Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips

Geraint Jones* and Jeremy Gibbons†

Abstract. This paper is about an application of the mathematics of the zip, reduce (fold) and accumulate (scan) operations on lists. It gives an account of the derivation of a linear-time breadth-first tree traversal algorithm, and of a subtle and efficient breadth-first tree labelling algorithm.

Keywords. Derivation, functional programming, breadth-first, traversal, labelling.

1 Introduction

The algorithms which are developed in this paper relate trees to sequences in a way that respects the breadth-first ordering of the nodes of the tree: nodes nearer the root are earlier in the ordering, and nodes on the same level are ordered from left to right.

We distinguish between finite and infinite sequences, which we call *lists* and *streams* respectively. Lists of elements of type α are modelled as the least solution $\mathsf{list}.\alpha$ of the equation

$$\mathsf{list}.\alpha \;=\; [\,] \;|\; \alpha :: \mathsf{list}.\alpha$$

That is, the empty list $[\,]$ has type $\mathsf{list}.\alpha$, and if x has type α and xs has type $\mathsf{list}.\alpha$ then $\mathsf{x} :: \mathsf{xs}$ has type $\mathsf{list}.\alpha$. We abbreviate the list $\mathsf{x} :: (\mathsf{y} :: (\mathsf{z} :: [\,]))$ by $[\mathsf{x}, \mathsf{y}, \mathsf{z}]$.

Streams of elements of type α are modelled as the *greatest* solution $\mathsf{stream}.\alpha$ of the equation

$$\mathsf{stream}.\alpha \;=\; \alpha :: \mathsf{stream}.\alpha$$

That is, every stream of type $\mathsf{stream}.\alpha$ has a 'head' of type α and a 'tail' of type $\mathsf{stream}.\alpha$. These components are extracted by the destructors hd and tl. (Strictly speaking, we should use different constructors for lists and streams; instead, we will trust to context to disambiguate '$::$'.)

If f takes objects of type α to objects of type β, then $\mathsf{f}*$ (pronounced 'f map') takes objects of type $\mathsf{list}.\alpha$ to objects of type $\mathsf{list}.\beta$, and objects of type $\mathsf{stream}.\alpha$

 Authors' addresses: Oxford University Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, email `geraint@prg.oxford.ac.uk` (*); Dept of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand, email `jeremy@cs.aukuni.ac.nz` (†). Presented at IFIP WG2.1 meeting number 45, as working paper 705 WIN-2.

to objects of type $\mathsf{stream}.\beta$, by applying the function f to every element of the sequence. For example,

$$\mathsf{f}{*}.[\mathsf{x},\mathsf{y},\mathsf{z}] \;=\; [\mathsf{f.x},\mathsf{f.y},\mathsf{f.z}]$$

Note that function application is written with an infix '$.$'; it is right-associative and tightest binding.

If $\oplus$ is an associative operator from $\alpha\times\alpha$ to α, then $\oplus/$ (pronounced 'plussle reduce') takes objects of type $\mathsf{list}.\alpha$ to objects of type α, by 'inserting' $\oplus$ between adjacent elements. For example,

$$\oplus/.[\mathsf{x},\mathsf{y},\mathsf{z}] \;=\; \mathsf{x}\oplus\mathsf{y}\oplus\mathsf{z}$$

The reduction $\oplus/$ of the empty list is the unit of $\oplus$, which is unique if it exists.

We also use *directed reductions* and *accumulations* on lists. The *leftwards reduction* $\oplus\not\leftarrow_{\mathsf{e}}$ of a list is defined by

$$\begin{aligned} \oplus\not\leftarrow_{\mathsf{e}}.[\,] &= \mathsf{e} \\ \oplus\not\leftarrow_{\mathsf{e}}.(\mathsf{x} :: \mathsf{xs}) &= \mathsf{x} \;\oplus\; \oplus\not\leftarrow_{\mathsf{e}}.\mathsf{xs} \end{aligned}$$

For example,

$$\oplus\not\leftarrow_{\mathsf{e}}.[\mathsf{x},\mathsf{y},\mathsf{z}] \;=\; \mathsf{x}\oplus(\mathsf{y}\oplus(\mathsf{z}\oplus\mathsf{e}))$$

Note that the $\oplus$ need not be associative.

The *leftwards accumulation* $\oplus\not\!\!\not\leftarrow_{\mathsf{e}}$ of a list is defined by

$$\oplus\not\leftarrow_{\mathsf{e}}{*}.\mathsf{tails.xs} \;=\; [\oplus\not\leftarrow_{\mathsf{e}}.\mathsf{xs}] \;+\!\!+\; \oplus\not\!\!\not\leftarrow_{\mathsf{e}}.\mathsf{xs}$$

Here, $\mathsf{tails.xs}$ is the list of suffixes of xs in order of decreasing length, from xs itself to the empty list, and $+\!\!+$ is list concatenation. (Later on, we will extend list concatenation to concatenate a list with a stream, in the obvious way.) Note that the accumulation of a list is the same length as the list itself, and that it does not depend on the head of the list if the list is non-empty.

Rightwards reduction satisfies the equations

$$\begin{aligned} \oplus\not\rightarrow_{\mathsf{e}}.[\,] &= \mathsf{e} \\ \oplus\not\rightarrow_{\mathsf{e}}.(\mathsf{x} :: \mathsf{xs}) &= \oplus\not\rightarrow_{\mathsf{e}\oplus\mathsf{x}}.\mathsf{xs} \end{aligned}$$

and rightwards accumulation satisfies

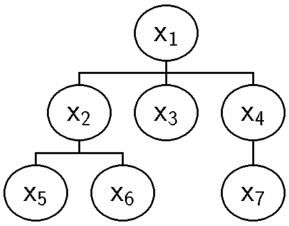
$$\oplus\not\rightarrow_{\mathsf{e}}{*}.\mathsf{inits.xs} \;=\; \oplus\not\!\!\not\rightarrow_{\mathsf{e}}.\mathsf{xs} +\!\!+ [\oplus\not\rightarrow_{\mathsf{e}}.\mathsf{xs}]$$

where inits returns the prefixes of a list.

