How to derive 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, there is essentially only one reasonable drawing algorithm satisfying the criteria: its development is almost mechanical.

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, program transformation, trees, upwards and downwards accumulations, drawing, layout.

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.

The algorithm that we derive (which is due originally to Reingold and Tilford (1981)) consists of an *upwards accumulation* followed by a *downwards accu-*

Copyright ©1994 Jeremy Gibbons. This extended abstract dated 1st November 1994. An earlier version appears in *Proceedings of Salodays in Auckland*, C. Calude, M. J. J. Lennon and H. Maurer, eds., Auckland, 1994. Full paper submitted for publication and available as Computer Science Report No. 82 from the above address. This work has been partially supported by University of Auckland Research Committee grant number A18/XXXXX/62090/3414013.

mulation (Gibbons, 1991, 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., 1993) 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 in (Gibbons, 1991). We do not discuss it here, because to do so would entail a significant increase in the number of definitions required.

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; more of the motivation for these definitions is given in the full paper. 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 sketched in Section 5; the details can be found in the full paper.

2 Notation

We will use the *Bird-Meertens Formalism* or 'BMF' (Meertens, 1986; Bird, 1987, 1988; Backhouse, 1989), 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 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* $\mathsf{always}(a)$ returns a for every argument; for example, $\mathsf{always}(1)(2) = 1$. (Function application is left-associative, so that this parses as '$(\mathsf{always}(1))(2)$'.) Function composition is written '$\circ$'; for example, $\mathsf{always}(1) \circ \mathsf{always}(2) = \mathsf{always}(1)$. The *identity function* is written 'id'. The *converse* $\mathsf{conv}(\oplus)$ of a binary operator $\oplus$ is obtained by swapping its arguments; for example, $\mathsf{conv}(-)(x, y) = y - x$.

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

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,

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

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 $\mathsf{list}(A)$ 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 '$[\cdot]$' 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, $\mathsf{map}(+1)([1, 2, 3]) = [2, 3, 4]$. 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,

$$\mathsf{h}(x +\!\!+ y) \;=\; \mathsf{h}(x) \otimes \mathsf{h}(y)$$

The condition of associativity on $\otimes$ is no great restriction. If h is $+\!\!+$ to $\otimes$ promotable then $\otimes$ is necessarily associative, at least on the range of h. In fact, if h is $+\!\!+$ to $\otimes$ promotable, then it is completely determined by its action on singleton lists; for example,

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

If h is $+\!\!+$ to $\otimes$ promotable and $\mathsf{h} \circ [\cdot] = \mathsf{f}$, then we write h as $\mathsf{lh}(\mathsf{f}, \otimes)$ ('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

$$\mathsf{h} \circ \mathsf{lh}(\mathsf{f}, \oplus) \;=\; \mathsf{lh}(\mathsf{h} \circ \mathsf{f}, \otimes)$$

$\diamondsuit$

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

For each f, $\mathsf{map}(\mathsf{f})$ is a homomorphism, for

$$\mathsf{map}(\mathsf{f})(x +\!\!+ y) \;=\; \mathsf{map}(\mathsf{f})(x) +\!\!+ \mathsf{map}(\mathsf{f})(y)$$

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

$$\mathsf{len} \;=\; \mathsf{lh}(\mathsf{always}(1), +)$$

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

$$\mathsf{head}(x +\!\!+ y) \;=\; \mathsf{head}(x) \;=\; \mathsf{fst}(\mathsf{head}(x), \mathsf{head}(y))$$

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

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

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

$$\mathsf{sum} \;=\; \mathsf{lh}(\mathsf{id}, +)$$

2.5 Binary trees

Finally, we come to binary trees. The type $\mathsf{btree}(A)$ consists of binary trees labelled with elements of type A. A binary tree is either a leaf $\mathsf{lf}(a)$ labelled with a single

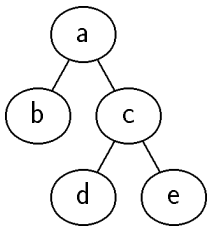

Figure 1: The tree five

element a, or a branch br(t, a, u) consisting of two children t and u and a label a. For example, the expression

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

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

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

$$\begin{aligned} \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{lf}(\mathsf{a})) &= \mathsf{f}(\mathsf{a}) \\ \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{t}) \oplus_{\mathsf{a}} \mathsf{bh}(\mathsf{f}, \oplus)(\mathsf{u}) \end{aligned}$$

