UPPSALA
UNIVERSITET

IT 19 035
Examensarbete 15 hp
September 2019

Cofree Traversable Functors

Love Waern

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Cofree Traversable Functors

Love Waern

Traversable functors see widespread use in purely functional programming as an approach to the iterator pattern. Unlike other commonly used functor families, free constructions of traversable functors have not yet been described. Free constructions have previously found powerful applications in purely functional programming, as they embody the concept of providing the minimal amount of structure needed to create members of a complex family out of members of a simpler, underlying family. This thesis introduces Cofree Traversable Functors, together with a provably valid implementation, thereby developing a family of free constructions for traversable functors. As free constructions, cofree traversable functors may be used in order to create novel traversable functors from regular functors. Cofree traversable functors may also be leveraged in order to manipulate traversable functors generically.

Handledare: Justin Pearson
Ämnesgranskare: Tjark Weber
Examinator: Johannes Borgström
IT 19 035
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Within purely functional programming, *Traversable Functors* represent a common approach to the *iterator pattern*: element-by-element access to a collection, such that elements may be modified or accumulated [9]. Traversable functors describe a family of data structures whose elements may be *traversed* in a particular order and acted upon in an effectful manner, creating a new container of the same shape out of the results of each action. Although the interface for traversable functors is rather abstract, associated with it are rigid laws which have powerful consequences for the nature of such data structures. Traversable functors extend regular *Functors*, which embody the more general concept of mappable contexts. Functors allow for creating a new container by applying a *pure* function to each element; they are weaker than traversable functors as the exposed interface does not allow the transformation to be effectful.

Traversable functors represent containers with a finite amount of directly accessible items [2]. For example, lists correspond to a simple traversable functor, where traversing a list is defined by applying an action to each element left to right, and then combining the results into a new list.

Due to the weaker interface, functors need not represent directly accessible containers. For example, functions are mappable, as it is possible to apply a function to each possible output through function composition. Functions therefore correspond to a functor, even though each item may only be accessed through providing an input to the function.

Traversable functors are formalized via *category theory*, an abstract branch of mathematics of which many concepts within purely functional programming stem from – including functors. Other families of functors significant within functional programming that are derived from or formalized via category theory include applicative functors, monads, [19] and comonads [21].

The relationship between functors within category theory and functors within functional programming allows for many category-theoretical concepts to have direct counterparts within functional programming. A concept that has found particularly powerful applications within functional programming is that of *free constructions*. A free construction of a particular family – or rather, *category* – may roughly be described as the augmentation of an object belonging to a *simpler* family, such that it is equipped with the simplest possible additional structure needed in order for it to become a member of the more complex family.

The most familiar class of free constructions are *free monoids* – i.e. lists. Monoids are sets equipped with an associative binary operation, together with a particular member that acts as the neutral element: for example,

real numbers are a monoid under addition, where 0 is the neutral element. Any set A may be augmented to form the set of lists with items of A. This set is a monoid under list concatenation, with the empty list as the neutral element. The construction of this monoid represents the addition of the simplest possible additional structure, as the behaviour of the monoid is agnostic of the underlying A, and there is no structure to lists beyond what is needed in order to merge these together in an associative manner.

Free constructions also exist in relation to functor families: for example, free constructions have been described for applicative functors, monads and comonads. Any functor has a corresponding free member of any of these families [3, 20, 21]. To my knowledge, free constructions in relation to traversable functors have never before been formally described. It is of interest to do so, as traversable functors have found widespread use within purely functional programming, and free constructions of other functor families have found powerful applications in relation to the use of these families.

This thesis describes free constructions in relation to traversable functors through the introduction of *Cofree Traversable Functors*. These are given by *The Cofree Traversable Functor Functor*, a construction that maps any functor to a corresponding cofree traversable functor. The cofree traversable functor functor is formally defined via category theory, and this definition is then used in order to derive a corresponding type-theoretical model.

This thesis proves the existence of the cofree traversable functor functor through developing a construction that satisfies the requirements of the derived model, and therefore corresponds to an implementation of the cofree traversable functor functor. The construction also reveals additional properties beyond those that are easily identifiable from the category-theoretical definition, and these have potentially significant applications in the context of manipulating traversable functors generically.

This report is written under the assumption that the reader possesses a basic understanding of the Haskell programming language. Moderately complex topics, such as parametric polymorphism and type classes, will be explained in brief. The Haskell compiler in use for this thesis is the Glasgow Haskell Compiler (GHC) version 8.6.5 [7]. Its standard library – the package `base` version 4.12.0.0 – is also in use [6]. I have made all code presented in this report available in a GitHub repository, [24] which is structured into separate modules. The repository also expands upon the code featured in Appendix E and Appendix F.

The report first covers the relevant background of the topic (Section 2), and the methodology used in the thesis (Section 3). This is followed by the category-theoretical definition of the cofree traversable functor functor (Section 4), which is then used to derive a corresponding type-theoretical

model (Section 5). Once the model is constructed, an implementation called the *Representational Encoding* is presented (Section 6), whose validity is later proven (Section 7). The representational encoding is then implemented in Haskell (Section 8) and used to demonstrate practical applications of cofree traversable functors (Section 9). Finally, related work (Section 10) and future work (Section 11) are discussed, and then the conclusion is drawn (Section 12). The report also features a number of appendices, which are referenced throughout the report but are too great in size and of too little interest to warrant inclusion in its body.

Section 4 is written under the assumption that the reader possesses a moderately advanced understanding of category theory, including topics such adjoint functors and monoidal functors. A reader who is not interested in the category-theoretical background of cofree traversable functors may skip this section, as it serves only to formally define the unique characteristics of the cofree traversable functor functor from which the type-theoretical description in Section 5 is derived. However, skipping Section 4 may cause the derivation performed in Section 5 to become difficult to understand.

For introductory material to category theory, I recommend *Category Theory* by Steve Awodey [1]. Monoidal categories and monoidal functors are not covered by this book; material for these topics may instead be found in the more advanced *Categories for the Working Mathematician* by Saunders Mac Lane [16].

2 Background

2.1 Parametric Polymorphism and Kinds

Parametric polymorphism is the ability to associate generic *type variables* as parameters to values or types. Expressions which are subject to parametric polymorphism are known as polymorphic.

An example of a polymorphic function in Haskell is the following:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

In Haskell, unless constraints are enforced upon the type variable, the value must be constructed without any information about the provided type itself. Therefore, the value is defined for any provided type, and the structure of the value is the same no matter what the provided type is. This property of polymorphic values is known as *parametricity*, and may be used to derive

results about polymorphic expressions from their types alone [23]. Choosing a particular type for the type variable is called *instantiating* the type variable.

Type variables associated to expressions that are brought into scope from the use of parametric polymorphism are called *universally quantified*. In Haskell, this is reflected by the fact that with the `ExplicitForAll` GHC language extension enabled, type variables may explicitly be introduced into scope through the use of the `forall` keyword:

```
reverse :: forall a. [a] -> [a]
```

In this thesis, the use of parametric polymorphism in expressions and the associated introduction of type variables is denoted through ∀. Any proofs presented in this thesis does not assume that parametricity holds.

The mechanism of parametric polymorphism differ when used to associate type variables to a *type*. An example of a data type that is defined through the use of parametric polymorphism is the following:

```
data Maybe a
  = Nothing
  | Just a
```

Like parametric polymorphism for values, this defines a corresponding type `Maybe a` for any `a`. Unlike values, this polymorphism is not achieved through introducing type-variables into scope via universal quantification. Instead, this definition introduces the *type-level expression* `Maybe`. This expression accepts any type `a` as an argument, and maps it to the corresponding type `Maybe a`. `Maybe` is also called a type, but it is markedly different from regular types: it has no values, and is instead used to transform a regular type to another regular type. `Maybe` is therefore called a *type constructor*.

`Maybe` is distinguished from regular types, such as `Bool`, through its *kind*. Kinds are the types of type-level expressions. Similar to type annotations, `a :: k` denotes that the type-level expression `a` has kind `k`. In Haskell, the kind of regular types is written as ∗; for example, `Bool :: *`. However, the kind of `Maybe` is `* -> *`, signifying that it takes a regular type, and from it creates a regular type. It is possible to create data types with multiple parameters, in which case the kind of the corresponding type constructor reflects the *arity* of the type constructor as well as the kind of its parameters. For example:

```
data Product f g a = Pair (f a) (g a)
```

The kind of `Product` is `(* -> *) -> (* -> *) -> * -> *`, reflecting that it takes *two* unary type constructors of kind `* -> *`, one regular type ∗, and

produces a type $*$. Note that type constructors may be partially applied, so it is also possible to view `Product` as a type-level expression that takes two unary type constructors, and produces a unary type constructor.

In this thesis, the kind of regular types will be written as Set, reflecting the connection that these types have to the category of sets within category theory.

2.2 Type Classes

Type classes are a construct that allows for defining *interfaces*, such that an implementation of such an interface is associated with a particular type, or groups of types.

The following is a simplification of the `Eq` type class, which is an interface for types with values that may be compared for equality [6].

```
class Eq a where
  (==) :: a -> a -> Bool
```

Implementing this interface for a particular type is called creating an *instance* of the `Eq` type class for that type. The following example showcases how an instance of this type class would be declared and implemented for booleans.

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
```

Instances may only be declared at the top level, and apply globally as long as they are in scope. It is not possible to create multiple instances that overlap for a type, nor is it possible to discard any instance in scope.

Type classes may be used to increase the power of polymorphic values through introducing *constraints* on the type variable. For example:

```
-- Checks whether the argument is an element of the list
elem :: Eq a => a -> [a] -> Bool
elem _ []     = False
elem a (x:xs) = a == x || elem a xs
```

`elem` constrains the type variable it universally quantifies to be of a type that is an instance of `Eq`. As such, it may make use of the `(==)` operation defined by this type class for values of that type.

Type variables and constraints upon them may also be used when defining instances for type classes. For example, the following instance states that

equality for lists is defined for all lists with elements that may be compared for equality.

```
instance Eq a => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```

Constraints may even be used when defining type classes, in which case the underlying type class of any constraint is called a *superclass* of the new type class. For example, the `Ord` type class, for ordered types, has the `Eq` type class as a superclass, requiring that any type with an instance of the `Ord` class also is an instance of the `Eq` class.

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
```

2.3 Functors and Natural Transformations

A *functor* F in the context of functional programming is a mapping of types and functions, such that:

- For each type $X : \mathsf{Set}$, there exists the corresponding type $F(X) : \mathsf{Set}$.
- For each function $f : A \to B$, there exists the corresponding function $F(f) : F(A) \to F(B)$.

The mapping of types F is expressed through a type constructor $F : \mathsf{Set} \to \mathsf{Set}$. The mapping of functions may be expressed through a polymorphic function:

$$\mathtt{map}^F : \forall x\ y.\ (x \to y) \to F(x) \to F(y)$$

such that

$$\mathtt{map}^F\ f = F(f)$$

The mapping of functions has two laws, that must be satisfied:

- *Preservation of Identity*

  For all $A : \mathsf{Set}$

  $$F(\mathsf{id}_A) = \mathsf{id}_{F(a)}$$

  where id is the identity function:

  $$\mathsf{id} : \forall x.\ x \to x$$
  $$\mathsf{id}\ x = x$$

- *Preservation of Composition*

  For all $A, B, C : \mathsf{Set}$, $f : B \to C$, $g : A \to B$

  $$F(f) \circ F(g) = F(f \circ g)$$

  where $\circ$ is function composition:

  $$\circ : \forall x\ y\ z.\ (y \to z) \to (x \to y) \to (x \to z)$$
  $$f \circ g = \lambda x.\ f\ (g\ x)$$

A functor can roughly be described as a mappable context, or mappable container. Each value $F(A)$ encapsulates some element(s) of type A, which may be mapped over by lifting a function $f : A \to B$ to $F(f) : F(A) \to F(B)$. In this thesis, the pseudotype $[\mathsf{Set}, \mathsf{Set}]$ representing the type of all functors will be used in the context of *universal quantification* over all functors. In addition, any value $x : F(A)$ will be called a *functorial value*.

Functors correspond to the Haskell type class `Functor`, defined as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- Infix operator for 'fmap'
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> fa = fmap f fa
```

Note that the types associated with the `Functor` type class are *type constructors* $\mathsf{Set} \to \mathsf{Set}$.

The type constructor for lists is a simple functor:

```
instance Functor [] where
  -- fmap :: (a -> b) -> [a] -> [b]
  fmap _ [] = []
  fmap f (x:xs) = f x : fmap f xs
```

However, functors are not always intuitively containers. For example, given any type $S : \mathsf{Set}$, the type constructor $\lambda x.\ S \to x$ is a functor. This functor corresponds to the type of functions where the domain is fixed to S, which is written in haskell as `(->) s`.[1] Mapping of functions then correspond to *function composition.*

[1] Type variables must be begin with a lowercase letter in Haskell.

```haskell
instance Functor ((->) s) where
  -- fmap :: (a -> b) -> (s -> a) -> (s -> b)
  fmap = (.)
```

It is at times desirable to speak of *mappings* between functors: transformations from one functor to another. These are expressed in the form of *natural transformations*. A natural transformation from a functor F to a functor G is a family of functions α, such that for any $X : \mathsf{Set}$

$$\alpha_X : F(X) \to G(X)$$

A natural transformation α may therefore be represented through a polymorphic function

$$\alpha : \forall x.\ F(x) \to G(x)$$

Natural transformations are subject to the following law, which expresses a form of structure-preservation:

For all $X, Y : \mathsf{Set}$, $f : X \to Y$

$$G(f) \circ \alpha_X = \alpha_Y \circ F(f)$$

This law is called the *naturality condition* for natural transformations.

Any polymorphic function satisfying this requirement is therefore a natural transformation. An example of a natural transformation in Haskell is `listToMaybe`, as shown below:

```haskell
data Maybe a = Nothing | Just a

instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f (Just a) = Just (f a)
  fmap _ Nothing = Nothing

listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (x:_) = Just x
```

The following shows how, for any function `f : x -> y`, `listToMaybe` satisfies the naturality condition.

```haskell
  fmap f (listToMaybe [])
= fmap f Nothing
= Nothing
```

```
= listToMaybe []
= listToMaybe (fmap f [])

  fmap f (listToMaybe (x:xs))
= fmap f (Just x)
= Just (f x)
= listToMaybe (f x : fmap f xs)
= listToMaybe (fmap f (x:xs))
```

The type of polymorphic functions corresponding to natural transformations from functor F to functor G is written as

$$\alpha : [\mathsf{Set}, \mathsf{Set}](F, G)$$

or

$$\alpha : F \to G$$

when unambiguous.

Both functors and natural transformations in functional programming are specializations of their category-theoretical counterparts. In category-theoretical terms, functors in functional programming are endofunctors in the category of sets. Such endofunctors are objects of another category, denoted $[\mathsf{Set}, \mathsf{Set}]$, from which the name of the pseudotype is derived. The notation used for the type of natural transformations reflects that natural transformations are the *morphisms* – i.e. *mappings* – between the objects in that category.

2.4 Applicative Functors

Traversable functors have been summarized as data structures that allow for ordered effectful transformation of their elements. Effects in this context are modelled by *Applicative Functors*: functors with a structure that is *monoidal* such that multiple functorial values may be combined together into a single functorial value.

Applicative functors are functors equipped with two additional operations: `pure` and $\circledast$, such that for any applicative functor F

$$\begin{aligned} &\texttt{pure} : \forall x.\ x \to F(x) \\ &\circledast : \forall x\ y.\ F(x \to y) \to F(x) \to F(y) \end{aligned}$$

subject to the following laws:

- *Identity*

  For all $A : \mathsf{Set}$, $v : F(A)$

  $$\texttt{pure}\ \mathsf{id}_A \circledast v = v$$

- *Composition/Associativity*

  For all $A, B, C : \mathsf{Set}$, $u : F(B \to C)$, $v : F(A \to B)$, $w : F(A)$

  $$u \circledast (v \circledast w) = ((\texttt{pure}\ (\circ) \circledast u) \circledast v) \circledast w$$

  Where $\circ$ is function composition.

- *Homomorphism*

  For all $A, B : \mathsf{Set}$, $f : A \to B$, $a : A$

  $$\texttt{pure}\ f \circledast \texttt{pure}\ a = \texttt{pure}\ (f\ a)$$

- *Interchange*

  For all $A, B : \mathsf{Set}$, $u : F(A \to B)$, $x : A$

  $$u \circledast \texttt{pure}\ x = \texttt{pure}\ (\lambda f.\ f\ x) \circledast u$$

- *Consistency*

  For all $A, B : \mathsf{Set}$, $f : A \to B$, $v : F(A)$

  $$\texttt{pure}\ f \circledast v = F(f)\ v$$

These operations, together with these laws, allows for wrapped values to be combined with $\circledast$, such that composition of contexts is associative, with `pure` wrapping a value in a neutral context. $\circledast$ infixes to the left, as the most common use of applicative functors is to lift functions of arbitrary rarity, and left-nested uses of $\circledast$ are common in such contexts. For example:

$$\begin{array}{ll} f & : A \to B \to C \\ u & : F(A) \\ v & : F(B) \\ (\texttt{pure}\ f \circledast u) \circledast v & : F(C) \end{array}$$

The pseudotype App represents the type of all applicative functors, and will be used in the context of *universal quantification* over all applicative functors.

Applicative functors correspond to the `Applicative` type class.[2] A simplified definition is as follows:

[2]Defined in the module `Control.Applicative` of `base` [6].

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  infixl 4 <*> -- <*> infixes to the left.
```

An example of an applicative functor is the one corresponding to the partially-applied two-tuple $\lambda a.\ (S, a)$, written `(,) s` in Haskell. This corresponds to an applicative functor when S is a monoid. The monoidal structure for this applicative functor is the monoid itself; `pure` pairs its argument with the neutral element, and `<*>` combines two tuples by combining the function and argument present at the right-hand side, and multiplying the monoidal values present at the left-hand side.

For example, consider the monoid on strings:

```haskell
ex :: (String, Int)
ex  = pure (+) <*> ("one.", 1) <*> ("two.", 2)
    = ("", (+)) <*> ("one.", 1) <*> ("two.", 2)
    = ("" ++ "one.", (+) 1) <*> ("two.", 2)
    = (("" ++ "one.") ++ "two.", (+) 1 2)
    = ("one.two.", 3)
```

This applicative functor encodes the effect of combining environmental information from multiple parts of a program, and is particularly useful for logging. It is more commonly known as the *Writer* applicative functor.

As a consequence of parametricity, valid instances of the `Functor` type class are unique, [11] meaning that in Haskell the *Consistency* law is automatically satisfied for applicative functors if they obey all other laws.

An *applicative morphism* $\tau : \mathsf{App}(F, G)$, also written $\tau : F \to G$ when unambiguous, is a mapping between applicative functors F to G. An applicative morphism is a natural transformation that also satisfies the following laws:

$$\tau\ (\mathtt{pure}^F\ a) = \mathtt{pure}^G\ a$$
$$\tau\ (v \circledast^F w) = \tau\ v \circledast^G \tau\ w$$

In this thesis, applicative morphisms are only relevant in the context of defining laws for traversable functors, and thus will not be explored in further detail.

2.5 Traversable Functors

A *Traversable Functor* is a functor equipped with an operation that allows mapping each element within the container to a value wrapped in an applicative context. These values are then combined together using applicative

operations to form a new container with the same shape as the original, wrapped in the applicative context. In effect, this allows mapping an effectful action over each element of a container, creating a new container of the same shape out of the result of each action.

A traversable functor is a functor T equipped with an additional operation `traverse`, such that for any applicative functor $F : \mathsf{App}$, and any types $A, B : \mathsf{Set}$

$$\texttt{traverse}^{F}_{A,B} : (A \to F(B)) \to T(A) \to F(T(B))$$

This may be written as a polymorphic function:

$$\texttt{traverse} : \forall (F : \mathsf{App})\ x\ y.\ (x \to F(y)) \to T(x) \to F(T(y))$$

Values of the form $t : T(A)$ will be referred to as *traversable containers*.

`traverse` is subject to the following laws:

- *Identity*:

  For all $X : Set$

  $$\texttt{traverse}^{\mathbf{1}}_{X,X}\ \mathsf{id}_X = \mathsf{id}_{T(X)}$$

  Where $\mathbf{1}$ is the identity (applicative) functor, given as follows:

  $$\begin{aligned}
  &\text{For any } X : Set, \mathbf{1}(X) = X\\
  &\texttt{map}^{\mathbf{1}}\ f = f\\
  &\texttt{pure}^{\mathbf{1}}\ a = a\\
  &u \circledast^{\mathbf{1}} w = u\ w
  \end{aligned}$$

- *Composition/Linearity*:

  For all $F, G : \mathsf{App}$, $X, Y, Z : \mathsf{Set}$, $f : X \to F(Y)$, $g : Y \to G(Z)$

  $$\begin{aligned}
  & F(\texttt{traverse}^{G}_{Y,Z}\ g) \circ \texttt{traverse}^{F}_{X,Y}\ f\\
  = \ & \texttt{traverse}^{(F \circ G)}_{X,Z}\ (F(g) \circ f)
  \end{aligned}$$

  Where $F \circ G$ is the composition of (applicative) functors F and G, defined as follows:

  $$\begin{aligned}
  &\text{For any } X : \mathsf{Set}, (F \circ G)(X) = F(G(X))\\
  &\texttt{map}^{(F \circ G)}\ f = F(G(f))\\
  &\texttt{pure}^{(F \circ G)}\ a = \texttt{pure}^{F}\ (\texttt{pure}^{G}\ a)\\
  &u \circledast^{(F \circ G)} w = \texttt{pure}^{F}\ (\circledast^{G})\ \circledast^{F}\ u\ \circledast^{F}\ w
  \end{aligned}$$

- *Naturality*

  For all $F, G : \mathsf{App}$, $A, B : \mathsf{Set}$, $\tau : \mathsf{App}(F, G)$, $f : A \to F(B)$

$$\begin{aligned} & \mathtt{traverse}^{G}_{A,B}\ (\tau_B \circ f) \\ = \ & \tau_{T(B)} \circ \mathtt{traverse}^{F}_{A,B}\ f \end{aligned}$$

- *Consistency*

  For all $A, B : \mathsf{Set}$, $f : A \to B$

$$\mathtt{traverse}^{\mathbf{1}}_{A,B}\ f = \mathtt{map}^{T}\ f$$

Traversable functors may alternatively be described through the polymorphic function `sequence`, which follows a similar set of laws to `traverse`.

$$\mathtt{sequence} : \forall (F : \mathsf{App})\ x.\ T(F(x)) \to F(T(x))$$

Any valid definition of `traverse` has a corresponding valid definition of `sequence`, and vice versa:

$$\begin{aligned} \mathtt{traverse}^{F}_{A,B}\ f &= \mathtt{sequence}^{F}_{B} \circ T(f) \\ \mathtt{sequence}^{F}_{A} &= \mathtt{traverse}^{F}_{A,A}\ \mathsf{id}_A \end{aligned}$$

Traversable functors correspond to the `Traversable` type class in Haskell.[3] A simplified definition is as follows:

```
class Functor t => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  -- Default implementation: requires sequenceA to be defined
  traverse f = sequenceA . fmap f

  sequenceA :: Applicative f => t (f a) -> f (t a)
  -- Default implementation: requires traverse to be defined
  sequenceA = traverse id
```

Due to the default implementations provided, any instance of `Traversable` is only required to define the implementation of one of the two methods. If neither method is defined within the instance, then any use of either will result in an infinite loop due to the mutual recursion.

Note that the type class as defined by `base` has an additional superclass named `Foldable`. This is because the methods of `Traversable` may be used

[3] Defined in the `Data.Traversable` module of `base` [6].

to define a legal instance of `Foldable`, and thus, any `Traversable` must be `Foldable`. The `Foldable` type class is of no interest in this thesis, and will be ignored.

Like applicative functors, the *Consistency* law for traversable functors is automatically satisfied in Haskell if the instance obeys all other laws. In addition, parametricity also implies that the *Naturality* law of traversable functors is always satisfied [11]. Therefore, the only significant laws are the *Identity* and *Linearity* laws, which together restrict traversals such that they fit the intuitive behaviour of iteration over containers [13].

The *Identity* law restricts traversals so that they may not duplicate, remove, or rearrange elements, nor alter the structure of the traversed container. Any element is traversed at least once, and the resulting value of an action on an element is placed within the resulting structure exactly at the position where the original element was located.

The *Linearity* law restricts traversals such that the same element is not traversed more than once: i.e. it forbids any traversal from using the provided action on the same item multiple times, repeating effects.

An example of a traversable functor is the one for lists. The `Traversable` instance for `[]` is given by traversing the elements left to right:

```
instance Traversable [] where
  -- sequenceA :: Applicative f => [f a] -> f [a]
  sequenceA [] = pure []
  sequenceA (x:xs) = (:) <$> x <*> sequenceA xs

sequenceA [("one.", 1), ("two.", 2), ("three.", 3)]
  = ("one.two.three.", [1,2,3])
```

Another example of a traversable functor is the one for binary trees. The `Traversable` instance is given by traversing the tree in order:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap f Leaf = Leaf
  fmap f (Node l a r) = Node (fmap f l) (f a) (fmap f r)

instance Traversable Tree where
  -- sequenceA :: Applicative f => Tree (f a) -> f (Tree a)
  sequenceA Leaf =
    pure Leaf
```

```
sequenceA (Node l a r) =
  Node <$> sequenceA l <*> a <*> sequenceA r
```

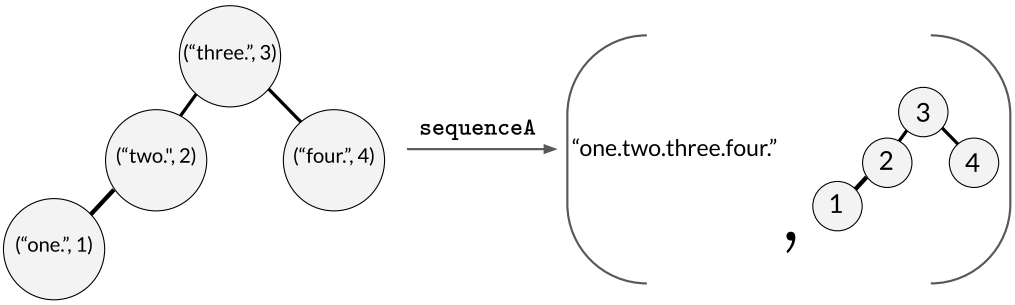


Figure 1: Demonstration of `sequenceA` for `Tree`.

An issue with the `Traversable` type class is that Haskell's type system enforces that only one instance of the class may exist for a given type constructor: however, there may exist *multiple* valid implementations with differing semantics. As Bird et al. have shown, these differ only in the order of which effects from each element are combined [2].

For example, rather than processing effects from left to right, lists could instead process these right to left:

```
sequenceARev :: Applicative f => [f a] -> f [a]
sequenceARev [] = pure []
sequenceARev (x:xs) = flip (:) <$> sequenceARev xs <*> x

sequenceARev [("one.", 1), ("two.", 2), ("three.", 3)]
  = ("three.two.one.", [1,2,3])
```

Note that although the actions are executed in reverse, the resulting values of each action (i.e. the right-hand side of each tuple) are placed at positions corresponding to the original container, as is required by the *Identity* law. This implementation would be a valid instance of `Traversable` for lists, but the instance provided by `base` instead uses the version that traverses the

list left to right [6]. Similarly, the traversable instance for `Tree` could be implemented by using a different tree traversal, such as pre- or post-order.

In this thesis, individual implementations of traversable functors for the same underlying functor T are of interest, and thus T is not associated with a canonical implementation of `traverse` or `sequence`. Instead, a traversable functor is represented by a particular pairing of a functor T and a valid definition of `traverse` (or `sequence`) for it. Such a definition is called a *traversal* of T.

The pseudotype Tra represents the type of all traversable functors, and will be used in the context of *universal quantification* over all traversable functors. Members of Tra are denoted $(T, \delta) : \mathsf{Tra}$, where T is the underlying functor, and δ is a traversal of it. The use of a specific traversal δ for `traverse` or `sequence` is indicated through ${}^{\delta}\texttt{traverse}$ and ${}^{\delta}\texttt{sequence}$, respectively.

2.6 Traversable Morphisms

Mappings between traversable functors are expressed through *traversable morphisms*. The term is original to this thesis, although the concept has previously been discussed by Bird et al. [2]

A traversable morphism $\alpha : (T, \delta) \to (U, \varepsilon)$ is a polymorphic function

$$\alpha : \forall x.\ T(x) \to U(x)$$

which corresponds to a natural transformation, and is subject to the following:

For all applicative functors $F : \mathsf{App}$, $A : \mathsf{Set}$

$${}^{\varepsilon}\texttt{sequence}^{F}_{A} \circ \alpha_A = F(\alpha_A) \circ {}^{\delta}\texttt{sequence}^{F}_{A}$$

This may equivalently be stated as follows:

For all applicative functors F, $A, B : \mathsf{Set}$, $f : A \to F(B)$

$${}^{\varepsilon}\texttt{traverse}^{F}_{A,B}\ f \circ \alpha_A = F(\alpha_B) \circ {}^{\delta}\texttt{traverse}^{F}_{A,B}\ f$$

The condition when formulated in terms of `traverse` implies that α satisfies the naturality condition for natural transformations, which thus does not need to be proven separately.

Proof For any $(T, \delta), (U, \varepsilon) : \mathsf{Tra}$, $\alpha : (T, \delta) \to (U, \varepsilon)$, $A, B : \mathsf{Set}$, $f : A \to B$

$$
\begin{aligned}
& U(f) \circ \alpha_A \\
= \; & \text{(Consistency law of traversable functors)} \\
& {}^{\varepsilon}\mathtt{traverse}^{\mathbf{1}}_{A,B}\ f \circ \alpha_A \\
= \; & (\alpha \text{ is a traversable morphism.}) \\
& \mathbf{1}(\alpha_B) \circ {}^{\delta}\mathtt{traverse}^{\mathbf{1}}_{A,B}\ f \\
= \; & \alpha_B \circ {}^{\delta}\mathtt{traverse}^{\mathbf{1}}_{A,B}\ f \\
= \; & \text{(Consistency law of traversable functors)} \\
& \alpha_B \circ T(f)
\end{aligned}
$$

□

As Bird et al. have shown, traversable morphisms are equivalent to what they call *contents-preserving functions* [2]. These are characterized by their interaction with the the operation `contents`, which may be defined for any traversable functor.

For a given traversable functor (T, δ), let

$$
\begin{aligned}
& {}^{\delta}\mathtt{contents}_A : T(A) \to \mathsf{List}(A) \\
& {}^{\delta}\mathtt{contents}_A = {}^{\delta}\mathtt{traverse}^{\mathsf{C}(\mathsf{List}A)}_{A,\bot}(\lambda x.\ [x])
\end{aligned}
$$

where $\bot$ is the empty type, and $\mathsf{C}(\mathsf{List}(A))$ is the *constant functor* of $\mathsf{List}(A)$, which maps any function $f : X \to Y$ to the identity function.

$$
\begin{aligned}
& \mathsf{C}(\mathsf{List}(A))(X) = \mathsf{List}(A) \\
& \mathtt{map}^{\mathsf{C}(\mathsf{List}(A))} : \forall x\ y.\ (x \to y) \to \mathsf{C}(\mathsf{List}(A))(x) \to \mathsf{C}(\mathsf{List}(A))(y) \\
& \mathtt{map}^{\mathsf{C}(\mathsf{List}(A))}_{X,Y}\ f = \mathsf{id}
\end{aligned}
$$

A value $\mathsf{C}(\mathsf{List}(A))(X)$ thus only consists of a list with elements of A, completely disregarding the parameter X. This is trivially applicative, as the entire structure is nothing but a monoidal value (the list), and thus `pure` and $\circledast$ are defined purely through the monoidal unit (empty list) and multiplication (list concatenation), respectively. C corresponds to the Haskell `Const` functor.[4]

${}^{\delta}\mathtt{contents}_A$ traverses a value $t : T(A)$, using $\mathsf{C}(\mathsf{List}(A))$ as the effectful context to gather the elements in the order they are given into a list.
Thus, ${}^{\delta}\mathtt{contents}_A\ t$ represents the elements of t, ordered according to the traversal δ.

[4] See `Data.Functor.Const` [6].

A *contents-preserving function* $\alpha : (T, \delta) \to (U, \epsilon)$ is a polymorphic function corresponding to a natural transformation $T \to U$, and is subject to the following condition:

For all A : Set

$$^{\varepsilon}\texttt{contents}_A \circ \alpha_A = {}^{\delta}\texttt{contents}_A$$

As contents-preserving functions are equivalent to traversable morphisms, this gives an intuitive understanding of what a traversable morphism is: traversable morphisms are natural transformations that preserve the exact elements of the transformed traversable container and the order of those elements.

An example of a traversable morphism is `contents` itself: it is a traversable morphism from any traversable functor to (List, σ), where σ is the traversal for List which processes effects left to right.[5] For a proof, see Appendix A.

For example, take the traversable functor yielded by Maybe and its only valid traversal:

```
data Maybe a = Nothing | Just a

instance Traversable Maybe where
  traverse f (Just a) = Just <$> f a
  traverse _ Nothing = pure Nothing
```

In order to define `contents` in Haskell through the use of `Const`, additional boilerplate is necessary in order to wrap and unwrap values within the `Const` data type:

```
contents :: Traversable t => t a -> [a]
contents t = getConst (traverse (\a -> Const [a]) t)
```

Using this definition, it is possible to show that `contents` for Maybe is a contents-preserving function:

```
contents (Just a) = [a]
contents (contents (Just a)) = [a]

contents Nothing = []
contents (contents Nothing) = []
```

`contents` is therefore a traversable morphism from the unique traversable functor for Maybe to (List, σ).

[5] See Section 2.5

The following is an example of a natural transformation that is not a traversable morphism:

```
listToMaybe :: [a] -> Maybe a
listToMaybe (x:_) = Just x
listToMaybe []    = Nothing
```

As shown in Section 2.3, this polymorphic function satisfies the naturality condition, and is thus a natural transformation. However, it is not a contents-preserving function, and therefore not a traversable morphism:

```
contents (listToMaybe [1,2,3]) = contents (Just 1) = [1]
contents [1,2,3] = [1,2,3]
```

2.7 Free Constructions

Free constructions have been summarized as strengthening an object with the simplest possible additional structure needed in order to create a member of a more complex family. However, this is only a description of the intuitive properties that free constructions tend to possess. The actual definition within category theory is significantly more abstract.

There are two variants of free constructions: *free objects* and *cofree objects*, which are yielded by what are known as *free functors* and *cofree functors*, respectively.[6] These are general category-theoretical functors, of which functors in functional programming are only a specialization of.

A *functor* is a mapping between *categories*. A category consists of one collection of *objects* and one of *morphisms* – which may be described as arrows or mappings between these objects. Chains of morphisms may be composed together into one, and each object has an *identity morphism* to itself, which acts like the neutral element of morphism composition; i.e. it does not have any effect.

A categorical functor $F : \mathscr{A} \to \mathscr{B}$ maps each object X of a category $\mathscr{A}$ to an object $F(X)$ of a category $\mathscr{B}$ (potentially the same one), and maps each morphism $f : X \to Y$ of $\mathscr{A}$ to a corresponding morphism $F(f) : F(X) \to F(Y)$ of $\mathscr{B}$.

In category theory, many categories extend others, such that objects are the same but equipped with additional power, and morphisms are the same

[6] In certain contexts where free constructions are of interest, it is impossible to validly define a corresponding co/free functor. In these cases, co/free objects are instead defined as objects possessing properties *as though* they were given by a particular co/free functor, even if that functor does not exist.

but have additional restrictions placed upon them respecting the additional power.

Co/free functors are particular functors that map a simpler category to a more complex category which is based upon the simpler one: each object of the simpler category is mapped to some structure that is based upon that object, but is a member of the more complex category. Morphisms are mapped to morphisms between these structures, such that the morphism of the simpler category is used to change what object the structure is based upon.

Co/free functors are uniquely identified through two families of morphisms that are associated with them, one for each category. These families are known as the *unit* and the *counit*, and are subject to two laws known together as the *triangle identities*, which state that certain combinations of the unit and counit produce identity morphisms. The notion of "simplest possible additional structure" originates from the restrictions placed by the triangle identities.

Roughly described, the unit gives a morphism from any object of the simpler category to the co/free object based upon it, and the counit gives a morphism from any co/free object to the object that it is based upon, thereby allowing these structures to be converted between each other.[7]

The difference between free functors and cofree functors lies in the nature of the unit and the counit. For free functors, the *unit* exists in the simpler category: any object has a morphism to the corresponding free object (as viewed in simplified lens of that category). However, the *counit* exists in the more complex category: in order to convert a free object back to the object it is based upon, the object *also* needs to be a member of the more complex category; i.e. it needs to be equipped with the additional strength that the more complex category requires. For cofree functors the reverse is true: the *counit* exists in the simpler category, allowing any cofree object to be converted to the object it is based upon, but the *unit* exists in the more complex category, such that each morphism it describes between objects to the cofree objects based upon them exist only if the underlying object may be equipped with the additional strength that the more complex category requires.

For example, free monoids – lists – are free objects. In category theory, sets are represented through the category of sets Set, where objects are sets and morphisms are functions. The category of monoids Mon extend upon this category by having its objects be monoids, and its morphisms be *monoid*

[7] Unlike co/free functors, which map each object *to* a co/free object, the unit and counit are mappings *between* these objects; they are families of morphisms, rather than functors.

homomorphisms: that is, functions subject to certain restrictions such that they respect the monoidal structure.

Lists map any set A to the set $\mathsf{List}(A)$, which is a monoid, and thus a member of Mon; lists, collectively, are a free functor. Any element of A may be converted to an element of $\mathsf{List}(A)$ by creating a singleton list out of that element. This is the family of morphisms described by the *unit*. However, in order to convert an element (a list) of $\mathsf{List}(A)$ to an element of A, it requires that the items of the list be merged together into one, as well as access to a particular element of A which the empty list may be mapped to. This may be done if A is a monoid, by merging items through monoidal multiplication, and mapping the empty list to the neutral element. This is the family of morphisms described by the *counit*, and as these lie in the category Mon, they are not only functions, but also monoid homomorphisms.

In comparison to free functors, cofree functors are considerably less common in the context of classical algebraic structures, and consequently a similarly simple example of cofree objects cannot be made [18].

This thesis presents *the Cofree Traversable Functor Functor*, which is a cofree functor from the category of endofunctors of Set – i.e. functors within functional programming – to the category of traversable functors. *Cofree Traversable Functors* are the corresponding *cofree objects*. This means that any traversable container of a traversable functor (T, δ) may be converted to a traversable container of the cofree traversable functor on T through a traversable morphism, and that any traversable container of the cofree traversable functor of any functor F may be converted to a functorial value of F through a natural transformation.

This thesis does not consider *free* traversable functors, as a general free traversable functor functor does not exist: a simple proof by contradiction is presented in Appendix B.

3 Method

This thesis makes use of a *formal* methodology [4]. A formal methodology is characterized by the use of theoretical computer science and mathematics by constructing an abstract model of the problem under study, and reasoning about the problem through this model. Any solutions to the problem are presented mathematically.

The problem to be addressed is defining a form of free constructions for traversable functors; specifically, *cofree objects* for traversable functors. This requires a concrete definition of these, which may then serve as the model of which an implementation must be found. The definition is done mathemati-

cally, by representing traversable functors in a category-theoretical setting, and from it, the cofree objects thereof. As category theory is too abstract to serve as the underlying language of the thesis, the definition is translated to a type-theoretical interface, represented by an abstract data type together with operations and laws thereof, such that any valid implementation of the interface corresponds to a *constructive* definition of cofree traversable functors. This interface thus serves as the target model: cofree traversable functors may be defined by finding an implementation of the interface, and proving that it is valid. Not only would this prove the existence of cofree traversable functors, but would also demonstrate how these may be constructed in the setting of functional programming.

This thesis provides an implementation of the interface, complete with proofs that it satisfies the associated laws. This is then followed by exploring the potential applications of that implementation.

4 Category-Theoretical Definition

Cofree traversable functors are defined using category theory. This definition will later be used to derive the abstract data type and interface for the construction of interest.

The description makes use of category-theoretical descriptions of *applicative functors*, *applicative morphisms*, and *traversals* as given by Jaskelioff and Rypacek [13]. These will be restated within this description. The concept of the category of traversable functors (and cofree traversable functors) is original to this thesis.

4.1 Applicative Functors and Applicative Morphisms

Definition 4.1 (Applicative Functor)**.** An applicative functor is a lax monoidal functor[8] $(F : \mathsf{Set} \to \mathsf{Set}, \epsilon : 1 \to F(1), \mu_{x,y} : F(x) \times (y) \to F(x \times y))$, where the tensor product is the cartesian product, such that the canonical tensorial strength $\sigma_{x,y} : x \times F(y) \to F(x \times y)$ for F is coherent with the monoidal structure. I.e. the lax monoidal functor (F, ϵ, μ) is an applicative functor if the following diagram commutes, where $\alpha_{x,y,z} : (x \times y) \times z \to x \times (y \times z)$ is

[8] Lax monoidal functors are monoidal functors as defined by Mac Lane [16]. *Lax* indicates that they lack the additional requirements of strong or strict monoidal functors.

the associator for the cartesian product:

$$\begin{array}{ccccc} (FX \times FY) \times Z & \xrightarrow{\alpha} & FX \times (FY \times Z) & \xrightarrow{FX \times \sigma} & FX \times F(Y \times Z) \\ \downarrow{\scriptstyle \mu \times Z} & & & & \downarrow{\scriptstyle \mu} \\ F(X \times Y) \times Z & \xrightarrow{\sigma} & F((X \times Y) \times Z) & \xrightarrow{F\alpha} & F(X \times (Y \times Z)) \end{array}$$

Applicative functors form the category App, where morphisms are *applicative morphisms*, as defined in Definition 4.2. In this report, abuse of notation is often used when quantifying applicative functors: $F : \mathsf{App}$ is used to refer to a triple (F, ϵ^F, μ^F).

The identity functor $\mathbf{1}$ is an applicative functor, and composition of applicative functors is applicative. Hence, App is a monoidal category, with functor composition $\circ$ as the tensor product, and identity functor $\mathbf{1}$ as unit.

It may be difficult to understand the connection between this definition of applicative functors and the `Applicative` type class in Haskell: what follows is a clarification of the relationship.

Lax monoidal functors $\mathsf{Set} \to \mathsf{Set}$ with respect to the cartesian product may intuitively be represented in Haskell through the following type class:

```
class Functor f => Monoidal f where
  unit :: f () -- Monoidal unit ϵ.
               -- Since 1 → x is isomorphic to x,
               -- the argument is dropped.
  (<.>) :: f a -> f b -> f (a, b)
     -- Monoidal action μ, but curried.
```

subject to the following laws:

- *Naturality*

  ```
  fmap (f *** g) (u <.> v) = fmap f u <.> fmap g u
  ```

  where

  ```
  (***) :: (a -> b) -> (c -> d) -> (a, c) -> (b, d)
  (f *** g) (a, c) = (f a, g c)
  ```

  This law signifies that the monoidal action μ which `(<.>)` corresponds to truly is a natural transformation $F(x) \times F(y) \to F(x \times y)$.

- *Left and Right Identities*

  ```
  fmap snd (unit <.> v) = v
  fmap fst (v <.> unit) = v
  ```

These laws correspond to the unitality condition for lax monoidal functors.

