

Effective Reasoning about Effectful Traversals

Jeremy Gibbons and Richard Bird
Oxford University Computing Laboratory

March 24, 2011

Abstract

A recent paper by Graham Hutton and Diana Fulger (‘Reasoning about Effects: Seeing the Wood through the Trees’, *Trends in Functional Programming*, 2008) addresses the problem of reasoning about effectful functional programs, using a relabelling function on binary trees as a simple example. We argue that their approach is less effective than it might be, because they miss two opportunities for higher-level reasoning: abstraction from the particular *kinds of effect* (the choice of monad) and from the *pattern of recursion* (the flow of computation). We present two alternative approaches using properties of idiomatic traversals, which cleanly separate the twin concerns of the kinds of effect and the pattern of recursion.

1 Introduction

Purely functional languages are appealing for the purposes of reasoning about programs: *referential transparency* allows the use of simple equational reasoning, substituting equals for equals, and *higher-order functions* encourage the statement and exploitation of higher-level theorems about patterns in programs.

But purity is—at least at first glance—rather limiting, ruling out many interesting computational effects such as state, exceptions, input/output, and non-determinism. Nevertheless, Moggi [9] famously observed that *monads* can be used to encapsulate such impure effects in a pure setting; more recently, related mathematical structures such as *arrows* [5] and *idioms* (or *applicative functors*) [8] have been shown to offer some additional advantages.

But how does the incorporation of impure effects affect equational reasoning about programs? In a recent paper [6], Hutton and Fulger argue that ‘there has been comparatively little progress’ on this problem, and propose by way of an example an approach to reasoning about programs with effects. The deliberately simple example they choose is one of relabelling binary trees, and the reasoning problem is to demonstrate that the relabelling (which is expressed using the state monad) generates distinct labels.

We argue that Hutton and Fulger’s development is somewhat unsatisfactory, for two reasons. Firstly, despite noting that ‘unfortunately, it still remains standard practice to just expand out the basic effectful operations when trying to reason’, their proof does exactly that—they reduce stateful computations to pure functions that accept and return a state. (This expansion is encapsulated in separate lemmas, but that is a surface textual abstraction rather than a deeper mathematical one.) This means that the effort invested into reasoning about a stateful program cannot be reused if the program is modified to exploit other kinds of effects. Moreover, it is unclear whether the approach can be used at all for effects such as input/output, which don’t have pure equivalents like state transformers do.

Secondly, they resort to explicit recursion in implementing the relabelling function, which consequently forces them to use explicit induction in reasoning about it. They conclude that ‘it would be interesting to see if there is any benefit to be gained from using a more structured approach to recursion’.

In this paper, we show that there is a great benefit to be gained from using structured recursion; indeed, the essence of this particular problem is precisely the structure of the recursion. In particular, the tree relabelling problem is an elegant example of an *idiomatic traversal*. Moreover, stating the problem in terms of a structured recursion operation such as idiomatic traversal forms a clear separation of concerns between the ‘plumbing’ aspects of the problem (which only concern the traversal) and the ‘computational’ aspects (which concern the particular effects); the majority of the reasoning can be performed independently of the effects, and so is applicable to kinds of effect rather than just one.

Nevertheless, we are not entirely satisfied with our development here: it depends on some properties of idiomatic traversal that seem entirely reasonable, but that might be reducible to simpler axioms. We tell the story so far, in the hope that others may be able to help to advance it.

The rest of the paper is structured as follows. Section 2 formalizes the illustrative problem of tree labelling; Section 3 gives the definitions of idioms and idiomatic traversal, and presents tree labelling as an idiomatic traversal; and Section 4 recalls some standard properties of idioms and idiomatic traversal. Our contribution starts in Section 5, where we present an unsuccessful attempt at solving the problem in terms of inversion using the composition of idioms; Section 6 presents another unsuccessful attempt, using composition of monads instead. Section 7 is more successful, making use of a notion of reverse traversals; Section 8 presents another promising approach, based on an idea of Ondřej Rypáček. Section 9 concludes.

2 Tree labelling

The benchmark problem introduced by Hutton and Fulger [6] around which to study reasoning about effects is that of the ‘correctness’ of a program for labelling binary trees. Here is the tree datatype in question:

```
data Tree a = Tip a | Bin (Tree a) (Tree a)
```

Hutton and Fulger offer several related labelling functions; the particular variation we will consider will annotate a tree of *as* with elements drawn from an infinite stream of *bs*, the latter threaded through the computation via the *State* monad.

$$\text{label} :: \text{Tree } a \rightarrow \text{State } [b] (\text{Tree } (a, b))$$

(The main differences between our version of the labelling function and Hutton and Fulger’s are firstly that we annotate the tree elements with stream labels, whereas they replace elements with labels, and secondly, that our function is polymorphic in the type *b* of labels, whereas they consider specifically streams of integers of the form $[n, n + 1 \dots]$.)

