Understanding Idiomatic Traversals Backwards and Forwards

Richard Bird
Jeremy Gibbons
Department of Computer Science,
University of Oxford, Wolfson Building,
Parks Rd, Oxford OX1 3QD, UK
{bird,jg}@cs.ox.ac.uk

Stefan Mehner
Janis Voigtländer
Institut für Informatik,
Universität Bonn,
Römerstr. 164, 53117 Bonn, Germany
{mehner,jv}@cs.uni-bonn.de

Tom Schrijvers
Department of Applied Mathematics and
Computer Science, Ghent University,
Krijgslaan 281, 9000 Gent, Belgium
tom.schrijvers@ugent.be

Abstract

We present new ways of reasoning about a particular class of effectful Haskell programs, namely those expressed as idiomatic traversals. Starting out with a specific problem about labelling and unlabelling binary trees, we extract a general inversion law, applicable to any monad, relating a traversal over the elements of an arbitrary traversable type to a traversal that goes in the opposite direction. This law can be invoked to show that, in a suitable sense, unlabelling is the inverse of labelling. The inversion law, as well as a number of other properties of idiomatic traversals, is a corollary of a more general theorem characterising traversable functors as finitary containers: an arbitrary traversable object can be decomposed uniquely into shape and contents, and traversal be understood in terms of those. Proof of the theorem involves the properties of traversal in a special idiom related to the free applicative functor.

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

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and recursion schemes, Type structure; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Polymorphism; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords applicative functors; finitary containers; idioms; monads; traversable functors

1. Introduction

How does the presence of effects change our ability to reason about functional programs? More specifically, can we formulate useful equational laws about particular classes of effectful programs in the same way as we can for pure functions? These questions have been around for some time, but such laws have been thin on the ground. The primary purpose of this paper is to state and prove one such law, the *inversion* law of monadic traversals.

Our point of departure is a paper by Hutton and Fulger (2008). In it, they pose a deliberately simple example involving labelling binary trees. Their objective was to find some way of demonstrating that the labelling, expressed using the state monad, generates distinct labels. The method they gave for solving the problem was to reduce stateful computations to pure functions that accept and return a state, and to carry out the necessary equational reasoning with pure functions alone.

We formulate an essentially equivalent version of the labelling problem in terms of a second effectful process that unlabels a binary tree, with the intention of arguing that unlabelling is the inverse of labelling. But our proof method is quite different: it relies on a single equational law about inverting effectful traversals. Moreover, in applying that law, the effects can be produced by an arbitrary monad, not just the state monad, the traversals can be over an arbitrary traversable type, not just binary trees, and all legitimate traversal strategies are allowed.

Apart from formulating the inversion law, the main technical contribution of the paper is the means of proving it. To do so we state and prove a powerful general result that, given a fixed traversal strategy for a type, characterises an arbitrary member of the type (and traversals over it) in terms of its shape and contents. The characterisation depends on traversing with a specific idiom derived from the free applicative functor. We claim that the theorem is a pivotal tool in the study of idiomatic traversals, and some of its other uses are explored in the paper.

Before we start, we make a remark about our equational framework. Although we employ Haskell notation to define types and functions, these entities are to be interpreted in the category Set of sets and total functions, not the Haskell category of domains and continuous functions. In particular, algebraic datatypes will consist of finite structures only. We also distinguish typographically between arbitrary but fixed types (in uppercase sans serif) and polymorphic type variables (in lowercase italics); for example, a particular instance of the $\mathsf{Functor}$ type class will have a method $\mathit{fmap} :: (a \to b) \to \mathsf{F}\ a \to \mathsf{F}\ b$.

2. Tree labelling

Here is the tree datatype in question:

$$\mathbf{data}\ \mathsf{Tree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$

In our version of the labelling problem, trees are annotated with additional elements drawn from an infinite stream, the stream being threaded through the computation via the state monad:

Haskell '13, September 23–24, 2013, Boston, MA, USA.

ACM 978-1-4503-2383-3/13/09…$15.00.
http://dx.doi.org/10.1145/2503778.2503781

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

For notational convenience we have written the infinite streams using Haskell list syntax, but they should not be thought of as an algebraic datatype—for example, they might be defined by total functions over the natural numbers.

The 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:

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

Hutton and Fulger's formulation of the labelling problem boils down to the assertion (essentially their Lemma 7) that

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

for all trees t and streams xs. Observe that the two functions *label* and *labels* are written in quite different styles, the first as an effectful monadic program and the second as a pure function. Hence 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.

As a first step, we might—with a little foresight—define unlabelling as an effectful program in the following way:

$$
\begin{aligned}
&\mathit{unlabel} :: \mathsf{Tree}\ (a,b) \to \mathsf{State}\ [b]\ (\mathsf{Tree}\ a) \\
&\mathit{unlabel}\ (\mathsf{Tip}\ (x,y)) \\
&\quad = \mathbf{do}\ \{ys \leftarrow \mathit{get}; \mathit{put}\ (y:ys); \mathit{return}\ (\mathsf{Tip}\ x)\} \\
&\mathit{unlabel}\ (\mathsf{Bin}\ u\ v) \\
&\quad = \mathbf{do}\ \{v' \leftarrow \mathit{unlabel}\ v; u' \leftarrow \mathit{unlabel}\ u; \mathit{return}\ (\mathsf{Bin}\ u'\ v')\}
\end{aligned}
$$

Unlabelling a tip means putting the second component of the label back on the stream. Unlabelling a node is like labelling one, but with a crucial difference: the process has to proceed in the *opposite* direction. After all, if you put on your socks and then your shoes in the morning, then in the evening you take off your shoes before taking off your socks. This insight is fundamental in what follows.

Now we can rewrite the requirement above in the form

$$
\begin{aligned}
&\mathit{runState}\ (\mathit{label}\ t)\ \mathit{xs} = (u, \mathit{ys}) \quad \Rightarrow \\
&\quad \mathit{runState}\ (\mathit{unlabel}\ u)\ \mathit{ys} = (t, \mathit{xs})
\end{aligned}
$$

Better, we can write the requirement without recourse to *runState*:

$$\mathit{unlabel} \Leftarrow\!\!< \mathit{label} = \mathit{return}$$

where $\Leftarrow\!\!<$ denotes Kleisli composition in a monad:

$$(g \Leftarrow\!\!< f)\ x = \mathbf{do}\ \{y \leftarrow f\ x; z \leftarrow g\ y; \mathit{return}\ z\}$$

But this is still an unsatisfactory way to formulate the problem, because *label* and *unlabel* are specific to the state monad and to binary trees. Instead, we argue that the labelling problem is but an instance of a more general one about *effectful traversals*. As far as possible, any reasoning should be abstracted both from the specific computational effect and the specific datatype. We encapsulate the effects as *idioms* (also called *applicative functors*) as defined by McBride and Paterson (2008), rather than the more familiar monads. Every monad is an idiom, but idioms are more flexible and have better compositional properties than monads, properties that we will need to exploit. And we encapsulate the data in terms of idiomatic *traversals*, also introduced by McBride and Paterson, and studied in more depth by Gibbons and Oliveira (2009) and Jaskelioff and Rypáček (2012).

3. Idiomatic traversals

According to McBride and Paterson (2008) a traversable datatype T is one that supports two interdefinable functions *traverse* and *dist*. For our purposes it suffices to concentrate entirely on *traverse*. This function applies a given effectful function to every element in a given T-structure, accumulating all the effects in order. In the case of a monadic idiom and when the type is lists, one possible choice for the operation is the monadic map *mapM* from the Haskell Prelude. Here is a cut-down version of the Traversable class in the Haskell library Data.Traversable:

$$
\begin{aligned}
&\mathbf{class}\ \mathsf{Functor}\ t \Rightarrow \mathsf{Traversable}\ t\ \mathbf{where} \\
&\quad \mathit{traverse} :: \mathsf{Applicative}\ m \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b)
\end{aligned}
$$

As an instance of the Functor class, each traversable type must satisfy the laws $\mathit{fmap}\ \mathit{id} = \mathit{id}$ and $\mathit{fmap}\ g \circ \mathit{fmap}\ f = \mathit{fmap}\ (g \circ f)$ that apply to all functors.

In particular, trees form a traversable type; traversal of a tip involves visiting its label, and one possible traversal of a binary node involves traversing the complete left subtree before the right subtree. To formulate this traversal, we need to review the class of applicative functors, which is declared by

$$
\begin{aligned}
&\mathbf{class}\ \mathsf{Functor}\ m \Rightarrow \mathsf{Applicative}\ m\ \mathbf{where} \\
&\quad \mathit{pure} \quad :: a \to m\ a \\
&\quad (\circledast) :: m\ (a \to b) \to m\ a \to m\ b
\end{aligned}
$$

The original name was Idiom, but Haskell uses Applicative; the words 'applicative functor' and 'idiom' are interchangeable and we will use both. The method $\circledast$ is called *idiomatic application* and associates to the left in expressions. The methods *pure* and $\circledast$ are required to satisfy four laws, called the *identity*, *composition*, *homomorphism*, and *interchange* laws, respectively:

$$
\begin{aligned}
&\mathit{pure}\ \mathit{id} \circledast u &&= u \\
&\mathit{pure}\ (\circ) \circledast u \circledast v \circledast w &&= u \circledast (v \circledast w) \\
&\mathit{pure}\ f \circledast \mathit{pure}\ x &&= \mathit{pure}\ (f\ x) \\
&u \circledast \mathit{pure}\ x &&= \mathit{pure}\ (\$x) \circledast u
\end{aligned}
$$

In addition, the mapping method *fmap* of the Functor superclass of Applicative should be related to these methods by the property $\mathit{fmap}\ f\ x = \mathit{pure}\ f \circledast x$; indeed, this equation can be used as the definition of *fmap* for applicative functors.

Every monad is an idiom. The connection is given by the equations

$$
\begin{aligned}
&\mathit{pure}\ x &&= \mathit{return}\ x \\
&\mathit{mf} \circledast \mathit{mx} &&= \mathbf{do}\ \{f \leftarrow \mathit{mf}; x \leftarrow \mathit{mx}; \mathit{return}\ (f\ x)\}
\end{aligned}
$$

That is, pure computations coincide with the unit of the monad, and idiomatic application yields the effects of evaluating the function before the effects of evaluating the argument. For monads the idiom laws follow from the monad laws:

$$
\begin{aligned}
&\mathit{return}\ x \gg\!= f &&= f\ x \\
&m \gg\!= \mathit{return} &&= m \\
&(m \gg\!= f) \gg\!= g &&= m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

Now we can define *traverse* for trees. One possible implementation is as follows:

```
instance Traversable Tree where
  traverse f (Tip x)
    = pure Tip <*> f x
  traverse f (Bin u v)
    = pure Bin <*> traverse f u <*> traverse f v
```

Another choice is to traverse the complete right subtree before the left one. And these are not the only two possible traversals. One could give a valid definition for traversing a tree in breadth-first order. Moreover, there is nothing at the moment to prevent us from defining a *traverse* that ignores part of the tree, or visits some subtree twice. For example, we might have defined

```
traverse f (Bin u v)
  = pure (const Bin) <*> traverse f u <*> traverse f u
                     <*> traverse f v
```

We would like to forbid such duplicitous definitions, and the way to do it is to impose constraints on lawful definitions of *traverse*, constraints we will examine in the next section. We return to the question of duplicitous traversals in Section 7.

Tree labelling can be formulated as a tree traversal using the monadic idiom State:

```
label :: Tree a → State [b] (Tree (a,b))
label = traverse adorn
```

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

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

Well and good, but the next problem is how to define *unlabel*. The function *strip* removes the additional label, returning it to the stream:

```
strip :: (a,b) → State [b] a
strip (x,y) = do { ys ← get; put (y : ys); return x }
```

However, we cannot define *unlabel* by *unlabel* = *traverse strip*, because the traversal is in the same direction as for *label* and we need one that goes in the opposite direction. What we have to provide is a function, *treverse* say, such that

```
unlabel = treverse strip
```

and then hope to be able to prove that *unlabel* ⇐< *label* = *return*. But can we do so without having to write a completely separate *treverse* function?

