J. Functional Programming **1** (1): 1–000, May 1995

Deriving tidy drawings of trees

Jeremy Gibbons†
Department of Computer Science,
University of Auckland,
Private Bag 92019,
Auckland, New Zealand.
Email `jeremy@cs.auckland.ac.nz`

Abstract

The *tree-drawing problem* is to produce a 'tidy' mapping of elements of a tree to points in the plane. In this paper, we derive an efficient algorithm for producing tidy drawings of trees. The specification, the starting point for the derivations, consists of a collection of intuitively appealing *criteria* satisfied by tidy drawings. The derivation shows constructively that these criteria completely determine the drawing. Indeed, the criteria completely determine a simple but inefficient algorithm for drawing a tree, which can be transformed into an efficient algorithm using just standard techniques and a small number of inventive steps.

The algorithm consists of an *upwards accumulation* followed by a *downwards accumulation* on the tree, and is further evidence of the utility of these two higher-order tree operations.

Keywords: Derivation, trees, upwards and downwards accumulations, drawing.

1 Introduction

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'. Our definition of tidiness consists of a collection of intuitively appealing criteria 'obviously' satisfied by tidy drawings.

We derive from these criteria an efficient algorithm for producing tidy drawings of binary trees. The derivation process is a constructive proof that the tidiness criteria completely determine the drawing. In other words, there is only one tidy drawing of any given tree. In fact, the derivation of the algorithm is a completely reasonable and almost routine calculation from the criteria: the algorithm itself, like the drawing, is essentially unique.

† Partially supported by University of Auckland Research Committee grant number A18/XXXXX/62090/3414013.

The algorithm that we derive (which is due originally to Reingold and Tilford (1981)) consists of an *upwards accumulation* followed by a *downwards accumulation* (Gibbons, 1991; Gibbons, 1993b) on the tree. Basically, an upwards accumulation on a tree replaces every element of that tree with some function of that element's descendents, while a downwards accumulation replaces every element with some function of that element's ancestors. These two higher-order operations on trees are fundamental components of many tree algorithms, such as tree traversals, the parallel prefix algorithm (Ladner and Fischer, 1980), evaluation of attributes in an attribute grammar (Deransart *et al.*, 1988), evaluation of structured queries on text (Skillicorn, 1993), and so on. Their isolation is an important step in understanding and modularizing a tree algorithm. Moreover, work is progressing (Gibbons, 1993a; Gibbons *et al.*, 1994) on the development of efficient *parallel* algorithms for evaluating upwards and downwards accumulations on a variety of parallel architectures. Identifying the accumulations as components of a known algorithm shows how to implement that algorithm efficiently in parallel.

For the purposes of exposition, we make the simplifying assumption that tree elements are unlabelled or, equivalently, that all labels are the same size. It is easy to generalize the algorithm to cover trees in which the labels may have greatly differing widths. A more interesting generalization covers the case in which tree labels may also have different *heights*. Bloesch (1993) gives two algorithms for this case. It is slightly more difficult to adapt the algorithm to cope with *general* trees, in which parents may have arbitrarily but finitely many children. Radack (1988) and Walker (1990) present two different approaches. Radack's algorithm is derived by Gibbons (1991) and described by Kennedy (1995).

The rest of this paper is organized as follows. In Section 2, we briefly describe our notation. In Section 3, we summarize the ideas behind upwards and downwards accumulations on trees. In Section 4, we present the tidiness criteria, and outline a simple but inefficient tree-drawing algorithm. The derivation of an efficient algorithm, the main part of the paper, is in Section 5.

The diagrams in this paper were drawn 'manually' using John Hobby's META-POST, rather than with the algorithms described here.

2 Notation

We will use the *Bird-Meertens Formalism* or 'BMF' (Backhouse, 1989; Bird, 1987; Bird, 1988; Meertens, 1986), a calculus for the construction of programs from their specifications by a process of equational reasoning. This calculus places great emphasis on notions and properties of *data*, as opposed to *program*, structure. The programs we produce are in a functional style, and are readily translated into a modern functional language such as Haskell or ML.

The BMF is known colloquially as 'Squiggol', because its protagonists make heavy use of unusual symbols and syntax. This approach is helpful to the cognoscenti, but tends to make their work appear unnecessarily obscure to the uninitiated. For this reason, we will use a more traditional notation here. We will use mostly words rather than symbols, and mostly prefix functions rather than infix operators, simply to make expressions easier to parse for those unfamiliar with the calculus. We hasten to add two points. First, this translation leaves the BMF 'philosophy' intact. Second, the presentation here, although more accessible, will be marginally less elegant than it might otherwise have been.

2.1 Basic combinators

Sectioning a binary operator involves providing it with one of its arguments, and results in a function of the other argument. For example, $(2+)$ and $(+2)$ are two ways of writing the function that adds two to its argument. The *constant function* $const\ a$ returns a for every argument; for example, $const\ 1\ 2 = 1$. (Function application is left-associative, so that this parses as '$(const\ 1)\ 2$', and tightest binding.) Function composition is written '$\circ$'; for example, $const\ 1 \circ const\ 2 = const\ 1$. The *identity function* is written 'id'. The *converse* $\tilde{\oplus}$ of a binary operator $\oplus$ is obtained by swapping its arguments; for example, $x \tilde{-} y = y - x$.

The *product type* $A \times B$ consists of pairs (a, b) of values, with $a :: A$ and $b :: B$. The *projection functions* fst and snd return the first and second elements of a pair. The fork $fork\ (f, g)$ of two functions f and g takes a single value and returns a pair; thus, $fork\ (f, g)\ a = (f\ \ a, g\ \ a)$.

2.2 Promotion

The notion of *promotion* comes up repeatedly in the BMF. We say that function f is '$\oplus$ to $\otimes$ promotable' if, for all a and b,

$$f\ (a \oplus b) \quad = \quad f\ a \otimes f\ b$$

Promotion is a generalization of distributivity: f distributes through $\oplus$ iff f is $\oplus$ to $\oplus$ promotable. We say that f 'promotes through $\oplus$' if there is a $\otimes$ such that f is $\oplus$ to $\otimes$ promotable.

2.3 Lists

The type $list\ A$ consists of lists of elements of type A. A list is either a singleton $[a]$ for some a, or the (associative) concatenation $x +\!\!+\, y$ of two lists x and y. In this paper, all lists are non-empty. We write '$wrapl$' for the function taking a to $[a]$, and write longer lists in square brackets too—for example, '$[a, b, c]$' is an abbreviation for $[a] +\!\!+ [b] +\!\!+ [c]$. For every initial datatype such as lists, there is a higher-order function map, which applies a function to every element of a member of that datatype; for example, $map\ (+1)\ [1, 2, 3] = [2, 3, 4]$. We will use map for other datatypes such as trees later, and will trust to context to reveal which particular map is meant.

2.4 Homomorphisms

An important class of functions on lists are those called *homomorphisms*. These are the functions that promote through list concatenation. That is, h is a list homomorphism iff there is an associative operator $\otimes$ such that, for all x and y,

$$h\ (x \mathbin{+\!\!+} y) \quad = \quad h\ x \otimes h\ y$$

The condition of associativity on $\otimes$ is no great restriction. If h is $\mathbin{+\!\!+}$ to $\otimes$ promotable then $\otimes$ is necessarily associative, at least on the range of h:

$$\begin{array}{cl} & h\ x \otimes (h\ y \otimes h\ z) \\ = & \qquad \{h \text{ is } \mathbin{+\!\!+} \text{ to } \otimes \text{ promotable}\} \\ & h\ x \otimes h\ (y \mathbin{+\!\!+} z) \\ = & \qquad \{\text{promotion again}\} \\ & h\ (x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z)) \\ = & \qquad \{\mathbin{+\!\!+} \text{ is associative}\} \\ & h\ ((x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z) \\ = & \qquad \{\text{promotion, twice}\} \\ & (h\ x \otimes h\ y) \otimes h\ z \end{array}$$

In fact, if h is $\mathbin{+\!\!+}$ to $\otimes$ promotable, then it is completely determined by its action on singleton lists; for example,

$$h\ [a, b, c] \quad = \quad h\ ([a] \mathbin{+\!\!+} [b] \mathbin{+\!\!+} [c]) \quad = \quad h\ [a] \otimes h\ [b] \otimes h\ [c]$$

If h is $\mathbin{+\!\!+}$ to $\otimes$ promotable and $h \circ wrapl = f$, then we write h as $lh\ (f, \otimes)$ ('lh' stands for 'list homomorphism').

Stated another way, we have the *Promotion Theorem on Lists*, a special case of the *Promotion Theorem* (Malcolm, 1990):

Theorem 1
If h is $\oplus$ to $\otimes$ promotable, then

$$h \circ lh\ (f, \oplus) \quad = \quad lh\ (h \circ f, \otimes)$$

Since $lh\ (wrapl, \mathbin{+\!\!+}) = id$, this gives us a vehicle for proving the equality of a function h and a homomorphism $lh\ (f, \otimes)$, in that we need only show that h is $\mathbin{+\!\!+}$ to $\otimes$ promotable, and that $h \circ wrapl = f$.

