An Investigation of the Laws of Traversals

Mauro Jaskelioff
CIFASIS, Rosario, Argentina
FCEIA, Universidad Nacional de Rosario, Argentina

Ondrej Rypacek
King's College, London, UK

Traversals of data structures are ubiquitous in programming. Consequently, it is important to be able to characterise those structures that are traversable and understand their algebraic properties. Traversable functors have been characterised by McBride and Paterson as those equipped with a distributive law over arbitrary applicative functors; however, laws that fully capture the intuition behind traversals are missing. This article is an attempt to remedy this situation by proposing laws for characterising traversals that capture the intuition behind them. To support our claims, we prove that finitary containers are traversable in our sense and argue that elements in a traversable structure are visited exactly once.

1 Introduction

Traversals of data structures are ubiquitous in programming. Consequently, it is essential for the writer of mathematically structured programs to have a precise understanding of the abstract structure of traversals and of their algebraic properties.

There are many notions of traversal, but in this article, we will focus on the traversals of functors $G : \mathsf{Set} \to \mathsf{Set}$ as given by a family of natural transformations $\delta^F : GF \to FG$ over applicative functors F [10]. This notion of traversal is quite abstract but practical for structuring and reasoning about programs [8, 5, 7], and encompasses other notions of traversals, such as generating a list of elements.

Usually, distributive laws are required to respect structure via coherence laws [12]. However, the characterisation of traversability in terms of a distributive law given by McBride and Paterson is incomplete as no coherence laws are required to hold and one can write distributive laws that do not follow the intuition of what a traversal should be. Consequently, a distributive law is not enough and the definition of a traversal needs to be strengthened to avoid bad instances. However, there seems to be no consensus of precisely what a traversal should be.

Moggi et al. [11] define traversals of a functor G over a functor F to be a family over $G1$ of distributive laws $\delta^F_{s:G1} : G_sF \to FG_s$. Here, G_s is a family over $G1$ of functors obtained from the pullback of the diagram $1 \xrightarrow{s} G1 \xleftarrow{G!} GX$. The family of morphisms provides an easy way to express shape preservation (the shape of G before traversing the structure is the same as the shape of G after traversing it). Nevertheless, we find this notion of traversability to be a bit unsatisfactory since it allows traversals that go over the same element more than once. Gibbons and Oliveira [8] proposed many properties that should hold for traversals, but they did not seek to obtain a lawful definition of traversability. Furthermore, they failed to recognize the law that would prevent traversing over an element twice.

The main contribution of this article is to establish coherence laws for the distributive law that capture the intuition of traversals. The laws can be expressed by simple equations; they are shown to hold for the largest class known of traversable functors, namely finitary containers; and they are shown to prohibit the "bad" traversals identified by Gibbons and Oliveira. Additionally, we complete the Haskell traversable class so that it contains the three ways in which distributive laws can be expressed [4, 3]. Furthermore,

Submitted to:
MSFP 2012

we characterize traversability categorically in two ways: as a 2-functor between particular 2-categories, and as a distributive law over a monoidal action.

The article is organised as follows. Section 2 is written with the Haskell programmer in mind; we review applicative and traversable functors, and we motivate and present the proposed laws. We work in the category of sets and total functions from section 3 onwards, where we show that finitary containers are traversable in our sense. In section 4 we analyze some consequences of the laws and we argue that the laws imply that every position in the structure is visited exactly once. Additionally, a categorical interpretation of traversability is given. Finally, in section 5 we conclude and discuss future work.

2 Traversals in Haskell

In this section we give our motivation for and introduce our proposed laws for traversals using the functional language Haskell. We give an intuition of why the laws are reasonable but we defer a more rigorous explanation to the following sections.

