};\ \mathsf{p}\lhd \circ \mathsf{f}* = \mathsf{f}* \circ (\mathsf{p} \circ \mathsf{f})\lhd \quad \text{(Bird, 1987)}\ \rrbracket \\ & \mathsf{mo} \cdot \mathsf{x} \ \downarrow_{\#}\ \downarrow_{\#}/ \cdot \langle \mathsf{x} +\!\!+ \rangle * \cdot (\mathsf{bal} \circ \mathsf{x} +\!\!+)\lhd \cdot \mathsf{inits} \cdot \mathsf{y} \end{array}$$

$$
\begin{array}{ll}
= & [\![\text{ wish: } x\mathbin{+\!\!+} \text{ distributes over } \downarrow_{\#} \ ]\!] \\
 & \mathsf{mo}\cdot x \downarrow_{\#} x \mathbin{+\!\!+} {\downarrow_{\#}}/\cdot(\mathsf{bal} \circ x\mathbin{+\!\!+})\triangleleft\cdot\mathsf{inits}\cdot y
\end{array}
$$

The wish, that $x\mathbin{+\!\!+}$ distributes over $\downarrow_{\#}$, is fulfilled on different-length lists if ω is a right zero of $\mathbin{+\!\!+}$.

This still does not produce an efficient algorithm, since we have no quick way of computing ${\downarrow_{\#}}/\cdot(\mathsf{bal} \circ x\mathbin{+\!\!+})\triangleleft\cdot\mathsf{inits}\cdot y$ from $\mathsf{mor}.y$. (We have *a* way, since mor is injective, but it consists essentially of reconstructing y and starting from scratch.) The solution is to perform a data refinement; this refinement is to write the string $\mathsf{mo}.y \neg y$ in the form $t \mathbin{+\!\!+} u_0 \mathbin{+\!\!+} v$, and then to write

$$
\begin{array}{lcl}
t & = & u_{-a} \mathbin{+\!\!+} \text{“\}”} \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} u_{-1} \mathbin{+\!\!+} \text{“\}”} \\
v & = & \text{“\{”} \mathbin{+\!\!+} u_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \text{“\{”} \mathbin{+\!\!+} u_b
\end{array}
$$

with each u_i balanced or empty; each u_i is then further subdivided:

$$u_i \quad = \quad u_{i,0} \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} u_{i,k_i-1}$$

such that each $u_{i,j}$ is non-empty, balanced and of minimum length. Every bracket string has a unique representation of this form: the u_i are the 'maximal balanced segments', the $u_{i,j}$ the 'minimal balanced segments', the remaining characters are the 'unmatched brackets' and the string $t \mathbin{+\!\!+} u_0 \mathbin{+\!\!+} v$ reduces to (a, b). The details of the refinement are beyond the scope of this example—we have already strayed a long way from tree algorithms—but suffice it to say that if the lists u_i are kept as balanced binary trees, and all strings are labelled with their length, then $\mathsf{mor}.(x \mathbin{+\!\!+} y)$ can be computed from $\mathsf{mor}.x$ and $\mathsf{mor}.y$ in logarithmic time sequentially.

Retracing our steps, we get mo—and symmetrically, mc—as the composition of a projection and a catamorphism,

$$
\begin{array}{lcl}
\mathsf{mo} & = & \ll \circ (\!| f, \oplus |\!) \\
\mathsf{mc} & = & \ll \circ (\!| g, \otimes |\!)
\end{array}
$$

for some f , g , and some $\oplus$ and $\otimes$ which take logarithmic effort; then

$$\begin{array}{ll}
 & \mathsf{mbs} \\
= & \qquad [\![\ \mathsf{mbs} \ ]\!] \\
 & \#* \circ (\mathsf{mc}* \circ \mathsf{inits} \ \hat{\curlyvee}_{\downarrow_\#} \ \mathsf{mo}* \circ \mathsf{tails}) \\
= & \qquad [\![\ \mathsf{mc} \text{ and } \mathsf{mo} \text{ as catamorphisms} \ ]\!] \\
 & \#* \circ (\ll* \circ (\!| \mathsf{f}, \oplus |\!)* \circ \mathsf{inits} \ \hat{\curlyvee}_{\downarrow_\#} \ \ll* \circ (\!| \mathsf{g}, \otimes |\!)* \circ \mathsf{tails}) \\
= & \qquad [\![\ \curlyvee \ ]\!] \\
 & \#* \circ ((\!| \mathsf{f}, \oplus |\!)* \circ \mathsf{inits} \ \hat{\curlyvee}_{\downarrow_\# \circ <^2} \ (\!| \mathsf{g}, \otimes |\!)* \circ \mathsf{tails})
\end{array}$$

which is an instance of the fix sums problem. It can be evaluated using the parallel fix algorithm in $\log^2.n$ time in parallel on input of size n .

This is not the fastest algorithm known—there are algorithms for matching brackets that run in logarithmic time on $n \,/\, \log.n$ processors (Dekel and Sahni, 1983b; Bar-On and Vishkin, 1985; Gibbons and Rytter, 1988; Gibbons and Ziani, 1991)—but it is interesting since it provides further evidence of 'the power of parallel prefix' (Kruskal et al., 1985; Blelloch, 1990).

6 Drawing trees tidily

The *tree drawing* problem is to produce a mapping from elements of a tree to points in the plane; this mapping should correspond to a drawing that is in some sense 'tidy'. We do not directly formalize the concept of tidiness; instead, we simply identify some properties enjoyed by 'tidy' drawings, and use these properties to determine a formal specification of the problem. This collection of properties constitutes our indirect definition of tidiness.

First we consider drawings of binary trees. The quest for an efficient algorithm will lead us naturally to a combination of upwards and downwards accumulations. We then generalize this solution to rose trees, which, it turns out, present some extra complications over binary trees.

We make the simplification of ignoring the labels of the tree, so that the drawing depends only on the structure. Thus, the source type for the drawing functions will be one of unlabelled trees.

The first property that we observe of tidy drawings is that all the elements at a given depth in a tree have the same y-coordinate in the drawing. That is, the y-coordinate is determined completely by the depth of an element, and the problem reduces to that of finding the x-coordinates. This gives us the type of bdraw, the function which draws a binary tree—its result is a homogeneous moo tree labelled with x-coordinates:

$$\mathsf{bdraw} \quad \in \quad \mathsf{umtree} \rightarrow \mathsf{hmtree}.\mathbb{D}$$

where coordinates range over $\mathbb{D}$, the type of distances. We require that $\mathbb{D}$ include the number 1, and be closed under subtraction (and hence also under addition) and halving. Sets satisfying these conditions include the reals, the rationals, and the rationals with finite binary expansions, the last being the smallest such set. We exclude discrete sets such as the integers, as Supowit and Reingold (1983) have shown that the problem is NP-hard with such coordinates.

Tidy drawings are also regular, in the sense that the drawing of a subtree is independent of the context in which it appears. Informally, this means that the drawings of children can be committed to (separate pieces of) paper before considering their parent; the drawing of the parent is constructed by translating the drawings of the children. In symbols,

$$\mathsf{bdraw}\cdot(\mathsf{x} \pm \mathsf{y}) \quad = \quad \langle +\mathsf{r}\rangle * \mathsf{bdraw}\cdot\mathsf{x} \;\pm_{\mathsf{a}}\; \langle +\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{y}$$

for some a, r and s.

Tidy drawings also exhibit no left to right bias. In particular, a parent should be centred over its children; we also specify that the root of a tree should be given x-coordinate 0. Hence, $\mathsf{r} + \mathsf{s}$ and a in the above equation should both be 0, as should the position given to the only element of a singleton tree:

$$\begin{array}{rcl} \mathsf{bdraw}\cdot\triangle & = & \triangle\cdot 0 \\ \mathsf{bdraw}\cdot(\mathsf{x} \pm \mathsf{y}) & = & \langle -\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{x} \;\pm_0\; \langle +\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{y} \end{array}$$

for some s. Indeed, a tidy drawing will have the left child to the left of the right child, and so $\mathsf{s} > 0$.

This lack-of-bias property implies that a tree and its mirror image produce drawings which are reflections of each other. That is, if we define the function brev, which reverses a binary tree, by

$$\mathsf{brev} \quad = \quad (\!| \triangle, \oplus |\!) \qquad \text{where} \quad \mathsf{u} \oplus_{\mathsf{a}} \mathsf{v} = \mathsf{v} \pm_{\mathsf{a}} \mathsf{u}$$

and denote unary negation by $-$, then we also require

$$\mathsf{bdraw} \circ \mathsf{brev} \quad = \quad -* \circ \mathsf{brev} \circ \mathsf{bdraw}$$

The fourth criterion is that in a tidy drawing, elements do not collide, or even get too close together: pictures of children do not overlap, and no two elements on the same level are less than one unit apart.

Finally, a tidy drawing should be as narrow as possible, given the above constraints. Supowit and Reingold (1983) showed that narrowness and regularity cannot be satisfied together—there are trees whose narrowest drawings can only be produced by drawing identical subtrees with different shapes—and so one of the two criteria must be made subordinate to the other; we choose to retain the regularity property, since it will lead us to a catamorphic solution.

These last two properties determine s, the distance through which children are translated: it should be the smallest distance that does not cause violation of the fourth criterion. Suppose the operator $\obar$, when given two drawings of trees, returns the width of the narrowest part of the gap between the trees; if the drawings overlap, this distance will be negative. The drawings should be moved apart or together to make this distance 1, that is,

$$s \quad = \quad (1 - \mathsf{bdraw}\cdot x \obar \mathsf{bdraw}\cdot y) \div 2$$

All that remains to be done to complete the specification is to formalize this description of $\obar$.

Levelorder traversal

When we introduced the zip operator Υ in Chapter 2, we defined it only on pairs of equal-length lists. We now extend the definition in two different ways to cover pairs of different-length (cons) lists. These two extensions are 'short zip', which we write $\overline{\Upsilon}$, and 'long zip', written $\overset{|}{\Upsilon}$; they differ in that the length of the result of a short zip is the length of its shorter argument, whereas the length of the result of a long zip is the length of the longer. For example,

$$\begin{array}{lcl} [a, b] \, \overline{\Upsilon}_{\oplus} \, [c, d, e] & = & [a \oplus c, b \oplus d] \\ [a, b] \, \overset{|}{\Upsilon}_{\oplus} \, [c, d, e] & = & [a \oplus c, b \oplus d, e] \end{array}$$

From the result of the long zip, we see that the $\oplus$ must be an endo-operator, that is, it must have type $A \parallel A \rightarrow A$. This is not necessary for short zip, but we do not use the general case in this chapter.

The two zips are given formally by the equations

$$\begin{array}{rcl}
\Box.a \;\Upsilon_{\oplus}\; \Box.b & = & \Box.(a \oplus b) \\
\Box.a \;\Upsilon_{\oplus}\; (b \cdot\!: y) & = & \Box.(a \oplus b) \\
(a \cdot\!: x) \;\Upsilon_{\oplus}\; \Box.b & = & \Box.(a \oplus b) \\
(a \cdot\!: x) \;\Upsilon_{\oplus}\; (b \cdot\!: y) & = & (a \oplus b) \cdot\!: (x \;\Upsilon_{\oplus}\; y) \\[6pt]
\Box.a \;\overset{+}{\Upsilon}_{\oplus}\; \Box.b & = & \Box.(a \oplus b) \\
\Box.a \;\overset{+}{\Upsilon}_{\oplus}\; (b \cdot\!: y) & = & (a \oplus b) \cdot\!: y \\
(a \cdot\!: x) \;\overset{+}{\Upsilon}_{\oplus}\; \Box.b & = & (a \oplus b) \cdot\!: x \\
(a \cdot\!: x) \;\overset{+}{\Upsilon}_{\oplus}\; (b \cdot\!: y) & = & (a \oplus b) \cdot\!: (x \;\overset{+}{\Upsilon}_{\oplus}\; y)
\end{array}$$

They share many properties, but we use two in particular.

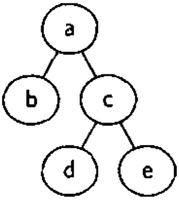
62. Fact $x \;\Upsilon_{\oplus}\; y$ and $x \;\overset{+}{\Upsilon}_{\oplus}\; y$ can both be evaluated with $\#.x \downarrow \#.y$ applications of $\oplus$. ◇

63. Lemma If f is $(\oplus, \otimes)$ promotable then $f*$ is both $(\Upsilon_{\oplus}, \Upsilon_{\otimes})$ and $(\overset{+}{\Upsilon}_{\oplus}, \overset{+}{\Upsilon}_{\otimes})$ promotable. ◇

We use long zip to define *levelorder traversal* of homogeneous binary trees. This is given by the function $\mathsf{levels} \in \mathsf{hmtree}.A \rightarrow \mathsf{cons{\cdot}cat{\cdot}}A$:

$$\mathsf{levels} \;=\; (\!|\Box \circ \Box, \oplus|\!) \qquad \text{where} \quad x \oplus_a y = \Box.a \cdot\!: (x \;\overset{+}{\Upsilon}_{+\!\!+}\; y)$$

For example, the levelorder traversal of the five element tree



is $[[\mathsf{a}], [\mathsf{b}, \mathsf{c}], [\mathsf{d}, \mathsf{e}]]$.

We can at last define the operator $\varobar$ on pictures, in terms of levelorder traversal: it is given by

$$\mathsf{p} \varobar \mathsf{q} \quad = \quad (\!|\mathsf{id}, \downarrow|\!) \cdot ((\!|\mathsf{id}, \downarrow|\!) * \mathsf{levels.q} \ \Upsilon_{-} \ (\!|\mathsf{id}, \uparrow|\!) * \mathsf{levels.p})$$

The catamorphisms $(\!|\mathsf{id}, \downarrow|\!)$ and $(\!|\mathsf{id}, \uparrow|\!)$ return the least and the greatest elements of a list, respectively. If u and v are levels at the same depth in p and q, then $(\!|\mathsf{id}, \uparrow|\!).\mathsf{u}$ and $(\!|\mathsf{id}, \downarrow|\!).\mathsf{v}$ are the rightmost point of u and the leftmost point of v, and so $(\!|\mathsf{id}, \downarrow|\!).\mathsf{v} - (\!|\mathsf{id}, \uparrow|\!).\mathsf{u}$ is the width of the gap at this level. Clearly, $\mathsf{p} \varobar \mathsf{q}$ is the minimum of these widths.

