Be Kind, Rewind

A Modest Proposal about Traversal

(FUNCTIONAL PEARL)

Jeremy Gibbons and Richard Bird

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road
Oxford OX1 3QD, United Kingdom
http://www.cs.ox.ac.uk/{jeremy.gibbons,richard.bird}/

Abstract

A recent paper by Graham Hutton and Diana Fulger ('Reasoning about Effects: Seeing the Wood through the Trees', in the preproceedings of *Trends in Functional Programming*, 2008) addresses the problem of reasoning about effectful functional programs, introducing a relabelling function on binary trees as a representative illustration. The example is a very fruitful one, but 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 an alternative approach using properties of idiomatic traversals, which cleanly separate the twin concerns of the kinds of effect and the pattern of recursion. In particular, we approach the problem by considering its inverse; and we argue that this is an important approach which has so far been missing from discussions of idiomatic traversal.

Life can only be understood backwards;
but it must be lived forwards.
— *Søren Kierkegaard*

1. Introduction

How does the presence of effects change our ability to reason about functional programs? The question has been around for years, but if you add 'equationally' after 'reasoning' and 'lazy' before 'functional', there doesn't seem to have been much written that will help Jane Programmer in her daily task of constructing Haskell code. One answer to the question is to say that it shouldn't. Instead the advice is to reason about pure functional programs, and then mechanically translate the results into impure ones, presumably for efficiency reasons. But that only works for programs that have pure counterparts. Another answer is given in a recent paper by Hutton and Fulger [5]. They study a deliberately simple example, that of labelling binary trees. Their objective was to find some way of reasoning that the labelling, expressed using the state monad, generates distinct labels.

While we appreciate that one good example is worth 1000 theories, we believe that Hutton and Fulger's development is somewhat unsatisfactory, and for two separate 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 in 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, that don't have pure equivalents. It's the same criticism that we gave for the first answer.

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'.

We agree. Our intention is to 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 labelling 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 major part of the reasoning can be performed independently of the effects, and so is applicable also to other kinds of effect.

2. Tree labelling

Here is the tree datatype in question:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Tip}\ \alpha \mid \mathit{Bin}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$

Hutton and Fulger offer several related labelling functions, but the one we will adopt in the first instance is subtly different, and we will come back to why later on. In our version, tree labelling annotates a tree with additional elements drawn from an infinite stream, the stream being threaded through the computation via the state monad:

$$\begin{aligned}
&\mathit{label} :: \mathit{Tree}\ \alpha \to \mathit{State}\ [\beta]\ (\mathit{Tree}\ (\alpha,\beta))\\
&\mathit{label}\ (\mathit{Tip}\ a) \quad = \textbf{do}\ \{(b:y) \leftarrow \mathit{get}; \mathit{put}\ y; \mathit{return}\ (\mathit{Tip}\ (a,b))\}\\
&\mathit{label}\ (\mathit{Bin}\ u\ v) = \textbf{do}\ \{u' \leftarrow \mathit{label}\ u; v' \leftarrow \mathit{label}\ v; \mathit{return}\ (\mathit{Bin}\ u'\ v')\}
\end{aligned}$$

The essential property that Hutton and Fulger wished to prove is that tree elements are annotated 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. In turn, this is a corollary of the following property: the sequence of labels used to label the tree, when prepended back on to the stream of unused labels, forms the original input stream of labels. The function *labels* extracts the annotations:

$$\begin{aligned}&\mathit{labels} :: \mathit{Tree}\ (\alpha,\beta) \to [\beta]\\ &\mathit{labels}\ (\mathit{Tip}\ (a,b)) = [b]\\ &\mathit{labels}\ (\mathit{Bin}\ u\ v) \quad = \mathit{labels}\ u +\!\!+ \mathit{labels}\ v\end{aligned}$$

The crucial property to be proved is that

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

3. Idiomatic traversals

We argue that the labelling problem is inherently one about *effectful iterations* over data structures; moreover, as far as possible, any reasoning should 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* [7], 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*, also introduced by McBride and Paterson, and studied in more depth in [3]. We briefly introduce those two notions here, and show their immediate connection to the tree labelling problem, before moving on to a review of the laws of idiomatic traversals.

First of all, idioms are functors with some additional structure:

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Idiom}\ m\ \textbf{where}\\ &\quad \mathit{pure} :: \alpha \to m\ \alpha\\ &\quad (\circledast)\ :: m\ (\alpha \to \beta) \to m\ \alpha \to m\ \beta\end{aligned}$$

satisfying four laws:

$$\begin{aligned}&\mathit{pure}\ \mathit{id} \circledast \mathit{ma} &&= \mathit{ma}\\ &\mathit{pure}\ (\circ) \circledast \mathit{mf} \circledast \mathit{mg} \circledast \mathit{ma} &&= \mathit{mf} \circledast (\mathit{mg} \circledast \mathit{ma})\\ &\mathit{pure}\ f \circledast \mathit{pure}\ a &&= \mathit{pure}\ (f\ a)\\ &\mathit{mf} \circledast \mathit{pure}\ a &&= \mathit{pure}\ (\lambda f \to f\ a) \circledast \mathit{mf}\end{aligned}$$

Every monad is an idiom. For example, the state monad is an idiom, with operations defined as follows:

$$\begin{aligned}&\textbf{instance}\ \mathit{Idiom}\ (\mathit{State}\ s)\ \textbf{where}\\ &\quad \mathit{pure}\ a \quad = \mathit{return}\ a\\ &\quad \mathit{mf} \circledast \mathit{ma} = \textbf{do}\ \{f \leftarrow \mathit{mf}; a \leftarrow \mathit{ma}; \mathit{return}\ (f\ a)\}\end{aligned}$$

