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

Geraint Jones[*] Jeremy Gibbons[†]

Abstract

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

1 Breadth-first ordering

The algorithms which are developed in this paper relate trees to lists in a way that respects 'breadth-first' ordering on the nodes of a tree. This is the order in which nodes nearer the root are considered to be earlier and – perhaps arbitrarily, but for definiteness – those at the same depth from the root are ordered lexicographically by the the 'left to right' seniority of their line of descent.

This paper works with rose trees, in which each node has an arbitrary finite number of immediate descendants

$$
\begin{aligned}
\mathit{tree}\ \alpha &= \mathit{ta} \quad \text{where } \mathit{ta} ::= \mathit{Nd}\ \alpha\ [\mathit{ta}] \\
\mathit{root}(\mathit{Nd}\ a\ \mathit{ts}) &= a \\
\mathit{subs}(\mathit{Nd}\ a\ \mathit{ts}) &= \mathit{ts}
\end{aligned}
$$

and will develop implementations of the breadth-first scanning and labelling functions bfs and lab

$$
\begin{aligned}
\mathit{bfs} &:: (\mathit{tree}\ \alpha) \rightarrow [\alpha] \\
\mathit{lab} &:: [\alpha] \rightarrow (\mathit{tree}\ \beta) \rightarrow (\mathit{tree}\ \alpha)
\end{aligned}
$$

The value of $\mathit{bfs}\ t$ is a list of the values associated with the nodes of t, starting with $\mathit{root}\ t$, in breadth-first order. The value of $\mathit{lab}\ \mathit{xs}\ t$ is a tree of the same shape as t but labelled with elements of xs in breadth-first order, starting with $\mathit{root}(\mathit{lab}\ \mathit{xs}\ t) = \mathit{hd}\ \mathit{xs}$. The list xs will be assumed to be long enough that it is not exhausted before the whole tree has been labelled.

[*]Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, United Kingdom.

[†]Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand.

2 Breadth-first scan

The development starts from a close relative of breadth-first scan, sometimes called the level-order scan [2].

$$\mathit{lfs} :: (\mathit{tree}\ \alpha) \rightarrow [[\alpha]]$$

The elements of $\mathit{lfs}\ t$ are lists of labels of nodes at the same depth in the tree, the first of these lists being of the only label at depth zero.

$$\mathit{hd}(\mathit{lfs}\ t) = [a] \tag{1}$$

The remainder of the $\mathit{lfs}\ t$ must contain the elements of the lists $\mathit{lfs} * (\mathit{subs}\ t)$ obtained by mapping lfs over the immediate sub-trees of t. Corresponding elements of these lists represent nodes at the same depth in the parent tree, so can be concatenated to give the levels of the parent tree.

$$\mathit{tl}(\mathit{lfs}\ t) = \Upsilon_{+\!\!+} \,/\, (\mathit{lfs} * (\mathit{subs}\ t)) \tag{2}$$

If $\oplus$ is an associative operator, $\oplus \,/\, \mathit{xs}$ denotes the result of folding or reducing xs with $\oplus$, and $\Upsilon_\oplus$ is the 'long zip' operation that combines corresponding elements of two lists with $\oplus$, returning a list as long as the longer argument.

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

Equations 1 and 2 are adequate to implement level-order scan, and the breadth-first scan can be obtained from this by concatenating the elements of the level-order scan, whence the following definition

$$
\begin{aligned}
\mathit{bfs} &= +\!\!+/ \cdot \mathit{lfs} \\
&\qquad \text{where } \mathit{lfs}(\mathit{Nd}\ a\ \mathit{ts}) = [a] : (\Upsilon_{+\!\!+} / \cdot \mathit{lfs}*)\mathit{ts}
\end{aligned} \tag{3}
$$

which is adequate as an implementation of bfs in a functional programming language. This program has time complexity $\mathrm{O}(n \log n)$ in the length n of its output, which is the number of nodes in the tree. Eventually the dominant term is the cost of constructing the 'levels', each of which is $\mathrm{O}(n)$ long, each by a tree of catenation operations.