For each f, $map\ f$ is a homomorphism, for

$$map\ f\ (x \mathbin{+\!\!+} y) \quad = \quad map\ f\ x \mathbin{+\!\!+} map\ f\ y$$

Indeed, $map\ f \ = \ lh\ (wrapl \circ f, \mathbin{+\!\!+})$, because $map\ f\ [a] \ = \ [f\ a] \ = \ (wrapl \circ f)\ a$. Another example of a homomorphism is the function len, which returns the length of a list:

$$len \quad = \quad lh\ (const\ 1, +)$$

The functions $head$ and $last$, returning the first and last elements of a list, are also homomorphisms. For example,

$$head\ (x \mathbin{+\!\!+} y) \quad = \quad head\ x \quad = \quad fst\ (head\ x, head\ y)$$

and so $head = lh\ (id, fst)$. Similarly, $last = lh\ (id, snd)$. Other examples that we will encounter are the functions *smallest* and *largest*, which return the smallest and largest elements of a list, respectively:

$$\begin{array}{rcl} smallest & = & lh\ (id, min) \\ largest & = & lh\ (id, max) \end{array}$$

and the function *sum*, which returns the sum of the elements of a list:

$$sum \quad = \quad lh\ (id, +)$$

2.5 Leftwards and rightwards functions

Two generalizations of the notion of list homomorphism are the *leftwards* and the *rightwards* functions. If there exist f and (not necessarily associative) $\oplus$ such that, for all a and x,

$$\begin{array}{rcl} h\ [a] & = & f\ a \\ h\ ([a] \mathbin{+\!\!+} y) & = & a \oplus h\ y \end{array}$$

then we say that h is *leftwards*, and write it $lw\ (f, \oplus)$. Similarly, if for all x and a,

$$\begin{array}{rcl} h\ [a] & = & f\ a \\ h\ (x \mathbin{+\!\!+} [a]) & = & h\ x \otimes a \end{array}$$

then we say that h is *rightwards*, and write it $rw\ (f, \otimes)$. Clearly, if h is a homomorphism then it is both leftwards and rightwards. What is not so obvious is that the converse holds: Bird's *Third Homomorphism Theorem* (Gibbons, 1993a; Gibbons, 1994c) states that if h is both leftwards and rightwards, then it is a homomorphism.

Consider the function *inits*, which takes a list and returns the list of lists consisting of its initial segments, in order of increasing length. For example,

$$inits\ [a, b, c] \quad = \quad [[a], [a, b], [a, b, c]]$$

Now, *inits* is leftwards, because

$$inits\ ([a] \mathbin{+\!\!+} x) \quad = \quad [[a]] \mathbin{+\!\!+} map\ ([a] \mathbin{+\!\!+})\ (inits\ x)$$

In fact,

$$inits \quad = \quad lw\ (wrapl \circ wrapl, \oplus) \qquad \text{where} \quad a \oplus v = [[a]] \mathbin{+\!\!+} map\ ([a] \mathbin{+\!\!+})\ v$$

It is also rightwards, because

$$\begin{array}{rcl} inits\ (x \mathbin{+\!\!+} [a]) & = & inits\ x \mathbin{+\!\!+} [x \mathbin{+\!\!+} [a]] \\ & = & inits\ x \mathbin{+\!\!+} [last\ (inits\ x) \mathbin{+\!\!+} [a]] \end{array}$$

since $last\ (inits\ x) = x$. In fact,

$$inits \quad = \quad rw\ (wrapl \circ wrapl, \otimes) \qquad \text{where} \quad w \otimes a = w \mathbin{+\!\!+} [last\ w \mathbin{+\!\!+} [a]]$$

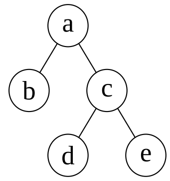

Fig. 1. The tree *five*

Thus, by the Third Homomorphism Theorem, *inits* is a list homomorphism.

2.6 Binary trees

Finally, we come to binary trees. The type *btree* A consists of binary trees labelled with elements of type A. A binary tree is either a leaf $lf\ a$ labelled with a single element a, or a branch $br\ (t, a, u)$ consisting of two children t and u and a label a. For example, the expression

$$br\ (lf\ \ b, a, br\ (lf\ \ d, c, lf\ \ e))$$

corresponds to the tree in Figure 1, which we will call *five* and use as an example later.

Homomorphisms on binary trees $bh\ (f, \oplus)$ ('binary tree homomorphism') promote through br. That is, they satisfy the equations:

$$\begin{array}{rcl} bh\ (f, \oplus)\ (lf\ \ a) & = & f\ \ a \\ bh\ (f, \oplus)\ (br\ (t, a, u)) & = & bh\ (f, \oplus)\ t\ \oplus_a\ bh\ (f, \oplus)\ u \end{array}$$

Note that, for binary trees, the second component of a homomorphism is a *ternary* function. We write its middle argument as a subscript, for lack of anywhere better to put it.

When instantiated to trees, Malcolm's Promotion Theorem states:

Theorem 2
If h satisfies

$$h\ (br\ (t, a, u)) \quad = \quad h\ t \oplus_a h\ u$$

then $h = bh\ (h \circ lf, \oplus)$.

The function *map* on binary trees satisfies

$$\begin{array}{rcl} map\ f\ (lf\ \ a) & = & lf\ (f\ \ a) \\ map\ f\ (br\ (t, a, u)) & = & br\ (map\ \ f\ \ t, f\ \ a, map\ \ f\ \ u) \end{array} \tag{1}$$

and so

$$map\ f \quad = \quad bh\ (lf \circ f, \oplus) \qquad \text{where} \quad v \oplus_a w = br\ (v, f\ \ a, w)$$

The function *root* is a binary tree homomorphism:

$$\begin{array}{rcl} root\ (lf\ \ a) & = & a \\ root\ (br\ (t, a, u)) & = & a \quad = \quad root\ t \oplus_a root\ u \qquad \text{where} \quad v \oplus_a w = a \end{array}$$

and so, with the same $\oplus$,

$$root \quad = \quad bh\ (id, \oplus)$$

So are the functions *size* and *depth*:

$$\begin{aligned} size &= bh\ (const\ 1, \oplus) \qquad \text{where} \quad v \oplus_a w = v + 1 + w \\ depth &= bh\ (const\ 1, \oplus) \qquad \text{where} \quad v \oplus_a w = 1 + max\ (v, w) \end{aligned}$$

and the function *brev*, which reverses a binary tree:

$$brev \quad = \quad bh\ (lf, \oplus) \qquad \text{where} \quad v \oplus_a w = br\ (w, a, v)$$

2.7 Variable-naming conventions

To help the reader, we make a few conventions about the choice of names. For alphabetic names, single-letter identifiers are typically 'local', their definitions persisting only for a few lines, whereas multi-letter identifiers are 'global', having the same definitions throughout the paper. Elements of lists and trees are denoted $a, b, c, \ldots$. Unary functions are denoted f, g, h. Lists and paths (introduced in Section 3.2) are denoted w, x, y, z. Trees are denoted t, u. The letters v and w are used as the 'results' of functions, for example, in the definitions of homomorphisms such as *brev* above.

We define a few infix binary operators such as $\oplus$ and $\boxtimes$, just as we might use alphabetic names for variables and unary functions. Round binary operators such as $\oplus$ and $\otimes$ are 'local', and square binary operators such as $\boxplus$ and $\boxtimes$ are 'global'.

3 Upwards and downwards accumulations on trees

The material in this section is adapted from (Gibbons, 1993b), which is in turn a summary of (Gibbons, 1991).

3.1 Upwards accumulations

Upwards and downwards accumulations arise from considering the list function *inits*. On trees, the obvious analogue of *inits* is the function *subtrees*, which takes a tree and returns a tree of trees. The result is the same shape as the original tree, but each element is replaced by its *descendents*, that is, by the subtree of the original tree rooted at that element. For example:

$$\begin{aligned} subtrees\ five \quad = \quad & br\ (lf\ (lf\ b), \\ & \quad br\ (lf\ b, a, br\ (lf\ d, c, lf\ e)), \\ & \quad br\ (lf\ (lf\ d), \\ & \qquad br\ (lf\ d, c, lf\ e), \\ & \qquad lf\ (lf\ e))) \end{aligned}$$

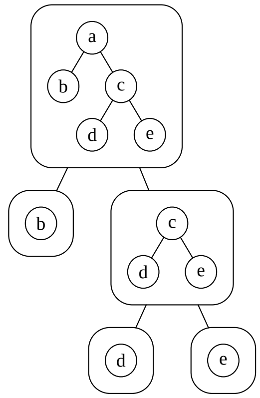

Fig. 2. The subtrees of *five*

which corresponds to the tree of trees in Figure 2. The function *subtrees* is a homomorphism, because it satisfies

$$\begin{array}{rcl} \mathit{subtrees}\ (\mathit{lf}\ a) & = & \mathit{lf}\ (\mathit{lf}\ a) \\ \mathit{subtrees}\ (\mathit{br}\ (t, a, u)) & = & \mathit{br}\ (\mathit{subtrees}\ t, \mathit{br}\ (t, a, u), \mathit{subtrees}\ u) \end{array} \tag{2}$$

Since $\mathit{root}\ (\mathit{subtrees}\ t) = t$, we have

$$\mathit{subtrees}\ (\mathit{br}\ (t, a, u)) \quad = \quad \mathit{subtrees}\ t \oplus_a \mathit{subtrees}\ u$$

where

$$v \oplus_a w \quad = \quad \mathit{br}\ (v, \mathit{br}\ (\mathit{root}\ v, a, \mathit{root}\ w), w)$$

and so, with the same $\oplus$,

$$\mathit{subtrees} \quad = \quad \mathit{bh}\ (\mathit{lf} \circ \mathit{lf}, \oplus)$$

The function *subtrees* replaces every element of a tree with its descendents. An upwards accumulation replaces every element with *some function of* its descendents. In other words, an upwards accumulation is of the form $\mathit{map}\ h \circ \mathit{subtrees}$ for some h. In fact, we do not allow h to be an arbitrary function of the descendents. Rather, we insist that h is a tree homomorphism, to ensure that the accumulation can be computed in linear time (assuming that the components of h take constant time). Consider $\mathit{map}\ h\ (\mathit{subtrees}\ (\mathit{br}\ (t, a, u)))$:

$$\begin{array}{ll} & \mathit{map}\ h\ (\mathit{subtrees}\ (\mathit{br}\ (t, a, u))) \\ = & \qquad \{(2)\} \\ & \mathit{map}\ h\ (\mathit{br}\ (\mathit{subtrees}\ t, \mathit{br}\ (t, a, u), \mathit{subtrees}\ u)) \\ = & \qquad \{(1)\} \\ & \mathit{br}\ (\mathit{map}\ h\ (\mathit{subtrees}\ t), h\ (\mathit{br}\ (t, a, u)), \mathit{map}\ h\ (\mathit{subtrees}\ u)) \end{array}$$

If this is to be computed in linear time, computing $h\ (\mathit{br}\ (t, a, u))$ must take only constant time. If $h = \mathit{bh}\ (f, \oplus)$ where f and $\oplus$ take constant time, then

$$h\ (\mathit{br}\ (t, a, u)) \quad = \quad h\ t \oplus_a h\ u$$

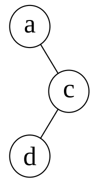

Fig. 3. The path in *five* to the element labelled d

and $h\ t$ and $h\ u$ are available in constant time as the roots of $map\ h\ (subtrees\ t)$ and $map\ h\ (subtrees\ u)$. Stated another way,

$$\begin{array}{ll} map\ (bh\ (f, \oplus)) \circ subtrees & \\ \quad = \ bh\ (lf \circ f, \otimes) \qquad \text{where} \quad v \otimes_a w = br\ (v, root\ v \oplus_a root\ u, u) \end{array}$$

and is therefore both a homomorphism and computable in linear time.