Yes, we can, by defining *treverse* in terms of *traverse* and a 'backwards' idiom. For each idiom M there is a corresponding idiom Backwards M with effects sequenced in the opposite order. This statement is not true when 'idiom' is replaced by 'monad', which is why we generalise from monads to idioms. We use the concepts and semantics from the Haskell library Control.Applicative.Backwards (the name *forwards* for the accessor is awkward but standard).

```
newtype Backwards m a = Backwards { forwards :: m a }
instance Applicative m ⇒ Applicative (Backwards m) where
  pure x = Backwards (pure x)
  Backwards mf <*> Backwards mx
      = Backwards (pure (flip ($)) <*> mx <*> mf)
```

Reversing the order of effects allows us to define *treverse*:

```
treverse :: (Traversable t, Applicative m) ⇒
            (a → m b) → t a → m (t b)
treverse f = forwards ∘ traverse (Backwards ∘ f)
```

Now we arrive at the central question: given an arbitrary monad M, what are sufficient conditions for us to be able to assert that

```
treverse g ⇐< traverse f = return
```

for effectful functions $f :: \mathsf{A} \to \mathsf{M\ B}$ and $g :: \mathsf{B} \to \mathsf{M\ A}$? The question makes sense for monads but not for arbitrary idioms, because Kleisli composition ⇐< is an operator only on monadic computations; that is why we still need to discuss monads, despite using idiomatic traversals. The answer to the question is not surprising: it is when *g* ⇐< *f* = *return*. We call this result the *inversion* law of monadic traversals. The solution to the tree labelling problem is a simple instance of the inversion law:

```
  unlabel ⇐< label
=    [[ definitions ]]
  treverse strip ⇐< traverse adorn
=    [[ since strip ⇐< adorn = return ]]
  return
```

We prove the inversion law in full generality, for all lawful implementations of *traverse* over any traversable type, and for all monads. The reader is warned that the proof involves some heavy calculational machinery, though many of the details are relegated to an appendix. In fact, we prove a general representation theorem for traversable functors, from which the inversion law follows.

We begin in the next section with what it means for an implementation of *traverse* to be lawful. It is essential to enforce some laws: even just the special case *unlabel* ⇐< *label* = *return* of the inversion law breaks when a traversal is duplicitous, say. One might think that duplication causes no problems, since it will occur both in the forwards and in the backwards traversal, and the duplicate effects will cancel out. But this is not what happens. Simply consider the duplicitous definition of *traverse* for binary trees given above; for *tree* = Bin (Tip 'a') (Tip 'b'), we have *runState* ((*unlabel* ⇐< *label*) *tree*) [1..] = (*tree*, [2,2,3,4,5,...]).

4. The laws of traversal

Firstly, we introduce a notational device, used throughout the rest of the paper, that may help with readability. Both traversable types and idioms are functors, so we can map over them with one and the same generic function *fmap*. However, uses of, say, *fmap* (*fmap f*) can prove confusing: which *fmap* refers to which functor? Accordingly we will employ two synonyms for *fmap*, namely *imap* to signify mapping over an idiom, and *tmap* to signify mapping over a traversable type.

Secondly, to avoid possible ambiguity when discussing the four type parameters in the type

```
traverse :: (Traversable t, Applicative m) ⇒
            (a → m b) → t a → m (t b)
```

we will refer to *a* and *b* as 'the elements', *m* as 'the idiom', and *t* as 'the datatype'. Bear in mind that by the word 'datatype' we do not mean just Haskell algebraic datatypes; *t* could be instantiated to any type constructor, for example $\mathsf{T}\ a = (\forall b. b \to (b \to b) \to b) \to a$.

The first law of *traverse* is the one called the *unitarity* law by Jaskelioff and Rypáček (2012), and concerns the identity idiom:

```
newtype Identity a = Identity { runIdentity :: a }
instance Applicative Identity where
  pure x                      = Identity x
  Identity f <*> Identity x = Identity (f x)
```

The unitarity law is simply that

```
traverse Identity = Identity
```

The second law of *traverse* is called the *linearity* law by Jaskelioff and Rypáček, and concerns idiom composition. Idioms compose nicely, in exactly the way that monads do not:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Compose}\ m\ n\ a = \mathsf{Compose}\ (m\ (n\ a)) \\
\textbf{instance}\ (\mathsf{Applicative}\ m, \mathsf{Applicative}\ n) \Rightarrow \\
\qquad\quad \mathsf{Applicative}\ (\mathsf{Compose}\ m\ n)\ \textbf{where} \\
\quad \mathit{pure}\ x = \mathsf{Compose}\ (\mathit{pure}\ (\mathit{pure}\ x)) \\
\quad \mathsf{Compose}\ \mathit{mf} \circledast \mathsf{Compose}\ \mathit{mx} \\
\qquad\quad = \mathsf{Compose}\ (\mathit{pure}\ (\circledast) \circledast \mathit{mf} \circledast \mathit{mx})
\end{array}
$$

We can introduce an idiomatic composition operator:

$$
\begin{array}{l}
(\mathbin{<\!\!\odot\!\!>}) :: (\mathsf{Applicative}\ m, \mathsf{Applicative}\ n) \Rightarrow \\
\qquad (b \to n\ c) \to (a \to m\ b) \to a \to \mathsf{Compose}\ m\ n\ c \\
g \mathbin{<\!\!\odot\!\!>} f = \mathsf{Compose} \circ \mathit{imap}\ g \circ f
\end{array}
$$

The linearity law of *traverse* states that

$$\mathit{traverse}\ g \mathbin{<\!\!\odot\!\!>} \mathit{traverse}\ f = \mathit{traverse}\ (g \mathbin{<\!\!\odot\!\!>} f)$$

The remaining two properties of *traverse* concern *naturality*. Recall the type of *traverse* given above. First off, *traverse* is *natural in the elements*, both a and b. Naturality in a means that for all functions $g :: \mathsf{A}' \to \mathsf{A}$ we have

$$\mathit{traverse}\ f \circ \mathit{tmap}\ g = \mathit{traverse}\ (f \circ g)$$

for all $f :: \mathsf{A} \to \mathsf{M}\ \mathsf{B}$. Naturality in b means that for all $g :: \mathsf{B} \to \mathsf{B}'$ we have

$$\mathit{imap}\ (\mathit{tmap}\ g) \circ \mathit{traverse}\ f = \mathit{traverse}\ (\mathit{imap}\ g \circ f)$$

for all $f :: \mathsf{A} \to \mathsf{M}\ \mathsf{B}$.

The second property is that *traverse* should also be *natural in the idiom*, which is to say that

$$\varphi \circ \mathit{traverse}\ f = \mathit{traverse}\ (\varphi \circ f)$$

for all *idiom morphisms* φ. A polymorphic function $\varphi :: \mathsf{M}\ a \to \mathsf{N}\ a$ is an idiom morphism (from idiom M to idiom N) if it satisfies

$$
\begin{array}{ll}
\varphi\ (\mathit{pure}\ x) & = \mathit{pure}\ x \\
\varphi\ (\mathit{mf} \circledast \mathit{mx}) & = \varphi\ \mathit{mf} \circledast \varphi\ \mathit{mx}
\end{array}
$$

(From these two equations it follows that φ itself is natural, that is, $\varphi \circ \mathit{imap}\ g = \mathit{imap}\ g \circ \varphi$ for all $g :: \mathsf{A} \to \mathsf{A}'$.) That concludes the constraints we impose on lawful traversals.

One consequence of those constraints is the *purity* law:

$$\mathit{traverse}\ \mathit{pure} = \mathit{pure}$$

This follows from the unitarity law, *traverse* being natural in the idiom, and the fact that $\mathit{pure} \circ \mathit{runIdentity}$ is an idiom morphism from $\mathsf{Identity}$ to any idiom M.

Despite its similarity to the linearity law, the equation

$$\mathit{traverse}\ g \mathbin{<\!\!=\!\!<} \mathit{traverse}\ f = \mathit{traverse}\ (g \mathbin{<\!\!=\!\!<} f)$$

does not hold in general, not even if all the constraints imposed on *traverse* in this section are fulfilled and if additionally $g \mathbin{<\!\!=\!\!<} f = \mathit{pure}\ (= \mathit{return})$ holds. The crucial difference is the order of effects: on the left the effects of applying f are triggered before those of g, while on the right the two take turns. The equation is valid however for so-called *commutative monads* (Gibbons and Oliveira 2009).

Another observation concerns backwards idioms. Although $\mathsf{Backwards}$ is natural (that is, $\mathsf{Backwards} \circ \mathit{imap}\ g = \mathit{imap}\ g \circ \mathsf{Backwards}$), it is not always an idiom morphism: in general, there is no relationship between $\mathsf{Backwards}\ (\mathit{mf} \circledast \mathit{mx})$ and $\mathsf{Backwards}\ \mathit{mf} \circledast \mathsf{Backwards}\ \mathit{mx}$. On the other hand, $\mathsf{Backwards} \circ \mathsf{Backwards}$ is an idiom morphism, expressing the fact that reversal of the order of effects is an involution. Consequently, we can derive a dual characterisation of *traverse* in terms of *treverse*:

$$
\begin{array}{l}
\quad \mathit{traverse}\ f \\
= \quad [\![\ \mathsf{Backwards}\ \text{is an isomorphism}\ ]\!] \\
\quad \mathit{forwards} \circ \mathit{forwards} \circ \mathsf{Backwards} \circ \mathsf{Backwards} \circ \mathit{traverse}\ f \\
= \quad [\![\ \mathsf{Backwards} \circ \mathsf{Backwards}\ \text{is an idiom morphism}\ ]\!] \\
\quad \mathit{forwards} \circ \mathit{forwards} \circ \mathit{traverse}\ (\mathsf{Backwards} \circ \mathsf{Backwards} \circ f) \\
= \quad [\![\ \text{definition of}\ \mathit{treverse}\ ]\!] \\
\quad \mathit{forwards} \circ \mathit{treverse}\ (\mathsf{Backwards} \circ f)
\end{array}
$$

Thus the inversion law can be stated in the dual form:

$$g \mathbin{<\!\!=\!\!<} f = \mathit{return} \quad \Rightarrow \quad \mathit{traverse}\ g \mathbin{<\!\!=\!\!<} \mathit{treverse}\ f = \mathit{return}$$

As a final remark to the eagle-eyed reader, one property of *traverse* seems to be missing from the list. We have imposed naturality in the elements a, b and in the idiom m, but what about naturality in the datatype t? We will return to exactly this point in Section 6, where we prove that a restricted kind of naturality in the datatype follows from the other conditions.

5. The Representation Theorem

Our challenge is to prove the inversion law for any traversable T, with any definition of *traverse* over T that satisfies the laws imposed in the previous section. The key fact is the Representation Theorem given below, which establishes that, in the category Set, traversable datatypes correspond exactly to finitary containers. This means that every member $t :: \mathsf{T}\ \mathsf{A}$ is determined by an *arity* n, a *shape* with *n holes*, and *n elements* $a_1, \ldots, a_n$ of type A. Finitary containers are also known as finitary *dependent polynomial functors* (Gambino and Hyland 2004), *normal functors* (Girard 1988), and *shapely functors* (Moggi et al. 1999).

We begin with two preliminary definitions. The first concerns the function *contents*, which has the general type

$$\mathit{contents} :: \mathsf{Traversable}\ t \Rightarrow t\ a \to [a]$$

This function is defined using *traverse* in a Const idiom. Provided A is a type carrying a monoid structure, the functor $\mathsf{Const}\ \mathsf{A}\ b = \mathsf{A}$ determines an idiom in which pure computations yield the neutral element *mempty* of A and idiomatic application reduces to the binary operator *mappend* of A:

$$
\begin{array}{l}
\textbf{newtype}\ \mathsf{Const}\ a\ b = \mathsf{Const}\ \{\mathit{getConst} :: a\} \\
\textbf{instance}\ \mathsf{Monoid}\ a \Rightarrow \mathsf{Applicative}\ (\mathsf{Const}\ a)\ \textbf{where} \\
\quad \mathit{pure}\ x \qquad\qquad\quad\ \ = \mathsf{Const}\ \mathit{mempty} \\
\quad \mathsf{Const}\ x \circledast \mathsf{Const}\ y = \mathsf{Const}\ (\mathit{mappend}\ x\ y)
\end{array}
$$

Now we can define

$$
\begin{array}{l}
\mathit{contents} :: \mathsf{Traversable}\ t \Rightarrow t\ a \to [a] \\
\mathit{contents} = \mathit{getConst} \circ \mathit{traverse}\ (\lambda x \to \mathsf{Const}\ [x])
\end{array}
$$

Of course, in the list monoid used here we have $\mathit{mempty} = [\,]$ and $\mathit{mappend} = (+\!\!+)$.

The second definition is as follows.