This completes the specification of $\varobar$, and hence of bdraw:

$$\mathsf{bdraw} \quad = \quad (\!|\triangle.0, \oplus|\!)$$

where

$$\begin{array}{lcll} \mathsf{p} \oplus \mathsf{q} & = & \langle -\mathsf{s} \rangle * \mathsf{p} \ \pm_0 \ \langle +\mathsf{s} \rangle * \mathsf{q} & \quad \text{where} \quad \mathsf{s} = (1 - \mathsf{p} \varobar \mathsf{q}) \div 2 \\ \mathsf{p} \varobar \mathsf{q} & = & (\!|\mathsf{id}, \downarrow|\!) \cdot ((\!|\mathsf{id}, \downarrow|\!) * \mathsf{levels.q} \ \Upsilon_{-} \ (\!|\mathsf{id}, \uparrow|\!) * \mathsf{levels.p}) & \end{array}$$

This specification is executable, but requires quadratic effort. We now derive a linear algorithm to satisfy it.

A downwards accumulation

We note that a major source of inefficiency in the program we have just developed is the occurrence of the two maps in the definition of $\oplus$. Intuitively, we have to shift the drawings of two children when assembling the drawing of their parent, and then to shift the whole lot once more when drawing the grandparent. This is because we are directly computing the absolute position of every element. If instead we were to compute the *relative* positions of each parent with respect to its children, these repeated translations would not occur; a second pass—a downwards accumulation—can fix the absolute positions by accumulating relative positions.

Suppose the function rootrel satisfies

$$\begin{aligned}\mathsf{rootrel}\cdot\triangle\cdot\mathsf{a} &= 0\\ \mathsf{rootrel}\cdot(\mathsf{x}\pm_{\mathsf{a}}\mathsf{y}) &= (\mathsf{a}-\mathsf{root}\cdot\mathsf{x})\,\square\,(\mathsf{root}\cdot\mathsf{y}-\mathsf{a})\end{aligned}$$

for some idempotent operator $\square$. The idea is that rootrel determines the position of a parent relative to its children, given the drawing of the parent. That is, if we define the function sep by

$$\mathsf{sep} = \mathsf{rootrel}\circ\mathsf{bdraw}$$

then

$$\begin{aligned}\mathsf{sep}\cdot\triangle &= 0\\ \mathsf{sep}\cdot(\mathsf{x}\pm\mathsf{y}) &= (1-\mathsf{bdraw}\cdot\mathsf{x}\obar\mathsf{bdraw}\cdot\mathsf{y})\div 2\end{aligned}$$

and

$$\begin{aligned}\mathsf{bdraw}\cdot(\mathsf{x}\pm\mathsf{y}) &= \langle -\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{x}\ \pm_{0}\ \langle +\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{y}\\ &\qquad\text{where}\quad \mathsf{s}=\mathsf{sep}\cdot(\mathsf{x}\pm\mathsf{y})\end{aligned}$$

Now, applying sep to each subtree gives the relative position of every parent; define the function rel by

$$\mathsf{rel} = \mathsf{sep}*\circ\mathsf{subtrees}$$

so that

$$\begin{aligned}\mathsf{rel}\cdot\triangle &= \triangle\cdot 0\\ \mathsf{rel}\cdot(\mathsf{x}\pm\mathsf{y}) &= \mathsf{rel}\cdot\mathsf{x}\pm_{\mathsf{sep}\cdot(\mathsf{x}\pm\mathsf{y})}\mathsf{rel}\cdot\mathsf{y}\end{aligned}$$

This gives us the first 'pass', computing the position of every parent relative to its children; how can we get from this to the absolute position of every element? We need a function abs satisfying the condition

$$\mathsf{abs}\circ\mathsf{rel} = \mathsf{bdraw}$$

On leaves, this condition reduces to

$$\begin{array}{ll} & \mathsf{abs}\cdot\mathsf{rel}\cdot\triangle \;=\; \mathsf{bdraw}\cdot\triangle \\ \equiv & \qquad [\![\ \mathsf{rel},\ \mathsf{bdraw}\ ]\!] \\ & \mathsf{abs}\cdot\triangle\cdot 0 \;=\; \triangle\cdot 0 \end{array}$$

while on branches, we require

$$\begin{array}{ll} & \mathsf{abs}\cdot\mathsf{rel}\cdot(\mathsf{x} \pm \mathsf{y}) \;=\; \mathsf{bdraw}\cdot(\mathsf{x} \pm \mathsf{y}) \\ \equiv & \qquad [\![\ \mathsf{rel},\ \mathsf{bdraw};\ \text{let } \mathsf{s} = \mathsf{sep}\cdot(\mathsf{x} \pm \mathsf{y})\ ]\!] \\ & \mathsf{abs}\cdot(\mathsf{rel}\cdot\mathsf{x} \pm_{\mathsf{s}} \mathsf{rel}\cdot\mathsf{y}) \;=\; \langle -\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{x} \pm_0 \langle +\mathsf{s}\rangle * \mathsf{bdraw}\cdot\mathsf{y} \end{array}$$

These requirements are satisfied if

$$\begin{array}{rcl} \mathsf{abs}\cdot\triangle\cdot\mathsf{a} & = & \triangle\cdot 0 \\ \mathsf{abs}\cdot(\mathsf{x} \pm_{\mathsf{a}} \mathsf{y}) & = & \langle -\mathsf{a}\rangle * \mathsf{abs}\cdot\mathsf{x} \ \pm_0\ \langle +\mathsf{a}\rangle * \mathsf{abs}\cdot\mathsf{y} \end{array}$$

that is, if

$$\mathsf{abs} \;=\; (!0, \bar{-}, +)\Downarrow$$

Thus, we have

$$\mathsf{bdraw} \;=\; \mathsf{abs} \circ \mathsf{rel}$$

where

$$\begin{array}{rcl} \mathsf{rel} & = & \mathsf{sep}* \circ \mathsf{subtrees} \\ \mathsf{abs} & = & (!0, \bar{-}, +)\Downarrow \end{array}$$

This is still inefficient, partly because rel is upwards but not an upwards accumulation. We show next how to compute rel quickly.

An upwards accumulation

We want to find an efficient way of computing the function rel satisfying

$$\mathsf{rel} \;=\; \mathsf{sep}* \circ \mathsf{subtrees}$$

where

$$\begin{aligned} \mathsf{sep}\cdot\triangle &= 0 \\ \mathsf{sep}\cdot(\mathsf{x} \pm \mathsf{y}) &= (1 - \mathsf{bdraw}\cdot\mathsf{x} \obar \mathsf{bdraw}\cdot\mathsf{y}) \div 2 \end{aligned}$$

We observed that rel is upwards but not an upwards accumulation, because sep is not a catamorphism—more information than the separations of the grandchildren is needed in order to compute the separation of the children. How much more information is needed?

We note that each level of a picture is sorted. Therefore,

$$\begin{aligned} \llparenthesis \mathsf{id}, \downarrow \rrparenthesis * \circ \mathsf{levels} &= \mathsf{head}* \circ \mathsf{levels} \\ \llparenthesis \mathsf{id}, \uparrow \rrparenthesis * \circ \mathsf{levels} &= \mathsf{last}* \circ \mathsf{levels} \end{aligned}$$

and so

$$\mathsf{p} \obar \mathsf{q} = \mathsf{right}\cdot\mathsf{p} \boxbar \mathsf{left}\cdot\mathsf{q}$$

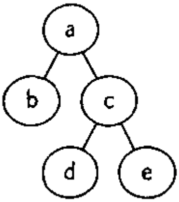
where

$$\begin{aligned} \mathsf{left} &= \mathsf{head}* \circ \mathsf{levels} \\ \mathsf{right} &= \mathsf{last}* \circ \mathsf{levels} \end{aligned}$$

and

$$\mathsf{u} \boxbar \mathsf{v} = \llparenthesis \mathsf{id}, \downarrow \rrparenthesis.(\mathsf{v} \curlyvee_{-} \mathsf{u})$$

Intuitively, left and right return the 'contours' of a drawing; for example, applying the function $\mathsf{left} \curlywedge \mathsf{right}$ to the tree



produces the pair of lists $([\mathsf{a}, \mathsf{b}, \mathsf{d}], [\mathsf{a}, \mathsf{c}, \mathsf{e}])$. These contours are exactly the extra information needed to make sep a catamorphism.

To show this, we need to show first that sep can be computed from the contours, and second, that computing the contours is a catamorphism. Define the function contours by

$$\mathsf{contours} \quad = \quad (\mathsf{left} \curlywedge \mathsf{right}) \circ \mathsf{bdraw}$$

Suppose the function f satisfies

$$\begin{aligned} \mathsf{f}.(\square.\mathsf{a}, \square.\mathsf{a}) \quad &= \quad 0 \\ \mathsf{f}.(\mathsf{a} \mathbin{\cdot\!:} \mathsf{x}, \mathsf{a} \mathbin{\cdot\!:} \mathsf{y}) \quad &= \quad (\mathsf{a} - \mathsf{head}.\mathsf{x}) \mathbin{\Box} (\mathsf{head}.\mathsf{y} - \mathsf{a}) \end{aligned}$$

on pairs of lists with the same head; then, as a short calculation shows,

$$\mathsf{f} \circ (\mathsf{left} \curlywedge \mathsf{right}) \quad = \quad \mathsf{rootrel}$$

Thus,

$$\begin{aligned} & \mathsf{sep} \\ = \quad & \qquad \llbracket \ \mathsf{sep} \ \rrbracket \\ & \mathsf{rootrel} \circ \mathsf{bdraw} \\ = \quad & \qquad \llbracket \ \mathsf{f} \circ (\mathsf{left} \curlywedge \mathsf{right}) = \mathsf{rootrel} \ \rrbracket \\ & \mathsf{f} \circ (\mathsf{left} \curlywedge \mathsf{right}) \circ \mathsf{bdraw} \\ = \quad & \qquad \llbracket \ \mathsf{contours} \ \rrbracket \\ & \mathsf{f} \circ \mathsf{contours} \end{aligned}$$

so indeed, sep can be computed from contours. Moreover, contours is catamorphic: since

$$\mathsf{sep}.(\mathsf{x} \mathbin{\not\pm} \mathsf{y}) \quad = \quad (1 - ({\gg} \cdot \mathsf{contours} \cdot \mathsf{x} \mathbin{\boxbar} {\ll} \cdot \mathsf{contours} \cdot \mathsf{y})) \div 2$$

and head and last are $(+\!\!+, \ll)$ and $(+\!\!+, \gg)$ promotable, respectively, we can calculate that $\mathsf{contours} = (\!| \, !(\square \cdot 0, \square \cdot 0), \otimes \, |\!)$ where

$$\begin{aligned} (\mathsf{w}, \mathsf{x}) \otimes (\mathsf{y}, \mathsf{z}) \quad = \quad & (0 \mathbin{\cdot\!:} (\langle -\mathsf{s} \rangle * \mathsf{w} \mathbin{\overset{\dagger}{\curlyvee}_{\ll}} \langle +\mathsf{s} \rangle * \mathsf{y}), 0 \mathbin{\cdot\!:} (\langle -\mathsf{s} \rangle * \mathsf{x} \mathbin{\overset{\dagger}{\curlyvee}_{\gg}} \langle +\mathsf{s} \rangle * \mathsf{z})) \\ & \qquad \text{where} \quad \mathsf{s} = (1 - \mathsf{x} \mathbin{\boxbar} \mathsf{y}) \div 2 \end{aligned}$$

Hence,

$$
\begin{array}{ll}
& \mathsf{bdraw} \\
= & \qquad [\![\text{ relative positions }]\!] \\
& \mathsf{abs} \circ \mathsf{rel} \\
= & \qquad [\![\text{ abs, rel }]\!] \\
& (!0, \bar{-}, +)\Downarrow \circ \mathsf{sep}{*} \circ \mathsf{subtrees} \\
= & \qquad [\![\text{ sep in terms of contours }]\!] \\
& (!0, \bar{-}, +)\Downarrow \circ \mathsf{f}{*} \circ \llparenthesis !(\square{\cdot}0, \square{\cdot}0), \otimes \rrparenthesis{*} \circ \mathsf{subtrees} \\
= & \qquad [\![\Uparrow]\!] \\
& (!0, \bar{-}, +)\Downarrow \circ \mathsf{f}{*} \circ (!(\square{\cdot}0, \square{\cdot}0), \otimes)\Uparrow
\end{array}
$$

There are still two sources of inefficiency here. The first is that abs is a paths but not a htaps accumulation, and so takes quadratic effort, and the second is that the operation $\otimes$ takes at least linear effort, resulting in quadratic effort for the upwards accumulation too. We solve these two problems next, producing at last a linear algorithm for drawing trees.

Making the downwards accumulation abs efficient is straightforward: we can use the tupling trick that has worked so well in the past. We note first that as it stands, it *is* inefficient: no operator $\oplus$ satisfies $a + !0{\cdot}b = !0{\cdot}a \oplus b$, and so the triple $(!0, \bar{-}, +)$ is not top-down and the accumulation not a htaps accumulation. However, consider the function

$$\mathsf{ab} \quad = \quad \llparenthesis \mathsf{thread}\colon !0, \bar{-}, + \rrparenthesis \curlywedge \llparenthesis \mathsf{thread}\colon \mathsf{id}, \gg, \gg \rrparenthesis$$

The second component of this fork returns the last (bottom) element of a thread. Clearly, $\ll \circ\, \mathsf{ab} = \llparenthesis !0, \bar{-}, + \rrparenthesis$. Moreover, by Theorem 13, ab is a catamorphism:

$$\mathsf{ab} \quad = \quad \llparenthesis !0 \curlywedge \mathsf{id}, (\bar{-} \circ (\mathsf{id} \parallel \ll)) \curlywedge (\gg \circ \gg), (+ \circ (\mathsf{id} \parallel \ll)) \curlywedge (\gg \circ \gg) \rrparenthesis$$

Finally, ab is top-down: it inverts to $\llparenthesis \mathsf{daerht}: !0 \curlywedge \mathsf{id}, - \parallel \mathsf{id}, + \parallel \mathsf{id} \rrparenthesis$; we therefore have

$$\begin{aligned}
& (!0, \dot{-}, +)\Downarrow \\
= & \qquad [\![\; \Downarrow \;]\!] \\
& \llparenthesis !0, \dot{-}, + \rrparenthesis * \circ \mathsf{paths} \\
= & \qquad [\![\; \llparenthesis !0, \dot{-}, + \rrparenthesis = \ll \circ\, \mathsf{ab} \;]\!] \\
& \ll * \circ\, \mathsf{ab} * \circ\, \mathsf{paths} \\
= & \qquad [\![\; \text{ab is top-down} \;]\!] \\
& \ll * \circ (!0 \curlywedge \mathsf{id}, - \parallel \mathsf{id}, + \parallel \mathsf{id}) \curlywedge
\end{aligned}$$

That removes the first inefficiency; the second is more involved. We have to find an easy way of evaluating the operator $\otimes$ where

$$\begin{aligned}
(w, x) \otimes (y, z) \quad = \quad & (0 \mathbin{\cdot\!:} (\langle -s \rangle * w \overset{\dagger}{\curlyvee}_{<} \langle +s \rangle * y), 0 \mathbin{\cdot\!:} (\langle -s \rangle * x \overset{\dagger}{\curlyvee}_{>} \langle +s \rangle * z)) \\
& \qquad \text{where} \quad s = (1 - x \boxbar y) \div 2
\end{aligned}$$

One way of doing this is with a data refinement, whereby instead of maintaining a list of distances w we maintain the list whose image under the invertible function $\llparenthesis \mathsf{id}, + \rrparenthesis * \circ\, \mathsf{inits}$ is w—that is, a refinement with abstraction function $\llparenthesis \mathsf{id}, + \rrparenthesis * \circ\, \mathsf{inits}$. Under this refinement, the maps can be performed in constant time, since

$$\begin{aligned}
\langle +s \rangle * \circ \llparenthesis \mathsf{id}, + \rrparenthesis * \circ\, \mathsf{inits} \quad = \quad & \llparenthesis \mathsf{id}, + \rrparenthesis * \circ\, \mathsf{inits} \circ \langle s \oplus \rangle \\
& \text{where} \qquad \begin{aligned} s \oplus \square\!\cdot\! a &= \square\!\cdot\!(s + a) \\ s \oplus (a \mathbin{\cdot\!:} x) &= (s + a) \mathbin{\cdot\!:} x \end{aligned}
\end{aligned}$$

The details of the refinement are not our concern here, but they are easily calculated—especially so since the abstraction function is invertible.