Intuitively, a traversal of a data structure is a function that collects all elements in a data structure in a given order. A more abstract formulation was proposed by Moggi et al. [11] where a traversal is a distributive law of a functor (representing the data structure in question) over an arbitrary monad. McBride and Paterson extended the notion of traversability to be a distributive law of a functor over an arbitrary applicative functor [10]. As we will see next, the notion of a distributive law is general enough to include the basic notion of a function listing the elements of a data structure.

2.1 Applicative Functors

An applicative functor [10] is an instance of the class

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

such that the following coherence conditions hold.

identity	$\mathit{pure}\ \mathit{id} \circledast u$	$= u$
composition	$\mathit{pure}\ (\cdot) \circledast u \circledast v \circledast w$	$= u \circledast (v \circledast w)$
homomorphism	$\mathit{pure}\ g \circledast \mathit{pure}\ x$	$= \mathit{pure}\ (g\ x)$
interchange	$u \circledast \mathit{pure}\ x$	$= \mathit{pure}\ (\lambda g \to g\ x) \circledast u$

Every monad is an applicative functor, but applicative functors are more general. For example, every monoid determines an applicative functor (which is not a monad):

$$
\begin{aligned}
&\mathbf{newtype}\ K\ a\ b = K\ \{\mathit{unK} :: a\}\\
&\mathbf{instance}\ \mathit{Monoid}\ a \Rightarrow \mathit{Applicative}\ (K\ a)\ \mathbf{where}\\
&\quad \mathit{pure}\ x = K\ \emptyset\\
&\quad f \circledast x \ \ = K\ (\mathit{unK}\ f \oplus \mathit{unK}\ x)
\end{aligned}
$$

where $\emptyset$ is the monoid unit, and $\oplus$ is the monoid multiplication.

Applicative functors are closed under identity and composition[1].

[1] For brevity, throughout the article we omit the required *Functor* instances. The complete source code can be found at `http://www.fceia.unr.edu.ar/~mauro/`.

```
newtype Id a = Id {unId :: a}
instance Applicative Id where
  pure  = Id
  f ⊛ x = Id (unId f (unId x))
newtype C f g a = Comp {unC :: f (g a)}
instance (Applicative f, Applicative g) ⇒ Applicative (C f g) where
  pure  = Comp · pure · pure
  f ⊛ x = Comp (pure (⊛) ⊛ unC f ⊛ unC x)
```

2.2 The class of Traversable functors

McBride and Paterson propose a traversal to be a distributive law of a functor over all applicative functors [10]. Hence, the class of traversable functors is defined:

```
class Functor t ⇒ Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
  dist :: Applicative f ⇒ t (f a) → f (t a)
  traverse f = dist · fmap f
  dist       = traverse id
```

A minimal instance should provide a definition of either *traverse* or *dist*, as one can be defined by the other, as shown by the default instances above.

Example 2.1. The canonical example of a traversable functor is the list functor.

```
instance Traversable [] where
  dist []       = pure []
  dist (x : xs) = pure (:) ⊛ x ⊛ dist xs
```

Another typical example is that of binary trees with information in the nodes:

```
data Bin a = Leaf | Node (Bin a) a (Bin a)
instance Traversable Bin where
  dist Leaf         = pure Leaf
  dist (Node l a r) = pure Node ⊛ dist l ⊛ a ⊛ dist r
```

Our last example is the identity functor.

```
instance Traversable Id where
  dist (Id x) = fmap Id x
```

Remark 2.2. Given a *Traversable* functor, it is possible to construct a list of its elements by traversing it over an accumulator, as done by the function *toList*:

```
toList :: Traversable t ⇒ t a → [a]
toList = unK · dist · fmap wrap
  where wrap   :: x → K [x] a
        wrap x = K [x]
```

2.3 The need for laws

Notice that no coherence conditions are required to hold for instances of *traverse* (or *dist*) in the class definition above. Hence, it is possible to instantiate definitions of *dist* that do not follow our intuition of what a traversal should be. Consider the following functions implementing a distributive law between lists and an arbitrary applicative functor.