DEFINITION. Let T be traversable and *traverse* its traversal function. A polymorphic function $\mathit{make} :: a \to \cdots \to a \to \mathsf{T}\ a$ with n arguments is called a *make function* (of arity n), provided that the following two conditions hold:

$$
\begin{array}{ll}
\mathit{tmap}\ f\ (\mathit{make}\ x_1 \ldots x_n) & = \mathit{make}\ (f\ x_1)\ (f\ x_2) \ldots (f\ x_n) \\
\mathit{contents}\ (\mathit{make}\ x_1 \ldots x_n) & = [x_1, x_2, \ldots, x_n]
\end{array}
$$

for all $x_1, \ldots, x_n$ of any type A and functions f of type $\mathsf{A} \to \mathsf{B}$ for any type B. □

The first condition in the above definition is the naturality property associated with polymorphic functions. Note in the second condition that *contents*, and hence the property of being a make function, depends on the given definition of *traverse*. For example,

$\lambda x_1\ x_2\ x_3 \to \mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ x_1)\ (\mathsf{Tip}\ x_2))\ (\mathsf{Tip}\ x_3)$ is a make function for the depth-first *traverse* of binary trees defined in Section 3, while $\lambda x_1\ x_2\ x_3 \to \mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ x_2)\ (\mathsf{Tip}\ x_3))\ (\mathsf{Tip}\ x_1)$ is a make function for breadth-first traversal. We see that make functions indeed serve as an abstract notion of shapes with holes.

THEOREM (Representation Theorem). Let T be traversable and *traverse* its traversal function, and let A be given. For every member $t :: \mathsf{T\ A}$, there is a unique n, a unique make function *make* of arity n, and unique values $a_1, \ldots, a_n$, all of type A, such that $t = make\ a_1 \ldots a_n$. Furthermore, the make function so obtained satisfies

$$
\begin{aligned}
&traverse\ f\ (make\ x_1 \ldots x_n)\\
&\quad = pure\ make \circledast f\ x_1 \circledast f\ x_2 \circledast \cdots \circledast f\ x_n
\end{aligned}
$$

for all $x_i :: \mathsf{A}'$ for any type A', for any idiom M, and any function $f :: \mathsf{A}' \to \mathsf{M\ B}$ for any type B. □

The proof is in Section 9 and contains the construction of the unique representation. Note that *make* depends both on t and on the given definition of *traverse* for T, but not on the elements of t.

The Representation Theorem characterises *traverse*, but we can also characterise *treverse*. Abbreviating $\mathsf{Backwards}$ to B, we have

$$
\begin{aligned}
&treverse\ f\ (make\ x_1 \ldots x_n)\\
=\ &[\![\ \text{definition of } treverse\ ]\!]\\
&forwards\ (traverse\ (\mathsf{B} \circ f)\ (make\ x_1 \ldots x_n))\\
=\ &[\![\ \text{Representation Theorem, and } pure\ f = \mathsf{B}\ (pure\ f)\ ]\!]\\
&forwards\ (\mathsf{B}\ (pure\ make) \circledast \mathsf{B}\ (f\ x_1) \circledast \cdots \circledast \mathsf{B}\ (f\ x_n))\\
=\ &[\![\ \text{formula for backwards idiom (see below)}\ ]\!]\\
&pure\ (\lambda x_n \ldots x_1 \to make\ x_1 \ldots x_n) \circledast f\ x_n \circledast \cdots \circledast f\ x_1
\end{aligned}
$$

The formula appealed to in the final step above is that

$$
\begin{aligned}
&\mathsf{B}\ (pure\ f) \circledast \mathsf{B}\ x_1 \circledast \cdots \circledast \mathsf{B}\ x_n\\
&\quad = \mathsf{B}\ (pure\ (\lambda x_n \ldots x_1 \to f\ x_1 \ldots x_n) \circledast x_n \circledast \cdots \circledast x_1)
\end{aligned}
$$

The proof is in Appendix A.1.

When the idiom is a monad we have the two specialisations

$$
\begin{aligned}
&traverse\ f\ (make\ x_1 \ldots x_n)\\
&\quad = \mathbf{do}\ \{y_1 \leftarrow f\ x_1; y_2 \leftarrow f\ x_2; \ldots; y_n \leftarrow f\ x_n;\\
&\qquad\quad return\ (make\ y_1\ y_2 \ldots y_n)\}\\
&treverse\ f\ (make\ x_1 \ldots x_n)\\
&\quad = \mathbf{do}\ \{y_n \leftarrow f\ x_n; \ldots; y_2 \leftarrow f\ x_2; y_1 \leftarrow f\ x_1;\\
&\qquad\quad return\ (make\ y_1\ y_2 \ldots y_n)\}
\end{aligned}
$$

These facts yield our inversion law. We have:

$$
\begin{aligned}
&(treverse\ g \Leftarrow\!\!< traverse\ f)\ (make\ a_1 \ldots a_n)\\
=\ &[\![\ \text{definition of } \Leftarrow\!\!<\ ]\!]\\
&\mathbf{do}\ \{t' \leftarrow traverse\ f\ (make\ a_1 \ldots a_n); treverse\ g\ t'\}\\
=\ &[\![\ \text{characterisation of } traverse\ ]\!]\\
&\mathbf{do}\ \{x_1 \leftarrow f\ a_1; \ldots; x_n \leftarrow f\ a_n;\\
&\quad treverse\ g\ (make\ x_1 \ldots x_n)\}\\
=\ &[\![\ \text{characterisation of } treverse\ ]\!]\\
&\mathbf{do}\ \{x_1 \leftarrow f\ a_1; \ldots; x_n \leftarrow f\ a_n;\\
&\quad y_n \leftarrow g\ x_n; \ldots; y_1 \leftarrow g\ x_1;\\
&\quad return\ (make\ y_1 \ldots y_n)\}
\end{aligned}
$$

Now suppose that $g \Leftarrow\!\!< f = return$, that is,

$$\mathbf{do}\ \{x \leftarrow f\ a; y \leftarrow g\ x; return\ y\} = return\ a$$

Then by induction on m we can prove for $0 \leqslant m \leqslant n$ that

$$
\begin{aligned}
&\mathbf{do}\ \{x_1 \leftarrow f\ a_1; \ldots; x_m \leftarrow f\ a_m;\\
&\quad y_m \leftarrow g\ x_m; \ldots; y_1 \leftarrow g\ x_1;\\
&\quad return\ (make\ y_1 \ldots y_m\ a_{m+1} \ldots a_n)\}\\
&= return\ (make\ a_1 \ldots a_n)
\end{aligned}
$$

which for $m = n$ completes the proof of the inversion law. The base case $m = 0$ is trivial. The induction step is also straightforward: thanks to the above cancelling rule and the monad laws, we have:

$$
\begin{aligned}
&\mathbf{do}\ \{x_1 \leftarrow f\ a_1; \ldots; x_m \leftarrow f\ a_m;\\
&\quad y_m \leftarrow g\ x_m; \ldots; y_1 \leftarrow g\ x_1;\\
&\quad return\ (make\ y_1 \ldots y_m\ a_{m+1} \ldots a_n)\}\\
=\ &\mathbf{do}\ \{x_1 \leftarrow f\ a_1; \ldots; x_{m-1} \leftarrow f\ a_{m-1};\\
&\quad y_{m-1} \leftarrow g\ x_{m-1}; \ldots; y_1 \leftarrow g\ x_1;\\
&\quad return\ (make\ y_1 \ldots y_{m-1}\ a_m \ldots a_n)\}
\end{aligned}
$$

6. 'Naturality' in the datatype

Another consequence of the Representation Theorem is that we can now prove that *traverse* is natural in the datatype, at least in a certain sense. As a first attempt we might ask whether

$$imap\ \psi \circ traverse\ f = traverse\ f \circ \psi \qquad (1)$$

holds for all polymorphic $\psi :: \mathsf{T}\ a \to \mathsf{T}'\ a$. The *traverse* on the left is over T and need bear no relationship to the *traverse* on the right, which is over T'. A little reflection should reveal that the equivalence is far too strong; ψ might reorder, drop or duplicate elements, which will induce reordered, dropped or duplicated effects arising from *traverse* on the right, as compared to *traverse* on the left.

Rather, (1) should be asserted only for ψ that do not reorder, drop or duplicate elements. To formulate this constraint, we restrict ψ to *contents-preserving* functions:

$$contents = contents \circ \psi$$

In fact, this contents-preservation property is a consequence of (1):

$$
\begin{aligned}
&contents\\
=\ &[\![\ \text{definition of } contents\ ]\!]\\
&getConst \circ traverse\ (\lambda x \to \mathsf{Const}\ [x])\\
=\ &[\![\ \text{property of } getConst \text{ and } imap\ ]\!]\\
&getConst \circ imap\ \psi \circ traverse\ (\lambda x \to \mathsf{Const}\ [x])\\
=\ &[\![\ \text{assumed property (1)}\ ]\!]\\
&getConst \circ traverse\ (\lambda x \to \mathsf{Const}\ [x]) \circ \psi\\
=\ &[\![\ \text{definition of } contents\ ]\!]\\
&contents \circ \psi
\end{aligned}
$$

A reasonable question to ask now is: Is this property, in conjunction with the naturality of ψ, also sufficient to establish (1)? Yes, it is, and the proof is another application of the Representation Theorem.

Let $t :: \mathsf{T\ A}$. To prove $imap\ \psi\ (traverse\ f\ t) = traverse\ f\ (\psi\ t)$, suppose $t = make\ a_1 \ldots a_n$, where *make* is given by the Representation Theorem. Define $make'$ by

$$make'\ x_1 \ldots x_n = \psi\ (make\ x_1 \ldots x_n)$$

If we can show that $make'$ satisfies the two conditions of a make function, then, by the Representation Theorem, $make'$ is the unique make function yielding $t' :: \mathsf{T}'\ \mathsf{A}$ where $t' = \psi\ t = make'\ a_1 \ldots a_n$.

Here is the proof that $make'$ is natural:

$$
\begin{aligned}
&tmap\ f\ (make'\ x_1 \ldots x_n)\\
=\ &[\![\ \text{definition of } make'\ ]\!]\\
&tmap\ f\ (\psi\ (make\ x_1 \ldots x_n))\\
=\ &[\![\ \psi \text{ is natural}\ ]\!]\\
&\psi\ (tmap\ f\ (make\ x_1 \ldots x_n))\\
=\ &[\![\ make \text{ is natural}\ ]\!]\\
&\psi\ (make\ (f\ x_1) \ldots (f\ x_n))\\
=\ &[\![\ \text{definition of } make'\ ]\!]\\
&make'\ (f\ x_1) \ldots (f\ x_n)
\end{aligned}
$$

And here is the *contents*-property:

$$
\begin{aligned}
&contents\ (make'\ x_1 \ldots x_n)\\
=\ &[\![\ \text{definition of } make'\ ]\!]
\end{aligned}
$$

$$\begin{aligned}
& \mathit{contents}\ (\psi\ (\mathit{make}\ x_1 \ldots x_n)) \\
= \quad & [\![\ \psi \text{ is contents-preserving }]\!] \\
& \mathit{contents}\ (\mathit{make}\ x_1 \ldots x_n) \\
= \quad & [\![\ \mathit{contents}\text{-property for } \mathit{make}\ ]\!] \\
& [x_1, \ldots, x_n]
\end{aligned}$$

Good: make' is indeed the make function for t', with elements $a_1, \ldots, a_n$. That means

$$\mathit{traverse}\ f\ t' = \mathit{pure}\ \mathit{make}' \circledast f\ a_1 \circledast \cdots \circledast f\ a_n$$

It remains to prove that the right-hand side is equal to

$$\mathit{imap}\ \psi\ (\mathit{traverse}\ f\ (\mathit{make}\ a_1 \ldots a_n))$$

For this we need a result which we will call the *flattening* formula of $\circledast$. For brevity, we write

$$x \oplus_{i=1}^{n} x_i = ((x \oplus x_1) \oplus x_2) \oplus \cdots \oplus x_n$$

for any binary operator $\oplus$. We also introduce the *generalised composition* operator $\circ_{m,n}$ for $0 \leqslant m \leqslant n$, defined by

$$(g \circ_{m,n} f)\ x_1 \ldots x_n = g\ x_1 \ldots x_m\ (f\ x_{m+1} \ldots x_n)$$

The flattening formula is then

$$\begin{aligned}
& (\mathit{pure}\ g\ \circledast_{i=1}^{m}\ x_i) \circledast (\mathit{pure}\ f\ \circledast_{i=m+1}^{n}\ x_i) \\
& \quad = \mathit{pure}\ (g \circ_{m,n} f)\ \circledast_{i=1}^{n}\ x_i
\end{aligned} \tag{2}$$

The proof is given in Appendix A.2. Now we can argue:

$$\begin{aligned}
& \mathit{imap}\ \psi\ (\mathit{traverse}\ f\ (\mathit{make}\ a_1 \ldots a_n)) \\
= \quad & [\![\ \text{Representation Theorem}\ ]\!] \\
& \mathit{imap}\ \psi\ (\mathit{pure}\ \mathit{make}\ \circledast_{i=1}^{n}\ f\ a_i) \\
= \quad & [\![\ \text{since } \mathit{imap}\ f\ x = \mathit{pure}\ f \circledast x\ ]\!] \\
& \mathit{pure}\ \psi \circledast (\mathit{pure}\ \mathit{make}\ \circledast_{i=1}^{n}\ f\ a_i) \\
= \quad & [\![\ \text{flattening formula of } \circledast\ ]\!] \\
& \mathit{pure}\ (\psi \circ_{0,n} \mathit{make})\ \circledast_{i=1}^{n}\ f\ a_i \\
= \quad & [\![\ \text{definition of } \mathit{make}'\ ]\!] \\
& \mathit{pure}\ \mathit{make}'\ \circledast_{i=1}^{n}\ f\ a_i
\end{aligned}$$

Thus, a restricted kind of naturality in the datatype—(1) for all natural and contents-preserving ψ—does indeed follow from the laws in Section 4. For brevity, in the rest of the paper we call this restricted naturality condition 'naturality' in the datatype, in quotes.