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). But we could also have defined

$$\mathit{mf} \circledast \mathit{ma} = \textbf{do}\ \{a \leftarrow \mathit{ma}; f \leftarrow \mathit{mf}; \mathit{return}\ (f\ a)\}$$

evaluating arguments before functions. So, monads translate into idioms in two reasonable and dual ways. On the other hand there are idioms that do not arise from monads. 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:

$$\begin{aligned}&\textbf{newtype}\ K\ \alpha\ \beta = K\ \{\mathit{unK} :: \alpha\}\\ &\textbf{instance}\ \mathit{Monoid}\ \alpha \Rightarrow \mathit{Idiom}\ (K\ \alpha)\ \textbf{where}\\ &\quad \mathit{pure}\ a \quad = K\ \mathit{mempty}\\ &\quad K\ a \circledast K\ b = K\ (\mathit{mappend}\ a\ b)\end{aligned}$$

Next, a traversable datatype according to [7] is one that supports two inter-definable functions *traverse* and *dist*. 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 and when the datatype is lists, this reduces to the monadic map *mapM*. The function *dist* takes a data structure of computations to a computation yielding a data structure, chaining together all the effects to distribute the data structure over the idiom. For monadic idioms and lists, this is the function *sequence*.

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \textbf{where}\\ &\quad \mathit{traverse} :: \mathit{Idiom}\ m \Rightarrow (\alpha \to m\ \beta) \to t\ \alpha \to m\ (t\ \beta)\\ &\quad \mathit{traverse}\ f = \mathit{dist} \circ \mathit{fmap}\ f\\ &\quad \mathit{dist} :: \mathit{Idiom}\ m \Rightarrow t\ (m\ \alpha) \to m\ (t\ \alpha)\\ &\quad \mathit{dist} = \mathit{traverse}\ \mathit{id}\end{aligned}$$

For example, trees form a traversable datatype; traversal of a tip involves visiting its label, and one possible traversal of a binary node involves traversing the left subtree before the right:

$$\begin{aligned}&\textbf{instance}\ \mathit{Traversable}\ \mathit{Tree}\ \textbf{where}\\ &\quad \mathit{traverse}\ f\ (\mathit{Tip}\ a) \quad = \mathit{pure}\ \mathit{Tip} \circledast f\ a\\ &\quad \mathit{traverse}\ f\ (\mathit{Bin}\ u\ v) = \mathit{pure}\ \mathit{Bin} \circledast \mathit{traverse}\ f\ u \circledast \mathit{traverse}\ f\ v\end{aligned}$$

Another choice is to traverse the right subtree before the left one. And these are not the only two possible definitions of *traverse*. We will come back to this important point later on. Tree labelling can be formulated as a tree traversal, using the monadic idiom arising from the state monad:

$$\begin{aligned}&\mathit{label} :: \mathit{Tree}\ \alpha \to \mathit{State}\ [\beta]\ (\mathit{Tree}\ (\alpha,\beta))\\ &\mathit{label} = \mathit{traverse}\ \mathit{adorn}\end{aligned}$$

The body of the traversal consumes a single label from the stream, using it to adorn a single tree element:

$$\begin{aligned}&\mathit{adorn} :: \alpha \to \mathit{State}\ [\beta]\ (\alpha,\beta)\\ &\mathit{adorn}\ a = \textbf{do}\ \{(b : y) \leftarrow \mathit{get}; \mathit{put}\ y; \mathit{return}\ (a,b)\}\end{aligned}$$

The crunch question is: Can we reason about labelling simply by reasoning about the properties of *traverse*? The answer is a qualified 'yes', but we won't mention the qualification just yet.

4. Properties of idioms and traversal

We briefly remind ourselves of some properties of idioms and of idiomatic traversal, following [7] and [3]. First off, idioms compose nicely, in the exactly the way monads don't:

$$\begin{aligned}&\textbf{data}\ C\ m\ n\ \alpha = C\ \{\mathit{unC} :: m\ (n\ \alpha)\}\\ &\textbf{instance}\ (\mathit{Idiom}\ m, \mathit{Idiom}\ n) \Rightarrow \mathit{Idiom}\ (C\ m\ n)\ \textbf{where}\\ &\quad \mathit{pure}\ a \quad = C\ (\mathit{pure}\ (\mathit{pure}\ a))\\ &\quad C\ \mathit{mnf} \circledast C\ \mathit{mna} = C\ (\mathit{pure}\ (\circledast) \circledast \mathit{mnf} \circledast \mathit{mna})\end{aligned}$$

We can introduce a composition operator for idiomatic computations:

$$\begin{aligned}&(\odot) :: (\mathit{Idiom}\ m, \mathit{Idiom}\ n) \Rightarrow\\ &\quad (\beta \to n\ \gamma) \to (\alpha \to m\ \beta) \to \alpha \to C\ m\ n\ \gamma\\ &g \odot f = C \circ \mathit{fmap}\ g \circ f\end{aligned}$$

Of course, there's an identity idiom too:

$$\begin{aligned}&\textbf{newtype}\ I\ \alpha = I\ \{\mathit{unI} :: \alpha\}\\ &\textbf{instance}\ \mathit{Idiom}\ I\ \textbf{where}\\ &\quad \mathit{pure}\ a \quad = I\ a\\ &\quad I\ f \circledast I\ a = I\ (f\ a)\end{aligned}$$

We require traversal to respect the compositional structure of idioms. That is to say, for a pure function $f :: \alpha \to \beta$ we require that

$$\mathit{traverse}\ (I \circ f) = I \circ \mathit{fmap}\ f$$

and, for effectful bodies $f :: \alpha \to m\ \beta$ and $g :: \beta \to n\ \gamma$, that

$$traverse\ (C \circ fmap\ g \circ f) = C \circ fmap\ (traverse\ g) \circ traverse\ f$$

Using idiom composition, the latter equation is equivalent to

$$traverse\ (g \odot f) = traverse\ g \odot traverse\ f$$

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

The next property is that traversal should be natural in the idiom. By definition, an *idiom morphism* $\phi : m \dot{\to} n$ from idiom m to idiom n is a polymorphic function $\phi : m\ \alpha \to n\ \alpha$ such that

$$\begin{aligned}&\phi\ (pure\ a) &&= pure\ a\\ &\phi\ (mf \circledast ma) &&= (\phi\ mf) \circledast (\phi\ ma)\end{aligned}$$