We write '$up\ (f, \oplus)$' for an upwards accumulation. This satisfies

$$up\ (f, \oplus) \quad = \quad map\ (bh\ (f, \oplus)) \circ subtrees \tag{3}$$

but, as described above, requires no longer to compute than $bh\ (f, \oplus)$ does. The function $subtrees$ is itself an upwards accumulation, since $subtrees = map\ id \circ subtrees$ and id is a homomorphism; so is id, since $id = map\ root \circ subtrees$ and $root$ is a homomorphism. A more interesting example is the function $ndescs$, which replaces every element with the number of descendents it has. Letting $\oplus$ satisfy $v \oplus_a w = v + 1 + w$, so that $size = bh\ (const\ 1, \oplus)$, we have

$$\begin{array}{lll} ndescs & = & map\ (bh\ (const\ 1, \oplus)) \circ subtrees \\ & = & up\ (const\ 1, \oplus) \end{array}$$

Note that the expression involving the map takes quadratic time to compute, whereas the accumulation takes linear time.

3.2 Downwards accumulations

Upwards accumulations replace every element of a tree with some function of that element's descendents. For downwards accumulations, on the other hand, we consider an element's *ancestors*. The ancestors of an element form a *path*. For example, the ancestors of the element labelled d in *five* form the path in Figure 3, which could be thought of as a list with two different kinds of concatenation, 'left' and 'right', or as a tree in which each parent has exactly one child. We choose the former view. The type $path\ A$ consists of paths of elements of type A. A path is either a single element $\langle a \rangle$ or two paths x and y joined with a 'left turn', $x \mathbin{+\!\!\!\!\!\!+} y$, or a 'right turn', $x \mathbin{+\!\!\!\!+} y$. The function taking a to $\langle a \rangle$ is written '$wrapp$'. Just as $\mathbin{+\!\!\!\!+}$ is associative, the operations $\mathbin{\overleftarrow{+\!\!+}}$ and $\mathbin{\overrightarrow{+\!\!+}}$ satisfy the four laws

$$
\begin{array}{rcl}
x \mathbin{\overleftarrow{+\!\!+}} (y \mathbin{\overleftarrow{+\!\!+}} z) & = & (x \mathbin{\overleftarrow{+\!\!+}} y) \mathbin{\overleftarrow{+\!\!+}} z \\
x \mathbin{\overleftarrow{+\!\!+}} (y \mathbin{\overrightarrow{+\!\!+}} z) & = & (x \mathbin{\overleftarrow{+\!\!+}} y) \mathbin{\overrightarrow{+\!\!+}} z \\
x \mathbin{\overrightarrow{+\!\!+}} (y \mathbin{\overleftarrow{+\!\!+}} z) & = & (x \mathbin{\overrightarrow{+\!\!+}} y) \mathbin{\overleftarrow{+\!\!+}} z \\
x \mathbin{\overrightarrow{+\!\!+}} (y \mathbin{\overrightarrow{+\!\!+}} z) & = & (x \mathbin{\overrightarrow{+\!\!+}} y) \mathbin{\overrightarrow{+\!\!+}} z
\end{array}
$$

We say that '$\mathbin{\overleftarrow{+\!\!+}}$ cooperates with $\mathbin{\overrightarrow{+\!\!+}}$', or '$\mathbin{\overleftarrow{+\!\!+}}$ and $\mathbin{\overrightarrow{+\!\!+}}$ cooperate with each other'. Thus, the path in Figure 3 is represented by $\langle a\rangle \mathbin{\overrightarrow{+\!\!+}} \langle c\rangle \mathbin{\overleftarrow{+\!\!+}} \langle d\rangle$. Because of the cooperativity property, brackets are unnecessary.

Path homomorphisms promote through both $\mathbin{\overleftarrow{+\!\!+}}$ and $\mathbin{\overrightarrow{+\!\!+}}$; if, for all a, x and y, the function h satisfies

$$
\begin{array}{rcl}
h\ \langle a\rangle & = & f\ a \\
h\ (x \mathbin{\overleftarrow{+\!\!+}} y) & = & h\ x \oplus h\ y \\
h\ (x \mathbin{\overrightarrow{+\!\!+}} y) & = & h\ x \otimes h\ y
\end{array}
$$

and $\oplus$ cooperates with $\otimes$, then we write $ph\ (f, \oplus, \otimes)$ for h.

Just as for lists, we generalize path homomorphisms to *upwards* and *downwards* functions on paths. If, for all a, x and y, the function h satisfies

$$
\begin{array}{rcl}
h\ \langle a\rangle & = & f\ a \\
h\ (\langle a\rangle \mathbin{\overleftarrow{+\!\!+}} y) & = & a \oplus h\ y \\
h\ (\langle a\rangle \mathbin{\overrightarrow{+\!\!+}} y) & = & a \otimes h\ y
\end{array}
$$

then we say that h is *upwards*, and write it $uw\ (f, \oplus, \otimes)$. The operators $\oplus$ and $\otimes$ need not enjoy any cooperativity properties. Similarly, if, for all a, x and y,

$$
\begin{array}{rcl}
h\ \langle a\rangle & = & f\ a \\
h\ (x \mathbin{\overleftarrow{+\!\!+}} \langle a\rangle) & = & h\ x \oplus a \\
h\ (x \mathbin{\overrightarrow{+\!\!+}} \langle a\rangle) & = & h\ x \otimes a
\end{array}
$$

then we say that h is *downwards*, and write it $dw\ (f, \oplus, \otimes)$. Path homomorphisms are clearly both upwards and downwards; a generalization of Bird's Third Homomorphism Theorem states the converse.

Theorem 3 (Third Homomorphism Theorem for Paths (Gibbons, 1993a))
A path function that is both upwards and downwards is necessarily a path homomorphism.

The dual for downwards accumulations of the function *subtrees* is the function *paths*, which replaces each element of a tree with that element's ancestors. For example:

$$
\begin{array}{rcl}
\mathit{paths\ five} & = & br\ (lf\ (\langle a\rangle \mathbin{\overleftarrow{+\!\!+}} \langle b\rangle), \\
 & & \quad \langle a\rangle, \\
 & & \quad br\ (lf\ (\langle a\rangle \mathbin{\overrightarrow{+\!\!+}} \langle c\rangle \mathbin{\overleftarrow{+\!\!+}} \langle d\rangle), \\
 & & \qquad \langle a\rangle \mathbin{\overrightarrow{+\!\!+}} \langle c\rangle, \\
 & & \qquad lf\ (\langle a\rangle \mathbin{\overrightarrow{+\!\!+}} \langle c\rangle \mathbin{\overrightarrow{+\!\!+}} \langle e\rangle)))
\end{array}
$$

which corresponds to the tree of paths in Figure 4. The function *paths* is another

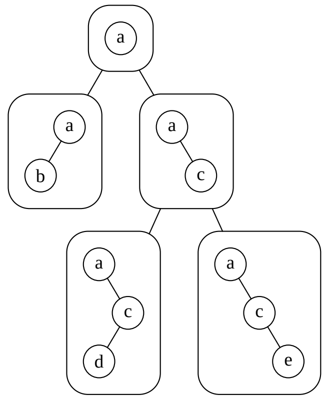

Fig. 4. The paths of *five*

tree homomorphism; it satisfies

$$\begin{array}{rcl} paths\ (lf\ a) & = & lf\ \langle a \rangle \\ paths\ (br\ (t, a, u)) & = & br\ (map\ (\langle a \rangle +\!\!\!+)\ (paths\ t), \\ & & \quad \langle a \rangle, \\ & & \quad map\ (\langle a \rangle +\!\!\!+)\ (paths\ u)) \end{array}$$

and so

$$paths \quad = \quad bh\ (lf \circ wrapp, \oplus)$$

where

$$v \oplus_a w \quad = \quad br\ (map\ (\langle a \rangle +\!\!\!+)\ v, \langle a \rangle, map\ (\langle a \rangle +\!\!\!+)\ w)$$

A *downwards accumulation* replaces every element of a tree with *some function of* that element's ancestors. In other words, downwards accumulations are of the form $map\ h \circ paths$ for some h. Again, we make a restriction on the choice of h, but this time it is not so clear just what that restriction should be. On the one hand, we would like h to be upwards, for

$$\begin{array}{l} map\ (uw\ (f, \oplus, \otimes))\ (paths\ (br\ (t, a, u))) \\ \quad = \quad br\ (map\ (a\oplus)\ (map\ (uw\ (f, \oplus, \otimes))\ (paths\ t)), \\ \qquad\qquad f\ a, \\ \qquad\qquad map\ (a\otimes)\ (map\ (uw\ (f, \oplus, \otimes))\ (paths\ u))) \end{array}$$

and so $map\ (uw\ (f, \oplus, \otimes)) \circ paths$ is a homomorphism:

$$map\ (uw\ (f, \oplus, \otimes)) \circ paths \quad = \quad bh\ (lf \circ f, \circledast)$$

where

$$v \circledast_a w \quad = \quad br\ (map\ (a\oplus)\ v, f\ a, map\ (a\otimes)\ w)$$

In terms of the Promotion Theorem, this could be stated as follows:

Theorem 4
If

$$\begin{array}{rcl} g\ (lf\ a) & = & lf\ (f\ a) \\ g\ (br\ (t, a, u)) & = & br\ (map\ (a\oplus)\ (g\ t), f\ a, map\ (a\otimes)\ (g\ u)) \end{array}$$

then

$$g \quad = \quad map\ (uw\ (f, \oplus, \otimes)) \circ paths$$

(We will use this theorem later.)