The essential correctness property that Hutton and Fulger wish to prove is that tree elements are replaced with distinct labels. Because our version is polymorphic in the label type, we cannot talk about ‘distinctness’; instead, we require that the labels used are drawn without repetition from the given stream—consequently, if the stream has no duplicates, the labels will be distinct.

This in turn is a corollary of the following property: that the sequence of labels with which the tree is annotated, when prepended back on to the stream of unused labels, forms the original input stream of labels. Let us provide a function *labels* to extract the annotations from a tree:

$$\begin{aligned} \text{labels} &:: \text{Tree } (a, b) \rightarrow [b] \\ \text{labels } (\text{Tip } (a, b)) &= [b] \\ \text{labels } (\text{Bin } t \ u) &= \text{labels } t \ ++ \ \text{labels } u \end{aligned}$$

Then the crucial property is that

$$\text{runState } (\text{label } t) \ xs = (u, ys) \quad \Rightarrow \quad \text{labels } u \ ++ \ ys = xs$$

3 Idiomatic traversals

We argue that the problem is inherently one about *effectful iterations* over data structures; moreover, any reasoning conducted should as far as possible be abstracted both from the specific computational effect and from the shape of the data structure being iterated over. We encapsulate the effects as *idioms* [8], rather than the more familiar monads; as we shall see, they have better compositional properties, which will prove useful in what follows. And we encapsulate the shape of the data structure in terms of idiomatic *traversals*, as introduced by McBride and Paterson [8] and studied in more depth by Gibbons and Oliveira [3].

We briefly introduce those two notions here, and show their immediate connection to the tree labelling problem, before moving on in Section 4 to a review of the laws of idiomatic traversals.

3.1 Idioms

Idioms are functors with some additional structure:

```
class Functor m => Idiom m where
  pure :: a -> m a
  ( $\otimes$ ) :: m (a -> b) -> m a -> m b
```

satisfying four laws:

```
pure id  $\otimes$  u           = u           -- identity
pure (o)  $\otimes$  u  $\otimes$  v  $\otimes$  w = u  $\otimes$  (v  $\otimes$  w) -- composition
pure f  $\otimes$  pure x      = pure (f x)  -- homomorphism
u  $\otimes$  pure x           = pure ( $\lambda$ f -> f x)  $\otimes$  u -- interchange
```

For clarity, we will sometimes subscript the *Idiom* methods with the particular idiom in question: $\text{pure}_m a$ and $x \otimes_m y$.

3.2 Monadic idioms

Idioms generalize monads, in the sense that every monad is also an idiom. For example, the *State* monad is also an idiom, with operations defined as follows:

```
instance Idiom (State s) where
  pure a      = return a
  mf  $\otimes$  mx = do { f <- mf; x <- mx; return (f x) }
```

That is, pure computations coincide with the ‘return’ of the monad, and idiomatic application yields the effects of evaluating the function before the effects of evaluating the argument (as captured by the function *ap* in the Haskell libraries). (The same construction works for any monad, and can be given once and for all, assuming common extensions to the type class mechanism of Haskell 98.)

3.3 Monoidal idioms

Idioms strictly generalize monads; there are idioms that do not arise from an underlying monad. In particular, any constant functor returning a monoidal type yields a ‘phantom’ idiom, in which the type parameter is a phantom type: pure computations yield the neutral element of the monoid, and idiomatic application reduces to the binary operator.

```
newtype K a b = K { unK :: a }
instance Functor (K a) where
  fmap f (K a) = K a
instance Monoid a => Idiom (K a) where
  pure a      = K mempty
  K a  $\otimes$  K b = K (mappend a b)
```

(Recall that *mappend* and *mempty* are the names given to the binary operation and its unit in the Haskell library *Data.Monoid*.)

3.4 Traversal

Traversable datatypes support *traverse* and *dist*, which are interdefinable. The former applies an effectful body to every element of a data structure, collecting all the effects in order; in the case of a monadic idiom, this reduces to a ‘monadic map’—called *mapM* in the Haskell libraries in the further specialized case of traversals of lists. The latter takes a data structure of computations, and chains together all the effects, distributing the data structure over the monad; for monadic idioms and lists, this is called *sequence* in the Haskell libraries.

```
class Functor t => Traversable t where
  traverse :: Idiom m => (a -> m b) -> t a -> m (t b)
  traverse f = dist o fmap f
  dist :: Idiom m => t (m a) -> m (t a)
  dist = traverse id
```

Again, for clarity, we will sometimes subscript the *Traversable* methods with the particular idiom in question: *traverse_m* and *dist_m*.