Given an idiom morphism ϕ we expect the two laws

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

to hold (each of which implies the other). In particular, $pure \circ unI : I \dot{\to} m$ is an idiom morphism; as a consequence, we get the purity law that traversal with *pure* is itself just *pure*:

$$traverse\ pure = pure$$

Also, when m is the idiom arising from a commutative monad and $join :: m\ (m\ \alpha) \to \alpha$ is monad multiplication, we have that $join \circ unC : C\ m\ m \dot{\to} m$ is an idiom morphism; as a consequence, we get a special fusion law for monadic traversals in a common commutative monad:

$$traverse\ g \bullet traverse\ f = traverse\ (g \bullet f)$$

where $\bullet$ is Kleisli composition:

$$(g \bullet f)\ a = \mathbf{do}\ \{b \leftarrow f\ a; g\ b\}$$

5. Composing idiomatic traversals

How might we set about proving the property at the end of Section 2 stating the correctness of tree labelling? The first observation is that 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.

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:

$$\begin{aligned}&strip :: (\alpha, \beta) \to State\ [\beta]\ \alpha\\ &strip\ (a, b) = \mathbf{do}\ \{y \leftarrow get; put\ (b : y); return\ a\}\end{aligned}$$

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

$$\begin{aligned}&unlabel :: Tree\ (\alpha, \beta) \to State\ [\beta]\ (Tree\ \alpha)\\ &unlabel = traverse\ strip\end{aligned}$$

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 in some sense each other's inverses. More precisely, suppose that m and n are idioms, and that $f :: \alpha \to m\ \beta$ and $g :: \beta \to n\ \alpha$. We can reason

$$\begin{aligned}&traverse\ g \odot traverse\ f\\ =\ &\llbracket\ \text{traversal respects composition}\ \rrbracket\\ &traverse\ (g \odot f)\\ =\ &\llbracket\ \text{suppose } g \odot f = pure\ \rrbracket\\ &traverse\ pure\\ =\ &\llbracket\ \text{purity law}\ \rrbracket\\ &pure\end{aligned}$$

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

$$strip \odot adorn = pure$$

But this property doesn't hold; the two sides are observably different. It turns out that $\odot$ is a rather odd kind of 'translucent composition': in the composition $g \odot f$, the intermediate value returned by f is hidden, but the effects of f are not. (Concretely, defining observation function h by

$$h\ mna = pure\ (const\ ()) \circledast unC\ mna$$

the reader may confirm that

$$h\ ((strip \odot adorn)\ a) = \mathbf{do}\ \{(b : y) \leftarrow get; put\ y; return\ ()\}$$

whereas *details are in Appendix A.1*

$$h\ (pure\ a) = return\ ()$$

The two right-hand sides are clearly different, and so $strip \odot adorn$ and *pure* must differ too.)

Let's try again, this time with Kleisli composition and monads. Both *strip* and *adorn* use the same monadic idiom and this time we do have

$$strip \bullet adorn = return$$

Now we can reason (but spot the bug!)

$$\begin{aligned}&traverse\ g \bullet traverse\ f\\ =\ &\llbracket\ \text{traversal respects Kleisli composition}\ \rrbracket\\ &traverse\ (g \bullet f)\\ =\ &\llbracket\ \text{assuming } g \bullet f = return\ \rrbracket\\ &traverse\ return\\ =\ &\llbracket\ \text{purity law}\ \rrbracket\\ &return\end{aligned}$$

Did you spot the bug? Traversal respects Kleisli composition only for *commutative* monads and the state monad is the epitome of non-commutativity. (Incidentally, Gibbons and Oliveira [3] conjectured that traversal respects Kleisli composition *see Appendix A.2*

$$traverse\ g \bullet traverse\ f = traverse\ (g \bullet f)$$

even for a non-commutative monad, provided that f and g themselves commute, but that conjecture is false. It is not too difficult to construct a counterexample with the state monad.)

6. Knowing it forwards and backwards

Perhaps the reader has been ahead of us, and seen already that the two attempts above cannot possibly work. Both *label* and *unlabel* traverse the tree in the same direction, and so the supposedly inverse effects of *unlabel* take place in the wrong order, and the two functions can't possibly be inverse. After all, if you put your socks and then your shoes in in the morning, then in the evening you take off your shoes and then your socks.

Another property enjoyed by idioms and not in general by monads is that, for each idiom m, there is a corresponding 'backwards' idiom $B\ m$ with the effects sequenced in the opposite order.

$$\mathbf{newtype}\ B\ m\ \alpha = B\ \{unB :: m\ \alpha\}$$

$$\begin{aligned}&\mathbf{instance}\ Idiom\ m \Rightarrow Idiom\ (B\ m)\ \mathbf{where}\\ &\quad pure\ a &&= B\ (pure\ a)\\ &\quad B\ mf \circledast B\ ma &&= B\ (pure\ (flip\ (\$)) \circledast ma \circledast mf)\end{aligned}$$

Reversing the order of effects allows us to define backwards traversal (one meaning of 'recurse' is 'go backwards', from the Latin 'recurrere'):

$$\begin{aligned}&\mathit{recurse} :: (\mathit{Idiom}\ m, \mathit{Traversable}\ t) \Rightarrow (\alpha \to m\ \beta) \to t\ \alpha \to m\ (t\ \beta)\\&\mathit{recurse}\ f = \mathit{unB} \circ \mathit{traverse}\ (B \circ f)\end{aligned}$$

In other words, *recurse* is an instance *traverse*, visiting every element of a data structure, but collecting the effects in the opposite order; for example, with the earlier definition of *traverse* on binary trees, we have:

proof is in Figure A.1

$$\mathit{recurse}\ f\ (\mathit{Bin}\ u\ v) = \mathit{pure}\ (\mathit{flip}\ \mathit{Bin}) \circledast \mathit{recurse}\ f\ v \circledast \mathit{recurse}\ f\ u$$