The refined $\otimes$ still takes linear effort, but the important observation is that it now takes effort proportional to the length of its *shorter* argument (that is, to the lesser of the common lengths of w and x and the common lengths of y and z, when $\otimes$ is 'called' with arguments (w, x) and (y, z)). This is because the only remaining non-constant effort parts are the three zips, each of which can also be evaluated with effort proportional to the length of their shorter argument. Reingold and Tilford (1981) showed that, if evaluating $h.x \oplus h.y$ from $h.x$ and $h.y$ takes effort proportional to the lesser of the depths of the trees x and y, then the tree catamorphism $h = (\!| f, \oplus |\!)$ can be evaluated with linear effort. Actually, what they show is that if g satisfies

$$\begin{array}{rcl} g \cdot \triangle \cdot a & = & 0 \\ g \cdot (x \mathbin{\underline{\ast}} y) & = & g \cdot x + (\mathrm{depth} \cdot x \downarrow \mathrm{depth} \cdot y) + g \cdot y \end{array}$$

then

$$g \cdot x \quad = \quad \mathrm{elements} \cdot x - \mathrm{depth} \cdot x$$

which can easily be proved by induction. Intuitively, g counts the number of pairs of horizontally adjacent elements in a tree.

So, the refined upwards accumulation can be evaluated with linear effort, as can the map and the downwards accumulation; therefore, we have a linear effort tree drawing algorithm.

Drawing rose trees tidily

We now proceed to generalize the problem to that of drawing rose trees. This latter problem is rather more interesting, because rose trees present complications that do not occur with binary trees.

The specification starts off in the same way as for binary trees. The problem is to find an efficient algorithm for computing a function

$$\mathrm{rdraw} \quad \in \quad \mathrm{urtree} \to \mathrm{hrtree}.\mathbb{D}$$

This corresponds to the first criterion for binary trees. The second criterion is that the drawing is regular, and so the drawing of a parent is assembled from shifted drawings of its children. This is formalized by

$$\mathsf{rdraw}\cdot\prec\cdot x \quad = \quad a \prec \mathsf{position}\cdot(\mathsf{rdraw} * x)$$

for some distance a and function position which satisfies, for some f ,

$$\mathsf{position}\cdot ps \quad = \quad f\cdot ps \ \Upsilon_{\oplus}\ ps \qquad \text{where} \quad a \oplus p = (a+) * p$$

Informally, f takes a list of drawings ps and returns a list of displacements, one per drawing.

The third condition is that the drawing should be unbiased. If the function rrev , which reverses the structure of a rose tree without affecting its elements, is defined by

$$\mathsf{rrev} \quad = \quad (\!|\triangle, \prec \circ\, \mathsf{id} \parallel \mathsf{rev}|\!)$$

where the function rev reverses a snoc list, then this condition is stated

$$\mathsf{rdraw} \circ \mathsf{rrev} \quad = \quad -\!* \circ \mathsf{rrev} \circ \mathsf{rdraw}$$

Informally, the drawing of the reverse of a tree should be the same as the reflection of the drawing of the original tree. In particular, the drawing is rooted at 0

$$\mathsf{root} \circ \mathsf{rdraw} \quad = \quad !0$$

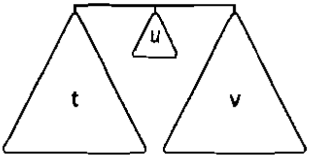
and a parent is centred over its children:

$$\mathsf{midpoint} \circ \mathsf{position} \circ \mathsf{rdraw}* \quad = \quad !0$$

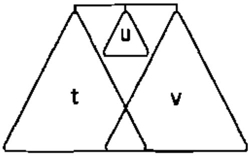
where

$$\begin{aligned}\mathsf{midpoint} \quad = \quad & (\mathsf{root} \circ \mathsf{head})\ \hat{\odot}\ (\mathsf{root} \circ \mathsf{last}) \\ & \qquad \text{where} \quad a \odot b = (a + b) \div 2\end{aligned}$$

The last two criteria are that the elements of the drawing should not collide, and that the drawing should be as narrow as this permits. This is the reason why rose trees are more difficult to draw than binary trees: it is not sufficient to say that the drawings of adjacent children should be as close as possible. For example, consider a rose tree with three children t , u and v , such that t and v are large but u is small:



If the children are positioned with the statutory narrow gap between t and u and between u and v, then t and v may well still collide:



The children that interact may not be adjacent; the condition should instead be that *all* pairs of children are collision-free. We now formalize this condition.

Define the functions twoinits and twotails, from non-empty lists to bags of lists of length at least two, by

$$\begin{array}{lcl} \mathsf{twoinits} & = & \mathsf{two}\triangleleft \circ \mathsf{inits} \\ \mathsf{twotails} & = & \mathsf{two}\triangleleft \circ \mathsf{tails} \end{array}$$

where

$$\mathsf{two} \quad = \quad (\geqslant 2) \circ \#$$

These functions return the (non-empty,) non-singleton inits and tails, respectively, of a list. For example, twoinits.[a, b, c] is $\Lbag [a,b],[a,b,c] \Rbag$, whereas twotails.[a, b, c] is $\Lbag [a,b,c],[b,c] \Rbag$. Both twoinits and twotails return the empty bag when applied to a singleton list. Now define the function twosegs, returning non-empty, non-singleton segments of a list, in terms of these:

$$\mathsf{twosegs} \quad = \quad \uplus/ \circ \mathsf{twotails}* \circ \mathsf{twoinits}$$

(Recall that $\oplus/$ is $\llparenthesis \mathsf{bag}: \mathsf{e}, \mathsf{id}, \oplus \rrparenthesis$, where $\oplus$ has unit e.) For example, $\mathsf{twosegs}.[\mathsf{a}, \mathsf{b}, \mathsf{c}]$ is $\Lbag [\mathsf{a}, \mathsf{b}], [\mathsf{a}, \mathsf{b}, \mathsf{c}], [\mathsf{b}, \mathsf{c}] \Rbag$.

The *endpoints* of a list are its head and its last element:

$$\mathsf{endpoints} \quad = \quad \mathsf{head} \curlywedge \mathsf{last}$$

Now, a list of pictures is 'collision free' if it satisfies the predicate $\mathsf{disjoint}$:

$$\mathsf{disjoint} \quad = \quad \mathsf{all}.(\geqslant 1 \circ \text{⦶}) \circ \mathsf{endpoints}* \circ \mathsf{twosegs}$$

where $\mathsf{all}.\mathsf{p} = \wedge/ \circ \mathsf{p}*$. Here, ⦶ is the analogue for rose trees of the corresponding operator on binary trees: if levelorder traversal of rose trees is given by

$$\mathsf{levels} \quad = \quad \llparenthesis \mathsf{hrtree}: \square \circ \square, \oplus \rrparenthesis \qquad \text{where} \quad \mathsf{a} \oplus \mathsf{xs} = \square \cdot \mathsf{a} \mathbin{:\!\cdot} \llparenthesis \mathsf{id}, \overset{\dagger}{\Upsilon}_{+\!+} \rrparenthesis \cdot \mathsf{xs}$$

then, as before,

$$\begin{aligned} \mathsf{left} \quad &= \quad \llparenthesis \mathsf{id}, \downarrow \rrparenthesis * \circ \mathsf{levels} \\ \mathsf{right} \quad &= \quad \llparenthesis \mathsf{id}, \uparrow \rrparenthesis * \circ \mathsf{levels} \\ \text{◫} \quad &= \quad \llparenthesis \mathsf{id}, \downarrow \rrparenthesis \circ \Upsilon_{\dot{-}} \\ \text{⦶} \quad &= \quad \text{◫} \circ (\mathsf{right} \parallel \mathsf{left}) \end{aligned}$$

Recall the function $\mathsf{position}$, satisfying

$$\mathsf{rdraw} \cdot \prec \cdot \mathsf{x} \quad = \quad 0 \prec \mathsf{position} \cdot (\mathsf{rdraw} * \mathsf{x})$$

The no-collision criterion can now be stated

$$\mathsf{disjoint} \circ \mathsf{position} \circ \mathsf{rdraw}* \quad = \quad !\mathsf{true}$$

This completes the specification of rdraw. We have first to synthesize an executable program from these conditions, and then to make this program efficient.

An algorithm

Although our definition of segments, $\mathsf{twosegs}$, is not the same as Bird's, we can still apply his Segment Decomposition Theorem (Bird, 1987) to the predicate $\mathsf{disjoint}$:

$$
\begin{aligned}
& \mathsf{disjoint} \\
= \quad & \qquad \llbracket \text{ definition } \rrbracket \\
& \mathsf{all}.(\geqslant 1 \circ \obar) \circ \mathsf{endpoints}* \circ \mathsf{twosegs} \\
= \quad & \qquad \llbracket \text{ } \mathsf{twosegs} \text{ } \rrbracket \\
& \mathsf{all}.(\geqslant 1 \circ \obar) \circ \mathsf{endpoints}* \circ \uplus/ \circ \mathsf{twotails}* \circ \mathsf{twoinits} \\
= \quad & \qquad \llbracket \text{ promotion } \rrbracket \\
& \wedge/ \circ (\mathsf{all}.(\geqslant 1 \circ \obar) \circ \mathsf{endpoints}* \circ \mathsf{twotails})* \circ \mathsf{twoinits} \\
= \quad & \qquad \llbracket \text{ letting } \mathsf{delta} = \mathsf{all}.(\geqslant 1 \circ \obar) \circ \mathsf{endpoints}* \circ \mathsf{twotails} \text{ } \rrbracket \\
& \mathsf{all}.\mathsf{delta} \circ \mathsf{twoinits}
\end{aligned}
$$

In particular, a singleton list is always disjoint, because it has no $\mathsf{twoinits}$; indeed, we could say that $\mathsf{disjoint}$ is 'prefix-closed with derivative delta', although this is using the term in a way slightly different from Bird's:

$$
\begin{aligned}
& \mathsf{disjoint}\cdot(\mathsf{ps} \mathbin{:\!\cdot} \mathsf{p}) \\
\equiv \quad & \qquad \llbracket \text{ above } \rrbracket \\
& (\mathsf{all}.\mathsf{delta})\cdot\mathsf{twoinits}\cdot(\mathsf{ps} \mathbin{:\!\cdot} \mathsf{p}) \\
\equiv \quad & \qquad \llbracket \text{ } \mathsf{twoinits} \text{ } \rrbracket \\
& (\mathsf{all}.\mathsf{delta})\cdot\mathsf{two}\triangleleft\cdot\mathsf{inits}\cdot(\mathsf{ps} \mathbin{:\!\cdot} \mathsf{p}) \\
\equiv \quad & \qquad \llbracket \text{ } \mathsf{inits} \text{ } \rrbracket \\
& (\mathsf{all}.\mathsf{delta})\cdot\mathsf{two}\triangleleft\cdot(\mathsf{inits}\cdot\mathsf{ps} \mathbin{:\!\cdot} (\mathsf{ps} \mathbin{:\!\cdot} \mathsf{p}))
\end{aligned}
$$

$$
\begin{array}{ll}
\equiv & \qquad [\![\ \mathsf{two}\cdot(\mathsf{ps} :\cdot \mathsf{p}) = \mathsf{true} \ ]\!] \\
& (\mathsf{all.delta})\cdot(\mathsf{two}\lhd\cdot\mathsf{inits}\cdot\mathsf{ps} \uplus \{\!|\mathsf{ps} :\cdot \mathsf{p}|\!\}) \\
\equiv & \qquad [\![\ \mathsf{twoinits,\ all} \ ]\!] \\
& (\mathsf{all.delta})\cdot\mathsf{twoinits}\cdot\mathsf{ps} \wedge \mathsf{delta}\cdot(\mathsf{ps} :\cdot \mathsf{p}) \\
\equiv & \qquad [\![\ \mathsf{disjoint} \ ]\!] \\
& \mathsf{disjoint}\cdot\mathsf{ps} \wedge \mathsf{delta}\cdot(\mathsf{ps} :\cdot \mathsf{p})
\end{array}
$$

Now, we know that

$$\mathsf{position} \circ \Box \circ \mathsf{rdraw} \quad = \quad \Box \circ \mathsf{rdraw}$$

because an only child must be rooted at the origin. Suppose that $\mathsf{position}$ is a snoc list catamorphism $(\!|\mathsf{snoc}\colon \Box, \boxtimes|\!)$ such that

$$\mathsf{ps} \boxtimes \mathsf{p} \quad = \quad \mathsf{centre}\cdot(\mathsf{ps} :\cdot \langle(\mathsf{ps} \otimes \mathsf{p})+\rangle * \mathsf{p})$$

where

$$\mathsf{centre}\cdot\mathsf{x} \quad = \quad \langle-\mathsf{midpoint}\cdot\mathsf{x}\rangle* * \mathsf{x}$$

The idea here is that $\mathsf{ps} \otimes \mathsf{p}$ gives the distance by which p must be shifted in order that it fit snugly against ps; then centre translates the whole list of children bodily to the left, putting the midpoint at the origin and hence recentering the parent.