The trees in this paper are *non-empty homogeneous rose trees* (Meertens, 1988); every tree of type $\mathsf{tree}.\alpha$ consists of a root labelled with an element of type α, and a list of children each of which is itself a tree of type $\mathsf{tree}.\alpha$. Trees are modelled as the least solution $\mathsf{tree}.\alpha$ of the equation

$$\mathsf{tree}.\alpha \;=\; \alpha \prec \mathsf{list.tree}.\alpha$$

For example, the tree



is represented by the expression

$$x_1 \prec [x_2 \prec [x_5 \prec [\,], x_6 \prec [\,]], x_3 \prec [\,], x_4 \prec [x_7 \prec [\,]]]$$

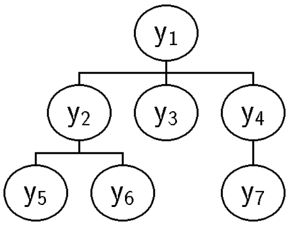
We will call this tree seven, and use it as an example throughout.

The two tree destructors rt and ch are defined by

$$t \;=\; \mathsf{rt}.t \prec \mathsf{ch}.t$$

The two functions we will concern ourselves with in this paper are for breadth-first *traversal* and *labelling* of a tree. As we shall see, they are in a sense each other's inverses, and we will derive an algorithm for relabelling by inverting one for traversal.

Informally, breadth-first traversal takes a tree of elements and returns a list of those elements in breadth-first order; for example, the breadth-first traversal of the tree seven is $[x_1, x_2, x_3, x_4, x_5, x_6, x_7]$. Conversely, breadth-first labelling takes a stream of elements and returns a tree whose breadth-first traversal is a finite prefix of that stream. Since traversal destroys information about the 'shape' of a tree, the labelling function must be provided with a tree—informally, the tree to be relabelled—to determine the shape of its result. For example, labelling seven with the stream $[y_1, y_2, y_3, \ldots]$ yields the tree



2 Breadth-first traversal

Breadth-first traversal is defined in terms of a close relative, *levelorder traversal* (Gibbons, 1991). Levelorder traversal takes a tree of αs to a list of lists of αs—in fact, to a non-empty list of non-empty lists of αs. The first element of the traversal is a singleton list containing just the root of the tree, the second element of the traversal consists of all the nodes of the tree at depth two, and so on. For example, the levelorder traversal of seven is the list of lists

$$[[x_1], [x_2, x_3, x_4], [x_5, x_6, x_7]]$$

Formally, the function levels has type $\mathsf{tree}.\alpha \to \mathsf{list}.\mathsf{list}.\alpha$ and is defined by

$$\mathsf{levels}.(\mathsf{x} \prec \mathsf{ts}) \;=\; [\mathsf{x}] :: \curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ts}$$

Here, $\curlyvee_{\oplus}$ is 'long zip with plussle'; it combines corresponding elements of two lists using $\oplus$, returning a list as long as its longer argument. For example,

$$[\mathsf{x},\mathsf{y},\mathsf{z}] \curlyvee_{\oplus} [\mathsf{u},\mathsf{v}] \;=\; [\mathsf{x} \oplus \mathsf{u}, \mathsf{y} \oplus \mathsf{v}, \mathsf{z}]$$

and in general,

$$\begin{aligned} \mathsf{xs} \curlyvee_{\oplus} [\,] &= \mathsf{xs} \\ [\,] \curlyvee_{\oplus} \mathsf{ys} &= \mathsf{ys} \\ (\mathsf{x} :: \mathsf{xs}) \curlyvee_{\oplus} (\mathsf{y} :: \mathsf{ys}) &= (\mathsf{x} \oplus \mathsf{y}) :: (\mathsf{xs} \curlyvee_{\oplus} \mathsf{ys}) \end{aligned}$$

(Later on, we apply long zips to pairs of sequences, rather than simply pairs of lists, in the obvious way.)

The breadth-first traversal of a tree is obtained simply by concatenating the levels of the tree:

$$\mathsf{bft} \;=\; +\!\!+/ \circ \mathsf{levels}$$

where $\circ$ is backwards functional composition.

Together, these definitions give an executable program in a functional language for computing the breadth-first traversal of a tree:

$$\begin{aligned} \mathsf{bft} \;=\; &+\!\!+/ \circ \mathsf{levels} \\ &\quad \text{where } \mathsf{levels}.(\mathsf{x} \prec \mathsf{ts}) = [\mathsf{x}] :: \curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ts} \end{aligned} \qquad \text{— (1)}$$

This program has time complexity $\mathsf{O}(\mathsf{n} \log \mathsf{n})$ in the size n of its output, which is the size of the tree: the dominant factor is constructing the levels, each of which has length $\mathsf{O}(\mathsf{n})$ and is built by a tree of concatenations.

3 A linear breadth-first traversal

It is well known—a 'folk fact'—that the breadth-first traversal of a tree can be computed in linear time in the length of its output. This linear-time algorithm can be calculated straightforwardly from the characterisation of bft given above.

We observe that

$$\begin{aligned} & \mathsf{bft}.(\mathsf{x} \prec \mathsf{ts}) \\ = & \quad \{ \ \mathsf{bft} \ \} \\ & +\!\!+/.\mathsf{levels}.(\mathsf{x} \prec \mathsf{ts}) \\ = & \quad \{ \ \mathsf{levels} \ \} \\ & +\!\!+/.([\mathsf{x}] :: \curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ts}) \\ = & \quad \{ \ \oplus/.(\mathsf{x} :: \mathsf{xs}) = \mathsf{x} \oplus \oplus/.\mathsf{xs} \ \} \\ & [\mathsf{x}] \;+\!\!+\; +\!\!+/.\curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ts} \\ = & \quad \{ \ [\mathsf{x}] +\!\!+ \mathsf{xs} = \mathsf{x} :: \mathsf{xs} \ \} \\ & \mathsf{x} \;::\; +\!\!+/.\curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ts} \end{aligned}$$

$$= \quad \{ \text{ introduce } \mathsf{bfts} = +\!\!+/ \circ \curlyvee_{+\!\!+}/ \circ \mathsf{levels}* \}$$

$$\mathsf{x} :: \mathsf{bfts.ts}$$

Informally, bfts computes the traversal of a forest. Expanding bfts, we see that

$$\mathsf{bfts}.[\,]$$
$$= \quad \{ \ \mathsf{bfts} \ \}$$
$$+\!\!+/.\curlyvee_{+\!\!+}/.\mathsf{levels}*.[\,]$$
$$= \quad \{ \ \mathsf{f}*.[\,] = [\,] \ \}$$
$$+\!\!+/.\curlyvee_{+\!\!+}/.[\,]$$
$$= \quad \{ \ \curlyvee_{+\!\!+} \text{ has unit } [\,] \ \}$$
$$+\!\!+/.[\,]$$
$$= \quad \{ \ +\!\!+ \text{ has unit } [\,] \ \}$$
$$[\,]$$

so $\mathsf{bfts}.[\,] = [\,]$.