It is well known – a 'folk fact' – that there is a neat program that implements bfs in time linear in the length of its output. The first step towards discovering it is to apply bfs to an example tree and calculate the consequences.

$$
\begin{aligned}
\mathit{bfs}(\mathit{Nd}\ a\ \mathit{ts}) &= +\!\!+ \,/\, ([a] : (\Upsilon_{+\!\!+} / \cdot \mathit{lfs}*)\ \mathit{ts}) \\
&= a : \mathit{bfss}\ \mathit{ts} \\
&\qquad \text{where } \mathit{bfss} = +\!\!+/ \cdot \Upsilon_{+\!\!+} / \cdot \mathit{lfs}*
\end{aligned}
$$

The function *bfss* generalizes *bfs* by calculating a breadth-first scan of a forest of trees whose roots are regarded as being at the same level, however this intuition is not needed to be able to synthesize a recursive definition for it.

$$
\begin{array}{lll}
bfss[\,] & = & \{\text{definition of } bfss\} \\
 & & \mathbin{+\!\!+} / \, (\Upsilon_{+\!\!+} \, / \, (lfs * [\,])) \\
 & = & \{[\,] \text{ is a unit for } \mathbin{+\!\!+} \text{ and } \Upsilon_{+\!\!+}\} \\
 & & [\,]
\end{array}
$$

In the calculation for non-empty forests, an interleaving lemma is used: that for any associative homogeneous $\oplus$

$$\oplus \, / \, ((x:xs) \Upsilon_{\oplus} ys) \quad = \quad x \oplus (\oplus \, / \, (ys \Upsilon_{\oplus} xs))$$

which can be proved by induction on xs and ys.

$$
\begin{array}{ll}
 & bfss((Nd \; a \; ts) : us) \\
= & \quad \{\text{definition of } bfss\} \\
 & \mathbin{+\!\!+} / \, (\Upsilon_{+\!\!+} \, / \, (lfs * ((Nd \; a \; ts) : us))) \\
= & \quad \{\text{map and fold distribute over cons}\} \\
 & \mathbin{+\!\!+} / \, ((lfs (Nd \; a \; ts)) \Upsilon_{+\!\!+} (\Upsilon_{+\!\!+} \, / \, (lfs * us))) \\
= & \quad \{\text{definition of } lfs\} \\
 & \mathbin{+\!\!+} / \, (([a] : (\Upsilon_{+\!\!+} \, / \, (lfs * ts))) \Upsilon_{+\!\!+} (\Upsilon_{+\!\!+} \, / \, (lfs * us))) \\
= & \quad \{\text{lemma above}\} \\
 & [a] \mathbin{+\!\!+} (\mathbin{+\!\!+} / \, ((\Upsilon_{+\!\!+} \, / \, (lfs * us)) \Upsilon_{+\!\!+} (\Upsilon_{+\!\!+} \, / \, (lfs * ts)))) \\
= & \quad \{\text{fold and map distribute over } \mathbin{+\!\!+}\} \\
 & a : (\mathbin{+\!\!+} / \, (\Upsilon_{+\!\!+} \, / \, (lfs * (us \mathbin{+\!\!+} ts)))) \\
= & \quad \{\text{definition of } bfss\} \\
 & a : bfss(us \mathbin{+\!\!+} ts)
\end{array}
$$

Noticing that $bfs(Nd \; a \; ts) = bfss[Nd \; a \; ts]$ one arrives at the following program

$$
\begin{array}{lllll}
bfs \; t & = & bfss[t] & & \\
 & & \text{where} \quad bfss[\,] & = & [\,] \\
 & & bfss((Nd \; a \; ts) : us) & = & a : bfss(us \mathbin{+\!\!+} ts)
\end{array}
$$

Although this program is often seen in use, and may even be slightly more efficient for small trees, it has a worse asymptotic complexity than that in equation 3. Each of $O(n)$ collections of siblings ts in the tree is appended to a queue us of trees higher in the ordering. However us grows linearly with the size of the tree, so this operation alone costs $O(n^2)$.

