Breadth-First Traversal Via Staging

Jeremy Gibbons[⋆][0000−0002−8426−9917],
Donnacha Oisín Kidney[†][0000−0003−4952−7359],
Tom Schrijvers[‡][0000−0001−8771−5559], and Nicolas Wu[†][0000−0002−4161−985X]

[⋆] University of Oxford
[†] Imperial College London
[‡] KU Leuven

Abstract. An effectful traversal of a data structure iterates over every element, in some predetermined order, collecting computational effects in the process. Depth-first effectful traversal of a tree is straightforward to define in a compositional way, since it precisely follows the shape of the data. What about breadth-first effectful traversal? An indirect route is to factorize the data structure into shape (a structure of units) and contents (a vector of elements, in breadth-first order), traverse the vector, then rebuild the data structure with new contents. We show that this can instead be done directly, using staging. That staging is captured using a construction related to free applicative functors. Moreover, the staged traversals lend themselves well to fusion; we prove a novel fusion rule for effectful traversals, and use it in another solution to Bird's 'repmin' problem.

Keywords: Traversal · staging · applicative functor · fusion.

1 Introduction

This paper is about effectful traversals of data structures, in which the effects are modelled as applicative functors. This encompasses monadic effects (since every monad is also an applicative functor), but also generalizes to include monoidal aggregation, among other possibilities.

Applicative traversals capture the essence of the Iterator design pattern [8]. Informally, an applicative traversal processes a container data structure in a predetermined order, visiting each element precisely once, collecting computational effects as it goes (for example, in the state monad), and replacing each element with a new one while preserving the shape of the data structure.

For any polynomial datatype, it is completely straightforward to define an applicative traversal that follows the structure of the data. Indeed, the definition is so straightforward that it can be automated, or expressed as a single datatype-generic program. That straightforward traversal will be depth-first, completely traversing one child before moving on to the next. We give an example on lists in Section 2.1, and one on trees in Section 2.2.

But there are other possible traversal orders. In particular, what about breadth-first traversal of a tree? This is more awkward, because it goes against

the grain of the tree, so to speak. One approach is to factorize the tree into shape (a tree of unit values) and contents (by breadth-first enumeration to a sequence of elements). One can then traverse the contents in isolation from the shape, then reassemble the unchanged shape and new contents into a new tree.

Executing breadth-first traversal in multiple passes in this way is a bit clumsy and inefficient. Can we do the same thing in a single pass? We can! The key idea is to construct a *multi-stage computation*, with one phase per level of the tree. This multi-stage computation can be assembled in a single pass over the tree; then the phases of this computation are run one after the other. Although breadth-first traversal itself is not compositional (one cannot construct the breadth-first traversal of a tree from the traversals of its children), the multi-staged computation is compositional, because it is conveniently broken up into layers.

We present a novel approach to staging in terms of applicative functors. In particular, we use a data representation due recently to Kidney and Wu [11], isomorphic to the *free applicative functor* on a given base functor but using a different applicative instance than that of the free applicative. Informally, we need to 'zip' together phases rather than concatenating them in order to combine computations.

We show that this approach also provides alternative solutions to other problems involving transforming multiple passes into one, such as Bird's 'repmin' problem [2] and some other problems using circular definitions, but avoiding the need for laziness, such as breadth-first relabelling of a tree [10].

The remainder of the paper is structured as follows. Section 2 relates background material on applicative functors and applicative traversal. Section 3 presents the indirect breadth-first approach via factorization into shape and contents. Section 4 introduces two-phase computation, using the Day convolution of two functors, and discusses how to fuse traversals at multiple phases into a single multi-stage traversal. Section 5 generalizes this to arbitrarily many phases, by iterating Day convolution. Section 6 returns to breadth-first traversal and breadth-first relabelling, but now expressed compositionally in multi-staged terms and without needing laziness. Section 7 concludes. The key result about fusion of traversals used in Section 4 is proved in Appendix A.

We use Haskell as a vehicle of expression, but almost always read it in terms of sets and total functions rather than CPOs with strictness considerations. (The only divergence from that position is in discussing Bird's essentially lazy repmin solution, but our solution avoids this essential laziness.) The code in the paper is slightly simplified for presentation purposes, but the full details are available online [7]. We follow Haskell in using lowercase letters for polymorphic type parameters; but as a presentation convention, in prose though not in code, we use uppercase letters when discussing specific types. For example, *map* has the polymorphic type $(a \to b) \to [a] \to [b]$ (the type parameters a, b are implicitly universally quantified), but if applied to a specific function $f :: A \to B$ then we say that $map\ f$ has type $[A] \to [B]$.

2 Applicative functors

We focus on effects modelled as *applicative functors*, a slightly more general perspective than the more familiar monads. But we use the simpler categorically-inspired presentation rather than the more program-oriented one in the Haskell libraries:

$$\begin{aligned}&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \textbf{ where}\\ &\quad \mathit{unit} :: f\ ()\\ &\quad (\otimes)\ :: f\ a \to f\ b \to f\ (a, b)\end{aligned}$$

Thus, there is a distinguished collection *unit* of unit values, and one can combine two collections into a collection of pairs. Categorically, an applicative functor is "strong lax-monoidal functor". Strength comes for free in a higher-order language like Haskell; and lax monoidality amounts to left- and right-unit and associativity properties:

$$\begin{aligned}&\mathit{fmap}\ \mathit{unitl}\ (\mathit{unit} \otimes \mathit{ys}) &&= \mathit{ys}\\ &\mathit{fmap}\ \mathit{unitr}\ (\mathit{xs} \otimes \mathit{unit}) &&= \mathit{xs}\\ &\mathit{fmap}\ \mathit{assoc}\ (\mathit{xs} \otimes (\mathit{ys} \otimes \mathit{zs})) &&= (\mathit{xs} \otimes \mathit{ys}) \otimes \mathit{zs}\end{aligned}$$

that hold not on the nose, but only up to some conversions; we write

$$\begin{aligned}&\mathit{unitl}\ :: ((), a) \to a && \mathit{unitl}^{-1}\ :: a \to ((), a)\\ &\mathit{unitr} :: (a, ()) \to a && \mathit{unitr}^{-1} :: a \to (a, ())\\ &\mathit{assoc} :: (a, (b, c)) \to ((a, b), c) && \mathit{assoc}^{-1} :: ((a, b), c) \to (a, (b, c))\end{aligned}$$

for the obvious isomorphisms witnessing the (so-called 'strong') monoidal structure of the product.

While we are on the subject of isomorphisms for pairs, we will also make use of two involutions involving commutativity:

$$\begin{aligned}&\mathit{twist}\ :: (a, b) \to (b, a)\\ &\mathit{exch4} :: ((a, b), (c, d)) \to ((a, c), (b, d))\end{aligned}$$

Our categorical and the usual Haskell presentations of applicative functors are equivalent, and the interfaces interdefinable. In particular, we will still find the Haskell interface convenient for programming, and it can be implemented as follows:

$$\begin{aligned}&\mathit{pure} :: \mathit{Applicative}\ f \Rightarrow a \to f\ a\\ &\mathit{pure}\ x = \mathit{fmap}\ (\mathit{const}\ x)\ \mathit{unit}\\ &(<\!\!*\!\!>) :: \mathit{Applicative}\ f \Rightarrow f\ (a \to b) \to f\ a \to f\ b \quad \text{-- left-associative}\\ &\mathit{fs} <\!\!*\!\!> \mathit{xs} = \mathit{fmap}\ (\lambda(f, x) \to f\ x)\ (\mathit{fs} \otimes \mathit{xs})\\ &(*\!\!>) :: \mathit{Applicative}\ f \Rightarrow f\ a \to f\ b \to f\ b \quad \text{-- left-associative}\\ &\mathit{xs} *\!\!> \mathit{ys} = \mathit{fmap}\ \mathit{snd}\ (\mathit{xs} \otimes \mathit{ys})\end{aligned}$$

This situation is analogous to the distinction between the presentation of monads in terms of bind ($\gg=$), which is more convenient for programming, and in terms of multiplication *join*, which is more categorically perspicuous.

Every monad is an applicative functor, with the following implementation:

```
instance Monad m ⇒ Applicative m where
  unit     = return ()
  xs ⊗ ys = do {x ← xs; y ← ys; return (x, y)}
```

—that is, $(\otimes)$ can be seen as a form of sequencing. An illuminating applicative functor instance that does not arise from a monad is that of colists $[\,]^\omega$—that is, finite and infinite lists together—under zipping:

```
instance Applicative []^ω where
  unit = repeat ()
  (⊗)  = zip
```

Another is the constant applicative functor *Const A* for monoid A:

```
data Const a b = Const {getConst :: a}

instance Monoid a ⇒ Applicative (Const a) where
  unit   = Const mempty
  x ⊗ y = Const (mappend (getConst x) (getConst y))
```

2.1 Applicative traversal

Applicative traversal is "the essence of the Iterator design pattern" [8], capturing computations that iterate over a data structure, in a predetermined order, processing each element in turn and collecting effects as they go:

```
class Functor t ⇒ Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
```

For example, left-to-right traversal of (finite) lists just follows the shape of the list datatype:

```
instance Traversable [] where
  traverse f []       = pure []
  traverse f (x : xs) = pure (:) <*> f x <*> traverse f xs
```

Well-behaved traversals are those satisfying three axioms of *naturality*, *linearity*, and *unitarity* [9]. These essentially say that *traverse* is 'natural' in the applicative functor, and respects the compositional structure of applicative functors (the identity functor is applicative, and composition of applicative functors is again applicative). This implies, among other consequences, that a well-behaved traversal preserves the shape of the data structure it traverses, and visits every element precisely once.