In particular, we can define:

$$\mathit{unlabel} = \mathit{recurse}\ \mathit{strip}$$

Of course, by definition $B : m \dot{\to} B\ m$ is a natural transformation between two idioms. But it is not an idiom morphism; there is in general no relationship between $B\ \mathit{mf} \circledast B\ \mathit{ma}$ and $B\ (\mathit{mf} \circledast \mathit{ma})$. However, the composition $B \circ B : m \dot{\to} m$ is an idiom morphism, expressing the observation that reversal of the order of effects is an involution. As a consequence, we can derive a dual characterization of *traverse* in terms of *reverse*:

proof is in Figure A.2

$$\begin{aligned}&\mathit{traverse}\ f\\=\quad&[\![\ B \text{ is an isomorphism}\ ]\!]\\&\mathit{unB} \circ \mathit{unB} \circ B \circ B \circ \mathit{traverse}\ f\\=\quad&[\![\ B \circ B \text{ is an idiom morphism}\ ]\!]\\&\mathit{unB} \circ \mathit{unB} \circ \mathit{traverse}\ (B \circ B \circ f)\\=\quad&[\![\ \text{definition of } \mathit{recurse}\ ]\!]\\&\mathit{unB} \circ \mathit{recurse}\ (B \circ f)\end{aligned}$$

Now, we can say something about Kleisli compositions of *traverse* and *recurse*. We might hope for a general composition law of the form

$$\mathit{recurse}\ f \bullet \mathit{traverse}\ g = \mathit{traverse}\ (f \bullet g) \quad \text{-- invalid!}$$

but such a hope is still forlorn—the effects of f happen in opposite orders on the two sides of the equation. Besides, this law is suspicious on grounds of symmetry alone: why *traverse* on the right-hand side, rather than *recurse*? However, we can impose the law in the rather special case that the traversal bodies are inverse, that is, $f \bullet g = \mathit{return}$. Then the asymmetry disappears, because of the purity law:

$$\mathit{traverse}\ \mathit{return} = \mathit{return} = \mathit{recurse}\ \mathit{return}$$

That is, we expect $\mathit{recurse}\ f$ and $\mathit{traverse}\ g$ to be inverses whenever f and g are:

$$f \bullet g = \mathit{return} \quad \Rightarrow \quad \mathit{recurse}\ f \bullet \mathit{traverse}\ g = \mathit{return}$$

We impose this inverse traversal law as an additional axiom; then we have

$$\begin{aligned}&\mathit{unlabel} \bullet \mathit{label}\\=\quad&[\![\ \text{definitions}\ ]\!]\\&\mathit{recurse}\ \mathit{strip} \bullet \mathit{traverse}\ \mathit{adorn}\\=\quad&[\![\ \text{given that } \mathit{strip} \bullet \mathit{adorn} = \mathit{return}\ ]\!]\\&\mathit{return}\end{aligned}$$

Finally our proof obligation is complete, and everything in the garden is rosy.

Let's now return to a point made in Section 2 about tree labelling, and confess that our version of the problem is not the one Hutton and Fulger actually chose. The calculation above, brief and attractive as it is, depends on being able to invert the original function. What if the function is not invertible? For instance, Hutton and Fulger originally presented the labelling problem non-injectively—in terms of stateful computations working on an integer state, and replacing rather than annotating the existing tree labels:

$$\begin{aligned}&\mathit{labeln} :: \mathit{Tree}\ \alpha \to \mathit{State}\ \mathit{Int}\ (\mathit{Tree}\ \mathit{Int})\\&\mathit{labeln} = \mathit{traverse}\ \mathit{fresh}\\&\\&\mathit{fresh} :: \alpha \to \mathit{State}\ \mathit{Int}\ \mathit{Int}\\&\mathit{fresh}\ a = \mathbf{do}\ \{n \leftarrow \mathit{get}; \mathit{put}\ (n+1); \mathit{return}\ n\}\end{aligned}$$

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

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

where the predicate *nodups* tests whether a list has no duplicates, and *contents* is a traversal in the monoidal idiom arising from the list monad:

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

But the traversal *labeln* is not invertible, so at first blush reasoning expressed in terms of backwards traversal does not seem to be very helpful. All is not lost, however, because we can redefine *labeln* so that it becomes invertible. Define new versions of *adorn* and *strip* by

$$\begin{aligned}&\mathit{adorn} :: \alpha \to \mathit{State}\ ([\alpha], [\beta])\ \beta\\&\mathit{adorn}\ a = \mathbf{do}\ \{(x, b : y) \leftarrow \mathit{get}; \mathit{put}\ (a : x, y); \mathit{return}\ b\}\\&\mathit{strip} :: \beta \to \mathit{State}\ ([\alpha], [\beta])\ \alpha\\&\mathit{strip}\ b\ \ = \mathbf{do}\ \{(a : x, y) \leftarrow \mathit{get}; \mathit{put}\ (x, b : y); \mathit{return}\ a\}\end{aligned}$$

The function *adorn* replaces the original labels with new ones but remembers the old ones, while *strip* does the reverse. We have $\mathit{strip} \bullet \mathit{adorn} = \mathit{return}$, as before. Secondly, define new versions of *label* and *unlabel* by

$$\begin{aligned}&\mathit{label}\ \ = \mathit{traverse}\ \mathit{adorn}\\&\mathit{unlabel} = \mathit{recurse}\ \mathit{strip}\end{aligned}$$

Then, as before, $\mathit{unlabel} \bullet \mathit{label} = \mathit{return}$. Moreover,

$$\mathit{runState}\ (\mathit{labeln}\ t)\ m = \mathit{runState}\ (\mathit{label}\ t)\ ([\,], [m\,..])$$

and our reasoning remains intact.

7. Discussion

We have shown a proof 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 (indeed, we overlooked them in earlier related work [3]).