Fortunately there is a standard way of efficiently eliminating queues from functional programs, provided they are never copied. This is the same technique as linearises naïve reverse. Let $f \; xs \; ys = bfss(xs \mathbin{+\!\!+} rev \; ys)$ where rev is list reverse, then synthesizing a definition of f

$$
\begin{array}{rcl}
f \; [\,] \; [\,] & = & bfss \; [\,] \\
 & = & [\,] \\
f \; [\,] \; (y : ys) & = & bfss(rev(y : ys)) \\
 & = & f \; (rev \; ys) \; [y] \\
f \; (Nd \; a \; ts : xs) \; ys & = & bfss((Nd \; a \; ts : xs) \mathbin{+\!\!+} rev \; ys) \\
 & = & a : bfss(xs \mathbin{+\!\!+} (rev \; ys) \mathbin{+\!\!+} ts) \\
 & = & a : f \; xs \; ((rev \; ts) \mathbin{+\!\!+} ys)
\end{array}
$$

and finally to eliminate the remaining rev and $\mathbin{+\!\!+}$ operations, let $g \; xs \; ys = (rev \; xs) \mathbin{+\!\!+} ys$. A definition for g recursing on its first argument can be synthesized in the same way, to give the following program

$$
\begin{array}{lllll}
bfs \; t & = & f \; [t] \; [\,] & & \\
 & & \text{where} \quad f \; [\,] \; [\,] & = & [\,] \\
 & & f \; [\,] \; (y : ys) & = & f \; (g \; ys \; [y]) \; [\,] \\
 & & f \; (Nd \; a \; ts : xs) \; ys & = & a : f \; xs \; (g \; ts \; ys) \\
 & & g \; [\,] \; ys & = & ys \\
 & & g \; (x : xs) \; ys & = & g \; xs \; (x : ys)
\end{array}
$$

The asymptotic complexity of this program is linear, because g is linear in the length of its first argument, and f causes it to be applied once to each group ts of siblings, and once to each of the reversed queues ys, and the total length of each of these collections is bounded by the number of nodes.

3 Directed folds and accumulations

The left-fold $\oplus \nrightarrow_e$ of $\oplus$ seeded by e can be defined by

$$
\begin{array}{rcl}
\oplus \nrightarrow_e [\,] & = & e \\
\oplus \nrightarrow_e (x : xs) & = & \oplus \nrightarrow_{x \oplus e} xs
\end{array}
$$

The $\oplus$ operator need not be associative in this case, and the above equations suggest a linear sequential implementation of left-fold. The corresponding left-accumulation $\oplus \not\nrightarrow_e$ also has a linear sequential implementation, but is adequately defined by

$$(\oplus \nrightarrow_e) * (inits \; xs) \quad = \quad (\oplus \not\nrightarrow_e xs) \mathbin{+\!\!+} [\oplus \nrightarrow_e xs]$$

where *inits xs* is the list of initial segments of xs in order of increasing length. Notice that for later convenience left-accumulation here is not quite the same as the standard *scan* function, but returns a list of the same length as its list argument.

Right-handed folds and accumulations are similar,

$$
\begin{array}{rcll}
\oplus \nleftarrow_e [\,] & = & e & \\
\oplus \nleftarrow_e (x : xs) & = & x \oplus (\oplus \nleftarrow_e xs) & \\
(\oplus \nleftarrow_e) * (tails \; xs) & = & [\oplus \nleftarrow_e xs] \mathbin{+\!\!+} (\oplus \not\nleftarrow_e xs) & (4)
\end{array}
$$

where *tails xs* is the list of final segments of xs in order of decreasing length. Right-fold and scan also have linear sequential implementations, of course.