$$
\begin{array}{ll}
\mathit{distL}, \mathit{distL}', \mathit{distL}'' & :: \mathit{Applicative}\, f \Rightarrow [f\ a] \to f\ [a] \\
\mathit{distL}\ \_ & = \mathit{pure}\ [\,] \\
\mathit{distL}'\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{distL}'\ [x] & = \mathit{pure}\ [\,] \\
\mathit{distL}'\ (x : y : xs) & = \mathit{pure}\ (:) \circledast x \circledast \mathit{distL}'\ (y : xs) \\
\mathit{distL}''\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{distL}''\ (x : xs) & = \mathit{pure}\ (:) \circledast x' \circledast \mathit{distL}''\ xs \\
 & \quad \mathbf{where}\ x' = \mathit{pure}\ (\lambda x\ y \to x) \circledast x \circledast x
\end{array}
$$

The functions have the correct type but they are not traversals: the first one (*distL*) does not even try to traverse the list, the second one (distL') does not visit the last element of the list, and the last one (distL'') collects each applicative effect twice (but not the data). Consequently, in order for the *Traversable* class to capture the intuition behind traversals, it needs to require certain laws to hold in order to prohibit definitions such as those of the three functions above.

2.4 Reformulation of the *Traversable* class

We conclude this section with our proposed class of Traversable functors. It differs from the previous one in two aspects.

- In addition to *traverse* and *dist*, a minimal instance can also be given by defining a function $\mathit{consume} :: \mathit{Applicative}\, f \Rightarrow (t\ a \to b) \to t\ (f\ a) \to f\ b$.
- It requires two laws to hold.

We define the class of traversable functors to be:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \mathbf{where} \\
\quad \begin{array}{ll}
\mathit{traverse} & :: \mathit{Applicative}\, f \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b) \\
\mathit{dist} & :: \mathit{Applicative}\, f \Rightarrow t\ (f\ a) \to f\ (t\ a) \\
\mathit{consume} & :: \mathit{Applicative}\, f \Rightarrow (t\ a \to b) \to t\ (f\ a) \to f\ b
\end{array} \\
\quad \begin{array}{ll}
\mathit{traverse}\, f & = \mathit{dist} \cdot \mathit{fmap}\, f \\
\mathit{dist} & = \mathit{consume}\ \mathit{id} \\
\mathit{consume}\, f & = \mathit{fmap}\, f \cdot \mathit{traverse}\ \mathit{id}
\end{array}
\end{array}
$$

subject to the following laws:

$$
\begin{array}{lll}
\textbf{Unitarity} & \mathit{dist} \cdot \mathit{fmap}\ \mathit{Id} & = \mathit{Id} \\
\textbf{Linearity} & \mathit{dist} \cdot \mathit{fmap}\ \mathit{Comp} & = \mathit{Comp} \cdot \mathit{fmap}\ \mathit{dist} \cdot \mathit{dist}
\end{array}
$$

The same laws expressed in terms of *traverse* are:

$$\begin{aligned} &\mathit{traverse}\ (\mathit{Id} \cdot f) &&= \mathit{Id} \cdot \mathit{fmap}\, f \\ &\mathit{traverse}\ (\mathit{Comp} \cdot \mathit{fmap}\ g \cdot f) &&= \mathit{Comp} \cdot \mathit{fmap}\ (\mathit{traverse}\ g) \cdot \mathit{traverse}\, f \end{aligned}$$

We leave as an exercise to the reader to express the laws in terms of *consume*.

Note that, since the identity functor is applicative, the laws for traverse imply functoriality even without the type class requirement. Hence an alternative definition would not require traversable functors to be an instance of the *Functor* class, but just to provide an instance of *traverse* subject to the two laws above. In this case, providing definitions just for *dist* or *consume* would not be enough.

In the following sections we show that these laws are reasonable for a large class of functors, and analyse how they prohibit wrong definitions such as *distL*, *distL′* and *distL″*.