- *Associativity*

  ```
  fmap assoc (u <.> (v <.> w)) = (u <.> v) <.> w
  ```

  where

  ```
  assoc :: (a, (b, c)) -> ((a, b), c)
  assoc (a, (b, c)) = ((a, b), c)
  ```

  This law corresponds to the associativity condition for lax monoidal functors.

Note that given an `Applicative` instance for any `f`, it is possible to create an instance of `Monoidal` for `f`, and vice-versa.

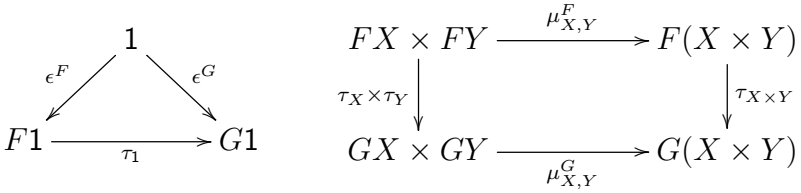
```
unit :: Applicative f => f ()
unit = pure ()

(<.>) : Applicative f => f a -> f b -> f (a, b)
fa <.> fb = (,) <$> fa <*> fb


pure :: Monoidal f => a -> f a
pure a = fmap (const a) unit

(<*>) :: Monoidal f => f (a -> b) -> f a -> f b
ff <*> fa = fmap (\(f,a) -> f a) (ff <.> fa)
```

In addition, if either instance is valid, the other must also be. These type classes are therefore equivalent in the power they provide, which is why applicative functors in category theory are represented through lax monoidal functors [17].

Applicative functors have an additional requirement compared to lax monoidal functors: that the monoidal structure is coherent with tensorial strength. The reason for this is that Haskell allows for using environmental variables within function expressions, and together with `fmap`, this allows for implicitly embedding environmental data into a functor. It is through this mechanism that `pure` may be implemented with `Monoidal`. Within category theory, this corresponds to the use of tensorial strength $\sigma_{x,y} : x \times F(y) \to F(x \times y)$. Lax monoidal functors do not require that μ remains coherent with

the involvement with strength, whereas for `Monoidal` – which is equivalent to `Applicative` – this is an implicit requirement [17]. This is addressed by the addition of the coherence requirement as given by the diagram in definition 4.1. However, this requirement is redundant, as in the setting of endofunctors of Set – the only setting of concern – this condition is satisfied for any lax monoidal functor [19].

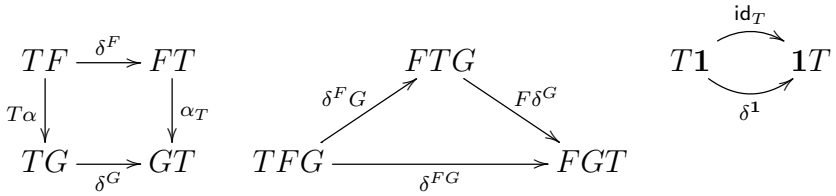
Definition 4.2 (Applicative Morphism)**.** Let (F, ϵ^F, μ^F) and (G, ϵ^G, μ^G) be two applicative functors. An *applicative morphism* is a natural transformation $\tau : F \to G$ that respects the unit and multiplication. That is, a natural transformation τ such that the following diagrams commute:



This definition and the laws thereof correspond to those of applicative morphisms as defined in Section 2.4, but expressed through the operations associated with lax monoidal functors rather than `pure` and ⊛.

4.2 Traversals and Traversable Functors

Definition 4.3 (Traversal)**.** A traversal δ of a functor $T : \mathsf{Set} \to \mathsf{Set}$ is a family of natural transformations in $\mathsf{Set} \to \mathsf{Set}$ such that for any applicative functor $F : \mathsf{App}$, $\delta^F : TF \to FT$ is a natural transformation. δ must be natural in the choice of applicative functor, and must respect the monoidal structure of applicative functor composition. Explicitly, for all applicative functors $F, G : \mathsf{App}$ and applicative morphisms $\alpha : F \to G$, the following diagrams of natural transformations commute:



The family of natural transformations δ corresponds to a valid definition of `sequence` for a functor T, and as such corresponds to the definition of a *traversal* as given in Section 2.5.

Definition 4.4 (Traversable Functor)**.** A *Traversable Functor* is an endofunctor $T : \mathsf{Set} \to \mathsf{Set}$ equipped with a traversal δ of T. Traversable functors are represented through the ordered pair (T, δ).

Note that if δ, ε are two different traversals for the same endofunctor T, then (T, δ) and (T, ε) represent two different traversable functors.

This definition deviates from the one given by Jaskelioff and Rypacek, who define a traversable functor to be an endofunctor $T : \mathsf{Set} \to \mathsf{Set}$ such that there exists *some* traversal of it [13].

4.3 The Category of Traversable Functors

Definition 4.5 (Traversable Morphism)**.** I define a *traversable morphism* between traversable functors $\alpha : (T, \delta) \to (U, \varepsilon)$ as a natural transformation $\alpha' : T \to U$ such that for all applicative functors $F : \mathsf{App}$, the following diagram commutes:

$$\begin{array}{ccc} TF & \xrightarrow{\delta^F} & FT \\ {\scriptstyle \alpha' F}\downarrow & & \downarrow{\scriptstyle F\alpha'} \\ UF & \xrightarrow[\varepsilon^F]{} & FU \end{array}$$

Composition of traversable morphisms is validly defined as composition of the underlying natural transformations.

Proof As composition of natural transformations is associative, so too is the composition of traversable morphisms under this definition.

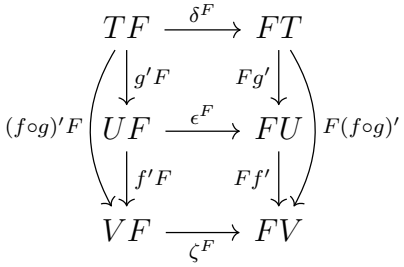
Thus, it only remains to be proven that the composition of two traversable morphisms always forms a traversable morphism.

Let

$$\begin{aligned} f &: (U, \epsilon) \to (V, \zeta) \\ g &: (T, \delta) \to (U, \epsilon) \\ f \circ g &: (T, \delta) \to (V, \zeta) \\ (f \circ g)' &= f' \circ g' \end{aligned}$$

If f and g are traversable morphisms, then so is $f \circ g$, as shown in the following

commutative diagram:

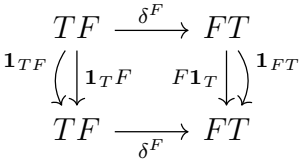


The definition is therefore valid. □

The identity natural transformation $\mathbf{1}_T$ forms the identity traversable morphism of any traversable functor (T, δ).

Proof As composition of traversable morphisms is defined through composition of the underlying natural transformations, if $\mathbf{1}_T$ is a traversable endomorphism of any traversable functor (T, δ) then it automatically satisfies the conditions of the identity morphism $\mathbf{1}_{(T,\delta)}$. Thus, it only remains to be proven that $\mathbf{1}_T$ is a traversable endomorphism of any traversable functor (T, δ).

This is shown through the following commutative diagram:



$\mathbf{1}_T$ is a traversable endomorphism of any traversable functor (T, δ), and is therefore the identity traversable morphism $\mathbf{1}_{(T,\delta)}$. □

Definition 4.6 (Category of Traversable Functors)**.** I define *the category of traversable functors* Tra as the category of which

- Objects are traversable functors (T, δ)
- Morphisms are traversable morphisms.
- Composition of morphisms is defined through composition of underlying natural transformations.
- The identity morphism of any object (T, δ) is defined as the natural transformation $\mathbf{1}_T$.

As the composition and identity of traversable morphisms have been proven to be valid, this is a valid category.

4.4 Cofree Traversable Functors

Definition 4.7 (Forgetful Functor $\mathsf{Tra} \to [\mathsf{Set}, \mathsf{Set}]$)**.** I define $\mathsf{For} : \mathsf{Tra} \to [\mathsf{Set}, \mathsf{Set}]$ as the forgetful functor formed by mapping each traversable functor (T, δ) to the underlying endofunctor T, and mapping each traversable morphism $\alpha : (T, \delta) \to (U, \epsilon)$ to the underlying natural transformation $\alpha' : T \to U$.

As any identity traversable morphism $\mathbf{1}_{(T,\delta)}$ directly corresponds to the identity natural transformation $\mathbf{1}_T$, and composition of traversable morphisms $f \circ g$ directly correspond to composition of natural transformations $f' \circ g'$, For is trivially a valid functor.

For any two functors $L : \mathscr{A} \to \mathscr{B}$ and $R : \mathscr{B} \to \mathscr{A}$, if there exists a *functor adjunction* $L \dashv R$ then L is called the left adjoint of R, and R is called the right adjoint of L. A functor adjunction $L \dashv R$ is a relationship between L and R such that there exists the following natural transformations:

$$\eta : \mathbf{1}_{\mathscr{A}} \to R \circ L \qquad \text{(unit)}$$
$$\epsilon : L \circ R \to \mathbf{1}_{\mathscr{B}} \qquad \text{(counit)}$$

where $\mathbf{1}_{\mathscr{A}}$ and $\mathbf{1}_{\mathscr{B}}$ are the identity endofunctors of categories $\mathscr{A}$ and $\mathscr{B}$, respectively.

Expressed componentwise, for all $X : \mathscr{A}$, $Y : \mathscr{B}$

$$\eta_X = X \to R(L(X))$$
$$\epsilon_Y = L(R(Y)) \to Y$$

These natural transformations are subject to the *triangle identities*:

$$\mathbf{1}_L = \epsilon L \circ L\eta$$
$$\mathbf{1}_R = R\epsilon \circ \eta R$$

Expressed componentwise, for all $X : \mathscr{A}$, $Y : \mathscr{B}$

$$\mathbf{1}_{L(X)} = \epsilon_{L(X)} \circ L(\eta_X)$$
$$\mathbf{1}_{R(Y)} = R(\epsilon_Y) \circ \eta_{R(Y)}$$

A *cofree functor* is defined as the right adjoint of a forgetful functor. As For is a forgetful functor $\mathsf{Tra} \to [\mathsf{Set}, \mathsf{Set}]$, its right adjoint, if it exists, is the *cofree functor* which maps any functor F to the *cofree traversable functor* on F; i.e. *the cofree traversable functor functor.*

Definition 4.8 (Cofree Traversable Functors)**.** I define the *cofree traversable functor functor* Cot as the right adjoint of For. It is therefore a *cofree functor*. For any functor F, I call the traversable functor $\mathsf{Cot}(F)$ the *cofree traversable functor on* F.

Adjoints are unique up to isomorphism, [1] and thus Cot may be characterized through the unit and counit of the adjunction:

$$\eta : \mathbf{1}_{\mathsf{Tra}} \to \mathsf{Cot} \circ \mathsf{For}$$
$$\epsilon : \mathsf{For} \circ \mathsf{Cot} \to \mathbf{1}_{[\mathsf{Set},\mathsf{Set}]}$$

I.e. for all traversable functors $(T, \delta) : \mathsf{Tra}$, there exists a traversable morphism

$$\eta_{(T,\delta)} : (T, \delta) \to \mathsf{Cot}(T)$$

natural in (T, δ), and for all functors $F : [\mathsf{Set}, \mathsf{Set}]$, there exists a natural transformation

$$\epsilon_F : \mathsf{For}(\mathsf{Cot}(F)) \to F$$

natural in F.

The triangle identities are as follows:

For all $(T, \delta) : \mathsf{Tra}$, $F : [\mathsf{Set}, \mathsf{Set}]$

$$\mathbf{1}_T = \epsilon_T \circ \mathsf{For}(\eta_{(T,\delta)})$$
$$\mathbf{1}_{\mathsf{Cot}(F)} = \mathsf{Cot}(\epsilon_F) \circ \eta_{\mathsf{Cot}(F)}$$

Note that no proof has yet been given that Cot exists. This definition serves only as the basis of the type-theoretical model presented in Section 5, which is then implemented in the form of the representational encoding, presented in Section 6. The validity of the representational encoding, as proven in Section 7, serves as the proof that Cot exists.

5 Description of the Cofree Traversable Functor Functor

Using the category-theoretical definition of the cofree traversable functor functor, a description of it may be derived in type-theoretical terms.

This description defines an interface, consisting of an abstract type, together with a set of operations which follow a particular set of laws. An implementation of this interface is required to provide a definition of the

type and implementations of the operations such that the laws are satisfied. The interface is defined such that an implementation of it corresponds to a constructive definition of a functor $[\mathsf{Set}, \mathsf{Set}] \to \mathsf{Tra}$ possessing the unique characteristics of Cot, the cofree traversable functor functor. Such an implementation is therefore called an *encoding* of the cofree traversable functor functor.

The abstract type of the interface, called Cotra, has the following kind:

$$\mathsf{Cotra} : (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set} \to \mathsf{Set}$$

Cotra corresponds to the mapping of functors and sets that $\mathsf{For} \circ \mathsf{Cot}$ performs, such that for all $F : [\mathsf{Set}, \mathsf{Set}]$, $A : \mathsf{Set}$

$$\mathsf{Cotra}(F)(A) = \mathsf{For}(\mathsf{Cot}(F))(A)$$

The interface requires Cotra to be a functor $[\mathsf{Set}, \mathsf{Set}] \to [\mathsf{Set}, \mathsf{Set}]$. Cotra must therefore map functors F to functors $\mathsf{Cotra}(F)$, and natural transformations $\alpha : F \to G$ to natural transformations $\mathsf{Cotra}(\alpha) : \mathsf{Cotra}(F) \to \mathsf{Cotra}(G)$. Mapping of functors is represented by the fact that for each functor F, the interface defines a mapping operation $\mathtt{map}^{\mathsf{Cotra}(F)}$ as per Section 2.3, subject to its laws.

Mapping of natural transformations is represented through the polymorphic operation `hoist`, which is part of the interface. `hoist` is defined as follows:

$$\begin{aligned}\mathtt{hoist} : \forall(F, G : [\mathsf{Set}, \mathsf{Set}]).\ & [\mathsf{Set}, \mathsf{Set}](F, G) \\ & \to \\ & [\mathsf{Set}, \mathsf{Set}](\mathsf{Cotra}(F), \mathsf{Cotra}(G))\end{aligned}$$

Where $[\mathsf{Set}, \mathsf{Set}](F, G)$ is represented through the type of polymorphic functions $\forall x.\ F(x) \to G(x)$ natural in x, as given in Section 2.3.

As the mapping operation of a (categorical) functor, `hoist` needs to preserve identity morphisms and composition of morphisms. This is represented through the following laws:

- *Preservation of identity*

  For all $F : [\mathsf{Set}, \mathsf{Set}], A : \mathsf{Set}$

  $$(\mathtt{hoist}\ \mathsf{id})_A = \mathsf{id}_{\mathsf{Cotra}(F)(A)}$$

- *Preservation of Composition*

  For all $F, G, H : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : [\mathsf{Set}, \mathsf{Set}](G, H)$, $\beta : [\mathsf{Set}, \mathsf{Set}](F, G), A : \mathsf{Set}$

  $$(\mathtt{hoist}\ \alpha)_A \circ (\mathtt{hoist}\ \beta)_A = \mathtt{hoist}(\alpha \circ \beta)_A$$

Associated with the interface is a family of traversals ω such that for each functor F, ${}^F\omega$ is a traversal of $\mathsf{Cotra}(F)$. Cotra and ω together form the definition of Cot, through $\mathsf{Cot}(F) = (\mathsf{Cotra}(F), {}^F\omega)$. $\mathtt{hoist}$ is used to define the mapping of natural transformations to traversable morphisms for this definition of Cot, such that for all $F, G : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : [\mathsf{Set}, \mathsf{Set}](F, G)$

$$\mathsf{Cot}(\alpha)' = \mathsf{Cotra}(\alpha) = \mathtt{hoist}\ \alpha$$

As such, $\mathtt{hoist}\ \alpha$ is required to be a traversable morphism for any α. This requirement is represented through the $\mathtt{hoist}$ *traversable morphism law* defined as follows:

Given $F, G : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : [\mathsf{Set}, \mathsf{Set}](F, G)$, $H : \mathsf{App}$, $A, B : \mathsf{Set}$, $f : A \to H(B)$

$$\begin{aligned} & {}^{G\omega}\mathtt{traverse}^H_{A,B}\ f \circ (\mathtt{hoist}\ \alpha)_A \\ = \ & H((\mathtt{hoist}\ \alpha)_B) \circ {}^{F\omega}\mathtt{traverse}^H_{A,B}\ f \end{aligned}$$

The *unit* and *counit* admitted by the $\mathsf{For} \dashv \mathsf{Cot}$ adjunction are represented through the polymorphic operations:

$$\begin{aligned} & \mathtt{unit} : \forall((T, \delta) : \mathsf{Tra})\ x.\ T(x) \to \mathsf{Cotra}(T)(x) \\ & \mathtt{counit} : \forall(F : [\mathsf{Set}, \mathsf{Set}])\ x.\ \mathsf{Cotra}(F)(x) \to F(x) \end{aligned}$$

such that for all $(T, \delta) : \mathsf{Tra}$, $A : \mathsf{Set}$

$$(\eta'_{(T,\delta)})_A = \mathtt{unit}_{(T,\delta),A}$$

and for all $F : [\mathsf{Set}, \mathsf{Set}]$, $A : \mathsf{Set}$

$$(\epsilon_F)_A = \mathtt{counit}_{F,A}$$

This yields the obvious laws that $\mathtt{unit}_F$ is a natural transformation for any F, and $\mathtt{counit}_{(T,\delta)}$ is a traversable morphism $(T, \delta) \to (\mathsf{Cotra}(T), {}^T\omega)$ for any (T, δ).

The *unit traversable morphism law* is stated as follows:

For all $(T, \delta) : \mathsf{Tra}$, $H : \mathsf{App}$, $A, B : \mathsf{Set}$, $f : A \to H(B)$,

$$\begin{aligned} & H(\mathtt{unit}_{(T,\delta),B}) \circ {}^{\delta}\mathtt{traverse}^H_{A,B}\ f \\ = \ & {}^{T\omega}\mathtt{traverse}^H_{A,B}\ f \circ \mathtt{unit}_{(T,\delta),A} \end{aligned}$$

The *counit natural transformation law* is stated as follows:

For all $F : [\mathsf{Set}, \mathsf{Set}], A, B : \mathsf{Set}, f : A \to B$

$$\begin{aligned} & \mathtt{counit}_{F,B} \circ \mathsf{Cotra}(F)(f) \\ = \; & F(f) \circ \mathtt{counit}_{F,A} \end{aligned}$$

The laws for `unit` and `counit` must also reflect that η and ϵ are natural transformations $\mathbf{1}_{\mathsf{Tra}} \to \mathsf{Cot} \circ \mathsf{For}$ and $\mathsf{For} \circ \mathsf{Cot} \to \mathbf{1}_{[\mathsf{Set},\mathsf{Set}]}$ respectively.

The naturality condition for η is as follows:

For all $(T, \delta), (U, \varepsilon) : \mathsf{Tra}, \alpha : (T, \delta) \to (U, \varepsilon)$

$$\eta_{(U,\varepsilon)} \circ \alpha = \mathsf{Cot}(\mathsf{For}(\alpha)) \circ \eta_{(T,\delta)}$$

This gives rise to the *unit naturality law*:

For all $(T, \delta), (U, \varepsilon) : \mathsf{Tra}$, $\alpha : (T, \delta) \to (U, \varepsilon)$, $A : \mathsf{Set}$

$$\begin{aligned} & \mathtt{unit}_{(U,\varepsilon),A} \circ \alpha_A \\ = \; & (\mathtt{hoist}\ \alpha)_A \circ \mathtt{unit}_{(T,\delta),A} \end{aligned}$$

The naturality condition for ϵ is as follows:

For all $F, G : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : F \to G$

$$\epsilon_G \circ \mathsf{For}(\mathsf{Cot}(\alpha)) = \alpha \circ \epsilon_F$$

This gives rise to the *counit naturality law*:

For all $F, G : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : [\mathsf{Set}, \mathsf{Set}](F, G)$, $A : \mathsf{Set}$

$$\mathtt{counit}_{G,A} \circ (\mathtt{hoist}\ \alpha)_A = \alpha_A \circ \mathtt{counit}_{F,A}$$

.

Finally, the triangle identities must be represented.

$$\begin{aligned} \mathbf{1}_{\mathsf{For}} &= \epsilon \mathsf{For} \circ \mathsf{For} \eta \\ \mathbf{1}_{\mathsf{Cot}} &= \mathsf{Cot} \epsilon \circ \eta \mathsf{Cot} \end{aligned}$$

These will be referred to as the *left* and *right* triangle identity, respectively.

The left triangle identity holds if the identity holds for all components $(T, \delta) : \mathsf{Tra}$, $A : \mathsf{Set}$

$$\begin{aligned} & ((\mathbf{1}_{\mathsf{For}})_{(T,\delta)})_A = ((\epsilon \mathsf{For} \circ \mathsf{For} \eta)_{(T,\delta)})_A \\ & \mathbf{1}_{\mathsf{For}(T,\delta)(A)} = \epsilon_{\mathsf{For}(T,\delta)(A)} \circ (\mathsf{For}(\eta_{(T,\delta)}))_A \\ & \mathbf{1}_{T(A)} = (\epsilon_T)_A \circ (\eta_{(T,\delta)})'_A \end{aligned}$$

This gives rise to the *left triangle identity law*:
For all $(T, \delta) : \mathsf{Tra}$, $A : \mathsf{Set}$

$$\mathsf{id}_{T(A)} = \mathtt{counit}_{T,A} \circ \mathtt{unit}_{(T,\delta),A}$$

The right triangle identity holds if the identity holds for all components $F : [\mathsf{Set}, \mathsf{Set}]$

$$(\mathbf{1}_{\mathsf{Cot}})_F = (\mathsf{Cot}\epsilon \circ \eta\mathsf{Cot})_F$$
$$\mathbf{1}_{\mathsf{Cot}(F)} = \mathsf{Cot}(\epsilon_F) \circ \eta_{\mathsf{Cot}(F)}$$

This holds iff.

$$(\mathbf{1}_{\mathsf{Cot}(F)})' = (\mathsf{Cot}(\epsilon_F) \circ \eta_{\mathsf{Cot}(F)})'$$
$$\mathbf{1}_{\mathsf{Cotra}(F)} = \mathsf{Cotra}(\epsilon_F) \circ (\eta_{(\mathsf{Cotra}(F),{}^F\omega)})'$$

This, in turn, holds if the identity holds for all components $A : \mathsf{Set}$

$$\mathbf{1}_{\mathsf{Cotra}(F)(A)} = \mathsf{Cotra}(\epsilon_F)_A \circ (\eta_{(\mathsf{Cotra}(F),{}^F\omega)})'_A$$

This gives rise to the *right triangle identity law*:
For all $F : [\mathsf{Set}, \mathsf{Set}]$, $A : \mathsf{Set}$

$$\mathsf{id}_{\mathsf{Cotra}(F)(A)} = (\mathtt{hoist}\ \mathtt{counit}_F)_A \circ \mathtt{unit}_{(\mathsf{Cotra}(F),{}^F\omega),A}$$

6 The Representational Encoding

The *Representational Encoding* of the cofree traversable functor functor is based on the concept of separating the elements of an encapsulated container from its *shape*. By representing the elements separately, in the form of an ordered list, these may traversed even if the encapsulated container is not traversable. The *shape* allows for reconstructing the encapsulated container out of the elements, even if they have been modified by a traversal.