3.5 Tree labelling, idiomatically

For example, trees form a traversable datatype; traversal of a tip involves visiting the sole element, and traversal of a binary node involves traversing the left and then the right subtree.

```
instance Functor Tree where
  fmap f (Tip a)   = Tip (f a)
  fmap f (Bin t u) = Bin (fmap f t) (fmap f u)
instance Traversable Tree where
  traverse f (Tip a)   = pure Tip * f a
  traverse f (Bin t u) = pure Bin * traverse f t * traverse f u
```

(Indeed, recent Haskell compilers can automatically derive these two instances.)

Then tree labelling is a tree traversal in the monadic idiom arising from the *State* monad—the ‘body’ function consumes a single label from the stream, using that to adorn a single tree element.

```
adorn :: a -> State [b] (a, b)
adorn x = do { y : ys ← get; put (ys); return (x, y) }
label = traverse adorn
```

So reasoning about labelling ought to reduce to properties of idiomatic *traverse*.

4 Properties of idioms and traversal

We briefly remind ourselves of some properties of idioms and of idiomatic traversal, following [8] and [3].

4.1 Composition of idioms

Idioms compose nicely, in exactly the way that monads don't:

```

data (m·n) a = C { unC :: m (n a) }
instance (Functor m, Functor n) => Functor (m·n) where
  fmap f (C x) = C (fmap (fmap f) x)
instance (Idiom m, Idiom n) => Idiom (m·n) where
  pure a          = C (pure (pure a))
  C mnf ⊗ C mnx = C (pure (⊗) ⊗ mnf ⊗ mnx)

```

This prompts the introduction of a composition operator for idiomatic computations:

```

(⊙) :: (Idiom m, Idiom n) => (b → n c) → (a → m b) → a → (m·n) c
g ⊙ f = C ∘ fmap g ∘ f

```

Of course, there's an identity idiom too:

```

newtype Id a = I { unI :: a }
instance Functor Id where
  fmap f (I x) = I (f x)
instance Idiom Id where
  pure a    = I a
  I f ⊗ I x = I (f x)

```

4.2 Traversal respects composition of idioms

We expect traversal to respect the compositional structure of idioms: for a pure function $f :: a \rightarrow b$,

$$\text{traverse}_{Id} (I \circ f) = I \circ \text{fmap } f$$

and, for effectful bodies $f :: a \rightarrow m b$ and $g :: b \rightarrow n c$,

$$\text{traverse}_{m \cdot n} (C \circ \text{fmap } g \circ f) = C \circ \text{fmap} (\text{traverse}_n g) \circ \text{traverse}_m f$$

Using composition of idioms, the latter is equivalent to

$$\text{traverse}_{m \cdot n} (g \odot f) = \text{traverse}_n g \odot \text{traverse}_m f$$

which can be seen as a fusion law for idiomatic traversals.

4.3 Traversal is natural in the idiom

An *idiom morphism* from idiom m to idiom n is natural transformation $\phi : m \dot{\rightarrow} n$ such that

$$\begin{aligned}\phi (\text{pure}_m a) &= \text{pure}_n a \\ \phi (mf \otimes_m mx) &= \phi mf \otimes_n \phi mx\end{aligned}$$

We also expect traversal to be natural in the idiom: for an idiom morphism $\phi : m \dot{\rightarrow} n$, we expect the laws

$$\begin{aligned}\phi \circ \text{traverse}_m f &= \text{traverse}_n (\phi \circ f) \\ \phi \circ \text{dist}_m &= \text{dist}_n \circ \text{fmap } \phi\end{aligned}$$

to hold (each of which implies the other).

In particular, $\text{pure}_m \circ \text{unI} : \text{Id} \dot{\rightarrow} m$ is an idiom morphism; as a consequence, we get the purity law that traversal with *pure* is itself just *pure*:

$$\text{traverse}_m \text{pure}_m = \text{pure}_m$$

Also, when m is the idiom arising from a commutative monad, then $\text{join} \circ \text{unC} : m \cdot m \dot{\rightarrow} m$ is also an idiom morphism; as a consequence, we get a special fusion law for monadic traversals in a common commutative monad:

$$\text{traverse}_m g \bullet \text{traverse}_m f = \text{traverse}_m (g \bullet f)$$

where ‘ \bullet ’ is Kleisli composition, $\text{join} :: m (m a) \rightarrow m a$ is monad multiplication, and $f :: a \rightarrow m b$ and $g :: b \rightarrow m c$ both use the same (commutative) monad.