7. Two other consequences

Here are two other consequences of the Representation Theorem. First, datatypes containing infinite data structures are not traversable; we illustrate this by proving that in Set the datatype of streams is not traversable. We define

$$\begin{aligned}
& \mathbf{data}\ \mathsf{Nat} = \mathsf{Zero} \mid \mathsf{Succ\ Nat} \\
& \mathbf{type}\ \mathsf{Stream}\ a = \mathsf{Nat} \to a
\end{aligned}$$

The Functor instance for Stream is given by $\mathit{tmap}\ f\ g = f \circ g$. Assume there is a lawful implementation of *traverse* on streams, so *traverse* has type

$$\begin{aligned}
& \mathit{traverse} :: \mathsf{Applicative}\ m \Rightarrow \\
& \qquad (a \to m\ b) \to \mathsf{Stream}\ a \to m\ (\mathsf{Stream}\ b)
\end{aligned}$$

Consider $\mathit{nats} = \mathit{id} :: \mathsf{Stream\ Nat}$, the stream of natural numbers in ascending order. By the Representation Theorem, there exists an n, a make function $\mathit{make} :: a \to \cdots \to a \to \mathsf{Stream}\ a$ of arity n, and n values $a_1, \ldots, a_n :: \mathsf{Nat}$ such that $\mathit{id} = \mathit{make}\ a_1 \ldots a_n$. It follows that for every $f :: \mathsf{Nat} \to \mathsf{Bool}$ we have

$$\begin{aligned}
& f \\
= \quad & [\![\ \text{since } \mathit{id} \text{ is the identity function}\ ]\!] \\
& f \circ \mathit{id} \\
= \quad & [\![\ \text{definition of } \mathit{tmap} \text{ on streams}\ ]\!] \\
& \mathit{tmap}\ f\ \mathit{id} \\
= \quad & [\![\ \text{since } \mathit{id} = \mathit{make}\ a_1 \ldots a_n\ ]\!] \\
& \mathit{tmap}\ f\ (\mathit{make}\ a_1 \ldots a_n) \\
= \quad & [\![\ \text{naturality of } \mathit{make}\ ]\!] \\
& \mathit{make}\ (f\ a_1) \ldots (f\ a_n)
\end{aligned}$$

But this implies that any two functions of type $\mathsf{Nat} \to \mathsf{Bool}$ that agree on the values $a_1, \ldots, a_n$ must be equal, and this is clearly not true. The same reasoning shows that for any infinite type K, the datatype $\mathsf{T}\ a = \mathsf{K} \to a$ is not traversable. But for finite K, say $\mathsf{T}\ a = \mathsf{Bool} \to a$, the datatype is traversable.

The second consequence settles an open question as to whether all possible lawful definitions of *traverse* for the same datatype coincide up to the order of effects: they do. In particular, computing the contents of an object using different lawful *traverse* functions will always result in a permutation of one and the same list.

This second consequence also shows that we cannot have two lawful traversals over the same datatype of which one is duplicitous and the other is not. Here, a *duplicitous* traversal is one that visits some entries more than once. In particular, showing that the obvious depth-first *traverse* of binary trees satisfies the laws rules out the duplicitous traversal mentioned in Section 3 from being lawful. On the other hand, the breadth-first traversal is, of course, a perfectly legitimate alternative traversal, because the effects are the same though in a different order. Similarly, *half-hearted* traversals, which ignore some entries, are also excluded.

For the proof, suppose we have two lawful implementations $\mathit{traverse}_1$ and $\mathit{traverse}_2$ for a single T, and both lawful with respect to the same implementation of *tmap* for T. Let make_1 be any n-ary make function with respect to $\mathit{traverse}_1$. Let Fin_n be a finite type having exactly n values $1, \ldots, n$, so $\mathit{make}_1\ 1 \ldots n$ is a member of $\mathsf{T}\ \mathsf{Fin}_n$. By the Representation Theorem for $\mathit{traverse}_2$ there is an m, a make function make_2 of arity m, and m values $a_1, \ldots, a_m$ such that

$$\mathit{make}_1\ 1 \ldots n = \mathit{make}_2\ a_1 \ldots a_m$$

Hence, using the naturality of make_2 we have

$$\mathit{make}_1\ 1 \ldots n = \mathit{tmap}\ a\ (\mathit{make}_2\ 1 \ldots m)$$

where $a :: \mathsf{Fin}_m \to \mathsf{Fin}_n$ is some function such that $a\ i = a_i$ for each i.

Now we switch horses. By the Representation Theorem for $\mathit{traverse}_1$ there is a p, a make function make'_1 of arity p, and p values $b_1, \ldots, b_p$ such that

$$\mathit{make}_2\ 1 \ldots m = \mathit{make}'_1\ b_1 \ldots b_p$$

Hence, using the naturality of make'_1 we have

$$\mathit{make}_2\ 1 \ldots m = \mathit{tmap}\ b\ (\mathit{make}'_1\ 1 \ldots p)$$

where $b :: \mathsf{Fin}_p \to \mathsf{Fin}_m$ is some function such that $b\ i = b_i$. Putting these two results together (and using naturality twice), we obtain

$$\mathit{make}_1\ 1 \ldots n = \mathit{make}'_1\ (a\ (b\ 1)) \ldots (a\ (b\ p))$$

But by the Representation Theorem for $\mathit{traverse}_1$, applied to exactly this member of $\mathsf{T}\ \mathsf{Fin}_n$, we can conclude that because of uniqueness $\mathit{make}'_1 = \mathit{make}_1$, $p = n$ and that $a \circ b :: \mathsf{Fin}_n \to \mathsf{Fin}_n$ is the identity function. Moreover, by using various of the equations we know by now, we obtain

$$\begin{aligned}
& \mathit{make}_2\ 1 \ldots m \\
& = \mathit{tmap}\ b\ (\mathit{make}'_1\ 1 \ldots p) \\
& = \mathit{tmap}\ b\ (\mathit{make}_1\ 1 \ldots n) \\
& = \mathit{tmap}\ b\ (\mathit{tmap}\ a\ (\mathit{make}_2\ 1 \ldots m))
\end{aligned}$$

and thus

$$\mathit{make}_2\ 1 \ldots m = \mathit{make}_2\ (b\ (a\ 1)) \ldots (b\ (a\ m))$$

Hence $b \circ a :: \mathsf{Fin}_m \to \mathsf{Fin}_m$ is also the identity function. Thus $m = n$ because only functions between sets of the same size can be bijections. Consequently, a and b are two mutually inverse permutations of $1,\ldots,n$.

So we have learned that for every make function $make_1$ with respect to $traverse_1$ there is a make function $make_2$ with respect to $traverse_2$ of the same arity, say n, and a permutation a on $1,\ldots,n$ such that $make_1\ 1\ldots n = make_2\ (a\ 1)\ldots(a\ n)$. By naturality of make functions, this implies that for $x_1,\ldots,x_n$ of arbitrary type A, $make_1\ x_1\ldots x_n = make_2\ x_{a(1)}\ldots x_{a(n)}$. Specifically, for every $t :: \mathsf{T\ A}$ which $make_1$ yields, it holds that if $contents_1\ t = [x_1,\ldots,x_n]$, then $contents_2\ t = [x_{a(1)},\ldots,x_{a(n)}]$. A similar property can be stated for arbitrary traversals with $traverse_1$ and $traverse_2$ of the same t with the same effect function (rather than just with $\lambda x \to \mathsf{Const}\ [x]$).

8. The batch idiom

The Representation Theorem claims both the existence and uniqueness of a representation $t = make\ a_1\ldots a_n$ for each t. The representation can be calculated by traversing t with a special function *batch* that depends on a specially designed idiom Batch related to the *free* idiom (Capriotti and Kaposi 2013). This section is devoted to explaining *batch* and Batch, and a related function *runWith*, and thus preparing for the proof of the Representation Theorem in Section 9. We begin by developing further intuition about what idioms actually are.

Idiomatic results arise in three ways. Firstly, they arise by atomic actions that are truly effectful, and thus make essential use of the idiom; for example, functions like *get* and *put* in stateful computations. Secondly, they arise by lifting pure values into the idiom, using only the method *pure*. Finally, they arise by combining two idiomatic values into one, using idiomatic application $\circledast$.

In the framework of sets and total functions it is a fact (McBride and Paterson 2008, Exercise 2) that every idiomatic expression can be written in the form

$$pure\ f\ \circledast_{i=1}^{n}\ x_i$$

where $x_1,\ldots,x_n$ are atomic effectful computations. Pure results are already in this form (take $n = 0$), and atomic actions x can be written as $pure\ id \circledast x$ by applying an idiom law. The interesting case is the third one about combining two calculations with $\circledast$. This is handled by (2), the flattening formula of $\circledast$.

The batch idiom is a reification of this normal form, mimicking the syntactical building blocks of idiomatic expressions with its constructors. Instead of actually performing effectful computations, Batch just provides a structure for the computations to be performed. This is somewhat reminiscent of batch processing; hence the name. Moreover, Batch is tailored to gain specific insight into *traverse*: since the only effectful computations *traverse* can ever perform are the results of the function (of type $\mathsf{A} \to \mathsf{M\ B}$, say) given to it as the first argument, all atomic actions will have the same type (M B, then), and will be obtained from elements of another fixed type (A, then). Hence, Batch is a specialised version of the type **data** $\mathsf{Free}\ f\ c = \mathsf{P}\ c \mid \forall x.\mathsf{Free}\ f\ (x \to c) :\!*\!: f\ x$, one possible formulation of the free applicative functor (Capriotti and Kaposi 2013); specialised by avoiding the existential variable x.

Specifically, Batch is declared as a *nested datatype* (Bird and Meertens 1998) as follows:

$$\textbf{data}\ \mathsf{Batch}\ a\ b\ c = \mathsf{P}\ c \mid \mathsf{Batch}\ a\ b\ (b \to c) :\!*\!: a$$

Like $\circledast$, the constructor $:\!*\!:$ associates to the left in expressions. Every member $u :: \mathsf{Batch\ A\ B\ C}$ is finite and takes the form

$$u = \mathsf{P}\ f\ :\!*\!:_{i=1}^{n}\ x_i$$

for some n and some values $x_1,\ldots,x_n$ of type A, where f is some function of type $\mathsf{B} \to \cdots \to \mathsf{B} \to \mathsf{C}$ with n arguments.

Here is the functor instance for Batch:

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ (\mathsf{Batch}\ a\ b)\ \textbf{where} \\
\quad fmap\ f\ (\mathsf{P}\ c) \quad = \mathsf{P}\ (f\ c) \\
\quad fmap\ f\ (u :\!*\!: a) = fmap\ (f\circ)\ u :\!*\!: a
\end{array}$$

The applicative functor instance is trickier, but the intuition for the definition of $\circledast$ is that we want both $fmap\ f\ x = pure\ f \circledast x$ and the *flattening* formula of $:\!*\!:$ to hold:

$$\begin{array}{l}
(\mathsf{P}\ g\ :\!*\!:_{i=1}^{m}\ x_i) \circledast (\mathsf{P}\ f\ :\!*\!:_{i=m+1}^{n}\ x_i) \\
\quad = \mathsf{P}\ (g \circ_{m,n} f)\ :\!*\!:_{i=1}^{n}\ x_i
\end{array} \qquad (3)$$

for $0 \leqslant m \leqslant n$. The applicative functor instance is:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ \mathsf{Applicative}\ (\mathsf{Batch}\ a\ b)\ \textbf{where}} \\
\quad pure & & & = \mathsf{P} \\
\quad \mathsf{P}\ f & \circledast & \mathsf{P}\ x & = \mathsf{P}\ (f\ x) \\
\quad (u :\!*\!: a) & \circledast & \mathsf{P}\ x & = (\mathsf{P}\ ((\$x)\circ) \circledast u) :\!*\!: a \\
\quad u & \circledast & (v :\!*\!: a) & = (\mathsf{P}\ (\circ) \circledast u \circledast v) :\!*\!: a
\end{array}$$

The proof that $fmap\ f\ x = pure\ f \circledast x$ is an easy induction from the definitions, and the flattening formula (3) is proved in Appendix A.2. In Appendix A.3 we prove that the last three clauses above indeed define a total binary operator, and that *pure* and $\circledast$ satisfy the necessary idiom laws (and *imap* the necessary functor laws), so $\mathsf{Batch}\ a\ b$ is a lawful instance of the Applicative class.