Formally, an *applicative morphism* $\phi : F \to G$ between applicative functors F and G is a polymorphic function $\phi :: \forall a \,.\, F\ a \to G\ a$ that respects the applicative structure. To be explicit, we write $(\otimes_F)$, $unit_F$ for the applicative operations for applicative functor F. Then for ϕ to be an applicative morphism, it must satisfy:

$$\begin{aligned} \phi\ unit_F &= unit_G \\ \phi\ (xs \otimes_F ys) &= \phi\ xs \otimes_G \phi\ ys \end{aligned}$$

for $xs :: F\ A$ and $ys :: F\ B$. Then the *naturality* axiom of a well-behaved traversal states that it respects applicative morphisms: for $f :: A \to F\ B$ and applicative morphism $\phi : F \to G$,

$$traverse\ (\phi \circ f) = \phi \circ traverse\ f$$

Applicative functors are closed under functor composition: the identity functor I is applicative, and if F and G are applicative then so is their composition $F \circ G$. The other two conditions on well-behaved traversals are that they should respect this compositional structure. To be explicit again, we subscript generic functions with the applicative functor. The *unitarity* axiom states that traversal with the identity function $id :: A \to I\ A$ is itself the identity:

$$traverse_I\ id = id$$

Together with the free theorem of the type of *traverse*, we get more generally that traversal in the identity applicative functor—that is, traversal with a pure function $f :: A \to I\ B$—is just a map:

$$traverse_I\ f = fmap\ f$$

The *linearity* axiom states that traversal in the composition of applicative functors is a composition of traversals. That is, given applicative functors F and G, and traversal bodies $f :: A \to F\ B$ and $g :: B \to G\ C$, write $(\mathbin{<\!\circ\!>})$ for the obvious composition:

$$g \mathbin{<\!\circ\!>} f = fmap_F\ g \circ f :: A \to F\ (G\ C)$$

Then we have:

$$traverse_{F \circ G}\ (g \mathbin{<\!\circ\!>} f) = traverse_G\ g \mathbin{<\!\circ\!>} traverse_F\ f$$

Here we have equated $I\ A$ with A, and $(F \circ G)\ A$ with $F\ (G\ A)$. This isn't possible in Haskell; we would have to introduce *Identity* and *Compose* datatype wrappers, with corresponding injection and projection isomorphisms, and include those isomorphisms in the statements of the properties.

One more important result about traversals, a theorem rather than an axiom: the Representation Theorem for traversals [1] states that well-behaved traversals over an arbitrary traversable datatype are fully characterised by the corresponding list-based traversals over their contents, and so results about traversals in general follow from results about traversals over lists in particular. We will cover this theorem when we need it, in Section 4.

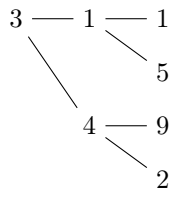
2.2 Trees

We will use the following datatype of trees throughout the paper:

```
data Tree a    = Node a (Forest a)
type Forest a = [Tree a]
```

For example, here is a tree of integers:

```
t :: Tree Int
t = Node 3 [Node 1 [Node 1 []
                   , Node 5 []]
           , Node 4 [Node 9 []
                   , Node 2 []]]
```



Depth-first traversal of a tree is easily captured as a *Traversable* instance:

```
instance Traversable Tree where
  traverse f (Node x ts) = pure Node <*> f x <*> traverseF f ts
    where traverseF f = traverse (traverse f)
```

This again follows the shape of the *Tree* datatype, which is mutually recursive with the *Forest* type. We will encounter several times this pattern of mutual recursion between the function on trees and the corresponding function on forests; here is another example, for computing the depths of a tree and a forest:

```
depth :: Tree a → Int
depth (Node x ts) = 1 + depthF ts

depthF :: Forest a → Int
depthF [] = 0
depthF ts = maximum (map depth ts)
```

The definition of traversal over trees is actually very similar in principle to the traversal of lists, as defined in Section 2.1. In fact, the outermost *traverse* in the definition of *traverseF* is that list instance of traversal. Moreover, each definition can in principle be derived automatically from the corresponding datatype definition.

But what about breadth-first traversal? It is not obvious how to do that structurally, as we have done for lists and for depth-first traversal of trees; in particular, it does not follow directly from the structure of the datatype definition.

3 Shape, contents, relabelling

An indirect approach to breadth-first traversal can be made by factoring a tree into its shape and contents [5]; here we see how.

Let us first consider breadth-first enumeration, returning just the list of elements in the tree. This is not compositional, because one cannot compute the breadth-first enumeration of a tree from the enumerations of its children. But the related "level-order enumeration", giving a list of lists, one list per level, is compositional:

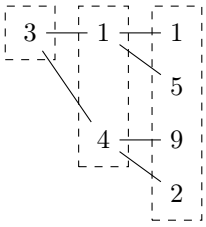
$$
\begin{aligned}
&\mathit{levels} :: \mathit{Tree}\ a \to [[a]] \\
&\mathit{levels}\ (\mathit{Node}\ x\ ts) = [x] : \mathit{levelsF}\ ts \\
&\\
&\mathit{levelsF} :: \mathit{Forest}\ a \to [[a]] \\
&\mathit{levelsF} = \mathit{foldr}\ (\mathit{lzw}\ (+\!\!+))\ [\,] \circ \mathit{map}\ \mathit{levels}
\end{aligned}
$$

Here, *lzw* (for "long zip with") is similar to *zipWith*, but returns a list as long as its longer argument [6]:

$$
\begin{aligned}
&\mathit{lzw} :: (a \to a \to a) \to [a] \to [a] \to [a] \\
&\mathit{lzw}\ f\ (x : xs)\ (y : ys) = f\ x\ y : \mathit{lzw}\ f\ xs\ ys \\
&\mathit{lzw}\ f\ [\,]\quad\ \ ys\quad\ \ = ys \\
&\mathit{lzw}\ f\ xs\quad\ \ [\,]\quad\ \ = xs
\end{aligned}
$$

For example, with t as in Section 2, we have:

$$
\begin{aligned}
\mathit{levels}\ t = [\ \ & [3], \\
& [1,4], \\
& [1,5,9,2] \\
]&
\end{aligned}
$$



Given level-order enumeration, breadth-first enumeration is obtained by concatenation:

$$
\begin{aligned}
&\mathit{bf} :: \mathit{Tree}\ a \to [a] \\
&\mathit{bf} = \mathit{concat} \circ \mathit{levels}
\end{aligned}
$$

so that $\mathit{bf}\ t = [3,1,4,1,5,9,2]$.

Now, enumeration is invertible, in the sense that one can reconstruct the tree given its shape (a tree of units) and its level-order enumeration (a list of elements). One way to define the inverse process is to pass the level-order enumeration around the tree, incrementally snipping bits off it. Here is a mutually recursive pair of functions to relabel a tree with a given list of lists, returning also the unused tails from the list of lists:

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

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

Assuming that the given list of lists is 'big enough'—that is, each list has enough elements for that level of the tree—the result is well-defined. Then *relabel* is determined by the equivalence

$$\begin{aligned}
&\mathit{relabel}\ (t, \mathit{xss}) = (u, \mathit{yss}) \iff\\
&\quad \mathit{shape}\ u = \mathit{shape}\ t \wedge \mathit{length}\ \mathit{yss} = \mathit{length}\ \mathit{xss} \wedge \mathit{lzw}\ (+\!\!+)\ (\mathit{levels}\ u)\ \mathit{yss} = \mathit{xss}
\end{aligned}$$

where *shape* discards the elements of a tree:

$$\begin{aligned}
&\mathit{shape} :: \mathit{Tree}\ a \to \mathit{Tree}\ ()\\
&\mathit{shape} = \mathit{fmap}\ (\mathit{const}\ ())
\end{aligned}$$

In particular, if the given list of lists is the level-order enumeration of the tree, and so is exactly the right size, then *yss* will have no remaining elements, consisting entirely of empty levels:

$$\mathit{relabel}\ (\mathit{shape}\ t, \mathit{levels}\ t) = (t, \mathit{replicate}\ (\mathit{depth}\ t)\ [\,])$$

So we can factor a tree into its shape and contents, and reconstruct the tree from such data:

$$\begin{aligned}
&\mathit{split} :: \mathit{Tree}\ a \to (\mathit{Tree}\ (), [[a]])\\
&\mathit{split}\ t = (\mathit{shape}\ t, \mathit{levels}\ t)\\
&\mathit{combine} :: \mathit{Tree}\ () \to [[a]] \to \mathit{Tree}\ a\\
&\mathit{combine}\ u\ \mathit{xss} = \mathit{fst}\ (\mathit{relabel}\ (u, \mathit{xss}))
\end{aligned}$$

This lets us traverse a tree in breadth-first order, by performing the traversal on the contents in isolation. We separate the tree into shape and contents, perform a list-based traversal, and reconstruct the tree:

$$\begin{aligned}
&\mathit{bftSC} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to \mathit{Tree}\ a \to f\ (\mathit{Tree}\ b)\\
&\mathit{bftSC}\ f\ t = \mathit{pure}\ (\mathit{combine}\ (\mathit{shape}\ t)) <\!\!*\!\!> \mathit{traverse}\ (\mathit{traverse}\ f)\ (\mathit{levels}\ t)
\end{aligned}$$

Incidentally, it is not necessary to have the enumeration of the tree conveniently partitioned into levels; one can also relabel the tree just from its breadth-first enumeration. The trick is to construct the appropriate partition of the breadth-first enumeration into levels. There is a clever cyclic program due to Geraint Jones [10] to do this at the same time as relabelling:

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

Note that *xss* is defined cyclically, so it is crucial that this **let** has **letrec** semantics—that is, the variables bound on the left of the equals sign are in

scope on the right as well as in the body. Informally, the output leftovers *xss* on one level also form the input elements to be used for relabelling all the lower levels. Given this definition, we have

$$bflabel\ (shape\ t)\ (bf\ t) = t$$

for any t. We can use this approach instead in the definition of breadth-first traversal:

$$\begin{aligned}&bftL :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to f\ (Tree\ b)\\&bftL\ f\ t = pure\ (bflabel\ (shape\ t)) <\!\!*\!\!> traverse\ f\ (bf\ t)\end{aligned}$$

However, both these implementations of breadth-first traversal are clunky and inefficient, because of having to factor into shape and contents. Also, breadth-first relabelling given only the enumeration is tricky: the program is cyclic, so doing it in a single pass seems to require laziness [12]. We will show that this impression is false: there are perfectly good ways of presenting it that make no use of laziness.

4 Fusing traversals via staged computation

The circular function for breadth-first relabelling is tricky because it fuses two passes (splitting the input stream into levels, then applying these levels across the tree) into one, entangling the two together. There is a whole class of programs of this form: circular definitions, fusing together multiple passes into one. The classic example is Bird's 'repmin' problem [2]. Studying the structure of repmin will help us see what is going on with breadth-first relabelling.

