Breadth-First Traversal Via Staging

Jeremy Gibbons

WG2.11#22, April 2023

1. Applicative functors

$$
\begin{array}{ll}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} & \\
\quad \mathit{unit} :: f\ () & \text{-- “skip”} \\
\quad (\otimes)\ :: f\ a \to f\ b \to f\ (a, b) & \text{-- “sequential composition”}
\end{array}
$$

with appropriate laws (“strong lax-monoidal”).

- every *monad* is applicative
- *colists* are applicative, under zipping
- *constant* functors over a monoid are applicative

Applicative traversal

$$\textbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \textbf{where}$$
$$\quad \mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \rightarrow f\ b) \rightarrow t\ a \rightarrow f\ (t\ b)$$

with laws (naturality, linearity, unitarity).

Eg left-to-right traversal of (finite) lists:

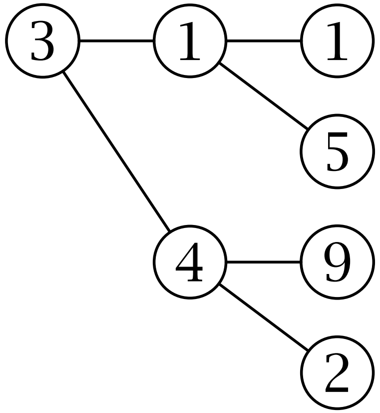
$$\textbf{instance}\ \mathit{Traversable}\ \mathit{List}\ \textbf{where}$$
$$\quad \mathit{traverse}\ f\ [\,] \qquad\quad = \mathit{pure}\ [\,]$$
$$\quad \mathit{traverse}\ f\ (x : \mathit{xs}) = \mathit{fmap}\ (\mathit{uncurry}\ (:))\ (f\ x \otimes \mathit{traverse}\ f\ \mathit{xs})$$

Trees

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a \quad = \mathit{Node}\ a\ (\mathit{Forest}\ a)\\
&\mathbf{type}\ \mathit{Forest}\ a = [\,\mathit{Tree}\ a\,]
\end{aligned}
$$

eg

$$
\begin{aligned}
&t :: \mathit{Tree}\ \mathit{Int}\\
&t = \mathit{Node}\ 3\ [\,\mathit{Node}\ 1\ [\,\mathit{Node}\ 1\ [\,]\\
&\qquad\qquad\qquad\qquad\qquad\quad ,\mathit{Node}\ 5\ [\,]\,]\\
&\qquad\qquad\qquad\quad ,\mathit{Node}\ 4\ [\,\mathit{Node}\ 9\ [\,]\\
&\qquad\qquad\qquad\qquad\qquad\quad ,\mathit{Node}\ 2\ [\,]\,]\,]
\end{aligned}
$$



Depth-first traversal

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Traversable\ Tree}\ \mathbf{where} \\
\quad \mathit{traverse}_{\mathit{Tree}}\ f\ (\mathit{Node}\ x\ \mathit{ts}) = \mathit{fmap}\ (\mathit{uncurry\ Node})\ (f\ x \otimes \mathit{traverseF}\ f\ \mathit{ts}) \\
\qquad \mathbf{where}\ \mathit{traverseF}\ f = \mathit{traverse}_{\mathit{List}}\ (\mathit{traverse}_{\mathit{Tree}}\ f)
\end{array}$$

- mutual recursion between *traverse* (trees) and *traverseF* (forests)
- similar in principle to left-to-right list traversal
- in fact, the outermost *traverse* in *traverseF* is that list traversal
- formulaic: can be derived from the datatype definition
- but what about *breadth-first* traversal?

Ask me later about. . .

- breadth-first *enumeration* using a queue

  one-pass, but non-compositional; how to preserve tree shape?