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

$$\mathsf{h}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) = \mathsf{h}(\mathsf{t}) \oplus_{\mathsf{a}} \mathsf{h}(\mathsf{u})$$

then $\mathsf{h} = \mathsf{bh}(\mathsf{h} \circ \mathsf{lf}, \oplus)$. ♢

The function map on binary trees satisfies

$$\begin{aligned} \mathsf{map}(\mathsf{f})(\mathsf{lf}(\mathsf{a})) &= \mathsf{lf}(\mathsf{f}(\mathsf{a})) \\ \mathsf{map}(\mathsf{f})(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{br}(\mathsf{map}(\mathsf{f})(\mathsf{t}), \mathsf{f}(\mathsf{a}), \mathsf{map}(\mathsf{f})(\mathsf{u})) \end{aligned}$$

and so

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

The function root is a binary tree homomorphism:

$$\begin{aligned} \mathsf{root}(\mathsf{lf}(\mathsf{a})) &= \mathsf{a} \\ \mathsf{root}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) &= \mathsf{a} \end{aligned}$$

and so

$$\mathsf{root} = \mathsf{bh}(\mathsf{id}, \oplus) \quad \text{where} \quad \mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = \mathsf{a}$$

So are the functions size and depth:

$$\begin{array}{rcll} \mathsf{size} & = & \mathsf{bh}(\mathsf{always}(1), \oplus) & \text{where} \quad \mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = \mathsf{v} + 1 + \mathsf{w} \\ \mathsf{depth} & = & \mathsf{bh}(\mathsf{always}(1), \oplus) & \text{where} \quad \mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = 1 + \mathsf{max}(\mathsf{v}, \mathsf{w}) \end{array}$$

and the function brev, which reverses a binary tree:

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

2.6 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 $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$. Unary functions are denoted $\mathsf{f}, \mathsf{g}, \mathsf{h}$. Lists and paths (introduced in Section 3.2) are denoted $\mathsf{w}, \mathsf{x}, \mathsf{y}, \mathsf{z}$. 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 presented only in summary; a more complete description, including motivation, is given in the full paper and in (Gibbons, 1991).

3.1 Upwards accumulations

Upwards and downwards accumulations arise from considering the list function inits, which takes a list x and returns the list of lists consisting of the non-empty initial segments of x in order of increasing length. 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:

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

which corresponds to the tree of trees in Figure 2. The function subtrees satisfies

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

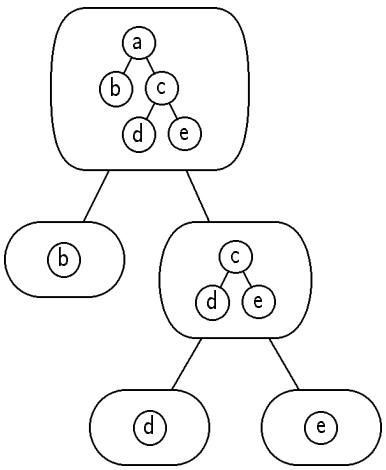

Figure 2: The subtrees of five

The *upwards accumulation* $\mathsf{up}(\mathsf{f}, \oplus)$ is obtained by mapping the tree homomorphism $\mathsf{bh}(\mathsf{f}, \oplus)$ over the subtrees of a tree:

$$\mathsf{up}(\mathsf{f}, \oplus) \;=\; \mathsf{map}(\mathsf{bh}(\mathsf{f}, \oplus)) \circ \mathsf{subtrees}$$

It can be computed in linear time (assuming that the f and $\oplus$ take constant time).

