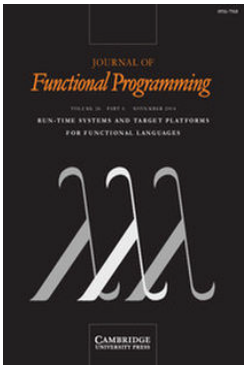
Journal of Functional Programming

http://journals.cambridge.org/JFP



A representation theorem for second-order functionals

MAURO JASKELIOFF and RUSSELL O'CONNOR

Journal of Functional Programming / Volume 25 / 2015 / e13
DOI: 10.1017/S0956796815000088, Published online: 08 September 2015

Link to this article: http://journals.cambridge.org/abstract_S0956796815000088

How to cite this article:
MAURO JASKELIOFF and RUSSELL O'CONNOR (2015). A representation theorem for second-order functionals. Journal of Functional Programming, 25, e13 doi:10.1017/S0956796815000088

JFP **25**, e13, 36 pages, 2015.

doi:10.1017/S0956796815000088

A representation theorem for second-order functionals

MAURO JASKELIOFF

CIFASIS, CONICET, Argentina, FCEIA, Universidad Nacional de Rosario, Argentina
(*e-mail:* mauro@fceia.unr.edu.ar)

RUSSELL O'CONNOR

Google Canada, Kitchener, Ontario, Canada
(*e-mail:* oconnorr@google.com)

Abstract

Representation theorems relate seemingly complex objects to concrete, more tractable ones. In this paper, we take advantage of the abstraction power of category theory and provide a datatype-generic representation theorem. More precisely, we prove a representation theorem for a wide class of second-order functionals which are polymorphic over a class of functors. Types polymorphic over a class of functors are easily representable in languages such as Haskell, but are difficult to analyse and reason about. The concrete representation provided by the theorem is easier to analyse, but it might not be as convenient to implement. Therefore, depending on the task at hand, the change of representation may prove valuable in one direction or the other. We showcase the usefulness of the representation theorem with a range of examples. Concretely, we show how the representation theorem can be used to prove that traversable functors are finitary containers, how coalgebras of a parameterised store comonad relate to very well-behaved lenses, and how algebraic effects might be implemented in a functional language.

1 Introduction

When dealing with a type which uses advanced features of modern type systems such as polymorphism and higher-order types and functions, it is convenient to analyse whether there is another datatype that can represent it, as the alternative representation might be easier to program or to reason about. A simple example of a datatype that might be better understood through a different representation is the type of polymorphic functions $\forall A.\ A \to A$ which, although it involves a function space and a universal quantifier, has only one non-bottom inhabitant: the identity function. Hence, a representation theorem opens the design space for programmers and computer scientists, providing and connecting different views on some construction. When a representation is an isomorphism, we say that it is *exact*, and the change of representation can be done in both directions.

In this article, we will consider second-order functionals that are polymorphic over a class of functors, such as monads or applicative functors. In particular, we

will give a concrete representation for inhabitants of types of the form

$$\forall F.\ (A_1 \to F\ B_1) \to (A_2 \to F\ B_2) \to \cdots \to F\ C$$

Here A_i, B_i, and C are fixed types, and F ranges over an appropriate class of functors. There is a condition on the class of functors which will be made precise during the presentation of the theorem, but basically it amounts to the existence of free constructions. The representation is exact, as it is an isomorphism.

We will express the representation theorem using category theory. Although the knowledge of category theory that is required should be covered by an introductory textbook such as (Awodey, 2006), we introduce the more important concepts in Section 2. The usefulness of the representation theorem (Section 3) is illustrated with a range of examples. Concretely, we show how coalgebras of a specific parameterised comonad are related to very well-behaved lenses (Section 4), and how traversable functors, subjected to certain coherence laws, are exactly the finitary containers (Section 5). Finally, we show how the representation theorem can help when implementing free theories of algebraic effects (Section 6) and discuss related work (Section 7).

There is a long tradition of categorically inspired functional programming (Bird & de Moor, 1997) even though functional programming languages like Haskell usually lack some basic structure such as products or coproducts. The implementation of our results in Haskell, as shown in Sections 4.1 and 6, should be taken simply as categorically-inspired code. Nevertheless, the code could be interpreted to be "morally correct" in a precise technical sense (Danielsson *et al., 2006)*.

1.1 A taste of the representation theorem

In order to get a taste of the representation theorem, we reason informally on a total polymorphic functional language. Consider the type

$$T = \forall F : \mathsf{Functor}.\ (A \to F\ B) \to F\ C\,.$$

What do the inhabitants of this type look like?

The inhabitants of T are functions $h = \lambda g.\, r$. Given that the functor F is universally quantified, the only way of obtaining a result in $F\ C$ is that in the expression r there is an application of the argument g to some $a : A$. This yields something in $F\ B$ rather than the sought $F\ C$, so a function $k : B \to C$ is needed in order to construct a map $F(k) : F\,B \to F\,C$. This informal argument suggests that all inhabitants of T can be built from a pair of an element of A and a function $B \to C$. Hence, it is natural to propose the type $A \times (B \to C)$ as a simpler representation of the inhabitants of type T.