Moreover, induction on xs, from equation 4 shows that

$$(\oplus \nleftarrow_e) * (tails \; xs) \quad = \quad (xs \Upsilon_{\oplus} (\oplus \not\nleftarrow_e xs)) \mathbin{+\!\!+} [e]$$

and so that the infinite list

$$xss \quad = \quad (xs \Upsilon_{\oplus} (\oplus \not\nleftarrow_e xs)) \mathbin{+\!\!+} es \quad \text{where } es = e : es$$

is the unique total solution to

$$xss \quad = \quad xs \Upsilon_{\oplus} (tl \; xss) \tag{5}$$

Right-folds can be used to implement folds of associative operators $\oplus$, for by induction on the list xs

$$
\begin{array}{rcll}
(\oplus \, / \, (f * xs)) \oplus e & = & \otimes \nleftarrow_e xs & (6) \\
 & & \text{where } x \otimes y = (f \; x) \oplus y &
\end{array}
$$

even though $\otimes$ may not itself be associative. Left-fold has a corresponding property, and if e is a unit of f it follows that $\oplus / = \oplus \nleftarrow_e = \oplus \nrightarrow_e$.

4 A second linear breadth-first scan algorithm

The algorithm for *lfs* in equation 3 reduces to one of linear complexity if the list catenations are replaced by constant time joins. The data type is one of binary tress

$$joinlist\ \alpha \quad = \quad ja \quad \text{where } ja ::= \langle\alpha\rangle \mid ja \circledast ja$$

where $\circledast$ represents catenation under the abstraction

$$\begin{aligned} flat &:: (joinlist\ \alpha) \rightarrow [\alpha] \\ flat\langle a\rangle &= [a] \\ flat(p \circledast q) &= (flat\ p) +\!\!+ (flat\ q) \end{aligned}$$

Concatenations can be eliminated by right-fold introduction

$$\begin{aligned} +\!\!+ / (flat * xs) \quad &= \quad \oplus \not\leftarrow_{[\,]} xs \\ &\quad\ \text{where } c \oplus xs = (flat\ c) +\!\!+ xs \end{aligned}$$

and $\oplus$ has an implementation linear in the size of its left argument. This can be synthesized in the usual way

$$\begin{aligned} \langle a\rangle \oplus xs &= flat\langle a\rangle +\!\!+ xs \\ &= a : xs \\ (p \circledast q) \oplus xs &= flat(p \circledast q) +\!\!+ xs \\ &= (flat\ p) +\!\!+ ((flat\ q) +\!\!+ xs) \\ &= p \oplus (q \oplus xs) \end{aligned}$$

Introducing this operation into the definition of *bfs*,

$$\begin{aligned} bfs\ t &= +\!\!+ / (lfs\ t) \\ &= +\!\!+ / (flat * (f\ t)) \\ &\quad \text{where } flat * (f\ t) = lfs\ t \end{aligned}$$

The auxiliary function f satisfies

$$\begin{aligned} & flat * (f\ (Nd\ a\ ts)) \\ =\ & \quad \{\text{definition of } lfs\} \\ & [a] : (\Upsilon_{+\!\!+} / \cdot lfs*) ts \\ =\ & \quad \{\text{right-fold introduction, equation 6}\} \\ & flat\langle a\rangle : (\odot \not\leftarrow_{[\,]} ts) \\ & \text{where } t \odot xs = (lfs\ t)\Upsilon_{+\!\!+} xs \end{aligned}$$

The operations on lists in the definition of $\odot$ can be replaced by operations on join lists, for

$$\begin{aligned} t \odot (flat * ps) \quad =\ & \quad \{\text{definition of } lfs\} \\ & (flat * (f\ t))\Upsilon_{+\!\!+}(flat * ps) \\ =\ & \quad \{\text{definition of } flat\} \\ & flat * ((f\ t)\Upsilon_{\circledast} ps) \\ =\ & flat * (t \otimes ps) \\ & \text{where } t \otimes ps = (f\ t)\Upsilon_{\circledast} ps \end{aligned}$$

and substituting back into the calculation for f

$$\begin{aligned} flat * (f\ (Nd\ a\ ts)) &= flat\langle a\rangle : flat * (\otimes \not\leftarrow_{[\,]} ts) \\ &= flat * (\langle a\rangle : (\otimes \not\leftarrow_{[\,]} ts)) \end{aligned}$$

