Understanding Idiomatic Traversals Backwards and Forwards

Richard Bird, Jeremy Gibbons, Stefan Mehner, Tom Schrijvers, and Janis Voigtländer

July 3rd, 2013

Traversals

- What is a traversal (strategy), for a given datatype $\mathsf{T} :: * \to *$?
- J.G. and B.O. in "The Essence of the Iterator Pattern": A function of type

$$\texttt{traverse} :: (a \to \mathsf{M}\ b) \to \mathsf{T}\ a \to \mathsf{M}\ (\mathsf{T}\ b)$$

- … where $\mathsf{M} :: * \to *$ is a type constructor that captures effectful computations (think: monads, or idioms)
- … where in fact `traverse` should be polymorphic in such M (which hence should be written *m*), but *not* polymorphic in T
- … and where the behaviour of `traverse` should be governed by some laws

Traversals — Examples

Let: **data** Tree a = Tip a | Bin (Tree a) (Tree a).

Depth-first-traversal (left-to-right):

```
traverse :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
traverse f (Tip x)   = do x' <- f x
                          return (Tip x')
traverse f (Bin u v) = do u' <- traverse f u
                          v' <- traverse f v
                          return (Bin u' v')
```

or (equivalently):

```
traverse :: Applicative m => (a -> m b) -> Tree a -> m (Tree b)
traverse f (Tip x)   = pure Tip <*> f x
traverse f (Bin u v) = pure Bin <*> traverse f u
                                <*> traverse f v
```

Traversals — Examples

Let: **data** Tree a = Tip a | Bin (Tree a) (Tree a).

Depth-first-traversal (right-to-left):

$$
\begin{aligned}
&\texttt{traverse} :: \text{Monad } m \Rightarrow (a \to m\ b) \to \text{Tree } a \to m\ (\text{Tree } b)\\
&\texttt{traverse}\ f\ (\text{Tip } x) \quad = \textbf{do}\ x' \leftarrow f\ x\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \texttt{return}\ (\text{Tip } x')\\
&\texttt{traverse}\ f\ (\text{Bin } u\ v) = \textbf{do}\ v' \leftarrow \texttt{traverse}\ f\ v\\
&\qquad\qquad\qquad\qquad\qquad\quad\ u' \leftarrow \texttt{traverse}\ f\ u\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \texttt{return}\ (\text{Bin } u'\ v')
\end{aligned}
$$

or (equivalently):

$$
\begin{aligned}
&\texttt{traverse} :: \text{Applicative } m \Rightarrow (a \to m\ b) \to \text{Tree } a \to m\ (\text{Tree } b)\\
&\texttt{traverse}\ f\ (\text{Tip } x) \quad = \texttt{pure}\ \text{Tip} <\!\!*\!\!>\ f\ x\\
&\texttt{traverse}\ f\ (\text{Bin } u\ v) = \texttt{pure}\ (\texttt{flip}\ \text{Bin}) <\!\!*\!\!>\ \texttt{traverse}\ f\ v\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ <\!\!*\!\!>\ \texttt{traverse}\ f\ u
\end{aligned}
$$

Traversals — Examples

Let: **data** Tree a = Tip a | Bin (Tree a) (Tree a).

Breadth-first-traversal: left as an exercise

What about implementations like:

$$\texttt{traverse} :: \text{Applicative } m \Rightarrow (a \to m\ b) \to \text{Tree } a \to m\ (\text{Tree } b)$$
$$\texttt{traverse}\ f\ (\text{Tip } x) \quad = \texttt{pure}\ \text{Tip} \circledast f\ x$$
$$\texttt{traverse}\ f\ (\text{Bin } u\ v) = \texttt{pure}\ (\lambda u' \to \text{Bin } u'\ u') \circledast \texttt{traverse}\ f\ u$$

or:

$$\texttt{traverse} :: \text{Applicative } m \Rightarrow (a \to m\ b) \to \text{Tree } a \to m\ (\text{Tree } b)$$
$$\texttt{traverse}\ f\ (\text{Tip } x) \quad = \texttt{pure}\ \text{Tip} \circledast f\ x$$
$$\texttt{traverse}\ f\ (\text{Bin } u\ v) = \texttt{pure}\ \text{Bin} \circledast \texttt{traverse}\ f\ v \circledast \texttt{traverse}\ f\ u$$

Traversals — Examples

Let: **data** Tree a = Tip a | Bin (Tree a) (Tree a).

