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 \( \alpha \) are modelled as the least solution list.\( \alpha \) of the equation

\[
\text{list.} \alpha = [ ] \mid \alpha :: \text{list.} \alpha
\]

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

Streams of elements of type \( \alpha \) are modelled as the greatest solution stream.\( \alpha \) of the equation

\[
\text{stream.} \alpha = \alpha :: \text{stream.} \alpha
\]

That is, every stream of type stream.\( \alpha \) has a ‘head’ of type \( \alpha \) and a ‘tail’ of type stream.\( \alpha \). These components are extracted by the destructors \( \text{hd} \) and \( \text{tl} \). (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 \( \alpha \) to objects of type \( \beta \), then \( f* \) (pronounced ‘\( f \) map’) takes objects of type list.\( \alpha \) to objects of type list.\( \beta \), and objects of type stream.\( \alpha \)
to objects of type \texttt{stream.β}, by applying the function \(f\) to every element of the sequence. For example,

\[
f.*[x,y,z] = [fx, fy, fz]
\]

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

If \(⊕\) is an associative operator from \(α \times α\) to \(α\), then \(⊕/\) (pronounced ‘plussle reduce’) takes objects of type \texttt{list.α} to objects of type \(α\), by ‘inserting’ \(⊕\) between adjacent elements. For example,

\[
⊕/[x,y,z] = x ⊕ y ⊕ z
\]

The reduction \(⊕/\) of the empty list is the unit of \(⊕\), which is unique if it exists.

We also use directed reductions and accumulations on lists. The leftwards reduction \(⊕/σ_e\) of a list is defined by

\[
⊕/σ_e.[] = e
\]

\[
⊕/σ_e.(x :: xs) = x ⊕ ⊕/σ_e.xs
\]

For example,

\[
⊕/σ_e.[x,y,z] = x ⊕ (y ⊕ (z ⊕ e))
\]

Note that the \(⊕\) need not be associative.

The leftwards accumulation \(⊕/σ_e\) of a list is defined by

\[
⊕/σ_e.*.tails.xs = [⊕/σ_e.xs] ⊕ ⊕/σ_e.xs
\]

Here, \(tails.xs\) 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

\[
⊕/σ_e.[] = e
\]

\[
⊕/σ_e.(x :: xs) = ⊕/σ_e.x :: ⊕/σ_e.xs
\]

and rightwards accumulation satisfies

\[
⊕/σ_e.*.inits.xs = ⊕/σ_e.xs ⊕ [⊕/σ_e.xs]
\]

where \(inits\) returns the prefixes of a list.

The trees in this paper are \textit{non-empty homogeneous rose trees} (Meertens, 1988); every tree of type \texttt{tree.α} 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 \texttt{tree.α}. Trees are modelled as the least solution \(\texttt{tree.α}\) of the equation

\[
\texttt{tree.α} = \texttt{α} → \texttt{list.tree.α}
\]

For example, the tree
Linear-time breadth-first tree algorithms

![Tree Diagram](image)

is represented by the expression

\[ x_1 < [x_2 < [x_5], x_6], x_3], x_4 < [x_7] \]

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

The two tree destructors \( \text{rt} \) and \( \text{ch} \) are defined by

\[ t = \text{rt}.t \leftarrow \text{ch}.t \]

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

![Labeled Tree Diagram](image)

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 elements to a list of lists of elements—in fact, to a non-empty list of non-empty lists of elements. 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 \texttt{levels} has type \texttt{tree.\(\alpha\) \rightarrow list.list.\(\alpha\)} and is defined by

\[
\texttt{levels.}(x \leftarrow ts) = [x] :: \gamma_\oplus / .\texttt{levels}^*.ts
\]

Here, \(\gamma_\oplus\) 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,

\[
[x, y, z] \gamma_\oplus [u, v] = [x \oplus u, y \oplus v, z]
\]

and in general,

\[
\begin{align*}
\texttt{xs} \gamma_\oplus [] & = \texttt{xs} \\
[] \gamma_\oplus \texttt{ys} & = \texttt{ys} \\
(x :: \texttt{xs}) \gamma_\oplus (y :: \texttt{ys}) & = (x \oplus y) :: (x \gamma_\oplus \texttt{ys})
\end{align*}
\]

(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:

\[
\texttt{bft} = + / \circ \texttt{levels}
\]

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:

\[
\texttt{bft} = + / \circ \texttt{levels} \\
\quad \text{where } \texttt{levels.}(x \leftarrow ts) = [x] :: \gamma_\oplus / .\texttt{levels}^*.ts
\] — (1)

This program has time complexity \(O(n \log n)\) 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 \(O(n)\) 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 \texttt{bft} given above.