If $\mathsf{position}$ is of this form, then the no-collision condition reduces to

$$\mathsf{disjoint}\cdot\mathsf{ps} \quad \Rrightarrow \quad \mathsf{delta}\cdot(\mathsf{ps} \boxtimes \mathsf{p})$$

This follows since a singleton is always disjoint, and because, with this condition, adding another child in the correct position maintains disjointness:

$$\begin{array}{ll}
 & \mathsf{disjoint} \cdot \mathsf{position} \cdot (\mathsf{ps} :\!\cdot\, \mathsf{p}) \\
\equiv & \quad [\![\ \mathsf{position} \ ]\!] \\
 & \mathsf{disjoint} \cdot (\mathsf{position} \cdot \mathsf{ps} \boxtimes \mathsf{p}) \\
\equiv & \quad [\![\ \boxtimes \ ]\!] \\
 & \mathsf{disjoint} \cdot \mathsf{centre} \cdot (\mathsf{position} \cdot \mathsf{ps} :\!\cdot\, \langle (\mathsf{position} \cdot \mathsf{ps} \otimes \mathsf{p}) + \rangle * \mathsf{p}) \\
\equiv & \quad [\![\ \mathsf{disjoint} \text{ is invariant under translation }]\!] \\
 & \mathsf{disjoint} \cdot (\mathsf{position} \cdot \mathsf{ps} :\!\cdot\, \langle (\mathsf{position} \cdot \mathsf{ps} \otimes \mathsf{p}) + \rangle * \mathsf{p}) \\
\equiv & \quad [\![\ \mathsf{disjoint} \text{ is prefix-closed }]\!] \\
 & \mathsf{disjoint} \cdot \mathsf{position} \cdot \mathsf{ps} \wedge \mathsf{delta} \cdot (\mathsf{position} \cdot \mathsf{ps} \boxtimes \mathsf{p}) \\
\equiv & \quad [\![\text{ assuming } \mathsf{disjoint} \cdot \mathsf{qs} \Rrightarrow \mathsf{delta} \cdot (\mathsf{qs} \boxtimes \mathsf{p}) \ ]\!] \\
 & \mathsf{disjoint} \cdot \mathsf{position} \cdot \mathsf{ps}
\end{array}$$

so by the unique extension property,

$$\mathsf{disjoint} \circ \mathsf{position} \quad = \quad !\mathsf{true}$$

This gives us some information about $\boxtimes$, namely that its result should satisfy delta if its left argument is disjoint. A calculation involving $\oplus$ and properties of $\uparrow$, $-$, $\downarrow$ and zips shows that

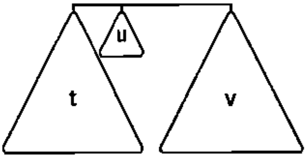
$$\mathsf{delta} \cdot (\mathsf{ps} :\!\cdot\, \mathsf{p}) \quad \equiv \quad (\![\mathsf{id}, \overset{\dagger}{\curlyvee}_{\uparrow}]\!) \cdot (\mathsf{right} * \mathsf{ps}) \boxbar \mathsf{left} \cdot \mathsf{p} \geqslant 1$$

Thus,

$$\begin{array}{ll}
 & \mathsf{delta} \cdot (\mathsf{ps} \boxtimes \mathsf{p}) \\
\equiv & \quad [\![\ \boxtimes \ ]\!] \\
 & \mathsf{delta} \cdot \mathsf{centre} \cdot (\mathsf{ps} :\!\cdot\, \langle (\mathsf{ps} \otimes \mathsf{p}) + \rangle * \mathsf{p}) \\
\equiv & \quad [\![\ \mathsf{delta} \text{ is invariant under translation }]\!] \\
 & \mathsf{delta} \cdot (\mathsf{ps} :\!\cdot\, \langle (\mathsf{ps} \otimes \mathsf{p}) + \rangle * \mathsf{p})
\end{array}$$

$$
\begin{array}{ll}
\equiv & \qquad [\![\text{ above }]\!] \\
 & (\!|\mathsf{id}, \overset{\dagger}{\curlyvee}_{\uparrow}|\!)\cdot(\mathsf{right} * \mathsf{ps}) \boxplus \mathsf{left}\cdot(\langle(\mathsf{ps} \otimes \mathsf{p})+\rangle * \mathsf{p}) \geqslant 1 \\
\equiv & \qquad [\![\; \mathsf{left} \circ \langle \mathsf{a}+\rangle * = \langle \mathsf{a}+\rangle * \circ \mathsf{left};\ \mathsf{u}\boxplus \circ \langle \mathsf{a}+\rangle * = \mathsf{a}+ \circ \mathsf{u}\boxplus \;]\!] \\
 & (\mathsf{ps} \otimes \mathsf{p}) + (\!|\mathsf{id}, \overset{\dagger}{\curlyvee}_{\uparrow}|\!)\cdot(\mathsf{right} * \mathsf{ps}) \boxplus \mathsf{left}\cdot\mathsf{p} \geqslant 1 \\
\equiv & \qquad [\![\text{ arithmetic }]\!] \\
 & (\mathsf{ps} \otimes \mathsf{p}) \geqslant 1 - (\!|\mathsf{id}, \overset{\dagger}{\curlyvee}_{\uparrow}|\!)\cdot(\mathsf{right} * \mathsf{ps}) \boxplus \mathsf{left}\cdot\mathsf{p} \\
\Leftarrow & \qquad [\![\text{ fix } \mathsf{ps} \otimes \mathsf{p} \text{ as small as possible }]\!] \\
 & (\mathsf{ps} \otimes \mathsf{p}) = 1 - (\!|\mathsf{id}, \overset{\dagger}{\curlyvee}_{\uparrow}|\!)\cdot(\mathsf{right} * \mathsf{ps}) \boxplus \mathsf{left}\cdot\mathsf{p}
\end{array}
$$

Now, the list of drawings produced by position is by no means unbiased; each child except the leftmost is packed tightly against its left siblings. For example, position would draw our example rose tree with three children in the form



Therefore, let us rename position to leftwards :

$$\mathsf{leftwards} \quad = \quad (\!|\square, \boxtimes|\!)$$

It is not difficult to show that rdraw is unbiased precisely if position is unbiased. In other words, define the function mirror , which reflects a whole list of drawings, by

$$\mathsf{mirror} \quad = \quad \mathsf{rev} \circ \mathsf{rrev}* \circ -**$$

Then if $\mathsf{rdraw}\cdot\!\prec\!\cdot\mathsf{x} = 0 \prec \mathsf{position}\cdot(\mathsf{rdraw} * \mathsf{x})$, as we have above, then

$$\begin{array}{ll}
 & \mathsf{-* \circ rrev \circ rdraw} \;=\; \mathsf{rdraw \circ rrev} \\
\Leftarrow & \qquad [\![\text{ a short calculation }]\!] \\
 & \mathsf{mirror \circ position} \;=\; \mathsf{position \circ mirror} \\
\equiv & \qquad [\![\text{ mirror is its own inverse }]\!] \\
 & \mathsf{position} \;=\; \mathsf{mirror \circ position \circ mirror}
\end{array}$$

We have just seen, though, that $\mathsf{leftwards}$ does *not* commute with mirror: if we define

$$\mathsf{rightwards} \quad = \quad \mathsf{mirror \circ leftwards \circ mirror}$$

then $\mathsf{rightwards}$, which packs children to the right, is different from $\mathsf{leftwards}$. However, their average *is* unbiased—if we redefine $\mathsf{position}$ so that

$$\mathsf{position} \quad = \quad \mathsf{leftwards} \; \hat{\Upsilon}_{\Upsilon_{\odot}} \; \mathsf{rightwards}$$

then it does commute with mirror, since negation is $(\odot, \odot)$-promotable, and $\odot$ is commutative.

All that we have left to check is that this redefined $\mathsf{position}$ produces disjoint lists of drawings; we showed earlier that $\mathsf{leftwards}$ does, but it is not immediately obvious from this that $\mathsf{position}$ does too. Nevertheless, it is not too arduous to show that the disjointness of rightwards drawings is equivalent to the disjointness of leftwards drawings:

$$\mathsf{disjoint \circ rightwards} \quad = \quad \mathsf{disjoint \circ leftwards}$$

and that disjointness of the means follows from disjointness of the two components:

$$\mathsf{disjoint} \cdot (x \; \Upsilon_{\Upsilon_{\odot}} \; y) \quad \Leftarrow \quad \mathsf{disjoint} \cdot x \wedge \mathsf{disjoint} \cdot y$$

This means that we now have an executable, albeit inefficient, program for rdraw:

$$\mathsf{rdraw} \quad = \quad (\!| \, !(\triangle \cdot 0), \langle 0 \prec \rangle \circ \mathsf{position} \, |\!)$$

$$\begin{aligned}
\mathsf{position} &= \mathsf{leftwards}\ \hat{\Upsilon}_{\Upsilon_{\odot}}\ \mathsf{mirror} \circ \mathsf{leftwards} \circ \mathsf{mirror} \\
\mathsf{leftwards} &= (\![\square, \boxtimes]\!) \\
&\qquad \text{where} \quad \mathsf{ps} \boxtimes \mathsf{p} = \mathsf{centre}\cdot(\mathsf{ps} \mathbin{:\!>} \langle(\mathsf{ps} \otimes \mathsf{p}) +\rangle * \mathsf{p}) \\
\mathsf{centre}\cdot\mathsf{x} &= \langle -(\mathsf{root}\cdot\mathsf{head}\cdot\mathsf{x} \odot \mathsf{root}\cdot\mathsf{last}\cdot\mathsf{x})\rangle * * \mathsf{x} \\
\mathsf{ps} \otimes \mathsf{p} &= 1 - (\![\mathsf{id}, \overset{+}{\Upsilon}_{\uparrow}]\!)\cdot(\mathsf{right} * \mathsf{ps}) \boxbar \mathsf{left}\cdot\mathsf{p}
\end{aligned}$$

An efficient algorithm

Effectively the same optimizations that applied to binary trees can be used for rose trees. Informally, these are that

- drawing should be split into two stages: first, find the position of every child relative to its parent, and second, perform a downwards accumulation to compute the absolute positions
- drawings are 'sorted': the contours are the endpoints of the levels, and if $\mathsf{disjoint}\cdot\mathsf{ps}$ then

  $$(\![\mathsf{id}, \overset{+}{\Upsilon}_{\uparrow}]\!)\cdot(\mathsf{right} * \mathsf{ps}) \quad = \quad (\![\mathsf{id}, \overset{+}{\Upsilon}_{>}]\!)\cdot(\mathsf{right} * \mathsf{ps})$$

  and so $\mathsf{ps} \otimes \mathsf{p}$ depends only on the 'right contour' of ps
- maintain these contours during the first stage, making it an upwards accumulation

Along with the observation that centering need not be performed so often, that is, that

$$\begin{aligned}
\mathsf{leftwards} &= \mathsf{centre} \circ (\![\square, \boxtimes]\!) \\
&\qquad \text{where} \quad \mathsf{ps} \boxtimes \mathsf{p} = \mathsf{ps} \mathbin{:\!>} \langle(\mathsf{ps} \otimes \mathsf{p}) +\rangle * \mathsf{p}
\end{aligned}$$

these optimizations provide us with an efficient algorithm.

Related work

The problem of drawing trees has quite a long and interesting history. Knuth (1968a, 1971b) and Wirth (1976) both present simple algorithms in which the x-coordinate of an element is determined purely by its position in inorder traversal. Wetherell and Shannon (1979) first considered 'aesthetic criteria', but their algorithms all produce biased drawings. Independently of Wetherell and Shannon, Vaucher (1980) gives an algorithm which produces drawings that are simultaneously biased, irregular, and wider than necessary, despite his claims to have 'overcome the problems' of Wirth's simple algorithm. Reingold and Tilford (1981) tackle the problems in Wetherell and Shannon's and Vaucher's algorithms by proposing the criteria concerning bias and regularity; their algorithm is the one derived for binary trees here. Supowit and Reingold (1983) show that it is not possible to satisfy regularity and minimal width simultaneously, and that the problem is NP-hard when restricted to discrete (for example, integer) coordinates. Brüggemann-Klein and Wood (1990) implement Reingold and Tilford's algorithm as macros for the text formatting system $\TeX$.

The more difficult problem of drawing rose trees has had rather less coverage in the literature. Reingold and Tilford (1981) mention them in passing, but make no reference to the difficulty of producing unbiased drawings. Radack (1988) presents the algorithm that we derive here. Walker (1990) uses a slightly different method: he positions children from left to right, but when a child touches against a left sibling other than the nearest one, the extra displacement is apportioned among the intervening siblings.

7 Attribute grammars

Our third and last illustration of the applications of accumulations on trees is provided by evaluation mechanisms for attribute grammars. Attribute grammars were proposed by Knuth (1968b) as a tool for presenting the semantics of programming languages. They arose as an extension of the 'syntax-directed' compilation techniques of the early sixties (Irons, 1961). Using these techniques, the parse tree of a program is *decorated* with *attributes*, the decoration attached to an element of the parse tree representing some aspect of the semantics of the subtree rooted there. In Irons' formulation, the attribute attached to an element depends only on the descendants of that element; Knuth showed that although no extra power is gained by doing so, the description of the semantics of a language can be considerably simplified by allowing attributes to depend on other parts of the parse tree as well. The reader is referred to the comprehensive survey by Deransart et al. (1988) for further information about the history of attribute grammars; their report includes a bibliography of over five hundred items.

Traditionally, an attribute grammar for a context free language is an extension of the grammar which describes the syntax of that language. Each symbol in the grammar has associated with it a number of attributes, and each production in the grammar comes with some rules that give values to some of the attributes attached to symbols appearing in that production, in terms of the values of the other attributes that appear. The attributes are classified into two categories, *inherited* and *synthesized*; inherited attributes are those appearing on the right hand side of the production in which their value is defined, and hence concern the 'children' of the production, whereas synthesized attributes appear on the left, and concern the parents. Irons' syntax-directed translation corresponds to attribute grammars with only synthesized attributes. Intuitively, inherited attributes carry information into a subtree and synthesized attributes carry it back out again; in Knuth's (1971a) words, 'inherited attributes are, roughly speaking, those aspects of meaning which come from the context of a phrase, while synthesized attributes are those aspects which are built up from within the phrase.'

Our view of attribute grammars differs somewhat from this traditional view. We suppose that a tree has been built already, and that the task is to evaluate the attributes of the root of the tree. We make several simplifying assumptions in order to prevent the proliferation of symbols and indices, but none of these significantly affect the mathematics. Throughout this chapter, A is the type of labels of the tree, and I and S the types of inherited and synthesized attributes, respectively.

64. Definition An attribute grammar consists simply of an evaluation rule

$$\oplus \quad \in \quad \mathsf{A} \parallel (\mathsf{I} \parallel \mathsf{S} \parallel \mathsf{S}) \to \mathsf{I} \parallel \mathsf{I} \parallel \mathsf{S}$$

◇

We make the simplification, after Fokkinga et al. (1991), that every element has exactly one inherited and one synthesized attribute, and that all inherited attributes have the same type, as do all synthesized attributes. This entails no loss of generality, since attribute types may be sums of products. We assume also that the tree is homogeneous, and a binary tree at that: rose trees can be treated in an entirely analogous way.