On the other hand, mapping an upwards function over the paths of a tree takes quadratic time to compute, and so we would like h to be downwards, for

$$\begin{array}{ll} & map\ (dw\ (f, \oplus, \otimes))\ (paths\ (br\ (t, a, u))) \\ = & br\ (map\ (dw\ (((f\ \ a)\oplus), \oplus, \otimes))\ (paths\ t), \\ & \quad f\ \ a, \\ & \quad map\ (dw\ (((f\ \ a)\otimes), \oplus, \otimes))\ (paths\ u)) \end{array}$$

which can be computed in linear time, at the cost of no longer being homomorphic (since the result of applying $map\ (dw\ (f, \oplus, \otimes)) \circ paths$ to $br\ (t, a, u)$ depends on the results of applying *different* functions, $map\ (dw\ (((f\ \ a)\oplus), \oplus, \otimes)) \circ paths$ and $map\ (dw\ (((f\ \ a)\otimes), \oplus, \otimes)) \circ paths$ to the children t and u). To satisfy both of these requirements, we insist that h be both upwards and downwards. Theorem 3 concludes that h is therefore a path homomorphism. We write '$down\ (f, \oplus, \otimes)$' for a downwards accumulation; it satisfies

$$down\ (f, \oplus, \otimes) \quad = \quad map\ (ph\ (f, \oplus, \otimes)) \circ paths \tag{4}$$

but again can be computed in linear time (if f, $\oplus$ and $\otimes$ each take constant time). Note that $\oplus$ and $\otimes$ must cooperate with each other.

For example, consider the function $plen$, which returns the length of a path. The function $depths$ replaces every element of a tree with that element's depth in the tree, that is, with the length of its path of ancestors:

$$depths \quad = \quad map\ plen \circ paths$$

As it stands, it is not obvious whether $depths$ is a homomorphism, nor whether it can be computed efficiently. However, $plen$ is upwards,

$$plen \quad = \quad uw\ (const\ 1, \oplus, \oplus) \qquad \text{where} \quad a \oplus v = 1 + v$$

and so $depths$ is a tree homomorphism. Moreover, $plen$ is downwards,

$$plen \quad = \quad dw\ (const\ 1, \otimes, \otimes) \qquad \text{where} \quad v \otimes a = v + 1$$

and so $depths$ can also be computed in linear time. Writing

$$depths \quad = \quad down\ (const\ 1, +, +)$$

(since $+$ is associative, it cooperates with itself) shows that $depths$ is both homomorphic and efficiently computable.

We might ask, when can we generalize an upwards function h so that it is also downwards? This would give us an efficient way of computing $map\ h \circ paths$.

Suppose h is upwards but not downwards—we cannot write $h\ (x +\!\!\!+ \langle a \rangle)$ and $h\ (x +\!\!\!+ \langle a \rangle)$ in terms of $h\ x$ and a. Suppose, however, that there is another function g such that $h\ (x +\!\!\!+ \langle a \rangle)$ and $h\ (x +\!\!\!+ \langle a \rangle)$ *can* be computed from $h\ x$, $g\ x$ and a: for some $\ominus$ and $\oslash$,

$$
\begin{array}{rcl}
h\ (x \mathbin{\leftarrow\!\!\!\!\!\!+\!\!\!+} \langle a\rangle) & = & (h\ x, g\ x) \ominus a \\
h\ (x \mathbin{+\!\!\!+\!\!\!\!\!\!\rightarrow} \langle a\rangle) & = & (h\ x, g\ x) \oslash a
\end{array}
$$

In a sense, g is the 'extra information' needed to compute $h\ (x \mathbin{\leftarrow\!\!\!\!\!\!+\!\!\!+} \langle a\rangle)$ and $h\ (x \mathbin{+\!\!\!+\!\!\!\!\!\!\rightarrow} \langle a\rangle)$ from $h\ x$ and a. Now h could be computed downwards, if only we could somehow compute g. This, of course, begs the question, how do we compute g? Suppose further that g is 'self-sustaining', in that no further information is required in order to compute g: for some $\odot$ and $\circledast$,

$$
\begin{array}{rcl}
g\ (x \mathbin{\leftarrow\!\!\!\!\!\!+\!\!\!+} \langle a\rangle) & = & (h\ x, g\ x) \odot a \\
g\ (x \mathbin{+\!\!\!+\!\!\!\!\!\!\rightarrow} \langle a\rangle) & = & (h\ x, g\ x) \circledast a
\end{array}
$$

Then $\mathit{fork}\ (h, g)$ is downwards.

Theorem 5
If

$$
\begin{array}{rclcrcl}
h\ \langle a\rangle & = & f_1\ a & \qquad & g\ \langle a\rangle & = & f_2\ a \\
h\ (x \mathbin{\leftarrow\!\!\!\!\!\!+\!\!\!+} \langle a\rangle) & = & (h\ x, g\ x) \ominus a & & g\ (x \mathbin{\leftarrow\!\!\!\!\!\!+\!\!\!+} \langle a\rangle) & = & (h\ x, g\ x) \odot a \\
h\ (x \mathbin{+\!\!\!+\!\!\!\!\!\!\rightarrow} \langle a\rangle) & = & (h\ x, g\ x) \oslash a & & g\ (x \mathbin{+\!\!\!+\!\!\!\!\!\!\rightarrow} \langle a\rangle) & = & (h\ x, g\ x) \circledast a
\end{array}
$$

then

$$
\begin{array}{rclcrcl}
\mathit{fork}\ (h, g) & = & \mathit{dw}\ (f, \oplus, \otimes) & \quad \text{where} \quad & f\ a & = & (f_1\ a, f_2\ a) \\
 & & & & (v, w) \oplus a & = & (v \ominus a, w \odot a) \\
 & & & & (v, w) \otimes a & = & (v \oslash a, w \circledast a)
\end{array}
$$

Then we have $h = \mathit{fst} \circ \mathit{fork}\ (h, g)$, and so h is 'almost' downwards—it is the composition of the projection fst with the downwards function $\mathit{fork}\ (h, g)$. However, it is not obvious whether $\mathit{fork}\ (h, g)$ is still upwards. Fortunately, if g is itself upwards, then so is $\mathit{fork}\ (h, g)$, as shown by the following theorem.

Theorem 6

$$
\begin{array}{l}
\mathit{fork}\ (\mathit{uw}\ (f_1, \ominus, \oslash), \mathit{uw}\ (f_2, \odot, \circledast)) \\
\qquad = \quad \mathit{uw}\ (f, \oplus, \otimes) \quad \text{where} \quad
\begin{array}{rcl}
f\ a & = & (f_1\ a, f_2\ a) \\
a \oplus (v, w) & = & (a \ominus v, a \odot w) \\
a \otimes (v, w) & = & (a \oslash v, a \circledast w)
\end{array}
\end{array}
$$

In this case, $\mathit{fork}\ (h, g)$ is both upwards and downwards, and hence a path homomorphism. Then

$$
\mathit{map}\ h \circ \mathit{paths} \quad = \quad \mathit{map}\ \mathit{fst} \circ \mathit{map}\ (\mathit{fork}\ (h, g)) \circ \mathit{paths}
$$

which is a (cheap) map composed with a downwards accumulation, and is efficiently computable.

4 Drawing binary trees tidily

In this section, we define 'tidiness' and specify the function *bdraw*, which draws a binary tree. We make the simplifying assumption that all tree labels are the same size, because, for the purposes of positioning the elements of the tree, we can then ignore the labels altogether.

The first property that we observe of tidy drawings is that all of 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 argument is of type *btree* A for some A, and its result is a binary tree labelled with x-coordinates:

$$bdraw \quad :: \quad btree\ A \rightarrow btree\ \mathbb{D}$$

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 then constructed by translating the drawings of the children. In symbols:

$$bdraw\ (br\ (t, a, u)) \quad = \quad br\ (map\ (+r)\ (bdraw\ t), b, map\ (+s)\ (bdraw\ u))$$

for some b, 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, $r+s$ and b in the above equation should both be 0, as should the position given to the only element of a singleton tree:

$$\begin{aligned} bdraw\ (lf\ a) &= lf\ 0 \\ bdraw\ (br\ (t, a, u)) &= br\ (map\ (-s)\ (bdraw\ t), 0, map\ (+s)\ (bdraw\ u)) \end{aligned}$$

for some s. Indeed, a tidy drawing will have the left child to the left of the right child, and so $s > 0$.

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 write '-' for unary negation†, then we also require

$$bdraw \circ brev \quad = \quad map\ \text{-} \circ brev \circ bdraw$$

The fourth criterion is that, in a tidy drawing, elements do not collide, or even get too close together. That is, pictures of children do not overlap, and no two elements on the same level are less than one unit apart.

† The presence of sectioning means that, strictly speaking, we should distinguish between the number 'minus one', written '-1', and the function 'minus one', written '(-1)'.