One example of an upwards accumulation is the function ndescs, which replaces every element with the number of descendents it has. Letting $\oplus$ satisfy $\mathsf{v} \oplus_{\mathsf{a}} \mathsf{w} = \mathsf{v} + 1 + \mathsf{w}$, so that $\mathsf{size} = \mathsf{bh}(\mathsf{always}(1), \oplus)$, we have

$$\begin{aligned} \mathsf{ndescs} &= \mathsf{map}(\mathsf{bh}(\mathsf{always}(1), \oplus)) \circ \mathsf{subtrees} \\ &= \mathsf{up}(\mathsf{always}(1), \oplus) \end{aligned}$$

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 option. The type $\mathsf{path}(\mathsf{A})$ consists of paths of elements of type A. A path is either a single element $\langle \mathsf{a} \rangle$ or two paths x and y joined with a 'left turn', $\mathsf{x} \mathbin{+\!\!\!\!\!\!\leftarrow} \mathsf{y}$, or a 'right turn', $\mathsf{x} \mathbin{+\!\!\!\!\!\!\rightarrow} \mathsf{y}$. The function taking a to $\langle \mathsf{a} \rangle$ is written '$\langle \cdot \rangle$'. Just as $+\!\!+$ is associative, the operations $+\!\!\!\!\!\!\leftarrow$ and $+\!\!\!\!\!\!\rightarrow$ satisfy the four laws

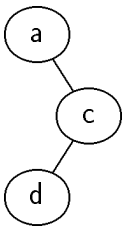

Figure 3: The path in five to the element labelled d

$$\begin{aligned}
x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) &= (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \\
x \mathbin{+\!\!+} (y \mathbin{+\!\!\Rightarrow} z) &= (x \mathbin{+\!\!+} y) \mathbin{+\!\!\Rightarrow} z \\
x \mathbin{+\!\!\Rightarrow} (y \mathbin{+\!\!+} z) &= (x \mathbin{+\!\!\Rightarrow} y) \mathbin{+\!\!+} z \\
x \mathbin{+\!\!\Rightarrow} (y \mathbin{+\!\!\Rightarrow} z) &= (x \mathbin{+\!\!\Rightarrow} y) \mathbin{+\!\!\Rightarrow} z
\end{aligned}$$

We say that '$+\!\!+$ associates with $+\!\!\Rightarrow$', or '$+\!\!+$ and $+\!\!\Rightarrow$ associate with each other'. Thus, the path shown above to the element labelled d is represented by $\langle a \rangle +\!\!\Rightarrow \langle c \rangle +\!\!+ \langle d \rangle$. Because of the associativity property, brackets are unnecessary.

Path homomorphisms promote through both $+\!\!+$ and $+\!\!\Rightarrow$; if, for all a, x and y, the function h satisfies

$$\begin{aligned}
h(\langle a \rangle) &= f(a) \\
h(x \mathbin{+\!\!+} y) &= h(x) \oplus h(y) \\
h(x \mathbin{+\!\!\Rightarrow} y) &= h(x) \otimes h(y)
\end{aligned}$$

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

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

$$\begin{aligned}
h(\langle a \rangle) &= f(a) \\
h(\langle a \rangle \mathbin{+\!\!+} y) &= a \oplus h(y) \\
h(\langle a \rangle \mathbin{+\!\!\Rightarrow} y) &= a \otimes h(y)
\end{aligned}$$

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

$$\begin{aligned}
h(\langle a \rangle) &= f(a) \\
h(x \mathbin{+\!\!+} \langle a \rangle) &= h(x) \oplus a \\
h(x \mathbin{+\!\!\Rightarrow} \langle a \rangle) &= h(x) \otimes a
\end{aligned}$$

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 (Gibbons, 1994) 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 homomor-

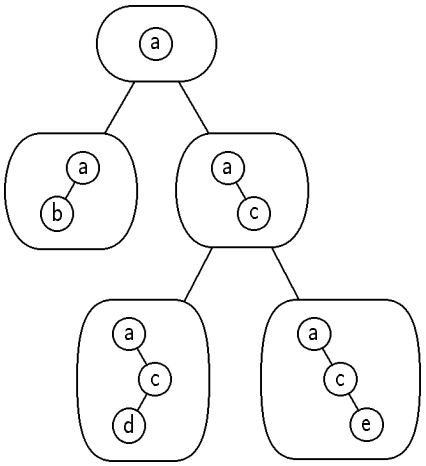

Figure 4: The paths of five

phism. ♢

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:

$$\begin{aligned}\mathsf{paths}(\mathsf{five}) \;=\; & \mathsf{br}(\mathsf{lf}(\langle \mathsf{a}\rangle \twoheadleftarrow \langle \mathsf{b}\rangle),\\ & \quad \langle \mathsf{a}\rangle,\\ & \quad \mathsf{br}(\mathsf{lf}(\langle \mathsf{a}\rangle \twoheadrightarrow \langle \mathsf{c}\rangle \twoheadleftarrow \langle \mathsf{d}\rangle),\\ & \qquad \langle \mathsf{a}\rangle \twoheadrightarrow \langle \mathsf{c}\rangle,\\ & \qquad \mathsf{lf}(\langle \mathsf{a}\rangle \twoheadrightarrow \langle \mathsf{c}\rangle \twoheadrightarrow \langle \mathsf{e}\rangle))))\end{aligned}$$

which corresponds to the tree of paths in Figure 4. The function paths is a tree homomorphism; it satisfies

$$\begin{aligned}\mathsf{paths}(\mathsf{br}(\mathsf{t},\mathsf{a},\mathsf{u})) \;=\; & \mathsf{br}(\mathsf{map}(\langle \mathsf{a}\rangle \twoheadleftarrow)(\mathsf{paths}(\mathsf{t})),\\ & \quad \langle \mathsf{a}\rangle,\\ & \quad \mathsf{map}(\langle \mathsf{a}\rangle \twoheadrightarrow)(\mathsf{paths}(\mathsf{u})))\end{aligned}$$

The *downwards accumulation* $\mathsf{down}(\mathsf{f},\oplus,\otimes)$ is obtained by mapping the path homomorphism $\mathsf{ph}(\mathsf{f},\oplus,\otimes)$ over the paths of a tree:

$$\mathsf{down}(\mathsf{f},\oplus,\otimes) \;=\; \mathsf{map}(\mathsf{ph}(\mathsf{f},\oplus,\otimes)) \circ \mathsf{paths}$$

Note that $\oplus$ and $\otimes$ must associate with each other for the path homomorphism to be valid.

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:

$$\mathsf{depths} \;=\; \mathsf{map}(\mathsf{plen}) \circ \mathsf{paths}$$

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

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

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

$$\mathsf{plen} \;=\; \mathsf{dw}(\mathsf{always}(1), \oplus, \oplus) \qquad \text{where} \quad \mathsf{v} \oplus \mathsf{a} = \mathsf{v} + 1$$

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

$$\mathsf{depths} \;=\; \mathsf{down}(\mathsf{always}(1), +, +)$$

(since $+$ is associative, it associates with itself) shows that depths is both homomorphic and 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 a 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 $\mathsf{btree}(\mathsf{A})$ for some A, and its result is a binary tree labelled with x-coordinates:

$$\mathsf{bdraw} \;\in\; \mathsf{btree}(\mathsf{A}) \to \mathsf{btree}(\mathbb{D})$$

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:

$$\mathsf{bdraw}(\mathsf{br}(\mathsf{t}, \mathsf{a}, \mathsf{u})) \;=\; \mathsf{br}(\mathsf{map}(+\mathsf{r})(\mathsf{bdraw}(\mathsf{t})), \mathsf{b}, \mathsf{map}(+\mathsf{s})(\mathsf{bdraw}(\mathsf{u})))$$

for some b, r and s.

Tidy drawings also exhibit no left-to-right bias. In particular, a parent should

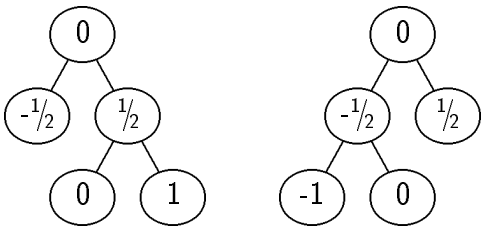

Figure 5: Drawings pic_1 and pic_2, for which $\mathsf{pic}_1 \boxplus \mathsf{pic}_2 = \text{-}2$

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:

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

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[1], then we also require

$$\mathsf{bdraw} \circ \mathsf{brev} = \mathsf{map}(\text{-}) \circ \mathsf{brev} \circ \mathsf{bdraw}$$

The fourth criterion is that, in a tidy drawing, elements do not collide, or even get too close together. That is, 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) 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 $\mathsf{pic}_1 \boxplus \mathsf{pic}_2 = \text{-}2$. The drawings should be moved apart or together to make this distance 1, that is,

[1]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)'.

$$\mathsf{s} \;=\; (1 - (\mathsf{bdraw}(\mathsf{t}) \boxplus \mathsf{bdraw}(\mathsf{u}))) \div 2$$

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

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:

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

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

The two zips are given formally by the equations

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

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

Note that both $\mathsf{szip}(\oplus)(\mathsf{x},\mathsf{y})$ and $\mathsf{lzip}(\oplus)(\mathsf{x},\mathsf{y})$ can be evaluated with $\mathsf{min}(\mathsf{len}(\mathsf{x}),\mathsf{len}(\mathsf{y}))$ applications of $\oplus$.