The core of our proof is the relationship between forwards and backwards traversal. The two functions *traverse* and *recurse* form a very natural pair. Moreover, their interaction does not seem to us at present to be something that arises from properties of *traverse* alone; it has to be stated as a separate axiom. We therefore propose that *recurse* should be elevated to first-class status as a third method in the *Traversable* class—following the principle that the most appropriate place in which to state an axiom constraining implementations of an abstract operation is on the type class of which the operation is a member. The *recurse* operation can be given a default definition in terms of *traverse*, so this is no additional burden on the programmer; but if an independent definition is given, it is the programmer's responsibility to ensure that the law $B \circ \mathit{recurse}\ f = \mathit{traverse}\ (B \circ f)$ is satisfied.

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. The curious reader may wish to read our related paper [4] about what more can be said about other patterns of control, and specific classes of effects.

There is also an intriguing connection with the factorization of datatypes into shape and contents, as in Jay's notion of 'shapely over lists' datatypes [6] and Abbott's 'container datatypes' [1]. We already saw in Section 5 a definition of *contents* in terms of traversal in the constantly-lists idiom:

$$contents^t :: Traversable\ t \Rightarrow t\ \alpha \to [\alpha]$$

(explicitly annotating with a superscript to denote the particular traversable datatype). Defining *shape* is even easier: it is a traversal in the identity idiom, that is, a map:

$$shape^t :: Functor\ t \Rightarrow t\ \alpha \to t\ ()$$
$$shape^t = fmap\ (const\ ())$$

Rypáček [8] has observed that traversal should be independent of the shape of a data structure, in a certain sense: if the natural transformation $\psi : t \dot{\to} u$ preserves contents:

$$contents^u \circ \psi = contents^t$$

then traversal respects ψ:

$$fmap_m\ \psi \circ traverse^t\ f = traverse^u\ f \circ \psi$$

Assuming this additional axiom, one can reason about tree relabelling in terms of simpler list relabelling [2]. What is more, we can specify datatype-generic reversal $reverse^t$ in terms of shape and contents—the reverse of a data structure has the same shape and reversed contents:

$$shape^t\ (reverse^t\ x) = shape^t\ x$$
$$contents^t\ (reverse^t\ x) = reverse\ (contents^t\ x)$$

Then we might expect that backwards traversal over the reverse data structure corresponds to forwards traversal over the original:

$$fmap_m\ reverse^t\ (recurse\ f\ x) = traverse\ f\ (reverse^t\ x)$$

It remains to be seen whether the inverse traversal property of *traverse* and *recurse* is a consequence of some deeper structure that is yet to be discovered. In particular, symmetry suggests a dual law

$$f \bullet g = return \quad \Rightarrow \quad traverse\ f \bullet recurse\ g = return$$

We conjecture that this is equivalent to the inverse traversal axiom used in Section 6, but we have not yet managed to prove this fact. Similarly, perhaps the inverse traversal property can be deduced by looking at datatype-generic reversal.

Acknowledgements

We are grateful to Graham Hutton and Diana Fulger for the illuminating example that first inspired this work. Discussions with members of IFIP Working Group 2.1 and of the Algebra of Programming research group at Oxford were also extremely helpful.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew Gordon, editor, *Foundations of Software Science and Computation Structures*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003.

[2] Jeremy Gibbons. Effective reasoning about effectful linear traversals. In *Higher-Order Programming with Effects*, 2012. Talk proposal, under review.

[3] Jeremy Gibbons and Bruno César dos Santos Oliveira. The essence of the Iterator pattern. *Journal of Functional Programming*, 19(3,4):377–402, 2009.

[4] Jeremy Gibbons and Ralf Hinze. Just `do` it: Simple monadic equational reasoning. In *International Conference on Functional Programming*, pages 2–14, September 2011.

[5] Graham Hutton and Diana Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *Trends in Functional Programming*, pre-proceedings, Nijmegen, The Netherlands, May 2008.

[6] C. Barry Jay. Shape in computing. *ACM Computing Surveys*, 28(2), 1996.

[7] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[8] Ondřej Rypáček. Labelling polynomial functors: A coherent approach. Manuscript, March 2010.

Appendix

This appendix contains justifications for some of the assertions made in the body of the paper, as an aid to reviewers. We suggest that it should not be formally published, even if the paper is accepted; but perhaps it should form supplementary materials in the ACM Digital Library. At least then it would provide a venue for one of us (JG) to confess to earlier unsuccessful conjectures [3].

A.1 Translucent composition

In Section 5, we claimed that the idiomatic composition *strip* $\odot$ *adorn* was not equal to simply *pure* in the composite idiom. More generally, $\odot$ is a kind of 'transparent 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, we have:

$$\begin{array}{ll} & \mathit{pure}\ (\mathit{const}\ ()) \circledast \mathit{unC}\ ((g \odot f)\ a) \\ = & \quad [\![\ \text{definition of } \odot;\ \mathit{unC} \circ \mathit{C} = \mathit{id}\ ]\!] \\ & \mathit{pure}\ (\mathit{const}\ ()) \circledast \mathit{fmap}\ g\ (f\ a) \\ = & \quad [\![\ \text{naturality—see below}\ ]\!] \\ & \mathit{pure}\ (\mathit{const}\ () \circ g) \circledast f\ a \\ = & \quad [\![\ \mathit{const}\ ]\!] \\ & \mathit{pure}\ (\mathit{const}\ ()) \circledast f\ a \end{array}$$

The naturality property cited is that

$$\mathit{pure}\ g \circledast \mathit{fmap}\ f\ ma = \mathit{pure}\ (g \circ f) \circledast ma$$

which is obtained by specializing $f = id$, $p = q \circ f'$, and $h = \lambda g \to (\mathit{pure}_m\ g \circledast)$ in the free theorem

$$f \circ p = q \circ f' \quad \Rightarrow \quad \mathit{fmap}_m f \circ h\ p = h\ q \circ \mathit{fmap}_m f'$$

for a function h of type $(\alpha \to \beta) \to (m\ \alpha \to m\ \beta)$. This calculation shows that, 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 $\mathit{pure}_{C\ (\mathit{State}\ [\beta])\ (\mathit{State}\ [\beta])}$—even if *strip* subsequently undoes the effects, the intermediate state betraying the effects is observable.