3 Canonical Traverse for Finitary Containers

Categorically, *applicative functors* are functors $F : \mathsf{Set} \to \mathsf{Set}$, together with natural transformations:

$$\begin{aligned} \eta_X &: X \to FX && \text{(unit)} \\ \circledast_{X,Y} &: F(Y^X) \times FX \to FY && \text{(application under } F\text{)} \ , \end{aligned}$$

where Y^X is the function space (cartesian closure), together with equations expressing basically that η_{Y^X} injects pure functions Y^X to functions under in $F(Y^X)$ and $\circledast$ respects this (see [10] for the details).

In category theory it is more convenient to work with the following equivalent definitions:

Definition 3.1 (Applicative Functor). *An* applicative functor *is equivalently either of:*

(i) A functor $F : \mathsf{Set} \to \mathsf{Set}$ which is lax monoidal with respect to the cartesian product and whose strength is coherent with the monoidal structure, which is to say that the following diagram commutes:

$$\begin{tikzcd} (FX \times FY) \times Z \arrow[r, "\alpha"] \arrow[d, "\mu \times Z"'] & FX \times (FY \times Z) \arrow[r, "FX \times \sigma"] & FX \times F(Y \times Z) \arrow[d, "\mu"] \\ F(X \times Y) \times Z \arrow[r, "\sigma"] & F((X \times Y) \times Z) \arrow[r, "F\alpha"] & F(X \times (Y \times Z)) \end{tikzcd}$$

where σ is strength, μ is the monoidal action and α is associativity. Note that all Set *functors are strong so the key requirement here is the coherence with the monoidal action.*

(ii) A pointed lax monoidal functor $F : \mathsf{Set} \to \mathsf{Set}$, where the unit of F, $\eta_X : X \to FX$, coincides with the unit of the monoidal structure $\nu : 1 \to F1$ in that $\eta_1 = \nu$; and the multiplication $\mu_{X,Y} : FX \times FY \to F(X \times Y)$ is coherent with η in the sense that

$$\begin{tikzcd} & X \times Y \arrow[dl, "\eta_X \times \eta_Y"'] \arrow[dr, "\eta_{X \times Y}"] & \\ FX \times FY \arrow[rr, "\mu_{X,Y}"'] & & F(X \times Y) \end{tikzcd}$$

commutes.

The equivalence of (i) and (ii) is straightforward.

Definition 3.2 (Applicative Morphism). *Let F and G be applicative functors. An* applicative morphism *is a natural transformation $\tau : F \to G$ that respects the unit and multiplication. That is, a natural transformation τ such that the following diagrams commute.*

$$\begin{tikzcd}
& X \arrow[dl, "\eta^F_X"'] \arrow[dr, "\eta^G_X"] & \\
FX \arrow[rr, "\tau_X"'] & & GX
\end{tikzcd}
\qquad
\begin{tikzcd}
FX \times FY \arrow[r, "\mu^F_{X,Y}"] \arrow[d, "\tau_X \times \tau_Y"'] & F(X \times Y) \arrow[d, "\tau_{X \times Y}"] \\
GX \times GY \arrow[r, "\mu^G_{X,Y}"'] & G(X \times Y)
\end{tikzcd}$$

Applicative functors and applicative morphisms form a category $\mathscr{A}$.

The identity functor $\mathbf{1}$ is an applicative functor, and composition of applicative functors is applicative. Hence, applicative functors form a (large) monoid, with functor composition $\circ$ as multiplication and identity functor $\mathbf{1}$ as unit.