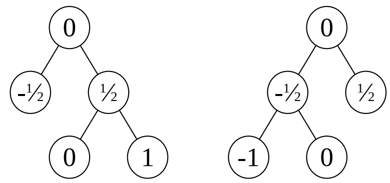

Fig. 5. Drawings pic_1 and pic_2, for which $pic_1 \boxplus pic_2 = \text{-}2$

Finally, a tidy drawing should be as narrow as possible, given the above constraints. Supowit and Reingold (1983) show 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 homomorphic solution.

These last two properties determine s, the distance through which children are translated. That distance should be the smallest distance that does not cause violation of the fourth criterion. Suppose the operator $\boxplus$, 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.) For example, if pic_1 and pic_2 are as in Figure 5, then $pic_1 \boxplus pic_2 = \text{-}2$, the minimum of $0 - 0$, $\text{-}\tfrac{1}{2} - \tfrac{1}{2}$ and $\text{-}1 - 1$. The drawings should be moved apart or together to make this distance 1, that is,

$$s \quad = \quad (1 - (bdraw\ t \boxplus bdraw\ u)) \div 2$$

(In the example above, s will be $1\tfrac{1}{2}$.)

All that remains to be done to complete the specification is to formalize this description of $\boxplus$.

4.1 Levelorder traversal

We define two different 'zip' operators, each of which takes a pair of lists and returns a single list by combining corresponding elements in some way. These two operators are 'short zip', which we write *szip*, and 'long zip', written *lzip*. These operators 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 its longer argument. For example:

$$\begin{array}{rcl} szip\ (\oplus)\ ([a, b], [c, d, e]) & = & [a \oplus c, b \oplus d] \\ lzip\ (\oplus)\ ([a, b], [c, d, e]) & = & [a \oplus c, b \oplus d, e] \end{array}$$

From the result of the long zip, we see that the $\oplus$ must have type $A \times A \rightarrow A$. This is not necessary for short zip, but we do not use the general case.

The two zips are given formally by the equations

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

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

They share many properties, but we use two in particular.

Fact 7
Both $szip\ (\oplus)\ (x, y)$ and $lzip\ (\oplus)\ (x, y)$ can be evaluated using just $min\ (len\ x, len\ y)$ applications of $\oplus$.

Lemma 8
If f is $\oplus$ to $\otimes$ promotable, then $map\ f$ is both $szip\ (\oplus)$ to $szip\ (\otimes)$ and $lzip\ (\oplus)$ to $lzip\ (\otimes)$ promotable.

We use long zip to define *levelorder traversal* of binary trees. This is given by the function $levels :: btree\ A \rightarrow list\ (list\ A)$:

$$levels \quad = \quad bh\ (wrapl \circ wrapl, \oplus) \qquad \text{where} \quad x \oplus_a y = [[a]] \mathbin{+\!\!+} lzip\ (\mathbin{+\!\!+})\ (x, y)$$

For example, the levelorder traversals of $lf\ b$ and $br\ (lf\ d, c, lf\ e)$ are $[[b]]$ and $[[c], [d, e]]$, respectively, and so

$$\begin{array}{cl}
 & levels\ five \\
= & [[a]] \mathbin{+\!\!+} lzip\ (\mathbin{+\!\!+})\ ([[b]], [[c], [d, e]]) \\
= & [[a]] \mathbin{+\!\!+} [[b] \mathbin{+\!\!+} [c], [d, e]] \\
= & [[a], [b, c], [d, e]]
\end{array}$$

We can at last define the operator $\boxplus$ on pictures, in terms of levelorder traversal. It is given by

$$\begin{array}{rcl}
p \boxplus q & = & smallest\ (szip\ (\tilde{-})\ (map\ largest\ (levels\ p), \\
 & & \qquad map\ smallest\ (levels\ q)))
\end{array}$$

If v and w are levels at the same depth in p and q, then $largest\ v$ and $smallest\ w$ are the rightmost point of v and the leftmost point of w, respectively, and so $smallest\ w - largest\ v$ is the width of the gap at this level. Clearly, $p \boxplus q$ is the minimum over all levels of these gap widths. For example, with pic_1 and pic_2 as in Figure 5, we have

$$\begin{array}{rcl}
map\ largest\ (levels\ pic_1) & = & [0, 1/2, 1] \\
map\ smallest\ (levels\ pic_2) & = & [0, \text{-}1/2, \text{-}1]
\end{array}$$

and so

$$pic_1 \boxplus pic_2 \quad = \quad smallest\ [0-0, \text{-}{}^1\!/\!_2 - {}^1\!/\!_2, \text{-}1-1] \quad = \quad \text{-}2$$

This completes the specification of $\boxplus$, and hence of *bdraw*:

$$bdraw \quad = \quad bh\ (const\ (lf\ 0), \boxdot) \tag{5}$$

where

$$\begin{array}{rcl} p \boxdot_a q & = & br\ (map\ (-s)\ p, 0, map\ (+s)\ q) \qquad \text{where} \quad s = (1-(p \boxplus q)) \div 2 \\ p \boxplus q & = & smallest\ (szip\ (\tilde{-})\ (map\ largest\ (levels\ p), \\ & & \qquad map\ smallest\ (levels\ q))) \end{array}$$

This specification is executable, but requires quadratic effort. We now derive a linear algorithm to satisfy it.

5 Drawing binary trees efficiently

A major source of inefficiency in the program that we have just developed is the occurrence of the two maps in the definition of $\boxdot$. Intuitively, we have to shift the drawings of two children when assembling the drawing of their parent, and then shift the whole lot once more when drawing the grandparent. This is because we are computing directly the absolute position of every element. If instead we were to compute the *relative* position 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* on drawings of trees satisfies

$$\begin{array}{rcl} rootrel\ (lf\ a) & = & 0 \\ rootrel\ (br\ (t, a, u)) & = & (a - root\ t) \odot (root\ u - a) \end{array}$$

for some idempotent operator $\odot$. The idea here is that *rootrel* determines the position of a parent relative to its children, given the drawing of the parent. For example, with pic_1 as in Figure 5, we have:

$$rootrel\ pic_1 \quad = \quad (0 - \text{-}{}^1\!/\!_2) \odot ({}^1\!/\!_2 - 0) \quad = \quad {}^1\!/\!_2$$

That is, if we define the function *sep* by

$$sep \quad = \quad rootrel \circ bdraw \tag{6}$$

then

$$\begin{array}{rcl} sep\ (lf\ a) & = & 0 \\ sep\ (br\ (t, a, u)) & = & (1 - (bdraw\ t \boxplus bdraw\ u)) \div 2 \end{array} \tag{7}$$

For example:

$$\begin{array}{rcl} sep\ five & = & (1 - (bdraw\ (lf\ b) \boxplus bdraw\ (br\ (lf\ d, c, lf\ e)))) \div 2 \\ & = & (1-0) \div 2 \\ & = & {}^1\!/\!_2 \end{array}$$

Then

$$bdraw\ (br\ (t, a, u)) \quad = \quad br\ (map\ (-s)\ (bdraw\ t), 0, map\ (+s)\ (bdraw\ u)) \\ \text{where} \quad s = sep\ (br\ (t, a, u))$$

Now, applying *sep* to each subtree gives the relative (to its children) position of every parent. Define the function *rel* by

$$rel \quad = \quad map\ sep \circ subtrees \tag{8}$$

From this, we calculate that

$$\begin{array}{ll} & rel\ (lf\ a) \\ = & \quad \{(8)\} \\ & map\ sep\ (subtrees\ (lf\ a)) \\ = & \quad \{(2)\} \\ & map\ sep\ (lf\ (lf\ a)) \\ = & \quad \{(1)\} \\ & lf\ (sep\ (lf\ a)) \\ = & \quad \{(7)\} \\ & lf\ 0 \end{array}$$

and

$$\begin{array}{ll} & rel\ (br\ (t, a, u)) \\ = & \quad \{(8)\} \\ & map\ sep\ (subtrees\ (br\ (t, a, u))) \\ = & \quad \{(2)\} \\ & map\ sep\ (br\ (subtrees\ t, br\ (t, a, u), subtrees\ u)) \\ = & \quad \{(1)\} \\ & br\ (map\ sep\ (subtrees\ t), sep\ (br\ (t, a, u)), map\ sep\ (subtrees\ u))) \\ = & \quad \{(8)\} \\ & br\ (rel\ t, sep\ (br\ (t, a, u)), rel\ u) \end{array}$$

That is,

$$\begin{array}{rcl} rel\ (lf\ a) & = & lf\ 0 \\ rel\ (br\ (t, a, u)) & = & br\ (rel\ t, sep\ (br\ (t, a, u)), rel\ u) \end{array} \tag{9}$$

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

$$abs \circ rel \quad = \quad bdraw \tag{10}$$

We can calculate from this requirement a definition of *abs*. On leaves, the condition reduces to

$$abs\ (rel\ (lf\ a)) = bdraw\ (lf\ a)$$

$$\begin{array}{ll} \Leftrightarrow & \{(9), (5)\} \\ & abs\ (lf\ 0)\ =\ lf\ 0 \end{array}$$

while on branches we require

$$\begin{array}{ll} & abs\ (rel\ (br\ (t, a, u)))\ =\ bdraw\ (br\ (t, a, u)) \\ \Leftrightarrow & \{(9), (5); \text{let } s = sep\ (br\ (t, a, u))\} \\ & abs\ (br\ (rel\ t, s, rel\ u))\ =\ br\ (map\ (-s)\ (bdraw\ t), 0, map\ (+s)\ (bdraw\ u)) \\ \Leftrightarrow & \{\text{assuming (10) holds on smaller trees}\} \\ & abs\ (br\ (rel\ t, s, rel\ u))\ =\ br\ (map\ (-s)\ (abs\ (rel\ t)), 0, map\ (+s)\ (abs\ (rel\ u))) \end{array}$$

These requirements are satisfied if

$$\begin{array}{rcl} abs\ (lf\ a) & = & lf\ 0 \\ abs\ (br\ (t, a, u)) & = & br\ (map\ (-a)\ (abs\ t), 0, map\ (+a)\ (abs\ u)) \end{array}$$

By Theorem 4, this implies that

$$abs \quad = \quad map\ (uw\ (const\ 0, \tilde{-}, +)) \circ paths$$

We give the upwards function $uw\ (const\ 0, \tilde{-}, +)$ a name, $pabs$ ('the absolute position of the bottom of a path'), for brevity:

$$pabs \quad = \quad uw\ (const\ 0, \tilde{-}, +)$$

so that

$$abs \quad = \quad map\ pabs \circ paths \tag{11}$$

Thus, we have

$$bdraw \quad = \quad abs \circ rel \tag{12}$$

where

$$\begin{array}{rcl} rel & = & map\ sep \circ subtrees \\ abs & = & map\ pabs \circ paths \end{array}$$

This is still inefficient, as computing rel takes quadratic time (because sep is not a tree homomorphism) and computing abs takes quadratic time (because $pabs$ is not path homomorphism). We show next how to compute rel and abs quickly.