As for non-empty forests, we note first that, for associative $\oplus$,

$$\oplus/.((\mathsf{x} :: \mathsf{xs}) \curlyvee_{\oplus} \mathsf{ys}) \ = \ \mathsf{x} \oplus \oplus/.(\mathsf{ys} \curlyvee_{\oplus} \mathsf{xs})$$

For example,

$$\oplus/.((\mathsf{x}_1 :: [\mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4]) \curlyvee_{\oplus} [\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3])$$
$$= \quad \{ \ \curlyvee, \text{ reduction } \}$$
$$(\mathsf{x}_1 \oplus \mathsf{y}_1) \oplus (\mathsf{x}_2 \oplus \mathsf{y}_2) \oplus (\mathsf{x}_3 \oplus \mathsf{y}_3) \oplus \mathsf{x}_4$$
$$= \quad \{ \text{ associativity } \}$$
$$\mathsf{x}_1 \oplus (\mathsf{y}_1 \oplus \mathsf{x}_2) \oplus (\mathsf{y}_2 \oplus \mathsf{x}_3) \oplus (\mathsf{y}_3 \oplus \mathsf{x}_4)$$
$$= \quad \{ \text{ reduction}, \curlyvee \ \}$$
$$\mathsf{x}_1 \ \oplus \ \oplus/.([\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3] \curlyvee_{\oplus} [\mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4])$$

The general proof, by induction on xs and ys, is straightforward and is omitted.

Returning now to the traversal of a non-empty forest, we have

$$\mathsf{bfts}.((\mathsf{x} \prec \mathsf{ts}) :: \mathsf{us})$$
$$= \quad \{ \ \mathsf{bfts} \ \}$$
$$+\!\!+/.\curlyvee_{+\!\!+}/.\mathsf{levels}*.((\mathsf{x} \prec \mathsf{ts}) :: \mathsf{us})$$
$$= \quad \{ \text{ map, reduce } \}$$
$$+\!\!+/.(\mathsf{levels}.(\mathsf{x} \prec \mathsf{ts}) \ \curlyvee_{+\!\!+} \ \curlyvee_{+\!\!+}/.\mathsf{levels}*.\mathsf{us})$$
$$= \quad \{ \ \mathsf{levels} \ \}$$
$$+\!\!+/.(([\mathsf{x}] :: \curlyvee_{+\!\!+}/.\mathsf{levels}*.\mathsf{ts}) \ \curlyvee_{+\!\!+} \ \curlyvee_{+\!\!+}/.\mathsf{levels}*.\mathsf{us})$$

$$
\begin{array}{ll}
= & \{\ \text{lemma}\ \} \\
 & [\mathsf{x}] \mathbin{+\!\!+} \mathbin{+\!\!+}/.(\curlyvee_{+\!\!+}/.\mathsf{levels}*.\mathsf{us} \curlyvee_{+\!\!+} \curlyvee_{+\!\!+}/.\mathsf{levels}*.\mathsf{ts}) \\
= & \{\ \text{promotion}\ \} \\
 & [\mathsf{x}] \mathbin{+\!\!+} \mathbin{+\!\!+}/.\curlyvee_{+\!\!+}/.\mathsf{levels}*.(\mathsf{us} \mathbin{+\!\!+} \mathsf{ts}) \\
= & \{\ \mathsf{bfts}\ \} \\
 & [\mathsf{x}] \mathbin{+\!\!+} \mathsf{bfts}.(\mathsf{us} \mathbin{+\!\!+} \mathsf{ts}) \\
= & \{\ ::\ \} \\
 & \mathsf{x} :: \mathsf{bfts}.(\mathsf{us} \mathbin{+\!\!+} \mathsf{ts})
\end{array}
$$

Noting that

$$\mathsf{bft.t} \;=\; \mathsf{bfts}.[\mathsf{t}]$$

yields the program

$$
\mathsf{bft.y} \;=\; \mathsf{bfts}.[\mathsf{t}] \quad \text{where} \quad
\begin{array}{rcl}
\mathsf{bfts}.[\,] & = & [\,] \\
\mathsf{bfts}.((\mathsf{x} \prec \mathsf{ts}) :: \mathsf{us}) & = & \mathsf{x} :: \mathsf{bfts}.(\mathsf{us} \mathbin{+\!\!+} \mathsf{ts})
\end{array}
$$

This is essentially how breadth-first traversal is usually implemented in an imperative programming language—using a queue of trees 'yet to be traversed'. Executed naively in a typical functional language, however, this program takes quadratic time, because appending ts to us takes time proportional to the length of us, which grows linearly in the size of the tree.

Fortunately, there is a standard technique for eliminating queues in functional languages, provided that their use is only 'single threaded'. This is the same technique that turns the naive quadratic program for reversing a list into the linear 'fast reverse'. It involves representing a queue xs as a pair of lists $(\mathsf{ys}, \mathsf{zs})$ such that $\mathsf{xs} = \mathsf{ys} \mathbin{+\!\!+} \mathsf{rev.zs}$. We define

$$\mathsf{fastbfts}.(\mathsf{us}, \mathsf{vs}) \;=\; \mathsf{bfts}.(\mathsf{us} \mathbin{+\!\!+} \mathsf{rev.vs})$$

(so that $\mathsf{bfts.ts} = \mathsf{fastbfts}.(\mathsf{ts}, [\,])$) and calculate

$$
\begin{array}{ll}
 & \mathsf{fastbfts}.([\,], [\,]) \\
= & \{\ \mathsf{fastbfts}\ \} \\
 & \mathsf{bfts}.[\,] \\
= & \{\ \mathsf{bfts}\ \} \\
 & [\,]
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathsf{fastbfts}.([\,], \mathsf{v} :: \mathsf{vs}) \\
= & \{\ \mathsf{fastbfts}\ \} \\
 & \mathsf{bfts.rev}.(\mathsf{v} :: \mathsf{vs})
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \ \mathsf{fastbfts} \ \} \\
& \mathsf{fastbfts}.(\mathsf{rev}.(\mathsf{v} :: \mathsf{vs}), [\,])
\end{array}
$$