A.2 Traversal fusion for non-commutative monads

At the end of Section 5, we mentioned the conjecture by Gibbons and Oliveira [3, Section 5.4] that

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

even for a non-commutative monad m, provided that f and g themselves commute. In fact, that conjecture is false, as we show here.

Let $f :: \alpha \to m\ \beta$ and $g :: \beta \to m\ \gamma$. For a counter-example, we first need f, g that commute; for that to make sense, we certainly need $\alpha = \beta = \gamma$. We will use the state monad $m = \mathit{State}\ s$, with s being streams of labels as throughout; we will also make $\alpha = \beta = \gamma = s$ (so that each data structure element is itself a stream of labels). Define:

$$\begin{array}{l} f, g :: [\alpha] \to \mathit{State}\ [\alpha]\ [\alpha] \\ f\ x \quad\ = \mathit{State}\ (\lambda (b : y) \to (b : x, y)) \\ g\ (b : x) = \mathit{State}\ (\lambda y \to (x, b : y)) \end{array}$$

Then clearly

$$f \bullet g = g \bullet f = \mathit{return}$$

as required for the premise. But with the usual left-to-right traversal over pairs, we have

$$\begin{array}{l} \mathit{traverse}\ f\ (x, y) \quad\ = \mathit{State}\ (\lambda (a : b : z) \to ((a : x, b : y), z)) \\ \mathit{traverse}\ g\ (a : x, b : y) = \mathit{State}\ (\lambda z \to ((x, y), b : a : z)) \end{array}$$

Therefore $\mathit{traverse}\ g \bullet \mathit{traverse}\ f$ is not *return*, since it does not leave the state unchanged—it swaps the first two elements of the stream around.

A.3 Linear traversals

We have identified various levels of naturality property for *traverse* (and hence for *dist* and *recurse* too). Of course, traversal should be natural in the elements of the data structure being traversed:

$$\mathit{fmap}_m\ (\mathit{fmap}_t\ h) \circ \mathit{traverse}_m\ f = \mathit{traverse}_m\ (\mathit{fmap}_m\ h \circ f)$$

We also required traversal to be natural in the idiom; for idiom morphism $\phi : m \dot{\to} n$,

$$\phi \circ \mathit{traverse}_m\ f = \mathit{traverse}_n\ (\phi \circ f)$$

Finally, we required traversal to respect the compositional structure of idioms: for a pure function $f :: \alpha \to \beta$ we expect that

$$\mathit{traverse}_I\ (I \circ f) = I \circ \mathit{fmap}_t\ f$$

and, for effectful bodies $f :: \alpha \to m\ \beta$ and $g :: \beta \to n\ \gamma$, that

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

Gibbons and Oliveira [3] showed that these conditions ruled out various kinds of bogus traversal, including traversals that skip elements or that change the shape of the data structure. However, it wasn't clear whether the conditions ruled out traversals that duplicate elements, such as the following bogus traversal of binary trees that visits every tip twice:

```
instance Traversable Tree where
  traverse f (Tip a)   = pure (const Tip) ⊛ f a ⊛ f a
  traverse f (Bin u v) = pure Bin ⊛ traverse f u ⊛ traverse f v
```

It still isn't clear; but recently, Jaskelioff and Rypacek [9] have offered some evidence—albeit not yet a proof—to support the conjecture that all such duplicitous traversals are ruled out by respect for the compositional structure of idioms. They present an example of a duplicitous traversal that does not respect composition, using the list functor three times (traversal of a list, with a body in the composite idiom arising from the list monad composed with itself). A perhaps more perspicuous example uses three different functors: a duplicitous traversal of the identity functor

```
instance Traversable I where
  traverse f (I x) = pure (const I) ⊛ f x ⊛ f x
```

and traversal bodies in the idioms arising from the exception and list monads:

```
f :: Integer → Maybe Integer
f n = if odd n then Just n else Nothing

g :: Integer → [Integer]
g n = [n, n+1]
```

Then we have

$$\begin{array}{l} \mathit{traverse}\ (g \odot f)\ (I\ 0) \\ \quad = C\ [\mathit{Nothing}, \mathit{Nothing}, \mathit{Nothing}, \mathit{Just}\ (I\ 1)] \\ (\mathit{traverse}\ g \odot \mathit{traverse}\ f)\ (I\ 0) \\ \quad = C\ [\mathit{Nothing}, \mathit{Just}\ (I\ 1), \mathit{Nothing}, \mathit{Just}\ (I\ 1)] \end{array}$$

These plainly differ, so this traversal does not respect idiom composition. But it remains an open question whether any such duplicitous traversal will be similarly betrayed.

Additional references

[9] Mauro Jaskelioff and Ondřej Rypáček. An investigation of the laws of traversals. In James Chapman and Paul Blain Levy, editors, *Mathematically Structured Functional Programming*, volume 76 of *EPTCS*, pages 40–49, 2012.