In [10], *traversable functors* are characterised as those which *distribute over* all applicative functors. There, distributivity of a traversable T over applicative F's meant only the existence of natural transformations of type $TF \Rightarrow FT$. However, without further constraints this characterisation is too coarse. Here we refine the notion of a traversable functor as follows:

Definition 3.3 (Traversable Functor). *A functor $T : \mathsf{Set} \to \mathsf{Set}$ is said to be* traversable *if there is a family of natural transformations*

$$\delta^F_X : TFX \to FTX$$

natural in F and respecting the monoidal structure of applicative functor composition. Explicitly, for all applicative F, $G : \mathsf{Set} \to \mathsf{Set}$ and applicative morphisms $\alpha : F \to G$, the following diagrams of natural transformations commute:

$$\begin{tikzcd}
TF \arrow[r, "\delta^F"] \arrow[d, "T\alpha"'] & FT \arrow[d, "\alpha_T"] \\
TG \arrow[r, "\delta^G"'] & GT
\end{tikzcd}
\qquad
\begin{tikzcd}
& FTG \arrow[dr, "F\delta^G"] & \\
TFG \arrow[ur, "\delta^F G"] \arrow[rr, "\delta^{FG}"'] & & FGT
\end{tikzcd}
\qquad
\begin{tikzcd}
T\mathbf{1} \arrow[r, bend left, "\mathsf{id}_T"] \arrow[r, bend right, "\delta^{\mathbf{1}}"'] & \mathbf{1}T
\end{tikzcd}$$

naturality *linearity* *unitarity*

We sometimes call the family δ a traversal of *T.*

Next we introduce a class of functors, which are always traversable: so-called *finitary containers* [1].

Definition 3.4 (Finitary Container). *A* finitary container *is given by*

(i) a set S of shapes

(ii) an arity $\mathsf{ar} : S \to \mathbb{N}$

To each container (S, ar) one can assign a functor $\mathsf{Ext}(S, \mathsf{ar}) : \mathsf{Set} \to \mathsf{Set}$ called the extension of (S, ar) *defined for each set X as the set of (dependent) pairs (s, f) where $s \in S$ and $f \in X^{\mathsf{ar}\, s}$, where X^n, for $n \in \mathbb{N}$, is the n-fold product $\underbrace{X \times \cdots \times X}_{n \text{ times}}$*

Finitary containers are also known as finitary *dependent polynomial functors* [6] or functors *shapely over lists* [11]. Moggi et al. [11] define a canonical traversal by monads for shapely functors. It is also indicated that this traversal could be generalised from monads to all monoidal functors. Here we show that all finitary containers (shapely functors) are traversable in our sense. Explicitly: for each extension of a finitary container we construct a natural family of distributive laws δ satisfying *linearity* and *unitarity*.

As extensions of containers are sums of products we proceed in stages.

Lemma 3.5. *Traversable functors are closed under arbitrary sums.*

Proof. Let T_s, $s \in S$ be a family of traversable functors with traversals δ_s. We must show that $\sum_{s\in S} T_s$ is traversable. To this end we construct, for each X:

$$\begin{tikzcd}\sum_{s\in S} T_s F X \arrow[r, "\sum_{s\in S}\delta_s"] & \sum_{s\in S} F T_s X \arrow[r, "{[F(\mathsf{inj}_s\, T_s X)]_{s\in S}}"] & F\sum_{s\in S} T_s X\end{tikzcd} \quad ,$$

where $\mathsf{inj}_s y = (s,y)$ and where $[\,]_{s\in S}$ is case splitting on S. Naturality, unitarity and linearity are all easily checked. ☐

Lemma 3.6. *Traversable functors are closed under finite products.*

Proof. Let δ be a traversal for T, S finite. Then T^S is just iterated product $T \times \cdots \times T$, $|S|$-times, and therefore we can use multiplication μ of any applicative functor F to construct a $\delta^S : T^S F \Rightarrow F T^S$ by finite iteration. Namely, define $\mu^k : F^k X \to F(X^k)$, $k \in \mathbb{N}$ by $\mu^0 = \nu$, $\mu^1 = \mathbf{1}_F$, $\mu^{k+1} = \mu \cdot (\mathbf{1}_F \times \mu^k)$ and put $\delta^S = \mu^{|S|} \cdot \delta^{|S|}$. Now naturality follows by naturality of everything in sight; unitarity from the fact that μ for the identity monoidal functor is just the identity. To see linearity just observe commutativity of the following for each $k \in \mathbb{N}$:

$$\begin{tikzcd} & F(GX)^k \arrow[dr, "F(\mu^G)^k_X"] & \\ (FGX)^k \arrow[ur, "(\mu^F)^k_{GX}"] \arrow[rr, "(F\mu^G \cdot \mu^F)^k_X"'] & & FG(X^k) \end{tikzcd} \quad .$$