and finally,

$$
\begin{array}{ll}
& \mathsf{fastbfts}.((\mathsf{x} \prec \mathsf{ts}) :: \mathsf{us}, \mathsf{vs}) \\
= & \{ \ \mathsf{fastbfts} \ \} \\
& \mathsf{bfts}.((\mathsf{x} \prec \mathsf{ts}) :: \mathsf{us} \mathbin{+\!\!+} \mathsf{rev}.\mathsf{vs}) \\
= & \{ \ \mathsf{bfts} \ \} \\
& \mathsf{x} :: \mathsf{bfts}.(\mathsf{us} \mathbin{+\!\!+} \mathsf{rev}.\mathsf{vs} \mathbin{+\!\!+} \mathsf{ts}) \\
= & \{ \ \text{reverse} \ \} \\
& \mathsf{x} :: \mathsf{bfts}.(\mathsf{us} \mathbin{+\!\!+} \mathsf{rev}.(\mathsf{rev}.\mathsf{ts} \mathbin{+\!\!+} \mathsf{vs})) \\
= & \{ \ \mathsf{fastbfts} \ \} \\
& \mathsf{x} :: \mathsf{fastbfts}(\mathsf{us}, \mathsf{rev}.\mathsf{ts} \mathbin{+\!\!+} \mathsf{vs})
\end{array}
$$

This program takes linear amortized time in the length of its output, assuming rev is computed in linear time—the time is proportional to the time spent on rev, and every list of children gets reversed once when it is placed on the second list and once more when it migrates to the first list; the sum of the lengths of all lists of children of a tree is one less than the size of the tree. More formally, if we define

$$
\begin{array}{rcl}
\mathsf{f}.([\,], [\,]) & = & 0 \\
\mathsf{f}.([\,], \mathsf{v} :: \mathsf{vs}) & = & \#.(\mathsf{v} :: \mathsf{vs}) + \mathsf{f}.(\mathsf{rev}.(\mathsf{v} :: \mathsf{vs}), [\,]) \\
\mathsf{f}.((\mathsf{x} \prec \mathsf{ts}) :: \mathsf{us}, \mathsf{vs}) & = & \#.\mathsf{ts} + \mathsf{f}.(\mathsf{us}, \mathsf{rev}.\mathsf{ts} \mathbin{+\!\!+} \mathsf{vs})
\end{array}
$$

(where $\#$ returns the length of a list), so that $\mathsf{f}.(\mathsf{us}, \mathsf{vs})$ counts the 'amount of time' spent on rev in computing $\mathsf{fastbfts}.(\mathsf{us}, \mathsf{vs})$, then we can show by induction that

$$
\mathsf{f}.(\mathsf{ts}, [\,]) \ = \ 2 \times +/.\mathsf{s}{*}.\mathsf{ts} \qquad \text{where } \mathsf{s}.\mathsf{t} = \mathsf{size}.\mathsf{t} - 1
$$

4 A second linear breadth-first traversal

There is an entirely different linear program for breadth-first traversal, again arising by calculation from the characterisation (1). This time the linearisation comes from representing the levels in the levelorder traversal—non-empty lists—in such a way that they can be concatenated in constant time.

One such representation uses the least solution $\mathsf{join}.\alpha$ of the equation

$$
\mathsf{join}.\alpha \ = \ \Box.\alpha \ \mid \ \mathsf{join}.\alpha \mathbin{+\!\!+\!\!+} \mathsf{join}.\alpha
$$

—that is, as non-empty leaf-labelled binary trees in which every parent has exactly two children. The abstraction function is $\mathsf{fringe} : \mathsf{join}.\alpha \to \mathsf{list}.\alpha$, given by

$$
\begin{array}{rcl}
\mathsf{fringe}.\Box.\mathsf{x} & = & [\mathsf{x}] \\
\mathsf{fringe}.(\mathsf{j} \mathbin{+\!\!+\!\!+} \mathsf{k}) & = & \mathsf{fringe}.\mathsf{j} \mathbin{+\!\!+} \mathsf{fringe}.\mathsf{k}
\end{array}
$$

Under this abstraction, levels is implemented by a function $\mathsf{jlevels}$, of type $\mathsf{tree}.\alpha \to \mathsf{list}.\mathsf{join}.\alpha$, such that

$$\mathsf{fringe}* \circ \mathsf{jlevels} \;=\; \mathsf{levels}$$

It is easy to see that the definition

$$\mathsf{jlevels}.(\mathsf{x} \prec \mathsf{ts}) \;=\; \Box.\mathsf{x} :: \curlyvee_{+\!\!+\!\!+}/.\mathsf{jlevels}*.\mathsf{ts}$$

satisfies this requirement. (There are other definitions of jlevels that satisfy, because fringe is not injective, but this definition is the 'simplest'.)

Computing jlevels.t involves linearly many applications of $+\!\!+\!\!+$—in fact, $\mathsf{size.t} - \mathsf{depth.t}$ applications, since there is one application for each pair of adjacent elements in the levelorder traversal—and so takes linear time.

In terms of join lists, bft is given by

$$\mathsf{bft} \;=\; +\!\!+/ \circ \mathsf{fringe}* \circ \mathsf{jlevels}$$

and so it remains only to compute $+\!\!+/ \circ \mathsf{fringe}*$ in time proportional to the length of its result. This can be done by introducing a leftwards reduction: the Specialisation Theorem (Bird, 1987) states that, for associative $\oplus$,

$$\oplus/.\mathsf{f}*.\mathsf{xs} \;\oplus\; \mathsf{e} \;=\; \otimes\not\leftarrow_{\mathsf{e}}.\mathsf{xs} \qquad \text{where } \mathsf{x} \otimes \mathsf{y} = \mathsf{f.x} \oplus \mathsf{y} \qquad \text{— (2)}$$

In particular, if $\oplus$ has unit e then

$$\oplus/.\mathsf{f}*.\mathsf{xs} \;=\; \otimes\not\leftarrow_{\mathsf{e}}.\mathsf{xs} \qquad \text{— (3)}$$

Thus,

$$+\!\!+/ \circ \mathsf{fringe}* \;=\; \otimes\not\leftarrow_{[]} \qquad \text{where } \mathsf{j} \otimes \mathsf{xs} = \mathsf{fringe.j} +\!\!+ \mathsf{xs}$$