5.1 An upwards accumulation

We want to find an efficient way of computing the function rel satisfying

$$rel \quad = \quad map\ sep \circ subtrees$$

where

$$\begin{array}{rcl} sep\ (lf\ a) & = & 0 \\ sep\ (br\ (t, a, u)) & = & (1 - (bdraw\ t \boxplus bdraw\ u)) \div 2 \end{array}$$

We have already observed that rel is not an upwards accumulation, because sep is not a homomorphism—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? It is not hard to see that, in order to compute the separation of the children, we need to know the 'outlines' of their drawings.

Each level of a picture is sorted. Therefore,

$$\begin{aligned} \mathit{map\ smallest} \circ \mathit{levels} &= \mathit{map\ head} \circ \mathit{levels} \\ \mathit{map\ largest} \circ \mathit{levels} &= \mathit{map\ last} \circ \mathit{levels} \end{aligned}$$

and so

$$p \boxplus q \quad = \quad \mathit{right}\ p \boxtimes \mathit{left}\ q \tag{13}$$

where

$$\begin{aligned} \mathit{left} &= \mathit{map\ head} \circ \mathit{levels} \\ \mathit{right} &= \mathit{map\ last} \circ \mathit{levels} \end{aligned}$$

and

$$v \boxtimes w \quad = \quad \mathit{smallest}\ (\mathit{szip}\ (\tilde{-})\ (v, w))$$

Intuitively, *left* and *right* return the 'contours' of a drawing. For example, applying the function *fork* (*left*, *right*) to the tree pic_1 in Figure 5 produces the pair of lists $([0, \text{-}1/2, 0], [0, 1/2, 1])$. These contours are precisely the extra information needed to make *sep* a homomorphism.

To show this, we need to show first that *sep* can be computed from the contours, and second that computing the contours is a homomorphism. Define the function *contours* by

$$\mathit{contours} \quad = \quad \mathit{fork}\ (\mathit{left}, \mathit{right}) \circ \mathit{bdraw} \tag{14}$$

How do we find *sep t* from *contours t*? By definition, the head of each contour is 0, and (if t is not just a leaf) the second elements in the contours are -(*sep t*) and *sep t*. Thus,

$$\mathit{sep} \quad = \quad \mathit{spread} \circ \mathit{contours} \tag{15}$$

where, for some idempotent $\odot$,

$$\begin{aligned} \mathit{spread}\ ([0], [0]) &= 0 \\ \mathit{spread}\ ([0] \mathbin{+\!\!+} x, [0] \mathbin{+\!\!+} y) &= \text{-}(\mathit{head}\ x) \odot \mathit{head}\ y \end{aligned}$$

on pairs of lists, each with head 0.

Now we show that *contours* is a homomorphism. On leaves, we have

$$\begin{aligned} & \mathit{contours}\ (\mathit{lf}\ a) \\ = & \qquad \{(14)\} \\ & \mathit{fork}\ (\mathit{left}, \mathit{right})\ (\mathit{bdraw}\ (\mathit{lf}\ a)) \\ = & \qquad \{(5)\} \\ & \mathit{fork}\ (\mathit{left}, \mathit{right})\ (\mathit{lf}\ 0) \\ = & \qquad \{\mathit{left},\ \mathit{right}\} \\ & ([0], [0]) \end{aligned}$$

For branches, we will consider just the left contour, as the right contour is symmetric. We have

$$
\begin{array}{ll}
& \mathit{left}\;(\mathit{bdraw}\;(\mathit{br}\;(t,a,u))) \\
= & \quad \{(5),\text{ setting } s = (1 - (\mathit{bdraw}\;t \boxplus \mathit{bdraw}\;u)) \div 2\} \\
& \mathit{left}\;(\mathit{br}\;(\mathit{map}\;(-s)\;(\mathit{bdraw}\;t), 0, \mathit{map}\;(+s)\;(\mathit{bdraw}\;u))) \\
= & \quad \{\mathit{left}\} \\
& \mathit{map}\;\mathit{head}\;(\mathit{levels}\;(\mathit{br}\;(\mathit{map}\;(-s)\;(\mathit{bdraw}\;t), 0, \mathit{map}\;(+s)\;(\mathit{bdraw}\;u)))) \\
= & \quad \{\mathit{levels}\} \\
& \mathit{map}\;\mathit{head}\;([[0]] +\!\!+\; \mathit{lzip}\;(+\!\!+)\;(\mathit{levels}\;(\mathit{map}\;(-s)\;(\mathit{bdraw}\;t)), \\
& \qquad\qquad\qquad\qquad\qquad \mathit{levels}\;(\mathit{map}\;(+s)\;(\mathit{bdraw}\;u)))) \\
= & \quad \{\mathit{map},\ \mathit{head}\} \\
& [0] +\!\!+\; \mathit{map}\;\mathit{head}\;(\mathit{lzip}\;(+\!\!+)\;(\mathit{levels}\;(\mathit{map}\;(-s)\;(\mathit{bdraw}\;t)), \\
& \qquad\qquad\qquad\qquad\qquad \mathit{levels}\;(\mathit{map}\;(+s)\;(\mathit{bdraw}\;u)))) \\
= & \quad \{\mathit{head}\text{ is } +\!\!+ \text{ to } \mathit{fst}\text{ promotable; Lemma 8}\} \\
& [0] +\!\!+\; \mathit{lzip}\;\mathit{fst}\;(\mathit{map}\;\mathit{head}\;(\mathit{levels}\;(\mathit{map}\;(-s)\;(\mathit{bdraw}\;t))), \\
& \qquad\qquad \mathit{map}\;\mathit{head}\;(\mathit{levels}\;(\mathit{map}\;(+s)\;(\mathit{bdraw}\;u)))) \\
= & \quad \{\mathit{levels} \circ \mathit{map}\;f = \mathit{map}\;(\mathit{map}\;f) \circ \mathit{levels}\} \\
& [0] +\!\!+\; \mathit{lzip}\;\mathit{fst}\;(\mathit{map}\;\mathit{head}\;(\mathit{map}\;(\mathit{map}\;(-s))\;(\mathit{levels}\;(\mathit{bdraw}\;t))), \\
& \qquad\qquad \mathit{map}\;\mathit{head}\;(\mathit{map}\;(\mathit{map}\;(+s))\;(\mathit{levels}\;(\mathit{bdraw}\;u)))) \\
= & \quad \{\mathit{head} \circ \mathit{map}\;f = f \circ \mathit{head}\} \\
& [0] +\!\!+\; \mathit{lzip}\;\mathit{fst}\;(\mathit{map}\;(-s)\;(\mathit{map}\;\mathit{head}\;(\mathit{levels}\;(\mathit{bdraw}\;t))), \\
& \qquad\qquad \mathit{map}\;(+s)\;(\mathit{map}\;\mathit{head}\;(\mathit{levels}\;(\mathit{bdraw}\;u)))) \\
= & \quad \{\mathit{left}\} \\
& [0] +\!\!+\; \mathit{lzip}\;\mathit{fst}\;(\mathit{map}\;(-s)\;(\mathit{left}\;(\mathit{bdraw}\;t)), \\
& \qquad\qquad \mathit{map}\;(+s)\;(\mathit{left}\;(\mathit{bdraw}\;u)))
\end{array}
$$

Similarly,

$$
\begin{array}{l}
\mathit{right}\;(\mathit{bdraw}\;(\mathit{br}\;(t,a,u))) \\
\qquad = \quad [0] +\!\!+\; \mathit{lzip}\;\mathit{snd}\;(\mathit{map}\;(-s)\;(\mathit{right}\;(\mathit{bdraw}\;t)), \\
\qquad\qquad\qquad\qquad\qquad \mathit{map}\;(+s)\;(\mathit{right}\;(\mathit{bdraw}\;u)))
\end{array}
$$

Now,

$$
\begin{array}{ll}
& \mathit{bdraw}\;t \boxplus \mathit{bdraw}\;u \\
= & \quad \{(13)\} \\
& \mathit{right}\;(\mathit{bdraw}\;t) \boxtimes \mathit{left}\;(\mathit{bdraw}\;u) \\
= & \quad \{(14)\} \\
& \mathit{snd}\;(\mathit{contours}\;t) \boxtimes \mathit{fst}\;(\mathit{contours}\;u)
\end{array}
$$

and so

$$
\mathit{contours}\;(\mathit{br}\;(t,a,u)) \quad = \quad \mathit{contours}\;t \boxminus_a \mathit{contours}\;u
$$

where

$$
\begin{aligned}
&(w,x) \boxminus_a (y,z) \\
&\quad = \quad ([0] \mathbin{+\!\!+} \mathit{lzip\ fst}\ (\mathit{map}\ (-s)\ w, \mathit{map}\ (+s)\ y), \\
&\qquad\qquad [0] \mathbin{+\!\!+} \mathit{lzip\ snd}\ (\mathit{map}\ (-s)\ x, \mathit{map}\ (+s)\ z)) \\
&\qquad\qquad\qquad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2
\end{aligned}
\tag{16}
$$