- breadth-first *traversal* via shape and contents

  compositional, but multi-pass

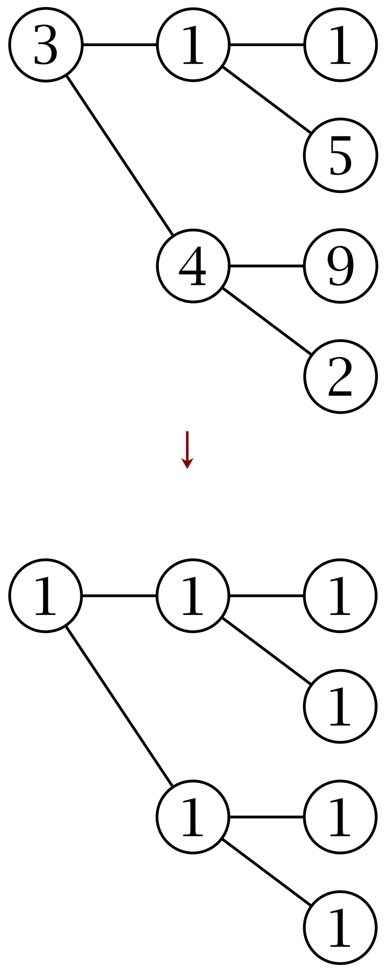
- breadth-first *relabelling* as a circular program

  compositional, one-pass, but needs laziness

2. Repmin

Replace every element of a tree with the minimum element in that tree:

$$
\begin{aligned}
&\mathit{repmin} :: \mathit{Tree\ Int} \to \mathit{Tree\ Int}\\
&\mathit{repmin}\ t = \mathit{replaceT}\ t\ (\mathit{minT}\ t)\ \mathbf{where}\\
&\quad \mathit{minT} :: \mathit{Tree\ Int} \to \mathit{Int}\\
&\quad \mathit{minT}\ (\mathit{Node}\ x\ [\,])\ = x\\
&\quad \mathit{minT}\ (\mathit{Node}\ x\ \mathit{ts})\ = \mathit{min}\ x\ (\mathit{minF}\ \mathit{ts})\\
&\quad \mathit{minF} :: \mathit{Forest\ Int} \to \mathit{Int}\\
&\quad \mathit{minF} = \mathit{minimum} \circ \mathit{map}\ \mathit{minT}\\
&\quad \mathit{replaceT} :: \mathit{Tree}\ a \to b \to \mathit{Tree}\ b\\
&\quad \mathit{replaceT}\ (\mathit{Node}\ x\ \mathit{ts})\ y = \mathit{Node}\ y\ (\mathit{replaceF}\ \mathit{ts}\ y)\\
&\quad \mathit{replaceF} :: \mathit{Forest}\ a \to b \to \mathit{Forest}\ b\\
&\quad \mathit{replaceF}\ \mathit{ts}\ y = [\,\mathit{replaceT}\ t\ y \mid t \leftarrow \mathit{ts}\,]
\end{aligned}
$$



but do so in a single pass rather than two.

Richard Bird's circular program

$$\mathit{repmin}_{\mathrm{RSB}} :: \mathit{Tree\ Int} \to \mathit{Tree\ Int}$$
$$\mathit{repmin}_{\mathrm{RSB}}\ t = \mathbf{let}\ (u, m) = \mathit{auxT}\ t\ m\ \mathbf{in}\ u \quad \text{-- circular!}$$
$$\quad \mathbf{where}$$
$$\qquad \mathit{auxT} :: \mathit{Tree\ Int} \to a \to (\mathit{Tree}\ a, \mathit{Int})$$
$$\qquad \mathit{auxT}\ (\mathit{Node}\ x\ [\,])\ y = (\mathit{Node}\ y\ [\,], x)$$
$$\qquad \mathit{auxT}\ (\mathit{Node}\ x\ ts)\ y = (\mathit{Node}\ y\ us, \mathit{min}\ x\ z)$$
$$\qquad\qquad\qquad \mathbf{where}\ (us, z) = \mathit{auxF}\ ts\ y$$
$$\qquad \mathit{auxF} :: \mathit{Forest\ Int} \to a \to (\mathit{Forest}\ a, \mathit{Int}) \quad \text{-- non-empty forest}$$
$$\qquad \mathit{auxF}\ ts\ y = (us, \mathit{minimum}\ ys)$$
$$\qquad\qquad \mathbf{where}\ (us, ys) = \mathit{unzip}\ [\,\mathit{auxT}\ t\ y \mid t \leftarrow ts\,]$$

(the **let** must be a **letrec**).