We can synthesize a definition of $\otimes$ taking time proportional to the size of its left argument:

$$
\begin{array}{cl}
 & \Box.\mathsf{x} \otimes \mathsf{xs} \\
= & \quad \{\ \otimes\ \} \\
 & \mathsf{fringe}.\Box.\mathsf{x} +\!\!+ \mathsf{xs} \\
= & \quad \{\ \mathsf{fringe}\ \} \\
 & \mathsf{x} :: \mathsf{xs}
\end{array}
$$

and

$$
\begin{array}{cl}
 & (\mathsf{j} +\!\!+\!\!+ \mathsf{k}) \otimes \mathsf{xs} \\
= & \quad \{\ \otimes\ \} \\
 & \mathsf{fringe}.(\mathsf{j} +\!\!+\!\!+ \mathsf{k}) +\!\!+ \mathsf{xs} \\
= & \quad \{\ \mathsf{fringe}\ \} \\
 & \mathsf{fringe.j} +\!\!+ \mathsf{fringe.k} +\!\!+ \mathsf{xs} \\
= & \quad \{\ \otimes\ \} \\
 & \mathsf{j} \otimes (\mathsf{k} \otimes \mathsf{xs})
\end{array}
$$

hence

$$\mathsf{bft} \;=\; \otimes\not\leftarrow_{[]} \circ \mathsf{jlevels}$$

where

$$\mathsf{jlevels}.(\mathsf{x} \prec \mathsf{ts}) \;=\; \square.\mathsf{x} :: \curlyvee_{+\!\!+\!\!+}/.\mathsf{jlevels}{*}.\mathsf{ts}$$

and

$$\begin{aligned} \square.\mathsf{x} \otimes \mathsf{xs} &= \mathsf{x} :: \mathsf{xs} \\ (\mathsf{j} +\!\!+\!\!+ \mathsf{k}) \otimes \mathsf{xs} &= \mathsf{j} \otimes (\mathsf{k} \otimes \mathsf{xs}) \end{aligned}$$

This program takes linear time.

5 Breadth-first labelling

The breadth-first labelling $(\mathsf{bfl}.\mathsf{t}).\mathsf{xs}$ of a tree t with a stream xs is determined by two facts. The first is that the result is a tree of the same shape as t:

$$\mathsf{shape}.(\mathsf{bfl}.\mathsf{t}).\mathsf{xs} \;=\; \mathsf{shape}.\mathsf{t}$$

Here, shape takes a tree of αs to a tree of $\mathbb{1}$s, $\mathbb{1}$ being the unit type:

$$\mathsf{shape} \;=\; \mathsf{unit}{*} \qquad \text{where } \mathsf{unit} : \alpha \to \mathbb{1}$$

The second fact is that the breadth-first traversal of $(\mathsf{bfl}.\mathsf{t}).\mathsf{xs}$ is a finite prefix of xs. Since $(\mathsf{bfl}.\mathsf{t}).\mathsf{xs}$ and t have the same shape, their traversals have the same length, and so the breadth-first traversal of $(\mathsf{bfl}.\mathsf{t}).\mathsf{xs}$ consists of the first $\#.\mathsf{bft}.\mathsf{t}$ elements of xs:

$$\mathsf{bft}.(\mathsf{bfl}.\mathsf{t}).\mathsf{xs} \;=\; \#.\mathsf{bft}.\mathsf{t} \rightharpoondown \mathsf{xs}$$

Here, $\rightharpoondown$ (pronounced 'take') takes a number n and a stream xs and returns the list consisting of the first n elements of xs:

$$\begin{aligned} 0 \rightharpoondown \mathsf{xs} &= [\,] \\ (\mathsf{n}+1) \rightharpoondown (\mathsf{x} :: \mathsf{xs}) &= \mathsf{x} :: (\mathsf{n} \rightharpoondown \mathsf{xs}) \end{aligned}$$

The derivation of bfl proceeds by inverting the specification of bft. Recall that

$$\begin{aligned} & \mathsf{levels}.(\mathsf{x} \prec \mathsf{ts}) \\ = \quad & \{\ \mathsf{levels}\ \} \\ & [\mathsf{a}] :: \curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ts} \\ = \quad & \{\ \text{specialisation (3): let } \mathsf{t} \oplus \mathsf{xss} = \mathsf{levels}.\mathsf{t} \curlyvee_{+\!\!+} \mathsf{xss}\ \} \\ & [\mathsf{a}] :: \oplus\not\leftarrow_{[\,]}.\mathsf{ts} \end{aligned}$$

Informally, $\mathsf{t} \oplus \mathsf{xss}$ is the sequence of sequences obtained by placing the levelorder traversal of t 'beside' the sequence of sequences xss. The operator $\oplus$ is invertible, in the sense that t and $\mathsf{t} \oplus \mathsf{xss}$ determine xss; moreover, $\mathsf{t} \oplus \mathsf{xss}$ and $\mathsf{shape}.\mathsf{t}$ determine t. That is, the equation

$$\mathsf{xss} \;=\; (\mathsf{xss} \odot \mathsf{t}) \oplus (\mathsf{xss} \ominus \mathsf{t}) \qquad \text{— (4)}$$

—together with the requirement that $\mathsf{xss} \odot \mathsf{t}$ be the same shape as t—determines $\mathsf{xss} \odot \mathsf{t}$ and $\mathsf{xss} \ominus \mathsf{t}$.

The requirement that $\mathsf{xss} \odot \mathsf{t}$ be the same shape as t suggests that there is some connection between $\odot$ and relabelling. For which xss, if any, does the equation

$$(\mathsf{bfl}.\mathsf{t}).\mathsf{xs} \;=\; \mathsf{xss} \odot \mathsf{t} \qquad \text{— (5)}$$

hold?

In answering this question, we will encounter streams as solutions to the equation

$$\mathsf{ys} \;=\; \mathsf{xs} \curlyvee_{\oplus} \mathsf{tl.ys} \qquad \text{— (6)}$$

in ys. The solutions of such equations are leftwards accumulations—the stream solutions are exactly streams ys of the form

$$\mathsf{ys} \;=\; (\mathsf{xs} \curlyvee_{\oplus} \oplus\!\not\leftarrow_{\mathsf{e}}.\mathsf{xs}) +\!\!\!+\, \mathsf{es} \qquad \text{where } \mathsf{es} = \mathsf{e} :: \mathsf{es}$$

for various values of e. Further, the equation

$$\mathsf{y} \;=\; \mathsf{hd.ys} \qquad \text{where } \mathsf{ys} = \mathsf{xs} \curlyvee_{\oplus} \mathsf{tl.ys}$$

has solutions in y

$$\mathsf{y} \;=\; \oplus\!\not\leftarrow_{\mathsf{e}}.\mathsf{xs}$$

for various e. This can be seen by writing out (6) as a series of equations on the elements of ys; if $\mathsf{xs} = [\mathsf{x}_1, \ldots, \mathsf{x}_n]$ and $\mathsf{ys} = [\mathsf{y}_1, \mathsf{y}_2, \ldots]$, then (6) reduces to

$$\begin{array}{rcl}
\mathsf{y}_1 & = & \mathsf{x}_1 \oplus \mathsf{y}_2 \\
\mathsf{y}_2 & = & \mathsf{x}_2 \oplus \mathsf{y}_3 \\
 & \vdots & \\
\mathsf{y}_n & = & \mathsf{x}_n \oplus \mathsf{y}_{n+1} \\
\mathsf{y}_{n+1} & = & \mathsf{y}_{n+2} \\
\mathsf{y}_{n+2} & = & \mathsf{y}_{n+3} \\
 & \vdots &
\end{array}$$

which can be solved from last to first.