☐

Theorem 3.7. *All extensions of finitary containers are traversable.*

Proof. Extensions of finitary containers are sums of finite products so the results follows directly by the previous two Lemmas 3.6 and 3.5. ☐

It remains an open question whether the traversals defined in the above theorem are essentially unique. In other words, whether there is an isomorphism between permutations of arities on finitary containers and their traversals.

4 Analysis of the Laws

4.1 Unitarity

Unitarity implies the so-called "purity law" $\delta^F \cdot T(\eta) = \eta_T$ [8]. In fact, the two laws are equivalent.

To see this, we notice that the identity functor is initial in the category $\mathscr{A}$ of applicative functors and applicative morphisms, with the universal map given by η. Hence we obtain the purity law by the following naturality square.

$$\begin{tikzcd}
T\mathbf{1} \arrow[r, "\delta^{\mathbf{1}}=\mathsf{id}"] \arrow[d, "T(\eta)"'] & \mathbf{1}T \arrow[d, "\eta_T"] \\
TF \arrow[r, "\delta^F"'] & FT
\end{tikzcd}$$

Conversely, instantiating the purity law for the identity functor, we obtain unitarity.

Unitarity (and hence, the purity law) is stronger than shape preservation. For example, returning the mirror of a binary tree is forbidden. By requiring unitarity to hold we are implicitly stating that what matters in a traversal is the order in which the effects are collected.

Note that unitarity implies that each element is traversed at least once. Hence distributive laws such as $distL$ and $distL'$ of Section 2.3 do not respect unitarity.

4.2 Linearity

One of the sought-after effects of the laws is to rule out traversals that go more than once over each element. We will show that this kind of definition does not respect the linearity law by analysing the simple example of traversing the identity functor.

Consider the following distributive law of the identity functor over an arbitrary applicative functor,

$$\delta^F_X : \mathbf{1}(FX) = FX \xrightarrow{d_{FX}} FX \times FX \xrightarrow{\mu^F_X} F(X \times X) \xrightarrow{F\pi_1} FX = F(\mathbf{1}X)$$

where $d_X(x : X) = (x,x) : X \times X$ is the diagonal function. The distributive law δ traverses over the data twice so it is not a proper traversal. Although unitarity holds for δ, linearity does not, as the following counter-example shows.

Consider the applicative functor L arising from the list monad, and $[[],[1]] \in L(L(\mathbb{N}))$. Calculating, we obtain that linearity does not hold:

$$\delta^{L\circ L}_{\mathbb{N}}[[],[1]] \quad = \quad [[],[],[],[1]] \quad \neq \quad [[],[1],[],[1]] \quad = \quad \delta^{L}_{\mathbb{N}}(L\delta^{L}_{\mathbb{N}}[[],[1]])$$

A similar counter-example can be constructed for the traversal of lists $distL''$ of Section 2.3.

$$distL''^{L\circ L}[[[],[[1]]]] \;=\; [[],[],[],[[[1]]]] \quad \neq \quad [[],[],[[[1]]],[[[1]]]] \;=\; distL''^{L} \circ L(distL''^{L})[[[],[[1]]]]$$

These counter-examples suggest that distributive laws that traverse over elements more than once will violate the linearity law. This consequence of the linearity law was apparently overlooked in [8].

4.3 Preservation of Kleisli Composition

The following lemma was proved as a property of the canonical distributivity of finitary containers by Moggi et al. [11] and under label *sequential composition of monadic traversals* in [8]. Here, however, we can prove it as a consequence of the definition of traversability.