Alberto Pettorossi's higher-order program

$$
\begin{aligned}
& \mathit{repmin}_{\mathrm{ADP}} :: \mathit{Tree\ Int} \to \mathit{Tree\ Int} \\
& \mathit{repmin}_{\mathrm{ADP}}\ t = \mathbf{let}\ (u, m) = \mathit{auxT}\ t\ \mathbf{in}\ u\ m \quad \text{-- not circular} \\
& \quad \mathbf{where} \\
& \qquad \mathit{auxT} :: \mathit{Tree\ Int} \to (a \to \mathit{Tree}\ a, \mathit{Int}) \\
& \qquad \mathit{auxT}\ (\mathit{Node}\ x\ [\,]) = (\lambda y \to \mathit{Node}\ y\ [\,], x) \\
& \qquad \mathit{auxT}\ (\mathit{Node}\ x\ \mathit{ts}) = (\lambda y \to \mathit{Node}\ y\ (\mathit{us}\ y), \mathit{min}\ x\ z) \\
& \qquad\qquad\qquad\qquad \mathbf{where}\ (\mathit{us}, z) = \mathit{auxF}\ \mathit{ts} \\
& \qquad \mathit{auxF} :: \mathit{Forest\ Int} \to (a \to \mathit{Forest}\ a, \mathit{Int}) \quad \text{-- non-empty forest} \\
& \qquad \mathit{auxF}\ \mathit{ts} = (\lambda y \to \mathit{map}\ (\$ y)\ \mathit{us}, \mathit{minimum}\ \mathit{ys}) \\
& \qquad\qquad \mathbf{where}\ (\mathit{us}, \mathit{ys}) = \mathit{unzip}\ [\,\mathit{auxT}\ t \mid t \leftarrow \mathit{ts}\,]
\end{aligned}
$$

(the **let** need not be a **letrec**).

3. Fusing traversals

For traversal bodies $f :: A \to F\ B$ and $g :: A \to F\ C$, hope that:

$$traverse\ f\ t \otimes traverse\ g\ t = fmap\ unzip\ (traverse\ (\lambda x \to f\ x \otimes g\ x)\ t)$$

Cannot hold in general, because *different interleavings* of effects.

Interleaving irrelevant for *commutative* F. But that's very restrictive.

Also irrelevant if f-effects *commute with* g-effects, even for non-commutative F:

$$f\ x \otimes g\ y = fmap\ twist\ (g\ y \otimes f\ x)$$

In particular, whenever f-effects and g-effects occur in *distinct phases* of a two-phase computation: "do X now; do Y later" vs "do Y later; do X now".

Day convolution

$$\mathbf{data}\ \mathit{Day}\ f\ g\ a\ \mathbf{where}$$
$$\quad \mathit{Day} :: ((a, b) \to c) \to f\ a \to g\ b \to \mathit{Day}\ f\ g\ c$$

- $\mathit{Day}\ f\ xs\ ys$ with $xs :: F\ A$, $ys :: G\ B$ represents a *two-phase* computation
- subcomputation xs in phase one, generating effects in F
- subcomputation ys in phase two, generating effects in G
- package up with a function to combine the results ("*co-Yoneda trick*")
- $\mathit{Day}\ F\ G$ is applicative when F, G are

Injecting and projecting

Two ways to inject a computation, one for each phase:

$$\mathit{phase1} :: (\mathit{Applicative}\ f, \mathit{Applicative}\ g) \Rightarrow f\ a \to \mathit{Day}\ f\ g\ a$$
$$\mathit{phase1}\ \mathit{xs} = \mathit{Day}\ \mathit{unitr}\ \mathit{xs}\ \mathit{unit}$$

$$\mathit{phase2} :: (\mathit{Applicative}\ f, \mathit{Applicative}\ g) \Rightarrow g\ a \to \mathit{Day}\ f\ g\ a$$
$$\mathit{phase2}\ \mathit{xs} = \mathit{Day}\ \mathit{unitl}\ \mathit{unit}\ \mathit{xs}$$

Computations in different phases commute:

$$\mathit{phase1}\ \mathit{xs} \otimes \mathit{phase2}\ \mathit{ys} = \mathit{fmap}\ \mathit{twist}\ (\mathit{phase2}\ \mathit{ys} \otimes \mathit{phase1}\ \mathit{xs})$$

Collapse two phases into one, if they share the same class of effects:

$$\mathit{runDay} :: \mathit{Applicative}\ f \Rightarrow \mathit{Day}\ f\ f\ a \to f\ a$$
$$\mathit{runDay}\ (\mathit{Day}\ f\ \mathit{xs}\ \mathit{ys}) = \mathit{fmap}\ f\ (\mathit{xs} \otimes \mathit{ys})$$

Greeting in pieces

For example, we can send a two-part greeting in separate phases:

```
⟫ runDay (phase1 (putStr "Hello ") ∗⟩ phase2 (putStr "World"))
Hello World
```