This section concerns the concepts related to the representational encoding, such as shapes, their relation to traversable functors, and the definition of Cot under the representational encoding.

6.1 Shapes, the Representation Theorem, and Characterizations

I define a *shape* with arity $n : \mathbb{N}$ of a functor F to be a value that allows for inserting n different values of any one type in a particular order into some

predefined existing context of F. A shape of F with arity $n : \mathbb{N}$ is a member of any of the following types:

- The type of polymorphic functions $\forall x.\ x \to x \to \cdots \to x \to F(x)$ with exactly n arguments, natural in x such that for any member s

  $$s\ (f\ x_1)\ (f\ x_2)\ \ldots\ (f\ x_n) = F(f)\ (s\ x_1\ x_2\ \ldots\ x_n)$$

  Insertion of elements $x_1 \ldots x_n$ into such a shape is defined as simply providing these elements as arguments in the chosen order.

- The type of natural transformations $\mathsf{Vec}\ n \to F$, where $\mathsf{Vec}\ n$ is the functor corresponding to a fixed-size vector with length n. Insertion of elements $x_1 \ldots x_n$ into such a shape is defined as applying the natural transformation to the fixed-size vector containing these elements in the chosen order.

- The type $F(\mathsf{Fin}\ n)$, where $\mathsf{Fin}\ n$ is defined as a type with exactly n inhabitants, ordered such that $\texttt{fin}\ i : \mathsf{Fin}\ n$ represents the i:th unique inhabitant. Insertion of elements $x_1 \ldots x_n$ into such a shape s is defined as $F(f)\ s$, where $f\ (\texttt{fin}\ i) = x_i$. Thus, each value of $fn : \mathsf{Fin}\ n$ is replaced with the x whose chosen ordering corresponds to fn.

These types are *isomorphic*, meaning that any one of these types has an *isomorphism* – an invertible function – to any other. Isomorphic types are equivalent in the sense that values may be converted between these types without any loss of information.

For the purposes of eliminating ambiguity, the pseudotype $\mathsf{Shape}\ F\ n$ will be used to encompass all shapes of F with arity n, equipped with isomorphisms to and from each type defined above. *Insertion of elements in a particular order*, as defined for each type above, corresponds to a unique isomorphism:

$$\texttt{insert}\ : \mathsf{Shape}\ F\ n \to (\forall x.\ x \to x \to \cdots \to x \to F(x))$$

Two shapes s_1, s_2 are considered equal if $\texttt{insert}\ \ s_1 = \texttt{insert}\ \ s_2$.

Shapes are a generalization of *make-functions*, which were introduced by Bird et al. Using these, Bird et al. present and prove the *Representation Theorem* for traversable functors [2]: a powerful result which the representational encoding is based upon.

Reformulated in terms of shapes, the representation theorem is as follows:

Theorem 6.1 (Representation Theorem)**.** Given a traversable functor (T, δ) and a traversable container $t : T(A)$, there exists a unique triple

$$(n : \mathbb{N},\ s : \mathsf{Shape}\ F\ n,\ l : (\mathsf{Vec}\ n)(A))$$

such that

$$t = \texttt{insert}\ s\ l_1\ \ldots\ l_n \qquad \textit{(Reconstruction formula)}$$

where l_i is the i:th element of l; and for all applicative functors $F : \mathsf{App}$, types $X : \mathsf{Set}$, elements $x_1, \ldots, x_n : X$, and functions $X \to F(Y)$

$$\begin{aligned} & {}^{\delta}\texttt{traverse}^{F}_{X,Y}\ f\ (\texttt{insert}\ s\ x_1\ \ldots\ x_n) \\ = \ & \texttt{pure}\ (\texttt{insert}\ s) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n \qquad \textit{(Strong Construction formula)} \end{aligned}$$

I call the triple (n, s, l) the *characterization* of t under traversal δ. In particular, I call s the *characterizing shape* of t under traversal δ.

The *reconstruction formula* and the *strong construction formula* imply what is known as the *weak construction formula*:

Let (n, s, l) be the characterization of $t : T(A)$ under δ. Then:

$$\begin{aligned} & {}^{\delta}\texttt{traverse}^{F}_{X,Y}\ f\ t \\ = \ & \texttt{pure}\ (\texttt{insert}\ s) \circledast f\ l_1 \circledast \ldots \circledast f\ l_n \qquad \textit{(Weak Construction formula)} \end{aligned}$$

6.2 Calculating Characterizations

Given a traversable functor (T, δ), and a value $t : T(A)$, it is possible to calculate the characterization of t under δ through the use of an applicative functor, defined in Haskell as follows: [17]

```
data Batch a b c = P c | Batch a b (b -> c) :*: a
infixl 1 :*: -- :*: infixes to the left

instance Functor (Batch a b) where
  fmap f (P c) = P (f c)
  fmap f (r :*: a) = fmap (f .) r :*: a

instance Applicative (Batch a b) where
  pure = P
  u <*> P a = fmap ($ a) u
  u <*> (v :*: a) = ((.) <$> u <*> v) :*: a

batch :: a -> Batch a b b
batch a = P id :*: a

build :: Traversable t => t a -> Batch a b (t b)
build = traverse batch
```

A value of $\mathsf{Batch}(A)(S)(T)$ represents an ordered list of values of type A, together with a function that takes as many values of type S and produces a value of type T. The applicative instance corresponds to the following definition of $\circledast$ for $\mathsf{Batch}(A)(S)(T)$:

For all $X, Y : \mathsf{Set}$, $(\mathtt{P}\ f\ :\!*\!:\ l_1\ :\!*\!:\ \ldots\ :\!*\!:\ l_n) : \mathsf{Batch}(A)(S)(X \to Y)$, $(\mathtt{P}\ g\ :\!*\!:\ r_1\ :\!*\!:\ \ldots\ :\!*\!:\ r_m) : \mathsf{Batch}(A)(S)(X)$

$$\begin{aligned} & (\mathtt{P}\ f\ :\!*\!:\ l_1\ :\!*\!:\ \ldots\ :\!*\!:\ l_n) \\ & \circledast^{\mathsf{Batch}(A)(S)}\ (\mathtt{P}\ g\ :\!*\!:\ r_1\ :\!*\!:\ \ldots\ :\!*\!:\ r_m) \\ = \ & \mathtt{P}\ (\lambda x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ (f\ x_1\ \ldots\ x_n)\ (g\ y_1\ \ldots\ y_m)) \\ & :\!*\!:\ l_1\ :\!*\!:\ \ldots\ :\!*\!:\ l_n\ :\!*\!:\ r_1\ :\!*\!:\ \ldots\ :\!*\!:\ r_m \end{aligned}$$

As shown by Bird et al., [2] for any $t : T(A)$ with an associated traversal δ of T

$$\mathtt{build}\ t : \forall x.\ \mathsf{Batch}(A)(x)(T(x))$$

must be equal to

$$\mathtt{P}\ \mathtt{make}_x\ :\!*\!:\ a_1\ :\!*\!:\ a_2\ :\!*\!:\ \ldots\ :\!*\!:\ a_n : \forall x.\ \mathsf{Batch}(A)(x)(T(X))$$

for some $a_1, ..., a_n : A$, $\mathtt{make} : \forall x.\ x \to \cdots \to x \to T(x)$, where $\mathtt{make}$ is natural in x. $\mathtt{make}$ is therefore a shape $s : \mathsf{Shape}\ T\ n$; in fact, (n, s, l) must be the characterization of t under δ, where $l_i = a_i$.

Deriving this triple constructively may be done by first calculating n. This is done by instantiating some arbitrary type for $\mathtt{build}\ t$, and retrieving the arity by calculating the number of elements. The exact elements of $\mathtt{build}\ t$ is the same no matter what type it is instantiated with, and thus, the number of elements must be n. For proof, see Appendix C.

Next, $\mathtt{build}\ t$ is instantiated with the type $\mathsf{Fin}\ n$, resulting in:

$$\mathtt{P}\ \mathtt{make}_{(\mathsf{Fin}\ n)}\ :\!*\!:\ a_1\ :\!*\!:\ a_2\ :\!*\!:\ \ldots\ :\!*\!:\ a_n.$$

The vector l is constructed through $l_i = a_i$. The following is also constructed:

$$s = \mathtt{make}_{(\mathsf{Fin}\ n)}\ (\mathtt{fin}\ 1)\ (\mathtt{fin}\ 2)\ \ldots\ (\mathtt{fin}\ n)$$

s is a member of the type $T(\mathsf{Fin}\ n)$, and is therefore a shape; in fact, it is equal to the characterizing shape $\mathtt{make}$, as is proven below:

Given $x_1, \ldots, x_n : X$

$$\mathtt{insert}\ s\ x_1\ \ldots\ x_n = \mathtt{insert}\ \mathtt{make}\ x_1 \ldots x_n$$

Proof

$$
\begin{aligned}
& \texttt{insert}\ s\ x_1\ \ldots\ x_n \\
= \ & T(f)\ s \\
& \text{where } f\ (\texttt{fin}\ i) = x_i \\
= \ & \text{(Definition of } s\text{)} \\
& T(f)\ (\texttt{make}_{(\mathsf{Fin}\ n)}\ (\texttt{fin}\ 1)\ \ldots\ (\texttt{fin}\ n)) \\
= \ & \text{(naturality of } \texttt{make}\text{)} \\
& \texttt{make}_X\ (f\ (\texttt{fin}\ 1))\ \ldots\ (f\ (\texttt{fin}\ n)) \\
= \ & \texttt{make}_X\ x_1\ \ldots\ x_n \\
= \ & \texttt{insert}\ \texttt{make}\ x_1\ \ldots\ x_n
\end{aligned}
$$

thus $s = \texttt{make}$. □

Therefore, (n, s, l) is the characterization of t.

6.3 Description of the Representational Encoding

The associated data type of the representational encoding is defined as the following dependent type:

$$\mathsf{Cotra}(F)(A) = \Sigma_{n:\mathbb{N}}\ F(\mathsf{Fin}\ n) \times (\mathsf{Vec}\ n)(A)$$

Each inhabitant of this type is a triple

$$(n : \mathbb{N},\ s : \mathsf{Shape}\ F\ n,\ l : (\mathsf{Vec}\ n)(A))$$

and these are therefore called *representational triples*. Note however that in the context of this encoding, shapes are represented as values of type $F(\mathsf{Fin}\ n)$, and will be treated as such.

Any *characterization* of a traversable container is a representational triple: if F is a traversable functor together with any traversal δ, the characterization of any $t : F(A)$ under δ is a unique member of $\mathsf{Cotra}(F)(A)$.

$\mathsf{Cotra}(F)$ is a functor: the mapping operation is defined as mapping over the elements as given by the representational triple.

$$\texttt{map}^{\mathsf{Cotra}(F)}\ f\ (n, s, l) = (n, s, \texttt{map}^{(\mathsf{Vec}\ n)}\ f\ l)$$

where

$$\texttt{map}^{(\mathsf{Vec}\ n)}\ f\ [l_1, \ldots, l_n] = [f\ l_1, \ldots, f\ l_n]$$

The behaviour of the traversal ${}^{F}\omega$ defined for $\mathsf{Cotra}(F)$ is independent of the parametrized functor, and is defined as traversing the elements of l of the representational triple (n, s, l) in order.

For all $H : \mathsf{App}$, $A, B : \mathsf{Set}$, $f : A \to H(B)$, $F : [\mathsf{Set}, \mathsf{Set}]$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$ {}^{F}\omega\texttt{traverse}^{H}_{A,B}\ f\ (n, s, l) = \texttt{pure}\ (\lambda k.\ (n, s, k)) \circledast ({}^{n}\sigma\texttt{traverse}^{H}_{A,B}\ f\ l) $$

where ${}^{n}\sigma$ is the traversal of $\mathsf{Vec}\ n$ which traverses its elements in order:

$$ {}^{n}\sigma\texttt{traverse}\ f\ l = \texttt{pure}\ (\lambda x_1\ \ldots\ x_n.\ [x_1, \ldots, x_n]) \circledast f\ l_1 \circledast \ldots \circledast f\ l_n $$

Thus, Cot has been defined: $\mathsf{Cot}(F) = (\mathsf{Cotra}(F), {}^{F}\omega)$

The mapping of natural transformations – `hoist` – is defined through applying the natural transformation provided to the shape present in the representational triple.

$$ \texttt{hoist}\ \alpha\ (n, s, l) = (n, \alpha_{(Fin\ n)}\ s, l) $$

The proof that this definition of Cot is a valid functor $[\mathsf{Set}, \mathsf{Set}] \to \mathsf{Tra}$ – that it satisfies the corresponding laws – is given in Section 7.1.

The *unit* is defined such that given a traversable functor (T, δ), the morphism $\texttt{unit}_{(T,\delta),A}$ is defined as the function returning the characterization (n, s, l) of the parameter $t : T(A)$ under δ. This function may be constructed using the method described in Section 6.2.

$$ \texttt{unit}_{(T,\delta),A}\ t = (n, s, l) $$
where (n, s, l) is the characterization of t under traversal δ.

The *counit* is defined such that given a representational triple

$$ (n, s, l) : \mathsf{Cotra}(F)(A) $$

$\texttt{counit}_{F,A}\ (n, s, l)$ inserts the elements of l into the shape, in order.

$$ \texttt{counit}_{F,A}\ (n, s, l) = \texttt{insert}\ s\ l_1\ \ldots\ l_n $$

This operation may be constructed such that given $(n, s, l) : \mathsf{Cotra}(F)(A)$, the resulting $F(A)$ is defined as $F(\texttt{index}\ l)\ s$, where

$$ \begin{aligned} &\texttt{index}\ : \forall(n : \mathbb{N})\ A.\ (\mathsf{Vec}\ n)(A) \to \mathsf{Fin}\ n \to A \\ &\texttt{index}\ l\ (\texttt{fin}\ i) = l_i \end{aligned} $$

As shown below, this is equivalent to `insert` $s\ l_1\ \ldots\ l_n$

$$\begin{aligned} & \texttt{insert}\ s\ l_1\ \ldots\ l_n \\ = \ & F(f)\ s \\ & \text{where } f\ (\texttt{fin}\ i) = l_i. \\ = \ & F(\texttt{index}\ l)\ s \end{aligned}$$

See Section 7.2 for proof that `unit` and `counit` satisfy the *unit traversable morphism law*, *counit natural transformation law* and the *co/unit naturality laws*. The proofs for the triangle identity laws are located in Section 7.3.

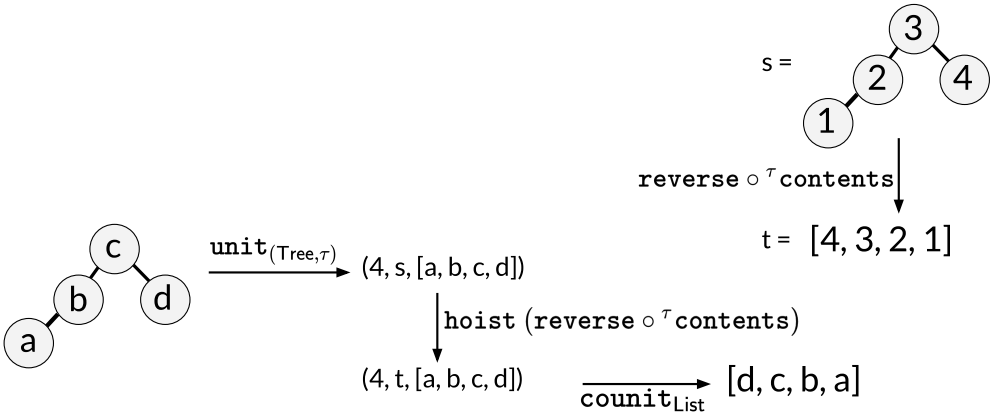


Figure 2: Demonstration of how `unit`, `counit`, and `hoist` interact. Each item i of shapes s and t represent values `fin` i : Fin 4. τ is the in-order traversal for binary trees, as defined in Section 2.5.

7 Proof of the Validity of the Representational Encoding

7.1 Functor

Given any functor F, $\mathsf{Cot}(F) = (\mathsf{Cotra}(F), {}^{F}\omega)$ is a traversable functor.

Proof The proofs that $\mathsf{Cotra}(F)$ is a valid functor and that ${}^F\omega$ satisfies the *Consistency* law are trivial, and will be omitted from this section. If the reader is interested in these proofs, see Appendix D.

Jaskelioff and Rypacek [13] define a valid, "canonical" traversal for any extension of a *finitary container*. A finitary container is given by:

(a) A type S of *shapes*; unrelated to shapes in the context of the representational encoding.

(b) An *arity* $\mathsf{ar} : S \to \mathbb{N}$

The *extension* of any container (S, ar), denoted $\mathsf{Ext}(S, \mathsf{ar})$, is a functor which maps each type X as follows:

$$\mathsf{Ext}(S, \mathsf{ar})(X) = \Sigma_{s:S}\ (\mathsf{Vec}\ (\mathsf{ar}\ s))(X)$$

that is, each type X is mapped to the type of dependent pairs (s, f), where $s : S$, $f : (\mathsf{Vec}\ (\mathsf{ar}\ s))(X)$. Mapping of functions is defined simply as mapping these over the vector.

For any F, the functor $\mathsf{Cotra}(F)$ directly corresponds to the extension of the following finitary container:

$$\begin{aligned} S &= \Sigma_{n:\mathbb{N}}\ F(\mathsf{Fin}\ n) \\ \mathsf{ar}\ (n, f) &= n \end{aligned}$$

Which may be observed from the fact that

$$\begin{aligned} & \mathsf{Ext}(s, \mathsf{ar})(X) \\ = \ & \Sigma_{s:S}\ (\mathsf{Vec}\ (\mathsf{ar}\ s))(X) \\ = \ & \Sigma_{(n,t):(\Sigma_{n:\mathbb{N}}\ F(\mathsf{Fin}\ n))}\ (\mathsf{Vec}\ n)(X) \end{aligned}$$

is trivially isomorphic to:

$$\mathsf{Cotra}(F)(X) = \Sigma_{n:\mathbb{N}}\ F(\mathsf{Fin}\ n) \times (\mathsf{Vec}\ n)(X)$$

The traversal ${}^F\omega$ corresponds to the "canonical" traversal defined for this finitary container as given by Jaskelioff and Rypacek, and is therefore valid. □

The representational encoding satisfies the `hoist` *traversable morphism law*: given any natural transformation $\alpha : F \to G$, `hoist` α is a traversable morphism $\mathsf{Cot}(F) \to \mathsf{Cot}(G)$.

Proof For all $A, B : \mathsf{Set}$, $H : \mathsf{App}$, $f : A \to HB$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$
\begin{aligned}
& {}^{G}\omega\mathtt{traverse}^{H}_{A,B}\ f\ ((\mathtt{hoist}\ \alpha)_A\ (n, s, l)) \\
= \ & \text{(Definition of } \mathtt{hoist}\text{)} \\
& {}^{G}\omega\mathtt{traverse}^{H}_{A,B}\ f\ (n, \alpha_{(\mathsf{Fin}\ n)}\ s, l) \\
= \ & \text{(Definition of } {}^{G}\omega\mathtt{traverse}\text{)} \\
& \mathtt{pure}\ (\lambda k.\ (n, \alpha_{(\mathsf{Fin}\ n)}\ s, k)) \circledast {}^{n}\sigma\mathtt{traverse}^{H}_{A,B}\ f\ l \\
= \ & \text{(Consistency law of applicative functors)} \\
& H(\lambda k.\ (n, \alpha_{(\mathsf{Fin}\ n)}\ s, k))({}^{n}\sigma\mathtt{traverse}^{H}_{A,B}\ f\ l) \\
= \ & \text{(Definition of } \mathtt{hoist}\text{)} \\
& H((\mathtt{hoist}\ \alpha)_B \circ (\lambda k.\ (n, s, k)))({}^{n}\sigma\mathtt{traverse}^{H}_{A,B}\ f\ l) \\
= \ & \text{(Preservation of Composition)} \\
& H((\mathtt{hoist}\ \alpha)_B)(H(\lambda k.\ (n, s, k))({}^{n}\sigma\mathtt{traverse}^{H}_{A,B}\ f\ l)) \\
= \ & \text{(Consistency law of applicative functors)} \\
& H((\mathtt{hoist}\ \alpha)_B)(\mathtt{pure}\ (\lambda k.\ (n, s, k)) \circledast ({}^{n}\sigma\mathtt{traverse}^{H}_{A,B}\ f\ l)) \\
= \ & \text{(Definition of } {}^{F}\omega\mathtt{traverse}\text{)} \\
= \ & H((\mathtt{hoist}\ \alpha)_B)({}^{F}\omega\mathtt{traverse}^{H}_{A,B}\ f\ (n, s, l))
\end{aligned}
$$

□

The *preservation of identity* and *preservation of composition* laws are satisfied.