Here, we show that

$$recurse\ f\ (Bin\ u\ v) = pure\ (flip\ Bin) \circledast recurse\ f\ v \circledast recurse\ f\ u$$

given the natural definition of *traverse* for binary trees, and the characterization of *recurse* in terms of *traverse*:

$$\begin{aligned} traverse\ f\ (Bin\ u\ v) &= pure\ Bin \circledast traverse\ f\ u \circledast traverse\ f\ v \\ recurse\ f &= unB \circ traverse\ (B \circ f) \end{aligned}$$

To be explicit, we write '$pure_B$' and '$\circledast_B$' for *pure* and $\circledast$ in the backwards idiom.

$$\begin{aligned}
& recurse\ f\ (Bin\ u\ v) \\
= \quad & [\![\text{ definition of } recurse \text{ in terms of } traverse \ ]\!] \\
& unB\ (traverse\ (B \circ f)\ (Bin\ u\ v)) \\
= \quad & [\![\text{ definition of } traverse \ ]\!] \\
& unB\ (pure_B\ Bin \circledast_B traverse\ (B \circ f)\ u \circledast_B traverse\ (B \circ f)\ v) \\
= \quad & [\![\textit{pure} \text{ in a backwards idiom; } recurse \text{ in terms of } traverse \ ]\!] \\
& unB\ (B\ (pure\ Bin) \circledast_B B\ (recurse\ f\ u) \circledast_B B\ (recurse\ f\ v)) \\
= \quad & [\![\circledast \text{ in a backwards idiom} \ ]\!] \\
& unB\ ((B\ (pure\ (flip\ (\$)) \circledast recurse\ f\ u \circledast pure\ Bin)) \circledast_B B\ (recurse\ f\ v)) \\
= \quad & [\![\circledast \text{ in a backwards idiom again} \ ]\!] \\
& unB\ (B\ (pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast (pure\ (flip\ (\$)) \circledast recurse\ f\ u \circledast pure\ Bin))) \\
= \quad & [\![\ unB \circ B = id \ ]\!] \\
& pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast (pure\ (flip\ (\$)) \circledast recurse\ f\ u \circledast pure\ Bin) \\
= \quad & [\![\text{ idiom interchange} \ ]\!] \\
& pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast (pure\ (\$Bin) \circledast (pure\ (flip\ (\$)) \circledast recurse\ f\ u)) \\
= \quad & [\![\text{ idiom composition} \ ]\!] \\
& pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast (pure\ (\circ) \circledast pure\ (\$Bin) \circledast pure\ (flip\ (\$)) \circledast recurse\ f\ u) \\
= \quad & [\![\text{ idiom homomorphism} \ ]\!] \\
& pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast (pure\ ((\circ)\ (\$Bin)\ (flip\ (\$))) \circledast recurse\ f\ u) \\
= \quad & [\![\text{ sectioning} \ ]\!] \\
& pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast (pure\ ((\$Bin) \circ flip\ (\$)) \circledast recurse\ f\ u) \\
= \quad & [\![\text{ idiom composition} \ ]\!] \\
& pure\ (\circ) \circledast (pure\ (flip\ (\$)) \circledast recurse\ f\ v) \circledast pure\ ((\$Bin) \circ flip\ (\$)) \circledast recurse\ f\ u \\
= \quad & [\![\text{ idiom composition} \ ]\!] \\
& pure\ (\circ) \circledast pure\ (\circ) \circledast pure\ (flip\ (\$)) \circledast recurse\ f\ v \circledast pure\ ((\$Bin) \circ flip\ (\$)) \circledast recurse\ f\ u \\
= \quad & [\![\text{ idiom homomorphism} \ ]\!] \\
& pure\ ((\circ)\ (\circ)\ (flip\ (\$))) \circledast recurse\ f\ v \circledast pure\ ((\$Bin) \circ flip\ (\$)) \circledast recurse\ f\ u \\
= \quad & [\![\text{ sectioning} \ ]\!] \\
& pure\ ((\circ) \circ flip\ (\$)) \circledast recurse\ f\ v \circledast pure\ ((\$Bin) \circ flip\ (\$)) \circledast recurse\ f\ u \\
= \quad & [\![\text{ idiom interchange} \ ]\!] \\
& pure\ (\$((\$Bin) \circ flip\ (\$))) \circledast (pure\ ((\circ) \circ flip\ (\$)) \circledast recurse\ f\ v) \circledast recurse\ f\ u \\
= \quad & [\![\text{ idiom composition} \ ]\!] \\
& pure\ (\circ) \circledast pure\ (\$((\$Bin) \circ flip\ (\$))) \circledast pure\ ((\circ) \circ flip\ (\$)) \circledast recurse\ f\ v \circledast recurse\ f\ u \\
= \quad & [\![\text{ idiom homomorphism} \ ]\!] \\
& pure\ ((\circ)\ (\$((\$Bin) \circ flip\ (\$)))\ ((\circ) \circ flip\ (\$))) \circledast recurse\ f\ v \circledast recurse\ f\ u \\
= \quad & [\![\text{ sectioning} \ ]\!] \\
& pure\ ((\$((\$Bin) \circ flip\ (\$))) \circ (\circ) \circ flip\ (\$)) \circledast recurse\ f\ v \circledast recurse\ f\ u \\
= \quad & [\![\text{ combinators: } (\$((\$h) \circ flip\ (\$))) \circ (\circ) \circ flip\ (\$) = flip\ h \ ]\!] \\
& pure\ (flip\ Bin) \circledast recurse\ f\ v \circledast recurse\ f\ u
\end{aligned}$$

The final step with the hint 'combinators' just depends on the definitions of \$ and *flip*: for any h,

$$\begin{aligned}
& ((\$((\$h) \circ flip\ (\$))) \circ (\circ) \circ flip\ (\$))\ x\ y = (\$((\$h) \circ flip\ (\$)))\ ((\circ)\ (\$x))\ y = (\circ)\ (\$x)\ ((\$h) \circ flip\ (\$))\ y = ((\$x) \circ ((\$h) \circ flip\ (\$)))\ y \\
= \ & (\$x)\ (((\$h) \circ flip\ (\$))\ y) = ((\$h) \circ flip\ (\$))\ y\ x = (\$h)\ (flip\ (\$)\ y)\ x = (\$h)\ (\$y)\ x = (\$y)\ h\ x = h\ y\ x
\end{aligned}$$

Figure A.1. Proof of the claim in Section 6 that *recurse* collects the effects of *traverse* in the opposite order.