It doesn't matter if we specify those two phases in the opposite order:

```
⟫ runDay (phase2 (putStr "World") ∗⟩ phase1 (putStr "Hello "))
Hello World
```

We can even interleave the specification of fragments from different phases:

```
⟫ runDay (phase1 (putStr "Hel") ∗⟩
              phase2 (putStr "World") ∗⟩
              phase1 (putStr "lo "))
Hello World
```

4. Repmin in two phases

Core of repmin:

$$\mathit{repminAux} :: \mathit{Tree}\ \mathit{Int} \rightarrow \mathit{Day}\ (\mathit{Writer}\ (\mathit{Min}\ \mathit{Int}))\ (\mathit{Reader}\ (\mathit{Min}\ \mathit{Int}))\ (\mathit{Tree}\ \mathit{Int})$$

Ask me later about:

- *Writer* and *Reader* monads
- *Min* monoid
- each phase of *repmin* is an instance of *traverse*
- the traversals *fuse*

Repmin, RSB-style

Extract the writer and reader components *in parallel*:

$$\mathit{parWR} :: \mathit{Day}\ (\mathit{Writer}\ s)\ (\mathit{Reader}\ s)\ a \to a$$
$$\begin{aligned}\mathit{parWR}\ (\mathit{Day}\ f\ \mathit{xs}\ \mathit{ys}) = {} & \mathbf{let}\ ((x, s), y) = (\mathit{runWriter}\ \mathit{xs}, \mathit{runReader}\ \mathit{ys}\ s) \\ & \mathbf{in}\ f\ (x, y)\end{aligned}$$

Circular, so **let** must have **letrec** semantics. In particular,

$$\mathit{repminWR}_{\mathrm{RSB}} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Tree}\ \mathit{Int}$$
$$\mathit{repminWR}_{\mathrm{RSB}}\ t = \mathit{parWR}\ (\mathit{repminAux}\ t)$$

is Bird's circular, lazy solution to the repmin problem.

Repmin, ADP-style

Conversely, extract writer then reader components *sequentially*:

$$
\begin{aligned}
&\mathit{seqWR} :: \mathit{Day}\ (\mathit{Writer}\ s)\ (\mathit{Reader}\ s)\ a \to a \\
&\mathit{seqWR}\ (\mathit{Day}\ f\ \mathit{xs}\ \mathit{ys}) = \mathbf{let}\ (x, s) = \mathit{runWriter}\ \mathit{xs} \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ y \qquad = \mathit{runReader}\ \mathit{ys}\ s \\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ f\ (x, y)
\end{aligned}
$$

Now no circularity, so plain non-recursive **let** suffices. In particular,

$$
\begin{aligned}
&\mathit{repminWR}_{\mathrm{ADP}} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Tree}\ \mathit{Int} \\
&\mathit{repminWR}_{\mathrm{ADP}}\ t = \mathit{seqWR}\ (\mathit{repminAux}\ t)
\end{aligned}
$$

is Pettorossi's non-circular, higher-order solution to the repmin problem.

Lazily, clearly $\mathit{parWR} = \mathit{seqWR}$. Hence also $\mathit{repminWR}_{\mathrm{RSB}} = \mathit{repminWR}_{\mathrm{ADP}}$.

5. Multiple phases

Generalize from two to *multiple* phases:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Phases}\ f\ a\ \mathbf{where}\\
&\quad \mathit{Pure} :: a \to \mathit{Phases}\ f\ a\\
&\quad \mathit{Link} :: ((a, b) \to c) \to f\ a \to \mathit{Phases}\ f\ b \to \mathit{Phases}\ f\ c
\end{aligned}$$

- *Pure* produces a chain with no effectful phases
- *Link* adds one more effectful phase to the chain
- *homogeneous* iteration of Day convolution
- cf lists as homogeneous iteration of pairing
- single initial value; each link adds combining function, collection of values
- eg $\mathit{Link}\ f\ xs\ (\mathit{Link}\ g\ ys\ (\mathit{Pure}\ z)) :: \mathit{Phases}\ F\ E$ where

$$z :: A, ys :: F\ B, g :: (B, A) \to C, xs :: F\ D, f :: (D, C) \to E$$

Free applicatives

Phases F is the *free applicative* on functor *F*, using *concatenation*:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ (\mathit{Phases}\ f)\ \mathbf{where} \\
\quad \mathit{unit} = \mathit{Pure}\ () \\
\quad \mathit{Pure}\ x \otimes \mathit{ys} = \mathit{fmap}\ (x,)\ \mathit{ys} \\
\quad \mathit{Link}\ f\ \mathit{xs}\ \mathit{ys} \otimes \mathit{zs} = \mathit{Link}\ (\lambda(x,(y,z)) \rightarrow (f\ (x,y),z))\ \mathit{xs}\ (\mathit{ys} \otimes \mathit{zs})
\end{array}
$$

But concatenation is not what we want.