Breadth-first-traversal: left as an exercise

What about implementations like:

...

or:

```
traverse :: Applicative m => (a -> m b) -> Tree a -> m (Tree b)
traverse f (Tip x)   = pure Tip <*> f x
traverse f (Bin u v) = pure Bin <*> traverse f v
                                <*> traverse f u
```

or:

```
traverse :: Applicative m => (a -> m b) -> Tree a -> m (Tree b)
traverse f (Tip x)   = pure (\x' _ -> Tip x') <*> f x <*> f x
traverse f (Bin u v) = ...
```

Traversals — Examples and Need for Laws

Let: **data** Tree a = Tip a | Bin (Tree a) (Tree a).

Breadth-first-traversal: left as an exercise

What about implementations like:

...

???

That's what laws are for, right?

- Set of laws proposed in "The Essence of the Iterator Pattern".
- Further studied by Mauro Jaskelioff and Ondřej Rypáček in "An Investigation of the Laws of Traversals".
- No comprehensive characterization (but according conjectures).
- Useful for answering concrete questions?

A Concrete Question about Inverse Traversals

- One can generically, without knowing T, define an inverse version `treverse` for each `traverse`.

- The idea is to use `traverse` with a variant of $<\!\!*\!\!>$ defined via: $g <\!\!*\!\!>' y = \texttt{pure}\ (\lambda y'\ g' \rightarrow g'\ y') <\!\!*\!\!> y <\!\!*\!\!> g$.

- For the special case of monads, one can feed the value result of one effectful function into another effectful function, and get the combined effects (Kleisli composition):

  $(<\!\!=\!\!<) :: \text{Monad}\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ b) \rightarrow (a \rightarrow m\ c)$
  $(g <\!\!=\!\!< f)\ x = \textbf{do}\ \{x' \leftarrow f\ x; g\ x'\}$

- Now, does the following property hold?

$$
\begin{array}{ll}
 & g <\!\!=\!\!< f = \texttt{return} \\
\Rightarrow & \texttt{treverse}\ g <\!\!=\!\!< \texttt{traverse}\ f = \texttt{return}
\end{array}
$$

A Concrete Question about Inverse Traversals

From Jeremy's talk at the last meeting:

The Un of Programming 22

4.5. Linking forwards and backwards traversal

Inverse traversal law

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

does not seem to follow from other properties.

Nevertheless, I don't know of a *traverse* that respects idiom composition and idiom morphisms but not reversal.

Is it the consequence of some deeper structure?

By now we know. And more!

Backdrop: The Applicative Class (Idioms)

$$\begin{array}{l}
\textbf{class}\ \text{Functor}\ m \Rightarrow \text{Applicative}\ m\ \textbf{where} \\
\quad \texttt{pure}\ \ :: a \to m\ a \\
\quad (<\!\!*\!\!>) :: m\ (a \to b) \to m\ a \to m\ b
\end{array}$$

Laws (along with $\texttt{fmap id} = \texttt{id}$, $\texttt{fmap}\ (g \circ f) = \texttt{fmap}\ g \circ \texttt{fmap}\ f$):

$$\begin{array}{ll}
\texttt{fmap}\ f\ x & = \texttt{pure}\ f <\!\!*\!\!> x \\
\texttt{pure}\ (\circ) <\!\!*\!\!> u <\!\!*\!\!> v <\!\!*\!\!> w & = u <\!\!*\!\!> (v <\!\!*\!\!> w) \\
\texttt{pure}\ f <\!\!*\!\!> \texttt{pure}\ x & = \texttt{pure}\ (f\ x) \\
u <\!\!*\!\!> \texttt{pure}\ x & = \texttt{pure}\ (\$x) <\!\!*\!\!> u
\end{array}$$