4.4 Backwards idioms

Another property enjoyed by idioms and not in general by monads is that, for each idiom m , there is a corresponding ‘backwards’ idiom \tilde{m} —the same type, but with the effects happening in the opposite order [3], as defined by:

$$\begin{aligned}\text{pure}_{\tilde{m}} a &= \text{pure}_m a \\ mf \otimes_{\tilde{m}} mx &= \text{pure}_m (\text{flip } \$) \otimes_m mx \otimes_m mf\end{aligned}$$

Let us define a simple wrapper type for converting an idiom to run backwards:

```
newtype Back m a = B { unB :: m a }
instance Functor m => Functor (Back m) where
  fmap f (B x) = B (fmap f x)
instance Idiom m => Idiom (Back m) where
  pure a          = B (pure a)
  B mf ⊗ B mx    = B (pure (flip $)) ⊗ mx ⊗ mf
```

This wrapper type is useful for qualifying idiomatic computations to indicate that the order of effects should be reversed: rather than defining an alternative backwards instance for a given type (and dealing with the consequent issues of overlapping instances), we make use of a distinct wrapped type for which the instance is deduced to be the reverse of that for the given type.

The backwards construction makes $B : \tilde{m} \rightarrow \text{Back } m$ an idiom morphism; consequently, following Section 4.3, it respects traversal:

$$B \circ \text{traverse}_{\tilde{m}} f = \text{traverse}_{\text{Back } m} (B \circ f)$$

or equivalently:

$$\text{traverse}_{\tilde{m}} f = \text{unB} \circ \text{traverse}_{\text{Back } m} (B \circ f)$$

Therefore, let us define

$$\begin{aligned} \text{recurse} &:: (\text{Idiom } m, \text{Traversable } t) \Rightarrow (a \rightarrow m \ b) \rightarrow t \ a \rightarrow m \ (t \ b) \\ \text{recurse } f &= \text{unB} \circ \text{traverse} \ (B \circ f) \end{aligned}$$

so that

$$\text{recurse}_m f = \text{traverse}_{\tilde{m}} f$$

That is, $\text{recurse}_m f$ is a traversal, like $\text{traverse}_m f$, visiting every element of a data structure; but the effects occur in the opposite order.

5 Reasoning about idiomatic traversal

How might we set about proving the property at the end of Section 2 stating the correctness of tree relabelling? It is difficult to see how to make progress, because the two functions *label* and *labels* are written in quite different styles—the first as an effectful traversal, and the second as a pure function—and so their combination requires flattening the ‘state’ abstraction (via the *runState* function). Unifying the two styles entails either writing *label* in a pure style (which is possible, but which amounts to falling back to first principles), or writing *labels* in an effectful style. Hutton and Fulger took the former approach; we take the latter.

In fact, we can extract the labels from an annotated tree as another stateful traversal, operating on the same state type of streams of labels. The body of the traversal strips the label from a single labelled element, and returns it to the stream of unused labels:

$$\begin{aligned} \text{strip} &:: (a, b) \rightarrow \text{State } [b] \ a \\ \text{strip } (x, y) &= \mathbf{do} \ \{ ys \leftarrow \text{get}; \text{put } (y : ys); \text{return } x \} \end{aligned}$$

We can then traverse with this body, stripping all the labels off a labelled tree.

$unlabel :: Tree (a, b) \rightarrow State [b] (Tree a)$
 $unlabel = traverse strip$

Now one might expect that *label*—which adorns a tree with labels from a stream—and *unlabel*—which strips those labels off again, and returns them to the stream—would be each other’s inverses, in some sense.

Suppose that m, n are idioms, that $f :: a \rightarrow m b$ and $g :: b \rightarrow n a$, and that $g \odot f = pure_{m.n}$. Then

$$\begin{aligned}
& traverse_n g \odot traverse_m f \\
= & \llbracket \text{traversal respects composition} \rrbracket \\
& traverse_{m.n} (g \odot f) \\
= & \llbracket \text{hypothesis} \rrbracket \\
& traverse_{m.n} (pure_{m.n}) \\
= & \llbracket \text{purity} \rrbracket \\
& pure_{m.n}
\end{aligned}$$

This would be applicable in our situation, if we could show that

$$strip \odot adorn = pure_{State [b].State [b]}$$

Sadly, this property does not hold: $strip \odot adorn$ is observably different from $pure_{State [b].State [b]}$. It turns out that \odot is a rather odd kind of composition, in the sense that it doesn’t fully hide the intermediate stage of a composite computation $f \odot g$: it is possible to recover the effects of g alone, after the fact. Specifically, one can show that

$$pure (const ()) \otimes unC ((g \odot f) a) = pure (const ()) \otimes f a$$

That is, from the application $(g \odot f) a$ of the idiomatic composition $g \odot f$ to a , it is possible to extract the effect of $f a$ alone—albeit not the result, which is discarded above by the application of $const ()$ —regardless of what g does. In particular, since *adorn* has a non-trivial effect on the state, $strip \odot adorn$ is distinguishable from $pure_{State [b].State [b]}$ —even if *strip* subsequently undoes the effects, the intermediate state betraying the effects is observable.