Proof For all $F : [\mathsf{Set}, \mathsf{Set}]$, $A : \mathsf{Set}$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$
\begin{aligned}
& (\mathtt{hoist}\ \mathsf{id})_A\ (n, s, l) \\
= \ & \text{(Definition of } \mathtt{hoist}\text{)} \\
& (n, \mathsf{id}_{F(\mathsf{Fin}\ n)}\ s, l) \\
= \ & (n, s, l) \\
= \ & \mathsf{id}_{\mathsf{Cotra}(F)(A)}\ (n, s, l)
\end{aligned}
$$

For all $F, G, H : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : [\mathsf{Set}, \mathsf{Set}](G, H)$, $\beta : [\mathsf{Set}, \mathsf{Set}](F, G)$, $A : \mathsf{Set}$,

$(n, s, l) : \mathsf{Cotra}(F)(A)$

$$
\begin{aligned}
& (\texttt{hoist}\ \alpha)_A\,((\texttt{hoist}\ \beta)_A\ (n, s, l)) \\
= \ & \text{(Definition of } \texttt{hoist}\text{)} \\
& (\texttt{hoist}\ \alpha)_A\,\big(n, \beta_{(\mathsf{Fin}\ n)}\ s, l\big) \\
= \ & \text{(Definition of } \texttt{hoist}\text{)} \\
& \big(n, \alpha_{(\mathsf{Fin}\ n)}\ (\beta_{(\mathsf{Fin}\ n)}\ s), l\big) \\
= \ & \big(n, (\alpha \circ \beta)_{(\mathsf{Fin}\ n)}\ s, l\big) \\
= \ & \text{(Definition of } \texttt{hoist}\text{)} \\
& (\texttt{hoist}(\alpha \circ \beta))_A\ (n, s, l) \qquad \square
\end{aligned}
$$

Therefore, the definition of Cot under the representational encoding is a valid (categorical) functor $[\mathsf{Set}, \mathsf{Set}] \to \mathsf{Tra}$.

7.2 Naturality of Unit and Counit

Lemma 7.1. For all $F, G : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : F \to G$, $n : \mathbb{N}$, $s : F(\mathsf{Fin}\ n)$, $X : \mathsf{Set}$, $x_1, \ldots, x_n : X$

$$
\begin{aligned}
& \alpha_X\ (\texttt{insert}\ s\ x_1\ \ldots\ x_n) \\
= \ & \texttt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\ x_1\ \ldots\ x_n
\end{aligned}
$$

Proof

$$
\begin{aligned}
& \alpha_X\ (\texttt{insert}\ s\ x_1\ \ldots\ x_n) \\
= \ & \text{(Definition of } \texttt{insert}\ s\text{)} \\
& \alpha_X\ (F(f)\ s) \\
& \text{where } f\ (\texttt{fin}\ i) = x_i \\
= \ & \text{(Naturality of } \alpha\text{)} \\
& G(f)\ (\alpha_{(\mathsf{Fin}\ n)}\ s) \\
= \ & \text{(Definition of } \texttt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\text{)} \\
& \texttt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\ x_1\ \ldots\ x_n \qquad \square
\end{aligned}
$$

Lemma 7.2. For all $F : \mathsf{App}$, $X_1, \ldots, X_n : \mathsf{Set}$, $Y, Z : \mathsf{Set}$, values $u_1 : F(X_1), \ldots, u_n : F(X_n)$, $f : Y \to Z$, $g : X_1 \to \cdots \to X_n \to Y$

$$
\begin{aligned}
& \texttt{pure}\ f \circledast (\texttt{pure}\ g \circledast u_1 \circledast \ldots \circledast u_n) \\
= \ & \texttt{pure}\ (\lambda x_1\ \ldots\ x_n.\ f\ (g\ x_1\ \ldots\ x_n)) \circledast u_1 \circledast \ldots \circledast u_n
\end{aligned}
$$

Which, through the *Consistency* law of applicative functors, implies:

$$\begin{aligned} & H(f)\,(\texttt{pure}\ g \circledast u_1 \circledast \ldots \circledast u_n) \\ = \ & \texttt{pure}\ (\lambda x_1\ \ldots\ x_n.\ f\ \ (g\ x_1\ \ldots\ x_n)) \circledast u_1 \circledast \ldots \circledast u_n \end{aligned}$$

Proof Trivial consequence of the *flattening formula* of $\circledast$ as presented by Bird et al. [2] □

Lemma 7.3. For all $(T, \delta) : \mathsf{Tra}$, $A : \mathsf{Set}$, $t : T(A)$ such that (n, s, l) is the characterization of t under δ:

For all $X : \mathsf{Set}$, $x_1, \ldots, x_n : X$

$$(n, s, [x_1, \ldots, x_n])$$

is the characterization of $\texttt{insert}\ s\ x_1\ \ldots\ x_n$ under δ, and thus:

$$\texttt{unit}_{(T,\delta),X}\ (\texttt{insert}\ s\ x_1\ \ldots\ x_n) = (n, s, [x_1, \ldots, x_n])$$

Proof The reconstruction formula is trivially satisfied, and the strong construction formula has been established to hold as s is the characterizing shape of t. □

Using the representation theorem, it is possible to prove that the *unit traversable morphism law* is satisfied:

For all $(T, \delta) : \mathsf{Tra}$, $H : \mathsf{App}$, $A, B : \mathsf{Set}$, $f : A \to H(B)$, $t : T(A)$

$$\begin{aligned} & H(\texttt{unit}_{(T,\delta),B})({}^{\delta}\texttt{traverse}^{H}_{A,B}\ f\ t) \\ = \ & {}^{T_\omega}\texttt{traverse}^{H}_{A,B}\ f\ (\texttt{unit}_{(T,\delta),A}\ t) \end{aligned}$$

and thus $\texttt{unit}_{(T,\delta)}$ is a traversable morphism $(T, \delta) \to \mathsf{Cot}(T)$

Proof Let $(n, s, l) = (\mathtt{unit}_{(T,\delta),A})\ t$

$$
\begin{aligned}
& H(\mathtt{unit}_{(T,\delta),B})({}^{\delta}\mathtt{traverse}^{H}_{A,B}\ f\ t) \\
= \ & \text{(Weak construction formula)} \\
& H(\mathtt{unit}_{(T,\delta),B})(\mathtt{pure}\ (\mathtt{insert}\ s) \circledast f\ l_1 \circledast \ldots \circledast f\ l_n) \\
= \ & \text{(Lemma 7.2)} \\
& \mathtt{pure}\ (\lambda x_1\ \ldots\ x_n.\ \mathtt{unit}_{(T,\delta),B}\ (\mathtt{insert}\ s\ x_1\ \ldots\ x_n)) \circledast f\ l_1 \circledast \ldots \circledast f\ l_n \\
= \ & \text{(Lemma 7.3)} \\
& \mathtt{pure}\ (\lambda x_1\ \ldots\ x_n.\ (n, s, [x_1, \ldots, x_n])) \circledast f\ l_1 \circledast \ldots \circledast f\ l_n \\
= \ & \text{(Lemma 7.2)} \\
& H(\lambda k.\ (n, s, k))\ (\mathtt{pure}\ (\lambda x_1\ \ldots\ x_n.\ [x_1, \ldots, x_n]) \circledast f\ l_1 \circledast \ldots \circledast f\ l_n) \\
= \ & \text{(Definition of } {}^{n\sigma}\mathtt{traverse}) \\
& H(\lambda k.\ (n, s, k))({}^{n\sigma}\mathtt{traverse}^{H}_{A,B}\ f\ [l_1, \ldots, l_n]) \\
= \ & \text{(Consistency law of applicative functors)} \\
& \mathtt{pure}\ (\lambda k.\ (n, s, k)) \circledast ({}^{n\sigma}\mathtt{traverse}^{H}_{A,B}\ f\ [l_1, \ldots, l_n]) \\
= \ & \text{(Definition of } {}^{T\omega}\mathtt{traverse}) \\
& {}^{T\omega}\mathtt{traverse}^{H}_{A,B}\ f\ (n, s, l) \\
= \ & {}^{T\omega}\mathtt{traverse}^{H}_{A,B}\ f\ (\mathtt{unit}_{(T,\delta),A}\ t)
\end{aligned}
$$

□

The representation theorem may also be used to prove the *unit naturality law*, completing the proof that $\mathtt{unit}$ is a natural transformation $\mathbf{1}_{\mathsf{Tra}} \to \mathsf{Cot} \circ \mathsf{For}$.

For all $(T, \delta), (U, \varepsilon) : \mathsf{Tra}, \alpha : (T, \delta) \to (U, \varepsilon), A : \mathsf{Set}, t : T(A)$,

$$
\begin{aligned}
& \mathtt{unit}_{(U,\varepsilon),A}\ (\alpha_A\ t) \\
= \ & (\mathtt{hoist}\ \alpha)_A\ (\mathtt{unit}_{(T,\delta),A}\ t)
\end{aligned}
$$

Proof Let $(n, s, l) = \mathtt{unit}_{(T,\delta),A}\ t$

$$
\begin{aligned}
& (\mathtt{hoist}\ \alpha)_A\ (\mathtt{unit}_{(T,\delta),A}\ t) \\
= \ & (\mathtt{hoist}\ \alpha)_A\ (n, s, l) \\
= \ & \text{(Definition of } \mathtt{hoist}) \\
& (n, \alpha_{(\mathsf{Fin}\ n)}\ s, l)
\end{aligned}
$$

Thus, it needs to be proven that

$$\mathtt{unit}_{(U,\varepsilon),A}\ (\alpha_A\ t) = (n, \alpha_{(\mathsf{Fin}\ n)}\ s, l)$$

i.e. that $(n, \alpha_{(\mathsf{Fin}\ n)}\ s, l)$ is the characterization of $(\alpha_A\ t)$ under traversal ε. $\alpha_{(\mathsf{Fin}\ n)}\ s$ with l therefore needs to satisfy the reconstruction formula with respect to t, and $\alpha_{(\mathsf{Fin}\ n)}\ s$ needs to satisfy the strong construction formula with respect to ε.

It follows trivially from Lemma 7.1 that

$$\alpha_X\ t = \alpha_X\ (\mathtt{insert}\ s\ l_1\ \ldots\ l_n) = \mathtt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\ l_1\ \ldots\ l_n$$

and therefore $\alpha_{(\mathsf{Fin}\ n)}\ s$ with l satisfies the *reconstruction formula*.

For all $X, Y : \mathsf{Set}$, $H : \mathsf{App}$, $f : X \to H(Y)$, $x_1 \ldots x_n : X$

$$\begin{aligned}
& {}^{\varepsilon}\mathtt{traverse}^{H}_{X,Y}\ f\ (\mathtt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\ x_1\ \ldots\ x_n) \\
= \ & \text{(Lemma 7.1)} \\
& {}^{\varepsilon}\mathtt{traverse}^{H}_{X,Y}\ f\ (\alpha_X\ (\mathtt{insert}\ s\ x_1\ \ldots\ x_n)) \\
= \ & (\alpha \text{ is a traversable morphism}) \\
& H(\alpha_Y)({}^{\delta}\mathtt{traverse}^{H}_{X,Y}\ f\ (\mathtt{insert}\ s\ x_1\ \ldots\ x_n)) \\
= \ & \text{(Strong construction formula)} \\
& H(\alpha_Y)(\mathtt{pure}\ (\mathtt{insert}\ s) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n) \\
= \ & \text{(Lemma 7.2)} \\
& \mathtt{pure}\ (\lambda y_1\ \ldots\ y_n.\ \alpha_Y\ (\mathtt{insert}\ s\ y_1 \ldots\ y_n)) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n \\
= \ & \text{(Lemma 7.1)} \\
& \mathtt{pure}\ (\lambda y_1\ \ldots\ y_n.\ \mathtt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\ y_1 \ldots\ y_n)) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n \\
= \ & \mathtt{pure}\ (\mathtt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n
\end{aligned}$$

Therefore, $(\alpha_{(\mathsf{Fin}\ n)}\ s)$ satisfies the *strong construction formula*.

$(n, \alpha_{(\mathsf{Fin}\ n)}\ s, l)$ is therefore the characterization of t under ε, and as characterizations are unique:

$$\mathtt{unit}_{(U,\varepsilon),A}\ (\alpha_A\ t) = (n, \alpha_{(\mathsf{Fin}\ n)}\ s, l)$$

Therefore

$$\begin{aligned}
& \mathtt{unit}_{(U,\varepsilon),A}\ (\alpha_A\ t) \\
= \ & (\mathtt{hoist}\ \alpha)_A\ (\mathtt{unit}_{(T,\delta),A}\ t) \qquad \square
\end{aligned}$$

The required properties of the counit are simpler to prove.

For all $F : [\mathsf{Set}, \mathsf{Set}]$, $A, B : \mathsf{Set}$, $f : A \to B$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$\begin{aligned}
& \mathtt{counit}_{F,B}\ (\mathsf{Cotra}(F)(f)\ (n, s, l)) \\
= \ & F(f)\ (\mathtt{counit}_{F,A}\ (n, s, l))
\end{aligned}$$

and thus the *counit natural transformation law* is satisfied, and $\texttt{counit}_F$ is a natural transformation for any F.

Proof

$$\begin{aligned}
& \texttt{counit}_{F,B}\ (\mathsf{Cotra}(F)(f)\ (n,s,l)) \\
= & \ \text{(Definition of } \mathsf{Cotra}(F)(f)\text{)} \\
& \texttt{counit}_{F,B}\ (n,s,[f\ l_1,\ldots,f\ l_n]) \\
= & \ \text{(Definition of } \texttt{counit}_{F,B}\text{)} \\
& \texttt{insert}\ s\ (f\ l_1)\ \ldots\ (f\ l_n) \\
= & \ \text{(Naturality of } \texttt{insert}\ s\text{)} \\
& F(f)\ (\texttt{insert}\ s\ l_1\ \ldots\ l_n) \\
= & \ \text{Definition of } \texttt{counit}_{F,A}\text{)} \\
& F(f)\ (\texttt{counit}_{F,A}\ (n,s,l)) \qquad \square
\end{aligned}$$

`counit` satisfies the *counit naturality law*, thus completing the proof that it is a natural transformation $\mathsf{For} \circ \mathsf{Cot} \to \mathbf{1}_{[\mathsf{Set},\mathsf{Set}]}$.

For all $F, G : [\mathsf{Set}, \mathsf{Set}]$, $\alpha : F \to G$, $A : \mathsf{Set}$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$\begin{aligned}
& \texttt{counit}_{G,A}\ ((\texttt{hoist}\ \alpha)_A\ (n,s,l)) \\
= & \ \alpha_A\ (\texttt{counit}_{F,A}\ (n,s,l))
\end{aligned}$$

Proof

$$\begin{aligned}
& \texttt{counit}_{G,A}\ ((\texttt{hoist}\ \alpha)_A\ (n,s,l)) \\
= & \ \text{(Definition of } \texttt{hoist}\text{)} \\
& \texttt{counit}_{G,A}\ (n, \alpha_{(\mathsf{Fin}\ n)}\ s, l) \\
= & \ \text{(Definition of } \texttt{counit}_{G,A}\text{)} \\
& \texttt{insert}\ (\alpha_{(\mathsf{Fin}\ n)}\ s)\ l_1\ \ldots\ l_n \\
= & \ \text{(Lemma 7.1)} \\
& \alpha_A\ (\texttt{insert}\ s\ l_1\ \ldots\ l_n) \\
= & \ \text{Definition of } \texttt{counit}_{F,A}\text{)} \\
& \alpha_A\ (\texttt{counit}_{F,A}\ (n,s,l)) \qquad \square
\end{aligned}$$

7.3 Triangle Identities

The *left triangle identity law* may easily be proven using the representation theorem.

For all $(T, \delta) : \mathsf{Tra}$, $A : \mathsf{Set}$

$$\mathsf{id}_{T(A)} = \mathtt{counit}_{T,A} \circ \mathtt{unit}_{(T,\delta),A}$$

holds.

Proof For all elements $t : T(A)$

$$\mathsf{id}_{T(A)}\ t = \mathtt{counit}_{T,A}\ (\mathtt{unit}_{(T,\delta),A}\ t)$$

This may be proven as follows:

Let $(n, s, l) = \mathtt{unit}_{(T,\delta),A}\ t$

$$\begin{array}{ll} & \mathtt{counit}_{T,A}\ (\mathtt{unit}_{(T,\delta),A}\ t) \\ = & \mathtt{counit}_{T,A}\ (n, s, l) \\ = & \mathtt{insert}\ s\ l_1\ \ldots\ l_n \\ = & \text{(Reconstruction formula)} \\ & t \\ = & \mathsf{id}_{T(A)}\ t \end{array}$$ □

The right triangle identity follows from the fact that the characterization of any representational triple $(n, s, l) : \mathsf{Cotra}(F)(A)$ under ${}^{F}\omega$ is the representational triple

$$(n, (n, s, \mathtt{self}^n), l) : \mathsf{Cotra}(\mathsf{Cotra}(F))(A)$$

where

$$\begin{array}{l} \mathtt{self}^n : (\mathsf{Vec}\ n)(\mathsf{Fin}\ n) \\ \mathtt{self}^n_i = \mathtt{fin}\ i \end{array}$$

This will be proven through the use of the following lemma:

Lemma 7.4. For all $F : [\mathsf{Set}, \mathsf{Set}]$, $n : \mathbb{N}$, $s : F(\mathsf{Fin}\ n)$, $X : \mathsf{Set}$, $x_1 \ldots x_n : X$

$$\begin{array}{ll} & \mathtt{insert}\ (n, s, \mathtt{self}^n)\ x_1\ \ldots\ x_n \\ = & (n, s, [x_1, \ldots, x_n]) \end{array}$$

Proof

$$
\begin{aligned}
& \texttt{insert}\ (n, s, \texttt{self}^n)\ x_1\ \ldots\ x_n \\
= & \ (\text{Definition of } \texttt{insert}\ (n, s, \texttt{self}^n)) \\
& \mathsf{Cotra}(F)(f)\ (n, s, \texttt{self}^n) \\
& \text{where } f\ (\texttt{fin}\ i) = x_i \\
= & \ (\text{Definition of } \mathsf{Cotra}(F)(f)) \\
& (n, s, (\mathsf{Vec}\ n)(f)\ \texttt{self}^n) \\
= & \ (\text{Definition of } (\mathsf{Vec}\ n)(f)) \\
& (n, s, [f\ (\texttt{fin}\ 1), \ldots, f\ (\texttt{fin}\ n)]) \\
= & \ (n, s, [x_1, \ldots, x_n]) \qquad \square
\end{aligned}
$$

The proof that $(n, (n, s, \texttt{self}^n), l) : \mathsf{Cotra}(\mathsf{Cotra}(F))(A)$ is the characterization of $(n, s, l) : \mathsf{Cotra}(F)(A)$ under traversal ${}^F\omega$ is as follows:

Proof The following trivially follows from Lemma 7.4

$$
\begin{aligned}
& \texttt{insert}\ (n, s, \texttt{self}^n)\ l_1 \ldots l_n \\
= & \ (n, s, [l_1, \ldots, l_n]) \\
= & \ (n, s, l)
\end{aligned}
$$

Thus, $(n, s, \texttt{self}^n)$ satisfies the *reconstruction formula*.

For all $X, Y : \mathsf{Set}$, $H : \mathsf{App}$, $f : X \to H(Y)$, $x_1 \ldots x_n : X$

$$
\begin{aligned}
& {}^F\omega\texttt{traverse}^H_{X,Y}\ f\ (\texttt{insert}\ (n, s, \texttt{self}^n)\ x_1 \ldots x_n) \\
= & \ (\text{Lemma 7.4}) \\
& {}^F\omega\texttt{traverse}^H_{X,Y}\ f\ (n, s, [x_1, \ldots x_n]) \\
= & \ (\text{Definition of } {}^F\omega\texttt{traverse}) \\
& \texttt{pure}\ (\lambda k.\ (n, s, k)) \circledast ({}^n\sigma\texttt{traverse}^H_{X,Y}\ f\ [x_1, \ldots, x_n]) \\
= & \ (\text{Definition of } {}^n\sigma\texttt{traverse}) \\
& \texttt{pure}\ (\lambda k.\ (n, s, k)) \circledast (\texttt{pure}\ (\lambda y_1\ \ldots\ y_n.\ [y_1, \ldots, y_n]) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n) \\
= & \ (\text{Lemma 7.2}) \\
& \texttt{pure}\ (\lambda y_1\ \ldots\ y_n.\ (\lambda k.\ (n, s, k))\ [y_1, \ldots, y_n]) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n \\
= & \ \texttt{pure}\ (\lambda y_1\ \ldots\ y_n.\ (n, s, [y_1, \ldots, y_n])) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n \\
= & \ (\text{Lemma 7.4}) \\
& \texttt{pure}\ (\lambda y_1\ \ldots\ y_n.\ \texttt{insert}\ (n, s, \texttt{self}^n)\ y_1\ \ldots\ y_n) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n \\
= & \ \texttt{pure}\ (\texttt{insert}\ (n, s, \texttt{self}^n)) \circledast f\ x_1 \circledast \ldots \circledast f\ x_n
\end{aligned}
$$

Thus, $(n, s, \texttt{self}^n)$ satisfies the *strong construction formula.*

$(n, s, \texttt{self}^n)$ is therefore the characterizing shape of (n, s, l), and $(n, (n, s, \texttt{self}^n), l)$ is the characterization of $(n, s, l) : \mathsf{Cotra}(F)(A)$ under ${}^F\omega$. □

From this, it may now be proven that *the right triangle identity law* is satisfied:

For all $F : [\mathsf{Set}, \mathsf{Set}]$, $A : \mathsf{Set}$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$\mathsf{id}_{\mathsf{Cotra}(F)(A)}\ (n, s, l) = (\texttt{hoist counit}_F)_A \left(\texttt{unit}_{(\mathsf{Cotra}(F), {}^F\omega), A}\ (n, s, l)\right)$$

Proof

$$\begin{aligned}
& (\texttt{hoist counit}_F)_A \left(\texttt{unit}_{(\mathsf{Cotra}(F), {}^F\omega), A}\ (n, s, l)\right) \\
= \ & ((n, (n, s, \texttt{self}^n), l) \text{ is the characterization of } (n, s, l)) \\
& (\texttt{hoist counit}_F)_A\ (n, (n, s, \texttt{self}^n), l) \\
= \ & (\text{Definition of } \texttt{hoist}) \\
& \left(n, \texttt{counit}_{F, (\mathsf{Fin}\ n)}\ (n, s, \texttt{self}^n), l\right) \\
= \ & (\text{Definition of } \texttt{counit}_{F, (\mathsf{Fin}\ n)}) \\
& (n, \texttt{insert}\ s\ \texttt{self}_1^n\ \ldots\ \texttt{self}_n^n, l) \\
= \ & (n, \texttt{insert}\ s\ (\texttt{fin}\ 1) \ldots (\texttt{fin}\ n), l) \\
= \ & (\text{Definition of } \texttt{insert}\ s) \\
& (n, F(f)\ s, l) \\
& \text{where } f\ (\texttt{fin}\ i) = \texttt{fin}\ i \\
= \ & \left(n, F(\mathsf{id}_{(\mathsf{Fin}\ n)})\ s, l\right) \\
= \ & (\text{Preservation of identity}) \\
& (n, s, l) \\
= \ & \mathsf{id}_{\mathsf{Cotra}(F)(A)}\ (n, s, l) \qquad \square
\end{aligned}$$

8 The Representational Encoding in Haskell

The representational encoding is not easily adaptable to Haskell, due to the associated data type being a dependent type, which Haskell does not intrinsically support. However, through the use of several GHC language extensions, it is possible to emulate a portion of the functionality that dependent types provide. By doing so, the representational encoding may be implemented. The following extensions will be used:

- `DataKinds`: This extension allows for the use of any data type as a *kind*, with the data constructors thereof as its members. In order to disambiguate the uses of type-level data constructors from the names of types, a tick `'` is used as a prefix to these. Unlike true dependent types, this extension does not allow for promoting arbitrary functions.