4.1 The repmin problem

The repmin problem is to replace every element of a tree with the minimum element in that tree:

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

but to do so in one pass rather than two. Bird's technique is to define a composite function that computes the minimum in a tree and replaces all elements with a given element all in one go, and to feed the minimum output back in as the replacement input:

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

Note that the m in $repmin_{\mathrm{RSB}}$ is defined cyclically, as with Jones's breadth-first relabelling, so that **let** must have **letrec** semantics.

But Bird's circular program making essential use of **letrec** semantics and laziness is not the only way to solve the repmin problem. Pettorossi and Skowron [13,14] show how to get the same results using only higher-order functions, and needing only call-by-value evaluation. Pettorossi's solution is as follows:

$$
\begin{array}{l}
repmin_{\mathrm{ADP}} :: Tree\ Int \to Tree\ Int \\
repmin_{\mathrm{ADP}}\ t = \mathbf{let}\ (u, m) = auxT\ t\ \mathbf{in}\ u\ m \\
\quad \mathbf{where} \\
\qquad auxT :: Tree\ Int \to (a \to Tree\ a, Int) \\
\qquad auxT\ (Node\ x\ [\,]) = (\lambda y \to Node\ y\ [\,], x) \\
\qquad auxT\ (Node\ x\ ts) = (\lambda y \to Node\ y\ (us\ y), min\ x\ z) \\
\qquad\qquad\qquad\qquad \mathbf{where}\ (us, z) = auxF\ ts \\
\qquad auxF :: Forest\ Int \to (a \to Forest\ a, Int) \quad \text{-- non-empty forest} \\
\qquad auxF\ ts = (\lambda y \to map\ (\$y)\ us, minimum\ ys) \\
\qquad\qquad\qquad \mathbf{where}\ (us, ys) = unzip\ [auxT\ t \mid t \leftarrow ts]
\end{array}
$$

Where Bird's $auxT$ takes a replacement value as an input, and returns an updated tree, Pettorossi's $auxT$ takes no such input, and returns instead a function from replacement value to updated tree. Where Bird's main function has a circular definition, feeding the output minimum back in as an input and returning only the updated tree, Pettorossi's main function is not circular (the **let** need not be treated as a **letrec**), and the output function is applied to the output minimum. Danvy et al. [4] say more about the relationship between these two approaches.

Pettorossi's approach neatly makes explicit the data dependencies, and therefore the sense in which the components of the solution are compositional, in a way that Bird's approach does not. Specifically, it is explicit in Pettorossi's program that the output minimum does not depend on the input replacement value—because there is no input replacement value upon which to depend. In contrast,

the input replacement value y in Bird's $auxT$ is in scope for the definition of the output minimum; it takes some kind of program analysis to confirm that x, $min\ x\ z$, and $minimum\ ys$ are all independent of y, as required in order for the **letrec** to be productive.

4.2 Fusing traversals

The crucial ingredient in fitting this development into our framework is an appropriate fusion rule for traversals. Informally, two consecutive traversals over the same data structure with the same class of effects can be fused into a single traversal with a composite body. Formally, with applicative functor F, traversable data structure $t :: T\ A$, and traversal bodies $f :: A \to F\ B$ and $g :: A \to F\ C$, we have:

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

where *unzip* separates a structure of pairs into a pair of structures:

$$unzip :: Functor\ t \Rightarrow t\ (a, b) \to (t\ a, t\ b)$$
$$unzip\ t = (fmap\ fst\ t, fmap\ snd\ t)$$

We will only use this rule in the special case in which f returns unit (that is, $B = ()$), or returns values that we discard. Then no unzipping is required:

$$traverse\ f\ t \mathbin{*\!\!>} traverse\ g\ t = traverse\ (\lambda x \to f\ x \mathbin{*\!\!>} g\ x)\ t$$

But neither fusion result can hold in general, because of the order of effects: on the left, all the f-effects precede any of the g-effects, and on the right they are interleaved.

The interleaving would be irrelevant if F were *commutative*: that is, if

$$xs \otimes ys = fmap\ twist\ (ys \otimes xs)$$

for all $xs :: F\ A, ys :: F\ B$. But that condition is quite restrictive, ruling out in particular anything stateful. However, the interleaving is still irrelevant even when F is not commutative, provided more specifically that the f-effects commute with g-effects, in the sense that

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

for all x, y. We can prove this fusion rule using the Representation Theorem for applicative traversals [1]; the proof is given in Appendix A.

In particular, whenever f and g specify effects that occur in distinct phases of a two-phase computation, they will commute: it doesn't matter whether you say "do X now and Y later" or "do Y later and X now", because either way X is enacted before Y. So let us consider two-phase computations.

4.3 Day convolution

The Day convolution [15] $Day\ F\ G$ of two functors F, G is given by:

$$\begin{array}{l}\textbf{data}\ Day\ f\ g\ a\ \textbf{where}\\ \quad Day :: ((a,b) \to c) \to f\ a \to g\ b \to Day\ f\ g\ c\end{array}$$

Thus, $Day\ f\ xs\ ys$ with $xs :: F\ A, ys :: G\ B$ represents a two-phase computation, with subcomputation xs happening in phase one generating effects in F, and ys in phase two generating effects in G. It is convenient to package this pair up with a function $f :: (A, B) \to C$ to combine the results from the two phases. This packaging is known as the "co-Yoneda trick", and it straightforwardly turns $Day\ F\ G$ into a functor:

$$\begin{array}{l}\textbf{instance}\ Functor\ (Day\ f\ g)\ \textbf{where}\\ \quad fmap\ g\ (Day\ f\ xs\ ys) = Day\ (g \circ f)\ xs\ ys\end{array}$$

Moreover, $Day\ F\ G$ is applicative when F, G are applicative, with pointwise combination:

$$\begin{array}{l}\textbf{instance}\ (Applicative\ f, Applicative\ g) \Rightarrow Applicative\ (Day\ f\ g)\ \textbf{where}\\ \quad unit \qquad\qquad\qquad\qquad = Day\ unitr\ unit\ unit\\ \quad Day\ f\ xs\ ys \otimes Day\ g\ zs\ ws = Day\ (cross\ f\ g \circ exch4)\ (xs \otimes zs)\ (ys \otimes ws)\end{array}$$

where

$$\begin{array}{l}cross :: (a \to b) \to (c \to d) \to (a, c) \to (b, d)\\ cross\ f\ g\ (x, y) = (f\ x, g\ y)\end{array}$$

And there are two ways to inject a computation, one for each phase:

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

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

Crucially for us, computations in different phases commute:

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

When the two phases share a class of effects, we can combine the two phases, running one after the other and post-processing the results:

$$\begin{array}{l}runDay :: Applicative\ f \Rightarrow Day\ f\ f\ a \to f\ a\\ runDay\ (Day\ f\ xs\ ys) = fmap\ f\ (xs \otimes ys)\end{array}$$