6 Reasoning about monadic traversal

Only in the case that f and g both use the same idiom, and this idiom is monadic, can that intermediate state be hidden—by using Kleisli composition. Recall that $g \bullet f = join \circ fmap_m g \circ f = join \circ unC \circ (g \odot f)$ for $f :: a \rightarrow m b$ and $g :: b \rightarrow m c$; it is really the application of *join* that collapses the two stages of effects into one, and hides the intermediate stage.

Happily, both *strip* and *adorn* use the same monadic idiom; moreover, $strip \bullet adorn$ really is *return*, the identity Kleisli arrow for the state monad. So we might hope to apply the following reasoning:

$$\begin{aligned}
& \text{traverse}_m g \bullet \text{traverse}_m f \\
= & \llbracket \text{traversal respects commutative Kleisli composition} \rrbracket \\
& \text{traverse}_m (g \bullet f) \\
= & \llbracket \text{hypothesis} \rrbracket \\
& \text{traverse}_m (\text{pure}_m) \\
= & \llbracket \text{purity} \rrbracket \\
& \text{pure}_m
\end{aligned}$$

Unfortunately, the state monad is the epitome of noncommutativity, so this reasoning is not applicable to our problem either.

(Incidentally, Gibbons and Oliveira [3, Section 5.4] conjectured that $\text{traverse}_m g \bullet \text{traverse}_m f = \text{traverse}_m (g \bullet f)$ even for a non-commutative monad m , provided that f and g themselves commute. In fact, that conjecture is false; it is not too difficult to construct a counterexample using the monad $\text{State } [a]$.)

7 Reasoning about backwards traversal

On reflection, so to speak, it is not surprising that the above attempts do not work: surely the inverse of an effectful traversal over a data structure would not be another traversal with cancelling effects, but rather a *reverse* traversal with cancelling effects?

It seems that the following property ought to hold of idiomatic traversals. Suppose we have two effectful computations $f :: a \rightarrow m b$ and $g :: b \rightarrow \tilde{m} c$. In general, there will still be no fusion rule relating composite traversals of the form $\text{recurse}_m g \bullet \text{traverse}_m f$ and traversal with the composite $\text{traverse}_m (g \bullet f)$ —the former yields an overall effect partitioned into g -effects and f -effects, whereas the latter interleaves the g - and f -effects, and these are different when the effects do not commute. (Besides, such a law would be suspicious on grounds of symmetry alone: why traverse_m rather than recurse_m for the composite traversal?) However, suppose that the effects of the two bodies cancel each other out:

$$g \bullet f = \text{return}_m = \text{return}_{\tilde{m}}$$

(Recall that m and \tilde{m} are the same type, so $g \bullet f$ is well-typed.) Then we assert as another axiom that

$$\text{recurse}_m g \bullet \text{traverse}_m f = \text{traverse}_m (g \bullet f)$$

(which in turn equals $\text{traverse}_m \text{return}_m$ and hence just return_m). This looks much more promising, not least because it is symmetric in m and \tilde{m} .

Given the reversion axiom

$$\text{recurse}_m g \bullet \text{traverse}_m f = \text{return}_m \iff g \bullet f = \text{return}_m$$

it is straightforward to demonstrate the correctness of tree labelling. The body of a traversal to unlabel a tree is still given by *strip*; however, the traversal should be conducted backwards, rather forwards:

$unlabel :: Tree (a, b) \rightarrow State [b] (Tree a)$
 $unlabel = recurse strip$

Then we have:

$unlabel \bullet label$
 $= \llbracket \text{definitions} \rrbracket$
 $recurse strip \bullet traverse adorn$
 $= \llbracket strip \bullet adorn = return \rrbracket$
 $return$

as required.

8 Reasoning about linear traversal

We now turn to an alternative approach, based on an idea by Ondřej Rypáček [10]. Let us forget about reasoning about a traversal in terms of its inverse, and think instead about relating traversals over different datatypes. To that end, in this section we annotate $traverse$ also to indicate the datatype involved, so that

$traverse_m^t :: (Idiom m, Traversable t) \Rightarrow (a \rightarrow m b) \rightarrow t a \rightarrow m (t b)$

But the choice of idiom is less important here, so often we omit the subscript.

8.1 Traversal is (in some sense) natural in the datatype

Rypáček's idea boils down to a kind of naturality in the datatype being traversed, orthogonal to our earlier condition about naturality in the idiom. Straight naturality in the datatype would assert that a natural transformation $\psi : t \rightarrow u$ between traversable functors t and u induces a corresponding relationship between $dist^t$ and $dist^u$, and similarly between $traverse^t$ and $traverse^u$:

$$\begin{array}{ccc}
 t (m b) & \xrightarrow{dist^t} & m (t b) \\
 \psi \downarrow & & \downarrow m \psi \\
 u (m b) & \xrightarrow{dist^u} & m (u b)
 \end{array}
 \qquad
 \begin{array}{ccc}
 t a & \xrightarrow{traverse^t f} & m (t b) \\
 \psi \downarrow & & \downarrow m \psi \\
 u a & \xrightarrow{traverse^u f} & m (u b)
 \end{array}$$

(For brevity, we have followed the categorist's convention of writing just the functor m in place of its map operation $fmap_m$ in these diagrams.) This is clearly too strong a condition: the traversal strategy determines an element ordering on the elements of a t -structure, which ψ need not respect. Besides, ψ may drop or duplicate elements, which will induce dropped or duplicated effects arising from $traverse^u$ when compared to $traverse^t$.