More formally, in order to check that the inhabitants of T are in a one-to-one correspondence with the inhabitants of $A \times (B \to C)$, we want to find an

isomorphism

$$\forall F : \mathsf{Functor}.\ (A \to F\ B) \to F\ C \quad \underset{\varphi^{-1}}{\overset{\varphi}{\cong}} \quad A \times (B \to C).$$

We define φ^{-1} using the procedure described above.

$$\varphi^{-1} : A \times (B \to C) \to \forall F : \mathsf{Functor}.\ (A \to F\ B) \to F\ C$$
$$\varphi^{-1}\ (a, k) = \lambda g.\ F\ (k)\ (g\ a).$$

In order to define φ, notice that $R\ C = A \times (B \to C)$ is functorial on C, with action on morphisms given by $R\ (f)\ (a, g) = (a, f \circ g)$. Hence, we can instantiate a polymorphic function $h : T$ to the functor R and obtain $h_R : (A \to R\ B) \to R\ C$, which amounts to the type $h_R : (A \to (A \times (B \to B))) \to A \times (B \to C)$.

$$\varphi : (\forall F : \mathsf{Functor}.\ (A \to F\ B) \to F\ C) \to A \times (B \to C)$$
$$\varphi\ h = h_R\ (\lambda a.\ (a, id_B))$$

The proof that φ and φ^{-1} are indeed inverses will be given for a Set model in Section 3.

The simple representation $A \times (B \to C)$ is possible due to the restrictive nature of the type T: all we know about F is that it is a functor. What happens when F has more structure?

Consider now the type

$$T' = \forall F : \mathsf{Pointed}.\ (A \to F\ B) \to F\ C.$$

In this case, F ranges over *pointed functors*. That is, F is a functor equipped with a natural transformation $\eta_X : X \to F\ X$. An inhabitant of T' is a function $h = \lambda g.r$, where r can be obtained in the same manner as before, or else by applying the point η_C to a given $c \in C$. Hence, a simpler type representing T' seems to be $(A \times (B \to C)) + C$.

More formally, we want an isomorphism

$$\forall F : \mathsf{Pointed}.\ (A \to F\ B) \to F\ C \quad \underset{\varphi'^{-1}}{\overset{\varphi'}{\cong}} \quad (A \times (B \to C)) + C.$$

The definition of φ'^{-1} is the following.

$$\varphi'^{-1} : (A \times (B \to C)) + C \to \forall F : \mathsf{Pointed}.\ (A \to F\ B) \to F\ C$$
$$\varphi'^{-1}\ (\mathsf{inl}\ (a, k)) = \lambda g.\ F\ (k)\ (g\ a)$$
$$\varphi'^{-1}\ (\mathsf{inr}\ c) \quad = \lambda \_.\ \eta_C\ c$$

In order to define φ', notice that $R'\ C = (A \times (B \to C)) + C$ is a pointed functor on C, with $\eta = \mathsf{inr}$. Hence, we can instantiate a polymorphic function $h : T'$ to the pointed functor R' to obtain $h_{R'} : (A \to R'\ B) \to R'\ C$, or equivalently $h_{R'} : (A \to ((A \times (B \to B)) + B)) \to (A \times (B \to C)) + C$.

$\varphi' : (\forall F : \mathsf{Pointed}.\ (A \to F\ B) \to F\ C) \to (A \times (B \to C)) + C$
$\varphi'\ h = h_{R'}\ (\lambda a.\ \mathsf{inl}\ (a, id_B))$

We can play the same game in the case where the universally quantified functor is an applicative functor.

$$T'' = \forall F : \mathsf{Applicative}.\ (A \to F\ B) \to F\ C.$$

An applicative functor is a pointed functor F equipped with a multiplication operation $\star_{X,Y} : (FX \times FY) \to F(X \times Y)$ natural in X and Y, which is coherent with the point (a precise definition is given in Section 5.1). An inhabitant of T'' is a function $h = \lambda g.r$, where r can be obtained by applying the argument g to n elements of A to obtain an $(F\ B)^n$, then joining the results with the multiplication of the applicative functor to obtain an $F\ (B^n)$, and finally applying a function $B^n \to C$ which takes n elements of B and yields a C.

$$\forall F : \mathsf{Applicative}.\ (A \to F\ B) \to F\ C \quad \underset{\varphi''^{-1}}{\overset{\varphi''}{\cong}} \quad \sum_{n \in \mathbb{N}} (A^n \times (B^n \to C)).$$

The definition of φ''^{-1} is the following.

$\varphi''^{-1} : (\sum_{n \in \mathbb{N}} (A^n \times (B^n \to C))) \to \forall F : \mathsf{Applicative}.\ (A \to F\ B) \to F\ C$
$\varphi''^{-1}\ (n, as, k) = \lambda g.\ F\ (k)\ (\mathsf{collect}_n\ g\ as)$

Here, $\mathsf{collect}_n : \forall F : \mathsf{Applicative}.\ (A \to F\ B) \to A^n \to F\ (B^n)$ is the function that uses the applicative multiplication to collect all the applicative effects, i.e.

$$\mathsf{collect}_n\ h\ (x_1, \ldots, x_n) = h\ x_1 \star \ldots \star h\ x_n.$$

In order to define φ'', notice that $R''\ C = \sum_{n \in \mathbb{N}} (A^n \times (B^n \to C))$ is an applicative functor on C, with $\eta c = (0, *, \lambda x : 1.\ c)$, where $*$ is the sole inhabitant of 1, and the multiplication is given by

$$(n, as, k) \star (n', as', k') = (n + n', as +\!\!+ as', \lambda bs.\ (k\ (\mathsf{take}\ n\ bs), k'\ (\mathsf{drop}\ n\ bs)))$$

Hence, we can instantiate a polymorphic function $h : T''$ to the applicative functor R'' to obtain $h_{R''} : (A \to R''\ B) \to R''\ C$, or equivalently $h_{R''} : (A \to \sum_{n \in \mathbb{N}} (A^n \times (B^n \to B))) \to \sum_{n \in \mathbb{N}} (A^n \times (B^n \to C))$.