We observe that

\[
\begin{align*}
\texttt{bft.}(x \leftarrow ts) & = \{ \texttt{bft} \} \\
& = + / .\texttt{levels.}(x \leftarrow ts) \\
& = \{ \texttt{levels} \} \\
& = + / .([x] :: \gamma_\oplus / .\texttt{levels}^*.ts) \\
& = \{ \oplus /. (x :: \texttt{xs}) = x \oplus / \texttt{xs} \} \\
& = x :: + / .\gamma_\oplus / .\texttt{levels}^*.ts
\end{align*}
\]
Informally, \texttt{bfts} computes the traversal of a forest. Expanding \texttt{bfts}, we see that

\begin{equation}
\begin{align*}
\texttt{bfts} & = \{ \texttt{bfts} \} \\
& \quad \oplus / \cdot \gamma_\oplus / \cdot \texttt{levels\ast}.[] \\
& \quad \{ \texttt{f\ast}.[] = [] \} \\
& \quad \oplus / \cdot \gamma_\oplus / .[] \\
& \quad \{ \gamma_\oplus \text{ has unit } [] \} \\
& \quad \oplus / .[] \\
& \quad \{ \cdot \text{ has unit } [] \} \\
& \quad [] \\
\end{align*}
\end{equation}

so \texttt{bfts}.[ ] = [].

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

\[
\oplus / .((\langle x :: \mathbf{xs} \rangle \gamma_\oplus \mathbf{ys}) = x \oplus \oplus / .(\mathbf{ys} \gamma_\oplus \mathbf{xs})
\]

For example,

\[
\begin{align*}
\oplus / .((x_1 :: [x_2, x_3, x_4]) \gamma_\oplus [y_1, y_2, y_3]) \\
& = \{ \gamma, \text{ reduction } \} \\
& \quad (x_1 \oplus y_1) \oplus (x_2 \oplus y_2) \oplus (x_3 \oplus y_3) \oplus x_4 \\
& = \{ \text{associativity } \} \\
& \quad x_1 \oplus (y_1 \oplus x_2) \oplus (y_2 \oplus x_3) \oplus (y_3 \oplus x_4) \\
& = \{ \text{reduction, } \gamma \} \\
& \quad x_1 \oplus \oplus / .([y_1, y_2, y_3] \gamma_\oplus [x_2, x_3, x_4])
\end{align*}
\]

The general proof, by induction on \(\mathbf{xs}\) and \(\mathbf{ys}\), is straightforward and is omitted.

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

\[
\texttt{bfts}((x \leftarrow \mathbf{ts}) :: \mathbf{us}) \\
= \quad \{ \texttt{bfts} \} \\
& \quad \oplus / .\cdot \gamma_\oplus .\cdot \texttt{levels\ast}.((x \leftarrow \mathbf{ts}) :: \mathbf{us}) \\
& \quad \{ \map, \reduce \} \\
& \quad \oplus / .(\texttt{levels} .(x \leftarrow \mathbf{ts}) \gamma_\oplus \gamma_\oplus .\cdot \texttt{levels\ast}.\mathbf{us}) \\
& \quad \{ \texttt{levels} \} \\
& \quad \oplus / .([x] :: \gamma_\oplus .\cdot \texttt{levels\ast}.\mathbf{ts}) \gamma_\oplus \gamma_\oplus .\cdot \texttt{levels\ast}.\mathbf{us})
\]
Linear-time breadth-first tree algorithms

\[
\begin{align*}
\text{lemma } \quad & \{ [x] \mathbin{+}=/(\mathit{prefix} \mathbin{+} \mathit{levels}.\mathit{us} \mathbin{+} \mathit{prefix} \mathbin{+} \mathit{levels}.\mathit{ts}) \} \\
= & \quad \{ \text{promotion } \} \\
& \{ \text{bfts } \} \\
& \{ x :: \text{bfts.}(\mathit{us} \mathbin{+} \mathit{ts}) \}
\end{align*}
\]