Rather, the naturality condition should be asserted only for natural transformations ψ that do not reorder, drop, or duplicate elements. We say that $\psi : t \rightarrow u$ *preserves contents* if:

$$\text{contents}^u \circ \psi = \text{contents}^t$$

where *contents* is traversal in the monoidal idiom arising from the list monoid:

$$\begin{aligned} \text{contents}^t &:: \text{Traversable } t \Rightarrow t \ a \rightarrow [a] \\ \text{contents}^t &= \text{unK} \circ \text{traverse}^t (\lambda a \rightarrow K [a]) \end{aligned}$$

Then we impose on traversal the axiom that it respects contents-preserving natural transformations: for $\psi : t \rightarrow u$ that preserves contents, we require that

$$\text{fmap}_m \psi \circ \text{traverse}^t f = \text{traverse}^u f \circ \psi$$

In particular, *contents* itself is the canonical contents-preserving ψ —provided that we insist that traversals of lists are in left-to-right order:

$$\begin{aligned} \text{traverse}^{[]} f [] &= \text{pure} [] \\ \text{traverse}^{[]} f (x : xs) &= \text{pure } (:) \otimes f \ x \otimes \text{traverse}^{[]} f \ xs \end{aligned}$$

(There are many other possible orders in which lists may be traversed, but this left-to-right traversal is special: it is the only one for which $\text{contents}^{[]} \circ \text{contents}^t = \text{contents}^t$, that is, for which *contents* is itself contents-preserving.) Then respect for this particular contents-preserving transformation implies a ‘naturality in the contents’ axiom, expressed by letting $\psi = \text{contents}$ and $u = []$ in the commuting squares above. Informally, the effects of $\text{traverse}^t f$ are independent of the shape t of the data structure being traversed: clearly, contents^t discards that shape, and yet $\text{traverse}^{[]} f$ can still deliver the same effects.

One might say that the essence of idiomatic traversal for a particular datatype is the ordering that it imposes on the elements; other properties, such as that every element is visited, and that the shape of the data structure remains unchanged, are consequences of the various healthiness conditions discussed in Section 4. Therefore, the interesting aspect of reasoning about idiomatic traversals is precisely the ordering of effects; and to study this, it suffices to consider linear traversals over lists.

8.2 Shape and contents

The reasonableness of respect for contents-preserving transformations as an axiom of *traverse* can be seen most clearly by actually viewing traversable datatypes in terms of their shape and contents. We have seen the latter already, as the function *contents*; the former is obtained simply by discarding the elements:

$$\begin{aligned} \text{shape} &:: \text{Functor } t \Rightarrow t \ a \rightarrow t \ () \\ \text{shape} &= \text{fmap} (\text{const } ()) \end{aligned}$$

Now, a traversable data structure is completely determined by its shape and contents. A total function of type $t\ a \rightarrow (t\ (), [a])$ can be constructed from the functions *shape* and *contents*; its right inverse *populate* is a partial function, and can be expressed as another traversal in the state monad (called *reassemble* in [3]). A dependently typed setting allows us to make this relationship precise at the type level, in a way that Hindley-Milner does not—for traversable t , the type $t\ a$ is exactly isomorphic to the dependent product $\Sigma\ (s :: t\ ())$. *Vector* (*size* s) a , consisting of pairs (s, xs) with a shape $s :: t\ ()$ and a vector of elements xs of length precisely *size* s , where *size* is computed by a traversal in the monoidal idiom of integers with addition:

$$\begin{aligned} \text{size} &:: \text{Traversable } t \Rightarrow t\ a \rightarrow \text{Int} \\ \text{size} &= \text{unK} \circ \text{traverse } (K \circ \text{const } 1) \end{aligned}$$

Given such a representation of traversable datatypes in terms of shape and contents, it is clear that *traverse* f operates solely on the sequence of elements in the contents (preserving the length of this sequence, and without interfering with the shape), and a contents-preserving transformation ψ operates solely on the shape (maintaining its size, and without interfering with the contents)—and so of course *traverse* f and ψ should commute:

$$\begin{aligned} &fmap_m\ \psi\ (\text{traverse}^t\ f\ (\text{populate } s\ xs)) \\ &= fmap_m\ (\text{populate } (\psi\ s))\ (\text{traverse}^u\ f\ xs) \end{aligned}$$