$\varphi'' : (\forall F : \mathsf{Applicative}.\ (A \to F\ B) \to F\ C) \to \sum_{n \in \mathbb{N}} (A^n \times (B^n \to C))$
$\varphi''\ h = h_{R''}\ (\lambda a.\ (1, a, id_B))$

We have seen three different isomorphisms which yield concrete representations for second-order functionals which quantify over a certain class of functors (plain functors, pointed functors, and applicative functors, respectively). The construction of each of the three isomorphisms has a similar structure, so it is natural to ask what the common pattern is. In order to answer this question and provide a general representation theorem we will make good use of the power of abstraction of category theory.

2 Categorical preliminaries

A category $\mathscr{C}$ is said to be *locally small* when the collection of morphisms between any two objects X and Y is a proper set. A locally small category is said to be *small* if its collection of objects is a proper set. We denote by $X \xrightarrow{\mathscr{C}} Y$ the (not necessarily small) set of morphisms between X and Y and extend it to a functor $X \xrightarrow{\mathscr{C}} -$ (the covariant Hom functor). When the category is Set (the category of sets and total functions), we will omit the category from the notation and write $X \to Y$. Given two categories $\mathscr{C}$ and $\mathscr{D}$, we will denote by $\mathscr{D}^{\mathscr{C}}$ the category which has as objects functors $F : \mathscr{C} \to \mathscr{D}$ and natural transformations as morphisms. A *subcategory* $\mathscr{D}$ of a category $\mathscr{C}$ consists of a collection of objects and morphisms of $\mathscr{C}$ which is closed under the operations domain, codomain, composition, and identity. When, for every object X and Y of $\mathscr{D}$ subcategory of $\mathscr{C}$, we have $X \xrightarrow{\mathscr{D}} Y = X \xrightarrow{\mathscr{C}} Y$, we say that $\mathscr{D}$ is a *full* subcategory of $\mathscr{C}$.

2.1 The Yoneda lemma

The main result of this article hinges on the following famous result:

Theorem 2.1 (*Yoneda lemma*)
Given a locally small category $\mathscr{C}$, the Yoneda Lemma establishes the following isomorphism

$$(B \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} F \quad \cong \quad F\,B$$

natural in object $B : \mathscr{C}$ and functor $F : \mathscr{C} \to \mathsf{Set}$.

That is, the set $F\,B$ is naturally isomorphic to the set of natural transformations between the functor $(B \xrightarrow{\mathscr{C}} -)$ and the functor F.

Naturality in B means that given any morphism $h : B \to C$, the following diagram commutes

$$\begin{array}{ccc} ((B \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} F) & \xrightarrow{\cong} & FB \\ {\scriptstyle (h \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} F}\downarrow & & \downarrow{\scriptstyle Fh} \\ ((C \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} F) & \xrightarrow[\cong]{} & FC \end{array}$$

Naturality in F means that given any natural transformation $\alpha : F \to G$, the following diagram commutes

$$\begin{array}{ccc} ((B \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} F) & \xrightarrow{\cong} & FB \\ {\scriptstyle (B \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} \alpha}\downarrow & & \downarrow{\scriptstyle \alpha_B} \\ ((B \xrightarrow{\mathscr{C}} -) \xrightarrow{\mathsf{Set}^{\mathscr{C}}} G) & \xrightarrow[\cong]{} & GB \end{array}$$

The construction of the isomorphism is as follows:

- Given a natural transformation $\alpha : (B \xrightarrow{\mathscr{C}} -) \to F$, its component at B is a function $\alpha_B : (B \xrightarrow{\mathscr{C}} B) \to FB$. Then, the corresponding element of $F\,B$ is $\alpha_B(id_B)$.
- For the other direction, given $x : F\,B$, we construct a natural transformation $\alpha : (B \xrightarrow{\mathscr{C}} -) \to F$ in the following manner: the component at each object C, namely $\alpha_C : (B \xrightarrow{\mathscr{C}} C) \to FC$ is given by $\lambda f : B \to C.\, F(f)(x)$.

We leave as an exercise for the reader to check that this construction indeed yields a natural isomorphism.

In order to make the relation between the programs and the category theory more evident, it is convenient to express the Yoneda lemma in end form:

$$\int_{X \in \mathscr{C}} (B \xrightarrow{\mathscr{C}} X) \to F\,X \quad \cong \quad F\,B \tag{2.1}$$

The intuition is that an end corresponds to a universal quantification in a programming language (Bainbridge *et al.*, 1990), and therefore the above isomorphism could be understood as stating an isomorphism of types:

$$\forall X.\, (B \to X) \to F\,X \quad \cong \quad FB$$

Hence, functional programmers not used to categorical ends can get the intuitive meaning just by replacing in their minds ends by universal quantifiers. The complete definition of end can be found in Appendix 7. More details can be found in the standard reference (Mac Lane, 1971).

A simple application of the Yoneda lemma which will be used in the next section is the following proposition.