Lemma 4.1. *Let (T,δ) be a traversable functor. Then for any commutative monad M, we have that δ preserves Kleisli composition, i.e.*

$$\begin{tikzcd}
TMM \arrow[r, "\delta^M M"] \arrow[d, "T\mathsf{flat}"'] & MTM \arrow[r, "M\delta^M"] & MMT \arrow[d, "\mathsf{flat}T"] \\
TM \arrow[rr, "\delta^M"'] & & MT
\end{tikzcd}$$

where flat *is the multiplication of the monad* M*.*

Proof. The commutative monad M induces a commutative applicative functor. The multiplication of the monad $\mathsf{flat} : M \circ M \to M$ is an applicative morphism (thanks to commutativity). Hence, the following diagram commutes because of preservation of composition and naturality of δ.

$$\begin{tikzcd}
TMM \arrow[r, "\delta^M M"] \arrow[d, "T\mathsf{flat}"'] \arrow[rr, bend right, "\delta^{M\circ M}"'] & MTM \arrow[r, "M\delta^M"] & MMT \arrow[d, "\mathsf{flat}T"] \\
TM \arrow[rr, "\delta^M"'] & & MT
\end{tikzcd}$$

□

4.4 Categorical meaning of the laws

Let App be the 2-category with one object, arrows: applicative functors and 2-cells: applicative morphisms. And let $J : \mathsf{App} \to \mathsf{Cat}$ be the inclusion into the 2-category of categories sending the object of App to Set. Then a traversable functor is exactly a co-lax natural transformation $J \to J : \mathsf{App} \to \mathsf{Cat}$. See [9] for the elementary 2-categorical notions.

It's interesting to compare this to the following characterisation in terms of distributive laws of monoidal actions [13]. Consider the strict monoidal category $(\mathscr{A}, \circ, \mathbf{1})$ of applicative functors and applicative morphisms with the monoidal action applicative functor composition. This category induces a monoidal action over Set, where the action $\Diamond : \mathscr{A} \times \mathsf{Set} \to \mathsf{Set}$ is simply application of the functor, i.e. $F\Diamond X \mapsto FX$ and $F\Diamond f \mapsto Ff$.

A distributive law of a functor T over the monoidal action $\Diamond$ of $\mathscr{A}$ is a binatural transformation $\delta : T(\_\Diamond\_) \to \_\Diamond(T\_)$, satisfying the axioms:

$$\begin{tikzcd}
T((F\circ G)\Diamond X) \arrow[rr, "\delta_X^{F\circ G}"] \arrow[d, "\mathsf{id}"'] & & (F\circ G)\Diamond TX \arrow[d, "\mathsf{id}"] \\
T(F\Diamond(G\Diamond X)) \arrow[r, "\delta^F_{G\Diamond X}"'] & F\Diamond T(G\Diamond X) \arrow[r, "F\Diamond\delta^G_X"'] & F\Diamond(G\Diamond TX)
\end{tikzcd}
\qquad
\begin{tikzcd}
 & TX \arrow[dl, "T\mathsf{id}"'] \arrow[dr, "\mathsf{id}"] & \\
T(\mathbf{1}\Diamond X) \arrow[rr, "\delta^{\mathbf{1}}_X"'] & & \mathbf{1}\Diamond TX
\end{tikzcd}$$

Hence, a functor is traversable when it comes equipped with a distributive law over the monoidal action of applicative functors.

5 Conclusion

The definition of traversability as a distributive law needs to be strenghtened in order to capture the intuitive notion of a traversal. We have provided two simple laws and shown that these laws hold for finitary containers, and we have provided evidence that they restrict traversals to those that go over each position exactly once.

Finitary containers are the largest known class of traversable functors, and we are not aware of any functor that is traversable and it is not a finitary container. It has been conjectured that every traversable functor is a finitary container [11], but proof of this has eluded us. One possiblity for proving it might be to take a syntactic approach and restrict the proof to functors in a given universe.