The idea is that the evaluation rule $\oplus$ takes an element a of the tree and a triple $(\mathsf{i}, \mathsf{s}, \mathsf{t})$ of attributes, with i being the inherited attribute of that element and s and t the synthesized attributes of its children; it yields a triple $(\mathsf{j}, \mathsf{k}, \mathsf{u})$, with u the synthesized attribute of that element and j and k the attributes its children will inherit. The simplification that the inherited attributes of the children and the synthesized attribute of the parent depend only on the inherited attribute of the parent and the synthesized attributes of the children is due to Bochmann (1976). To avoid the need for different evaluation rules for leaves and branches, we assume also that there is a 'dummy' inherited attribute ω_I that can be produced as the inherited attribute for the 'children' of a leaf, and a dummy synthesized attribute ω_S, distinct and distinguishable from 'valid' synthesized attributes, that can be used for their synthesized attributes; such an element can always be adjoined to S if none exists already.

The process of attribute evaluation according to an evaluation rule $\oplus$ is performed by the operator $\circledast_\oplus \in (\mathsf{hmtree}.A \parallel I) \rightarrow S$; applied to a pair consisting of a homogeneous binary tree and the inherited attribute of its root, it returns the synthesized attribute of the root of the tree. For the rest of this chapter, we will assume a fixed evaluation rule $\oplus$, and write simply $\circledast$ for attribute evaluation. This operator satisfies

$$\begin{array}{rcllcl} \triangle\cdot a \circledast i & = & \pi_2\cdot(a \oplus (i, \omega_S, \omega_S)) & & & \\ (x \pm_a y) \circledast i & = & \pi_2\cdot u & \text{where}\quad u & = & a \oplus (i, s, t) \\ & & & s & = & x \circledast \pi_0\cdot u \\ & & & t & = & y \circledast \pi_1\cdot u \end{array}$$

Examples of attribute grammars

We consider now a number of examples of attribute grammars. Any homogeneous moo tree catamorphism can be expressed as an attribute grammar with inherited attributes of type $\mathbf{1}$. We have that the catamorphism $(\!|\mathsf{hmtree}: f, \otimes|\!)$ is equal to the evaluation $(\circledast it)$ where

$$\begin{array}{rcll} a \oplus (it, \omega_S, \omega_S) & = & (it, it, f.a) & \\ a \oplus (it, s, t) & = & (it, it, s \otimes_a t) & \quad \text{if} \quad s, t \neq \omega_S \end{array}$$

Upwards accumulations and paths downwards accumulations are catamorphisms, so they can both be written in this way; htaps accumulations, though, are generally not catamorphic, and for these we need to use the inherited attributes. The idea here is that the inherited attribute holds the 'context dependent' part of the accumulation, and the synthesized attribute returns the tree which is the result. We get $(\mathsf{f}, \oslash, \otimes)\curlywedge = \langle\circledast \mathsf{f}\rangle$ where

$$\begin{array}{rcll} \mathsf{a} \oplus (\mathsf{f}, \omega_S, \omega_S) & = & (\omega_I, \omega_I, \triangle\cdot\mathsf{f}\cdot\mathsf{a}) & \\ \mathsf{a} \oplus (\mathsf{f}, \mathsf{x}, \mathsf{y}) & = & (\mathsf{f}.\mathsf{a}\oslash, \mathsf{f}.\mathsf{a}\otimes, \mathsf{x} \pm_{\mathsf{f},\mathsf{a}} \mathsf{y}) & \text{if } \mathsf{x}, \mathsf{y} \neq \omega_S \end{array}$$

Informally, information flows down through the inherited attributes and then back up through the synthesized ones.

Another example involving both inherited and synthesized attributes is the function rank from Chapter 5, which replaces every element of an unlabelled tree with the number of leaves to the left of and including that element in inorder traversal. Here, the inherited attribute of an element gives the number of leaves to the left of but excluding the subtree rooted at that element; this gives us $\mathsf{rank} = \langle\circledast 0\rangle$ where

$$\begin{array}{rcll} \mathsf{it} \oplus (\mathsf{i}, \omega_S, \omega_S) & = & (\omega_I, \omega_I, \triangle.(1 + \mathsf{i})) & \\ \mathsf{it} \oplus (\mathsf{i}, \mathsf{x}, \mathsf{y}) & = & (\mathsf{i}, (\!|\mathsf{id}, \otimes|\!).\mathsf{x}, \mathsf{x} \pm_{(\!|\mathsf{id},\otimes|\!).\mathsf{x}} \mathsf{y}) & \text{if} \quad \mathsf{x}, \mathsf{y} \neq \omega_S \end{array}$$

where $\mathsf{u} \otimes_{\mathsf{a}} \mathsf{v} = \mathsf{v}$. The inherited attribute of the right child here, $(\!|\mathsf{id}, \otimes|\!).\mathsf{x}$, is the number of leaves to the left of and including subtree x in inorder traversal; the inherited attribute of a right child depends on the synthesized attribute of its left sibling, and information flows from left to right in the same way that it does for a depth-first search. Most of the applications of attribute grammars to programming languages involve dependencies like this, because of the close correspondence between the hierarchical structure of the parse tree and the linear structure of the program it represents.

Our final example is Bird's 'repmin' problem (Bird, 1984b); the problem here is to replace every element of a tree of numbers with the smallest element in that tree. For this we require one inherited and two synthesized attributes; the first synthesized attribute attached to an element records the smallest element of the tree originally rooted there, the inherited attribute gives the smallest element of the whole tree, and the second synthesized attribute contains the tree of minimum values which is the result. Attribute evaluation is given by

$$\begin{array}{rcll} a \oplus (i, (\infty, x), (\infty, y)) & = & (\omega_I, \omega_I, (a, \triangle.i)) & \\ a \oplus (i, (m, x), (n, y)) & = & (i, i, (m \downarrow a \downarrow n, x \pm_i y)) & \text{if} \quad m, n \neq \infty \end{array}$$

with 'dummy' synthesized attribute ∞. The function repmin itself is given by

$$\mathsf{repmin}.t \;=\; x \quad \text{where} \quad (m, x) = t \circledast m$$

Note that the second synthesized attribute depends on the inherited attribute, which depends on the first synthesized attribute, which depends only on the original tree. In fact, the crux of this problem is to evaluate repmin in a single pass over the tree; we see shortly how to do this for any attribute evaluation.

Circularity

The meticulous reader will have noticed that the definition we gave for attribute evaluation is circular:

$$\begin{array}{rclcrcl} (x \pm_a y) \circledast i & = & \pi_2 \cdot u & \quad \text{where} & u & = & a \oplus (i, s, t) \\ & & & & s & = & x \circledast \pi_0 \cdot u \\ & & & & t & = & y \circledast \pi_1 \cdot u \end{array}$$

so u depends on s and t, which themselves depend on u. In the examples we have given, the circularity disappeared because there was an ordering on the attributes—or on their components—that respected the data dependencies. Jazayeri et al. (1975) have shown that, in general, the check for circularity is inherently exponential; indeed, it was one of the first naturally occurring problems to be shown so.

Heavy emphasis has traditionally been placed on the various possible evaluation orders for attributes, and on restrictions to the evaluation rules that make particular orders valid. This problem can be avoided altogether if the evaluation rule is seen as inducing a collection of equations, and the attribute evaluator is seen as an equation solver: the equations form a program in a lazy functional language, and as Johnsson (1987) says,

> *The lazy evaluator has taken over the job normally done by the special purpose attribute evaluation machinery. Normally in other attribute grammar systems the order in which the attributes are evaluated is determined at evaluator-generation time. In our scheme this order is implicitly determined by the lazy evaluator at run time. The order is entirely determined by the data dependencies, and may vary depending on the order in which the values of the various attributes are demanded.*

Under this view, circular dependencies correspond to mutually recursive equations (Chirica and Martin, 1979; Mayoh, 1981). Farrow (1986) and Johnsson (1987) both observe that such grammars can be useful; Johnsson concludes that 'the only practical road open to us seems to be to detect the circularities at run-time; fortunately, though, this can be done at very little extra cost', at least on the G-machine, the context of Johnsson's paper.

We sidestep the issue completely, following the lead of the majority of the literature on attribute grammars; we treat recursion in the definition of $\circledast$ in the same way that we treat recursion elsewhere, namely, we assume that it is 'sensible'.

Attribute evaluations as catamorphisms

In general, an attribute grammar evaluation $(\circledast i) \in \mathsf{hmtree}.A \rightarrow S$ is not a catamorphism, because it depends on evaluations of the children using different inherited attributes. However, many people (Chirica and Martin, 1979; Jourdan, 1984; Katayama, 1984; Johnsson, 1987; Fokkinga et al., 1991) have shown that the curried evaluation $(\circledast)$ with type $\mathsf{hmtree.A} \longrightarrow (\mathsf{I} \rightarrow \mathsf{S})$ *is* a catamorphism. This observation is similar to the one we exploited in Chapter 5 in order to apply the parallel prefix algorithm to a snoc list catamorphism. We have

$$\begin{aligned} \langle\circledast\rangle\cdot\triangle\cdot a &= \pi_2 \circ \langle a\oplus\rangle \circ (id \curlywedge\, !\omega_S \curlywedge\, !\omega_S) \\ \langle\circledast\rangle\cdot(x \pm_a y) &= \pi_2 \circ h \end{aligned}$$

where

$$\begin{aligned} h.i \quad = \quad u \qquad \text{where} \quad u &= a \oplus (i, s, t) \\ s &= \langle\circledast\rangle.x.(\pi_0\cdot u) \\ t &= \langle\circledast\rangle.y.(\pi_1\cdot u) \end{aligned}$$

That is,

$$\langle\circledast\rangle\cdot(x \pm_a y) \quad = \quad \langle\circledast\rangle\cdot x \otimes_a \langle\circledast\rangle\cdot y$$

where

$$(f \otimes_a g)\cdot i \quad = \quad \pi_2\cdot u \qquad \text{where} \quad u = a \oplus (i, f\cdot\pi_0\cdot u, g\cdot\pi_1\cdot u)$$

A little manipulation of function arguments allows this to be simplified to

$$f \otimes_a g \quad = \quad \pi_2 \circ h \qquad \text{where} \quad h = \langle a\oplus\rangle \circ (id \curlywedge (f \circ \pi_0 \circ h) \curlywedge (g \circ \pi_1 \circ h))$$

Looking back at the examples of attribute grammars that we gave earlier, we are reminded that any tree catamorphism can be written as an attribute grammar with purely synthesized attributes; hence, any attribute grammar can be rewritten to involve only synthesized attributes, returning a function from the original inherited attribute of the root to the original synthesized attribute. As Johnsson shows, it is precisely this construction that gives a single pass solution to Bird's repmin problem.

Attribute evaluations as accumulations

Attribute evaluation is conventionally understood to mean evaluation of a single attribute, the synthesized attribute of the root of the parse tree;

all the other attributes are 'intermediate results' and are of no further interest. For most applications, and in particular for one-off compilation, this is exactly what is required; once the translation of part of a program has been constructed, the translations of subexpressions are no longer needed. However, for some applications we are interested in the intermediate results as well; for example, incremental compilers and structure editors such as the Cornell Synthesizer Generator (Reps and Teitelbaum, 1984, 1989) make use of these intermediate results in order to avoid having to recompile parts of a program that remain unchanged. For such applications, we would like attribute evaluation to return the whole tree of attributes, not just the synthesized attribute of the root.

We have seen that the sectioned evaluation $\langle\circledast\rangle$ is a catamorphism; therefore, $\langle\circledast\rangle * \circ \mathsf{subtrees}$ is an upwards accumulation,

$$\langle\circledast\rangle * \circ \mathsf{subtrees} \quad \in \quad \mathsf{hmtree.A} \rightarrow \mathsf{hmtree.(I} \rightarrow \mathsf{S)}$$

yielding a tree of inherited-to-synthesized-attribute functions. This is nearly but not quite enough to allow us to compute all the attributes in the tree—given the inherited attribute of the root, we can certainly find the synthesized attribute of the root, but what will the inherited attributes of the children be? We have thrown that information away.

We want a slightly different attribute evaluation that returns the whole triple of type $\mathsf{I} \parallel \mathsf{I} \parallel \mathsf{S}$, consisting of the inherited attributes of the children as well as the synthesized attribute of the parent. To this end, we define the *complete attribute evaluator*

$$\boxed{*}_{\oplus} \quad \in \quad (\mathsf{hmtree.A} \parallel \mathsf{I}) \longrightarrow (\mathsf{I} \parallel \mathsf{I} \parallel \mathsf{S})$$

which we will abbreviate to $\boxed{*}$. The intention is that $\pi_2 \circ \boxed{*} = \circledast$.

The definition is straightforward, given the definition of $\circledast$:

$$\triangle{\cdot}\mathsf{a} \; \boxed{*} \; \mathsf{i} \quad = \quad \mathsf{a} \oplus (\mathsf{i}, \omega_S, \omega_S)$$

$$\begin{array}{lclccl} (x \pm_a y) \boxast i & = & u \quad \text{where} & u & = & a \oplus (i, s, t) \\ & & & s & = & \pi_2 \cdot (x \boxast \pi_0 \cdot u) \\ & & & t & = & \pi_2 \cdot (y \boxast \pi_1 \cdot u) \end{array}$$

Again, the sectioned evaluator $\langle \boxast \rangle$ is a catamorphism,

$$\langle \boxast \rangle \quad \in \quad \mathsf{hmtree}.A \rightarrow (I \rightarrow I \parallel I \parallel S)$$

from which we can construct an upwards accumulation,

$$\langle \boxast \rangle * \circ \mathsf{subtrees} \quad \in \quad \mathsf{hmtree}.A \rightarrow \mathsf{hmtree}.(I \rightarrow I \parallel I \parallel S)$$

yielding a tree of inherited-attribute-to-triple functions.

The operator $\odot$ that takes the result of this upwards accumulation and the inherited attribute of the root of the tree, and returns the tree with every element replaced with its $I \parallel I \parallel S$ triple, satisfies

$$\begin{array}{rcl} \triangle \cdot f \odot i & = & \triangle \cdot f \cdot i \\ (x \pm_f y) \odot i & = & (x \odot \pi_0 \cdot f \cdot i) \pm_{f \cdot i} (y \odot \pi_1 \cdot f \cdot i) \end{array}$$

That is, $\langle \odot i \rangle$ is a htaps downwards accumulation:

$$\langle \odot i \rangle \quad = \quad (\langle \cdot i \rangle, \oslash, \obslash)\text{⅄}$$

where

$$\begin{array}{rcl} a \oslash g & = & g \cdot \pi_0 \cdot a \\ a \obslash g & = & g \cdot \pi_1 \cdot a \end{array}$$

Clearly, there is a strong analogy between complete attribute evaluation and upwards and downwards accumulations: an upwards accumulation is the complete evaluation of an attribute grammar with only synthesized attributes, and a downwards accumulation is the complete evaluation of a grammar with only inherited attributes; moreover, any complete attribute evaluation consists of an upwards accumulation followed by a downwards accumulation.