Proposition 2.2
Consider an endofunctor $F : \mathsf{Set} \to \mathsf{Set}$, and the functor $R : \mathsf{Set} \times \mathsf{Set}^{\mathrm{op}} \times \mathsf{Set} \to \mathsf{Set}$ defined as $R\,(A, B, X) = A \times (B \to X)$, $R\,(f, g, h)(a, x) = (f\,a, g \circ x \circ h)$, where we write $R_{A,B}X$ for $R\,(A, B, X)$. Then

$$A \to F\,B \;\cong\; R_{A,B} \xrightarrow{\mathsf{Set}^{\mathsf{Set}}} F \tag{2.2}$$

Proof

$$\begin{array}{ll}
 & A \to F\,B \\
\cong & \quad \{\ \text{Yoneda}\ \} \\
 & A \to \int_X ((B \to X) \ \to \ F\,X) \\
\cong & \quad \{\ \text{Hom functors preserve ends (Remark A.4)}\ \} \\
 & \int_X A \to ((B \to X) \ \to \ F\,X) \\
\cong & \quad \{\ \text{Adjoints (currying)}\ \} \\
 & \int_X A \times (B \to X) \ \to \ F\,X \\
\cong & \quad \{\ \text{Definition of } R_{A,B}\ \} \\
 & \int_X R_{A,B}\,X \ \to \ F\,X \\
\cong & \quad \{\ \text{Natural transformations as ends}\ \} \\
 & R_{A,B} \xrightarrow{\mathsf{Set}^{\mathsf{Set}}} F
\end{array}$$

More concretely, the isomorphism is witnessed by the following functions

$$\begin{array}{lll}
\alpha_F & : (A \to F\ B) \to R_{A,B} \xrightarrow{\mathsf{Set}^{\mathsf{Set}}} F \\
\alpha_F(f) & = \tau \quad \text{where} \quad \tau_X & : A \times (B \to X) \to F\,X \\
 & \qquad\qquad\quad \tau_X(a,g) = F(g)(f(a)) & \\
\alpha_F^{-1} & : (R_{A,B} \xrightarrow{\mathsf{Set}^{\mathsf{Set}}} F) \to A \to F\ B & \\
\alpha_F^{-1}(h) & = \lambda a.\ h_B\,(a, id_B). &
\end{array}$$

This isomorphism is natural in A and B. □

2.2 Adjunctions

An adjunction is a relation between two categories which is weaker than isomorphism of categories.

Definition 2.3 (*Adjunction*)
Given categories $\mathscr{C}$ and $\mathscr{D}$, functors $L : \mathscr{C} \to \mathscr{D}$ and $R : \mathscr{D} \to \mathscr{C}$, an *adjunction* is given by a tuple $(L, R, \lfloor - \rfloor, \lceil - \rceil)$, where $\lfloor - \rfloor$ and $\lceil - \rceil$ are the components of the following isomorphism:

$$\lfloor - \rfloor : L\,C \xrightarrow{\mathscr{D}} D \quad \cong \quad C \xrightarrow{\mathscr{C}} R\,D : \lceil - \rceil \tag{2.3}$$

which is natural in $C \in \mathscr{C}$ and $D \in \mathscr{D}$. That is, for $f : L\,C \to D$ and $g : C \to R\,D$ we have

$$\lfloor f \rfloor = g \quad \Leftrightarrow \quad f = \lceil g \rceil \tag{2.4}$$

The components of the isomorphism $\lfloor - \rfloor$ and $\lceil - \rceil$ are called *adjuncts*. That the isomorphism is natural means that for any $C, C' \in \mathscr{C}$; $D, D' \in \mathscr{D}$; $h : C' \to C$; $k : D \to D'$; $f : L\,C \to D$; and $g : C \to R\,D$, the following equations hold:

$$R\,k \circ \lfloor f \rfloor \circ h = \lfloor k \circ f \circ L\,h \rfloor \tag{2.5}$$

$$k \circ \lceil g \rceil \circ L\,h = \lceil R\,k \circ g \circ h \rceil \tag{2.6}$$

We indicate the categories involved in an adjunction by writing $\mathscr{C} \rightharpoonup \mathscr{D}$ (note the asymmetry in the notation), and often leave the components of the isomorphism implicit and simply write $L \dashv R$.

The *unit* η and *counit* ε of the adjunction are defined as:

$$\eta = \lfloor id \rfloor \qquad \varepsilon = \lceil id \rceil ; \tag{2.7}$$

The adjuncts can be characterised in terms of the unit and counit:

$$\lfloor f \rfloor = R\,f \circ \eta \qquad \lceil g \rceil = \varepsilon \circ L\,g. \tag{2.8}$$

For more details, see (Mac Lane, 1971; Awodey, 2006).

3 A representation theorem for second-order functionals

Consider a small subcategory $\mathscr{F}$ of $\mathsf{Set}^{\mathsf{Set}}$, the category of endofunctors on Set.[1] By Yoneda,

$$\int_{F\in\mathscr{F}} (G \xrightarrow{\mathscr{F}} F) \to H\,F \quad \cong \quad H\,G \tag{3.1}$$

Note that G is any functor in $\mathscr{F}$ and H is any functor $\mathscr{F} \to \mathsf{Set}$. In particular, given a set X, we obtain the functor $(-X) : \mathscr{F} \to \mathsf{Set}$ that applies a functor in $\mathscr{F}$ to X. That is, the action on objects is $F \mapsto F\,X$. The above equation, specialised to $(-X)$ is

$$\forall G \in \mathscr{F}. \qquad \int_{F} (G \xrightarrow{\mathscr{F}} F) \to F\,X \quad \cong \quad G\,X \tag{3.2}$$

For example, let $R_{A,B}\,X = A \times (B \to X)$ as in Proposition 2.2, and let $\mathscr{E}$ be a small full subcategory of $\mathsf{Set}^{\mathsf{Set}}$ such that $R_{A,B} \in \mathscr{E}$.