- `GADTs`: Generalized Algebraic Data Types. Allows for defining algebraic data types in terms of the types of its *constructors*. This extension allows for constructors to be constricted to particular parametrized types, as well as *existential quantification*: embedding types which are unknown at run-time, but may be subject to constraints.

- `KindSignatures`: This extension allows for explicitly annotating types with their kind.

- `ScopedTypeVariables`: Allows for type variables to apply over entire *scopes* through the use of the `forall` keyword. This extension also allows for using type annotations in patterns.

- `RankNTypes`: This extension permits the use of polymorphic functions as first-class citizens, which are represented through the `forall` keyword. For example, a function of type `(forall x. x -> [x]) -> [Int]` accepts any *polymorphic* function `x -> [x]` as an argument and returns a value of `[Int]`.

- `EmptyCase`: This extension allows for case expressions with no patterns. If any such case is evaluated, the expression matched upon is first evaluated, and if that evaluation succeeds, the program raises an exception (as no patterns exist). This extension is useful when used together with data types without any valid inhabitant: any expression of that type must be such that evaluating it is either a nonterminating operation, or causes an exception to be raised.

- `DeriveTraversable`: This extension allows for deriving instances of `Functor`, `Foldable`, and `Traversable` for data types. Such instances are guaranteed to be valid[9] and their behaviour is controlled by how the data type is defined.

- `StandaloneDeriving`: This extension allows for making derived instances of a data type separate from the data type itself, and provide additional context to the instance. Standalone deriving is used by

[9] With certain caveats related to data types that support infinite values: see the discussion in Section 11 as well as Bird et al. [2]

declaring an instance without a body together with the `deriving` prefix. Standalone deriving is often needed in order to derive instances of generalized algebraic data types.

Before defining Cotra, it is necessary to encode natural numbers $\mathbb{N}$, finitary types Fin, and fixed-size vectors Vec in Haskell. This is done as follows:

```
data Nat
  = Z
  | S Nat

data SNat (n :: Nat) where
  SZ :: SNat 'Z
  SS :: SNat n -> SNat ('S n)

data Fin (n :: Nat) where
  FZ :: Fin ('S n)
  FS :: Fin n -> Fin ('S n)

{-
  absurdFin encodes the notion of "from falsehood, anything follows."
  If we were ever to get a valid inhabitant of Fin 'Z,
  which is impossible, we may derive anything.
-}
absurdFin :: Fin 'Z -> a
absurdFin x = case x of

data Vec (n :: Nat) a where
  End :: Vec 'Z a
  (:-) :: a -> Vec n a -> Vec ('S n) a

deriving instance Functor (Vec n)
deriving instance Foldable (Vec n)
deriving instance Traversable (Vec n)
```

`SNat n` is a *singleton* for a given `n : Nat`; only a single valid inhabitant of `SNat n` exists. This type is needed in order to recover information about what natural number `n` corresponds to at run-time.

`Functor`, `Foldable`, and `Traversable` are derived for fixed-size vectors. The derived `Traversable` instance traverses elements left-to-right in the order elements appear within the fields of any constructor [8]. As the element `a` precedes the recursive `Vec n a` within the constructor `(:-)`, this means that

elements are traversed left to right within the fixed-size vector, which is the desired behaviour.

Cotra may now be defined as follows:

```
data Cotra f a where
  Cotra :: SNat n -> f (Fin n) -> Vec n a -> Cotra f a

deriving instance Functor (Cotra f)
deriving instance Foldable (Cotra f)
deriving instance Traversable (Cotra f)
```

The natural number associated with the triple is *existentially quantified*, but may be recovered through the singleton `SNat n`. The derived `Traversable` instances reuses that of `Vec n`, meaning that its semantics is that of the traversal ${}^{F}\omega$ as defined for the representational encoding.

Implementation of `hoist` and `counit` is straight-forward:

```
hoist :: (forall x. f x -> g x) -> Cotra f a -> Cotra g a
hoist nat (Cotra n s l) = Cotra n (nat s) l

index :: Vec n a -> Fin n -> a
index (a :- _) FZ = a
index (_ :- r) (FS fn) = index r fn
index End g = absurdFin g

counit :: Functor f => Cotra f a -> f a
counit (Cotra _ s l) = fmap (index l) s
```

The unit is significantly more complicated. It makes use of the `Batch` applicative functor as described in Section 6.2. It also requires the following type and functions in order to be defined safely:

```
data SomeNEVec a where
  SomeNEVec :: Vec ('S n) a -> SomeNEVec a

vecToSNat :: Vec n a -> SNat n
vecToSNat End = SZ
vecToSNat (_ :- r) = SS (vecToSNat r)

toFin :: SNat n -> Fin ('S n)
toFin SZ = FZ
toFin (SS n) = FS (toFin n)
```

```
raise :: Fin n -> Fin ('S n)
raise FZ = FZ
raise (FS s) = FS (raise s)
```

- `SomeNEVec` is the type of non-empty vectors of any size.
- `vecToSNat` given a vector of size n yields the singleton for n.
- `toFin`, given `SNat n`, creates the highest ordered value of `Fin ('S n)`.
- `raise` takes a value of `Fin n` and creates the value with corresponding ordering in `Fin ('S n)`.

The unit is split into two functions; `fromTraversal`, which generalizes the operation, and `unit` proper. `fromTraversal` may be used in order to obtain the characterization of a value under some traversal other than the one associated with the `Traversable` type class, by providing the corresponding *traverse* operation.

```
unit :: Traversable t => t a -> Cotra t a
unit = fromTraversal traverse

fromTraversal :: forall s f a.
                 (    forall x.
                      (a -> Batch a x x)
                   -> s
                   -> Batch a x (f x)
                 )
              -> s
              -> Cotra f a
fromTraversal tr s =
  let
    g :: forall x. Batch a x (f x)
    g = tr batch s
  in
    case g of
      P h -> Cotra SZ h End
      r :*: a -> case makeElems r (SomeNEVec (a :- End)) of
        SomeNEVec (elems :: Vec ('S n) a) ->
          let
            sz :: SNat n
            SS sz = vecToSNat elems
```

```
          shape :: f (Fin ('S n))
          shape = makeShape (toFin sz) g
        in
          Cotra (SS sz) shape elems
  where
    makeElems :: Batch a b c -> SomeNEVec a -> SomeNEVec a
    makeElems (r :*: a) (SomeNEVec v) =
      makeElems r (SomeNEVec (a :- v))
    makeElems _         v             =
      v

    makeShape :: Fin ('S n) -> Batch a (Fin ('S n)) c -> c
    makeShape FZ (P l :*: _)   =
      l FZ
    makeShape (FS n) (r :*: _) =
      makeShape (raise n) r (FS n)
    makeShape _      _         =
      error "Impossible: elements of g dependent on type"
```

This definition is daunting, but it follows the procedure detailed in Section 6.2 to constructively find the characterization of any traversable container. The boilerplate is in order to accommodate Haskell's type system. The unit executes `fromTraversal`, providing `traverse` as the first argument. This is used to build the polymorphic `Batch` corresponding to the characterization; this is called `g`. In order to determine the number of elements in the batch, `g` is first instantiated with `Fin 'Z` as the type variable, and then pattern matched upon. If no `:*:` are present, then no elements are in the batch, and the resulting `h :: f (Fin 'Z)` may be used as the characterizing shape when constructing the characterization, which will contain no elements. If, however, the batch contains at least one `:*:` constructor, then it will contain at least one element. `makeElems` is used to determine how many elements there are, as well as storing these in a vector that may be used as the vector of elements in the characterization.

With the number of elements – `'S n` – determined, `g` is instantiated with `Fin ('S n)`, which is then used to construct the shape `f (Fin ('S n))` by using `makeShape`. `makeShape` converts `g` to the underlying function `make :: Fin ('S n) -> ... -> Fin ('S n) -> f (Fin ('S n))`, and provides each valid inhabitant `fn :: Fin ('S n)` as the argument of `make` whose ordering corresponds to `fn`. As detailed in Section 6.2, this reconstructs the characterizing shape.

The use of `makeShape` presumes that the number of elements in `g` is the same no matter what type it is instantiated with: the "impossible" exception will be raised if that is not the case. As discussed in Section 6.2, the exact elements is `g` are independent of the type variable assuming the provided traversal is valid. However, even if it is not, Haskell's type system guarantees this property anyway: it is impossible to construct a value of type `forall x. Batch a x (f x)` for which the exact elements of the batch depends on the provided type, as type variables may only be analyzed through constraints or other provided environmental information. `g` has access to neither, and therefore its structure must be independent of how the type variable is instantiated.

9 Practical Applications

9.1 As Novel Traversable Functors

As shown by the representational encoding, cofree traversable functors amount to pairing a list of elements together with some structure which these elements may be inserted into. Consequently, cofree traversable functors are not very varied: the differences lie only in what power the associated shape provides.

For example, the cofree traversable functor on the Maybe functor yields representational triples where each shape is of the form $s : \mathsf{Maybe}\ (\mathsf{Fin}\ n)$. This may be interpreted as each list of elements being associated with a *focus* into that list; `just (fin i)` references the i:th element of the list, and `nothing` representing the end of the list, past the last element. The focus may be moved to a different part of the list through modifying the shape.

In contrast, consider the cofree traversable functor on the *selection functor of* S: $\mathsf{Select}(S) = \lambda x.\ (x \to S) \to x$, which is used for modelling search algorithms [5]. Shapes have the form $s : (\mathsf{Fin}\ n \to S) \to \mathsf{Fin}\ n$; or, expressed differently, $s : \forall x.\ \mathsf{Vec}\ n\ x \to (x \to S) \to x$. The shape may be interpreted as a *search* that, provided n objects of any one type X, as well as an interpreter $X \to S$, makes a selection among the provided objects from the information provided by the interpreter. A representational triple may therefore be interpreted as associating a particular search with its *search space*, such that the search space may be inspected and modified.

Note that the type $(\mathsf{Fin}\ 0 \to S) \to \mathsf{Fin}\ 0$ is uninhabited for any choice S, and therefore any representational triples corresponding to $\mathsf{Cotra}(\mathsf{Select}(S))$ must contain at least one element. This demonstrates how the choice of functor may have bearing on the nature of the resulting cofree traversable functor.

The cofree traversable functor on the function functor $\lambda x.\ S \to x$ corresponds to a weaker variant of $\mathsf{Cotra}(\mathsf{Select}(S))$. Shapes have the form $s : S \to \mathsf{Fin}\ n$, or $s : \forall x.\ \mathsf{Vec}\ n\ x \to S \to x$; the interpreter $\times \to S$ is replaced with a simple value of S. This corresponds to a simple indexing operation based upon the provided value of S, as it is not possible to retrieve information about each individual object. A representational triple may therefore be interpreted as associating a list with a function that converts values of S into indices of that list.

Another construction of note is the cofree traversable functor on the *indexed continuation functor of S and T*: $\mathsf{IxCont}(S)(T) = \lambda x.\ (x \to S) \to T$. $\mathsf{Cotra}(\mathsf{IxCont}(S)(T))(A)$ is isomorphic to the data type $\mathsf{Batch}(A)(S)(T)$, which was used in Section 6.2 in order to implement the unit of the representational encoding. Shapes of $\mathsf{Cotra}(\mathsf{IxCont}(S)(T))$ have the form $(\mathsf{Fin}\ n \to S) \to T$, which is isomorphic to $(\mathsf{Vec}\ n)(S) \to T$; i.e. a function that takes n values of type S and produces a value of type T. A value of $\mathsf{Cotra}(\mathsf{IxCont}(S)(T))(A)$ therefore represents a list of values of type A, together with a function that takes as many values of type S and produces a T. This matches the definition of $\mathsf{Batch}(A)(S)(T)$.

9.2 As Intermediate Structures

The greatest apparent use of cofree traversable functors lies in the flexibility that the representational encoding offers when used with existing traversable functors.

The full extent of the power inherent to traversable functors, as implied by the representation theorem (Theorem 6.1), is not easily accessible through `traverse` or `sequence`, which are the only operations provided by the interface of traversable functors. The representational encoding may be used to address this: given any traversable functor (T, δ), and a value $t : T(A)$, $\mathtt{unit}_{(T,\delta)}\ t$ derives the characterization of t under δ, thereby revealing the underlying shape and elements which may then easily be manipulated. Any triple $\mathsf{Cotra}(T)(A)$ may be collapsed to $T(A)$ through `counit`, thereby integrating any changes made.

Cofree traversable functors may therefore be valuable as intermediate structures, where values of traversable functors may be converted to representational triples, manipulated, and then converted back to a functorial value. This may be leveraged to manipulate traversable functors generically.

Two examples of such use cases of cofree traversable functors will be demonstrated: creating traversals, and creating generic zippers.

9.2.1 Creating Traversals

Day Convolution, as provided in the `Data.Functor.Day` module of the `kan-extensions` package, [15] is a functor that is defined as follows:[10]

```
data Day f g a where
  Day :: f x -> g y -> (x -> y -> a) -> Day f g a
```

`kan-extensions` provides a `Functor` instance for `Day f g`, as well as instances for other functor families such as `Applicative`. However, it does not provide a `Traversable` instance, presumably because a suitable implementation was not found.

By leveraging cofree traversable functors, it is possible to create an instance of `Traversable` as follows:

```
instance (Traversable f, Traversable g)
      => Traversable (Day f g) where
  -- traverse :: Applicative h => (a -> h b)
  --           -> Day f g a -> h (Day f g b)
  traverse f (Day fx gy xya) = case (unit fx, unit gy) of
    (Cotra _ s l, Cotra _ t r) ->
          (\v -> Day s t (\i j -> index (index v i) j))
      <$> traverse (\x -> traverse (\y -> f (xya x y)) r) l
```

By using `unit`, it is possible to separate the elements of `fx` and `gy` from their shapes. The elements of `Day f g a` that are traversed is each possible combination of elements `(x,y)` from `fx` and `gy` – which are gathered by `l` and `r` – converted to be of type `a` via `xya`. This results in a value of type `h (Vec n (Vec m b))`: a $n \times m$ matrix wrapped inside of the applicative context. Each element of this matrix at any indices i and j corresponds to the result that was yielded by combining the i:th element of `fx` with the j:th element of `gy`. The mapping of `(\v -> Day s t (\i j -> index (index v i) j))` converts this matrix to a value of `Day f g b` with the same shape as the original traversable container. `fx` and `gy` are replaced by their respective shapes `s` and `t`. As the type of elements will become hidden under existential quantification, and the structure of any traversable container and its characterizing shape are identical, it is not possible to discern between the original containers and their shapes once the constructor has been applied. The arguments `i` and `j` of the function used for the `Day` constructor correspond to positions within the shapes `s` and `t`. These are used as indices within the

[10] Instead of GADTs, `kan-extensions` makes use of the `ExistentialQuantification` language extension in order to define `Day`. The resulting data type is the same.

matrix. As the value at those indices was created from the elements that were originally located at the positions that `i` and `j` correspond to, each element of the new container corresponds to the mapping of the element at the corresponding position of the original container; preserving the structure.

A formal proof of the validity of this instance has not been made, but I conjecture that it is valid, as it does not exhibit any of the behaviour that the laws of traversable functors are meant to forbid [13].

The *Naturality* and *Consistency* laws follow from parametricity, assuming the *Identity* and *Linearity* laws are satisfied.

The *Identity* law enforces that the structure is preserved, each element is traversed over at least once, and mapped elements are not removed, duplicated, or rearranged. As discussed, the `Traversable` instance for `Day f g` satisfies these properties.

The *Linearity* law enforces that each element is not traversed multiple times. Although each element `x` of `fx` and `y` of `gy` is used multiple times in order to create the matrix, the elements that are *traversed* are pairs `(x,y)`, converted via `xya`; and each possible pair is only traversed once. Therefore, the `Traversable` instance for `Day f g` satisfies this property.

9.2.2 Generic Bidirectional Zipper

Cofree traversable functors may be used to create a *bidirectional zipper* [12], that is traversable, for any arbitrary traversable functor. Its functionality lies in using `unit` in order to separate the elements of a traversable container from its shape, and then placing a focus by separating the elements after the focus with those before the focus. The elements before the focus are stored in reverse order, such that moving the focus to a neighbouring element may be performed in constant time by moving the first element of either vector to the other.

In order to implement this zipper, the following additional language extensions are required:

- `TypeFamilies`

  Type Families provide limited type-indexed data types and named functions on types, in order to facilitate type-level programming.

- `UndecidableInstances`

  This extension relaxes GHC's termination rules for instances of type classes and definitions of type synonyms. These rules guarantee that type-checking will terminate, but are too restrictive to accept the implementation of the zipper.

The implementation is as follows:

```
import Control.Applicative.Backwards
import Data.Foldable (foldl')

type family Add (n :: Nat) (m :: Nat) where
  Add 'Z m = m
  Add ('S n) m = Add n ('S m)

-- | Appends the elements of the left vector to
-- the right in reverse order.
reverseApp :: Vec n a -> Vec m a -> Vec (Add n m) a
reverseApp End      r = r
reverseApp (a :- l) r = reverseApp l (a :- r)

{- |
  A GADT that may be used together with 'Cotra' in order
  to allow an additional number of "holes" in the predefined
  context of the parametrized functor.
-}
data Partial (n :: Nat) f a where
  Partial :: f (Fin (Add n m)) -> Partial n f (Fin m)

-- | A traversable bidirectional zipper for any
-- arbitrary traversable.
data Walk t a where
  Walk :: SNat n
       -> Vec n a
       -> Cotra (Partial n t) a
       -> Walk t a

deriving instance Functor (Walk t)

instance Foldable (Walk t) where
  foldr c b (Walk _ l r) = foldl' (flip c) (foldr c b r) l

instance Traversable (Walk t) where
  traverse f (Walk n l t) =
        Walk n
    <$> forwards (traverse (Backwards . f) l)
    <*> traverse f t
```

```
unzip :: Traversable t => t a -> Walk t a
unzip t = case unit t of
  Cotra n s l -> Walk SZ End (Cotra n (Partial s) l)

rezip :: Functor t => Walk t a -> t a
rezip (Walk _ ll (Cotra _ (Partial s) rl)) =
  index (reverseApp ll rl) <$> s

right :: Walk t a -> Maybe (Walk t a)
right (Walk ln ll (Cotra (SS rn) (Partial s) (a :- rl))) =
  Just $
    Walk
      (SS ln)
      (a :- ll)
      (Cotra rn (Partial s) rl)
right _ =
  Nothing

left :: Walk t a -> Maybe (Walk t a)
left (Walk (SS ln) (a :- ll) (Cotra rn (Partial s) rl)) =
  Just $
    Walk
      ln
      ll
      (Cotra (SS rn) (Partial s) (a :- rl))
left _ =
  Nothing

focus :: Walk t a -> Maybe a
focus (Walk _ _ (Cotra _ _ (a :- _))) =
  Just a
focus _ =
  Nothing

write :: a -> Walk t a -> Maybe (Walk t a)
write a (Walk ln ll (Cotra rn s (_ :- rl))) =
  Just $
    Walk
      ln
      ll
```

```
      (Cotra rn s (a :- rl))
write _ _ =
  Nothing
```

The zipper – represented by the `Walk` data type – may be converted to and from traversable containers through `unzip` and `rezip`; the focus may be moved left or right through the respective operations, and the element at the focus may be read or modified through `focus` and `write`. These operations fail if the focus can't be moved any further, or if the focus is past the final element.