8 Conclusion

In this thesis we have looked at three different tree algebras, namely, *moo trees*, *rose trees* and *hip trees*. Hip trees are an original contribution, though we did not make much use of them in this thesis. They intuitively form a partial algebra, in that some of the terms do not obviously correspond to trees. We showed in Chapter 2 how to avoid the introduction of partial functions by demonstrating that the 'tree-like' subset of the terms of the algebra is consistent with the intuitive model; this means that the algebra can remain total, and we can simply ignore the terms that do not correspond to objects in our model.

Each of the algebras we presented came with a class of structure-respecting functions called catamorphisms, determined completely by the definition of the algebra. Each also came with classes of structure-preserving functions called accumulations; these did not come for free from the definitions. Of these accumulations, we defined first the notion of an upwards function, being a function mapped over the subtrees of a tree; upwards functions are exactly those functions that pass information up through a tree from the leaves towards the root. We observed that upwards functions need be neither catamorphic nor efficient; this led us to define upwards accumulations, a special case in which the function being mapped over the subtrees is a catamorphism. Upwards accumulations are both catamorphic and efficient.

The development of downwards accumulations was rather more interesting, because it presented some problems that did not occur with upwards accumulations. We started by defining the paths of a hip tree as a hip tree of hip trees; then we defined a downwards function as a function mapped over the paths of a tree, and a downwards accumulation as a catamorphism mapped over the paths. We discovered that some functions that we expected to be accumulations were not, because of the restrictions imposed by the representation of paths as hip trees; this led to the definition of the free algebra of threads to represent paths. We then discovered that this caused downwards accumulation to be no longer catamorphic, so we redefined it in terms of moo trees of threads; this gave us a catamorphic but not generally efficient downwards accumulation. Finally, we defined another representation of paths, daerhts, which provided an efficient but not generally catamorphic downwards accumulation; we showed under what conditions these last two classes would coincide, to produce catamorphic and efficient downwards accumulations.

The remaining three chapters of the thesis provided a number of examples to illustrate the applicability of upwards and downwards accumulations to algorithms about trees. In Chapter 5, we derived the parallel prefix algorithm for the prefix sums problem; this turned out to consist of an upwards accumulation followed by a downwards accumulation, both accumulations being efficient and catamorphic. We then showed how the prefix sums problem encompassed non-associative as well as associative sums. We also presented an algorithm for the suffix sums problem, and showed how prefix and suffix sums could be calculated together. We used the resulting 'fix sums' algorithm to derive a solution to the bracket matching problem.

In Chapter 6 we derived algorithms for drawing binary and rose trees. The specification of the binary tree drawing problem was executable, taking quadratic effort in the size of the tree. We discovered that splitting it into two phases—an upwards accumulation that computed the relative positions of each parent with respect to its children, followed by a downwards accumulation that fixed the absolute position of each child by accumulating the relative positions of its ancestors—produced an algorithm requiring only linear effort. The same procedure worked for drawing rose trees, except that there the problem of producing unbiased drawings was more difficult; we solved it by 'averaging' the drawings of a tree and its mirror image.

Finally, in Chapter 7, we discussed the evaluation of attributes according to an attribute grammar. We showed the well-known result that this evaluation can be performed in a single pass by constructing a catamorphism yielding a function from the inherited attribute of the root of the tree. This catamorphism could be generalized to an upwards accumulation yielding a tree of functions; we showed that this tree of functions could in turn be evaluated with a downwards accumulation to produce the whole tree decorated with attributes. Again we discovered the pattern of an upwards accumulation followed by a downwards accumulation.

It is the identification of these upwards and downwards accumulations as commonly-occurring patterns of computation that is the most significant contribution of this thesis; the number of problems to which they can be applied testifies to their importance. The next most significant aspect of the material presented here is the algebraic approach taken to algorithm design using these accumulations; other people, as we discuss below, have introduced ideas similar to our upwards and downwards accumulations, but to the best of our knowledge the exploitation of their algebraic properties is original.

Accumulations provide a valuable method of abstraction, as the various examples that we have given show. For example, O'Donnell's derivation of the parallel prefix algorithm commences in a similar fashion to ours, but his result consists in effect of a tree of processors passing messages around in parallel. Had accumulations been available to him as a tool, he might have been able to split the solution into two phases, making its structure much clearer. Now, splitting the parallel prefix algorithm into two phases is not a new idea, and it does not require accumulations; one of the clearest published explanations of the parallel prefix algorithm is by Blelloch (1989):

> *The technique consists of two sweeps of the tree, an up sweep and a down sweep, and requires* $2\log.n$ *steps. The values to be scanned start at the leaves of the tree. On the up sweep, each unit executes* $\oplus$ *on its two children units and passes the sum to its parent. Each unit also keeps a copy of the value from its left child in its memory. On the down sweep, each unit passes to its left child the value from its parent, and passes to its right child* $\oplus$ *applied to its parent and the value stored in the memory (this value originally came from the left child). After the down sweep, the values at the leaves are the result of the scan.*

However, this description could be made clearer still by phrasing it in terms of accumulations. Blelloch's account is *procedural*, describing the actions that each process performs; the construction of the 'invariant' is left to the reader. In contrast, the initial characterizations of accumulations as catamorphisms mapped over the 'generators' $\mathsf{subtrees}$, paths and htaps are *declarative* descriptions, giving the invariants and omitting the method of achieving them. A declarative description makes the properties of the accumulations much clearer. For example, the composition of an accumulation $\llparenthesis \mathsf{f} \rrparenthesis * \circ \mathsf{gen}$ consisting of catamorphism $\llparenthesis \mathsf{f} \rrparenthesis$ and generator gen with a map $\mathsf{g}*$ is another accumulation, since

$$\begin{array}{ll} & \llparenthesis \mathsf{f} \rrparenthesis * \circ \mathsf{gen} \circ \mathsf{g}* \\ = & \quad [\![\text{ generators are natural transformations }]\!] \\ & \llparenthesis \mathsf{f} \rrparenthesis * \circ \mathsf{g}** \circ \mathsf{gen} \\ = & \quad [\![\text{ Corollary 12, introducing some } \mathsf{f}' \ ]\!] \\ & \llparenthesis \mathsf{f}' \rrparenthesis * \circ \mathsf{gen} \end{array}$$

If we had defined accumulations as single monolithic catamorphisms, as Blelloch describes each phase of the parallel prefix algorithm, this calculation would have been rather less straightforward.

Comparisons

We now discuss the published descriptions of which we are aware of operations similar to our accumulations.

Wile (1973) is concerned with a basis for a general-purpose programming language that exploits the connection between control and data structure; to this end, he defines operations on sequences and on nested sequences, the latter being equivalent to our leaf-labelled rose trees. Two particular operations on these are of interest to us; Wile calls them 'top-down accumulation' and 'bottom-up recursion'. Top-down accumulation **tda** satisfies

$$\begin{aligned} \mathbf{tda}.(e,\oplus).(\triangle\cdot a) &= \triangle\cdot e \\ \mathsf{tda}.(e,\oplus).(\prec\cdot x) &= \prec\cdot(\mathsf{tda}.(\prec\cdot x \oplus e,\oplus) * x) \end{aligned}$$

In effect, this is a downwards accumulation composed with $\mathsf{subtrees}$:

$$\begin{aligned} \mathsf{tda}.(e,\oplus) &= (\mathsf{id},!\mathsf{it})* \circ (e\bar{\oplus},\otimes)\curlywedge \circ \mathsf{subtrees} \\ &\qquad \text{where} \quad u \otimes_n a = u \tilde{\oplus} a \end{aligned}$$

in which $\otimes$ ignores its numeric argument. The seed value—the first component of the argument to tda —that is used for a subtree t_n of the tree will be of the form $t_{n-1} \oplus (t_{n-2} \oplus \cdots \oplus (t_0 \oplus e))$ where t_0 is the whole tree, t_1 a child of t_0 , and so on, and t_{n-1} the parent of t_n . As such, Wile's top-down accumulation is very general, encompassing all our examples of an upwards accumulation followed by a downwards accumulation, but rather unstructured.

Wile defines two 'bottom-up recursions', corresponding loosely to our rose tree catamorphisms; he gives no upwards accumulations. The first of these recursions we would write

$$(\!|\mathsf{lrtree}\colon f,\oplus|\!) \circ (\mathsf{id}\ \hat{\mathrm{Y}}\ \mathsf{depths})$$

which depends on the depths of the elements as well as their values. The second satisfies the equation in f

$$\mathsf{f}\cdot\prec\cdot\mathsf{x} \quad = \quad \prec\cdot\mathsf{x} \oplus (\mathsf{f} * \mathsf{x})$$

and so is a rose tree *paramorphism* (Meertens, 1990) rather than simply a catamorphism.

In many respects, Wile's basis resembles APL (Iverson, 1962). He gives a powerful collection of operators which operate on 'structured' objects as a whole, not just on scalars, and his notation is just as concise. It even shares the greatest fault of APL, in that the rich collection of operators is not matched by a rich collection of laws, and their characterizations are given more procedurally than declaratively.

Myers (1980) discusses a language involving infinite sequences and trees; the latter are functions from lists of numbers ('path specifications') to elements, and correspond to our branch-labelled rose trees except that they may have infinite depth. The only sensible kind of accumulation on these trees is a downwards accumulation; infuriatingly, Myers only hints at a definition: 'the reader should at this point be able to look back at the [rightwards accumulation] operator ... and extend this definition to the analogous tscan operator which moves an accumulation in all directions down a tree'. One can only guess that he means

$$\begin{array}{rcl} \mathsf{tscan}.(\mathsf{e},\oplus).\triangle & = & \triangle\cdot\mathsf{e} \\ \mathsf{tscan}.(\mathsf{e},\oplus).(\mathsf{a} \prec \mathsf{x}) & = & \mathsf{e} \prec (\mathsf{tscan}.(\mathsf{e} \oplus \mathsf{a}, \oplus) * \mathsf{x}) \end{array}$$

Neither does he present any examples using tscan, which might have given us a clue as to his intention.

Dekel and Sahni (1983a) talk about the 'binary tree method', in which a computation consists of a series of alternating upwards and downwards passes over a tree. For the upwards pass, 'we proceed from the leaves to the root solving the subproblem associated with each node', and for the downwards pass 'we proceed from the root to the leaves' similarly; their presentation is distinctly informal. They discuss one-pass uses of the binary tree method—they treat these as catamorphisms rather than as upwards accumulations—and two-pass uses—the parallel prefix algorithm—but give no other examples. Indeed, all the interesting examples that we have been able to find are no more intricate than two passes of Dekel and Sahni's binary tree method; we do not know whether there are natural examples consisting of, say, a downwards accumulation followed by an upwards accumulation.

Turner (1986) talks about 'up-down parsing' of prefix grammars: 'the method works in two passes, the first proceeding up the input intermediate representation [i.e., parse] tree and the second proceeding down', the whole process labelling the parse tree with nonterminals from the grammar. Again, there are no precise definitions of accumulations, nor for that matter a precise definition of anything.

Huang (1985) defines 'type-1' and 'type-2' tree functions which he uses as building blocks for parallel graph algorithms on a mesh-of-trees network. Type-1 tree functions correspond to our downwards accumulation, and type-2 to upwards. According to Huang's definition, the binary operators involved must be associative, but since he does not specify an order in which elements should be combined the operators should really be commutative too; the examples he gives involve only commutative operators such as $+$, $\uparrow$ and $\wedge$. In our notation, these tree functions would be given by

$$\begin{aligned} \mathsf{typeone}.(\mathsf{f},\oplus) &= (\mathsf{f},\oplus,\oplus)\Yup \\ \mathsf{typetwo}.(\mathsf{f},\oplus) &= (\mathsf{f},\oplus)\Uparrow \end{aligned}$$

where $\oplus$ is associative and commutative. In effect, the accumulations are bag catamorphisms mapped over trees of bags.

Gibbons and Rytter (1986) introduce the 'paths problem', which would be stated $(\mathsf{id},\oplus,\oplus)\Downarrow$ in our terminology. The operator $\oplus$ is constrained to be associative, and left and right children are treated the same; in effect, it consists of a cat list 'reduction' $\llparenthesis \mathsf{cat}\colon \mathsf{id},\oplus \rrparenthesis$ mapped over the paths. They actually associate values with the 'edges' of the tree rather than the 'vertices', but the problem is equivalent if each element of the tree holds the value of the 'edge' between it and its parent, and some sensible value is given to the root of the tree. Their definition is precise, but still not as general as either of our downwards accumulations because of its identical treatment of left and right children. For example, this definition excludes the downwards accumulation $(\mathsf{id}, \gg, +)\Downarrow$ forming part of the function rank in Chapter 4, which has different components for left and for right children.

Leiserson and Maggs (1988) describe the 'rootfix' and 'leaffix' problems on homogeneous binary trees. The rootfix problem is identical to Gibbons and Rytter's paths problem. The leaffix problem we would state as

$$(\llbracket \mathsf{cat}\colon \mathsf{id}, \oplus \rrbracket \circ \mathsf{preorder}){*} \circ \mathsf{subtrees}$$

for some associative operator $\oplus$. Again, this is less general than our upwards accumulation, as the catamorphism can depend only on what is discernible of the structure of a tree from its preorder traversal; it precludes, for example, the computations of contours in Chapter 6 and of inherited-to-synthesized-attribute functions in Chapter 7.

Wright (1988) has done some work similar to ours, but on two-dimensional 'arrays' or matrices rather than trees; we referred to him in Chapter 2. Wright talks about upwards, downwards, leftwards and rightwards accumulations on arrays, which perform 'list' accumulations on each column or each row in parallel, but the closest analogues of our upwards and downwards accumulations would be 'northwest', 'northeast', 'southwest' and 'southeast' accumulations, each accumulating towards one corner of the array; the 'northwest subarrays' of an array x, for example, would form an array of arrays, the same shape as x, consisting of all contiguous subarrays of x that share its northwest corner.

Jeuring (1989) considers problems about 'hypotrees' of homogeneous moo trees, which are to subtrees what the contiguous 'segments' of a list are to its inits. He defines the catamorphism chop with type $\mathsf{hmtree}.A \rightarrow \mathsf{cat}.(\mathsf{hmtree}.A)$ by

$$\mathsf{chop} \;=\; (\!|\mathsf{hmtree}: \square \circ \triangle, \oplus|\!) \qquad \text{where} \quad u \oplus_a v = (\pm_a * (u \mathsf{X} v)) \mathbin{+\!\!+} \square\cdot\triangle\cdot a$$

The operator X here returns the cartesian product—a list of pairs—of its arguments. 1nformally, $\mathsf{chop.t}$ is a list giving all the ways of chopping off pairs of children in t. The hypotrees of a tree are given by applying chop to each subtree and flattening the result:

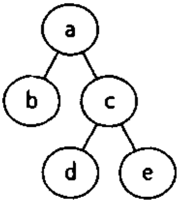
$$\mathsf{hypotrees} \;=\; (\!|\mathsf{cat}: \mathsf{id}, \mathbin{+\!\!+}|\!) \circ \mathsf{postorder} \circ \mathsf{chop}* \circ \mathsf{subtrees}$$

(The postorder traversal is an arbitrary choice; arguably, X, chop and $\mathsf{hypotrees}$ should all return bags.) Jeuring then proceeds to investigate a promotion theorem for hypotrees, that is, conditions under which a function—actually, he only considers catamorphisms—composed with $\mathsf{hypotrees}$ is itself a catamorphism. This work is rather closer to that of Bird (1987) and de Moor (1990) than to ours; there is no notion of 'accumulation' here because the generator $\mathsf{hypotrees}$ does not return a structured object.