8.3 Back to tree relabelling

The upshot of this property of respect for contents-preserving transformations is that correctness of labelling on an arbitrary data structure reduces to correctness of labelling on lists. We can express the latter more concretely as follows:

$$\begin{aligned} \text{traverse}^{\square}\ \text{adorn } xs &= \text{State } (\lambda bs \rightarrow (\text{zip } xs\ (\text{take } n\ bs), \text{drop } n\ bs)) \\ &\quad \mathbf{where } n = \text{length } xs \end{aligned}$$

In words, labelling a list xs with elements drawn from an infinite stream bs amounts to pairing the n list elements with the first n labels from bs , in the same order, and returning the remaining labels. This condition is straightforward to prove, since it is specialized to lists; in fact both sides are folds, so the following proof uses the universal property of folds.

The explicit definition of list traversal can be written as a fold:

$$\begin{aligned} \text{traverse}^{\square}\ f &= \text{foldr } (tcons\ f)\ \text{tnil} \\ \mathbf{where } \text{tnil} &= \text{pure } [] \\ tcons\ f\ x\ mxs &= \text{pure } (:) \otimes f\ x \otimes mxs \end{aligned}$$

For monadic idioms, the two components *tnil* and *tcons* specialize:

$$\begin{aligned}
\mathit{tnil} &= \mathit{return} [] \\
\mathit{tcons} f x \mathit{mxs} &= f x \gg \lambda y \rightarrow \mathit{mxs} \gg \lambda ys \rightarrow \mathit{return} (y : ys)
\end{aligned}$$

Now, our claim was that labelling on lists reduces to

$$\begin{aligned}
\mathit{labell} &:: [a] \rightarrow \mathit{State} [b] [(a, b)] \\
\mathit{labell} x &= \mathit{State} (\lambda bs \rightarrow (\mathit{zip} x (\mathit{take} n bs), \mathit{drop} n bs)) \\
&\quad \mathbf{where} \ n = \mathit{length} \ x
\end{aligned}$$

It is straightforward to show that this is a fold; we have

$$\mathit{labell} [] = \mathit{State} (\lambda bs \rightarrow ([], bs)) = \mathit{return} []$$

and

$$\begin{aligned}
&\mathit{labell} (a : x) \\
= &\quad \llbracket \text{specification; let } n = \mathit{length} (a : x) \rrbracket \\
&\quad \mathit{State} (\lambda bs \rightarrow (\mathit{zip} (a : x) (\mathit{take} n bs), \mathit{drop} n bs)) \\
= &\quad \llbracket \text{let } m = \mathit{length} \ x, \text{ so } n = m + 1, \text{ and } bs = b : bs' \rrbracket \\
&\quad \mathit{State} (\lambda (b : bs') \rightarrow \\
&\quad \quad (\mathit{zip} (a : x) (\mathit{take} (m + 1) (b : bs')), \mathit{drop} (m + 1) (b : bs'))) \\
= &\quad \llbracket \text{properties of } \mathit{zip}, \mathit{take}, \mathit{drop} \rrbracket \\
&\quad \mathit{State} (\lambda b : bs' \rightarrow ((a, b) : \mathit{zip} x (\mathit{take} m bs'), \mathit{drop} m bs')) \\
= &\quad \llbracket \mathit{adorn} \rrbracket \\
&\quad \mathit{adorn} a \gg \lambda ab \rightarrow \\
&\quad \quad \mathit{State} (\lambda bs' \rightarrow (ab : \mathit{zip} x (\mathit{take} m bs'), \mathit{drop} m bs')) \\
= &\quad \llbracket \text{monads} \rrbracket \\
&\quad \mathit{adorn} a \gg \lambda ab \rightarrow \\
&\quad \quad \mathit{State} (\lambda bs' \rightarrow (\mathit{zip} x (\mathit{take} m bs'), \mathit{drop} m bs')) \gg \lambda abs \rightarrow \\
&\quad \quad \mathit{return} (ab : abs) \\
= &\quad \llbracket \text{specification} \rrbracket \\
&\quad \mathit{adorn} a \gg \lambda ab \rightarrow \mathit{labell} x \gg \lambda abs \rightarrow \mathit{return} (ab : abs) \\
= &\quad \llbracket \mathit{tcons} \rrbracket \\
&\quad \mathit{tcons} \mathit{adorn} a (\mathit{labell} x)
\end{aligned}$$

and hence

$$\mathit{labell} = \mathit{foldr} (\mathit{tcons} \mathit{adorn}) \mathit{tnil} = \mathit{traverse}^{[1]} \mathit{adorn}$$

as claimed.

8.4 Not all traversals have inverses

The calculation in Section 8.3, demonstrating the correctness of tree relabelling using the ‘respect for contents-preserving transformations’ approach, is longer and shallower than the calculation in Section 7 based on backwards traversal. On the

other hand, the latter approach depends on inverting the original function; this is not always feasible—and even when it is possible, there is no reason in general to believe that reducing a problem about a traversal to one about its inverse makes the problem simpler.