Noting that
\[\text{bft.t = bfts.[t]}\]
yields the program
\[\text{bft.y = bfts.[t]} \quad \text{where} \quad \text{bfts.}[\mathit{us} = [\text{]}]
\]
\[\text{bfts.}(\mathit{xs} :: \mathit{us} \mathbin{+} \mathit{ts}) = \mathit{x} :: \text{bfts.}(\mathit{us} \mathbin{+} \mathit{ts})\]

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 \(\mathit{ts}\) to \(\mathit{us}\) takes time proportional to the length of \(\mathit{us}\), 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 \(\mathit{xs}\) as a pair of lists \((\mathit{ys}, \mathit{zs})\) such that \(\mathit{xs} = \mathit{ys} \mathbin{+} \text{rev.} \mathit{zs}\). We define
\[\text{fastbfts.}(\mathit{us}, \mathit{vs}) = \text{bfts.}(\mathit{us} \mathbin{+} \text{rev.} \mathit{vs})\]

(so that \(\text{bfts.ts} = \text{fastbfts.}(\mathit{ts}, [\text{]}))\) and calculate
\[\text{fastbfts.}(\mathit{us}, [\text{]}, [\text{]}))\]

\[= \quad \{ \text{fastbfts } \} \\
\text{bfts.}[\mathit{us} = [\text{]}]
\]

and
\[\text{fastbfts.}(\mathit{us}, [\text{]}, \mathit{v} :: \mathit{vs})\]

\[= \quad \{ \text{fastbfts } \} \\
\text{bfts.} \text{rev.}(\mathit{v} :: \mathit{vs})\]
This program takes linear amortized time in the length of its output, assuming \texttt{rev} is computed in linear time—the time is proportional to the time spent on \texttt{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
\[
\begin{align*}
    f([], []) & = 0 \\
    f([v :: vs], []) & = \#(v :: vs) + f.(v :: vs), [] \\
    f.((x :: ts) :: [u :: vs]), [] & = \#.ts + f.(u, rev.ts :: vs)
\end{align*}
\]
(where \# returns the length of a list), so that \( f.(us, vs) \) counts the ‘amount of time’ spent on \texttt{rev} in computing \texttt{fastbfts(us, vs)}, then we can show by induction that
\[
    f.(ts, []) = 2 \times +/.s*t.s 
\]
where \( s.t = \text{size} - 1 \)

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 \texttt{join.\alpha} of the equation
\[
\texttt{join.\alpha} = \Box.\alpha | \texttt{join.\alpha} + \texttt{join.\alpha}
\]
—that is, as non-empty leaf-labelled binary trees in which every parent has exactly two children. The abstraction function is \texttt{fringe : join.\alpha \rightarrow list.\alpha}, given by
\[
\begin{align*}
    \texttt{fringe.\Box x} & = [x] \\
    \texttt{fringe.(j + k)} & = \texttt{fringe.j + fringe.k}
\end{align*}
\]
Under this abstraction, \texttt{levels} is implemented by a function \texttt{jlevels}, of type \texttt{tree.\alpha \rightarrow list.join.\alpha}, such that
It is easy to see that the definition
\[ \text{jlevels} (x \leftarrow ts) = \square \cdot x :: \gamma_* \cdot \text{jlevels} \cdot ts \]
satisfies this requirement. (There are other definitions of \text{jlevels} that satisfy, because \text{fringe} is not injective, but this definition is the ‘simplest’.)