This concludes our discussion of work related to our accumulations; we turn now to look at other questions that our work has raised.

Heterogeneous downwards accumulations

We defined paths and downwards accumulations in Chapter 4 only on homogeneous trees. The reasoning behind that decision was as follows. Suppose the tree



is not a homogeneous tree, so that b, d and e are not the same type as a and c. Then the path terminating at c in the tree, a 'homogeneous path', has a different type to the path terminating at d, which is a 'heterogeneous path'. Van der Woude has observed that this is no great obstacle: paths could return a heterogeneous tree, with homogeneous paths at the branches and heterogeneous paths at the leaves. However, downwards accumulation would then have to be defined with two catamorphisms, one for homogeneous and the other for heterogeneous paths. The problems are even worse for htaps accumulations than for paths accumulations, for then the 'heterogeneous daerht' catamorphism applied to the daerht terminating at d need not be a function of the 'homogeneous daerht' catamorphism applied to the thread terminating at c. Sometimes, certainly, we have to say difficult things, but this seems just a bit too difficult.

On a related topic, Backhouse has pointed out that paths could return a tree of lists, rather than of threads, and still be invertible; the information about whether a child is a left child or a right child, although absent from the path, is still held in the position of that path in the resulting tree. However, this information is not 'local', and some functions which would otherwise be downwards accumulations—such as the $(\mathsf{id}, \gg, +)\Downarrow$ we mentioned earlier—would not be a catamorphism mapped over the paths.

Hip trees: a negative result

Hip trees were introduced along with moo and rose trees at the beginning of this thesis. We had hoped that they would provide a natural formulation of paths and of downwards accumulation, but they turned out not to be as useful as we first thought they might. It was the study of hip trees (Gibbons, 1988) that first led us to the notion of downwards accumulation, but in the long run it seems that downwards accumulations are more elegantly expressed in terms of moo trees of threads than of hip trees of hip trees. The complications that hip trees caused in Chapter 4, and indeed the problems of partial algebras that they threatened to introduce in the first place, outweigh any benefit they may have brought us, at least as far as this thesis is concerned. It remains to be seen whether there are any convincing examples of the utility of hip trees—any problems towards whose solutions they lead more naturally than do other types of tree.

Application and apposition

Another experiment, this time a notational one, that we have performed in this thesis is that of using two different application operators, the conventional left associative $.$ from functional programming and Morgan's right associative $\cdot$. It would be nice if it were possible to perform all manipulations at the function level, but—perhaps because we do not yet have the right kinds of combinators—there are many occasions in which object level calculations are clearer. It seems that calculations at the the two different levels are more comfortable using the two different application operators, left associative for functions and right associative for objects.

We have also, privately, tried experimenting with what Meertens (1986) calls 'apposition', a contraction of 'application' and 'composition'. This idea capitalizes on the isomorphism between the types $\mathbf{1} \to \mathsf{A}$ and A : the isomorphisms are given in one direction by the function κ with type $\mathsf{A} \to \mathbf{1} \to \mathsf{A}$ satisfying

$$\kappa.\mathsf{a}.\mathsf{it} \quad = \quad \mathsf{a}$$

and in the other by application to it. If $\mathsf{a} \in \mathsf{A}$ and $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$ then we have

$$\kappa.(\mathsf{f}.\mathsf{a}) \quad = \quad \mathsf{f} \circ (\kappa.\mathsf{a}) \quad \in \quad \mathbf{1} \to \mathsf{B}$$

Apposition consists of making the κ invisible and writing, say, $\mathsf{f} \bullet \mathsf{a}$ for both $\mathsf{f.a}$ and $\mathsf{f} \circ \mathsf{a}$; it is tantamount to identifying the constant a and the function with unit source $\kappa.\mathsf{a}$.

Use of apposition brings with it potential pitfalls, as Meertens was well aware. The type information $\mathbb{1} \to \mathsf{B}$ above is crucial; in general, types A and $\mathsf{C} \to \mathsf{A}$ are *not* isomorphic, and the apposition $\mathsf{f}\bullet\mathsf{a}$ cannot be used unambiguously for non-constant a . Higher-order functions are necessary for constructing an ambiguous example, but the Bird-Meertens formalism thrives on such functions—we have seen them in constructing prefix problems from finite state machines in Chapter 5, and in translations of attribute grammars in Chapter 7, for example.

For this reason, we decided to avoid apposition and stick simply to right associative application. This leaves us with several awkward uses of the unit element it—for example, a leaf of type $\mathsf{bmtree.A}$ should be written $\vartriangle\cdot\mathsf{it}$ rather than just $\vartriangle$—but there seems to be no safe and easy way around this.

Paramorphisms and predecessors

Meertens (1990) gives a general construction for producing the 'predecessors' of a structured object, such as the inits of a list, the subtrees of a tree and so on. Suppose (X, σ) is the initial F-algebra; Meertens' construction proceeds by defining a functor G satisfying

$$\begin{aligned} \mathsf{G.A} &= \mathsf{F.(A} \parallel \mathsf{X)} \\ \mathsf{G.f} &= \mathsf{F.(f} \parallel \mathsf{id)} \end{aligned}$$

The initial G-algebra (Y, τ) is then the algebra of 'substructures' corresponding to X (Meertens himself does not give it a memorable name). On the naturals, this construction gives finite possibly empty lists of naturals, on non-empty snoc lists it gives non-empty snoc lists of non-empty snoc lists, on moo trees, homogeneous moo trees of moo trees, on rose trees, homogeneous rose trees of rose trees, and so on. He goes on to define a 'predecessors' function whether a similar result holds for a language containing our tree accumulations. Leiserson and Maggs (1988) show that their leaffix and rootfix operations, mentioned above, can be implemented in logarithmic time on the 'distributed random access machine', a restricted version of Skillicorn's 'tightly-coupled' processors, and Blelloch (1990) shows the same result for the Connection Machine, another of Skillicorn's four categories. Constant-valence topologies are more awkward: constant-dilation embeddings of trees are not possible on meshes, for example (Skillicorn, 1991). Note that the implementation must give logarithmic time even for degenerate trees that have greater than logarithmic depth; it seems that Leiserson and Maggs' restriction of their leaffix and rootfix operations to associative operators is necessary for this, for the same reason that it is necessary for the parallel prefix algorithm.

It is interesting to note that downwards accumulations that are both catamorphic and efficient, that is, both paths and htaps accumulations, must be expressible using associative operators—this is an extension of the Third Homomorphism Theorem (Barnard et al., 1991), which states that a function expressible as cons and as snoc list catamorphisms is also expressible as a cat list catamorphism—and so we have inadvertently come up with the same associativity condition ourselves.

The third area for further work that we will mention is that of constructing an algebraic theory of directed acyclic graphs ('dags' for short). Trees are just a special case of dags; a tree is a dag in which paths that diverge never rejoin, and a dag is a tree in which some of the elements overlap. It ought to be possible to construct an algebra of dags that reduces to some tree algebra—perhaps the homogeneous rose trees we have studied here, but with bags of children rather than lists—in the special case that the graph is a tree. We might then gain some insight about graph algorithms by applying tree concepts to this graph algebra; similarly, we may learn something new about trees by taking the special case of some graph properties. Bijlsma (1988, 1989) has done

$$\mathsf{preds} \quad = \quad \ll \circ \llparenthesis \mathsf{X}\colon \tau \curlywedge (\sigma \circ \mathsf{F}.\gg) \rrparenthesis \quad \in \quad \mathsf{X} \rightarrow \mathsf{Y}$$

The other half of the catamorphism involved here is the identity:

$$\gg \circ \llparenthesis \mathsf{X}\colon \tau \curlywedge (\sigma \circ \mathsf{F}.\gg) \rrparenthesis \quad = \quad \mathsf{id}$$

This definition gives the predecessors of the number n as the list of numbers $[0, 1, \ldots, \mathsf{n} - 1]$, and preds coincides with our inits on non-empty snoc lists and with subtrees on moo and rose trees.

Meertens' *paramorphisms*, the topic of his paper, are then simply G-catamorphisms composed with preds. Our rightwards, upwards and downwards functions are special cases of paramorphisms, in which the G-catamorphism is a map; accumulations are even more special, because then the function being mapped must itself be a catamorphism.

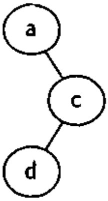
This construction is very elegant, but it is not clear how, or even whether, we can generalize it to cover *downwards* as well as upwards accumulations; in a sense, downwards accumulations run in the 'opposite' direction to the way trees are constructed. For this reason we have chosen to take the more sedate route presented in Chapters 3 and 4.

Further work

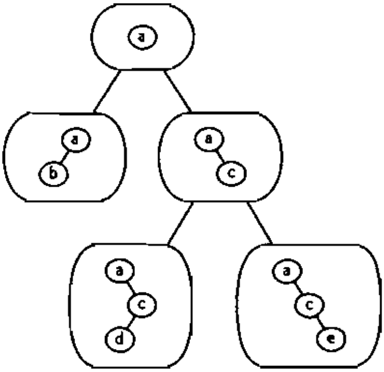
Like most research, this thesis has probably raised more questions than it has answered. We mention three here: a general construction for 'substructures' and accumulations, a universal model of parallel programming, and a theory of directed acyclic graphs.

Meertens' construction for 'substructures' was covered above, and we will not say much more about it; the obvious question is how to apply it to get downwards accumulations on trees.

Skillicorn's architecture-independent programming language was mentioned in Chapter 1; Skillicorn showed that Bird's list operators can be implemented with optimal efficiency on any of the four major classes of parallel architecture. It would certainly be interesting to know form the three-element *path*



Applying paths to the whole five-element tree produces the tree of trees



It seems that paths, and hence downwards accumulation, is expressed most naturally in terms of hip trees, since the paths themselves are trees in which every child is an only child. The type of paths is thus

$$\mathsf{paths} \quad \in \quad \mathsf{htree.A} \rightarrow \mathsf{htree{\cdot}htree{\cdot}A}$$

If we define the operations $\oslash_{\oplus}$ and $\obslash_{\oplus}$ for given $\oplus$ by

$$\begin{array}{rcl} x \oslash_{\oplus} y & = & ((\langle \oplus \mathsf{root}.y \rangle * x) \swarrow y \\ x \obslash_{\oplus} y & = & x \searrow ((\langle \mathsf{root}.x \oplus \rangle * y) \end{array}$$

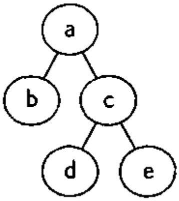
4 Downwards accumulation

In the previous chapter we discussed upwards accumulation, which embodies the notion of passing information up through a tree from the leaves towards the root. We now turn our attention to the inevitable counterpart, *downwards* accumulation, which captures the idea of passing information in the other direction, from the root towards the leaves.

It turns out that downwards accumulation is less straightforward than upwards; the latter follows the structure of the tree, but the former goes against the grain, so to speak. It also transpires that there are two different classes of downwards accumulation; one is catamorphic and the other is efficient, and where they intersect we get accumulations that are both well-behaved and practically useful.

A first attempt

As upwards accumulations arose by considering the function subtrees, which replaces every element of a tree with its descendants, so downward accumulations arise by considering the function paths, which replaces every element of a tree with that element's *ancestors*. The ancestors of an element in a tree form another tree, a tall thin one with that element as its only leaf. For example, the ancestors of the element d in the tree



some work relevant to this; he considers *constructs*, which are functions from partially-ordered sets to elements, just as lists are functions from totally-ordered sets to elements. However, not all constructs can built up from 'elementary' constructs using Bijlsma's constructors; in other words, there are arbitrarily complex irreducible constructs. This suggests that constructs cannot be defined in the Hagino-Malcolm style in the same way that our trees can. Another field that offers some promise here is that of *graph grammars* (Claus et al., 1979; Rozenberg and Salomaa, 1986), which describe sets of graphs in a way that is analogous to the grammatical description of strings.

References

Selim G. Akl (1989). *Design and Analysis of Parallel Algorithms*. Prentice-Hall.

Roland Backhouse (1989). *An exploration of the Bird-Meertens formalism*. In *International Summer School on Constructive Algorithmics, Hollum, Ameland* (1989). Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.

Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude (1990). *A relational theory of types*. Department of Computing Science, Rijksuniversiteit Groningen, and Department of Maths and Computing Science, Technische Universiteit Eindhoven.

Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude (1991). *Relational catamorphisms*. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*. Elsevier.

John Backus (1978). *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*. Communications of the ACM, 21(8):613–641.

John Backus, John Williams, and Edward Wimmers (1990). *An introduction to the programming language FL*. In Turner (1990).

D. T. Barnard, J. P. Schmeiser, and D. B. Skillicorn (1991). *Deriving associative operators for language recognition*. Bulletin of the EATCS, 43:131–139.

Ilan Bar-On and Uzi Vishkin (1985). *Optimal parallel generation of a computation tree form*. ACM Transactions on Programming Languages and Systems, 7(2):348–357.

F. L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brückner, H. Partsch, P. Pepper, and H. Wössner (1979). *Towards a wide-spectrum language to support program specification and development*. In F. L. Bauer and M. Broy, editors, *LNCS 69: Program Construction*, pages 543–552. Springer-Verlag.

A. Bijlsma (1988). *A unified approach to sequences, bags and trees*. Technical Report 88/13, Technische Universiteit Eindhoven.

A. Bijlsma (1989). *Transformational programming and forests*. In van de Snepscheut (1989), pages 157–173.

Richard S. Bird (1984a). *The promotion and accumulation strategies in transformational programming*. ACM Transactions on Programming Languages and Systems, 6(4):487–504. See also (Bird, 1985).

Richard S. Bird (1984b). *Using circular programs to eliminate multiple traversals of data*. Acta Informatica, 21:239–250.

Richard S. Bird (1985). Addendum to *"The promotion and accumulation strategies in transformational programming"*. ACM Transactions on Programming Languages and Systems, 7(3):490–492.

Richard S. Bird (1987). *An introduction to the theory of lists.* In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

Richard S. Bird (1988). *Lectures on constructive functional programming.* In Manfred Broy, editor, *Constructive Methods in Computer Science*. Springer-Verlag. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

Richard S. Bird and Lambert Meertens (1987). *Two exercises found in a book on algorithmics.* In Meertens (1987), pages 451–457.