Then, we calculate

$$\begin{array}{ll} & \int_{F\in\mathscr{E}} (A \to F\,B) \to F\,X \\ \cong & \quad \{\ \text{Equation (2.2)}\ \} \\ & \int_{F\in\mathscr{E}} (R_{A,B} \xrightarrow{\mathscr{E}} F) \to F\,X \\ \cong & \quad \{\ \text{Equation (3.2)}\ \} \\ & R_{A,B}\,X. \end{array}$$

That is, we have proven that

$$\int_{F} (A \to F\,B) \to F\,X \;\cong\; R_{A,B}\,X \tag{3.3}$$

This isomorphism provides a justification for the first isomorphism of the introduction, namely

$$\forall F : \mathsf{Functor}.\ (A \to F\ B) \to F\ C \quad \cong \quad A \times (B \to C)$$

3.1 Unary representation theorem

Let us now consider categories of endofunctors that carry some structure. For example, a category $\mathscr{F}$ may be the category of monads and monad morphisms, or the category of applicative functors and applicative morphisms. Then we have a functor that forgets the extra structure and yields a plain functor. For example, the forgetful functor $U : \mathscr{M}on \to \mathscr{E}$ maps a monad $(T, \mu, \eta) \in \mathscr{M}on$ to the endofunctor T, forgetting that the functor has a monad structure given by μ and η. It often happens that this forgetful functor has a left adjoint $(-)^* : \mathscr{E} \to \mathscr{F}$. Such an adjoint takes an arbitrary endofunctor F and constructs the *free* structure on F. For example, in

[1] We are interested in functors representable in a programming language, such as realisable functors (Bainbridge *et al.*, 1990; Reynolds & Plotkin, 1993). Therefore, it is reasonable to assume smallness.

the monad case, F^* would be the free monad on F. The adjunction establishes the following natural isomorphism between morphisms in $\mathscr{F}$ and $\mathscr{E}$:

$$E^* \xrightarrow{\mathscr{F}} F \quad \cong \quad E \xrightarrow{\mathscr{E}} UF \tag{3.4}$$

In this situation we have the following representation theorem.

Theorem 3.1 (*Unary representation*)
Consider an adjunction $((-)^*, U, \lfloor - \rfloor, \lceil - \rceil) : \mathscr{E} \rightharpoonup \mathscr{F}$, where $\mathscr{F}$ is small and $\mathscr{E}$ is a full subcategory of $\mathsf{Set}^{\mathsf{Set}}$ such that the family of functors $R_{A,B}\, X = A \times (B \to X)$ is in $\mathscr{E}$. Then, we have the following isomorphism natural in A, B, and X.

$$\int_F (A \to UF\, B) \to UF\, X \quad \cong \quad UR^*_{A,B}\, X \tag{3.5}$$

Proof

$$\begin{array}{ll} & \int_F (A \to UF\, B) \to UF\, X \\ \cong & \quad \{ \text{ Equation (2.2) } \} \\ & \int_F (R_{A,B} \xrightarrow{\mathscr{E}} UF) \to UF\, X \\ \cong & \quad \{ \ (-)^* \text{ is left adjoint to } U \text{ (see Eq. 3.4) } \} \\ & \int_F (R^*_{A,B} \xrightarrow{\mathscr{F}} F) \to UF\, X \\ \cong & \quad \{ \text{ Yoneda } \} \\ & UR^*_{A,B}\, X \end{array}$$

Every isomorphism in the proof is natural in X, the first one is natural in A and B, and the last two are natural in $R_{A,B}$. Therefore, the resulting isomorphism is also natural in A and B. □

Since the free pointed functor on F is simply $F^* = F + Id$, and the free applicative functor on small functors such as $R_{A,B}$ exists (Capriotti & Kaposi, 2014), this theorem explains all the isomorphisms in the introduction. Furthermore, it explains the structure of the representation functor (it is the free construction on $R_{A,B}$) and what is more, it tells us that the isomorphism is natural.

For the sake of concreteness, we present the functions witnessing the isomorphism in the theorem:

$$\begin{array}{ll} \varphi & : (\int_F (A \to UF\, B) \to UF\, X) \to UR^*_{A,B}\, X \\ \varphi(h) & = h_{R^*_{A,B}} (\alpha^{-1}_{UR^*_{A,B}} (\eta_{R_{A,B}})) \end{array}$$

$$\begin{array}{lll} \varphi^{-1} & : UR^*_{A,B}\, X \to \int_F (A \to UF\, B) \to UF\, X & \\ \varphi^{-1}(r) = \tau & \text{where} \quad \tau_F & : (A \to UF\, B) \to UF\, X \\ & & \tau_F(g) = (U \lceil \alpha_{UF}(g) \rceil_X)(r) \end{array}$$

Here, η is the unit of the adjunction, and α is the isomorphism in Proposition 2.2.