An example:

$$\textbf{newtype}\ \text{ConstM}\ a\ \_ = \text{Const}\ [a]$$

$$\begin{array}{l}
\textbf{instance}\ \text{Applicative}\ (\text{ConstM}\ _)\ \textbf{where} \\
\quad \begin{array}{ll}
\texttt{pure}\ _ & = \text{Const}\ [\,] \\
\text{Const}\ xs <\!\!*\!\!> \text{Const}\ ys & = \text{Const}\ (xs +\!\!+\ ys)
\end{array}
\end{array}$$

The (Undebated) Laws about Traversals

- `traverse` $\mathsf{Id} = \mathsf{Id}$ (for the identity idiom)
- `traverse` $g \mathbin{<\!\circ\!>}$ `traverse` $f =$ `traverse` $(g \mathbin{<\!\circ\!>} f)$, where

  $$\begin{aligned}(<\!\circ\!>) &:: (\mathsf{Applicative}\ m, \mathsf{Applicative}\ n) \Rightarrow \\ &\quad (b \to n\ c) \to (a \to m\ b) \to a \to \mathsf{Compose}\ m\ n\ c \\ g \mathbin{<\!\circ\!>} f &= \mathsf{Compose} \circ \mathtt{fmap}\ g \circ f\end{aligned}$$

  for the composition of idioms:

  $$\textbf{data}\ \mathsf{Compose}\ m\ n\ a = \mathsf{Compose}\ (m\ (n\ a))$$

  (with canonical definition of the Applicative instance)
- $\phi \circ$ `traverse` $f =$ `traverse` $(\phi \circ f)$ if ϕ is an idiom morphism
- two naturality properties concerning the a and b in
  `traverse` $:: \mathsf{Applicative}\ m \Rightarrow (a \to m\ b) \to \mathsf{T}\ a \to m\ (\mathsf{T}\ b)$

Analysing Traversals

Plan of attack:

- Use $\phi \circ \texttt{traverse}\ f = \texttt{traverse}\ (\phi \circ f)$ law to relate results of traversals in different idioms.
- Choose specific idioms that reveal information about the traversal behaviour.
- For example, generically accessing the contents of a traversable object:

$$\texttt{contents} :: \mathsf{T}\ a \rightarrow [a]$$
$$\texttt{contents}\ t = \textbf{case}\ \texttt{traverse}\ (\lambda a \rightarrow \mathsf{Const}\ [a])\ t\ \textbf{of}$$
$$\mathsf{Const}\ as \rightarrow as$$

Problems with initial attempts (as I saw them):

- missing point of reference (connect contents to *what*?)
- calculationally not very pleasing

Analysing Traversals — The Free Idiom

Actually use the free/initial structure:

data Free $f\ c = \mathrm{P}\ c \mid \forall b.\ \mathrm{Free}\ f\ (b \rightarrow c)$:∗: $f\ b$

Specifically for analysing traversals, refine by specialising f to F a b, where:

data F :: $* \rightarrow * \rightarrow * \rightarrow *$ **where**
  F :: $a \rightarrow \mathrm{F}\ a\ b\ b$

Then Free (F a b) c is equivalent to Batch a b c, where:

data Batch $a\ b\ c = \mathrm{P}\ c \mid \mathrm{Batch}\ a\ b\ (b \rightarrow c)$:∗: a

Values of type Batch A B C take the form

$$\mathrm{P}\ f \mathbin{:\!*\!:} x_1 \mathbin{:\!*\!:} \ldots \mathbin{:\!*\!:} x_n$$

where $f :: \mathrm{B} \rightarrow \ldots \rightarrow \mathrm{B} \rightarrow \mathrm{C}$ with n arguments, and $x_i :: \mathrm{A}$.