Returning to investigate the consequences of (5), we note that one of the requirements on bfl is that $\mathsf{bft.(bfl.t).xs}$ be a prefix of xs, that is, there is a stream ys such that

$$\begin{array}{ll}
 & \mathsf{xs} \\
= & \quad \{ \text{ requirement } \} \\
 & \mathsf{bft.(bfl.t).xs} +\!\!\!+\, \mathsf{ys} \\
= & \quad \{ \text{ (5) } \} \\
 & \mathsf{bft.(xss} \odot \mathsf{t)} +\!\!\!+\, \mathsf{ys} \\
= & \quad \{ \text{ } \mathsf{bft} \text{ } \} \\
 & +\!\!\!+/.\mathsf{levels.(xss} \odot \mathsf{t)} +\!\!\!+\, \mathsf{ys} \\
= & \quad \{ \text{ specialisation (2) } \} \\
 & +\!\!\!+\!\not\leftarrow_{\mathsf{ys}}.\mathsf{levels.(xss} \odot \mathsf{t)} \\
= & \quad \{ \text{ observation above } \} \\
 & \mathsf{hd.yss} \quad \text{where } \mathsf{yss} = \mathsf{levels.(xss} \odot \mathsf{t)} \curlyvee_{+\!\!\!+} \mathsf{tl.yss} \\
= & \quad \{ \obar \} \\
 & \mathsf{hd.yss} \quad \text{where } \mathsf{yss} = (\mathsf{xss} \odot \mathsf{t}) \obar \mathsf{tl.yss}
\end{array}$$

Here, the xs is known and the equation is to be solved for xss and yss. Equation (4) suggests a solution: if $\mathsf{tl.yss} = \mathsf{xss} \ominus \mathsf{t}$ then $\mathsf{yss} = \mathsf{xss}$ and $\mathsf{hd.yss} = \mathsf{xs}$. This determines xss in terms of xs and t:

$$\mathsf{xss} \;=\; \mathsf{xs} :: (\mathsf{xss} \ominus \mathsf{t})$$

and outlines a program for bfl:

$$(\mathsf{bfl.t}).\mathsf{xs} \;=\; \mathsf{xss} \odot \mathsf{t} \qquad \text{where } \mathsf{xss} = \mathsf{xs} :: (\mathsf{xss} \ominus \mathsf{t}) \qquad \text{— (7)}$$

It remains only to synthesize definitions of $\odot$ and $\ominus$.

We have

$$\begin{array}{ll}
& \mathsf{hd.xss} \\
= & \quad \{\ (4)\ \} \\
& \mathsf{hd}.((\mathsf{xss} \odot \mathsf{t}) \obar (\mathsf{xss} \ominus \mathsf{t})) \\
= & \quad \{\ \obar\ \} \\
& \mathsf{hd}.(\mathsf{levels}.(\mathsf{xss} \odot \mathsf{t}) \curlyvee_{+\!\!+} (\mathsf{xss} \ominus \mathsf{t})) \\
= & \quad \{\ \mathsf{hd} \text{ and } \curlyvee\ \} \\
& \mathsf{hd.levels}.(\mathsf{xss} \odot \mathsf{t}) \mathbin{+\!\!+} \mathsf{hd}.(\mathsf{xss} \ominus \mathsf{t}) \\
= & \quad \{\ \mathsf{hd.levels.t} = [\mathsf{rt.t}]\ \} \\
& [\mathsf{rt}.(\mathsf{xss} \odot \mathsf{t})] \mathbin{+\!\!+} \mathsf{hd}.(\mathsf{xss} \ominus \mathsf{t})
\end{array}$$

whence we deduce that

$$\begin{array}{rcl}
\mathsf{rt}.(\mathsf{xss} \odot \mathsf{t}) & = & \mathsf{hd.hd.xss} \\
\mathsf{hd}.(\mathsf{xss} \ominus \mathsf{t}) & = & \mathsf{tl.hd.xss}
\end{array}$$

Similarly, we have

$$\begin{array}{ll}
& \mathsf{tl.xss} \\
= & \quad \{\ (4)\ \} \\
& \mathsf{tl}.((\mathsf{xss} \odot \mathsf{t}) \obar (\mathsf{xss} \ominus \mathsf{t})) \\
= & \quad \{\ \obar\ \} \\
& \mathsf{tl}.(\mathsf{levels}.(\mathsf{xss} \odot \mathsf{t}) \curlyvee_{+\!\!+} (\mathsf{xss} \ominus \mathsf{t})) \\
= & \quad \{\ \mathsf{tl} \text{ and } \curlyvee \text{ (and } \mathsf{levels} \text{ yields non-empty lists)}\ \} \\
& \mathsf{tl.levels}.(\mathsf{xss} \odot \mathsf{t}) \curlyvee_{+\!\!+} \mathsf{tl}.(\mathsf{xss} \ominus \mathsf{t}) \\
= & \quad \{\ \mathsf{tl.levels.u} = \curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ch.u}\ \} \\
& \curlyvee_{+\!\!+}/.\mathsf{levels}{*}.\mathsf{ch}.(\mathsf{xss} \odot \mathsf{t}) \curlyvee_{+\!\!+} \mathsf{tl}.(\mathsf{xss} \ominus \mathsf{t}) \\
= & \quad \{\ \text{specialisation (2)}\ \} \\
& \obar \not\leftarrow_{\mathsf{tl}.(\mathsf{xss} \ominus \mathsf{t})} .\mathsf{ch}.(\mathsf{xss} \odot \mathsf{t})
\end{array}$$

in which

$$\mathsf{shape}{*}.\mathsf{ch}.(\mathsf{xss} \odot \mathsf{t}) \;=\; \mathsf{shape}{*}.\mathsf{ch.t}$$