3.2 Generalisation to many functional arguments

Let us consider functionals of the form

$$\forall F.\ (A_1 \to F\ B_1) \to \cdots \to (A_n \to F\ B_n) \to F\ X\,.$$

The representation theorem, Theorem 3.1, can be easily generalised to include the above functional.

Theorem 3.2 (*N-ary representation*)
Consider an adjunction $((-)^*, U, \lfloor - \rfloor, \lceil - \rceil) : \mathscr{E} \rightharpoonup \mathscr{F}$, where $\mathscr{F}$ is small and $\mathscr{E}$ is a full subcategory of $\mathsf{Set}^{\mathsf{Set}}$ closed under coproducts such that the family of functors $R_{A,B}\, X = A \times (B \to X)$ is in $\mathscr{E}$. Let A_i, B_i be sets for $i \in \{1, \ldots, n\}, n \in \mathbb{N}$. Then, we have the following isomorphism

$$\int_F \left(\prod_i (A_i \to UF\, B_i) \right) \to UF\, X \quad \cong \quad U \left(\sum_i R_{A_i,B_i} \right)^* X \tag{3.6}$$

natural in A_i, B_i, and X.

Proof
The proof follows the same path as the one in Theorem 3.1, except that now we use the isomorphism $(A \to C) \times (B \to C) \cong (A + B) \to C$ that results from the universal property of coproducts. More precisely, the proof is as follows:

$$\begin{array}{ll} & \int_F (\prod_i (A_i \to UF\, B_i)) \to UF\, X \\ \cong & \quad \{ \text{ Equation (2.2) } \} \\ & \int_F (\prod_i (R_{A_i,B_i} \xrightarrow{\mathscr{E}} UF)) \to UF\, X \\ \cong & \quad \{ \text{ Coproducts } \} \\ & \int_F (\sum_i R_{A_i,B_i} \xrightarrow{\mathscr{E}} UF) \to UF\, X \\ \cong & \quad \{ \ (-)^* \text{ is left adjoint to } U \text{ (see Equation (3.4)) } \} \\ & \int_F ((\sum_i R_{A_i,B_i})^* \xrightarrow{\mathscr{F}} F) \to UF\, X \\ \cong & \quad \{ \text{ Yoneda } \} \\ & U (\sum_i R_{A_i,B_i})^*\, X \end{array}$$

Naturality follows from naturality of its component isomorphisms. □

4 Parameterised comonads and very well-behaved lenses

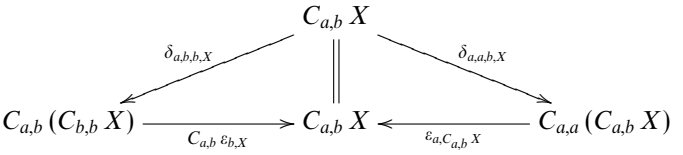
The functor $R_{A,B}\, X = A \times (B \to X)$ plays a fundamental role in Theorems 3.1 and 3.2. Such a functor R has the structure of a parameterised comonad (Atkey, 2009a; Atkey, 2009b) and is sometimes called a parameterised store comonad. As a first application of the representation theorem we analyse the relation between coalgebras for this parameterised comonad and very well-behaved lenses (Foster *et al., 2007)*.

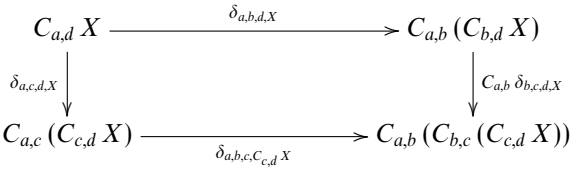
Definition 4.1 (*Parameterised comonad*)
Fix a category $\mathscr{P}$ of parameters. A $\mathscr{P}$-parameterised comonad on a category $\mathscr{C}$ is a triple (C, ε, δ), where:

- C is a functor $\mathscr{P} \times \mathscr{P}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{C}$. We write the parameters as (usually lowercase) subindexes. That is, $C_{a,b}\, X = C(a, b, X)$.
- the *counit* ε is a family of morphisms $\varepsilon_{a,X} : C_{a,a}\, X \to X$ which is natural in X and dinatural in a (dinaturality is defined in Appendix 7, Definition A.1),

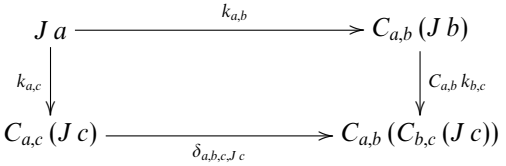
- the *comultiplication* δ is a family of morphisms $\delta_{a,b,c,X} : C_{a,c}\, X \to C_{a,b}\,(C_{b,c}\, X)$ natural in a, c and X and dinatural in b.

These must make the following diagrams commute:

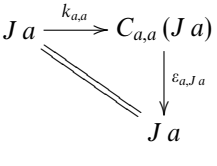




Definition 4.2 (*Coalgebra for a parameterised comonad*)
Let C be a $\mathscr{P}$-parameterised comonad on $\mathscr{C}$. Then a C-coalgebra is a pair (J, k) of a functor $J : \mathscr{P} \to \mathscr{C}$, and a family $k_{a,b} : J\, a \to C_{a,b}\,(J\, b)$, natural in a and dinatural in b, such that the following diagrams commute:



comultiplication-coalgebra law



counit-coalgebra law

The definitions of parameterised comonad and of coalgebra for a parameterised comonad are dualisations of the ones for monads found in Atkey (2009a).