When *F* is *itself* applicative and not just a functor, we can *(long) zip*:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Applicative}\ f \Rightarrow \mathit{Applicative}\ (\mathit{Phases}\ f)\ \mathbf{where} \quad \text{-- different instance!} \\
\quad \mathit{unit} = \mathit{Pure}\ () \\
\quad \mathit{Pure}\ x \otimes \mathit{ys} = \mathit{fmap}\ (x,)\ \mathit{ys} \\
\quad \mathit{xs} \otimes \mathit{Pure}\ y = \mathit{fmap}\ (,y)\ \mathit{xs} \\
\quad \mathit{Link}\ f\ \mathit{xs}\ \mathit{ys} \otimes \mathit{Link}\ g\ \mathit{zs}\ \mathit{ws} = \mathit{Link}\ (\mathit{cross}\ f\ g \circ \mathit{exch}4)\ (\mathit{xs} \otimes \mathit{zs})\ (\mathit{ys} \otimes \mathit{ws})
\end{array}
$$

Two phases, more or less

By design, *Phases f* generalizes *Day f f*. Hence injection:

$$inject :: Applicative\ f \Rightarrow Day\ f\ f\ a \rightarrow Phases\ f\ a$$
$$inject\ (Day\ f\ xs\ ys) = Link\ f\ xs\ (Link\ unitr\ ys\ (Pure\ ()))$$

Analogous to *phase1* and *phase2*, embed into an arbitrary phase:

$$phase :: Applicative\ f \Rightarrow Int \rightarrow f\ a \rightarrow Phases\ f\ a$$
$$phase\ 1 = now$$
$$phase\ i\ = later \circ phase\ (i - 1)$$

where

$$now :: Applicative\ f \Rightarrow f\ a \rightarrow Phases\ f\ a \qquad \text{-- embed at phase one}$$
$$now\ xs = Link\ unitr\ xs\ (Pure\ ())$$

$$later :: Applicative\ f \Rightarrow Phases\ f\ a \rightarrow Phases\ f\ a \qquad \text{-- shift everything one phase later}$$
$$later\ xs = Link\ unitl\ unit\ xs$$

Sorting leaves of a tree in one pass

$$\mathit{sortTree} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Tree}\ a$$
$$\mathit{sortTree}\ t = \mathit{evalState}\ (\mathit{runPhases}\ (\mathit{sortTreeAux}\ t))\ [\,]$$

$$\mathit{sortTreeAux} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Phases}\ (\mathit{State}\ [a])\ (\mathit{Tree}\ a)$$
$$\begin{array}{lll} \mathit{sortTreeAux}\ t = & \mathit{phase}\ 1\ (\mathit{traverse}\ \mathit{push}\ t)\ \ast\rangle & \text{-- } \mathit{push} :: a \to \mathit{State}\ [a]\ () \\ & \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort})\ \ast\rangle & \\ & \mathit{phase}\ 3\ (\mathit{traverse}\ (\lambda x \to \mathit{pop})\ t) & \text{-- } \mathit{pop} :: () \to \mathit{State}\ [a]\ a \end{array}$$

- commute phases, to bring the two traversals together
- traversal commutes with staging
- consecutive traversals in different phases fuse

$$\begin{array}{ll} \mathit{sortTreeAux}\ t = & \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort})\ \ast\rangle \\ & \mathit{traverse}\ (\lambda x \to \mathit{phase}\ 1\ (\mathit{push}\ x)\ \ast\rangle\ \mathit{phase}\ 3\ \mathit{pop})\ t \end{array}$$