This is a pair of equations of the form

$$\mathsf{u} \;=\; \obar\!\nleftarrow_{\mathsf{w}}.\mathsf{vs} \qquad \mathsf{shape}{*}.\mathsf{vs} \;=\; \mathsf{shape}{*}.\mathsf{ts}$$

in which u and ts are known, and which we have to solve for w and vs. This can be seen again as a series of equations

$$\begin{array}{rclclcrcl}
\mathsf{u} & = & \mathsf{u}_1 & = & \mathsf{v}_1 \obar \mathsf{u}_2 & & \mathsf{shape.v}_1 & = & \mathsf{shape.t}_1 \\
 & & \mathsf{u}_2 & = & \mathsf{v}_2 \obar \mathsf{u}_3 & & \mathsf{shape.v}_2 & = & \mathsf{shape.t}_2 \\
 & & & \vdots & & & & \vdots & \\
 & & \mathsf{u}_n & = & \mathsf{v}_n \obar \mathsf{w} & & \mathsf{shape.v}_n & = & \mathsf{shape.t}_n
\end{array}$$

where u and each t_i are known, and we must find w and each v_i. By equation (4), the pair of equations

$$\mathsf{u} \;=\; \mathsf{v} \obar \mathsf{w} \qquad \mathsf{shape.v} \;=\; \mathsf{shape.t}$$

is solved by

$$\mathsf{v} \;=\; \mathsf{u} \odot \mathsf{t} \qquad \mathsf{w} \;=\; \mathsf{u} \ominus \mathsf{t}$$

Thus we have

$$\begin{array}{rclcrcl}
\mathsf{v}_1 & = & \mathsf{u} \odot \mathsf{t}_1 & \quad & \mathsf{u}_2 & = & \mathsf{u} \ominus \mathsf{t}_1 \\
\mathsf{v}_2 & = & \mathsf{u}_2 \odot \mathsf{t}_2 & & \mathsf{u}_3 & = & \mathsf{u}_2 \ominus \mathsf{t}_2 \\
 & = & (\mathsf{u} \ominus \mathsf{t}_1) \odot \mathsf{t}_2 & & & = & (\mathsf{u} \ominus \mathsf{t}_1) \ominus \mathsf{t}_2 \\
 & \vdots & & & & \vdots & \\
\mathsf{v}_n & = & ((\mathsf{u} \ominus \mathsf{t}_1) \ominus \cdots \ominus \mathsf{t}_{n-1}) \odot \mathsf{t}_n & & \mathsf{w} & = & ((\mathsf{u} \ominus \mathsf{t}_1) \ominus \cdots \ominus \mathsf{t}_{n-1}) \ominus \mathsf{t}_n
\end{array}$$

That is,

$$\mathsf{vs} \;=\; \ominus\!\nrightarrow\!\!\!\!/_{\mathsf{u}}.\mathsf{ts} \;\Upsilon_{\odot}\; \mathsf{ts} \qquad \mathsf{w} \;=\; \ominus\!\nrightarrow_{\mathsf{u}}.\mathsf{ts}$$

Thus, the pair of equations

$$\begin{array}{rcl}
\mathsf{tl.xss} & = & \obar\!\nleftarrow_{\mathsf{tl.(xss\ominus t)}}.\mathsf{ch.(xss} \odot \mathsf{t)} \\
\mathsf{shape}{*}.\mathsf{ch.(xss} \odot \mathsf{t)} & = & \mathsf{shape}{*}.\mathsf{ch.t}
\end{array}$$

has solutions

$$\begin{array}{rcl}
\mathsf{ch.(xss} \odot \mathsf{t)} & = & \ominus\!\nrightarrow\!\!\!\!/_{\mathsf{tl.xss}}.\mathsf{ch.t} \;\Upsilon_{\odot}\; \mathsf{ch.t} \\
\mathsf{tl.(xss} \ominus \mathsf{t)} & = & \ominus\!\nrightarrow_{\mathsf{tl.xss}}.\mathsf{ch.t}
\end{array}$$

Assembling these discoveries, we get

$$\begin{array}{rcl}
\mathsf{xss} \odot \mathsf{t} & = & \mathsf{rt.(xss} \odot \mathsf{t)} \prec \mathsf{ch.(xss} \odot \mathsf{t)} \\
 & = & \mathsf{hd.hd.xss} \prec (\ominus\!\nrightarrow\!\!\!\!/_{\mathsf{tl.xss}}.\mathsf{ch.t} \;\Upsilon_{\odot}\; \mathsf{ch.t}) \\
\mathsf{xss} \ominus \mathsf{t} & = & \mathsf{hd.(xss} \ominus \mathsf{t)} :: \mathsf{tl.(xss} \ominus \mathsf{t)} \\
 & = & \mathsf{tl.hd.xss} :: \ominus\!\nrightarrow_{\mathsf{tl.xss}}.\mathsf{ch.t}
\end{array}$$

There is some inefficiency involved in computing the accumulation separately from the reduction that would be the next term in a longer accumulation. This can be avoided by tupling the two computations and defining a function something like

$$\mathsf{g}_{\oplus,\otimes}.(\mathsf{e},\mathsf{xs}) \;=\; (\otimes\!\nrightarrow\!\!\!\!/_{\mathsf{e}}.\mathsf{xs} \;\Upsilon_{\oplus}\; \mathsf{xs},\; \otimes\!\nrightarrow_{\mathsf{e}}.\mathsf{xs})$$

As it happens, this g has a familiar form: it arises from work on VLSI layout (Jones and Sheeran, 1990), where it is called row and has to do with a row of tiles in a two-dimensional rectangularly connected grid:

$$
\begin{aligned}
(\mathsf{row}.f).(e, ys) \;&=\; (\otimes\not\#_e.xs \curlyvee_{\oplus} xs, \otimes\not\rightarrow_e.xs) \\
&\qquad\qquad\qquad \text{where } (a \oplus x, a \otimes x) = f.(a, x)
\end{aligned}
$$

It might help to think of row in terms of an automaton whose state transition function f takes a state a and an input x and produces an output $a \oplus x$ and a new state $a \otimes x$; then $(\mathsf{row}.f).(a, xs)$ produces a pair (ys, c) consisting of a list ys of outputs and a final state c.