Here, we show the interesting case of the proof that $B \circ B$ is an idiom morphism. For brevity, define $BB = B \circ B$, so that $BB\ a = B\ (B\ a)$; we will also use B and BB as subscripts on idiomatic operations.

$$
\begin{array}{ll}
 & BB\ mf \circledast_{BB} BB\ ma \\
= & \quad [\![\ \circledast_{BB}\ ]\!] \\
 & B\ ((pure_B\ (flip\ (\$)) \circledast_B B\ ma) \circledast_B B\ mf) \\
= & \quad [\![\ pure_B\ ]\!] \\
 & B\ ((B\ (pure\ (flip\ (\$))) \circledast_B B\ ma) \circledast_B B\ mf) \\
= & \quad [\![\ \circledast_B\ ]\!] \\
 & B\ (B\ (pure\ (flip\ (\$)) \circledast ma \circledast pure\ (flip\ (\$))) \circledast_B B\ mf) \\
= & \quad [\![\ \text{(1) below: } pure\ (flip\ (\$)) \circledast ma \circledast pure\ (flip\ (\$)) = pure\ (flip\ (\$)) \circledast ma\ ]\!] \\
 & B\ (B\ (pure\ (flip\ (\$)) \circledast ma) \circledast_B B\ mf) \\
= & \quad [\![\ \circledast_B, BB\ ]\!] \\
 & BB\ (pure\ (flip\ (\$)) \circledast mf \circledast (pure\ (flip\ (\$)) \circledast ma)) \\
= & \quad [\![\ \text{(2) below: } (pure\ (flip\ (\$)) \circledast mf) \circledast (pure\ (flip\ (\$)) \circledast ma) = mf \circledast ma\ ]\!] \\
 & BB\ (mf \circledast ma)
\end{array}
$$

Here's subproof (1):

$$
\begin{array}{ll}
 & (pure\ (flip\ (\$)) \circledast ma) \circledast pure\ (flip\ (\$)) \\
= & \quad [\![\ \text{idiom interchange}\ ]\!] \\
 & pure\ (\lambda f \to f\ (flip\ (\$))) \circledast (pure\ (flip\ (\$)) \circledast ma) \\
= & \quad [\![\ \text{idiom composition}\ ]\!] \\
 & pure\ (\circ) \circledast pure\ (\lambda f \to f\ (flip\ (\$))) \circledast pure\ (flip\ (\$) \circledast ma) \\
= & \quad [\![\ \text{idiom homomorphism}\ ]\!] \\
 & pure\ ((\circ)\ (\lambda f \to f\ (flip\ (\$)))\ (flip\ (\$))) \circledast ma \\
= & \quad [\![\ \text{composition}\ ]\!] \\
 & pure\ ((\lambda f \to f\ (flip\ (\$))) \circ (flip\ (\$))) \circledast ma \\
= & \quad [\![\ \text{combinators: } (\lambda f \to f\ (flip\ (\$))) \circ (flip\ (\$)) = flip\ (\$)\ ]\!] \\
 & pure\ (flip\ (\$)) \circledast ma
\end{array}
$$

The 'combinators' step is straightforward:

$$((\lambda f \to f\ (flip\ (\$))) \circ (flip\ (\$)))\ h = (\lambda f \to f\ (flip\ (\$)))\ (flip\ (\$)\ h) = (\lambda f \to f\ (flip\ (\$)))\ (\$h) = (\$h)\ (flip\ (\$)) = flip\ (\$)\ h$$

And here's subproof (2):

$$
\begin{array}{ll}
 & (pure\ (flip\ (\$)) \circledast mf) \circledast (pure\ (flip\ (\$)) \circledast ma) \\
= & \quad [\![\ \text{idiom composition}\ ]\!] \\
 & pure\ (\circ) \circledast (pure\ (flip\ (\$)) \circledast mf) \circledast pure\ (flip\ (\$)) \circledast ma \\
= & \quad [\![\ \text{idiom composition}\ ]\!] \\
 & pure\ (\circ) \circledast pure\ (\circ) \circledast pure\ (flip\ (\$)) \circledast mf \circledast pure\ (flip\ (\$)) \circledast ma \\
= & \quad [\![\ \text{idiom homomorphism}\ ]\!] \\
 & pure\ ((\circ)\ (\circ)\ (flip\ (\$))) \circledast mf \circledast pure\ (flip\ (\$)) \circledast ma \\
= & \quad [\![\ \text{composition}\ ]\!] \\
 & pure\ ((\circ) \circ flip\ (\$)) \circledast mf \circledast pure\ (flip\ (\$)) \circledast ma \\
= & \quad [\![\ \text{idiom interchange}\ ]\!] \\
 & pure\ (\lambda f \to f\ (flip\ (\$))) \circledast (pure\ ((\circ) \circ flip\ (\$)) \circledast mf) \circledast ma \\
= & \quad [\![\ \text{idiom composition}\ ]\!] \\
 & pure\ (\circ) \circledast pure\ (\lambda f \to f\ (flip\ (\$))) \circledast pure\ ((\circ) \circ flip\ (\$)) \circledast mf \circledast ma \\
= & \quad [\![\ \text{idiom homomorphism}\ ]\!] \\
 & pure\ ((\circ)\ (\lambda f \to f\ (flip\ (\$)))\ ((\circ) \circ flip\ (\$))) \circledast mf \circledast ma \\
= & \quad [\![\ \text{composition}\ ]\!] \\
 & pure\ ((\lambda f \to f\ (flip\ (\$))) \circ (\circ) \circ flip\ (\$)) \circledast mf \circledast ma \\
= & \quad [\![\ \text{combinators: } ((\lambda f \to f\ (flip\ (\$))) \circ (\circ) \circ flip\ (\$)) = id\ ]\!] \\
 & pure\ id \circledast mf \circledast ma \\
= & \quad [\![\ \text{idiom identity}\ ]\!] \\
 & mf \circledast ma
\end{array}
$$

Figure A.2. Proof of the claim in Section 6 that $B \circ B : m \dot{\to} m$ is an idiom morphism.