Example 4.3
The functor $R_{a,b}\, X = a \times (b \to X)$ is a parameterised comonad, with the following counit and comultiplication:

$$\begin{array}{lcl} \varepsilon_{a,X} & : & R_{a,a}\, X \to X \\ \varepsilon_{a,X}\ (x, f) & = & f\, x \end{array}$$

$$\begin{array}{lcl} \delta_{a,b,c,X} & : & R_{a,c}\, X \to R_{a,b}\,(R_{b,c}\, X) \\ \delta_{a,b,c,X}\ (x, f) & = & (x, \lambda y.\,(y, f)). \end{array}$$

Example 4.4
Given a functor $K : \mathscr{P} \to Set$, define the functor $R^{(K)}_{a,b}\, X = Ka \times (Kb \to X) : \mathscr{P} \times \mathscr{P}^{\mathrm{op}} \times Set \to Set$. For every functor K, $R^{(K)}$ is a parameterised comonad, with

the following counit and comultiplication:

$$\begin{array}{lcl} \varepsilon_{a,X} & : & R^{(K)}_{a,a}\, X \to X \\ \varepsilon_{a,X}\ (x,f) & = & fx \end{array}$$

$$\begin{array}{lcl} \delta_{a,b,c,X} & : & R^{(K)}_{a,c}\, X \to R^{(K)}_{a,b}\, (R^{(K)}_{b,c}\, X) \\ \delta_{a,b,c,X}\ (x,f) & = & (x, \lambda y.\, (y,f)) \end{array}$$

The parameterised comonad R from Example 4.3 is the same as $R^{(I)}$ where I is the identity functor.

The proposition below shows how the comonadic structure of $R^{(K)}$ interacts nicely with the isomorphism of Proposition 2.2.

Proposition 4.5
Let $F, G : Set \to Set$, $f : a \to Fb$, and $g : b \to Gc$, then the following equations hold.

a. $\varepsilon_{a,X} = \alpha_I(id_{Ka})_X \quad : R^{(K)}_{a,a,X} \to X$
b. $(\alpha_F(f) \cdot \alpha_G(g))_X \circ \delta_{a,b,c,X} = \alpha_{F\cdot G}(Fg \circ f)_X \quad : R^{(K)}_{a,c}\, X \to F(G\, X)$

where $F \cdot G$ is functor composition and where $\alpha \cdot \beta$ is the horizontal composition of natural transformations. That is, given natural transformations $\alpha : F \to G$, and $\beta : F' \to G'$, horizontal composition $\alpha \cdot \beta : F \cdot F' \to G \cdot G'$ is given by $\alpha \cdot \beta = G(\beta) \circ \alpha_{F'}$.

Example 4.6
The pair $((\times C), k)$ is an R-coalgebra with

$$\begin{array}{lcl} k_{a,b} & : & a \times C \to R_{a,b}(b \times C) \\ k_{a,b}\ (a,c) & = & (a, \lambda b.\, (b,c)) \end{array}$$

Coalgebras of $R^{(K)}$ play an important role in functional programming as they are precisely the type of very well-behaved lenses, hereafter called lenses (Foster *et al., 2007)*. A lens provides access to a component B inside another type A. More formally a lens from A to B is an isomorphism $A \cong B \times C$ for some residual type C. A lens from A to B is most easily implemented by a pair of appropriately typed getter and setter functions

$$\begin{array}{lcl} get & : & A \to B \\ set & : & A \times B \to A \end{array}$$

satisfying three laws[2]

$$\begin{array}{rcl} set(x, get(x)) & = & x \\ get(set(x,y)) & = & y \\ set(set(x,y_1),y_2) & = & set(x,y_2) \end{array}$$

More generally, given two functors $J : \mathscr{P} \to Set$ and $K : \mathscr{P} \to Set$, we can form a parameterised lens from J to K with a family of getters and setters

$$\begin{array}{lcl} get_a & : & Ja \to Ka \\ set_{a,b} & : & Ja \times Kb \to Jb \end{array}$$

[2] In Foster *et al.,* (2007), the less well-behaved lenses do not satisfy all three laws.

satisfying the same three laws, and with *get* being natural in a and *set* being natural in b. By some simple algebra we see that the type of lenses is isomorphic to the type of coalgebras of the parameterised comonad $R^{(K)}$.

$$(Ja \to Ka) \times (Ja \times Kb \to Jb) \quad \cong \quad Ja \to R^{(K)}_{a,b}(Jb).$$

Furthermore the coalgebra laws are satisfied if and only if the corresponding lens laws are satisfied (O'Connor, 2010; Gibbons & Johnson, 2012). For instance, the coalgebra given in Example 4.6 is a parameterised lens into the first component of a pair.

Using the representation theorem and some simple manipulations we can define a third way to represent a parameterised lens from J to K. The so-called Van Laarhoven representation (Van Laarhoven, 2009a; O'Connor, 2011) is defined by a family of ends

$$\int_{F:\mathscr{E}} (Ka \to F(Kb)) \to Ja \to F(Jb)$$

that is natural in the sense that given two arrows from $\mathscr{P}$, $p : a \to a'$ and $q : b \to b'$, and given $f : Ka' \to F(Kb)$ for some $F : \mathscr{E}$ then

$$F(Jq) \circ v_{a',b,F}(f) \circ Jp = v_{a,b',F}(F(Kq) \circ f \circ Kp).$$

The corresponding laws for the Van Laarhoven representation of lenses are

- the linearity law
  For all $f : Ka \to F(Kb)$ and $g : Kb \to G(Kc)$,

$$v_{a,c,F\cdot G}(Fg \circ f) = Fv_{b,c,G}(g) \circ v_{a,b,F}(f)$$

- and the unity law

$$v_{a,a,I}(id_{Ka}) = id_{Ja}.$$

The following theorem proves that the coalgebra representation and Van Laarhoven representation of parameterised lenses are equivalent.