An efficient characterisation of row can be synthesized in the usual way:

$$
\begin{aligned}
& (\mathsf{row}.f).(a, [\,]) \\
= \quad & \quad \{\ \mathsf{row}\ \} \\
& (\otimes\not\#_a.[\,] \curlyvee_{\oplus} [\,], \otimes\not\rightarrow_a.[\,]) \\
= \quad & \quad \{\ \text{reduction and accumulation}\ \} \\
& ([\,], a)
\end{aligned}
$$

and

$$
\begin{aligned}
& (\mathsf{row}.f).(a, x :: xs) \\
= \quad & \quad \{\ \mathsf{row}\ \} \\
& (\otimes\not\#_a.(x :: xs) \curlyvee_{\oplus} (x :: xs), \otimes\not\rightarrow_a.(x :: xs)) \\
= \quad & \quad \{\ \text{reduction and accumulation}\ \} \\
& ((a :: \otimes\not\#_{a\otimes x}.xs) \curlyvee_{\oplus} (x :: xs), \otimes\not\rightarrow_{a\otimes x}.xs) \\
= \quad & \quad \{\ \curlyvee\ \} \\
& ((a \oplus x) :: (\otimes\not\#_{a\otimes x}.xs \curlyvee_{\oplus} xs), \otimes\not\rightarrow_{a\otimes x}.xs) \\
= \quad & \quad \{\ \mathsf{row}\ \} \\
& ((a \oplus x) :: ys, c) \quad \text{where } (ys, c) = (\mathsf{row}.f).(a \otimes x, xs) \\
= \quad & \quad \{\ \oplus \text{ and } \otimes\text{, from definition of } \mathsf{row}\ \} \\
& (y :: ys, c) \quad \text{where} \quad (y, b) \;=\; f.(a, x) \\
& \qquad\qquad\qquad\quad\;\; (ys, c) \;=\; (\mathsf{row}.f).(b, xs)
\end{aligned}
$$

If we define f by

$$
f.(xss, t) \;=\; (xss \odot t, xss \ominus t)
$$

then

$$
(\ominus\not\#_{tl.xss}.ch.t \curlyvee_{\odot} ch.t, \ominus\not\rightarrow_{tl.xss}.ch.t) \;=\; (\mathsf{row}.f).(tl.xss, ch.t)
$$

and so

$$
\begin{aligned}
& f.(xss, t) \\
= \quad & \quad \{\ f\ \} \\
& (xss \odot t, xss \ominus t) \\
= \quad & \quad \{\ \odot, \ominus\ \} \\
& (hd.hd.xss \prec (\ominus\not\#_{tl.xss}.ch.t \curlyvee_{\odot} ch.t), tl.hd.xss :: \ominus\not\rightarrow_{tl.xss}.ch.t)
\end{aligned}
$$

$$= \quad \{ \ \mathsf{f} \ \}$$

$$(\mathsf{hd.hd.xss} \prec \mathsf{ts}, \mathsf{tl.hd.xss} :: \mathsf{yss}) \quad \text{where } (\mathsf{ts}, \mathsf{yss}) = (\mathsf{row.f}).(\mathsf{tl.xss}, \mathsf{ch.t})$$

Moreover,

$$(\mathsf{bfl.t}).\mathsf{xss}$$

$$= \quad \{ \ (7) \ \}$$

$$\mathsf{xss} \odot \mathsf{t} \quad \text{where } \mathsf{xss} = \mathsf{xs} :: (\mathsf{xss} \ominus \mathsf{t})$$

$$= \quad \{ \ \mathsf{f} \ \}$$

$$\begin{array}{lrcl} \mathsf{yss} & \text{where } (\mathsf{yss}, \mathsf{zss}) & = & \mathsf{f}.(\mathsf{xss}, \mathsf{t}) \\ & \mathsf{xss} & = & \mathsf{xs} :: \mathsf{zss} \end{array}$$

$$= \quad \{ \ \text{substituting } \mathsf{xss} \ \}$$

$$\mathsf{yss} \quad \text{where } (\mathsf{yss}, \mathsf{zss}) = \mathsf{f}.(\mathsf{xs} :: \mathsf{zss}, \mathsf{t})$$

which completes the derivation—the program

$$(\mathsf{bfl.t}).\mathsf{xss} \ = \ \mathsf{yss} \qquad \text{where } (\mathsf{yss}, \mathsf{zss}) = \mathsf{f}.(\mathsf{xs} :: \mathsf{zss}, \mathsf{t})$$

where

$$\begin{array}{l} \mathsf{f}.((\mathsf{x} :: \mathsf{xs}) :: \mathsf{xss}, \mathsf{w} \prec \mathsf{ts}) \\ \qquad = \ (\mathsf{x} \prec \mathsf{us}, \mathsf{xs} :: \mathsf{yss}) \qquad \text{where } (\mathsf{us}, \mathsf{yss}) = (\mathsf{row.f}).(\mathsf{xss}, \mathsf{ts}) \end{array}$$

where

$$\begin{array}{rcllrcl} (\mathsf{row.f}).(\mathsf{a}, []) & = & ([], \mathsf{a}) & & & & \\ (\mathsf{row.f}).(\mathsf{a}, \mathsf{x} :: \mathsf{xs}) & = & (\mathsf{y} :: \mathsf{ys}, \mathsf{c}) & \text{where} & (\mathsf{y}, \mathsf{b}) & = & \mathsf{f}.(\mathsf{a}, \mathsf{x}) \\ & & & & (\mathsf{ys}, \mathsf{c}) & = & (\mathsf{row.f}).(\mathsf{b}, \mathsf{xs}) \end{array}$$

implements bfl with a cost which is linear in the size of the tree.

6 Acknowledgements

The problem of breadth-first labelling was originally posed to us by Joe Fasel. He had been selling functional programming to sceptical imperative programmers (perhaps as a technique for writing parallelisable code). One of his successes was showing someone how to do breadth-first traversal. His colleague was so taken with the elegance of the solution that he immediately came back with the labelling problem, expecting that it would be just as easy. Of course, it is once you have seen how to do it, but it seems difficult to explain how one might go about writing the program.

This calculation was hammered out with the assistance of the squiggolists at the Programming Research Group in Oxford, without whom it would have taken even longer; it finally came together after a conversation in the Usenet newsgroup comp.lang.functional. The notation and style are those of Richard Bird's Theory of Lists (Bird, 1987).

References

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.

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

Geraint Jones and Mary Sheeran (1990). *Circuit design in Ruby.* In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design.* North-Holland.

Lambert Meertens (1988). *First steps towards the theory of rose trees.* CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25.