Analysing Traversals — The Batch Idiom

Values of type Batch A B C take the form

$$\mathsf{P}\ f :\!\ast\!:\ x_1 :\!\ast\!: \ldots :\!\ast\!:\ x_n$$

where $f :: \mathsf{B} \to \ldots \to \mathsf{B} \to \mathsf{C}$ with n arguments, and $x_i :: \mathsf{A}$.

How is this an idiom?

instance Applicative (Batch a b) **where**
   . . .

such that

$$(\mathsf{P}\ g\ :\!\ast\!:_{i=1}^{m}\ x_i) <\!\ast\!> (\mathsf{P}\ f\ :\!\ast\!:_{i=m+1}^{n}\ x_i)$$
$$=$$
$$\mathsf{P}\ (\lambda y_1\ \ldots\ y_n \to g\ y_1\ \ldots\ y_m\ (f\ y_{m+1}\ \ldots\ y_n))\ :\!\ast\!:_{i=1}^{n}\ x_i$$

Analysing Traversals — The Batch Idiom

Given a concrete $t :: \mathrm{T}\ \mathrm{A}$, let's consider a specific use of `traverse` now:

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

where:

$\texttt{batch} :: a \to \mathrm{Batch}\ a\ b\ b$
$\texttt{batch}\ x = \mathrm{P}\ \texttt{id} :\!*\!: x$

Crucially, `traverse batch` t is still polymorphic in b, i.e., takes the form, for some n,

$$\mathrm{P}\ f :\!*\!: x_1 :\!*\!: \ldots :\!*\!: x_n$$

where $f :: b \to \ldots \to b \to \mathrm{T}\ b$ of arity n is polymorphic, and $x_i :: \mathrm{A}$.

This is extremely useful!

Analysing Traversals — The Batch Idiom

Crucially, `traverse batch` t is still polymorphic in b, i.e., takes the form, for some n,

$$\mathsf{P}\ f \mathbin{:\!*\!:} x_1 \mathbin{:\!*\!:} \ldots \mathbin{:\!*\!:} x_n$$

where $f :: b \to \ldots \to b \to \mathsf{T}\ b$ of arity n is polymorphic, and $x_i :: \mathsf{A}$.

This is extremely useful!

Some things we can show (using the laws about `traverse`):

1. $t = f\ x_1\ \ldots\ x_n$
2. `contents` $(f\ y_1\ \ldots\ y_n) = [y_1, \ldots, y_n]$
3. `traverse` $g\ (f\ y_1\ \ldots\ y_n) =$ `pure` $f \mathbin{<\!*\!>} g\ y_1 \mathbin{<\!*\!>} \ldots \mathbin{<\!*\!>} g\ y_n$

This is enough to prove the inversion law.

Proving the Inversion Law

Assume $g <\!\!=\!\!< h = \texttt{return}$, and $t = f\ x_1\ \ldots\ x_n$ as given. Then:

$$
\begin{aligned}
& (\texttt{treverse}\ g <\!\!=\!\!< \texttt{traverse}\ h)\ t \\
= {} & \textbf{do}\ \{ t' \leftarrow \texttt{traverse}\ h\ t; \texttt{treverse}\ g\ t' \} \\
= {} & \textbf{do}\ \{ t' \leftarrow \texttt{pure}\ f <\!\!*\!\!> h\ x_1 <\!\!*\!\!> \ldots <\!\!*\!\!> h\ x_n; \texttt{treverse}\ g\ t' \} \\
= {} & \textbf{do}\ \{ y_1 \leftarrow h\ x_1;\ \ldots; y_n \leftarrow h\ x_n; \texttt{treverse}\ g\ (f\ y_1\ \ldots\ y_n) \} \\
= {} & \textbf{do}\ \{ y_1 \leftarrow h\ x_1;\ \ldots; y_n \leftarrow h\ x_n; \\
& \qquad \texttt{pure}\ (\lambda z_n\ \ldots\ z_1 \rightarrow f\ z_1\ \ldots\ z_n) <\!\!*\!\!> g\ y_n <\!\!*\!\!> \ldots <\!\!*\!\!> g\ y_1 \} \\
= {} & \textbf{do}\ \{ y_1 \leftarrow h\ x_1;\ \ldots; y_n \leftarrow h\ x_n; \\
& \qquad z_n \leftarrow g\ y_n;\ \ldots; z_1 \leftarrow g\ y_1; \\
& \qquad \texttt{return}\ (f\ z_1\ \ldots\ z_n) \} \\
= {} & \textbf{do}\ \{ y_1 \leftarrow h\ x_1;\ \ldots; y_{n-1} \leftarrow h\ x_{n-1}; \\
& \qquad z_n \leftarrow \texttt{return}\ x_n; \\
& \qquad z_{n-1} \leftarrow g\ y_{n-1};\ \ldots; z_1 \leftarrow g\ y_1; \\
& \qquad \texttt{return}\ (f\ z_1\ \ldots\ z_n) \} \\
= {} & \ldots \\
= {} & \textbf{do}\ \{ \texttt{return}\ (f\ x_1\ \ldots\ x_n) \} = \texttt{return}\ t
\end{aligned}
$$

Doing without the Batch Idiom

Crucially, `traverse batch` t is still polymorphic in b, i.e., takes the form, for some n,

$$\mathsf{P}\ f :\!*\!:\ x_1 :\!*\!:\ \ldots :\!*\!:\ x_n$$

where $f :: b \to \ldots \to b \to \mathsf{T}\ b$ of arity n is polymorphic, and $x_i :: \mathsf{A}$.

This is extremely useful!

Some things we can show (using the laws about `traverse`):

1. $t = f\ x_1\ \ldots\ x_n$
2. `contents` $(f\ y_1\ \ldots\ y_n) = [y_1, \ldots, y_n]$
3. `traverse` $g\ (f\ y_1\ \ldots\ y_n) =$ `pure` $f \circledast g\ y_1 \circledast \ldots \circledast g\ y_n$

This is enough to prove the inversion law.

Moreover: 1. and 2. are enough to determine n, f, and the x_i.

The Representation Theorem

Theorem: Let t :: T A and a definition of `traverse` be given. There is a unique n, a unique polymorphic function $f :: b \to \ldots \to b \to \mathrm{T}\ b$ of arity n, and unique values $x_1, \ldots, x_n$, all of type A, such that $t = f\ x_1\ \ldots\ x_n$ and, for arbitrary y_i of arbitrary type, `contents` $(f\ y_1\ \ldots\ y_n) = [y_1, \ldots, y_n]$. Furthermore, `traverse` $g\ (f\ y_1 \ldots y_n) =$ `pure` $f \mathbin{<\!\!*\!\!>} g\ y_1 \mathbin{<\!\!*\!\!>} \ldots. \mathbin{<\!\!*\!\!>} g\ y_n$ for all g and y_i (of/for arbitrary types and idiom).

Beside the inversion law this also gives:

- Lawful instances of Traversable exactly correspond to finitary containers. (In particular, types containing infinite structures are not lawfully traversable.)
- Different lawful instances of Traversable for the same T only differ by fixed (per “shape”) permutation of positions.
- A coherence/naturality property holds for lawful instances of Traversable on T, T'.

References

J. Gibbons and B. Oliveira.
The Essence of the Iterator Pattern.
J. Funct. Program., 19(3–4):377–402, 2009.

M. Jaskelioff and O. Rypáček.
An Investigation of the Laws of Traversals.
In *MSFP, Proceedings*, volume 76 of *EPTCS*, pages 40–49, 2012.