This has a solution for f which completes an $O(n)$ program with a structure closer to that of the original program.

$$\begin{aligned} bfs\ t \quad = \quad & \oplus\not\leftarrow_{[\,]}(t \otimes [\,]) \\ & \text{where } \langle a\rangle \oplus xs = a : xs \\ & \quad (p \circledast q) \oplus xs = p \oplus (q \oplus xs) \\ & \quad (Nd\ a\ ts) \otimes ps = (\langle a\rangle : (\otimes \not\leftarrow_{[\,]} ts))\Upsilon_{\circledast} ps \end{aligned}$$

On reasonable assumptions this program has a cost per node higher than that given earlier, but only by a small factor, and it is much easier to understand.

5 Breadth-first labelling

The synthesis of a definition of *lab* begins by inversion of the definition of *bfs*. Recall that

$$\begin{aligned} & lfs(Nd\ a\ ts) \\ & \quad = [a] : (\Upsilon_{+\!\!+} / \cdot lfs*) ts \\ & \quad = \quad \{\text{right-fold introduction, equation 6}\} \\ & \qquad [a] : \odot \not\leftarrow_{[\,]} ts \\ & \qquad \text{where } t \odot ys = (lfs\ t)\Upsilon_{+\!\!+} ys \end{aligned}$$

The $\odot$ operator (which is the same as that in the previous section) is invertible in the sense that if $xss = t \odot yss$ the value of yss is determined by xss and t. That is to say, the equation

$$\begin{aligned} xss \quad &= \quad (xss \oplus t) \odot (xss \otimes t) \\ &\quad\ \text{where } xss \oplus t = lab\ xs\ t \end{aligned} \tag{7}$$

defines $xss \otimes t$. Conversely if xss and $xss \otimes t$ are known, $xss \oplus t$ is determined by this equation and the requirement that it have the same shape as t.

The specification of *lab* requires that the *bfs* of a tree labelled by xs be a prefix of xs, which is to say that there is a list ys for which

$$\begin{aligned} xs \quad =\ & bfs(xss \oplus t) +\!\!+ ys \\ =\ & \quad \{\text{definition of } bfs\} \\ & (+\!\!+ / (lfs(xss \oplus t))) +\!\!+ ys \\ =\ & \quad \{\text{right-fold introduction, equation 6}\} \\ & +\!\!+ \not\leftarrow_{ys} (lfs(xss \oplus t)) \\ =\ & hd\ yss \\ & \text{where } yss = (lfs(xss \oplus t))\Upsilon_{+\!\!+}(tl\ yss) \\ & \qquad\quad\ = (xss \oplus t) \odot (tl\ yss) \end{aligned}$$

Although this last equation may look like a program, it is not a definition of xs. (If it were and $xss \oplus t$ were a finite tree, xs would be a partial list.) In fact xs is known, and the equation is to be solved for xss and yss. Equation 7 suggests a solution: if $tl\ yss = xss \otimes t$ then the defining property of $\otimes$ gives $yss = xss$ and $xs = hd\ yss = hd\ xss$.

These equations fully define an xss and give the following outline of a program for *lab*

$$lab\ xs\ t \quad = \quad xss \oplus t \quad \text{where } xss = xs : (xss \otimes t)$$

It remains only to synthesize definitions for $\oplus$ and $\otimes$.

$$\begin{aligned} hd\ xss \quad =\ & hd((xss \oplus t) \odot (xss \otimes t)) \\ =\ & \quad \{\text{definition of } \odot\} \\ & hd(lfs(xss \oplus t)) +\!\!+ hd(xss \otimes t) \\ =\ & \quad \{\text{definition of } lfs\text{, elaborating } hd \cdot lfs\} \\ & [root(xss \oplus t)] +\!\!+ hd(xss \otimes t) \end{aligned}$$

whence deduce that

$$\begin{aligned} root(xss \oplus t) &= hd(hd\ xss) \\ hd(xss \otimes t) &= tl(hd\ xss) \end{aligned}$$