Breadth-first traversal in stages

$$bft :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to f\ (Tree\ b)$$
$$bft\ f = runPhases \circ bftAux\ f$$

$$bftAux :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to Phases\ f\ (Tree\ b)$$
$$bftAux\ f\ (Node\ x\ ts) = fmap\ (uncurry\ Node)\ (now\ (f\ x) \otimes later\ (traverse\ (bftAux\ f)\ ts))$$

cf depth-first, obtained by deleting staging annotations:

$$dft :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to f\ (Tree\ b)$$
$$dft\ f\ (Node\ x\ ts) = fmap\ (uncurry\ Node)\ (f\ x \otimes traverse\ (dft\ f)\ ts)$$

In particular, bf relabelling, needing neither queues nor cyclicity/laziness:

$$bfl :: Tree\ a \to [\,b\,] \to Tree\ b$$
$$bfl\ t\ xs = evalState\ (bft\ (\lambda x \to pop)\ t)\ xs$$

6. Conclusion

- Day convolution: *natural monoidal structure* underlying applicative functors
- multi-stage computation as *iterated Day convolution*
- same datatype as free applicatives, but *different applicative instance*
- unifying Bird's and Pettorossi's repmin solutions
- breadth-first traversal without shape/contents or laziness/circularity

- joint work with Oisín Kidney, Tom Schrijvers, Nick Wu
- paper at MPC 2022 (LNCS 13544, doi `10.1007/978-3-031-16912-0_1`)
- dedicated to Richard Bird
- `http://www.cs.ox.ac.uk/jeremy.gibbons/`

Extra slides

7. Queues

$$
\begin{array}{l}
\mathit{bfq} :: \mathit{Tree}\ a \rightarrow [\,a\,] \\
\mathit{bfq}\ t = \mathit{bfqAux}\ [\,t\,]\ \mathbf{where} \\
\quad \begin{array}{ll}
\mathit{bfqAux}\ (\mathit{Node}\ x\ \mathit{ts} : q) & = x : \mathit{bfqAux}\ (q +\!\!+\ \mathit{ts}) \\
\mathit{bfqAux}\ [\,] & = [\,]
\end{array}
\end{array}
$$

Straightforward for *enumeration*, but what about *traversal*?

Anyway, it's *non-compositional*. (More of an unfold than a fold. . .)

Shape and contents

$$shape :: Tree\ a \to Tree\ ()$$
$$shape = fmap\ (const\ ())$$

and

$$levels :: Tree\ a \to [[a]]$$
$$levels\ (Node\ x\ ts) = [x] : levelsF\ ts$$

$$levelsF :: Forest\ a \to [[a]]$$
$$levelsF = foldr\ (lzw\ (+\!\!+))\ [\,] \circ map\ levels \quad \text{-- ``long zip with''}$$

so $bf = concat \circ levels$. But now compositional.

Relabelling

$$
\begin{aligned}
&\mathit{relabel} :: (\mathit{Tree}\ (), [[a]]) \to (\mathit{Tree}\ a, [[a]]) \quad \text{-- given appropriate list of lists\ldots} \\
&\mathit{relabel}\ (\mathit{Node}\ ()\ \mathit{ts}, (x : \mathit{xs}) : \mathit{xss}) = \mathbf{let}\ (\mathit{us}, \mathit{yss}) = \mathit{relabelF}\ (\mathit{ts}, \mathit{xss}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\mathit{Node}\ x\ \mathit{us}, \mathit{xs} : \mathit{yss})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{relabelF} :: (\mathit{Forest}\ (), [[a]]) \to (\mathit{Forest}\ a, [[a]]) \\
&\mathit{relabelF}\ ([\,], \mathit{xss}) \quad\ = ([\,], \mathit{xss}) \\
&\mathit{relabelF}\ (t : \mathit{ts}, \mathit{xss}) = \mathbf{let}\ (u, \mathit{yss}) \ \ = \mathit{relabel}\ (t, \mathit{xss}) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{us}, \mathit{zss}) = \mathit{relabelF}\ (\mathit{ts}, \mathit{yss}) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (u : \mathit{us}, \mathit{zss})
\end{aligned}
$$

— in some sense, the inverse of the split into shape and contents. So

$$
\begin{aligned}
&\mathit{bftSC} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to \mathit{Tree}\ a \to f\ (\mathit{Tree}\ b) \\
&\mathit{bftSC}\ f\ t = \mathit{fmap}\ (\mathit{combine}\ (\mathit{shape}\ t))\ (\mathit{traverse}\ (\mathit{traverse}\ f)\ (\mathit{levels}\ t)) \\
&\quad \mathbf{where}\ \mathit{combine}\ u\ \mathit{xss} = \mathit{fst}\ (\mathit{relabel}\ (u, \mathit{xss}))
\end{aligned}
$$

Circular programs

Need not have the contents conveniently partitioned.
Instead, partition it on the fly:

$$\mathit{bflabel} :: \mathit{Tree}\ () \to [\,a\,] \to \mathit{Tree}\ a$$
$$\mathit{bflabel}\ t\ \mathit{xs} = \mathbf{let}\ (u, \mathit{xss}) = \mathit{relabel}\ (t, \mathit{xs} : \mathit{xss})\ \mathbf{in}\ u$$

(Due to Geraint Jones.)