Computing \text{jlevels} \cdot t involves linearly many applications of \#\#—in fact, \text{size} \cdot t — \text{depth} \cdot 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, \text{bft} is given by
\[ \text{bft} = \#\#/ \circ \text{fringe} \circ \text{jlevels} \]
and so it remains only to compute \#\#/ \circ \text{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,
\[ \oplus / \cdot f \cdot x :: e = \oplus / \cdot f' \cdot x :: s \]
where \( x \cdot y = f \cdot x \oplus y \)  \hfill (2)

In particular, if \oplus has unit \( e \) then
\[ \oplus / \cdot f \cdot x :: s = \oplus / \cdot f_0 \cdot x :: s \]  \hfill (3)

Thus,
\[ \#\# / \circ \text{fringe} = \#\# / \circ \text{fringe} \]
where \( j \cdot x :: s \)

We can synthesize a definition of \( \otimes \) taking time proportional to the size of its left argument:
\[ \square \cdot x \otimes x :: s \]
\[ \{ \otimes \} \]
\[ \text{fringe} \cdot \square \cdot x \otimes x :: s \]
\[ \{ \text{fringe} \} \]
\[ x :: s \]
and
\[ (j \#\# k) \otimes x :: s \]
\[ \{ \otimes \} \]
\[ \text{fringe} \cdot (j \#\# k) \otimes x :: s \]
\[ \{ \text{fringe} \} \]
\[ \text{fringe} \cdot j \# \text{fringe} \cdot k \otimes x :: s \]
\[ \{ \otimes \} \]
\[ j \otimes (k \otimes x) \]
hence
\[ \text{bft} = \#\# / \circ \text{fringe} \circ \text{jlevels} \]
where
\[ \text{levels } (x \leftarrow \text{ts}) = \Box x :: \gamma_+ / \text{levels} \ast \text{ts} \]

and

\[
\begin{align*}
\Box x \otimes x s &= x :: x s \\
(j \mathbin{\oplus} k) \otimes x s &= j \otimes (k \otimes x s)
\end{align*}
\]

This program takes linear time.

5 Breadth-first labelling

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

\[
\text{shape } (\text{bfl} t).x s = \text{shape } t
\]

Here, \(\text{shape}\) takes a tree of \(\alpha\)s to a tree of \(1\)s, \(1\) being the unit type:

\[
\text{shape } = \text{unit}^* \quad \text{where } \text{unit} : \alpha \rightarrow 1
\]

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

\[
\text{bft} (\text{bfl} t).x s = \# \cdot \text{bft} t \downarrow x s
\]

Here, \(\downarrow\) (pronounced ‘take’) takes a number \(n\) and a stream \(x s\) and returns the list consisting of the first \(n\) elements of \(x s\):

\[
\begin{align*}
0 \downarrow x s &= [ ] \\
(n + 1) \downarrow (x :: x s) &= x :: (n \downarrow x s)
\end{align*}
\]

The derivation of \(\text{bfl}\) proceeds by inverting the specification of \(\text{bft}\). Recall that

\[
\begin{align*}
\text{levels } (x \leftarrow \text{ts}) &= \{ \text{levels} \} \\
[a] :: \gamma_+ / \text{levels} \ast \text{ts} &= \{ \text{specialisation (3): let } t \odot x s = \text{levels} t \gamma_+ x s \} \\
[a] :: \gamma_+ / \text{ts} &= \{ \text{specialisation (3): let } t \odot x s = \text{levels} t \gamma_+ x s \}
\end{align*}
\]

Informally, \(t \odot x s\) is the sequence of sequences obtained by placing the levelorder traversal of \(t\) ‘beside’ the sequence of sequences \(x s\). The operator \(\odot\) is invertible, in the sense that \(t\) and \(t \odot x s\) determine \(x s\); moreover, \(t \odot x s\) and \(\text{shape } t\) determine \(t\). That is, the equation

\[
x s s = (x s s \odot t) \odot (x s s \ominus t) \quad \text{— (4)}
\]

— together with the requirement that \(x s s \odot t\) be the same shape as \(t\)— determines \(x s s \odot t\) and \(x s s \ominus t\).