We are going to consider just one traversal using Batch, namely *traverse batch*, where *batch* is defined by

$$\begin{array}{l}
batch :: a \to \mathsf{Batch}\ a\ b\ b \\
batch\ x = \mathsf{P}\ id :\!*\!: x
\end{array}$$

In Appendix A.4 we prove that *batch* has the useful property

$$u :\!*\!: x = u \circledast batch\ x \qquad (4)$$

Repeated applications of this fact give us the *translation* formula:

$$\mathsf{P}\ f\ :\!*\!:_{i=1}^{n}\ x_i = pure\ f\ \circledast_{i=1}^{n}\ batch\ x_i \qquad (5)$$

After some computations have been scheduled for batch processing, we may want to execute them. Accordingly we define the function *runWith*:

$$\begin{array}{l}
runWith :: \mathsf{Applicative}\ m \Rightarrow (a \to m\ b) \to \mathsf{Batch}\ a\ b\ c \to m\ c \\
runWith\ f\ (\mathsf{P}\ x) \quad = pure\ x \\
runWith\ f\ (u :\!*\!: x) = runWith\ f\ u \circledast f\ x
\end{array}$$

In effect, $runWith\ f$ replaces the constructors of Batch by the *pure* and $\circledast$ of idiom m, while applying f to the contained elements:

$$runWith\ f\ (\mathsf{P}\ g\ :\!*\!:_{i=1}^{n}\ x_i) = pure\ g\ \circledast_{i=1}^{n}\ f\ x_i$$

This result has a simple proof by induction, which we omit.

The first fact we need of *runWith* is that $runWith\ f \circ batch = f$:

$$\begin{array}{ll}
 & runWith\ f\ (batch\ x) \\
= & [\![\ \text{definition of}\ batch\ ]\!] \\
 & runWith\ f\ (\mathsf{P}\ id :\!*\!: x) \\
= & [\![\ \text{definition of}\ runWith\ ]\!] \\
 & pure\ id \circledast f\ x \\
= & [\![\ \text{identity law of idioms}\ ]\!] \\
 & f\ x
\end{array}$$

The second fact is that, when $f :: \mathsf{A} \to \mathsf{M\ B}$, the function $runWith\ f$, still polymorphic in c, is an idiom morphism from Batch A B to M:

$$\begin{array}{l}
runWith\ f\ (pure\ x) \ \ = pure\ x \\
runWith\ f\ (u \circledast v) = runWith\ f\ u \circledast runWith\ f\ v
\end{array}$$

The first equation is immediate by definitions. For the second, suppose without loss of generality that u and v take the form

$$\begin{array}{l}
u = \mathsf{P}\ g\ :\!*\!:_{i=1}^{m}\ x_i \\
v = \mathsf{P}\ h\ :\!*\!:_{i=m+1}^{n}\ x_i
\end{array}$$

Given what we know about *runWith*, we obtain

$$runWith\ f\ u = pure\ g\ \langle\circledast\rangle_{i=1}^{m}\ f\ x_i$$
$$runWith\ f\ v = pure\ h\ \langle\circledast\rangle_{i=m+1}^{n}\ f\ x_i$$

The flattening formula of $\langle\circledast\rangle$ (2) now gives

$$runWith\ f\ u \langle\circledast\rangle runWith\ f\ v = pure\ (g \circ_{m,n} h)\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i$$

But we can also appeal to the flattening formula of $:\!*\!:$ (3) to obtain

$$\begin{aligned} & runWith\ f\ (u \langle\circledast\rangle v) \\ & = runWith\ f\ (\mathsf{P}\ (g \circ_{m,n} h)\ :\!*\!:_{i=1}^{n}\ x_i) \\ & = pure\ (g \circ_{m,n} h)\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i \end{aligned}$$

completing the proof that *runWith f* is an idiom morphism.

9. Proof of the Representation Theorem

Let T be traversable and let $t :: \mathsf{T}\ \mathsf{A}$, so

$$traverse\ batch\ t :: \mathsf{Batch}\ \mathsf{A}\ b\ (\mathsf{T}\ b)$$

Crucially, the obtained value is still polymorphic in b. It follows from the definition of Batch that

$$traverse\ batch\ t = \mathsf{P}\ make\ :\!*\!:_{i=1}^{n}\ a_i \tag{6}$$

for some polymorphic function $make :: b \to \cdots \to b \to \mathsf{T}\ b$ of arity n and n values a_i, all of type A. Thus we have constructively obtained n, *make*, and $a_1, \ldots, a_n$. Our aim is to prove that that *make* is the unique make function yielding t, and that $a_1, \ldots, a_n$ are the unique values for which $t = make\ a_1 \ldots a_n$. Moreover, we want to prove that the general formula stated for $traverse\ f\ (make\ x_1 \ldots x_n)$ in the theorem holds.

For illustration, let us first consider a concrete example for (6). Let $\mathsf{T} = \mathsf{Tree}$, $t = \mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ 1)\ (\mathsf{Tip}\ 2))\ (\mathsf{Tip}\ 3)$, and

```
instance Traversable Tree where
  traverse f (Tip x)
    = pure Tip <⊛> f x
  traverse f (Bin u v)
    = pure (flip Bin) <⊛> traverse f v <⊛> traverse f u
```

Then it turns out we have

$$\begin{aligned} & traverse\ batch\ t \\ & = \mathsf{P}\ (\lambda x\ y\ z \to \mathsf{Bin}\ (\mathsf{Bin}\ (\mathsf{Tip}\ z)\ (\mathsf{Tip}\ y))\ (\mathsf{Tip}\ x)) :\!*\!: 3 :\!*\!: 2 :\!*\!: 1 \end{aligned}$$

For the $make :: b \to b \to b \to \mathsf{Tree}\ b$ and a_1, a_2, a_3 extracted from this, we indeed have $t = make\ a_1\ a_2\ a_3$, and the other claims (*make* being a make function, uniqueness, and the formula for $traverse\ f\ (make\ x_1\ x_2\ x_3)$) also hold.

Back to the general case. We start by proving

$$traverse\ f\ t = pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ f\ a_i \tag{7}$$

for an arbitrary $f :: \mathsf{A} \to \mathsf{M}\ \mathsf{B}$ (with A fixed as above, but a free choice of M and B). This result will be referred to as the *weak construction* formula.

$$\begin{aligned} & traverse\ f\ t \\ = & \ [\![\ \text{since } f = runWith\ f \circ batch\ ]\!] \\ & traverse\ (runWith\ f \circ batch)\ t \\ = & \ [\![\ \text{naturality } (runWith\ f \text{ is an idiom morphism})\ ]\!] \\ & runWith\ f\ (traverse\ batch\ t) \\ = & \ [\![\ \text{normal form (6)}\ ]\!] \\ & runWith\ f\ (\mathsf{P}\ make\ :\!*\!:_{i=1}^{n}\ a_i) \\ = & \ [\![\ \text{property of } runWith\ ]\!] \\ & pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ f\ a_i \end{aligned}$$

The weak construction formula can be used to prove the *reconstruction* formula

$$t = make\ a_1 \ldots a_n \tag{8}$$

For the proof we take M to be the identity idiom:

$$\begin{aligned} & \mathsf{Identity}\ t \\ = & \ [\![\ \text{unitarity law}\ ]\!] \\ & traverse\ \mathsf{Identity}\ t \\ = & \ [\![\ \text{weak construction formula (7)}\ ]\!] \\ & pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{Identity}\ a_i \\ = & \ [\![\ \text{calculation in the identity idiom}\ ]\!] \\ & \mathsf{Identity}\ (make\ a_1 \ldots a_n) \end{aligned}$$

Dropping the $\mathsf{Identity}$ wrapper yields the result.

Putting the two formulae together, we obtain

$$traverse\ f\ (make\ a_1 \ldots a_n) = pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ f\ a_i$$

But the Representation Theorem claims more, namely that

$$traverse\ f\ (make\ x_1 \ldots x_n) = pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i \tag{9}$$

for all $x_i :: \mathsf{A}'$ and $f :: \mathsf{A}' \to \mathsf{M}\ \mathsf{B}$. We call this the *strong construction* formula. For the proof it is sufficient (by injectivity of data constructors) to show that

$$\begin{aligned} & \mathsf{P}\ (\lambda x_1 \ldots x_n \to traverse\ f\ (make\ x_1 \ldots x_n))\ :\!*\!:_{i=1}^{n}\ a_i \\ & \quad = \mathsf{P}\ (\lambda x_1 \ldots x_n \to pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i)\ :\!*\!:_{i=1}^{n}\ a_i \end{aligned}$$

To this end, we use the linearity law of *traverse*. For brevity, we write C instead of $\mathsf{Compose}$:

$$\begin{aligned} & \mathsf{C}\ (\mathsf{P}\ (\lambda x_1 \ldots x_n \to traverse\ f\ (make\ x_1 \ldots x_n))\ :\!*\!:_{i=1}^{n}\ a_i) \\ = & \ [\![\ \text{flattening formula of } :\!*\!: \text{(3)}\ ]\!] \\ & \mathsf{C}\ (\mathsf{P}\ (traverse\ f) \langle\circledast\rangle (\mathsf{P}\ make\ :\!*\!:_{i=1}^{n}\ a_i)) \\ = & \ [\![\ \mathsf{P} = pure \text{ and property of } imap\ ]\!] \\ & \mathsf{C}\ (imap\ (traverse\ f)\ (\mathsf{P}\ make\ :\!*\!:_{i=1}^{n}\ a_i)) \\ = & \ [\![\ \text{normal form (6)}\ ]\!] \\ & \mathsf{C}\ (imap\ (traverse\ f)\ (traverse\ batch\ t)) \\ = & \ [\![\ \text{definition of } \langle\odot\rangle\ ]\!] \\ & (traverse\ f \langle\odot\rangle traverse\ batch)\ t \\ = & \ [\![\ \text{linearity law}\ ]\!] \\ & traverse\ (f \langle\odot\rangle batch)\ t \\ = & \ [\![\ \text{weak construction formula (7)}\ ]\!] \\ & pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ (f \langle\odot\rangle batch)\ a_i \\ = & \ [\![\ \text{definition of } \langle\odot\rangle\ ]\!] \\ & pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{C}\ (imap\ f\ (batch\ a_i)) \\ = & \ [\![\ \text{definitions of } batch \text{ and } imap\ ]\!] \\ & pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_i) \\ = & \ [\![\ \text{definition of } pure \text{ in idiom composition}\ ]\!] \\ & \mathsf{C}\ (pure\ (pure\ make))\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_i) \\ = & \ [\![\ \text{definition of } pure \text{ in batch idiom}\ ]\!] \\ & \mathsf{C}\ (\mathsf{P}\ (pure\ make))\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_i) \\ = & \ [\![\ \text{formula proved in Appendix A.5}\ ]\!] \\ & \mathsf{C}\ (\mathsf{P}\ (\lambda x_1 \ldots x_n \to pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i)\ :\!*\!:_{i=1}^{n}\ a_i) \end{aligned}$$

Dropping the C wrapper yields the result.