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

$$\mathsf{levels} \;=\; \mathsf{bh}([\cdot]\circ[\cdot],\oplus) \qquad \text{where} \quad \mathsf{x}\oplus_{\mathsf{a}}\mathsf{y} = [[\mathsf{a}]]+\!\!+\mathsf{lzip}(+\!\!+)(\mathsf{x},\mathsf{y})$$

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

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

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

$$\begin{aligned} \mathsf{p} \boxplus \mathsf{q} \;=\; & \mathsf{smallest}(\mathsf{szip}(\mathsf{conv}(-))(\mathsf{map}(\mathsf{largest})(\mathsf{levels}(\mathsf{p})), \\ & \qquad \mathsf{map}(\mathsf{smallest})(\mathsf{levels}(\mathsf{q})))) \end{aligned}$$

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

$$\begin{aligned} \mathsf{map}(\mathsf{largest})(\mathsf{levels}(\mathsf{pic}_1)) &= [0, \tfrac{1}{2}, 1] \\ \mathsf{map}(\mathsf{smallest})(\mathsf{levels}(\mathsf{pic}_2)) &= [0, -\tfrac{1}{2}, -1] \end{aligned}$$

and so

$$\mathsf{pic}_1 \boxplus \mathsf{pic}_2 \;=\; \mathsf{smallest}([0-0, -\tfrac{1}{2}-\tfrac{1}{2}, -1-1]) \;=\; -2$$

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

$$\mathsf{bdraw} \;=\; \mathsf{bh}(\mathsf{always}(\mathsf{lf}(0)), \oplus) \qquad \text{— (1)}$$

where

$$\begin{aligned} \mathsf{p} \oplus_{\mathsf{a}} \mathsf{q} \;=\; & \mathsf{br}(\mathsf{map}(-\mathsf{s})(\mathsf{p}), 0, \mathsf{map}(+\mathsf{s})(\mathsf{q})) \qquad \text{where} \quad \mathsf{s} = (1 - (\mathsf{p} \boxplus \mathsf{q})) \div 2 \\ \mathsf{p} \boxplus \mathsf{q} \;=\; & \mathsf{smallest}(\mathsf{szip}(\mathsf{conv}(-))(\mathsf{map}(\mathsf{largest})(\mathsf{levels}(\mathsf{p})), \\ & \qquad \mathsf{map}(\mathsf{smallest})(\mathsf{levels}(\mathsf{q})))) \end{aligned}$$

This specification is executable, but requires quadratic effort. We now sketch the derivation of a linear algorithm to satisfy it.

5 Drawing binary trees efficiently

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 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* 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 $\mathsf{rootrel}$ on drawings of trees satisfies

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

for some idempotent operator $\odot$. The idea here is that $\mathsf{rootrel}$ 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

$$\mathsf{rootrel}(\mathsf{pic}_1) \;=\; (0 - \text{-}\tfrac{1}{2}) \odot (\tfrac{1}{2} - 0) \;=\; \tfrac{1}{2}$$

That is, if we define the function sep by

$$\mathsf{sep} \;=\; \mathsf{rootrel} \circ \mathsf{bdraw}$$

then

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

For example:

$$\begin{aligned} \mathsf{sep}(\mathsf{five}) &= (1 - (\mathsf{bdraw}(\mathsf{lf}(b)) \boxplus \mathsf{bdraw}(\mathsf{br}(\mathsf{lf}(d),c,\mathsf{lf}(e))))) \div 2 \\ &= (1 - 0) \div 2 \\ &= \tfrac{1}{2} \end{aligned}$$

Then

$$\begin{aligned} \mathsf{bdraw}(\mathsf{br}(t,a,u)) \;=\; & \mathsf{br}(\mathsf{map}(-s)(\mathsf{bdraw}(t)), 0, \mathsf{map}(+s)(\mathsf{bdraw}(u))) \\ & \text{where} \quad s = \mathsf{sep}(\mathsf{br}(t,a,u)) \end{aligned}$$