Similarly

$$\begin{aligned}
tl\ xss &= tl((xss \oplus t) \odot (xss \otimes t)) \\
&= \qquad \{\text{ definition of } \odot \,\} \\
&\quad\ tl(lfs(xss \oplus t))\Upsilon_{+\!\!+}\, tl(xss \otimes t) \\
&= \qquad \{\text{ definition of } lfs\text{, elaborating } tl \cdot lfs \,\} \\
&\quad\ (\Upsilon_{+\!\!+} \,/\, (lfs * (subs(xss \oplus t))))\ \Upsilon_{+\!\!+}\ tl(xss \otimes t) \\
&= \qquad \{\text{ right-fold introduction, equation 6 }\} \\
&\quad\ \odot \mathbin{/\!\!\!-}_{tl(xss\otimes t)}\ (subs(xss \oplus t))
\end{aligned}$$

This is an equation of the form $u = \odot \mathbin{/\!\!\!-}_{w}\ vs$ where u is known. Moreover $vs = subs(xss \oplus t)$ must be a forest of the same shape as a given $ts = subs\ t$, since $xss \oplus t$ is to be a tree of the same shape as t. It is necessary to solve for w and vs.

This is just like solving a nested sequence of equations of the form $u = v \odot w$, each for a w of the same shape as a given t. By equation 7 one such equation is solved by $v = u \oplus t$ and $w = u \otimes t$. The analogous solution to the sequence of equations must involve $vs = zs\Upsilon_{\oplus}ts$ for some zs and by induction on ts the solution can be shown to be

$$\begin{aligned}
vs &= (\otimes \mathbin{/\!\!\!\#}_{u}\ ts)\Upsilon_{\oplus}ts \\
w &= \otimes \mathbin{/\!\!\!-}_{u}\ ts
\end{aligned}$$

so deduce that

$$\begin{aligned}
subs(xss \oplus t) &= (\otimes \mathbin{/\!\!\!\#}_{tl\ xss}\ (subs\ t))\Upsilon_{\oplus}(subs\ t) \\
tl(xss \otimes t) &= \otimes \mathbin{/\!\!\!-}_{tl\ xss}\ (subs\ t)
\end{aligned}$$

The $\oplus$ and $\otimes$ operators are now fully determined, for

$$\begin{aligned}
xss \oplus t &= Nd\ (root(xss \oplus t))\ (subs(xss \oplus t)) \\
&= Nd\ (hd(hd\ xss)) \\
&\qquad ((\otimes \mathbin{/\!\!\!\#}_{tl\ xss}\ (subs\ t))\Upsilon_{\oplus}(subs\ t)) \\
xss \otimes t &= hd(xss \otimes t) : tl(xss \otimes t) \\
&= tl(hd\ xss) : (\otimes \mathbin{/\!\!\!-}_{tl\ xss}\ (subs\ t))
\end{aligned}$$

There is some inefficiency in calculating both the accumulation and the fold which would be the next term in a longer accumulation. This can be avoided by tupling the two calculations and defining a function something like

$$g_{\oplus,\otimes}\ e\ ys = ((\otimes \mathbin{/\!\!\!\#}_{e}\ ys)\Upsilon_{\oplus}ys),\ (\otimes \mathbin{/\!\!\!-}_{e}\ ys)$$

It happens that g has a familiar pattern, so define

$$\begin{aligned}
row &:: (\alpha \to \beta \to (\gamma, \alpha)) \to \alpha \to [\beta] \to ([\gamma], \alpha) \\
row\ f\ e\ ys &= (\otimes \mathbin{/\!\!\!\#}_{e}\ ys)\Upsilon_{\oplus}ys,\ \otimes \mathbin{/\!\!\!-}_{e}\ ys \\
&\quad\ \text{where } e \oplus x, e \otimes x = f\ e\ x
\end{aligned}$$