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
```

4.4 Repmin in two phases

Returning now to the repmin problem, we have first to formulate it as an effectful computation. We could use the state monad, writing minimum values to the state in the first phase then reading the replacement value from the state in the second. But those two phases use the state in different ways: computing the minimum writes to the state without reading from it, and replacing tree elements reads from the state without writing to it. So a more precise expression of the two-phase solution would use two different classes of effect, writing and reading.

We therefore use the *Writer* and *Reader* monads respectively. For a monoid W, the writer monad $Writer\ W$ is essentially pairing with the written value W:

$$runWriter :: Writer\ w\ a \to (a, w)$$

and provides an operation

$$tell :: Monoid\ w \Rightarrow Writer\ w\ ()$$

to 'write' a value. For arbitrary type R, the reader monad $Reader\ R$ is essentially functions from the read value R:

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

and provides an operation

$$ask :: Reader\ r\ r$$

to 'read' a value. We will also use the wrapper type *Min* to construct the monoid *Min Int*, with minimum as the binary operator and the least *Int* value as unit:

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

Therefore we work in $\mathit{Day}\ (\mathit{Writer}\ (\mathit{Min}\ \mathit{Int}))\ (\mathit{Reader}\ (\mathit{Min}\ \mathit{Int}))$, the Day convolution of the two effects using a common value type. We introduce four abbreviations:

$$\begin{aligned}&\textbf{type}\ \mathit{WInt} = \mathit{Writer}\ (\mathit{Min}\ \mathit{Int})\\ &\mathit{tellMin} :: \mathit{Int} \to \mathit{WInt}\ ()\\ &\mathit{tellMin}\ x = \mathit{tell}\ (\mathit{Min}\ x)\\ &\textbf{type}\ \mathit{RInt} = \mathit{Reader}\ (\mathit{Min}\ \mathit{Int})\\ &\mathit{askMin} :: \mathit{RInt}\ \mathit{Int}\\ &\mathit{askMin} = \mathit{fmap}\ \mathit{getMin}\ \mathit{ask}\end{aligned}$$

The core of the computation is the following function:

$$\begin{aligned}&\mathit{repminAux} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Day}\ \mathit{WInt}\ \mathit{RInt}\ (\mathit{Tree}\ \mathit{Int})\\ &\mathit{repminAux}\ t = \mathit{phase1}\ (\mathit{minAux}\ t) \mathbin{*\!>} \mathit{phase2}\ (\mathit{replaceAux}\ t)\end{aligned}$$

where the first phase writes each element in turn:

$$\begin{aligned}&\mathit{minAux} :: \mathit{Tree}\ \mathit{Int} \to \mathit{WInt}\ ()\\ &\mathit{minAux}\ (\mathit{Node}\ x\ \mathit{ts}) = \mathit{tellMin}\ x \mathbin{*\!>} \mathit{mapM\_}\ \mathit{minAux}\ \mathit{ts}\end{aligned}$$

and the second phase reads a fixed replacement value for each element:

$$\begin{aligned}&\mathit{replaceAux} :: \mathit{Tree}\ \mathit{Int} \to \mathit{RInt}\ (\mathit{Tree}\ \mathit{Int})\\ &\mathit{replaceAux}\ (\mathit{Node}\ x\ \mathit{ts}) = \mathit{pure}\ \mathit{Node} \mathbin{<\!*\!>} \mathit{askMin} \mathbin{<\!*\!>} (\mathit{mapM}\ \mathit{replaceAux}\ \mathit{ts})\end{aligned}$$

In fact, both phases are (depth-first) instances of *traverse* over trees—at least, if we allow the result returned in the first phase, which we discard anyway, to be a tree instead of void:

$$\begin{aligned}&\mathit{repminAux}' :: \mathit{Tree}\ \mathit{Int} \to \mathit{Day}\ \mathit{WInt}\ \mathit{RInt}\ (\mathit{Tree}\ \mathit{Int})\\ &\mathit{repminAux}'\ t = \mathit{phase1}\ (\mathit{minAux}'\ t) \mathbin{*\!>} \mathit{phase2}\ (\mathit{replaceAux}'\ t)\\ &\mathit{minAux}' :: \mathit{Tree}\ \mathit{Int} \to \mathit{WInt}\ (\mathit{Tree}\ ())\\ &\mathit{minAux}' = \mathit{traverse}\ (\lambda x \to \mathit{tellMin}\ x)\\ &\mathit{replaceAux}' :: \mathit{Tree}\ \mathit{Int} \to \mathit{RInt}\ (\mathit{Tree}\ \mathit{Int})\\ &\mathit{replaceAux}' = \mathit{traverse}\ (\lambda x \to \mathit{askMin})\end{aligned}$$

Now, *phase1* and *phase2* are applicative morphisms, so by the naturality axiom of traversal we have

$$\mathit{phase1}\ (\mathit{traverse}\ f) = \mathit{traverse}\ (\mathit{phase1}\ f)$$

and similarly for *phase2*. Therefore we can move the phase coercions inside the traversals:

$$
\begin{aligned}
&\mathit{repminAux''} :: \mathit{Tree\ Int} \to \mathit{Day\ WInt\ RInt\ (Tree\ Int)} \\
&\mathit{repminAux''}\ t = \mathit{minAux''}\ t \mathbin{*\!\!>} \mathit{replaceAux''}\ t \\
&\mathit{minAux''} :: \mathit{Tree\ Int} \to \mathit{Day\ WInt\ RInt\ (Tree\ ())} \\
&\mathit{minAux''} = \mathit{traverse}\ (\lambda x \to \mathit{phase1}\ (\mathit{tellMin}\ x)) \\
&\mathit{replaceAux''} :: \mathit{Tree\ Int} \to \mathit{Day\ WInt\ RInt\ (Tree\ Int)} \\
&\mathit{replaceAux''} = \mathit{traverse}\ (\lambda x \to \mathit{phase2}\ \mathit{askMin})
\end{aligned}
$$

Finally, the two traversal bodies commute, because they are in different phases, so we can fuse the two traversals into one:

$$
\begin{aligned}
&\mathit{repminAux'''} :: \mathit{Tree\ Int} \to \mathit{Day\ WInt\ RInt\ (Tree\ Int)} \\
&\mathit{repminAux'''} = \mathit{traverse}\ (\lambda x \to \mathit{phase1}\ (\mathit{tellMin}\ x) \mathbin{*\!\!>} \mathit{phase2}\ \mathit{askMin})
\end{aligned}
$$

To summarize the development:

$$
\begin{aligned}
&\quad \mathit{phase1}\ (\mathit{traverse}\ \mathit{tellMin}\ t) \mathbin{*\!\!>} \mathit{phase2}\ (\mathit{traverse}\ (\lambda x \to \mathit{askMin})\ t) \\
&= \quad \{\ \text{naturality in applicative functor}\ \} \\
&\quad \mathit{traverse}\ (\mathit{phase1} \circ \mathit{tellMin})\ t \mathbin{*\!\!>} \mathit{traverse}\ (\lambda x \to \mathit{phase2}\ \mathit{askMin})\ t \\
&= \quad \{\ \text{fusion of traversals}\ \} \\
&\quad \mathit{traverse}\ (\lambda x \to \mathit{phase1}\ (\mathit{tellMin}\ x) \mathbin{*\!\!>} \mathit{phase2}\ \mathit{askMin})
\end{aligned}
$$

So $\mathit{repminAux'''}$ describes a one-pass traversal over the tree, generating a two-phase computation for later execution.

Now we turn to the question of extracting the $\mathit{Tree\ Int} \to \mathit{Tree\ Int}$ outer function from the above core. We can run a computation in the Day convolution of $\mathit{Writer}\ S$ and $\mathit{Reader}\ S$ for the same type S by extracting the writer and reader components in parallel, as follows:

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

Note that this is circular, with s appearing on both sides of the local declaration, so the **let** must have **letrec** semantics. In particular,

$$
\begin{aligned}
&\mathit{repminWR}_{\mathrm{RSB}} :: \mathit{Tree\ Int} \to \mathit{Tree\ Int} \\
&\mathit{repminWR}_{\mathrm{RSB}}\ t = \mathit{parWR}\ (\mathit{repminAux'''}\ t)
\end{aligned}
$$

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

Conversely, we can extract the writer then the reader components sequentially:

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

Now there is no circularity, and a plain non-recursive **let** suffices. In particular,

$$repminWR_{\mathrm{ADP}} :: Tree\ Int \to Tree\ Int$$
$$repminWR_{\mathrm{ADP}}\ t = seqWR\ (repminAux'''\ t)$$

is Pettorossi's non-circular, higher-order solution to the repmin problem. In a lazy language, clearly $parWR$ and $seqWR$ are equal, and so too therefore are $repminWR_{\mathrm{RSB}}$ and $repminWR_{\mathrm{ADP}}$.

Observe that in both the lazy and strict solutions the values xs and ys are nested tuples of values: the values in xs are all the unit, and the values in ys are all the minimum value. The function f picks the values out of ys and assembles them into the resulting tree; xs is ignored. A biased version

$$\textbf{data}\ Day'\ f\ g\ a\ \textbf{where}$$
$$\quad Day' :: f\ (b \to a) \to g\ b \to Day'\ f\ g\ a$$

of Day convolution would avoid constructing the nested structure for xs in the first place.

5 Multiple phases

We now generalize from two-phase computations to multiple (zero or more) phases:

$$\textbf{data}\ Phases\ f\ a\ \textbf{where}$$
$$\quad Pure :: a \to Phases\ f\ a$$
$$\quad Link :: ((a, b) \to c) \to f\ a \to Phases\ f\ b \to Phases\ f\ c$$

Here, $Pure$ produces a chain with no effectful phases, and $Link$ adds one more effectful phase to the chain. It is basically a homogeneous iteration of Day convolution ($Link$ constructs the Day convolution of f with $Phases\ f$), just as lists are essentially a homogeneous iteration of pairing (with cons pairing a list head with a tail). There is a single initial value as the base case; each additional link in the chain adds a combining function and a collection of values; and the types are all compatible "in the obvious way". For example, for some given base applicative functor F, we can link together components of types

$$z :: Char$$
$$ys :: F\ Float$$
$$g :: (Float, Char) \to String$$
$$xs :: F\ Bool$$
$$f :: (Bool, String) \to Int$$

to form a chain

$$example = Link\ f\ xs\ (Link\ g\ ys\ (Pure\ z)) :: Phases\ F\ Int$$

We will always have the f argument to $Phases$ be at least a functor (justifying the phrase "a collection of" above), in which case $Phases\ f$ is also a functor:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (\mathit{Phases}\ f)\ \textbf{where}\\&\quad \mathit{fmap}\ g\ (\mathit{Pure}\ x) &&= \mathit{Pure}\ (g\ x)\\&\quad \mathit{fmap}\ g\ (\mathit{Link}\ f\ xs\ ys) &&= \mathit{Link}\ (g \circ f)\ xs\ ys\end{aligned}$$

That is, *example* is a symbolic represention of a collection of *Int*s.

5.1 Free applicatives

Capriotti and Kaposi [3] show that the datatype *Phases* constructs the *free applicative functor* on a given functor argument. We won't dwell on what "free" means here; we will observe simply that $\mathit{Phases}\ f$ can be given applicative structure when f is a functor:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ (\mathit{Phases}\ f)\ \textbf{where} && \text{-- not used}\\&\quad \mathit{unit} &&= \mathit{Pure}\ ()\\&\quad \mathit{Pure}\ x \otimes ys &&= \mathit{fmap}\ (x,)\ ys\\&\quad \mathit{Link}\ f\ xs\ ys \otimes zs &&= \mathit{Link}\ (\lambda(x,(y,z)) \to (f\ (x,y),z))\ xs\ (ys \otimes zs)\end{aligned}$$

Informally, this defines $\otimes$ to concatenate two of these chains. Indeed, if we define the 'length' of such a chain to be the number of *Link* constructors:

$$\begin{aligned}&\mathit{chlen} :: \mathit{Phases}\ f\ a \to \mathit{Int}\\&\mathit{chlen}\ (\mathit{Pure}\ x) &&= 0\\&\mathit{chlen}\ (\mathit{Link}\ f\ xs\ ys) &&= 1 + \mathit{chlen}\ ys\end{aligned}$$