Note that this **let** must be a **letrec**; the program is *circular*.

Hence another definition of breadth-first traversal.

It's circular. So seems like it needs *laziness*?

But it's still a bit clunky to have to separate into shape and contents.

8. Repmin in two phases

Writer:

$$runWriter :: Writer\ w\ a \to (a, w)$$
$$tell :: Monoid\ w \Rightarrow Writer\ w\ ()$$

and reader:

$$runReader :: Reader\ r\ a \to (r \to a)$$
$$ask :: Reader\ r\ r$$

and minimum as a monoid over Int:

$$\begin{aligned} &Min &&:: Int \to Min\ Int \\ &getMin &&:: Min\ Int \to Int \end{aligned}$$

We work in the Day convolution $Day\ (Writer\ (Min\ Int))\ (Reader\ (Min\ Int))$.

Core of repmin

$$\mathit{repminAux} :: \mathit{Tree}\ \mathit{Int} \rightarrow \mathit{Day}\ (\mathit{Writer}\ (\mathit{Min}\ \mathit{Int}))\ (\mathit{Reader}\ (\mathit{Min}\ \mathit{Int}))\ (\mathit{Tree}\ \mathit{Int})$$
$$\mathit{repminAux}\ t = \mathit{phase1}\ (\mathit{minAux}\ t) \ast\rangle\ \mathit{phase2}\ (\mathit{replaceAux}\ t)$$

where

$$\mathit{minAux} :: \mathit{Tree}\ \mathit{Int} \rightarrow \mathit{Writer}\ (\mathit{Min}\ \mathit{Int})\ (\mathit{Tree}\ ())$$ -- write each element in turn
$$\mathit{minAux} = \mathit{traverse}\ (\lambda x \rightarrow \mathit{tellMin}\ x)$$
$$\mathit{tellMin} :: \mathit{Int} \rightarrow \mathit{WInt}\ ()$$
$$\mathit{tellMin}\ x = \mathit{tell}\ (\mathit{Min}\ x)$$

$$\mathit{replaceAux} :: \mathit{Tree}\ \mathit{Int} \rightarrow \mathit{Reader}\ (\mathit{Min}\ \mathit{Int})\ (\mathit{Tree}\ \mathit{Int})$$ -- replacement for each element
$$\mathit{replaceAux} = \mathit{traverse}\ (\lambda x \rightarrow \mathit{askMin})$$
$$\mathit{askMin} :: \mathit{RInt}\ \mathit{Int}$$
$$\mathit{askMin} = \mathit{fmap}\ \mathit{getMin}\ \mathit{ask}$$

Fusion

$$
\begin{array}{ll}
 & \mathit{repminAux}\ t \\
= & [\![\ \text{specification}\ ]\!] \\
 & \mathit{phase1}\ (\mathit{traverse}\ (\lambda x \rightarrow \mathit{tellMin}\ x)\ t) \ast\rangle\ \mathit{phase2}\ (\mathit{traverse}\ (\lambda x \rightarrow \mathit{askMin})\ t) \\
= & [\![\ \text{naturality in applicative functor}\ ]\!] \\
 & \mathit{traverse}\ (\lambda x \rightarrow \mathit{phase1}\ (\mathit{tellMin}\ x))\ t \ast\rangle\ \mathit{traverse}\ (\lambda x \rightarrow \mathit{phase2}\ \mathit{askMin})\ t \\
= & [\![\ \text{fusion of traversals}\ ]\!] \\
 & \mathit{traverse}\ (\lambda x \rightarrow \mathit{phase1}\ (\mathit{tellMin}\ x) \ast\rangle\ \mathit{phase2}\ \mathit{askMin})
\end{array}
$$

—a *one-pass* traversal, generating a *two-phase* computation for later execution.