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

$$\mathsf{rel} \;=\; \mathsf{map}(\mathsf{sep}) \circ \mathsf{subtrees}$$

From this, we can (and in the full paper, do) calculate that

$$\begin{aligned} \mathsf{rel}(\mathsf{lf}(a)) &= \mathsf{lf}(0) \\ \mathsf{rel}(\mathsf{br}(t,a,u)) &= \mathsf{br}(\mathsf{rel}(t), \mathsf{sep}(\mathsf{br}(t,a,u)), \mathsf{rel}(u)) \end{aligned}$$

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

$$\mathsf{abs} \circ \mathsf{rel} \;=\; \mathsf{bdraw}$$

We can (and again, in the full paper, do) calculate from this requirement a definition of abs:

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

This is equivalent to

$$\mathsf{abs} \;=\; \mathsf{map}(\mathsf{uw}(\mathsf{always}(0), \mathsf{conv}(-), +)) \circ \mathsf{paths}$$

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

$$\mathsf{pabs} \;=\; \mathsf{uw}(\mathsf{always}(0), \mathsf{conv}(-), +)$$

so that

$$\mathsf{abs} \;=\; \mathsf{map}(\mathsf{pabs}) \circ \mathsf{paths}$$

Thus, we have

$$\mathsf{bdraw} \;=\; \mathsf{abs} \circ \mathsf{rel} \qquad \text{— (2)}$$

where

$$\begin{aligned} \mathsf{rel} &= \mathsf{map}(\mathsf{sep}) \circ \mathsf{subtrees} \\ \mathsf{abs} &= \mathsf{map}(\mathsf{pabs}) \circ \mathsf{paths} \end{aligned}$$

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 a downwards function on paths). 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

$$\mathsf{rel} \;=\; \mathsf{map}(\mathsf{sep}) \circ \mathsf{subtrees}$$

where

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

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. That is, define the function contours by

$$\begin{aligned} \mathsf{contours} \;=\; & \mathsf{fork}(\mathsf{left},\mathsf{right}) \circ \mathsf{bdraw} \\ & \text{where} \quad \begin{aligned} \mathsf{left} &= \mathsf{map}(\mathsf{smallest}) \circ \mathsf{levels} \\ \mathsf{right} &= \mathsf{map}(\mathsf{largest}) \circ \mathsf{levels} \end{aligned} \end{aligned}$$

For example, bdraw(five) is pic_1 in Figure 5, and applying the function fork(left, right) to this tree produces the pair of lists $([0, -\frac{1}{2}, 0], [0, \frac{1}{2}, 1])$.

To show that these contours provide the extra information needed to make sep a homomorphism, we need to show that sep can be computed from the contours, and that computing the contours is a homomorphism.

For the first of these,

$$\mathsf{sep} \;=\; \mathsf{spread} \circ \mathsf{contours}$$

where, for some idempotent $\odot$,

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

on pairs of lists, each with head 0.

Now we show that contours is a homomorphism. In the full paper, we calculate that

$$\begin{aligned} \mathsf{contours}(\mathsf{lf}(a)) &= ([0],[0]) \\ \mathsf{contours}(\mathsf{br}(t,a,u)) &= \mathsf{contours}(t) \circledast_a \mathsf{contours}(u) \end{aligned}$$

where

$$\begin{aligned} &(w,x) \circledast_a (y,z) \\ &\quad = ([0] +\!\!+ \mathsf{lzip}(\mathsf{fst})(\mathsf{map}(-s)(w), \mathsf{map}(+s)(y)), \\ &\qquad [0] +\!\!+ \mathsf{lzip}(\mathsf{snd})(\mathsf{map}(-s)(x), \mathsf{map}(+s)(z))) \\ &\qquad\quad \text{where} \quad s = (1 - (x \boxtimes y)) \div 2 \end{aligned} \qquad \text{— (3)}$$

Hence,

$$\mathsf{contours} = \mathsf{bh}(\mathsf{always}(([0],[0])), \circledast)$$

Thus,

$$\mathsf{rel} = \mathsf{map}(\mathsf{spread}) \circ \mathsf{up}(\mathsf{always}(([0],[0])), \circledast) \qquad \text{— (4)}$$

This is now an upwards accumulation, but it is still expensive to compute. The operation $\circledast$ 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 $\circledast$ from (3):