Guy E. Blelloch (1989). *Scans as primitive parallel operations.* IEEE Transactions on Computers, 38(11):1526–1538.

Guy E. Blelloch (1990). *Vector Models for Data-Parallel Computing.* MIT Press.

Gregor V. Bochmann (1976). *Semantic evaluation from left to right.* Communications of the ACM, 19(2):55–62.

Nicolas Bourbaki (1942). *Éléments de Mathématique, Livre II: Algèbre.* Hermann et C^{ie}. English translation published in 1974 by Addison-Wesley.

Anne Brüggemann-Klein and Derick Wood (1990). *Drawing trees nicely with TEX.* In Malcolm Clark, editor, *TEX: Applications, Uses, Methods*, pages 185–206. Ellis Horwood.

R. M. Burstall (1969). *Proving properties of programs by structural induction.* Computer Journal, 12(1):41–48.

R. M. Burstall and John Darlington (1977). *A transformational system for developing recursive programs.* Journal of the ACM, 24(1):44–67.

Arthur Cayley (1857). *On the theory of the analytical forms called trees.* Philosophical Magazine, 13:172–176. Also in *The Collected Mathematical Papers of Arthur Cayley*, Volume III, p. 242–246, Cambridge, 1890.

Laurian M. Chirica and David F. Martin (1979). *An order-algebraic definition of Knuthian semantics.* Mathematical Systems Theory, 13(1):1–27.

V. Claus, H. Ehrig, and G. Rozenberg, editors (1979). *LNCS 73: Graph Grammars and their application to Computer Science and Biology.* Springer-Verlag.

John Darlington (1981). *The structured description of algorithm derivations.* In J. W. deBakker and H. van Vliet, editors, *Algorithmic Languages*, pages 221–250. Elsevier North-Holland, New York.

J. Darlington and R. M. Burstall (1976). *A system which automatically improves programs.* Acta Informatica, 6(1):41–60. Also in Proceedings of the Third International Joint Conference on Artificial Intelligence, Stanford, 1973.

Kei Davis and John Hughes, editors (1990). *Functional Programming, Glasgow 1989.* Springer-Verlag.

Oege de Moor (1990). *Categories, relations and dynamic programming.* Programming Research Group, Oxford.

Eliezer Dekel and Sartaj Sahni (1983a). *Binary trees and parallel scheduling algorithms.* IEEE Transactions on Computers, C-32(3):307–315.

Eliezer Dekel and Sartaj Sahni (1983b). *Parallel generation of postfix and tree forms.* ACM Transactions on Programming Languages and Systems, 5(3):300–317.

Pierre Deransart, Martin Jourdan, and Bernard Lorho (1988). *LNCS 323: Attribute Grammars—Definitions, Systems and Bibliography.* Springer-Verlag.

Edsger W. Dijkstra and W. H. J. Feijen (1988). *A Method of Programming.* Addison-Wesley.

G. Estrin (1960). *Organization of computer systems—the fixed plus variable structure computer.* In *Proceedings Western Joint Computer Conference*, pages 33–40.

Rodney Farrow (1986). *Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars.* In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98. SIGPLAN Notices Volume 21, Number 7.

Martin S. Feather (1987). *A survey and classification of some program transformation approaches and techniques.* In Meertens (1987), pages 165–195.

Maarten M. Fokkinga (1990). *Tupling and mutumorphisms.* The Squiggolist, 1(4):81–82.

Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer (1991). *A translation from attribute grammars to catamorphisms.* The Squiggolist, 2(1):20–26.

Maarten M. Fokkinga and Erik Meijer (1991). *Program calculation properties of continuous algebras.* Technical Report CS-R9104, CWI, Amsterdam.

Galileo Galilei (1623). *Il Saggiatore.* Rome.

Susan L. Gerhart (1975). *Correctness-preserving program transformations.* In *Proceedings of the Second Symposium on Principles of Programming Languages*, pages 54–66. ACM.

Alan Gibbons and Wojciech Rytter (1986). *An optimal parallel algorithm for dynamic expression evaluation and its applications.* In K. V. Nori, editor, *LNCS 241: Sixth Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 453–469. Springer-Verlag.

Alan Gibbons and Wojciech Rytter (1988). *Efficient Parallel Algorithms.* Cambridge University Press.

Alan Gibbons and Ridha Ziani (1991). *The balanced binary tree technique on mesh-connected computers.* Information Processing Letters, 37:101–109.

Jeremy Gibbons (1988). *A New View of Binary Trees.* Transferral dissertation, Programming Research Group, Oxford University. Abstract appears in the Bulletin of the EATCS, number 39, p. 214.

Tatsuya Hagino (1987a). *A Categorical Programming Language.* PhD thesis, Laboratory for the Foundations of Computer Science, Edinburgh.

Tatsuya Hagino (1987b). *A typed lambda calculus with categorical type constructors.* In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *LNCS 283: Category Theory and Computer Science,* pages 140–157. Springer-Verlag.

C. A. R. Hoare (1972). *Notes on data structuring.* In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming,* APIC studies in data processing, pages 83–174. Academic Press.

Ming-Deh A. Huang (1985). *Solving some graph problems with optimal or near optimal speed-up on mesh-of-trees networks.* In *26th IEEE Symposium on Foundations of Computer Science,* pages 232–240.

Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young (1990). *Report on the programming language Haskell, version 1.0.* Technical report, Yale University and University of Glasgow.

John Hughes (1990). *Compile-time analysis of functional programs.* In Turner (1990), pages 117-153.

Valerie Illingworth, Edward L. Glaser, and I. C. Pyle, editors (1990). *Dictionary of Computing.* Oxford University Press.

Edgar T. Irons (1961). *A syntax directed compiler for Algol 60.* Communications of the ACM, 4:51–55.

Kenneth E. Iverson (1962). *A Programming Language.* John Wiley.

Mehdi Jazayeri, William F. Ogden, and William C. Rounds (1975). *The intrinsically exponential complexity of the circularity problem for attribute grammars.* Communications of the ACM, 18(12):697–706.

Alan Jeffrey (1990). *Soft arrays.* The Squiggolist, 1(4):74–75.

Johan Jeuring (1989). *Deriving algorithms on binary labelled trees.* CWI, Amsterdam.

Thomas Johnsson (1987). *Attribute grammars as a functional programming paradigm.* In G. Kahn, editor, *LNCS 274: Functional Programming Languages and Computer Architecture,* pages 154–173. Springer-Verlag.

Geraint Jones (1989). *Calculating the Fast Fourier Transform as a divide and conquer algorithm.* Unpublished draft, Programming Research Group, Oxford University. Later version appears as 'Deriviug the fast Fourier algorithm by calculation', in (Davis and Hughes, 1990).

Geraint Jones and Mary Sheeran (1990a). *Circuit design in Ruby*. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland.

Geraint Jones and Mary Sheeran (1990b). *Relations and refinement in circuit design*. Technical Report PRG-TR-13-90, Programming Research Group, Oxford.

Martin Jourdan (1984). *Strongly non-circular attribute grammars and their recursive evaluation*. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 81–93. SIGPLAN Notices Volume 19, Number 6.

Takuya Katayama (1984). *Translation of attribute grammars into procedures*. ACM Transactions on Programming Languages and Systems, 6(3):345–369.

G. Kirchhoff (1847). *Über die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme gefürht wird*. Annalen der Physik und Chemie, 72(12):497–508. In German.

Donald E. Knuth (1968a). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.

Donald E. Knuth (1968b). *Semantics of context-free languages*. Mathematical Systems Theory, 2(2):127–145. Correction in (Knuth, 1971c).

Donald E. Knuth (1971a). *Examples of formal semantics*. In E. Engeler, editor, *Lecture Notes in Mathematics 188: Symposium on Semantics of Algorithmic Languages*, pages 212–235. Springer-Verlag.

Donald E. Knuth (1971b). *Optimum binary search trees*. Acta Informatica, 1:14–25.

Donald E. Knuth (1971c). *Semantics of context-free languages: Correction*. Mathematical Systems Theory, 5(1):95–96.

Peter M. Kogge and Harold S. Stone (1973). *A parallel algorithm for the efficient solution of a general class of recurrence equations*. IEEE Transactions on Computers, C-22(8):786–793.

Clyde P. Kruskal, Larry Rudolph, and Marc Snir (1985). *The power of parallel prefix*. IEEE Transactions on Computers, C-34(10):965–968.

Richard E. Ladner and Michael J. Fischer (1980). *Parallel prefix computation*. Journal of the ACM, 27(4):831–838.

Lao Tzŭ (4th century BC). *Tao Tê Ching*.

Charles E. Leiserson and Bruce M. Maggs (1988). *Communication-efficient parallel algorithms for distributed random-access machines*. Algorithmica, 3:53–77.

David B. Loveman (1977). *Program improvement by source-to-source transformation*. Journal of the ACM, 24(1):121–145.

Wayne Luk (1988). *Parametrised Design of Regular Processor Arrays*. D. Phil. thesis, Programming Research Group, Oxford University.

Grant Malcolm (1990). *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen.

Ernest G. Manes and Michael A. Arbib (1986). *Algebraic Approaches to Program Semantics.* AKM Series in Theoretical Computer Science. Springer-Verlag.

Brian H. Mayoh (1981). *Attribute grammars and mathematical semantics.* SIAM Journal on Computing, 10(3):503–518.

Lambert Meertens (1986). *Algorithmics: Towards programming as a mathematical activity.* In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland.

Lambert Meertens, editor (1987). *Program Specification and Transformation.* North-Holland.

Lambert Meertens (1988). *First steps towards the theory of rose trees.* Unpublished draft, CWI, Amsterdam.

Lambert Meertens (1989a). *Constructing a calculus of programs.* In van de Snepscheut (1989), pages 66–90. Also available as Report CS-R8914 from CWI, Amsterdam.

Lambert Meertens (1989b). *Variations on trees.* In *International Summer School on Constructive Algorithmics, Hollum, Ameland* (1989).

Lambert Meertens (1990). *Paramorphisms.* Technical Report CS-R9005, CWI, Amsterdam.

Lambert Meertens and Jaap van der Woude (1991). *A tribute to attributes.* The Squiggolist, 2(1):10–15.

Carroll Morgan (1989). *Whither application?* The Squiggolist, 1(2). CWI, Amsterdam.

Carroll Morgan (1990). *Programming from Specifications.* Prentice Hall.

James H. Morris, Jr (1973). *Types are not sets.* In *Proceedings of the First Symposium on Principles of Programming Languages*, pages 120–124. ACM.

Joe Morris (1987). *A theoretical basis for stepwise refinement and the programming calculus.* Science of Computer Programming, 9(3):287–306.

Thomas J. Myers (1980). *Infinite Structures in Programming Languages.* PhD thesis, University of Pennsylvania, Philadelphia, PA.

John T. O'Donnell (1990). *Derivation of fine-grain algorithms.* Presentation at IFIP Working Group 2.8 meeting, Rome.

G. M. Radack (1988). *Tidy drawing of* M*-ary trees.* Technical Report CES-88-24, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio.

Edward M. Reingold and John S. Tilford (1981). *Tidier drawings of trees.* IEEE Transactions on Software Engineering, 7(2):223–228.

Thomas Reps and Tim Teitelbaum (1984). *The synthesizer generator.* In Peter Henderson, editor, *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48. Software Engineering Notes Volume 9, Number 3, and SIGPLAN Notices Volume 19, Number 5.

Thomas Reps and Tim Teitelbaum (1989). *The Synthesizer Generator—A System for Constructing Language-Based Editors.* Springer-Verlag.

Grzegorz Rozenberg and Arto Salomaa, editors (1986). *The Book of L.* Springer-Verlag.

R. M. Schell, Jr. (1979). *Methods for Constructing Parallel Compilers for use in a Multiprocessor.* PhD thesis, University of Illinois at Urbana-Champaign.

David B. Skillicorn (1990). *Architecture independent parallel computation.* IEEE Computer, 23(12):38–51.

David B. Skillicorn (1991). Private communication.

M. B. Smyth and G. D. Plotkin (1982). *The category-theoretic solution of recursive domain equations.* SIAM Journal on Computing, 11(4):761–783.

STOP project (1989). *International Summer School on Constructive Algorithmics, Hollum, Ameland.*

Kenneth J. Supowit and Edward M. Reingold (1983). *The complexity of drawing trees nicely.* Acta Informatica, 18(4):377–392.

D. A. Turner (1982). *Recursion equations as a programming language.* In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications,* pages 1–28. Cambridge University Press.

David A. Turner (1985). *Miranda: A non-strict functional language with polymorphic types.* In Jean-Pierre Jouannaud, editor, *LNCS 201: Functional Programming Language and Computer Architecture,* pages 1–16. Springer-Verlag.

Prescott K. Turner (1986). *Up-down parsing with prefix grammars.* SIGPLAN Notices, 21(12):167–174.

David A. Turner, editor (1990). *Research Topics in Functional Programming.* University of Texas at Austin, Addison-Wesley.

J. L. A. van de Snepscheut, editor (1989). *LNCS 375: Mathematics of Program Construction.* Springer-Verlag.

A. J. M. van Gasteren (1988). *On the Shape of Mathematical Arguments.* PhD thesis, Technische Universiteit Eindhoven. Also available as LNCS 445.

Jean G. Vaucher (1980). *Pretty-printing of trees.* Software—Practice and Experience, 10:553–561.

Nico Verwer (1990). *Homomorphisms, factorisation and promotion.* The Squiggolist, 1(3). Also available as Report RUU-CS-90-5, Department of Computer Science, Utrecht University.

Xavier Gérard Viennot (1990). *Trees everywhere.* In A. Arnold, editor, *LNCS 431: CAAP '90,* pages 18–41. Springer-Verlag.

John Q. Walker, II (1990). *A node-positioning algorithm for general trees.* Software—Practice and Experience, 20(7):685–705.

Ben Wegbreit (1976). *Goal-directed program transformation.* IEEE Transactions on Software Engineering, SE-2(2):69–79.

Charles Wetherell and Alfred Shannon (1979). *Tidy drawings of trees*. IEEE Transactions on Software Engineering, 5(5):514–520.

David S. Wile (1973). *A Generative, Nested-Sequential Basis for General Purpose Programming Languages*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Niklaus Wirth (1976). *Algorithms + Data Structures = Programs*. Prentice Hall.

Gavin Wraith (1989). *A note on categorical datatypes*. In D. H. Pitt, D. E. Rydeheard, P. Dyjber, A. M. Pitts, and A. Poigné, editors, *LNCS 389: Category Theory and Computer Science*. Springer-Verlag.

Chris J. Wright (1988). *A theory of arrays for program derivation*. Transferral dissertation, Oxford University.

A good calculator has no need of artificial aids.

Lao Tzŭ, Tao Tê Ching, 4th century BC