then $\mathit{chlen}\ (xs \otimes ys) = \mathit{chlen}\ xs + \mathit{chlen}\ ys$.

However, this canonical applicative structure on chains is not helpful when considering multiple-phase computations, because concatenation is generally not the right thing to do; so we will pursue this avenue no further. Instead, the product operation should 'zip' together two chains; and this should be a 'long zip', returning a chain as long as its longer argument—as we already used for breadth-first enumeration. In order to combine elements of the chain pointwise, we need the stronger assumption that the f argument is itself applicative and not merely a functor [11]:

$$\begin{aligned}&\textbf{instance}\ \mathit{Applicative}\ f \Rightarrow \mathit{Applicative}\ (\mathit{Phases}\ f)\ \textbf{where}\\&\quad \mathit{unit} = \mathit{Pure}\ ()\\&\quad \mathit{Pure}\ x \otimes ys &&= \mathit{fmap}\ (x,)\ ys\\&\quad xs \otimes \mathit{Pure}\ y &&= \mathit{fmap}\ (,y)\ xs\\&\quad \mathit{Link}\ f\ xs\ ys \otimes \mathit{Link}\ g\ zs\ ws &&= \mathit{Link}\ (\mathit{cross}\ f\ g \circ \mathit{exch4})\ (xs \otimes zs)\ (ys \otimes ws)\end{aligned}$$

Now we have $\mathit{chlen}\ (xs \otimes ys) = \mathit{max}\ (\mathit{chlen}\ xs)\ (\mathit{chlen}\ ys)$.

Given that the chain is homogeneous and the functor argument is applicative, we can run each of the phases in a chain in turn to extract the collection of elements it represents:

$$\begin{aligned}&\mathit{runPhases} :: \mathit{Applicative}\ f \Rightarrow \mathit{Phases}\ f\ a \to f\ a\\&\mathit{runPhases}\ (\mathit{Pure}\ x) &&= \mathit{pure}\ x\\&\mathit{runPhases}\ (\mathit{Link}\ f\ xs\ ys) &&= \mathit{fmap}\ f\ (xs \otimes \mathit{runPhases}\ ys)\end{aligned}$$

Thus, *runPhases example* consists of values $f\ (x, g\ (y, z))$ where x, y are drawn pointwise from xs, ys (each of the first elements of xs, ys combined, then each of the second elements, and so on). In contrast, if we know no more about the functor argument, the only expansion we can give is to a nested collection of results: we could run the phases of *example* to yield an $F\ (F\ Int)$, containing values $f\ (x, g\ (y, z))$ where x, y are drawn from the cartesian product of xs, ys. On the other hand, if we knew further that the applicative argument were a monad, we could flatten the nesting to a single level.

5.2 Two phases, more or less

By design, *Phases f* is a generalization of the homogeneous Day convolution *Day f f*. So of course we can inject the latter into the former:

$$\begin{aligned}&inject :: Applicative\ f \Rightarrow Day\ f\ f\ a \to Phases\ f\ a\\&inject\ (Day\ f\ xs\ ys) = Link\ f\ xs\ (Link\ unitr\ ys\ (Pure\ ()))\end{aligned}$$

And of course, $chlen\ (inject\ xs) = 2$ for any two-phase computation xs.

Analogous to *phase1* and *phase2*, we define one function that embeds a computation into an arbitrary phase:

$$\begin{aligned}&phase :: Applicative\ f \Rightarrow Int \to f\ a \to Phases\ f\ a\\&phase\ 1 = now\\&phase\ i = later \circ phase\ (i - 1)\end{aligned}$$

where *now* embeds at phase one:

$$\begin{aligned}&now :: Applicative\ f \Rightarrow f\ a \to Phases\ f\ a\\&now\ xs = Link\ unitr\ xs\ (Pure\ ())\end{aligned}$$

and *later* shifts everything one phase later:

$$\begin{aligned}&later :: Applicative\ f \Rightarrow Phases\ f\ a \to Phases\ f\ a\\&later\ xs = Link\ unitl\ unit\ xs\end{aligned}$$

Note that we count phases from one, so that $chlen\ (phase\ i\ xs) = i$. Moreover, $inject \circ phase1$ corresponds to $phase\ 1$. They are not quite equal as values of type *Phases* (the chain length of the former is two, and of the latter is one), but we do have

$$runPhases \circ inject \circ phase1 = runPhases \circ phase\ 1 = id$$

5.3 The 'sort-tree' problem

A related problem to repmin is the 'sort-tree' problem [2,14], which extracts the elements of a tree as a list, sorts that list into ascending order, then relabels

the tree with the sorted list—but again does so in a single pass. (Bird called it 'sort-tips', because in his tree datatype the elements were all at the tips.)

We start with three phases, although only the first and last phase involve traversing the tree:

$$
\begin{aligned}
&\mathit{sortTree} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{sortTree}\ t = \mathit{evalState}\ (\mathit{runPhases}\ (\mathit{sortTreeAux}\ t))\ [\,] \\
&\mathit{sortTreeAux} :: \mathit{Ord}\ a \Rightarrow \mathit{Tree}\ a \to \mathit{Phases}\ (\mathit{State}\ [a])\ (\mathit{Tree}\ a) \\
&\mathit{sortTreeAux}\ t = \mathit{phase}\ 1\ (\mathit{traverse}\ \mathit{push}\ t) \mathbin{*\!\!>} \\
&\qquad\qquad\qquad \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort}) \mathbin{*\!\!>} \\
&\qquad\qquad\qquad \mathit{phase}\ 3\ (\mathit{traverse}\ (\lambda x \to \mathit{pop})\ t)
\end{aligned}
$$

The computation uses the state monad, where the state is a list of elements. We use the following operations provided by the state monad:

$$
\begin{aligned}
&\mathit{get} \quad :: \mathit{State}\ s\ s \\
&\mathit{put} \quad :: s \to \mathit{State}\ s\ () \\
&\mathit{modify} :: (s \to s) \to \mathit{State}\ s\ ()
\end{aligned}
$$

The auxilliary function constructs a three-phase computation in that monad. The main function initializes the state to the empty list, runs the three phases, discards the final state (which will again be the empty list), and returns the final tree. The first phase of the auxilliary function traverses the tree, pushing the elements one by one onto the stored list:

$$
\begin{aligned}
&\mathit{push} :: a \to \mathit{State}\ [a]\ () \\
&\mathit{push}\ x = \mathit{modify}\ (x{:})
\end{aligned}
$$

The second phase doesn't touch the tree; it just sorts the stored list. The third phase traverses the tree again, popping elements one by one off the stored list:

$$
\begin{aligned}
&\mathit{pop} :: \mathit{State}\ [a]\ a \\
&\mathit{pop} = \mathbf{do}\ \{x : xs \leftarrow \mathit{get}; \mathit{put}\ xs; \mathit{return}\ x\}
\end{aligned}
$$

Note that the first phase pushes the tree elements from left to right, so the resulting list is in reverse order; but that is irrelevant as input to sorting.

As before, the specification of the three phases can be rearranged:

$$
\begin{aligned}
&\mathit{sortTreeAux}'\ t = \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort}) \mathbin{*\!\!>} \\
&\qquad\qquad\qquad \mathit{phase}\ 1\ (\mathit{traverse}\ \mathit{push}\ t) \mathbin{*\!\!>} \\
&\qquad\qquad\qquad \mathit{phase}\ 3\ (\mathit{traverse}\ (\lambda x \to \mathit{pop})\ t)
\end{aligned}
$$

and traversal commutes with staging:

$$
\begin{aligned}
&\mathit{sortTreeAux}''\ t = \mathit{phase}\ 2\ (\mathit{modify}\ \mathit{sort}) \mathbin{*\!\!>} \\
&\qquad\qquad\qquad \mathit{traverse}\ (\lambda x \to \mathit{phase}\ 1\ (\mathit{push}\ x))\ t \mathbin{*\!\!>} \\
&\qquad\qquad\qquad \mathit{traverse}\ (\lambda x \to \mathit{phase}\ 3\ \mathit{pop})\ t
\end{aligned}
$$

and consecutive traversals with bodies in different phases fuse:

$$\begin{aligned} &sortTreeAux''' \; t = phase\ 2\ (modify\ sort) \mathbin{*\!>} \\ &\qquad\qquad traverse\ (\lambda x \to phase\ 1\ (push\ x) \mathbin{*\!>} phase\ 3\ pop)\ t \end{aligned}$$

Using $sortTreeAux'''$ in place of $sortTreeAux$ in $sortTree$ solves the sort-tree problem with—clearly!—a single traversal over the tree.

6 Breadth-first traversal in stages

Finally, let us return to breadth-first traversal. The key insight is that we can specify such a traversal *compositionally*, by constructing a multi-stage computation with one phase per level of the tree. This is achieved by the auxilliary function $bftAux$:

$$\begin{aligned} &bftAux :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to Phases\ f\ (Tree\ b) \\ &bftAux\ f\ (Node\ x\ ts) \\ &\quad = pure\ Node \mathbin{<\!*\!>} now\ (f\ x) \mathbin{<\!*\!>} later\ (traverse\ (bftAux\ f)\ ts) \end{aligned}$$

Informally, the root label x is processed 'now', in phase one. A multi-stage computation is constructed for each child in ts, zipped together by levels using $traverse$ for the list of children, then postponed until one phase 'later'. Finally, the resulting tree is assembled by applying the constructor $Node$ to the results of processing the root now and the children later. Then bft is obtained by collapsing the chain of phases into a single computation:

$$\begin{aligned} &bft :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to f\ (Tree\ b) \\ &bft\ f = runPhases \circ bftAux\ f \end{aligned}$$

It is instructive to compare $bftAux$ above with the depth-first instance of $traverse$ for trees that we had in Section 2.2—essentially:

$$\begin{aligned} &dft :: Applicative\ f \Rightarrow (a \to f\ b) \to Tree\ a \to f\ (Tree\ b) \\ &dft\ f\ (Node\ x\ ts) = pure\ Node \mathbin{<\!*\!>} f\ x \mathbin{<\!*\!>} traverse\ (dft\ f)\ ts \end{aligned}$$

which is recovered simply by deleting the staging annotations from $bftAux$.

In particular, we can relabel a tree in breadth-first order, without needing either queues [12] or cyclicity and laziness [10]:

$$\begin{aligned} &bfl :: Tree\ a \to [b] \to Tree\ b \\ &bfl\ t\ xs = evalState\ (bft\ (\lambda x \to pop)\ t)\ xs \end{aligned}$$

where pop is as defined for the sort-tree problem in Section 5.3.

7 Discussion

We have shown how staged computation can be expressed using a construction related to free applicative functors, but combining structures by 'zipping' instead of 'concatenating' them. Two-phase computation is captured by Day convolution, which Rivas and Jaskelioff [15] showed to be the natural monoidal structure underlying applicative functors. Multi-stage computation is captured by iterated Day convolution, which is the same datatype as for free applicative functors [3] but with a different applicative instance. Among other examples, we have used these constructions to clarify Bird's and Pettorossi's solutions to the 'repmin' problem, doing away with the laziness inherent in Bird's solution; and to provide an implementation of breadth-first effectful traversal that avoids a clumsy factorization into shape and contents.