Just as folds and accumulations have linear sequential implementations, so also does row, and this can be synthesized in the usual way.

$$\begin{aligned}
&row\ f\ e\ [\,] \\
&\quad = (\otimes \mathbin{/\!\!\!\#}_{e}\,[\,])\Upsilon_{\oplus}[\,],\ \otimes \mathbin{/\!\!\!-}_{e}\,[\,] \\
&\quad = [\,], e \\
&row\ f\ e\ (x : xs) \\
&\quad = (\otimes \mathbin{/\!\!\!\#}_{e}\ (x : xs))\Upsilon_{\oplus}(x : xs),\ \otimes \mathbin{/\!\!\!-}_{e}\ (x : xs) \\
&\quad = (e : (\otimes \mathbin{/\!\!\!\#}_{e\otimes x}\ xs))\Upsilon_{\oplus}(x : xs),\ \otimes \mathbin{/\!\!\!-}_{e\otimes x}\ xs \\
&\quad = (e \oplus x) : ((\otimes \mathbin{/\!\!\!\#}_{e\otimes x}\ xs)\Upsilon_{\oplus}xs),\ \otimes \mathbin{/\!\!\!-}_{e\otimes x}\ xs \\
&\quad = y : ys, d \\
&\qquad\ \text{where} \quad y = e \oplus x \\
&\qquad\qquad\quad\ ys, d = row\ f\ (e \otimes x)\ xs
\end{aligned}$$

The name of the row abstraction comes from work on VLSI layout [3], where it has to do with a row of tiles in a two-dimensional rectangularly connected grid. It might also help to think of it in terms of an automaton whose state-transition function is f, taking a state of type α and an input of type β and producing output of type γ as it moves into the next state; $row\ f$ describes the sequence of transitions involved in consuming a list of inputs.

Using row the definition of f can be recast as

$$\begin{aligned}
f\ xss\ t &= xss \oplus t, xss \otimes t \\
&= Nd\ (hd(hd\ xss))\ ts, (tl(hd\ xss)) : yss \\
&\quad\ \text{where } ts, yss = row\ f\ (tl\ xss)\ (subs\ t)
\end{aligned}$$

and substituting this into the skeleton definition of lab gives the complete program.

$$\begin{aligned}
lab\ xs\ t &= u \\
&\quad\ \text{where } u, xss = f\ (xs : xss)\ t \\
f\ ((x : xs) : xss)\ (Nd\ a\ ts) & \\
&= Nd\ x\ us, xs : yss \\
&\quad\ \text{where } us, yss = row\ f\ xss\ ts \\
row\ f\ a\ [\,] &= [\,], a \\
row\ f\ a\ (x : xs) &= y : ys, c \\
&\quad\ \text{where } y, b = f\ a\ x \\
&\qquad\qquad\ ys, c = row\ f\ b\ xs
\end{aligned}$$

This implements lab with a cost which is linear in the number of nodes in 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 scans. 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 set 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* news group *comp.lang.functional*. The notation and style of presentation are those of Richard Bird's theory of lists [1].

References

[1] Richard Bird, *An introduction to the theory of lists*, in Manfred Broy (ed.), *Logic of programming and calculi of discrete design*, NATO ASI Series F Vol. 36, Springer-Verlag 1987, ISBN 0 387 18003 6. pp. 5–42.

[2] Jeremy Gibbons, *Algebras for tree algorithms*, Oxford University Computing Laboratory, Technical monograph PRG–94, September 1991, ISBN 0 902928 72 4.

[3] Geraint Jones and Mary Sheeran, *Circuit design in Ruby* in Jørgen Staunstrup (ed.), *Formal methods for VLSI design*, North-Holland, 1990, ISBN 0 444 88858 6. pp. 13–70.