The requirement that \(x s s \odot t\) be the same shape as \(t\) suggests that there is some connection between \(\odot\) and relabelling. For which \(x s s\), if any, does the equation

\[
(bfl t).x s = x s s \odot t \quad \text{— (5)}
\]

hold?
In answering this question, we will encounter streams as solutions to the equation
\[ \text{ys} = \text{xs} \ U \text{tl.ys} \]
in \( \text{ys} \). The solutions of such equations are leftwards accumulations—the stream solutions are exactly streams \( \text{ys} \) of the form
\[ \text{ys} = (\text{xs} \ U \#_e \text{xs}) \# \text{es} \]
where \( \text{es} = e::e \) for various values of \( e \). Further, the equation
\[ \text{y} = \text{hd.ys} \]
has solutions in \( y \)
\[ y = \#_e \text{xs} \]
for various \( e \). This can be seen by writing out (6) as a series of equations on the elements of \( \text{ys} \): if \( \text{xs} = [x_1, \ldots, x_n] \) and \( \text{ys} = [y_1, y_2, \ldots] \), then (6) reduces to
\[
\begin{align*}
y_1 &= x_1 \# y_2 \\
y_2 &= x_2 \# y_3 \\
& \vdots \\
y_n &= x_n \# y_{n+1} \\
y_{n+1} &= y_{n+2} \\
y_{n+2} &= y_{n+3} \\
& \vdots
\end{align*}
\]
which can be solved from last to first.

Returning to investigate the consequences of (5), we note that one of the requirements on \( \text{bfl} \) is that \( \text{bfl.(bfl.t).xs} \) be a prefix of \( \text{xys} \), that is, there is a stream \( yss \) such that
\[
\begin{align*}
\text{xs} &= \{ \text{requirement} \} \\
\text{bfl.(bfl.t).xs} \# \text{ys} &= \{ (5) \} \\
\text{bfl.(xss \# t)} \# \text{ys} &= \{ \text{bfl} \} \\
# / \text{levels.(xss \# t)} \# \text{ys} &= \{ \text{specialisation (2)} \} \\
# / \text{ys.level.(xss \# t)} &= \{ \text{observation above} \} \\
\text{hd.yss} \quad \text{where yss = levels.(xss \# t) U tl.yss} &= \{ \text{tl} \} \\
\text{hd.yss} \quad \text{where yss = (xss \# t) U tl.yss}
\end{align*}
\]
Here, the \( x \) is known and the equation is to be solved for \( xss \) and \( yss \). Equation (4) suggests a solution: if \( t \cdot yss = xss \odot t \) then \( yss = xss \) and \( hd \cdot yss = x \). This determines \( xss \) in terms of \( x \) and \( t \):

\[
xss = x :: (xss \odot t)
\]

and outlines a program for \( bft \):

\[
(bft \cdot x) = xss \odot t \quad \text{where} \quad xss = x :: (xss \odot t)
\]  — (7)

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

We have

\[
hd \cdot xss
\]

\[
= \begin{cases} (4) \end{cases}
\]

\[
hd \cdot ((xss \odot t) \oplus (xss \ominus t))
\]

\[
= \begin{cases} \oplus \end{cases}
\]

\[
hd \cdot (\text{levels} \cdot (xss \odot t) \bigoplus (xss \ominus t))
\]

\[
= \begin{cases} \text{hd and } \bigoplus \end{cases}
\]

\[
hd \cdot \text{levels} \cdot (xss \odot t) \uplus hd \cdot (xss \ominus t)
\]

\[
= \begin{cases} \text{hd} \cdot \text{levels} \cdot t = [rt \cdot t] \end{cases}
\]

\[
[rt \cdot (xss \odot t)] \uplus hd \cdot (xss \ominus t)
\]

whence we deduce that

\[
rt \cdot (xss \odot t) = hd \cdot hd \cdot xss
\]

\[
hd \cdot (xss \odot t) = tl \cdot hd \cdot xss
\]