The characterisation of traversals as a distributive law over applicative functors is interesting because it leads to other kinds of traversability by changing distributivity over applicative functors by distributivity over others kinds of functor. For example, one might consider a distributive law over all commutative applicative functors i.e. applicative functors F for which

$$F\mathsf{swap}_{X,Y} \circ \mu = \mu \circ \mathsf{swap}_{FX,FY} : FX \times FY \to F(Y \times X) \ ,$$

where $\mathsf{swap}_{X,Y} : X \times Y \to Y \times X$ is the obvious morphism. Then, one would obtain a class of traversable functors that includes finitary quotient containers [2] such as unordered pairs, where commutativity of the applicative functor is essential.

Acknowledgement We thank the anonymous reviewers for their constructive criticism.

References

[1] Michael Abbott, Thorsten Altenkirch & Neil Ghani (2003): *Categories of Containers*. In: *Proceedings of Foundations of Software Science and Computation Structures*, pp. 23–38, doi:10.1007/3-540-36576-1_2.

[2] Michael Abbott, Thorsten Altenkirch, Neil Ghani & Conor McBride (2004): *Constructing Polymorphic Programs with Quotient Types*. In: *7th International Conference on Mathematics of Program Construction (MPC 2004)*, pp. 2–15, doi:10.1007/978-3-540-27764-4_2.

[3] Michael Barr (2005): *Beck Distributivity*. Available at `ftp://ftp.math.mcgill.ca/barr/pdffiles/distlaw.pdf`.

[4] Jon Beck (1969): *Distributive laws*. In: *Seminar on Triples and Categorical Homology Theory*, *Lecture Notes in Mathematics* 80, Springer Berlin / Heidelberg, pp. 119–140, doi:10.1007/BFb0083084.

[5] Germán A. Delbianco, Mauro Jaskelioff & Alberto Pardo (2011): *Applicative Shortcut Fusion*. In: *Proceedings of the 12th International Symposium on Trends in Functional Programming*, Madrid, Spain.

[6] Nicola Gambino & Martin Hyland (2004): *Wellfounded Trees and Dependent Polynomial Functors*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *Types for Proofs and Programs*, *Lecture Notes in Computer Science* 3085, Springer Berlin / Heidelberg, pp. 210–225, doi:10.1007/978-3-540-24849-1_14.

[7] Jeremy Gibbons & Richard Bird (2011): *Effective Reasoning about Effectful Traversals*. Available at `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/backwards.pdf`. Submitted for publication.

[8] Jeremy Gibbons & Bruno c. d. s. Oliveira (2009): *The essence of the iterator pattern*. *Journal of Functional Programming* 19, pp. 377–402, doi:10.1017/S0956796809007291.

[9] Gregory Kelly & Ross Street (1974): *Review of the elements of 2-categories*, pp. 75–103. 420, Springer Berlin / Heidelberg, doi:10.1007/BFb0063101.

[10] Conor McBride & Ross Paterson (2008): *Applicative programming with effects*. *Journal of Functional Programming* 18(01), pp. 1–13, doi:10.1017/S0956796807006326.

[11] Eugenio Moggi, Giana Bellè & C. Barry Jay (1999): *Monads, Shapely Functors and Traversals*. *Electronic Notes in Theoretical Computer Science* 29, pp. 187 – 208, doi:10.1016/S1571-0661(05)80316-0. CTCS '99, Conference on Category Theory and Computer Science.

[12] Ondřej Rypáček (2010): *Distributive Laws in Programming Structures*. Ph.D. thesis, University of Nottingham. Available at `http://etheses.nottingham.ac.uk/1077/`.

[13] Zoran Skoda (2004): *Distributive laws for actions of monoidal categories*. arXiv:math/0406310v2. Available at `http://arxiv.org/abs/math/0406310v2`.