The next task is to show that *make* is a make function. We have:

$$\begin{aligned} & \mathsf{Identity}\ (tmap\ f\ (make\ x_1 \ldots x_n)) \\ = & \ [\![\ \text{unitarity law}\ ]\!] \\ & traverse\ \mathsf{Identity}\ (tmap\ f\ (make\ x_1 \ldots x_n)) \\ = & \ [\![\ traverse \text{ is natural in the elements}\ ]\!] \\ & traverse\ (\mathsf{Identity} \circ f)\ (make\ x_1 \ldots x_n) \\ = & \ [\![\ \text{strong construction formula (9)}\ ]\!] \\ & pure\ make\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{Identity}\ (f\ x_i) \\ = & \ [\![\ \text{calculation in the identity idiom}\ ]\!] \\ & \mathsf{Identity}\ (make\ (f\ x_1) \ldots (f\ x_n)) \end{aligned}$$

Dropping the $\mathsf{Identity}$ wrapper yields the first make condition. It remains to show that *make* satisfies the *contents*-property. Since *contents* is defined as a traversal, we again use the strong construction formula:

$$\begin{array}{l} \quad \mathit{contents}\ (\mathit{make}\ x_1 \ldots x_n) \\ = \quad [\![\ \text{definition of } \mathit{contents}\ ]\!] \\ \quad \mathit{getConst}\ (\mathit{traverse}\ (\lambda x \to \mathsf{Const}\ [x])\ (\mathit{make}\ x_1 \ldots x_n)) \\ = \quad [\![\ \text{strong construction formula (9)}\ ]\!] \\ \quad \mathit{getConst}\ (\mathit{pure}\ \mathit{make}\ \circledast_{i=1}^{n}\ \mathsf{Const}\ [x_i]) \\ = \quad [\![\ \text{definitions of } \mathit{pure} \text{ and } \circledast \text{ in Const idiom}\ ]\!] \\ \quad \mathit{getConst}\ (\mathsf{Const}\ ([\,] +\!\!+_{i=1}^{n} [x_i])) \\ = \quad [\![\ \text{definitions of } \mathit{getConst} \text{ and } +\!\!+\ ]\!] \\ \quad [x_1, \ldots, x_n] \end{array}$$

We have shown that, given $t :: \mathsf{T\ A}$, there exists some make function *make* and values $a_1, \ldots, a_n$ of type A (in fact, defined by (6)) such that $t = \mathit{make}\ a_1 \ldots a_n$. But what about uniqueness? Perhaps there is some other representation $t = \mu\ b_1 \ldots b_m$ with some make function μ of arity m and elements $b_1, \ldots, b_m$. To prove that there is not, suppose we can show that for any make function $\mu :: b \to \cdots \to b \to \mathsf{T}\ b$ of arity m

$$\mathit{traverse}\ f\ (\mu\ x_1 \ldots x_m) = \mathit{pure}\ \mu\ \circledast_{i=1}^{m}\ f\ x_i \tag{10}$$

for all $x_i :: \mathsf{A}'$ and $f :: \mathsf{A}' \to \mathsf{M\ B}$. This generalises the strong construction formula (9) in that μ is no longer defined by (6).

Now assume that for $t = \mathit{make}\ a_1 \ldots a_n$ there is some other representation $t = \mu\ b_1 \ldots b_m$, so the following equation holds:

$$\mathit{make}\ a_1 \ldots a_n = \mu\ b_1 \ldots b_m$$

Then by (10) we have

$$\mathit{pure}\ \mathit{make}\ \circledast_{i=1}^{n}\ f\ a_i = \mathit{pure}\ \mu\ \circledast_{i=1}^{m}\ f\ b_i$$

In particular, taking $f = \mathit{batch}$ and using the translation formula (5), we obtain

$$\mathsf{P}\ \mathit{make}\ {:}{*}{:}_{i=1}^{n}\ a_i = \mathsf{P}\ \mu\ {:}{*}{:}_{i=1}^{m}\ b_i$$

and so $n = m$, $a_i = b_i$ for each i, and $\mathit{make} = \mu$.

It remains to prove (10), for a given make function μ of arity m. To this end, define make' by the condition

$$\mathit{traverse}\ \mathit{batch}\ (\mu\ 1 \ldots m) = \mathsf{P}\ \mathit{make}'\ {:}{*}{:}_{i=1}^{p}\ k_i$$

for some p and natural numbers $k_1, \ldots, k_p$. This condition is (6) for $\mu\ 1 \ldots m$ and the previous results persist, especially the reconstruction formula (8) and the strong construction formula (9):

$$\begin{array}{ll} \mu\ 1 \ldots m & = \mathit{make}'\ k_1 \ldots k_p \\ \mathit{traverse}\ f\ (\mathit{make}'\ x_1 \ldots x_p) & = \mathit{pure}\ \mathit{make}'\ \circledast_{i=1}^{p}\ f\ x_i \end{array}$$

We also know make' to be a make function.

Now we can argue:

$$\begin{array}{l} \quad [1, \ldots, m] \\ = \quad [\![\ \text{since } \mu \text{ is a make function by assumption}\ ]\!] \\ \quad \mathit{contents}\ (\mu\ 1 \ldots m) \\ = \quad [\![\ \text{reconstruction formula (8) in the above version}\ ]\!] \\ \quad \mathit{contents}\ (\mathit{make}'\ k_1 \ldots k_p) \\ = \quad [\![\ \text{since } \mathit{make}' \text{ is also a make function}\ ]\!] \\ \quad [k_1, \ldots, k_p] \end{array}$$

So $p = m$, $k_i = i$ and thus $\mu\ 1 \ldots m = \mathit{make}'\ 1 \ldots m$. Finally we have, for any $x_1, \ldots, x_m$, and taking x to be some function for which $x\ i = x_i$ for $1 \leqslant i \leqslant m$,

$$\begin{array}{l} \quad \mu\ x_1 \ldots x_m \\ = \quad [\![\ \text{since } \mu \text{ is a make function, thus natural}\ ]\!] \\ \quad \mathit{tmap}\ x\ (\mu\ 1 \ldots m) \\ = \quad [\![\ \mu\ 1 \ldots m = \mathit{make}'\ 1 \ldots m, \text{ just shown}\ ]\!] \\ \quad \mathit{tmap}\ x\ (\mathit{make}'\ 1 \ldots m) \\ = \quad [\![\ \text{since } \mathit{make}' \text{ is also a make function}\ ]\!] \\ \quad \mathit{make}'\ x_1 \ldots x_m \end{array}$$

Hence $\mu = \mathit{make}'$ and (10) is proved, because in the case of $\mu = \mathit{make}'$ it is the strong construction formula (9) in the above version for make'.

10. Discussion

We started out with a simple problem about effectful functions on the state monad and binary trees, and came up with a general inversion law that was independent of both the nature of the effects and the details of the datatype. Though simple to state, proof of the law seemed difficult—until we came up with the right tool. This tool is the Representation Theorem in Section 5, relating traversable data structures and their traversals to their shape and contents. So in addition to proving a simple program correct, we have discovered and developed a useful new tool.

Like many cherished tools in a crowded toolbox, the Representation Theorem is surprisingly versatile. Using it, we have resolved several more general open questions, in addition to the specific programming problem we designed it for: the property of 'naturality' in the datatype, the illegality of half-hearted and duplicitous traversals (Gibbons and Oliveira 2009), the correspondence between traversable datatypes and finitary containers (Moggi et al. 1999), and more precisely the bijection between traversal strategies for a data structure (shape) and permutations of its elements (Jaskelioff and Rypáček 2012).

More generally, *all* there is to know about lawful instances of Traversable can be learned from the Representation Theorem, because it is equivalent to the laws. Indeed, the theorem implies all the laws of *traverse* (which—without naturality in b, as the attentive reader may have noticed—suffice to establish the theorem).

The correspondence between traversable functors and finitary containers, long held as a folklore belief and the essence of our representation theorem, was independently proved by O'Connor and Jaskelioff (2012). Their proof, in Coq, is quite different from ours. It relies on coalgebraic machinery that did not surface in our proof. The two proofs share reliance on a special idiom related to the free applicative functor, though: in their case, that specialised idiom is the dependent type $\Pi_{n:\mathbb{N}}\ (\mathsf{Vec\ A}\ n, \mathsf{Vec\ B}\ n \to \mathsf{C})$, which is isomorphic to our type Batch A B C. We could have used size-indexed vectors, too; after all, one can fake dependent types quite well in Haskell these days. Instead, we have made judicious use of meta-level indexing (such as '$\circ_{m,n}$') and ellipses ('…') throughout; all these notations could easily be defined inductively, and the corresponding proofs made explicitly inductive.

There are still some avenues for future work.

- We believe that for a large range of datatypes T (including all regular datatypes), the naturality properties of *traverse* (but not 'naturality' in the datatype) actually hold for all polymorphic functions of type $\mathsf{Applicative}\ m \Rightarrow (a \to m\ b) \to \mathsf{T}\ a \to m\ (\mathsf{T}\ b)$; they should be *free theorems* (Wadler 1989; Voigtländer 2009).
- We conjecture the maybe somewhat surprising fact that, in Set, the monadic variant of *traverse*'s type is no more accommodating than the idiomatic one. In general, there are more inhabitants of a type of the form $\mathsf{Monad}\ m \Rightarrow \tau$ than of one of the form $\mathsf{Applicative}\ m \Rightarrow \tau$, where m is a free type variable in τ. After all, since every monad is an idiom, the terms written under the monad constraint can use the applicative primitives *pure* and $\circledast$ as well as the more expressive monadic primitive $\gg\!=$. Since not every idiom is a monad, the converse is not true. Nevertheless, we conjecture that for every term $t :: \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to \mathsf{T}\ a \to m\ (\mathsf{T}\ b)$ there is a term $t' :: \mathsf{Applicative}\ m \Rightarrow (a \to m\ b) \to \mathsf{T}\ a \to m\ (\mathsf{T}\ b)$ such that $t = t'$ when m is instantiated to any monad. Hence, even if we are interested mainly in monadic traversals, there is nothing to be gained from restricting to monads.

Acknowledgements

This paper has been through a long period of gestation, and has benefitted from interaction with numerous colleagues. We would like to thank the members of IFIP Working Group 2.1 and of the Algebra of Programming group in Oxford; Conor McBride and Ross Paterson, for helpful discussions; the anonymous referees of previous versions; Graham Hutton and Diane Fulger (Hutton and Fulger 2008), from whom we learnt about the tree relabelling problem; and especially Ondřej Rypáček, for sharing with us the unpublished note (Rypáček 2010) that pointed us towards the notion of 'naturality' in the datatype discussed in Section 6.

Jeremy Gibbons was supported by EPSRC grant EP/G034516/1 on *Reusability and Dependent Types*. Stefan Mehner was supported by DFG grant VO 1512-1/2.

References

R. Bird and L. Meertens. Nested datatypes. In *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998. doi: 10.1007/BFb0054285.

P. Capriotti and A. Kaposi. Free applicative functors. University of Nottingham. `http://paolocapriotti.com/blog/2013/04/03/free-applicative-functors/`, Apr. 2013.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2004. doi: 10.1007/978-3-540-24849-1_14.

J. Gibbons and B. C. d. S. Oliveira. The essence of the Iterator pattern. *Journal of Functional Programming*, 19(3,4):377–402, 2009. doi: 10.1017/S0956796809007291.

J.-Y. Girard. Normal functors, power series and λ-calculus. *Annals of Pure and Applied Logic*, 37(2):129–177, 1988. doi: 10.1016/0168-0072(88)90025-5.

G. Hutton and D. Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *Trends in Functional Programming*, preproceedings, Nijmegen, The Netherlands, 2008.

M. Jaskelioff and O. Rypáček. An investigation of the laws of traversals. In *Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 40–49, 2012. doi: 10.4204/EPTCS.76.5.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi: 10.1017/S0956796807006326.

E. Moggi, G. Bellè, and B. Jay. Monads, shapely functors, and traversals. *Electronic Notes in Theoretical Computer Science*, 29:187–208, 1999. doi: 10.1016/S1571-0661(05)80316-0. Proceedings of *Category Theory and Computer Science*.

R. O'Connor and M. Jaskelioff. On the static nature of traversals. `http://r6.ca/blog/20121209T182914Z.html`, Dec. 2012.

O. Rypáček. Labelling polynomial functors: A coherent approach. Manuscript, Mar. 2010.

J. Voigtländer. Free theorems involving type constructor classes. In *International Conference on Functional Programming*, pages 173–184. ACM, 2009. doi: 10.1145/1596550.1596577.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

A. Some lengthy but not so interesting proofs

A.1 Formula for backwards idiom

In this appendix (A.1) we extend the notation $\oplus_{i=1}^{n}$, defined in Section 6, to abbreviate repeated application of some operator $\oplus$ to a sequence of arguments with decreasing rather than increasing indices:

$$x \oplus_{i=n}^{1} x_i = ((x \oplus x_n) \oplus \cdots \oplus x_2) \oplus x_1$$

We prove

$$\mathsf{B}\ (pure\ f)\ \circledast_{i=1}^{n}\ \mathsf{B}\ x_i = \mathsf{B}\ (pure\ (swap_n\ f)\ \circledast_{i=n}^{1}\ x_i)$$

where $swap_n\ f\ x_n \ldots x_1 = f\ x_1 \ldots x_n$, by induction on n. The base case $n = 0$ is trivial, as both sides equal $\mathsf{B}\ (pure\ f)$. The induction step is

$$\begin{aligned}
& \mathsf{B}\ (pure\ f)\ \circledast_{i=1}^{n+1}\ \mathsf{B}\ x_i \\
= \quad & [\![\ \text{splitting off the last application}\ ]\!] \\
& \mathsf{B}\ (pure\ f)\ \circledast_{i=1}^{n}\ \mathsf{B}\ x_i \circledast \mathsf{B}\ x_{n+1} \\
= \quad & [\![\ \text{induction hypothesis}\ ]\!] \\
& \mathsf{B}\ (pure\ (swap_n\ f)\ \circledast_{i=n}^{1}\ x_i) \circledast \mathsf{B}\ x_{n+1} \\
= \quad & [\![\ \text{definition of } \circledast \text{ in backwards idiom}\ ]\!] \\
& \mathsf{B}\ (pure\ (flip\ (\$)) \circledast x_{n+1} \circledast (pure\ (swap_n\ f)\ \circledast_{i=n}^{1}\ x_i)) \\
= \quad & [\![\ \text{flattening formula of } \circledast\text{, see below}\ ]\!] \\
& \mathsf{B}\ (pure\ (swap_{n+1}\ f)\ \circledast_{i=n+1}^{1}\ x_i)
\end{aligned}$$

The *flattening* formula applied in the last step is stated as (2) in Section 6 as well as in more general form in Appendix A.2, where it is proved. The justification of its specific application here is that $swap_{n+1}\ f\ x_{n+1} \ldots x_1 = flip\ (\$)\ x_{n+1}\ (swap_n\ f\ x_n \ldots x_1)$.