Similarly, we have

\[
tl \cdot xss
\]

\[
= \begin{cases} (4) \end{cases}
\]

\[
tl \cdot ((xss \odot t) \oplus (xss \ominus t))
\]

\[
= \begin{cases} \oplus \end{cases}
\]

\[
tl \cdot (\text{levels} \cdot (xss \odot t) \bigoplus (xss \ominus t))
\]

\[
= \begin{cases} \text{tl and } \bigoplus \text{ (and levels yields non-empty lists) } \end{cases}
\]

\[
tl \cdot \text{levels} \cdot (xss \odot t) \bigoplus tl \cdot (xss \ominus t)
\]

\[
= \begin{cases} \text{tl} \cdot \text{levels} \cdot u = \bigoplus / \cdot \text{levels} \cdot \text{ch} \cdot u \end{cases}
\]

\[
\bigoplus / \cdot \text{levels} \cdot \text{ch} \cdot (xss \odot t) \bigoplus tl \cdot (xss \ominus t)
\]

\[
= \begin{cases} \text{specialisation (2) } \end{cases}
\]

\[
\bigoplus / \cdot tl \cdot (xss \odot t) \cdot \text{ch} \cdot (xss \odot t)
\]

in which

\[
\text{shape} \cdot \text{ch} \cdot (xss \odot t) = \text{shape} \cdot \text{ch} \cdot t
\]
This is a pair of equations of the form
\[ u = \square_{w}v, \quad \text{shape} \cdot v = \text{shape} \cdot ts \]
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
\[
\begin{align*}
u &= v_1 \oplus u_2 \\
\text{shape} \cdot v_1 &= \text{shape} \cdot t_1 \\
u_2 &= v_2 \oplus u_3 \\
\text{shape} \cdot v_2 &= \text{shape} \cdot t_2 \\
& \vdots \\
u_n &= v_n \oplus w \\
\text{shape} \cdot v_n &= \text{shape} \cdot t_n
\end{align*}
\]
where \( u \) and each \( t_i \) are known, and we must find \( w \) and each \( v_i \). By equation (4), the pair of equations
\[
\begin{align*}
u &= v \oplus w \\
\text{shape} \cdot v &= \text{shape} \\
\end{align*}
\]
is solved by
\[
\begin{align*}
v &= u \ominus t \\
w &= u \ominus t
\end{align*}
\]
Thus we have
\[
\begin{align*}
v_1 &= u \ominus t_1 \\
v_2 &= u_2 \ominus t_2 \\
& = (u \ominus t_1) \ominus t_2 \\
& = (u \ominus t_1) \ominus t_2 \\
& \vdots \\
v_n &= ((u \ominus t_1) \ominus \cdots \ominus t_{n-1}) \ominus t_n \\
w &= ((u \ominus t_1) \ominus \cdots \ominus t_{n-1}) \ominus t_n
\end{align*}
\]
That is,
\[
\begin{align*}
\text{vs} &= \ominus \#_{u}ts \ \text{by ts} \\
w &= \ominus \#_{u}ts
\end{align*}
\]
Thus, the pair of equations
\[
\begin{align*}
\text{tl} \cdot xss &= \square_{tl \cdot xss \cdot \text{ch}} \cdot (xss \ominus t) \\
\text{shape} \cdot \text{ch} (xss \ominus t) &= \text{shape} \cdot \text{ch} \cdot t
\end{align*}
\]
has solutions
\[
\begin{align*}
\text{ch} (xss \ominus t) &= \ominus \#_{tl \cdot xss} \cdot \text{ch} \cdot t \ \text{by ch} \cdot t \\
\text{tl} (xss \ominus t) &= \ominus \#_{tl \cdot xss} \cdot \text{ch} \cdot t
\end{align*}
\]
Assembling these discoveries, we get
\[
\begin{align*}
xss \ominus t &= \text{rt} (xss \ominus t) \leftarrow \text{ch} (xss \ominus t) \\
& = \text{hd} \cdot \text{hd} \cdot xss \leftarrow (\ominus \#_{tl \cdot xss} \cdot \text{ch} \cdot t \ \text{by ch} \cdot t) \\
xss \ominus t &= \text{hd} (xss \ominus t) :: \text{tl} (xss \ominus t) \\
& = \text{tl} \cdot \text{hd} \cdot xss :: \ominus \#_{tl \cdot xss} \cdot \text{ch} \cdot t
\end{align*}
\]
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
\[
g_{\ominus \ominus} (e, xss) = (\ominus \#_{e} \cdot xss \ \text{by e} \cdot xss, \ominus \#_{e} \cdot xss)
\]
As it happens, this \( g \) has a familiar form: it arises from work on VLSI layout (Jones and Sheeran, 1990), where it is called \text{row} and has to do with a row of tiles in a two-dimensional rectangularly connected grid:
Linear-time breadth-first tree algorithms

\[(\text{row}.f).(e, ys) = (\otimes \#_a.xs \gamma_{\oplus} xs, \otimes \bigoplus_a.xs)\]

where \((a \oplus x, a \otimes x) = f.(a, x)\)

It might help to think of \textit{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 \((\text{row}.f).(a, xs)\) produces a pair \((ys, c)\) consisting of a list \(ys\) of outputs and a final state \(c\).