Hence,

$$
\mathit{contours} \quad = \quad \mathit{bh}\ (\mathit{const}\ ([0],[0]), \boxminus) \tag{17}
$$

Thus,

$$
\begin{aligned}
& \mathit{rel} \\
= \quad & \{(8)\} \\
& \mathit{map\ sep} \circ \mathit{subtrees} \\
= \quad & \{(15)\} \\
& \mathit{map\ spread} \circ \mathit{map\ contours} \circ \mathit{subtrees} \\
= \quad & \{(17)\} \\
& \mathit{map\ spread} \circ \mathit{map}\ (\mathit{bh}\ (\mathit{const}\ ([0],[0]), \boxminus)) \circ \mathit{subtrees} \\
= \quad & \{(3)\} \\
& \mathit{map\ spread} \circ \mathit{up}\ (\mathit{const}\ ([0],[0]), \boxminus)
\end{aligned}
$$

That is,

$$
\mathit{rel} \quad = \quad \mathit{map\ spread} \circ \mathit{up}\ (\mathit{const}\ ([0],[0]), \boxminus) \tag{18}
$$

This is now an upwards accumulation, but it is still expensive to compute. The operation $\boxminus$ takes at least linear effort, resulting in quadratic effort for the upwards accumulation. One further step is needed before we have an efficient algorithm for *rel*.

We have to find an efficient way of evaluating the operator $\boxminus$ from (16):

$$
\begin{aligned}
(w,x) \boxminus_a (y,z) \quad = \quad & ([0] \mathbin{+\!\!+} \mathit{lzip\ fst}\ (\mathit{map}\ (-s)\ w, \mathit{map}\ (+s)\ y), \\
& \ [0] \mathbin{+\!\!+} \mathit{lzip\ snd}\ (\mathit{map}\ (-s)\ x, \mathit{map}\ (+s)\ z)) \\
& \qquad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2
\end{aligned}
$$

One way of doing this is with a data refinement whereby, instead of maintaining a list of absolute distances, we maintain a list of relative distances. That is, we make a data refinement using the invertible abstraction function $\mathit{msi} = \mathit{map\ sum} \circ \mathit{inits}$, which computes absolute distances from relative ones. Under this refinement, the maps can be performed in constant time, since

$$
\begin{aligned}
\mathit{map}\ (+s)\ (\mathit{msi}\ x) \quad = \quad & \mathit{msi}\ (\mathit{mapplus}\ (s,x)) \\
& \text{where} \quad \mathit{mapplus}\ (b,[a]) \quad = \quad [b+a] \\
& \qquad\quad \mathit{mapplus}\ (b,[a] \mathbin{+\!\!+} x) \quad = \quad [b+a] \mathbin{+\!\!+} x
\end{aligned}
\tag{19}
$$

Moreover, the zips can still be performed in time proportional to their shorter argument, since if $\mathit{len}\ x \geq \mathit{len}\ y$ then

$$
\mathit{lzip\ fst}\ (\mathit{msi}\ x, \mathit{msi}\ y) \quad = \quad \mathit{msi}\ x
$$

and if $len\ x < len\ y$ then, letting $(y_1, y_2) = split\ (len\ x, y)$ where

$$\begin{array}{rcll} split\ (1, [a] +\!\!+ x) & = & ([a], x) & \\ split\ (n+1, [a] +\!\!+ x) & = & ([a] +\!\!+ v, w) & \quad \text{where} \quad (v, w) = split\ (n, x) \end{array}$$

we have

$$\begin{array}{cl} & lzip\ fst\ (msi\ x, msi\ y) \\ = & \quad \{msi\ y = msi\ y_1 +\!\!+ map\ (+sum\ y_1)\ (msi\ y_2);\ len\ x = len\ y_1\} \\ & msi\ x +\!\!+ map\ (+sum\ y_1)\ (msi\ y_2) \\ = & \quad \{map\ (+sum\ x) \circ map\ (-sum\ x) = id\} \\ & msi\ x +\!\!+ map\ (+sum\ x)\ (map\ (-sum\ x + sum\ y_1)\ (msi\ y_2)) \\ = & \quad \{(19)\} \\ & msi\ x +\!\!+ map\ (+sum\ x)\ (msi\ (mapplus\ (sum\ y_1 - sum\ x, y_2))) \\ = & \quad \{msi\ (x +\!\!+ y) = msi\ x +\!\!+ map\ (+sum\ x)\ (msi\ y)\} \\ & msi\ (x +\!\!+ mapplus\ (sum\ y_1 - sum\ x, y_2)) \end{array}$$

By symmetry,

$$lzip\ snd\ (msi\ x, msi\ y) \quad = \quad lzip\ fst\ (msi\ y, msi\ x)$$

(Note that the guard $len\ x \geq len\ y$ must also be evaluated in time proportional to the lesser of $len\ x$ and $len\ y$, and so cannot be done simply by computing the two lengths. In Figure 6 we define the predicate *nst* (for 'no shorter than'), for which $nst\ (x, y) = (len\ x \geq len\ y)$ but which takes time proportional to the lesser of $len\ x$ and $len\ y$.)

The refined $\boxminus$ still takes linear effort because of the zips, 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 $\boxminus$ is 'called' with arguments (w, x) and (y, z)). Reingold and Tilford (1981) show that, if evaluating $h\ t \oplus_a h\ u$ from a, $h\ t$ and $h\ u$ takes effort proportional to the lesser of the depths of the trees t and u, then the tree homomorphism $h = bh\ (f, \oplus)$ can be evaluated with linear effort. Actually, what they show is that if g satisfies

$$\begin{array}{rcl} g\ (lf\ a) & = & 0 \\ g\ (br\ (t, a, u)) & = & g\ t + min\ (depth\ t, depth\ u) + g\ u \end{array}$$

then

$$g\ x \quad = \quad size\ x - depth\ x$$

which can easily be proved by induction. Intuitively, g counts the number of pairs of horizontally adjacent elements in a tree.

With this data refinement, *rel* can be computed in linear time.

5.2 A downwards accumulation

We now have an efficient algorithm for *rel*. All that remains to be done is to find an efficient algorithm for *abs*, where

$$\begin{array}{rcl} abs & = & map\ pabs \circ paths \\ pabs & = & uw\ (const\ 0, \tilde{-}, +) \end{array}$$

We note first that computing *abs* as it stands *is* inefficient. No operator $\oplus$ can satisfy $a + const\ 0\ \ b = const\ 0\ \ a \oplus b$ for all a and b, and so *pabs* cannot be computed downwards, and *abs* is not a downwards accumulation. Intuitively, *pabs* starts at the bottom of a path and discards the bottom element, but we cannot do this when starting at the top of the path.

What extra information do we need in order to be able to compute *pabs* downwards? It turns out that

$$\begin{array}{rcl} pabs\ (x \overleftarrow{+\!\!+} \langle a \rangle) & = & pabs\ x - bottom\ x \\ pabs\ (x \overrightarrow{+\!\!+} \langle a \rangle) & = & pabs\ x + bottom\ x \end{array} \tag{20}$$

where *bottom* returns the bottom element of a path:

$$bottom \quad = \quad uw\ (id, snd, snd)$$

Now, *pabs* and *bottom* together can be computed downwards, because of (20) and

$$\begin{array}{rcl} bottom\ (x \overleftarrow{+\!\!+} \langle a \rangle) & = & a \\ bottom\ (x \overrightarrow{+\!\!+} \langle a \rangle) & = & a \end{array}$$

Let

$$pabsb \quad = \quad fork\ (pabs, bottom) \tag{21}$$

Then, by Theorem 6, *pabsb* is upwards:

$$\begin{array}{rclcrcl} pabsb & = & uw\ (f, \oplus, \otimes) & \quad \text{where} \quad & f\ a & = & (0, a) \\ & & & & a \oplus (v, w) & = & (v - a, w) \\ & & & & a \otimes (v, w) & = & (v + a, w) \end{array}$$

Moreover, by Theorem 5, *pabsb* is downwards:

$$\begin{array}{rclcrcl} pabsb & = & dw\ (f, \oplus, \otimes) & \quad \text{where} \quad & f\ a & = & (0, a) \\ & & & & (v, w) \oplus a & = & (v - w, a) \\ & & & & (v, w) \otimes a & = & (v + w, a) \end{array}$$

Finally, by Theorem 3, *pabsb* is a path homomorphism:

$$\begin{array}{rcl} pabsb & = & ph\ (f, \oplus, \otimes) \\ & & \quad \text{where} \quad \begin{array}{rcl} f\ a & = & (0, a) \\ (v, w) \oplus (x, y) & = & (v - w + x, y) \\ (v, w) \otimes (x, y) & = & (v + w + x, y) \end{array} \end{array} \tag{22}$$

Putting all this together gives us

$$\begin{array}{ll} & abs \\ = & \quad \{(11)\} \end{array}$$

$$
\begin{array}{ll}
& map\ pabs \circ paths \\
= & \quad \{(21)\} \\
& map\ fst \circ map\ pabsb \circ paths \\
= & \quad \{(22), \text{ with } f, \oplus \text{ and } \otimes \text{ as defined there}\} \\
& map\ fst \circ map\ (ph\ (f, \oplus, \otimes)) \circ paths \\
= & \quad \{(4)\} \\
& map\ fst \circ down\ (f, \oplus, \otimes)
\end{array}
$$

That is,

$$abs \quad = \quad map\ fst \circ down\ (f, \oplus, \otimes) \tag{23}$$

which can be computed in linear time.

5.3 The program

To summarize, the program that we have derived is as in Figure 6.

6 Conclusion

6.1 Summary

We have presented a number of natural criteria satisfied by tidy drawings of unlabelled binary trees. From these criteria, we have derived an efficient algorithm for producing such drawings.