The elements behind the focus are stored in reverse order, and thus deriving the instance of `Traversable` for `Walk` would cause the order in which elements are traversed to depend on where the focus is placed. This behaviour is not desirable, and therefore the instance for `Traversable` (and `Foldable`) is implemented manually. The elements of the vector before the focus are traversed in reverse order, followed by the focus, and then the elements after the focus in regular order. The instance is implemented through the use of the `Backwards` applicative transformer, provided in the `Control.Applicative.Backwards` module of the `transformers` Hackage package [10]. `Backwards` applies effects of the parametrized applicative in reverse order, which, when used together with *traverse*, effectively performs the traversal *backwards*; from the last element to the first.

10 Related Work

To my knowledge, free constructions for traversable functors have previously been described only in the Hackage `free-functors` package [22]. The `Data.Functor.HCofree` module of this package describes a generic construction for developing cofree functors in relation to commonly used functor families – among them, traversable functors. Indeed, the implementation directly corresponds to the *Initial* encoding presented in Appendix F. The package lacks any accompanying definition of cofree traversable functors; it does not demonstrate that the construction it presents corresponds to free constructions of traversable functors. In contrast, this thesis presents a formal definition of cofree traversable functors, and proves that the provided implementation – in the form of the representational encoding – is correct. In addition, the encoding presented by `free-functors` is less expressive than the representational encoding, and cannot be used for any of the practical applications demonstrated in Section 9.2.

More broadly, the thesis relates to properties of traversable functors: it provides additional insight by introducing free constructions in relation to

these. As such, related work consists of investigations into the nature of traversable functors.

Jaskelioff and Rypacek, in their paper *An Investigation of the Laws of Traversals* [13] introduced a category-theoretical formalization of traversable functors through *traversals*. This formed the basis of the category-theoretical definition of the category of traversable functors Tra, and from it, the definition of cofree traversable functors in Section 4.

Bird et al. in their paper *Understanding Idiomatic Traversals Forwards and Backwards* [2] present the representation theorem, thereby showing that any traversable object is characterized in terms of its *shape* and its *elements*. In this thesis, this result has been used in order to develop the *representational encoding*, where objects of cofree traversable functors are represented through *representational triples* with the likeness of the characterizations that Bird et al. present. In particular, this thesis has shown that the *unit* of the representational encoding corresponds to deriving the characterization of any traversable container.

The thesis also relates to work done on developing free constructions for functor families for use within purely functional programming. For example, the package `free` on Hackage provides various implementations of free constructions for various functor families, including applicative functors, monads, and comonads [14]. This thesis contributes to this area by introducing free constructions for traversable functors.

11 Future Work

The implementation of the representational encoding in Section 8 causes the `unit` and `counit` operations to become extremely inefficient; typically quadratic in behaviour. `unit` is adversely affected by the implementation of the `Batch` applicative functor, and the use of `raise` in order to convert values of finitary types. `counit` is adversely affected by the choice of shape, as insertion into the shape is defined through the mapping `index 1` – an $O(n)$ operation – over the functorial value, applying the function to each item.

In addition, the representational encoding is not capable of converting traversable containers that contain infinite elements. As Bird et al. have shown, traversable functors that support such values technically violate the laws of traversable functors, [2] however the use of such unlawful traversable functors is common – even lists may be infinite in length – and supporting these is desirable.

These issues may be addressed through implementing optimized variants of the representational encoding. An example of such a variant is presented

in Appendix E. There also exists other potential encodings of the cofree traversable functor functor. These perform significantly better than even optimized variants of the representational encoding in certain contexts, and support a wider range of infinite traversable containers. However, these are less flexible as they do not provide direct access to the elements. These encodings are presented in Appendix F.

I conjecture that each of the alternative representations, including the variant of the representational encoding, are valid; however, I have not developed a formal proof for these. Further work is therefore required in order to prove that these are valid, or to develop other representations of cofree traversable functors.

The potential uses of cofree traversable functors need to be explored in greater detail. The most promising application – manipulating traversable functors generically – is specific to variants of the representational encoding. In addition, it only explores the use of cofree traversable functors as intermediate structures, converting to and from functors that are already traversable. The usefulness of particular instances of cofree traversable functors as novel data structures is unknown. Examples of these have been presented, but these are either of questionable use, or replicate existing data structures. Further work is therefore required in order to evaluate the potential of this application of cofree traversable functors.

12 Conclusion

Free constructions in relation to traversable functors have been developed in the form of *cofree traversable functors*. These have been formalized via category theory, from which a type-theoretical model was derived. The existence of cofree traversable functors have been proven via an implementation known as the *representational encoding of the cofree traversable functor functor*. This encoding has been implemented in Haskell, using the representation of traversable functors as given by the `Traversable` type class in `base` [6].

The representational encoding may also be used to derive the *characterization* of any traversable container, as given by the *Representation Theorem* for traversable functors introduced by Bird et al. [2] This allows for separating any traversable container into its *elements* and *shape*, such that these may be inspected and manipulated independently, and then be converted back to a traversable container, integrating any changes made. This mechanism allows for the generic manipulation of traversable functors which may otherwise be difficult to perform through the use of the interface alone. Examples of such applications include creating bidirectional zippers for any traversable

functor, or developing implementations of complex traversable functors which are based on other traversable functors.

Cofree traversable functors may also be used in order to develop novel traversable functors, represented as the cofree traversable functor of another functor. However, the greater potential of this use is as of yet unknown.

The implementation of the representational encoding as presented in this thesis has severe performance issues, and optimized implementations or different encodings are needed in order to address these. Other possible representations have been demonstrated, but have not yet been proven to be valid.

References

[1] Steve Awodey. *Category theory*. Oxford University Press, 2010.

[2] Richard Bird, Jeremy Gibbons, Stefan Mehner, Janis Voigtländer, and Tom Schrijvers. "Understanding Idiomatic Traversals Backwards and Forwards". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell '13. Boston, Massachusetts, USA: ACM, 2013, pp. 25–36. ISBN: 978-1-4503-2383-3. DOI: 10.1145/2503778.2503781.

[3] Paolo Capriotti and Ambrus Kaposi. "Free Applicative Functors". In: *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*. 2014, pp. 2–30. DOI: 10.4204/EPTCS.153.2.

[4] Renee Elio, Jim Hoover, Ioanis Nikolaidis, Mohammad Salavatipour, Lorna Stewart, and Ken Wong. *About computing science research methodology*. 2011.

[5] Martin Escardó and Paulo Oliva. "Selection functions, bar recursion and backward induction". In: *Mathematical structures in computer science* 20.2 (2010), pp. 127–168.

[6] GHC Team. *base: Basic libraries*. Version 4.12.0.0. Sept. 2018. URL: https://hackage.haskell.org/package/base-4.12.0.0.

[7] GHC Team. *The Glasgow Haskell Compiler*. 2019. URL: https://www.haskell.org/ghc/.

[8] GHC Wiki. *Support for deriving Functor, Foldable, and Traversable instances*. [Online; accessed 21-May-2019]. 2019. URL: https://gitlab.haskell.org/ghc/ghc/wikis/commentary/compiler/derive-functor?version_id=5673861776054761 9e264e529ce7af8de702937d.

[9] Jeremy Gibbons and Bruno C d S Oliveira. "The essence of the iterator pattern". In: *Journal of functional programming* 19.3-4 (2009), pp. 377–402. DOI: 10.1017/S0956796809007291.

[10] Andy Gill and Ross Paterson. *transformers: Concrete functor and monad transformers*. Version 0.5.6.2. Feb. 2019. URL: http://hackage.haskell.org/package/transformers-0.5.6.2.

[11] HaskellWiki. *Typeclassopedia — HaskellWiki*. [Online; accessed 21-May-2019]. 2019. URL: https://wiki.haskell.org/index.php?title=Typeclassopedia&oldid=62747.

[12] Gérard Huet. "The Zipper". In: *Journal of Functional Programming* 7.5 (1997), pp. 549–554.

[13] Mauro Jaskelioff and Ondrej Rypacek. "An Investigation of the Laws of Traversals". In: *Electronic Proceedings in Theoretical Computer Science* 76 (Feb. 2012). DOI: 10.4204/EPTCS.76.5.

[14] Edward Kmett. *free: Monads for free*. Version 5.1.1. May 2019. URL: https://hackage.haskell.org/package/free-5.1.1.

[15] Edward Kmett. *kan-extensions: Kan extensions, Kan lifts, the Yoneda lemma, and (co)density (co)monads*. Version 5.2. July 2018. URL: https://hackage.haskell.org/package/kan-extensions-5.2.

[16] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media, 2013.

[17] Conor Mcbride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (Jan. 2008), pp. 1–13. ISSN: 0956-7968. DOI: 10.1017/S0956796807006326.

[18] nLab authors. *free functor*. Jan. 2019. URL: https://ncatlab.org/nlab/show/free%20functor.

[19] Exequiel Rivas and Mauro Jaskelioff. "Notions of computation as monoids". In: *Journal of functional programming* 27 (2017). DOI: 10.1017/S0956796817000132.

[20] Wouter Swierstra. "Data types à la carte". In: *Journal of functional programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[21] Tarmo Uustalu and Varmo Vene. "Comonadic notions of computation". In: *Electronic Notes in Theoretical Computer Science* 203.5 (2008), pp. 263–284. DOI: 10.1016/j.entcs.2008.05.029.

[22] Sjoerd Visscher. *free-functors: Free functors, adjoint to functors that forget class constraints*. Version 1.0.1. Sept. 2018. URL: http://hackage.haskell.org/package/free-functors-1.0.1.

[23] Philip Wadler. "Theorems for Free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, 1989, pp. 347–359. ISBN: 0-89791-328-0. DOI: 10.1145/99370.99404.

[24] Love Waern. *Cofree-Traversable-Functors: Implementation of Cofree Traversable Functors in Haskell*. 2019. URL: https://github.com/KingoftheHomeless/Cofree-Traversable-Functors.

Appendices

A `contents` describe traversable morphisms

Lemma A.1. For all $A, B : \mathsf{Set}$, $F : \mathsf{App}$, $f : A \to F(B)$,

$$^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f : \mathsf{List}(A) \to F(\mathsf{List}(B))$$

is an applicative morphism $\mathsf{C}(\mathsf{List}(A)) \to (F \circ \mathsf{C}(\mathsf{List}(B))$, where σ is defined such that for a list $l : \mathsf{List}(A)$ of any length n

$$^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ l = \mathtt{pure}^{F}(\lambda x_1\ \ldots\ x_n.[x_1, \ldots, x_n]) \circledast^{F} f\ l_1 \circledast^{F} \ldots \circledast^{F} f\ l_n$$

Proof For all $X : \mathsf{Set}$, $x : X$

$$\begin{aligned}
& {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ (\mathtt{pure}^{\mathsf{C}(\mathsf{List}(A))}\ x) \\
= \ & \text{(Definition of } \mathtt{pure} \text{ for } \mathsf{C}(\mathsf{List}(A))) \\
& {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ [] \\
= \ & \text{(Definition of } {}^{\sigma}\mathtt{traverse}) \\
& \mathtt{pure}^{F}\ [] \\
= \ & \text{(Definition of } \mathtt{pure} \text{ for } \mathsf{C}(\mathsf{List}(B))) \\
& \mathtt{pure}^{F}\ (\mathtt{pure}^{\mathsf{C}(\mathsf{List}(B))}\ x) \\
= \ & \text{(Definition of } \mathtt{pure} \text{ for } F \circ \mathsf{C}(\mathsf{List}(B)) \\
& \mathtt{pure}^{F \circ \mathsf{C}(\mathsf{List}(B))}\ x
\end{aligned}$$

For all $l, r : \mathsf{List}(A)$, where l has length n and r has length m

$$
\begin{aligned}
& {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ (l \circledast^{\mathsf{C}(\mathsf{List}(A))} r) \\
= \ & {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ ([l_1, \ldots, l_n] \circledast^{\mathsf{C}(\mathsf{List}(A))} [r_1, \ldots, r_m]) \\
= \ & \text{(Definition of } \circledast \text{ for } \mathsf{C}(\mathsf{List}(A))) \\
& {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ ([l_1, \ldots, l_n, r_1, \ldots, r_m]) \\
= \ & \text{(Definition of } {}^{\sigma}\mathtt{traverse}) \\
& \quad \mathtt{pure}^{F}\ (\lambda x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ [x_1, \ldots, x_n, y_1, \ldots, y_m]) \\
& \circledast^{F} f\ l_1 \circledast^{F} \ldots \circledast^{F} f\ l_n \\
& \circledast^{F} f\ r_1 \circledast^{F} \ldots \circledast^{F} f\ r_m \\
= \ & \text{(Definition of } \circledast \text{ for } \mathsf{C}(\mathsf{List}(B))) \\
& \quad \mathtt{pure}^{F}\ (\lambda x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ [x_1, \ldots, x_n] \circledast^{\mathsf{C}(\mathsf{List}(B))} [y_1, \ldots, y_m]) \\
& \circledast^{F} f\ l_1 \circledast^{F} \ldots \circledast^{F} f\ l_n \\
& \circledast^{F} f\ r_1 \circledast^{F} \ldots \circledast^{F} f\ r_m \\
= \ & \text{(Flattening formula of } \circledast \text{ [2])} \\
& \quad \mathtt{pure}^{F}\ (\lambda x_1\ \ldots\ x_n\ ys.\ [x_1, \ldots, x_n] \circledast^{\mathsf{C}(\mathsf{List}(B))} ys) \\
& \circledast^{F} f\ l_1 \circledast^{F} \ldots \circledast^{F} f\ l_n \\
& \circledast^{F} (\mathtt{pure}^{F}\ (\lambda y_1\ \ldots\ y_m.\ [y_1, \ldots, y_m]) \circledast^{F} f\ r_1 \circledast^{F} \ldots f\ r_m) \\
= \ & \text{(Lemma 7.2)} \\
& \quad \mathtt{pure}^{F}\ (\lambda xs\ ys.\ xs \circledast^{\mathsf{C}(\mathsf{List}(B))} ys) \\
& \circledast^{F} (\mathtt{pure}^{F}\ (\lambda x_1\ \ldots\ x_n.\ [x_1, \ldots, x_n]) \circledast^{F} f\ l_1 \circledast^{F} \ldots \circledast^{F} f\ l_n) \\
& \circledast^{F} (\mathtt{pure}^{F}\ (\lambda y_1\ \ldots\ y_m.\ [y_1, \ldots, y_m]) \circledast^{F} f\ r_1 \circledast^{F} \ldots f\ r_m) \\
= \ & \text{(Definition of } {}^{\sigma}\mathtt{traverse}) \\
& \quad \mathtt{pure}^{F}\ (\lambda xs\ ys.\ xs \circledast^{\mathsf{C}(\mathsf{List}(B))} ys) \\
& \circledast^{F} {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ l \\
& \circledast^{F} {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ r \\
= \ & \mathtt{pure}^{F}\ (\circledast^{\mathsf{C}(\mathsf{List}(B))}) \circledast^{F} {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ l \circledast^{F} {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ r \\
= \ & \text{(Definition of } \circledast^{F \circ \mathsf{C}(\mathsf{List}(B))}) \\
& \mathtt{traverse}^{F}_{A,B}\ f\ l \circledast^{F \circ \mathsf{C}(\mathsf{List}(B))} \mathtt{traverse}^{F}_{A,B}\ f\ r
\end{aligned}
$$

□

For all traversable functors (T, δ), ${}^{\delta}\mathtt{contents}$ is a traversable morphism $(T, \delta) \to (\mathsf{List}, \sigma)$. I.e. for all $A, B : \mathsf{Set}$, $F : \mathsf{App}$, $f : A \to F(B)$

$$
{}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f \circ {}^{\delta}\mathtt{contents}_{A} = F({}^{\delta}\mathtt{contents}_{B}) \circ {}^{\delta}\mathtt{traverse}^{F}_{A,B}\ f
$$

Proof For all $t : T(A)$

$$
\begin{aligned}
& {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ \left({}^{\delta}\mathtt{contents}_{A}\ t\right) \\
= \ & \text{(Definition of } {}^{\delta}\mathtt{contents}\text{)} \\
& {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ \left({}^{\delta}\mathtt{traverse}^{\mathsf{C}(\mathsf{List}(A))}_{A,\bot}\ (\lambda x.\ [x])\ t\right) \\
= \ & \text{(Naturality law of traversable functors:} \\
& \text{Lemma A.1 states that } {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f \text{ is an applicative morphism.)} \\
& {}^{\delta}\mathtt{traverse}^{F\circ\mathsf{C}(\mathsf{List}(B))}_{A,\bot}\ ({}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f \circ (\lambda x.\ [x]))\ t \\
= \ & {}^{\delta}\mathtt{traverse}^{F\circ\mathsf{C}(\mathsf{List}(B))}_{A,\bot}\ (\lambda x.\ {}^{\sigma}\mathtt{traverse}^{F}_{A,B}\ f\ [x])\ t \\
= \ & \text{(Definition of } {}^{\sigma}\mathtt{traverse}\text{)} \\
& {}^{\delta}\mathtt{traverse}^{F\circ\mathsf{C}(\mathsf{List}(B))}_{A,\bot}\ (\lambda x.\ \mathtt{pure}^{F}\ (\lambda y.\ [y])\circledast^{F} f\ x)\ t \\
= \ & \text{(Consistency law of applicative functors)} \\
& {}^{\delta}\mathtt{traverse}^{F\circ\mathsf{C}(\mathsf{List}(B))}_{A,\bot}\ (\lambda x.\ F(\lambda y.\ [y])(f\ x))\ t \\
= \ & {}^{\delta}\mathtt{traverse}^{F\circ\mathsf{C}(\mathsf{List}(B))}_{A,\bot}\ (F(\lambda y.\ [y])\circ f)\ t \\
= \ & \text{(Linearity law of traversable functors)} \\
& F({}^{\delta}\mathtt{traverse}^{\mathsf{C}(\mathsf{List}(B))}_{B,\bot}\ (\lambda y.\ [y]))({}^{\delta}\mathtt{traverse}^{F}_{A,B}\ f\ t) \\
= \ & \text{(Definition of } {}^{\delta}\mathtt{contents}\text{)} \\
& F({}^{\delta}\mathtt{contents}_{B})({}^{\delta}\mathtt{traverse}^{F}_{A,B}\ f\ t)
\end{aligned}
$$

□

B The free traversable functor functor does not exist

The free traversable functor functor, which maps any functor $F : [\mathsf{Set}, \mathsf{Set}]$ to a corresponding free traversable functor on F, does not exist. In category-theoretical terms, the following proof implies that For, as defined in Definition 4.7, does not have a left adjoint.

Proof Assume the free traversable functor functor truly does exist, such that any functor $F : [\mathsf{Set}, \mathsf{Set}]$ is mapped to a corresponding free traversable functor $(\mathsf{Free}(F), {}^{F}\omega) : \mathsf{Tra}$, and for any natural transformation $\alpha : [\mathsf{Set}, \mathsf{Set}](F, G)$, the traversable morphism $\mathsf{Free}(f) : (\mathsf{Free}(F), {}^{F}\omega) \to (\mathsf{Free}(G), {}^{G}\omega)$ exists.

The free traversable functor functor also admits the existence of the *unit* and *counit*.

For any functor $F : [\mathsf{Set}, \mathsf{Set}]$, the unit defines a natural transformation:

$$\eta_F : F \to \mathsf{Free}(F)$$

For any traversable functor $(T, \delta) : \mathsf{Tra}$, the counit defines a traversable morphism:

$$\epsilon_{(T,\delta)} : (\mathsf{Free}(T), {}^{T}\omega) \to (T, \delta)$$

Two simple traversable functors will be used in order to prove that the free traversable functor functor cannot exist.

- The identity functor $\mathbf{1}$ can be seen as corresponding to containers that wraps nothing more than a single element, without any ambient context. It therefore has a unique traversal ${}^{\mathbf{1}}\delta$, which traverses the single element.
- $\mathsf{C}(\top)$, that is, the constant functor on the unit set, can be seen as corresponding to an empty container with no ambient context. It therefore has a unique traversal ${}^{\mathsf{C}(\top)}\delta$, which traverses no elements.

There exists the trivial natural transformation $\mathtt{term} : \mathbf{1} \to \mathsf{C}(\top)$, such that for any $X : \mathsf{Set}$, $x : X$

$$\begin{aligned} &\mathtt{term}_X : \mathbf{1}(X) \to \mathsf{C}(\top)(X) \\ &\mathtt{term}_X\ x = \top \end{aligned}$$

That is, the input container is completely disregarded, and mapped to the trivial empty container, represented by the only element of the unit set. The proof that the naturality condition is satisfied is trivial, and will be omitted.

Note that as $\epsilon_{(\mathbf{1},{}^{\mathbf{1}}\delta)} : (\mathsf{Free}(\mathbf{1}), {}^{\mathbf{1}}\omega) \to (\mathbf{1}, {}^{\mathbf{1}}\delta)$ must be a traversable morphism, any traversable container of $\mathsf{Free}(\mathbf{1})$ must contain exactly one element; otherwise it would possible for $\epsilon_{(\mathbf{1},{}^{\mathbf{1}}\delta)}$ to be forced to drop or insert elements, and therefore violate the preservation of contents. Similarly, as $\epsilon_{(\mathsf{C}(\top),{}^{\mathsf{C}(\top)}\delta)} : (\mathsf{Free}(\mathsf{C}(\top)), {}^{\mathsf{C}(\top)}\omega) \to (\mathsf{C}(\top), {}^{\mathsf{C}(\top)}\delta)$ exists, any traversable container of $\mathsf{Free}(\mathsf{C}(\top))$ must contain no elements.

Consider the mapping of the natural transformation $\mathtt{term}$ under the free traversable functor functor:

$$\mathsf{Free}(\mathtt{term}) : (\mathsf{Free}(\mathbf{1}), {}^{\mathbf{1}}\omega) \to (\mathsf{Free}(\mathsf{C}(\top)), {}^{\mathsf{C}(\top)}\omega)$$

For the free traversable functor functor to exist, this is a traversable morphism that is required to exist; but any traversable container of $\mathsf{Free}(\mathbf{1})$ must contain exactly one element, and any traversable container of $\mathsf{Free}(\mathsf{C}(\top))$ must contain no elements. Dropping elements are unavoidable if any traversable container were to be transformed.