That breadth-first effectful traversal can be used in particular to implement breadth-first relabelling of a tree with a stream of fresh labels. Our multi-stage solution to that problem avoids the circular definition and inherent laziness in our earlier program [10]. Okasaki [12] discusses the same problem, providing a solution that requires a more sophisticated queue datatype in order to achieve linear-time execution. Okasaki states that "lazy evaluation is required. Without lazy evaluation, you [...] would need [...] a separate pass"; we have shown that neither a fancy queue datatype nor laziness are needed in order to avoid multiple passes. One might even say that Okasaki's queues are a form of staging too, postponing actions for later execution.

To be fair, one might argue about the extent to which any of these solutions to the repmin problem and its ilk have "eliminated multiple traversals". It is clear that the original input data structure is traversed only once; but there is a case to be made that a copy is created in the first pass and traversed in the second pass. Pettorossi's solution $repmin_{\mathrm{ADP}}$ explicitly constructs a lambda abstraction that encodes a copy of the structure of the input tree, and then applies this function to the replacement value. One might say that Bird's solution $repmin_{\mathrm{RSB}}$ constructs that same copy implicitly.

Anyway, we make no claims that these transformations improve running time or space usage. More broadly, we have not concerned ourselves with making these traversals take linear time; for example, *bf* in Section 2.2 is not linear, because of repeated concatenations of lists. This issue can be addressed by using difference lists, and more generally by Cayley representations [11], but is orthogonal to our main argument.

Day convolution as defined in Section 4 is heterogeneous: the two phases can use different classes of effect, which will of course entail different methods of 'running'. In contrast, the multi-stage computations defined as iterated Day convolution in Section 5 are homogeneous: all phases must use the same class of effects, which can then be combined using the applicative multiplication. This is analogous to situation with lists: ordinary pairs are heterogeneous, but lists (iterated pairs) are homogeneous. However, with suitably expressive typing facilities, one can define a datatype of heterogeneous lists, perhaps indexed by a type-level list of types; and in the same way, one could define heterogeneous

multi-stage computations, indexed by a type-level list of applicative functors. But we know of no use for such a construction; nor is it clear how in general one would 'run' the hetereogeneous collection of phases.

A minor point to note is that, while pure *Day f f* computations have only one representation, those of *Phases f* have infinitely many, one for each chain length: *pure x*, *phase* 1 (*pure x*), *phase* 2 (*pure x*), ... This representational difference is meaningless and thus we consider *Phases f* computations equal up to a pure "tail". In fact, *phases i* is only an applicative morphism for this notion of equality. In case f is a *unital* applicative functor the representation could be normalised to eliminate pure tails. Following the notion of unital monads [16], a unital applicative functor is one for which it is possible to determine whether a given computation is in the image of *pure*.

Acknowledgements We are very grateful to the members of IFIP Working Group 2.1 and of the Algebra of Programming research group at Oxford, for patient listening and helpful comments while this work was gestating. Thanks are especially due to Alberto Pettorossi for furnishing copies of his papers [13,14], and to Alexander Vandenbroucke for helpful comments.

This paper is dedicated to the memory of Richard Bird, who inspired us all.

References

1. Bird, R., Gibbons, J., Mehner, S., Voigtländer, J., Schrijvers, T.: Understanding idiomatic traversals backwards and forwards. In: Haskell Symposium. ACM (2013). https://doi.org/10.1145/2503778.2503781
2. Bird, R.S.: Using circular programs to eliminate multiple traversals of data. Acta Informatica **21**, 239–250 (1984). https://doi.org/10.1007/BF00264249
3. Capriotti, P., Kaposi, A.: Free applicative functors. In: Levy, P.B., Krishnaswami, N. (eds.) Mathematically Structured Functional Programming. EPTCS, vol. 153, pp. 2–30 (2014). https://doi.org/10.4204/EPTCS.153.2
4. Danvy, O., Thiemann, P., Zerny, I.: Circularity and lambda abstraction: From Bird to Pettorossi and back. In: Plasmeijer, R. (ed.) Implementation and Application of Functional Languages. p. 85. ACM (2013). https://doi.org/10.1145/2620678.2620687
5. Gibbons, J.: Breadth-first traversal (Mar 2015), https://patternsinfp.wordpress.com/2015/03/05/breadth-first-traversal/
6. Gibbons, J., Jones, G.: The under-appreciated unfold. In: International Conference on Functional Programming. pp. 273–279. Baltimore, Maryland (Sep 1998). https://doi.org/10.1145/289423.289455
7. Gibbons, J., Kidney, D.O., Schrijvers, T., Wu, N.: Code for "Breadth-First Traversal Via Staging", http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/traversals.hs
8. Gibbons, J., dos Santos Oliveira, B.C.: The essence of the Iterator pattern. Journal of Functional Programming **19**(3,4), 377–402 (2009). https://doi.org/10.1017/S0956796809007291
9. Jaskelioff, M., Rypacek, O.: An investigation of the laws of traversals. In: Chapman, J., Levy, P.B. (eds.) Mathematically Structured Functional Programming. EPTCS, vol. 76, pp. 40–49 (2012). https://doi.org/10.4204/EPTCS.76.5

10. Jones, G., Gibbons, J.: Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Computer Science Report No. 71, Dept of Computer Science, University of Auckland (May 1993), http://www.cs.ox.ac.uk/publications/publication2363-abstract.html, also IFIP Working Group 2.1 working paper 705 WIN-2
11. Kidney, D.O., Wu, N.: Algebras for weighted search. Proceedings of the ACM on Programming Languages **5**(ICFP), 1–30 (2021). https://doi.org/10.1145/3473577
12. Okasaki, C.: Breadth-first numbering: Lessons from a small exercise in algorithm design. In: Odersky, M., Wadler, P. (eds.) International Conference on Functional Programming. pp. 131–136. ACM (2000). https://doi.org/10.1145/351240.351253
13. Pettorossi, A., Skowron, A.: Higher order generalization in program derivation. In: Ehrig, H., Kowalski, R.A., Levi, G., Montanari, U. (eds.) Theory and Practice of Software Development. Lecture Notes in Computer Science, vol. 250, pp. 182–196. Springer (1987). https://doi.org/10.1007/BFb0014981
14. Pettorossi, A., Skowron, A.: The lambda abstraction strategy for program derivation. Fundamenta Informaticae **XII**, 541–562 (1989). https://doi.org/10.3233/FI-1989-12407
15. Rivas, E., Jaskelioff, M.: Notions of computation as monoids. Journal of Functional Programming **27**, e21 (2017). https://doi.org/10.1017/S0956796817000132
16. Rivas, E., Jaskelioff, M., Schrijvers, T.: A unified view of monadic and applicative non-determinism. Science of Computer Programming **152**, 70–98 (2018). https://doi.org/10.1016/j.scico.2017.09.007

A Fusion of traversals

In this appendix we prove the fusion rule for traversals deployed in Section 4, using the Representation Theorem for traversals [1]. We take the opportunity to make the statement of the Representation Theorem more precise, using dependent types for indexing by size. Throughout the section, we fix an applicative functor F, and a traversable datatype T with corresponding *traverse* function (though of course they are arbitrary).

We will use the following gadgets for products of pure functions:

$$\begin{aligned}&\mathit{cross} :: (a \to b) \to (c \to d) \to (a, c) \to (b, d)\\ &\mathit{cross}\ f\ g\ (x, y) = (f\ x, g\ y)\end{aligned}$$

$$\begin{aligned}&\mathit{fork} :: (a \to b) \to (a \to c) \to a \to (b, c)\\ &\mathit{fork}\ f\ g\ x = \mathit{cross}\ f\ g\ (x, x)\end{aligned}$$

and their applicative counterparts for effectful functions:

$$\begin{aligned}&\mathit{crossA} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ c) \to (b \to f\ d) \to (a, b) \to f\ (c, d)\\ &\mathit{crossA}\ f\ g\ (x, y) = f\ x \otimes g\ y\end{aligned}$$

$$\begin{aligned}&\mathit{forkA} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to (a \to f\ c) \to a \to f\ (b, c)\\ &\mathit{forkA}\ f\ g\ x = \mathit{crossA}\ f\ g\ (x, x)\end{aligned}$$

The key property is for two effectful functions to commute, that is, for their effects not to interfere with each other:

Definition 1. *Given* $f :: A \to F\ B$ *and* $g :: C \to F\ D$*, say "*f *commutes with* g*" if*

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

for all $x :: A, y :: C$*.*

Now for two consecutive traversals of the same data structure, if the two bodies commute with each other, then the two traversals can be fused into one:

Theorem 2 (Fusion rule for traversals). *If* $f :: A \to F\ B$ *commutes with* $g :: A \to F\ C$*, then*

$$forkA\ (traverse\ f)\ (traverse\ g) = fmap\ unzip \circ traverse\ (forkA\ f\ g)$$

The rest of this appendix proves Theorem 2.

A.1 Length-indexed vectors

We can make the statement of the Representation Theorem more precise than was possible in Haskell at the time the theorem was published [1], by using dependent types to be explicit about the size of a data structure. But the results still hold without the size indexing.

The traditional datatype definition of Peano naturals

$$\mathbf{data}\ Nat = Z \mid S\ Nat$$

introduces a new type Nat with new value inhabitants $Z, S\ N$; but it also introduces a new kind Nat with new type inhabitants $Z, S\ N$. For example, here is a new type $Four$ of kind Nat, serving as a type-level representation of the number four:

$$\mathbf{type}\ Four = S\ (S\ (S\ (S\ Z)))$$

We can use these type-level numbers to specify the size of a data structure. Here are length-indexed vectors:

$$\begin{array}{l}\mathbf{data}\ Vec :: Nat \to * \to *\ \mathbf{where}\\ \quad VNil\ \ :: Vec\ Z\ a\\ \quad VCons :: (a, Vec\ n\ a) \to Vec\ (S\ n)\ a\end{array}$$

(the uncurried $VCons$ is for later convenience). Vectors are of course traversable, as lists are:

$$\begin{array}{l}\mathbf{instance}\ Traversable\ (Vec\ n)\ \mathbf{where}\\ \quad traverse\ f\ VNil\ \ \ \ \ \ \ \ \ \ \ \ \ \ = pure\ VNil\\ \quad traverse\ f\ (VCons\ (x, xs)) = fmap\ VCons\ (f\ x \otimes traverse\ f\ xs)\end{array}$$

A.2 Size-indexed trees

In the same way, we can define size-indexed trees and forests:

```
data TreeI :: Nat → ∗ → ∗ where
  NodeI  :: (a, ForestI n a) → TreeI (S n) a
data ForestI :: Nat → ∗ → ∗ where
  FNil   :: ForestI Z a
  FCons :: (TreeI m a, ForestI n a) → ForestI (Plus m n) a
```

Naturally, the size of a non-empty forest $FCons\ (t, ts)$ is the sum of the sizes of t and ts, so we need type-level addition:

```
type family Plus (n :: Nat) (m :: Nat) :: Nat
type instance Plus Z     n = n
type instance Plus (S m) n = S (Plus m n)
```

And of course, trees are traversable too—in various ways. Here is the definition of depth-first traversal:

```
instance Traversable (TreeI n) where
  traverse f (NodeI (x, ts)) = fmap NodeI (f x ⊗ traverseF f ts) where
    traverseF :: Applicative f ⇒ (a → f b) → ForestI n a → f (ForestI n b)
    traverseF f FNil          = pure FNil
    traverseF f (FCons (t, ts)) = fmap FCons (traverse f t ⊗ traverseF f ts)
```

A.3 Make functions

Definition 3. *Define*

```
type Make t n a = Vec n a → t n a
```

Then, for a given size-indexed traversable datatype T with corresponding traversal function traverse, a make function *for T is a polymorphic function*

```
make :: Make T n a
```

that constructs a data structure from its contents, preserving those contents:

```
contents ∘ make = contents
```

Here, *contents* returns the contents of a data structure as a list, in their order of traversal:

```
contents :: Traversable t ⇒ t a → [a]
contents = getConst ∘ traverse (λx → Const [x])
```

A.4 Representation Theorem

Note that what constitutes a make function for T depends on the corresponding *traverse* function, and in particular depends on the traversal order chosen for T. For example, given that we have defined traversal of trees to be depth-first, here is a make function of arity four for trees:

$$\begin{aligned}
&\mathit{makeDF} :: \mathit{Make\ TreeI\ Four\ a}\\
&\mathit{makeDF}\ (\mathit{VCons}\ (w, (\mathit{VCons}\ (x, \mathit{VCons}\ (y, \mathit{VCons}\ (z, \mathit{VNil}))))))) =\\
&\quad \mathit{NodeI}\ (w, \mathit{FCons}\ (\mathit{NodeI}\ (x, \mathit{FCons}\ (\mathit{NodeI}\ (y, \mathit{FNil}), \mathit{FNil})),\\
&\qquad\qquad \mathit{FCons}\ (\mathit{NodeI}\ (z, \mathit{FNil}), \mathit{FNil})))
\end{aligned}$$

Diagrammatically,

$$\langle w, x, y, z\rangle \quad \rightsquigarrow \quad \begin{array}{ccccc} w & \text{—} & x & \text{—} & y \\ & & \diagdown & & \\ & & & & z \end{array}$$

If we had defined traversal for trees instead to be breadth-first, this would not be a valid make function, because it would not preserve the order of elements.

The key insight is: for a given size-indexed traversable datatype and given definition of traversal, there is a unique corresponding make function, which uniquely relates data structures to the underlying vector of their contents, and traversal on the datatype to traversal on the underlying vector.

Theorem 4 (Representation Theorem). *For size-indexed traversable type T and data structure $t :: T\ N\ A$, there exists a unique make function $m :: (\forall a\ .\ \mathit{Make}\ T\ N\ a)$ and unique $xs :: \mathit{Vec}\ N\ A$ such that $t = m\ xs$ and $\mathit{contents}\ t = \mathit{contents}\ xs$. Moreover, for any $f :: B \to F\ C$ and $ys :: \mathit{Vec}\ N\ B$,*

$$\mathit{traverse}\ f\ (m\ ys) = \mathit{fmap}\ m\ (\mathit{traverse}\ f\ ys)$$

where the traverse on the right-hand side is on vectors.

The earlier statement and proof of this theorem [1] was rigorous but not formal, relying on ellipsis ("*make* x_1 ... x_n") for referring to the arity; we have made it more formal by using size-indexed datatypes. But the adapted proof is essentially the same, so we do not repeat it here.

A.5 Commutativity

Obviously, the notion of "commuting with" is symmetric, because *twist* is an involution. Moreover, pairing preserves commutability:

Lemma 5. *If f commutes with g and with h, then f commutes with $\mathit{crossA}\ g\ h$.*

Proof. We have:

$$
\begin{array}{ll}
 & \mathit{crossA}\ g\ h\ (y, z) \otimes f\ x \\
= & \quad \{\ \mathit{crossA}\ \} \\
 & (g\ y \otimes h\ z) \otimes f\ x \\
= & \quad \{\ \text{associativity}\ \} \\
 & \mathit{fmap}\ \mathit{assoc}\ (g\ y \otimes (h\ z \otimes f\ x)) \\
= & \quad \{\ f\ \text{commutes with}\ h\ \} \\
 & \mathit{fmap}\ \mathit{assoc}\ (g\ y \otimes (\mathit{fmap}\ \mathit{twist}\ (f\ x \otimes h\ z))) \\
= & \quad \{\ \text{naturality of}\ \otimes\ \} \\
 & \mathit{fmap}\ \mathit{assoc}\ (\mathit{fmap}\ (\mathit{cross}\ \mathit{id}\ \mathit{twist})\ (g\ y \otimes (f\ x \otimes h\ z))) \\
= & \quad \{\ \text{associativity}\ \} \\
 & \mathit{fmap}\ \mathit{assoc}\ (\mathit{fmap}\ (\mathit{cross}\ \mathit{id}\ \mathit{twist})\ (\mathit{fmap}\ \mathit{assoc}^{-1}\ ((g\ y \otimes f\ x) \otimes h\ z))) \\
= & \quad \{\ \mathit{assoc} \circ \mathit{cross}\ \mathit{id}\ \mathit{twist} \circ \mathit{assoc}^{-1} = \mathit{twist} \circ \mathit{assoc}^{-1} \circ \mathit{cross}\ \mathit{twist}\ \mathit{id}\ \} \\
 & \mathit{fmap}\ \mathit{twist}\ (\mathit{fmap}\ \mathit{assoc}^{-1}\ (\mathit{fmap}\ (\mathit{cross}\ \mathit{twist}\ \mathit{id})\ ((g\ y \otimes f\ x) \otimes h\ z))) \\
= & \quad \{\ \text{naturality of}\ \otimes\ \} \\
 & \mathit{fmap}\ \mathit{twist}\ (\mathit{fmap}\ \mathit{assoc}^{-1}\ (\mathit{fmap}\ \mathit{twist}\ (g\ y \otimes f\ x) \otimes h\ z)) \\
= & \quad \{\ f\ \text{commutes with}\ g\ \} \\
 & \mathit{fmap}\ \mathit{twist}\ (\mathit{fmap}\ \mathit{assoc}^{-1}\ ((f\ x \otimes g\ y) \otimes h\ z)) \\
= & \quad \{\ \text{associativity}\ \} \\
 & \mathit{fmap}\ \mathit{twist}\ (f\ x \otimes (g\ y \otimes h\ z)) \\
= & \quad \{\ \mathit{forkA}\ \} \\
 & \mathit{fmap}\ \mathit{twist}\ (f\ x \otimes \mathit{crossA}\ g\ h\ (y, z))
\end{array}
$$

(The step in the middle

$$\mathit{assoc} \circ \mathit{cross}\ \mathit{id}\ \mathit{twist} \circ \mathit{assoc}^{-1} = \mathit{twist} \circ \mathit{assoc}^{-1} \circ \mathit{cross}\ \mathit{twist}\ \mathit{id}$$

holds because both are the unique total function of type $((a, c), b) \to ((a, b), c)$. This proof would perhaps be clearer with appropriate string diagrams.)

Then traversal preserves commutability too:

Lemma 6. *If f commutes with g, then also traverse f commutes with g (and f commutes with traverse g, by symmetry of commutability).*

Proof. We prove that if f commutes with g then

$$g\ y \otimes \mathit{traverse}\ f\ \mathit{xs} = \mathit{fmap}\ \mathit{twist}\ (\mathit{traverse}\ f\ \mathit{xs} \otimes g\ y)$$

for $\mathit{xs} :: \mathit{Vec}\ N\ A$, $y :: B$, $f :: A \to F\ C$, $g :: B \to F\ D$, by induction on N. For the base case, we have:

$$
\begin{array}{ll}
 & g\ y \otimes \mathit{traverse}\ f\ \mathit{VNil} \\
= & \quad \{\ \text{definition of}\ \mathit{traverse}\ \text{on}\ \mathit{Vec}\ \} \\
 & g\ y \otimes \mathit{pure}\ \mathit{VNil} \\
= & \quad \{\ \text{applicative interchange law:}\ \mathit{us} \otimes \mathit{pure}\ v = \mathit{fmap}\ (,v)\ \mathit{us}\ \} \\
 & \mathit{fmap}\ (, \mathit{VNil})\ (g\ y)
\end{array}
$$

$$
\begin{array}{ll}
= & \{\ \mathit{twist}\ \} \\
& \mathit{fmap}\ \mathit{twist}\ (\mathit{fmap}\ (\mathit{VNil},)\ (g\ y)) \\
= & \{\ \mathit{fmap}\ (u,)\ vs = \mathit{pure}\ u \otimes vs\ \} \\
& \mathit{fmap}\ \mathit{twist}\ (\mathit{pure}\ \mathit{VNil} \otimes g\ y) \\
= & \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \} \\
& \mathit{fmap}\ \mathit{twist}\ (\mathit{traverse}\ f\ \ \mathit{VNil} \otimes g\ y)
\end{array}
$$

and for the inductive step:

$$
\begin{array}{ll}
& g\ y \otimes \mathit{traverse}\ f\ (\mathit{VCons}\ (x, xs)) \\
= & \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \} \\
& g\ y \otimes (\mathit{fmap}\ \mathit{VCons}\ (f\ x \otimes \mathit{traverse}\ f\ xs)) \\
= & \{\ \text{naturality of } \otimes\ \} \\
& \mathit{fmap}\ (\mathit{cross}\ \mathit{id}\ \mathit{VCons})\ (g\ y \otimes (f\ x \otimes \mathit{traverse}\ f\ xs)) \\
= & \{\ f \text{ and (by induction) } \mathit{traverse}\ f \text{ commute with } g\ \} \\
& \mathit{fmap}\ (\mathit{cross}\ \mathit{id}\ \mathit{VCons} \circ \mathit{twist})\ ((f\ x \otimes \mathit{traverse}\ f\ xs) \otimes g\ y) \\
= & \{\ \mathit{cross} \text{ and } \mathit{twist}\ \} \\
& \mathit{fmap}\ (\mathit{twist} \circ \mathit{cross}\ \mathit{VCons}\ \mathit{id})\ ((f\ x \otimes \mathit{traverse}\ f\ xs) \otimes g\ y) \\
= & \{\ \text{naturality of } \otimes\ \} \\
& \mathit{fmap}\ \mathit{twist}\ (\mathit{fmap}\ \mathit{VCons}\ (f\ x \otimes \mathit{traverse}\ f\ xs) \otimes g\ y) \\
= & \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \} \\
& \mathit{fmap}\ \mathit{twist}\ (\mathit{traverse}\ f\ (\mathit{VCons}\ (x, xs)) \otimes g\ y)
\end{array}
$$

Therefore, two traversals with bodies that commute will fuse:

Lemma 7. *If f commutes with g, then traverse f fuses with traverse g on vectors:*

$$\mathit{forkA}\ (\mathit{traverse}\ f)\ (\mathit{traverse}\ g)\ xs = \mathit{fmap}\ \mathit{unzip}\ (\mathit{traverse}\ (\mathit{forkA}\ f\ g)\ xs)$$

for all $xs :: \mathit{Vec}\ N\ A$.

Proof. Proof by induction on N. For the base case:

$$
\begin{array}{ll}
& \mathit{forkA}\ (\mathit{traverse}\ f)\ (\mathit{traverse}\ g)\ \mathit{VNil} \\
= & \{\ \text{definition of } \mathit{forkA}\ \} \\
& \mathit{traverse}\ f\ \ \mathit{VNil} \otimes \mathit{traverse}\ g\ \mathit{VNil} \\
= & \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \} \\
& \mathit{pure}\ \mathit{VNil} \otimes \mathit{pure}\ \mathit{VNil} \\
= & \{\ \text{applicative law}\ \} \\
& \mathit{pure}\ (\mathit{VNil}, \mathit{VNil}) \\
= & \{\ \text{definition of } \mathit{unzip}\ \} \\
& \mathit{pure}\ (\mathit{unzip}\ \mathit{VNil}) \\
= & \{\ \text{naturality of } \mathit{pure}\ \} \\
& \mathit{fmap}\ \mathit{unzip}\ (\mathit{pure}\ \mathit{VNil}) \\
= & \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \} \\
& \mathit{fmap}\ \mathit{unzip}\ (\mathit{traverse}\ (\mathit{forkA}\ f\ g)\ \mathit{VNil})
\end{array}
$$

For the inductive step, we use some abbreviations:

$$
\begin{aligned}
&\mathit{assoc4} :: (a,((b,c),d)) \to ((a,b),(c,d))\\
&\mathit{assoc4} = \mathit{assoc} \circ \mathit{cross}\ \mathit{id}\ \mathit{assoc}^{-1}\\
&\mathit{unassoc4} :: ((a,b),(c,d)) \to (a,((b,c),d))\\
&\mathit{unassoc4} = \mathit{cross}\ \mathit{id}\ \mathit{assoc} \circ \mathit{assoc}^{-1}\\
&\mathit{twist4} :: (a,((b,c),d)) \to (a,((c,b),d))\\
&\mathit{twist4} = \mathit{cross}\ \mathit{id}\ (\mathit{cross}\ \mathit{twist}\ \mathit{id})\\
&\mathit{exch4} :: ((a,b),(c,d)) \to ((a,c),(b,d))\\
&\mathit{exch4} = \mathit{assoc4} \circ \mathit{twist4} \circ \mathit{unassoc4}
\end{aligned}
$$

Then:

$$
\begin{aligned}
&\ \mathit{forkA}\ (\mathit{traverse}\ f)\ (\mathit{traverse}\ g)\ (\mathit{VCons}\ (x,xs))\\
=&\quad \{\ \text{definition of } \mathit{forkA}\ \}\\
&\ \mathit{traverse}\ f\ (\mathit{VCons}\ (x,xs)) \otimes \mathit{traverse}\ g\ (\mathit{VCons}\ (x,xs))\\
=&\quad \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \}\\
&\ \mathit{fmap}\ \mathit{VCons}\ (f\ x \otimes \mathit{traverse}\ f\ xs) \otimes \mathit{fmap}\ \mathit{VCons}\ (g\ x \otimes \mathit{traverse}\ g\ xs)\\
=&\quad \{\ \text{naturality of } \otimes\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons})\\
&\qquad ((f\ x \otimes \mathit{traverse}\ f\ xs) \otimes (g\ x \otimes \mathit{traverse}\ g\ xs))\\
=&\quad \{\ \text{associativity, twice}\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons} \circ \mathit{assoc4})\\
&\qquad (f\ x \otimes ((\mathit{traverse}\ f\ xs \otimes g\ x) \otimes \mathit{traverse}\ g\ xs))\\
=&\quad \{\ f \text{ commutes with } g, \text{ so } \mathit{traverse}\ f \text{ commutes with } g\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons} \circ \mathit{assoc4} \circ \mathit{twist4})\\
&\qquad (f\ x \otimes ((g\ x \otimes \mathit{traverse}\ f\ xs) \otimes \mathit{traverse}\ g\ xs))\\
=&\quad \{\ \text{associativity, twice}\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons} \circ \mathit{assoc4} \circ \mathit{twist4} \circ \mathit{unassoc4})\\
&\qquad ((f\ x \otimes g\ x) \otimes (\mathit{traverse}\ f\ xs \otimes \mathit{traverse}\ g\ xs))\\
=&\quad \{\ \mathit{exch4}\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons} \circ \mathit{exch4})\\
&\qquad ((f\ x \otimes g\ x) \otimes (\mathit{traverse}\ f\ xs \otimes \mathit{traverse}\ g\ xs))\\
=&\quad \{\ \text{definition of } \mathit{forkA}, \text{induction}\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons} \circ \mathit{exch4})\\
&\qquad (\mathit{forkA}\ f\ g\ x \otimes \mathit{fmap}\ \mathit{unzip}\ (\mathit{traverse}\ (\mathit{forkA}\ f\ g)\ xs))\\
=&\quad \{\ \text{naturality of } \otimes\ \}\\
&\ \mathit{fmap}\ (\mathit{cross}\ \mathit{VCons}\ \mathit{VCons} \circ \mathit{exch4} \circ \mathit{cross}\ \mathit{id}\ \mathit{unzip})\\
&\qquad (\mathit{forkA}\ f\ g\ x \otimes \mathit{traverse}\ (\mathit{forkA}\ f\ g)\ xs)\\
=&\quad \{\ \mathit{unzip}\ \text{(see below)}\ \}\\
&\ \mathit{fmap}\ (\mathit{unzip} \circ \mathit{VCons})\ (\mathit{forkA}\ f\ g\ x \otimes \mathit{traverse}\ (\mathit{forkA}\ f\ g)\ xs)\\
=&\quad \{\ \text{definition of } \mathit{traverse} \text{ on } \mathit{Vec}\ \}\\
&\ \mathit{fmap}\ \mathit{unzip}\ (\mathit{traverse}\ (\mathit{forkA}\ f\ g)\ (\mathit{VCons}\ (x,xs)))
\end{aligned}
$$

The penultimate step

$$unzip \circ VCons = cross\ VCons\ VCons \circ exch4 \circ cross\ id\ unzip$$

is basically the definition of *unzip* on a *VCons*, where both sides have type

$$((a, b), Vec\ n\ (a, b)) \to (Vec\ (S\ n)\ a, Vec\ (S\ n)\ b)$$

Now Theorem 2 reduces to Lemma 7, by the Representation Theorem (Theorem 4).

Proof (of Theorem 2). We have $t :: T\ N\ A$ for size-indexed traversable type T and arity N, a make function $m :: \forall a\ .\ Make\ T\ N\ a$ and contents $xs :: Vec\ N\ A$ such that $t = m\ xs$, and traversal bodies $f :: A \to F\ B$ and $g :: A \to F\ C$ that commute. Then:

$$\begin{array}{ll} & forkA\ (traverse\ f)\ (traverse\ g)\ t \\ = & \quad \{ \text{ Representation Theorem } \} \\ & forkA\ (fmap\ m \circ traverse\ f)\ (fmap\ m \circ traverse\ g)\ xs \\ = & \quad \{ \text{ naturality of } forkA \ \} \\ & fmap\ (cross\ m\ m)\ (forkA\ (traverse\ f)\ (traverse\ g)\ xs) \\ = & \quad \{ \text{ traverse fusion on vectors } \} \\ & fmap\ (cross\ m\ m \circ unzip)\ (traverse\ (forkA\ f\ g)\ xs) \\ = & \quad \{\ cross\ m\ m \circ unzip = unzip \circ m \text{ (see below) } \} \\ & fmap\ (unzip \circ m)\ (traverse\ (forkA\ f\ g)\ xs) \\ = & \quad \{ \text{ Representation Theorem } \} \\ & fmap\ unzip\ (traverse\ (forkA\ f\ g)\ t) \end{array}$$

where the penultimate step

$$cross\ m\ m \circ unzip = unzip \circ m$$

is discharged as follows:

$$\begin{array}{ll} & cross\ m\ m \circ unzip \\ = & \quad \{ \text{ definition of } unzip \ \} \\ & cross\ m\ m \circ fork\ (fmap\ fst)\ (fmap\ snd) \\ = & \quad \{ \text{ fusing } cross \text{ and } fork \ \} \\ & fork\ (m \circ fmap\ fst)\ (m \circ fmap\ snd) \\ = & \quad \{ \text{ naturality of } m \ \} \\ & fork\ (fmap\ fst \circ m)\ (fmap\ snd \circ m) \\ = & \quad \{ fork \text{ fusion } \} \\ & fork\ (fmap\ fst)\ (fmap\ snd) \circ m \\ = & \quad \{ \text{ definition of } unzip \ \} \\ & unzip \circ m \end{array}$$