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

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 $\mathsf{msi} = \mathsf{map}(\mathsf{sum}) \circ \mathsf{inits}$, which computes absolute distances from relative ones. Under this refinement, the maps can be performed in constant time, since

$$\begin{aligned} \mathsf{map}(+s)(\mathsf{msi}(x)) &= \mathsf{msi}(\mathsf{mapplus}(s,x)) \\ &\qquad \text{where} \quad \mathsf{mapplus}(b,[a]) = [b+a] \\ &\qquad\qquad \mathsf{mapplus}(b,[a] +\!\!+ x) = [b+a] +\!\!+ x \end{aligned} \qquad \text{— (5)}$$

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

what they show is that if g satisfies

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

then

$$\mathsf{g}(\mathsf{x}) \;=\; \mathsf{size}(\mathsf{x}) - \mathsf{depth}(\mathsf{x})$$

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

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

We note first that computing abs as it stands *is* inefficient. No operator $\oplus$ can satisfy $\mathsf{a} + \mathsf{always}(0)(\mathsf{b}) = \mathsf{always}(0)(\mathsf{a}) \oplus \mathsf{b}$ for all a and b, and so pabs can not 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 the function pabsb, where

$$\mathsf{pabsb} \;=\; \mathsf{fork}(\mathsf{pabs}, \mathsf{bottom})$$

and where bottom returns the bottom element of a path

$$\mathsf{bottom} \;=\; \mathsf{uw}(\mathsf{id}, \mathsf{snd}, \mathsf{snd})$$

is a path homomorphism:

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

Now, $\mathsf{pabs} = \mathsf{fst} \circ \mathsf{pabsb}$, and so

$$\mathsf{abs} \;=\; \mathsf{map}(\mathsf{fst}) \circ \mathsf{down}(\mathsf{f}, \oplus, \otimes) \qquad \text{— (6)}$$

which can be computed in linear time.