A.2 Flattening formula

In this appendix we prove the flattening formula

$$\begin{aligned}
&(pure\ g \oplus_{i=1}^{m} x_i) \circledast (pure\ f \oplus_{i=m+1}^{n} x_i) \\
&\quad = pure\ (g \circ_{m,n} f) \oplus_{i=1}^{n} x_i
\end{aligned}$$

for all $0 \leqslant m \leqslant n$, where $\circ_{m,n}$ is as defined in Section 6 and $\oplus$ is a binary operator we assume to satisfy three properties:

$$\begin{aligned}
pure\ f \circledast pure\ x\ &= pure\ (f\ x) \\
(u \oplus v) \circledast pure\ x &= pure\ (\$x) \circledast (u \oplus v) \\
u \circledast (v \oplus w)\ &= (pure\ (\circ) \circledast u \circledast v) \oplus w
\end{aligned}$$

We are interested in two special cases: If $\oplus$ is idiomatic application in some lawful idiom, we get the flattening formula (2). The three assumed properties are indeed satisfied, as they are simply the homomorphism, interchange, and composition law for idioms. If on the other hand $\oplus$ is the constructor $:*:$ of the batch idiom, we get the flattening formula (3). The first and third assumption on $\oplus$ are the first and third defining clause of $\circledast$ for the batch idiom. The second assumption also holds:

$$\begin{aligned}
& (u :*: v) \circledast pure\ x \\
= \quad & [\![\ \text{second clause of } \circledast\ ]\!] \\
& (pure\ ((\$x)\circ) \circledast u) :*: v \\
= \quad & [\![\ \text{first clause of } \circledast\ ]\!] \\
& (pure\ (\circ) \circledast pure\ (\$x) \circledast u) :*: v \\
= \quad & [\![\ \text{third clause of } \circledast\ ]\!] \\
& pure\ (\$x) \circledast (u :*: v)
\end{aligned}$$

In this case we need not assume $\circledast$ to be lawful.

Now for the proof. We first establish the special case when $m = 0$, namely

$$pure\ g \circledast (pure\ f \oplus_{i=1}^{n} x_i) = pure\ (g \circ_{0,n} f) \oplus_{i=1}^{n} x_i \qquad (11)$$

The proof is by induction on n. The base case $n = 0$ is just the first assumption, and for the inductive case we argue:

$$\begin{aligned}
& pure\ g \circledast (pure\ f \oplus_{i=1}^{n+1} x_i) \\
= \quad & [\![\ \text{splitting off the last } \oplus\ ]\!] \\
& pure\ g \circledast ((pure\ f \oplus_{i=1}^{n} x_i) \oplus x_{n+1}) \\
= \quad & [\![\ \text{third assumption}\ ]\!] \\
& (pure\ (\circ) \circledast pure\ g \circledast (pure\ f \oplus_{i=1}^{n} x_i)) \oplus x_{n+1} \\
= \quad & [\![\ \text{first assumption}\ ]\!] \\
& (pure\ ((\circ)\ g) \circledast (pure\ f \oplus_{i=1}^{n} x_i)) \oplus x_{n+1}
\end{aligned}$$

$$
\begin{array}{ll}
= & \llbracket \text{ induction hypothesis, since } ((\circ)\ g) \circ_{0,n} f = g \circ_{0,n+1} f \ \rrbracket \\
 & \mathit{pure}\ (g \circ_{0,n+1} f) \oplus_{i=1}^{n+1} x_i
\end{array}
$$

Using (11), we prove the flattening formula in general, again by induction on n. The base case $0 < n = m$ is proved by

$$
\begin{array}{ll}
 & (\mathit{pure}\ g \oplus_{i=1}^{m} x_i) \langle\circledast\rangle\ \mathit{pure}\ f \\
= & \llbracket \text{ second assumption } \rrbracket \\
 & \mathit{pure}\ (\$f) \langle\circledast\rangle (\mathit{pure}\ g \oplus_{i=1}^{m} x_i) \\
= & \llbracket \text{ (11) } \rrbracket \\
 & \mathit{pure}\ ((\$f) \circ_{0,m} g) \oplus_{i=1}^{m} x_i \\
= & \llbracket \text{ simplification—see below } \rrbracket \\
 & \mathit{pure}\ (g \circ_{m,m} f) \oplus_{i=1}^{m} x_i
\end{array}
$$

The simplification is justified by

$$
\begin{array}{l}
\ \ ((\$f) \circ_{0,m} g)\ x_1 \ldots x_m \\
= (\$f)\ (g\ x_1 \ldots x_m) \\
= g\ x_1 \ldots x_m\ f \\
= (g \circ_{m,m} f)\ x_1 \ldots x_m
\end{array}
$$

For the inductive case, we argue:

$$
\begin{array}{ll}
 & (\mathit{pure}\ g \oplus_{i=1}^{m} x_i) \langle\circledast\rangle (\mathit{pure}\ f \oplus_{i=m+1}^{n+1} x_i) \\
= & \llbracket \text{ splitting off the last } \oplus \ \rrbracket \\
 & (\mathit{pure}\ g \oplus_{i=1}^{m} x_i) \langle\circledast\rangle ((\mathit{pure}\ f \oplus_{i=m+1}^{n} x_i) \oplus x_{n+1}) \\
= & \llbracket \text{ third assumption } \rrbracket \\
 & (\mathit{pure}\ (\circ) \langle\circledast\rangle (\mathit{pure}\ g \oplus_{i=1}^{m} x_i) \langle\circledast\rangle (\mathit{pure}\ f \oplus_{i=m+1}^{n} x_i)) \\
 & \quad \oplus x_{n+1} \\
= & \llbracket \text{ (11) } \rrbracket \\
 & ((\mathit{pure}\ ((\circ) \circ_{0,m} g) \oplus_{i=1}^{m} x_i) \langle\circledast\rangle (\mathit{pure}\ f \oplus_{i=m+1}^{n} x_i)) \oplus x_{n+1} \\
= & \llbracket \text{ induction hypothesis } \rrbracket \\
 & (\mathit{pure}\ (((\circ) \circ_{0,m} g) \circ_{m,n} f) \oplus_{i=1}^{n} x_i) \oplus x_{n+1} \\
= & \llbracket \text{ simplification—see below } \rrbracket \\
 & (\mathit{pure}\ (g \circ_{m,n+1} f) \oplus_{i=1}^{n} x_i) \oplus x_{n+1} \\
= & \llbracket \text{ combining again with the last } \oplus \ \rrbracket \\
 & \mathit{pure}\ (g \circ_{m,n+1} f) \oplus_{i=1}^{n+1} x_i
\end{array}
$$

This time, the simplification is justified by

$$
\begin{array}{l}
\ \ (((\circ) \circ_{0,m} g) \circ_{m,n} f)\ x_1 \ldots x_n\ x_{n+1} \\
= ((\circ) \circ_{0,m} g)\ x_1 \ldots x_m\ (f\ x_{m+1} \ldots x_n)\ x_{n+1} \\
= (\circ)\ (g\ x_1 \ldots x_m)\ (f\ x_{m+1} \ldots x_n)\ x_{n+1} \\
= (g\ x_1 \ldots x_m \circ f\ x_{m+1} \ldots x_n)\ x_{n+1} \\
= g\ x_1 \ldots x_m\ (f\ x_{m+1} \ldots x_n\ x_{n+1}) \\
= (g \circ_{m,n+1} f)\ x_1 \ldots x_{n+1}
\end{array}
$$

A.3 Correctness of the Applicative instance for Batch

Totality The definition of $\langle\circledast\rangle$ for the batch idiom is not obviously total, because its second and third clauses contain recursive calls that are not structurally smaller than the left-hand side. To prove that this function is nevertheless total, we introduce a notion of size for values of batch idiom type:

$$
\begin{array}{l}
\mathit{size} :: \mathsf{Batch}\ a\ b\ c \to \mathsf{Int} \\
\mathit{size}\ (\mathsf{P}\ x) \quad = 0 \\
\mathit{size}\ (u :\!*\!: a) = \mathit{size}\ u + 1
\end{array}
$$

To conclude that the definition of $\langle\circledast\rangle$ is indeed terminating, we show that the sum of the sizes of the arguments for the recursive calls is smaller than the sum of the sizes of the original arguments. This property depends mutually on the invariant that $\mathit{size}\ (u \langle\circledast\rangle v) = \mathit{size}\ u + \mathit{size}\ v$, so we prove both together by induction on the sum of the sizes of the arguments.

1. For the first clause the recursive calls are vacuously smaller: there are none. Also we have that

$$
\begin{array}{ll}
 & \mathit{size}\ (\mathsf{P}\ f \langle\circledast\rangle \mathsf{P}\ x) \\
= & \llbracket \text{ first clause of } \langle\circledast\rangle \ \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ (f\ x)) \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ f) + \mathit{size}\ (\mathsf{P}\ x)
\end{array}
$$

2. For the second clause, we have one recursive call.

$$
\begin{array}{ll}
 & \mathit{size}\ (u :\!*\!: a) + \mathit{size}\ (\mathsf{P}\ x) \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ u + 1 \\
> & \llbracket \text{ basic arithmetic } \rrbracket \\
 & \mathit{size}\ u \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ ((\$x)\circ)) + \mathit{size}\ u
\end{array}
$$

   Also the invariant is preserved:

$$
\begin{array}{ll}
 & \mathit{size}\ ((u :\!*\!: a) \langle\circledast\rangle (\mathsf{P}\ x)) \\
= & \llbracket \text{ second clause of } \langle\circledast\rangle \ \rrbracket \\
 & \mathit{size}\ (((\mathsf{P}\ ((\$x)\circ)) \langle\circledast\rangle u) :\!*\!: a) \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ ((\mathsf{P}\ ((\$x)\circ)) \langle\circledast\rangle u) + 1 \\
= & \llbracket \text{ induction hypothesis } \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ ((\$x)\circ)) + \mathit{size}\ u + 1 \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ (u :\!*\!: a) + \mathit{size}\ (\mathsf{P}\ x)
\end{array}
$$

3. For the third clause, we have two recursive calls.
   (a) For $\mathsf{P}\ (\circ) \langle\circledast\rangle u$:

$$
\begin{array}{ll}
 & \mathit{size}\ u + \mathit{size}\ (v :\!*\!: a) \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ u + \mathit{size}\ v + 1 \\
> & \llbracket \text{ basic arithmetic } \rrbracket \\
 & \mathit{size}\ u \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ (\circ)) + \mathit{size}\ u
\end{array}
$$

   (b) For $(\mathsf{P}\ (\circ) \langle\circledast\rangle u) \langle\circledast\rangle v$:

$$
\begin{array}{ll}
 & \mathit{size}\ u + \mathit{size}\ (v :\!*\!: a) \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ u + \mathit{size}\ v + 1 \\
> & \llbracket \text{ basic arithmetic } \rrbracket \\
 & \mathit{size}\ u + \mathit{size}\ v \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ (\circ)) + \mathit{size}\ u + \mathit{size}\ v \\
= & \llbracket \text{ induction hypothesis } \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ (\circ) \langle\circledast\rangle u) + \mathit{size}\ v
\end{array}
$$

   Also the invariant is preserved:

$$
\begin{array}{ll}
 & \mathit{size}\ (u \langle\circledast\rangle (v :\!*\!: a)) \\
= & \llbracket \text{ third clause of } \langle\circledast\rangle \ \rrbracket \\
 & \mathit{size}\ ((\mathsf{P}\ (\circ) \langle\circledast\rangle u \langle\circledast\rangle v) :\!*\!: a) \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ (\mathsf{P}\ (\circ) \langle\circledast\rangle u \langle\circledast\rangle v) + 1 \\
= & \llbracket \text{ induction hypothesis } \rrbracket \\
 & \mathit{size}\ u + \mathit{size}\ v + 1 \\
= & \llbracket \text{ definition of } \mathit{size} \ \rrbracket \\
 & \mathit{size}\ u + \mathit{size}\ (v :\!*\!: a)
\end{array}
$$

So the sum of the *size*s decreases in recursive calls, and the recursion is well-founded.