$\mathsf{Free}(\mathtt{term})$ may therefore only exist if *no* traversable containers of $\mathsf{Free}(\mathbf{1})$ exist, in which case the necessary conditions for $\mathsf{Free}(\mathtt{term})$ to be a traversable morphism are vacuously satisfied. I.e. for all $X : \mathsf{Set}$, $\mathsf{Free}(\mathbf{1})(X)$ is uninhabited. However, this is not the case, as the *unit* may be used to create such an inhabitant: $\eta_{\mathbf{1},\top}\ \top : \mathsf{Free}(\mathbf{1})(\top)$.

The traversable morphism $\mathsf{Free}(\mathtt{term})$ cannot exist, which is contradictory to the assumption that the free traversable functor functor exists. Therefore, the free traversable functor functor does not exist. □

C Resulting elements of `build` are independent of type variable

Let

$$\begin{aligned}&\mathtt{contentsBatch} : \forall a\ s\ t.\ \mathsf{Batch}(a)(s)(t) \to \mathsf{List}(t)\\&\mathtt{contentsBatch}_{a,s,t}\ (\mathtt{P}\ f\ :\!*\!:\ a_1\ :\!*\!:\ \ldots\ :\!*\!:\ a_n) = [a_1, \ldots, a_n]\end{aligned}$$

$\mathtt{contentsBatch}$ is used to represent the elements of any $\mathsf{Batch}(A)(S)(T)$.

Lemma C.1. For all $A, S : \mathsf{Set}$

$$\mathtt{contentsBatch}_{A,S} : \forall t.\ \mathsf{Batch}(A)(S)(t) \to \mathsf{C}(\mathsf{List}(A))(t)$$

is an applicative morphism $\mathsf{Batch}(A)(S) \to \mathsf{C}(\mathsf{List}(A))$.

Proof The proof that $\mathtt{contentsBatch}$ satisfies the naturality condition for natural transformations is trivial, and will be omitted.

For all $X : \mathsf{Set}$, $x : X$

$$\begin{aligned}&\mathtt{contentsBatch}_{A,S,X}\ (\mathtt{pure}^{(}\mathsf{Batch}(A)(S))\ x)\\=\ &(\text{Definition of } \mathtt{pure} \text{ for } \mathsf{Batch}(A)(S))\\&\mathtt{contentsBatch}_{A,S,X}\ (\mathtt{P}\ x)\\=\ &(\text{Definition of } \mathtt{contentsBatch})\\&[]\\=\ &(\text{Definition of } \mathtt{pure} \text{ for } \mathsf{C}(\mathsf{List}(A)))\\&\mathtt{pure}^{\mathsf{C}(\mathsf{List}(B))}\ x\end{aligned}$$

For all $X, Y : \mathsf{Set}$, $(\mathtt{P}\ f :\!*\!:\ l_1 :\!*\!:\ \ldots\ :\!*\!:\ l_n) : \mathsf{Batch}(A)(S)(X \to Y)$, $(\mathtt{P}\ g :\!*\!:\ r_1 :\!*\!:\ \ldots\ :\!*\!:\ r_m) : \mathsf{Batch}(A)(S)(X)$

$$
\begin{aligned}
& \mathtt{contentsBatch}_{A,S,Y}\ (\\
& \qquad (\mathtt{P}\ f :\!*\!:\ l_1 :\!*\!:\ \ldots\ :\!*\!:\ l_n) \\
& \circledast^{\mathsf{Batch}(A)(S)}\ (\mathtt{P}\ g :\!*\!:\ r_1 :\!*\!:\ \ldots\ :\!*\!:\ r_m) \\
&) \\
= \ & \text{(Definition of } \circledast \text{ for } \mathsf{Batch}(A)(S)) \\
& \mathtt{contentsBatch}_{A,S,Y}\ (\\
& \qquad \mathtt{P}\ (\lambda x_1\ \ldots\ x_n\ y_1\ \ldots\ y_m.\ (f\ x_1\ \ldots\ x_n)(g\ y_1\ \ldots\ y_m)) \\
& \qquad :\!*\!:\ l_1 :\!*\!:\ \ldots\ :\!*\!:\ l_n :\!*\!:\ r_1 :\!*\!:\ \ldots\ :\!*\!:\ r_m \\
&) \\
= \ & \text{(Definition of } \mathtt{contentsBatch}) \\
& [l_1, \ldots, l_n, r_1 \ldots, r_m] \\
= \ & \text{(Definition of } \circledast^{\mathsf{C}(\mathsf{List}(A))}) \\
& [l_1, \ldots, l_n] \circledast^{\mathsf{C}(\mathsf{List}(A))} [r_1 \ldots, r_m] \\
= \ & \text{(Definition of } \mathtt{contentsBatch}) \\
& \qquad \mathtt{contentsBatch}_{A,S,X \to Y}\ (\mathtt{P}\ f :\!*\!:\ l_1 :\!*\!:\ \ldots\ :\!*\!:\ l_n) \\
& \circledast^{\mathsf{C}(\mathsf{List}(A))}\ \mathtt{contentsBatch}_{A,S,X}\ (\mathtt{P}\ g :\!*\!:\ r_1 :\!*\!:\ \ldots\ :\!*\!:\ r_m) \qquad \square
\end{aligned}
$$

Lemma C.2. For all $A, B : \mathsf{Set}$, either $A \to B$ or $B \to A$ is inhabited (or both).

Proof If A is uninhabited, then the function $\mathtt{absurd}^A : \forall\ x.\ A \to x$ exists; and thus $A \to B$ is inhabited. Similarly, if B is uninhabited, then the function $\mathtt{absurd}^B : \forall\ x.\ B \to x$ exists, and thus $B \to A$ is inhabited.

If both A and B are inhabited, then a function $f : A \to B$ may be created by choosing any inhabitant $b : B$ and defining

$$
\begin{aligned}
& f : A \to B \\
& f\ x = b
\end{aligned}
$$

And thus $A \to B$ is inhabited. $\square$

Lemma C.3. For all $(T, \delta) : \mathsf{Tra}$, $A, X, Y : \mathsf{Set}$, $t : T(A)$

$$
\begin{aligned}
& {}^{\delta}\mathtt{traverse}^{\mathsf{C}(\mathsf{List}(A))}_{A,X}\ (\lambda x.\ [x])\ t \\
= \ & {}^{\delta}\mathtt{traverse}^{\mathsf{C}(\mathsf{List}(A))}_{A,Y}\ (\lambda x.\ [x])\ t
\end{aligned}
$$

Proof If $X \to Y$ is inhabited, let $f : X \to Y$ be some inhabitant. Then:

$$
\begin{aligned}
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t \\
= \ & \mathsf{id}_{\mathsf{C}(\mathsf{List}(A))(T(Y))}\ \left({}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t\right) \\
= \ & (\text{Definition of } \mathsf{C}(\mathsf{List}(A))\ ({}^{\delta}\mathtt{traverse}_{X,Y}^{\mathbf{1}}\ f)) \\
& \mathsf{C}(\mathsf{List}(A))\ ({}^{\delta}\mathtt{traverse}_{X,Y}^{\mathbf{1}}\ f)\ \left({}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t\right) \\
= \ & (\text{Linearity law of traversable functors}) \\
& {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))\circ\mathbf{1}}\ (\mathsf{C}(\mathsf{List}(A))(f) \circ (\lambda x.\ [x]))\ t \\
= \ & (\text{Definition of } \mathsf{C}(\mathsf{List}(A))(f)) \\
& {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))\circ\mathbf{1}}\ \left(\mathsf{id}_{\mathsf{C}(\mathsf{List}(A))(Y)} \circ (\lambda x.\ [x])\right)\ t \\
= \ & {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))\circ\mathbf{1}}\ (\lambda x.\ [x])\ t \\
= \ & (\mathbf{1} \text{ is the neutral element to composition of applicative functors}) \\
& {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t
\end{aligned}
$$

If $Y \to X$ is inhabited, let $g : Y \to X$ be some inhabitant. Then:

$$
\begin{aligned}
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t \\
= \ & {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\mathsf{id}_{\mathsf{C}(\mathsf{List}(A)(X)} \circ (\lambda x.\ [x]))\ t \\
= \ & (\text{Definition of } \mathsf{C}(\mathsf{List}(A))(g)) \\
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\mathsf{C}(\mathsf{List}(A))(g) \circ (\lambda x.\ [x]))\ t \\
= \ & (\mathbf{1} \text{ is the neutral element to composition of applicative functors}) \\
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))\circ\mathbf{1}}\ (\mathsf{C}(\mathsf{List}(A))(g) \circ (\lambda x.\ [x]))\ t \\
= \ & (\text{Linearity law of traversable functors}) \\
& \mathsf{C}(\mathsf{List}(A))({}^{\delta}\mathtt{traverse}_{Y,X}^{\mathbf{1}}\ g)\ \left({}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t\right) \\
= \ & (\text{Definition of } \mathsf{C}(\mathsf{List}(A))(\mathtt{traverse}_{Y,X}^{\mathbf{1}}\ g)) \\
& \mathsf{id}_{\mathsf{C}(\mathsf{List}(A))(T(X))}\ \left({}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t\right) \\
= \ & {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t
\end{aligned}
$$

Lemma C.2 shows that either $X \to Y$ or $Y \to X$ must be inhabited. Thus

$$
\begin{aligned}
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x]) \\
= \ & {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])
\end{aligned}
$$

□

Let

$$\begin{aligned}&\texttt{batch} : \forall a\ x.\ a \to \mathsf{Batch}(a)(x)(x)\\&\texttt{batch}_{A,X}\ a = \mathsf{P}\ \mathsf{id}_X\ :\!*\!:\ a\end{aligned}$$

For all $(T, \delta) : \mathsf{Tra}$, let

$$\begin{aligned}&{}^{\delta}\texttt{build} : \forall a\ x.\ T(a) \to \mathsf{Batch}(a)(x)(T(x))\\&{}^{\delta}\texttt{build}_{A,X} = {}^{\delta}\texttt{traverse}_{A,X}^{\mathsf{Batch}(A)(X)}\ \texttt{batch}_{A,X}\end{aligned}$$

For all $(T, \delta) : \mathsf{Tra}$, $A, X, Y : \mathsf{Set}$, $t : T(A)$

$$\begin{aligned}&\texttt{contentsBatch}_{A,X,T(X)}\ ({}^{\delta}\texttt{build}_{A,X}\ t)\\=\ &\texttt{contentsBatch}_{A,Y,T(Y)}\ ({}^{\delta}\texttt{build}_{A,Y}\ t)\end{aligned}$$

Proof

$$
\begin{aligned}
& \mathtt{contentsBatch}_{A,X,T(X)}\ ({}^{\delta}\mathtt{build}_{A,X}\ t) \\
= \ & \text{(Definition of } {}^{\delta}\mathtt{build}\text{)} \\
& \mathtt{contentsBatch}_{A,X,T(X)}\ ({}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{Batch}(A)(X)}\ \mathtt{batch}_{A,X}\ t) \\
= \ & \text{(Naturality law of traversable functors:} \\
& \text{Lemma C.1 states that } \mathtt{contentsBatch}_{A,X} \text{ is an applicative morphism.)} \\
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\mathtt{contentsBatch}_{A,X,X} \circ \mathtt{batch}_{A,X})\ t \\
= \ & {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ \mathtt{contentsBatch}_{A,X,X}\ (\mathtt{batch}_{A,X} x))\ t \\
= \ & \text{(Definition of } \mathtt{batch}\text{)} \\
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ \mathtt{contentsBatch}_{A,X,X}\ (\mathsf{P}\ \mathsf{id}_X :\!*\!:\ x))\ t \\
= \ & \text{(Definition of } \mathtt{contentsBatch}\text{)} \\
& {}^{\delta}\mathtt{traverse}_{A,X}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t \\
= \ & \text{(Lemma C.3)} \\
& {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ [x])\ t \\
= \ & \text{(Definition of } \mathtt{contentsBatch}\text{)} \\
& {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ \mathtt{contentsBatch}_{A,Y,Y}\ (\mathsf{P}\ \mathsf{id}_Y :\!*\!:\ x))\ t \\
= \ & \text{(Definition of } \mathtt{batch}\text{)} \\
& {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\lambda x.\ \mathtt{contentsBatch}_{A,Y,Y}\ (\mathtt{batch}_{A,Y}\ x))\ t \\
= \ & {}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{C}(\mathsf{List}(A))}\ (\mathtt{contentsBatch}_{A,Y,Y} \circ \mathtt{batch}_{A,Y})\ t \\
= \ & \text{(Naturality law of traversable functors:} \\
& \text{Lemma C.1 states that } \mathtt{contentsBatch}_{A,Y} \text{ is an applicative morphism.)} \\
& \mathtt{contentsBatch}_{A,Y,T(Y)}\ ({}^{\delta}\mathtt{traverse}_{A,Y}^{\mathsf{Batch}(A)(Y)}\ \mathtt{batch}_{A,Y}\ t) \\
= \ & \text{(Definition of } {}^{\delta}\mathtt{build}\text{)} \\
& \mathtt{contentsBatch}_{A,Y,T(Y)}\ ({}^{\delta}\mathtt{build}_{A,Y}\ t)
\end{aligned}
$$

□

D Trivial proofs of the representational encoding

For any F, $\mathsf{Cotra}(F)$ is a valid functor.[11]

[11] As specified under the representational encoding: see Section 6.3.

Proof For all $A : \mathsf{Set}$, $(n, s, l) : \mathsf{Cotra}(F)(A)$

$$
\begin{aligned}
& \mathtt{map}^{\mathsf{Cotra}(F)}\ (\mathsf{id}_A)\ (n, s, l) \\
= \ & \text{(Definition of } \mathtt{map}^{\mathsf{Cotra}(F)}\text{)} \\
& (n, s, [\mathsf{id}_A\ l_1, \ldots, \mathsf{id}_A\ l_n]) \\
= \ & (n, s, [l_1, \ldots, l_n]) \\
= \ & (n, s, l) \\
= \ & id_{\mathsf{Cotra}(F)(A)}\ (n, s, l)
\end{aligned}
$$

Thus the *Preservation of Identity* law is satisfied.

For all $A, B, C : \mathsf{Set}$, $(n, s, l) : \mathsf{Cotra}(F)(A)$, $f : B \to C$, $g : A \to B$

$$
\begin{aligned}
& \mathtt{map}^{\mathsf{Cotra}(F)}\ f\ \left(\mathtt{map}^{\mathsf{Cotra}(F)}\ g\ (n, s, l)\right) \\
= \ & \text{(Definition of } \mathtt{map}^{\mathsf{Cotra}(F)}\text{)} \\
& \mathtt{map}^{\mathsf{Cotra}(F)}\ f\ (n, s, [g\ l_1, \ldots, g\ l_n]) \\
= \ & \text{(Definition of } \mathtt{map}^{\mathsf{Cotra}(F)}\text{)} \\
& (n, s, [f\ (g\ l_1), \ldots, f\ (g\ l_n)]) \\
= \ & (n, s, [(f \circ g)\ l_1, \ldots, (f \circ g)\ l_n]) \\
= \ & \text{(Definition of } \mathtt{map}^{\mathsf{Cotra}(F)}\text{)} \\
& \mathtt{map}^{\mathsf{Cotra}(F)}\ (f \circ g)\ (n, s, l)
\end{aligned}
$$

Thus the *Preservation of Composition* law is satisfied. □

For any F, ${}^F\omega$ satisfies the *Consistency* law of traversable functors.

Proof For all $A, B : \mathsf{Set}$, $(n, s, l) : \mathsf{Cotra}(F)(A)$, $f : A \to B$

$$
\begin{aligned}
& {}^{F}{}^{\omega}\mathtt{traverse}^{\mathbf{1}}_{A,B}\ f\ (n, s, l) \\
= \ & \text{(Definition of } {}^{F}{}^{\omega}\mathtt{traverse}\text{)} \\
& \mathtt{pure}^{\mathbf{1}}\ (\lambda k.\ (n, s, k)) \circledast^{\mathbf{1}} {}^{n}{}^{\sigma}\mathtt{traverse}^{\mathbf{1}}_{A,B}\ f\ l \\
= \ & \text{(Definition of } {}^{n}{}^{\sigma}\mathtt{traverse}\text{)} \\
& \mathtt{pure}^{\mathbf{1}}\ (\lambda k.\ (n, s, k)) \circledast^{\mathbf{1}} \left((\lambda x_1\ \ldots\ x_n.\ [x_1, \ldots, x_n]) \circledast^{\mathbf{1}} f\ l_1 \circledast^{\mathbf{1}} \ldots \circledast^{\mathbf{1}} f\ l_n\right) \\
= \ & \text{(Definition of } \mathtt{pure} \text{ and } \circledast \text{ for } \mathbf{1}\text{)} \\
& (\lambda k.\ (n, s, k))\ ((\lambda x_1\ \ldots\ x_n.\ [x_1, \ldots, x_n])\ (f\ l_1)\ \ldots\ (f\ l_n)) \\
= \ & (\lambda k.\ (n, s, k))\ [f\ l_1, \ldots, f\ l_n] \\
= \ & (n, s, [f\ l_1, \ldots, f\ l_n])) \\
= \ & \text{(Definition of } \mathtt{map}^{\mathsf{Cotra}(F)}\text{)} \\
& \mathtt{map}^{\mathsf{Cotra}(F)}\ f\ (n, s, l)
\end{aligned}
$$

□

E Optimized implementation of the representational encoding

The following variation of the representational encoding uses a variant of `Batch`, as well as a different representation of a representational triple, in order to address the performance issues and in order to support particular infinite values.[12] However, the implementation requires the use of unsafe operations.

```
import Unsafe.Coerce

data FunList a b t where
  Done :: t -> FunList a b t
  More :: (x -> b -> t) -> a -> FunList a b x -> FunList a b t

deriving instance Functor (FunList a b)

instance Applicative (FunList a b) where
  pure = Done
  Done f <*> fa = fmap f fa
  More ex a r <*> fa = More id a ((flip . ex) <$> r <*> fa)
```

[12] It only supports values with a well-defined leftmost element, such as infinite lists. For example, it does not support infinite trees.

```
flpure :: a -> FunList a b b
flpure a = More (\_ -> id) a (Done ())

data CotraOpt f a where
   Base :: (forall x. f x) -> CotraOpt f a
   Layer :: (forall x. g x -> x -> f x)
         -> a
         -> CotraOpt g a
         -> CotraOpt f a

deriving instance Functor (CotraOpt f)
deriving instance Foldable (CotraOpt f)
deriving instance Traversable (CotraOpt f)

unitOpt :: Traversable t => t a -> CotraOpt t a
unitOpt t = go (traverse flpure t)
  where
    go :: (forall x. FunList a x (u x)) -> CotraOpt u a
    go (Done t') = Base (unsafeCoerce t')
    go (More ex a r) = Layer (unsafeCoerce ex) a (go (unsafeCoerce r))

counitOpt :: CotraOpt f a -> f a
counitOpt (Base t) = t
counitOpt (Layer ex a r) = ex (counitOpt r) a

hoistOpt :: (forall x. f x -> g x)
          -> CotraOpt f a
          -> CotraOpt g a
hoistOpt n (Base t) = Base (n t)
hoistOpt n (Layer ex a r) = Layer (\x h -> n (ex x h)) a r
```

This implementation may still perform poorly when `unitCont` is used with traversable functors whose traversals use `<*>` in a *left-associative* manner. The code repository associated with this thesis shows how this may be addressed within the module `Data.Traversable.Cofree.Optimized` and its associated internal module [24].

F Other potential encodings of cofree traversable functors

The following implementations of `CotraInitial` and `CotraBazaar` represent two potential encodings of the cofree traversable functor functor other than the representational encoding. These are called the *Initial* encoding and the *Bazaar* encoding respectively. Note that the implementation of the Bazaar encoding requires the use of unsafe operations.

```
import Data.Traversable (foldMapDefault)
import Data.Functor.Compose
import Data.Functor.Identity
import Unsafe.Coerce

-- The Initial encoding
data CotraInitial f a where
  Initial :: Traversable t
          => (forall x. t x -> f x)
          -> t a -> CotraInitial f a

deriving instance Functor (CotraInitial f)
deriving instance Foldable (CotraInitial f)
deriving instance Traversable (CotraInitial f)

unitInitial :: Traversable t => t a -> CotraInitial t a
unitInitial = Initial id

counitInitial :: CotraInitial f a -> f a
counitInitial (Initial ex ta) = ex ta

hoistInitial :: (forall x. f x -> g x)
             -> CotraInitial f a -> CotraInitial g a
hoistInitial nt (Initial ex ta) = Initial (nt . ex) ta

-- The Bazaar encoding
newtype CotraBazaar f a =
  Baz { runBaz :: forall g x. Applicative g => (a -> g x) -> g (f x) }
  deriving (Functor)

newtype Bazaar a s t =
  Bazaar { runBazaar :: forall f. Applicative f => (a -> f s) -> f t }
```

```
  deriving (Functor)

instance Applicative (Bazaar a s) where
  pure a = Bazaar $ \_ -> pure a
  ff <*> fa = Bazaar $ \c -> runBazaar ff c <*> runBazaar fa c

sell :: a -> Bazaar a s s
sell a = Bazaar (\c -> c a)

instance Foldable (CotraBazaar f) where
  foldMap = foldMapDefault

instance Traversable (CotraBazaar f) where
  traverse f baz =
    fmap unsafeCoerce . getCompose
      $ runBaz baz (\a -> Compose (sell <$> f a))

unitBazaar :: Traversable t => t a -> CotraBazaar t a
unitBazaar t = Baz (`traverse` t)

counitBazaar :: CotraBazaar t a -> t a
counitBazaar baz = runIdentity (runBaz baz Identity)

hoistBazaar :: (forall x. f x -> g x)
            -> CotraBazaar f a -> CotraBazaar g a
hoistBazaar nt baz = Baz $ \c -> nt <$> runBaz baz c
```