5.3 The program

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

$$
\begin{array}{rcl}
\mathsf{bdraw} & = & \mathsf{abs} \circ \mathsf{rel} \\
\\
\mathsf{rel} & = & \mathsf{map}(\mathsf{spread}) \circ \mathsf{up}(\mathsf{always}(([0],[0])), \boxplus) \\
(\mathsf{w},\mathsf{x}) \boxplus_{\mathsf{a}} (\mathsf{y},\mathsf{z}) & = & ([0] +\!\!+ \mathsf{lzipfst}(\mathsf{mapplus}(\text{-}\mathsf{s},\mathsf{w}), \mathsf{mapplus}(\mathsf{s},\mathsf{y})), \\
 & & \quad [0] +\!\!+ \mathsf{lzipsnd}(\mathsf{mapplus}(\text{-}\mathsf{s},\mathsf{x}), \mathsf{mapplus}(\mathsf{s},\mathsf{z}))) \\
 & & \qquad \text{where} \quad \mathsf{s} = (1 - (\mathsf{x} \boxtimes \mathsf{y})) \div 2 \\
\\
\mathsf{mapplus}(\mathsf{b},[\mathsf{a}]) & = & [\mathsf{a}+\mathsf{b}] \\
\mathsf{mapplus}(\mathsf{b},[\mathsf{a}] +\!\!+ \mathsf{x}) & = & [\mathsf{a}+\mathsf{b}] +\!\!+ \mathsf{x} \\
\\
\mathsf{lzipfst}(\mathsf{x},\mathsf{y}) & = & \mathsf{x}, \qquad \text{if } \mathsf{nst}(\mathsf{x},\mathsf{y}) \\
 & = & \mathsf{x} +\!\!+ \mathsf{mapplus}(\mathsf{sum}(\mathsf{v}) - \mathsf{sum}(\mathsf{x}), \mathsf{w}), \quad \text{otherwise} \\
 & & \qquad \text{where} \quad (\mathsf{v},\mathsf{w}) = \mathsf{split}(\mathsf{len}(\mathsf{x}),\mathsf{y}) \\
\mathsf{lzipsnd}(\mathsf{x},\mathsf{y}) & = & \mathsf{lzipfst}(\mathsf{y},\mathsf{x}) \\
\\
\mathsf{nst}(\mathsf{x},[\mathsf{b}]) & = & \mathsf{true} \\
\mathsf{nst}([\mathsf{a}],[\mathsf{b}] +\!\!+ \mathsf{y}) & = & \mathsf{false} \\
\mathsf{nst}([\mathsf{a}] +\!\!+ \mathsf{x},[\mathsf{b}] +\!\!+ \mathsf{y}) & = & \mathsf{nst}(\mathsf{x},\mathsf{y}) \\
\\
\mathsf{split}(1,[\mathsf{a}] +\!\!+ \mathsf{x}) & = & ([\mathsf{a}],\mathsf{x}) \\
\mathsf{split}(\mathsf{n}+1,[\mathsf{a}] +\!\!+ \mathsf{x}) & = & ([\mathsf{a}] +\!\!+ \mathsf{v},\mathsf{w}) \qquad \text{where} \quad (\mathsf{v},\mathsf{w}) = \mathsf{split}(\mathsf{n},\mathsf{x}) \\
\\
\mathsf{spread}([0],[0]) & = & 0 \\
\mathsf{spread}([0] +\!\!+ \mathsf{x},[0] +\!\!+ \mathsf{y}) & = & \text{-}\mathsf{head}(\mathsf{x}) \odot \mathsf{head}(\mathsf{y}) \qquad \text{where} \quad \mathsf{a} \odot \mathsf{a} = \mathsf{a} \\
\\
\mathsf{v} \boxtimes \mathsf{w} & = & \mathsf{lh}(\mathsf{id},\mathsf{min})(\mathsf{szip}(\mathsf{conv}(-))(\mathsf{v},\mathsf{w})) \\
\\
\mathsf{abs} & = & \mathsf{map}(\mathsf{fst}) \circ \mathsf{down}(\mathsf{f},\oplus,\otimes) \\
 & & \qquad \text{where} \qquad \mathsf{f}(\mathsf{a}) = (0,\mathsf{a}) \\
 & & \qquad\qquad (\mathsf{v},\mathsf{w}) \oplus (\mathsf{x},\mathsf{y}) = (\mathsf{v} - \mathsf{w} + \mathsf{x},\mathsf{y}) \\
 & & \qquad\qquad (\mathsf{v},\mathsf{w}) \otimes (\mathsf{x},\mathsf{y}) = (\mathsf{v} + \mathsf{w} + \mathsf{x},\mathsf{y})
\end{array}
$$

Figure 6: The final program

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 sketched the derivation of an efficient algorithm for producing such drawings.

We started with an executable specification (1)—an 'obviously correct' but inefficient program. From this we needed only four inventions to yield a linear algorithm:

(i) 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 (2);
(ii) we made a step towards making the first pass efficient, by turning the function computing relative positions into an upwards accumulation (4), computing not just relative positions but also the outlines of the drawings;
(iii) we made a data refinement on the outline of a drawing (5), allowing us to shift it in constant time; and
(iv) we made the second pass efficient by turning the function computing absolute positions into a downwards accumulation (6), computing not just the absolute positions but also the bottom element of every path.

The derivation showed several things:

(i) the criteria uniquely determine the drawing of a tree;
(ii) the criteria also determine the algorithm—at each stage in the derivation there was effectively only one thing to do (this claim is more defensible given the detailed derivation in the full paper);
(iii) 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;
(iv) 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 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 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 Vaucher's and Reingold and Tilford's algorithms to cope with node labels of varying width and height) also appears not to 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 in (Gibbons, 1991). 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., 1993) to yield an optimal *parallel* algorithm to draw a binary tree of `n` elements in $\log \mathsf{n}$ time on $\mathsf{n}/\log \mathsf{n}$ processors, even when the tree is unbalanced—although this is complicated by having to pass non-constant-size contours around in computing $\boxtimes$.

6.4 Acknowledgements

Thanks are due to Sue Gibbons, for improving 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 (1993). *Efficient parallel algorithms for tree accumulations.* Computer Science Report No. 70, Department of Computer Science, University of Auckland. Accepted for publication in Science of Computer Programming.

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, *Proceedings of the 16th Australian Computer Science Conference*, pages 685–691. Available by anonymous `ftp` as `out/jeremy/papers/quickly.ps.Z` on `cs.auckland.ac.nz`.

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 (1994). *The Third Homomorphism Theorem.* Department of Computer Science, University of Auckland.

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.