The steps in the derivation were as follows:

1. we started with an executable specification (5)—an 'obviously correct' but inefficient program;
2. we eliminated one source of inefficiency, by computing first the position of every parent relative to its children, and then fixing the absolute positions in a second pass (12);
3. we made a step towards making the first pass efficient, by turning the function computing relative positions into an upwards accumulation (18), computing not just relative positions but also the outlines of the drawings;
4. we made a data refinement on the outline of a drawing (19), allowing us to shift it in constant time; and
5. we made the second pass efficient by turning the function computing absolute positions into a downwards accumulation (23), computing not just the absolute positions but also the bottom element of every path. (In fact, we could have calculated, using the technique of *strengthening invariants* (Gries, 1982) and no invention at all, that

$$fork\ (pabs, uw\ (id, \tilde{-}, +))$$

$$
\begin{aligned}
\mathit{bdraw} &= \mathit{abs} \circ \mathit{rel} \\[4pt]
\mathit{rel} &= \mathit{map}\ \mathit{spread} \circ \mathit{up}\ (\mathit{const}\ ([0],[0]), \boxminus) \\
(w,x) \boxminus_a (y,z) &= ([0] \mathbin{+\!\!+} \mathit{lzipfst}\ (\mathit{mapplus}\ (\text{-}s, w), \mathit{mapplus}\ (s,y)), \\
&\qquad [0] \mathbin{+\!\!+} \mathit{lzipsnd}\ (\mathit{mapplus}\ (\text{-}s, x), \mathit{mapplus}\ (s,z))) \\
&\qquad\quad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2 \\[4pt]
\mathit{mapplus}\ (b, [a]) &= [a+b] \\
\mathit{mapplus}\ (b, [a] \mathbin{+\!\!+} x) &= [a+b] \mathbin{+\!\!+} x \\[4pt]
\mathit{lzipfst}\ (x,y) &= x, \qquad \text{if } \mathit{nst}\ (x,y) \\
&= x \mathbin{+\!\!+} \mathit{mapplus}\ (\mathit{sum}\ v - \mathit{sum}\ x, w), \quad \text{otherwise} \\
&\qquad\quad \text{where} \quad (v,w) = \mathit{split}\ (\mathit{len}\ x, y) \\
\mathit{lzipsnd}\ (x,y) &= \mathit{lzipfst}\ (y,x) \\[4pt]
\mathit{nst}\ (x, [b]) &= \mathit{true} \\
\mathit{nst}\ ([a], [b] \mathbin{+\!\!+} y) &= \mathit{false} \\
\mathit{nst}\ ([a] \mathbin{+\!\!+} x, [b] \mathbin{+\!\!+} y) &= \mathit{nst}\ (x,y) \\[4pt]
\mathit{split}\ (1, [a] \mathbin{+\!\!+} x) &= ([a], x) \\
\mathit{split}\ (n+1, [a] \mathbin{+\!\!+} x) &= ([a] \mathbin{+\!\!+} v, w) \qquad \text{where} \quad (v,w) = \mathit{split}\ (n,x) \\[4pt]
\mathit{spread}\ ([0],[0]) &= 0 \\
\mathit{spread}\ ([0] \mathbin{+\!\!+} x, [0] \mathbin{+\!\!+} y) &= \text{-}(\mathit{head}\ x) \odot \mathit{head}\ y \qquad \text{where} \quad a \odot a = a \\[4pt]
v \boxtimes w &= \mathit{lh}\ (\mathit{id}, \mathit{min})\ (\mathit{szip}\ (\tilde{-})\ (v,w)) \\[4pt]
\mathit{abs} &= \mathit{map}\ \mathit{fst} \circ \mathit{down}\ (f, \oplus, \otimes) \\
&\qquad\quad \text{where} \quad
\begin{aligned}[t]
f\ a &= (0,a) \\
(v,w) \oplus (x,y) &= (v - w + x, y) \\
(v,w) \otimes (x,y) &= (v + w + x, y)
\end{aligned}
\end{aligned}
$$

Fig. 6. The final program

is downwards, and hence also a path homomorphism; this would have done just as well.)

The derivation showed several things:

1. the criteria uniquely determine the drawing of a tree;
2. the criteria also determine an inefficient algorithm for drawing a tree (step 1 in the derivation), and only three or four small inventive steps (steps 2 to 5 in the derivation) are needed to transform this into an efficient algorithm;
3. the algorithm (due to Reingold and Tilford (1981)) is just an upwards accumulation followed by a downwards accumulation, and is further evidence of the utility of these higher-order operations;
4. identifying these accumulations as major components of the algorithm may lead, using known techniques for computing accumulations in parallel, to an optimal *parallel* algorithm for drawing unlabelled binary trees.

6.2 Related work

The problem of drawing trees has quite a long and interesting history. Knuth (1968; 1971) 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 the algorithms of Wetherell and Shannon and of Vaucher, 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 problem of drawing general trees has had rather less coverage in the literature. General trees are harder to draw than binary trees, because it is not so clear what is meant by 'placing siblings as close as possible'. For example, consider a general tree with three children, t, u and v, in which t and v are large but u relatively small. It is not sufficient to consider just adjacent pairs of siblings when spacing the siblings out, because t may collide with v. Spacing the siblings out so that t and v do not collide allows some freedom in placing u, and care must be taken not to introduce any bias. Reingold and Tilford (1981) mention general trees in passing, but make no reference to the difficulty of producing unbiased drawings. Bloesch (1993) (who adapts the algorithms of Vaucher and of Reingold and Tilford to cope with node labels of varying width and height) also does not attempt to produce unbiased drawings, despite his claims to the contrary. Radack (1988) effectively constructs two drawings, one packing siblings together from the left and the other from the right, and then averages the results. That algorithm is derived by Gibbons (1991) and described by Kennedy (1995). 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.

6.3 Further work

Gibbons (1991) extends this derivation to general trees. We have yet to apply the methods used here to Bloesch's algorithm (Bloesch, 1993) for drawing trees in which the labels may have different heights, but do not expect it to yield any surprises. It may also be possible to apply the techniques in (Gibbons *et al.*, 1994) to yield an optimal *parallel* algorithm to draw a binary tree of n elements in $\log n$ time on

$n/\log n$ processors, even when the tree is unbalanced—although this is complicated by having to pass non-constant-size contours around in computing $\boxminus$.

We are currently exploring the application to graphs of some of the general notions—homomorphisms and accumulations—used here on lists and trees. See (Gibbons, 1994b) for further details.

6.4 Acknowledgements

Thanks are due to Sue Gibbons and the anonymous referees, whose suggestions improved the presentation of this paper considerably.

References

Roland Backhouse (1989). *An exploration of the Bird-Meertens formalism.* In *International Summer School on Constructive Algorithmics, Hollum, Ameland.* STOP project. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.

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.

Anthony Bloesch (1993). *Aesthetic layout of generalized trees.* Software—Practice and Experience, 23(8):817–827.

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.

Pierre Deransart, Martin Jourdan, and Bernard Lorho (1988). *LNCS 323: Attribute Grammars—Definitions, Systems and Bibliography.* Springer-Verlag.

Jeremy Gibbons, Wentong Cai, and David Skillicorn (1994). *Efficient parallel algorithms for tree accumulations.* Science of Computer Programming, 23:1–18.

Jeremy Gibbons (1991). *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University. Available as Technical Monograph PRG-94.

Jeremy Gibbons (1993a). *Computing downwards accumulations on trees quickly.* In Gopal Gupta, George Mohay, and Rodney Topor, editors, *16th Australian Computer Science Conference*, pages 685–691, Brisbane. Revised version submitted for publication.

Jeremy Gibbons (1993b). *Upwards and downwards accumulations on trees.* In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 122–138. Springer-Verlag. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.

Jeremy Gibbons (1994a). *How to derive tidy drawings of trees.* In C. Calude, M. J. J. Lennon, and H. Maurer, editors, *Proceedings of Salodays in Auckland*, pages 53–73, Department of Computer Science, University of Auckland.

Jeremy Gibbons (1994b). *An initial-algebra approach to directed acyclic graphs.* Department of Computer Science, University of Auckland. Accepted for publication in *Mathematics of Program Construction 1995.*

Jeremy Gibbons (1994c). *The Third Homomorphism Theorem.* In C. Barry Jay, editor, *Computing: The Australian Theory Seminar.* University of Technology, Sydney. Submitted for publication.

David Gries (1982). *A note on a standard strategy for developing loop invariants and loops.* Science of Computer Programming, 2:207–214.

Andrew Kennedy (1995). *Drawing trees.* Journal of Functional Programming, to appear.

Donald E. Knuth (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley.

Donald E. Knuth (1971). *Optimum binary search trees.* Acta Informatica, 1:14–25.

Richard E. Ladner and Michael J. Fischer (1980). *Parallel prefix computation.* Journal of the ACM, 27(4):831–838.

Grant Malcolm (1990). *Algebraic Data Types and Program Transformation.* PhD thesis, Rijksuniversiteit Groningen.

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.

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.

David B. Skillicorn (1993). *Parallel evaluation of structured queries in text.* Draft, Department of Computing and Information Sciences, Queen's University, Kingston, Ontario.

Kenneth J. Supowit and Edward M. Reingold (1983). *The complexity of drawing trees nicely.* Acta Informatica, 18(4):377–392.

Jean G. Vaucher (1980). *Pretty-printing of trees.* Software—Practice and Experience, 10:553–561.

John Q. Walker, II (1990). *A node-positioning algorithm for general trees.* Software—Practice and Experience, 20(7):685–705.

Charles Wetherell and Alfred Shannon (1979). *Tidy drawings of trees.* IEEE Transactions on Software Engineering, 5(5):514–520.

Niklaus Wirth (1976). *Algorithms + Data Structures = Programs.* Prentice Hall.