An efficient characterisation of \textit{row} can be synthesized in the usual way:

\[\begin{align*}
(\text{row}.f).(a, []) &= \{ \text{row} \} \\
(\otimes \#_{a,}].] \gamma_{\oplus} [], \otimes \bigoplus_{a,}].]) &= \{ \text{reduction and accumulation} \} \\
([], a)
\end{align*}\]

and

\[\begin{align*}
(\text{row}.f).(a, x :: xs) &= \{ \text{row} \} \\
(\otimes \#_{a,}(x :: xs) \gamma_{\oplus} (x :: xs), \otimes \bigoplus_{a,}(x :: xs)) &= \{ \text{reduction and accumulation} \} \\
((a :: \otimes \#_{a@x}.xs) \gamma_{\oplus} (x :: xs), \otimes \bigoplus_{a@x}.xs) &= \{ \gamma \} \\
((a + x) :: (\otimes \#_{a@x}.xs \gamma_{\oplus} xs), \otimes \bigoplus_{a@x}.xs) &= \{ \oplus \text{ and } \otimes, \text{ from definition of } \text{row} \} \\
(y :: ys, c) \text{ where } (ys, c) = (\text{row}.f).(a \oplus x, xs) &= \{ \oplus \text{ and } \otimes, \text{ from definition of } \text{row} \} \\
(y :: ys, c) \text{ where } (y, b) = f.(a, x) &\quad (ys, c) = (\text{row}.f).(b, xs)
\end{align*}\]

If we define \(f\) by

\[f.(xss, t) = (xss \odot t, xss \ominus t)\]

then

\[\otimes \#_{\text{tl}.xss}.\text{ch}.t \gamma_{\ominus} \text{ch}.t, \otimes \bigoplus_{\text{tl}.xss}.\text{ch}.t) = (\text{row}.f).(\text{tl}.xss, \text{ch}.t)\]

and so

\[f.(xss, t) = \{ f \} \]

\[(xss \odot t, xss \ominus t) = \{ \ominus, \ominus \} \]

\[(\text{hd.hd}.xss \Leftarrow (\otimes \#_{\text{tl}.xss}.\text{ch}.t \gamma_{\ominus} \text{ch}.t), \text{tl.hd}.xss :: \otimes \bigoplus_{\text{tl}.xss}.\text{ch}.t)\]
More over, 
\[(\text{bfl}.t).\text{xss} = f\{ \text{substituting } \text{xss} \}\]
where \((\text{yss}, \text{zss}) = f.(\text{xss}, t)\)
\[\text{xss} = \text{xs} :: \text{zss}\]

which completes the derivation—the program
\[(\text{bfl}.t).\text{xss} = \text{yss} \quad \text{where } (\text{yss}, \text{zss}) = f.(\text{xs} :: \text{zss}, t)\]

where
\[f.(\text{x} :: \text{xs}) :: \text{yss}, \text{w} \leftarrow \text{ts}) = (\text{x} \leftarrow \text{us}, \text{xs} :: \text{yss}) \quad \text{where } (\text{us}, \text{yss}) = (\text{row}.f).(\text{xss}, \text{ts})\]

\[\text{row}.f.(\text{a}, []) = ([], \text{a})\]
\[\text{row}.f.(\text{a}, \text{x} :: \text{xs}) = (\text{y} :: \text{ys}, \text{c}) \quad \text{where } (\text{y}, \text{b}) = f.(\text{a}, \text{x})\]
\[\text{ys}, \text{c} = (\text{row}.f).(\text{b}, \text{xs})\]

implies \text{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