Lawfulness Given $\mathit{imap}\ f\ x = \mathit{pure}\ f \langle\circledast\rangle x$ (as mentioned, an easy induction from the definitions), the correctness of the Functor instance follows from the correctness of the Applicative instance. Specifically,

$$\mathit{imap}\ \mathit{id}\ x = \mathit{pure}\ \mathit{id} \langle\circledast\rangle x = x$$

by the identity law of idioms. Also,

$$\begin{aligned}
&\mathit{imap}\ (g \circ f)\ x \\
=\ &[\![\ \text{above property}\ ]\!] \\
&\mathit{pure}\ (g \circ f) \langle\circledast\rangle x \\
=\ &[\![\ \text{homomorphism law of idioms, twice}\ ]\!] \\
&\mathit{pure}\ (\circ) \langle\circledast\rangle \mathit{pure}\ g \langle\circledast\rangle \mathit{pure}\ f \langle\circledast\rangle x \\
=\ &[\![\ \text{composition law of idioms}\ ]\!] \\
&\mathit{pure}\ g \langle\circledast\rangle (\mathit{pure}\ f \langle\circledast\rangle x) \\
=\ &[\![\ \text{above property, twice}\ ]\!] \\
&\mathit{imap}\ g\ (\mathit{imap}\ f\ x)
\end{aligned}$$

We will now check the Applicative instance of the batch idiom. In what follows, $u :: \mathsf{Batch\ A\ B\ C}$, so $u = \mathsf{P}\ f :\!*\!:_{i=1}^{n} a_i$ for some f of type $\mathsf{B} \to \mathsf{B} \to \cdots \to \mathsf{B} \to \mathsf{C}$ (of arity n) and $a_i :: \mathsf{A}$ for all i.

The identity law holds by

$$\begin{aligned}
&\mathit{pure}\ \mathit{id} \langle\circledast\rangle u \\
=\ &[\![\ \text{definitions of } \mathit{pure} \text{ for Batch and of } u\ ]\!] \\
&\mathsf{P}\ \mathit{id} \langle\circledast\rangle (\mathsf{P}\ f :\!*\!:_{i=1}^{n} a_i) \\
=\ &[\![\ \text{flattening formula of } :\!*\!: (3)\ ]\!] \\
&\mathsf{P}\ f :\!*\!:_{i=1}^{n} a_i \\
=\ &[\![\ \text{definition of } u\ ]\!] \\
&u
\end{aligned}$$

The homomorphism law is trivial:

$$\mathit{pure}\ f \langle\circledast\rangle \mathit{pure}\ x = \mathsf{P}\ f \langle\circledast\rangle \mathsf{P}\ x = \mathsf{P}\ (f\ x) = \mathit{pure}\ (f\ x)$$

Next, the interchange law:

$$\begin{aligned}
&u \langle\circledast\rangle \mathit{pure}\ x \\
=\ &[\![\ \text{definitions}\ ]\!] \\
&\mathsf{P}\ f :\!*\!:_{i=1}^{n} a_i \langle\circledast\rangle \mathsf{P}\ x \\
=\ &[\![\ \text{flattening formula of } :\!*\!: (3)\ ]\!] \\
&\mathsf{P}\ (f \circ_{n,n} x) :\!*\!:_{i=1}^{n} a_i \\
=\ &[\![\ \text{flattening formula of } :\!*\!: (3), \text{ since } f \circ_{n,n} x = (\$x) \circ_{0,n} f\ ]\!] \\
&\mathsf{P}\ (\$x) \langle\circledast\rangle (\mathsf{P}\ f :\!*\!:_{i=1}^{n} a_i) \\
=\ &[\![\ \text{definitions}\ ]\!] \\
&\mathit{pure}\ (\$x) \langle\circledast\rangle u
\end{aligned}$$

For $f \circ_{n,n} x = (\$x) \circ_{0,n} f$, see the first simplification in Appendix A.2.

Finally, the composition law asserts

$$\mathit{pure}\ (\circ) \langle\circledast\rangle u \langle\circledast\rangle v \langle\circledast\rangle w = u \langle\circledast\rangle (v \langle\circledast\rangle w)$$

Here we need similar formulae for v and w to those we had for u. The proof comes down to using the flattening formula of $:\!*\!:$ (3) five times, and we omit the gory details.

A.4 Idiomatic application to *batch*

We prove $u \langle\circledast\rangle \mathit{batch}\ x = u :\!*\!: x$, which by the definition of *batch* is equivalent to

$$u \langle\circledast\rangle (\mathsf{P}\ \mathit{id} :\!*\!: x) = u :\!*\!: x$$

To prove the latter equation, we use the defining clauses of $\langle\circledast\rangle$ in the batch idiom. Firstly, if $u = \mathsf{P}\ f$, we can argue:

$$\begin{aligned}
&\mathsf{P}\ f \langle\circledast\rangle (\mathsf{P}\ \mathit{id} :\!*\!: x) \\
=\ &[\![\ \text{third clause of } \langle\circledast\rangle\ ]\!] \\
&(\mathsf{P}\ (\circ) \langle\circledast\rangle \mathsf{P}\ f \langle\circledast\rangle \mathsf{P}\ \mathit{id}) :\!*\!: x \\
=\ &[\![\ \text{first clause (twice), and } f \circ \mathit{id} = f\ ]\!] \\
&\mathsf{P}\ f :\!*\!: x \\
=\ &[\![\ \text{case assumption}\ ]\!] \\
&u :\!*\!: x
\end{aligned}$$

Secondly, if $u = v :\!*\!: a$, we argue:

$$\begin{aligned}
&(v :\!*\!: a) \langle\circledast\rangle (\mathsf{P}\ \mathit{id} :\!*\!: x) \\
=\ &[\![\ \text{third clause}\ ]\!] \\
&(\mathsf{P}\ (\circ) \langle\circledast\rangle (v :\!*\!: a) \langle\circledast\rangle \mathsf{P}\ \mathit{id}) :\!*\!: x \\
=\ &[\![\ \text{third clause}\ ]\!] \\
&(((\mathsf{P}\ (\circ) \langle\circledast\rangle \mathsf{P}\ (\circ) \langle\circledast\rangle v) :\!*\!: a) \langle\circledast\rangle \mathsf{P}\ \mathit{id}) :\!*\!: x \\
=\ &[\![\ \text{first clause}\ ]\!] \\
&(((\mathsf{P}\ ((\circ)\circ) \langle\circledast\rangle v) :\!*\!: a) \langle\circledast\rangle \mathsf{P}\ \mathit{id}) :\!*\!: x \\
=\ &[\![\ \text{second clause}\ ]\!] \\
&(\mathsf{P}\ ((\$\mathit{id})\circ) \langle\circledast\rangle (\mathsf{P}\ ((\circ)\circ) \langle\circledast\rangle v)) :\!*\!: a :\!*\!: x \\
=\ &[\![\ \text{composition law of idioms}\ ]\!] \\
&(\mathsf{P}\ (\circ) \langle\circledast\rangle \mathsf{P}\ ((\$\mathit{id})\circ) \langle\circledast\rangle \mathsf{P}\ ((\circ)\circ) \langle\circledast\rangle v) :\!*\!: a :\!*\!: x \\
=\ &[\![\ \text{first clause (twice)}\ ]\!] \\
&(\mathsf{P}\ ((\circ)\ ((\$\mathit{id})\circ)\ ((\circ)\circ)) \langle\circledast\rangle v) :\!*\!: a :\!*\!: x \\
=\ &[\![\ \text{claim, see below}\ ]\!] \\
&(\mathsf{P}\ \mathit{id} \langle\circledast\rangle v) :\!*\!: a :\!*\!: x \\
=\ &[\![\ \text{identity law of idioms}\ ]\!] \\
&v :\!*\!: a :\!*\!: x \\
=\ &[\![\ \text{case assumption}\ ]\!] \\
&u :\!*\!: x
\end{aligned}$$

The claim is that $(\circ)\ ((\$\mathit{id})\circ)\ ((\circ)\circ) = \mathit{id}$. For type reasons in the expression above, the function $(\circ)\ ((\$\mathit{id})\circ)\ ((\circ)\circ)$ has to take at least two arguments, so we reason by applying it to two arguments:

$$\begin{aligned}
&(\circ)\ ((\$\mathit{id})\circ)\ ((\circ)\circ)\ x\ y \\
&= (((\$\mathit{id})\circ) \circ ((\circ)\circ))\ x\ y \\
&= ((\$\mathit{id})\circ)\ ((\circ) \circ x)\ y \\
&= ((\$\mathit{id}) \circ (\circ) \circ x)\ y \\
&= (\$\mathit{id})\ ((\circ)\ (x\ y)) \\
&= (x\ y) \circ \mathit{id} \\
&= x\ y
\end{aligned}$$

A.5 Formula for composite of batch and another idiom

We prove the formula

$$\begin{aligned}
&\mathsf{C}\ (\mathsf{P}\ (\mathit{pure}\ g))\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_i) \\
&\quad = \mathsf{C}\ (\mathsf{P}\ (\lambda x_1 \ldots x_n \to \mathit{pure}\ g\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i)\ :\!*\!:_{i=1}^{n}\ a_i)
\end{aligned}$$

by induction over n. The case $n = 0$ is trivial, since both sides equal $\mathsf{C}\ (\mathsf{P}\ (\mathit{pure}\ g))$. The induction step is:

$$\begin{aligned}
&\mathsf{C}\ (\mathsf{P}\ (\mathit{pure}\ g))\ \langle\circledast\rangle_{i=1}^{n+1}\ \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_i) \\
=\ &[\![\ \text{splitting off the last application}\ ]\!] \\
&(\mathsf{C}\ (\mathsf{P}\ (\mathit{pure}\ g))\ \langle\circledast\rangle_{i=1}^{n}\ \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_i)) \langle\circledast\rangle \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_{n+1}) \\
=\ &[\![\ \text{induction hypothesis}\ ]\!] \\
&\mathsf{C}\ (\mathsf{P}\ (\lambda x_1 \ldots x_n \to \mathit{pure}\ g\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i)\ :\!*\!:_{i=1}^{n}\ a_i) \\
&\quad \langle\circledast\rangle \mathsf{C}\ (\mathsf{P}\ f :\!*\!: a_{n+1}) \\
=\ &[\![\ \text{definition of } \langle\circledast\rangle \text{ in idiom composition}\ ]\!] \\
&\mathsf{C}\ (\mathit{pure}\ (\langle\circledast\rangle) \langle\circledast\rangle (\mathsf{P}\ (\lambda x_1 \ldots x_n \to \mathit{pure}\ g\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i) \\
&\qquad\qquad :\!*\!:_{i=1}^{n}\ a_i) \\
&\qquad\quad \langle\circledast\rangle (\mathsf{P}\ f :\!*\!: a_{n+1})) \\
=\ &[\![\ \text{flattening formula of } :\!*\!: (3)\ ]\!] \\
&\mathsf{C}\ ((\mathsf{P}\ (\lambda x_1 \ldots x_n \to (\langle\circledast\rangle)\ (\mathit{pure}\ g\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i))\ :\!*\!:_{i=1}^{n}\ a_i) \\
&\quad \langle\circledast\rangle (\mathsf{P}\ f :\!*\!: a_{n+1})) \\
=\ &[\![\ \text{flattening formula of } :\!*\!: (3)\ ]\!] \\
&\mathsf{C}\ (\mathsf{P}\ (\lambda x_1 \ldots x_{n+1} \to (\mathit{pure}\ g\ \langle\circledast\rangle_{i=1}^{n}\ f\ x_i) \langle\circledast\rangle f\ x_{n+1}) \\
&\quad :\!*\!:_{i=1}^{n+1}\ a_i) \\
=\ &[\![\ \text{combining again with the last application}\ ]\!] \\
&\mathsf{C}\ (\mathsf{P}\ (\lambda x_1 \ldots x_{n+1} \to \mathit{pure}\ g\ \langle\circledast\rangle_{i=1}^{n+1}\ f\ x_i)\ :\!*\!:_{i=1}^{n+1}\ a_i)
\end{aligned}$$