As a case in point, we should confess that from the start we massaged Hutton and Fulger’s presentation of the tree relabelling problem so that it was amenable to reasoning in terms of its inverse. Hutton and Fulger originally presented the problem non-injectively—in terms of stateful computations working on an integer state, and replacing rather than annotating the existing tree labels:

```

labeln :: Tree a → State Int (Tree Int)
labeln = traverseTree fresh
fresh :: a → State Int Int
fresh a = do { n ← get; put (n + 1); return n }

```

Presented in this way, the problem becomes one of showing that

$$\text{runState } (\text{labeln } t) \ m = (u, n) \quad \Rightarrow \quad \text{nodups } (\text{contents } u)$$

(where the predicate *nodups* tests whether a list has no duplicates). But the traversal *labeln* is not invertible, so reasoning tools expressed in terms of backwards traversal are not very helpful. Nevertheless, we can still relate destructively relabelling a tree to similarly relabelling a list; respect for contents-preserving transformations is precisely the property we need. The contents of the relabelled tree is, by fiat, the relabelling of the contents, which is computed by a list traversal; so it suffices to prove

$$\text{runState } (\text{traverse}^{\square} \text{fresh } xs) \ m = (ys, n) \quad \Rightarrow \quad \text{nodups } ys$$

Using the characterization from Section 8.3 of *traverse*[□] as an instance of *foldr*, and the universal property of folds, it is straightforward to show that

$$\text{traverse}^{\square} \text{fresh} = \text{enum}$$

where

$$\begin{aligned} \text{enum } xs &= \mathbf{do} \{ n \leftarrow \text{get}; \text{put } (n + m); \text{return } [n \dots n + m - 1] \} \\ &\quad \mathbf{where} \ m = \text{length } xs \end{aligned}$$

Moreover, the return value $[n \dots n + m - 1]$ patently has no duplicates.

9 Discussion

We have shown two proofs of correctness of a simple effectful program operating on a recursive data structure, using techniques that are modular in both the nature of the effects and the shape of the recursive datatype. We were somewhat surprised

that the correctness arguments did not simply fall out straight away; evidently they depend on properties of traversal that were not immediately obvious to us (given that we overlooked them in earlier related work [3]). It remains to be seen whether these properties are consequences of some deeper structure that is yet to be discovered.

In particular, the connection between traversal of data structures and the factorization of datatypes into shape and contents is quite intriguing, and one we believe that warrants further investigation. The same factorization was found in the 1990s to be very useful in structured parallel programming [2, 7], and more recently as a basis for datatype-generic programming [1]. We hope to explore this connection further.

Of course, not all effectful programs are instances of *traverse*, so the techniques we have discussed in this paper will not always be applicable. Nevertheless, it seems to us that it is important to exploit whatever patterns of recursion we find.

Similarly, not all properties of effectful programs can be analyzed independently of the nature of the effects; sometimes the specific effect is important, and must be taken into account. But still, we feel that it is important to distinguish the general from the particular, and worthwhile to see what can be said about the general case. (Together with Ralf Hinze, we are currently writing another paper [4] about what else can be said about particular classes of effects.)

Acknowledgements

We are grateful to Ondřej Rypáček for sharing with us the idea of reasoning about traversals over arbitrary traversable data structures in terms of the corresponding traversals over lists, and to Conor McBride for showing us the dependently typed representation of traversable datatypes. Discussions with Shin-Cheng Mu and with the Algebra of Programming research group at Oxford were also extremely helpful.

References

- [1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In A. Gordon, editor, *FOSSACS*, number 2620 in LNCS, pages 23–38. Springer-Verlag, 2003.
- [2] C. Banger. Arrays with categorical type constructors. In G. Hains and L. Mullin, editors, *ATABLE-92*, 1992.
- [3] J. Gibbons and B. C. dos Santos Oliveira. The essence of the Iterator pattern. *J. Funct. Prog.*, 19(3,4):377–402, 2009.
- [4] J. Gibbons and R. Hinze. Just do it: Simple monadic equational reasoning. Work in progress, Mar. 2011.
- [5] J. Hughes. Generalising monads to arrows. *Sci. Comput. Prog.*, 37:67–111, 2000.

- [6] G. Hutton and D. Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *TFP*, Nijmegen, The Netherlands, May 2008.
- [7] C. B. Jay. Shape in computing. *Comput. Surveys*, 28(2), 1996.
- [8] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Prog.*, 18(1):1–13, 2008.
- [9] E. Moggi. Notions of computation and monads. *Inform. & Comput.*, 93(1), 1991.
- [10] O. Rypáček. Labelling polynomial functors: A coherent approach. Manuscript, Mar. 2010.