Theorem 4.7 (*Lens representation*)
Given $\mathscr{E}$, a small full subcategory of Set^{Set} and given functors $J, K : \mathscr{P} \to Set$, then the families $k_{a,b} : Ja \to R^{(K)}_{a,b}(Jb)$ which form $R^{(K)}$-coalgebras (J, k) are isomorphic to the families of ends

$$\int_{F:\mathscr{E}} (Ka \to F(Kb)) \to Ja \to F(Jb)$$

which satisfy the linearity and unity laws.

Proof
First, we prove the isomorphism of families without regard to the laws

$$
\begin{array}{ll}
 & Ja \to R^{(K)}_{a,b}(Jb) \\
\cong & \quad \{ \text{ definition of } R^{(K)} \ \} \\
 & Ja \to R_{Ka,Kb}(Jb) \\
\cong & \quad \{ \text{ Equation } 3.3 \ \} \\
 & Ja \to \int_F (Ka \to F(Kb)) \to F(Jb) \\
\cong & \quad \{ \text{ Hom functors preserve ends (Remark A.4)} \ \} \\
 & \int_F Ja \to (Ka \to F(Kb)) \to F(Jb) \\
\cong & \quad \{ \text{ Swap argument } \} \\
 & \int_F (Ka \to F(Kb)) \to Ja \to F(Jb)
\end{array}
$$

This isomorphism is witnessed by the following functions:

$$
\begin{array}{ll}
\gamma & : (\int_F (Ka \to F(Kb)) \to Ja \to F(Jb)) \to (Ja \to R^{(K)}_{a,b}(Jb)) \\
\gamma(h) & = h_{R^{(K)}_{a,b}}(\alpha^{-1}_{R^{(K)}_{a,b}}(id))
\end{array}
$$

$$
\begin{array}{lll}
\gamma^{-1} & : (Ja \to R^{(K)}_{a,b}(Jb)) \to \int_F (Ka \to F(Kb)) \to (Ja \to F(Jb)) & \\
\gamma^{-1}(k) = \tau & \text{where} \quad \tau_F & : (Ka \to F(Kb)) \to Ja \to F(Jb) \\
 & & \tau_F(g) = \alpha_F(g)_{Jb} \circ k
\end{array}
$$

In order to prove that the laws of coalgebras for parameterised comonads correspond to unity and linearity, we first prove two technical lemmas.

Lemma 4.8

$$\gamma^{-1}(k_{a,c})_{F \cdot G}(Fg \circ f) = (\alpha_F(f) \cdot \alpha_G(g))_{Jc} \circ \delta_{a,b,c,Jc} \circ k_{a,c}$$

Proof
This follows from Proposition 4.5(b). □

Lemma 4.9

$$F(\gamma^{-1}(k_{b,c})_G(g)) \circ \gamma^{-1}(k_{a,b})_F(f) = (\alpha_F(f) \cdot \alpha_G(g))_{Jc} \circ R^{(K)}_{a,b}(k_{b,c}) \circ k_{a,b}$$

Proof
This follows from the definition of γ^{-1} and properties of functors and natural transformations. □

Generalised versions of Lemmas 4.8 and 4.9 appear with detailed proofs in Appendix 7, Lemmas A.8 and A.9.

By the previous two lemmas, to prove that the comultiplication-coalgebra law is equivalent to the linearity law it suffices to prove the following:

$$
\begin{array}{rcl}
R^{(K)}_{a,b}(k_{b,c}) \circ k_{a,b} & = & \delta_{a,b,c,Jc} \circ k_{a,c} \\
 & \Longleftrightarrow & \\
\forall F, G, f, g.(\alpha_F(f) \cdot \alpha_G(g)) \circ R^{(K)}_{a,b}(k_{b,c}) \circ k_{a,b} & = & (\alpha_F(f) \cdot \alpha_G(g)) \circ \delta_{a,b,c,Jc} \circ k_{a,c}
\end{array}
$$

The forward implication is clear. To prove the reverse implication take $F = R^{(K)}_{a,b}$ and $f = \alpha^{-1}_{R^{(K)}_{a,b}}(id)_{Jb}$. Also take $G = R^{(K)}_{b,c}$ and $g = \alpha^{-1}_{R^{(K)}_{b,c}}(id)_{Jc}$. Then $\alpha_F(f) = id$ and $\alpha_G(g) = id$. Therefore, $\alpha_F(f) \cdot \alpha_G(g) = id$ and the result follows.

To prove that the counit-coalgebra law is equivalent to the unity law it suffices to prove that $\varepsilon_{a,Ja} \circ k_{a,a} = \gamma^{-1}(k_{a,a})_I(id)$.

$$
\begin{aligned}
& \gamma^{-1}(k_{a,a})_I(id) \\
= & \quad \{ \text{ definition of } \gamma^{-1} \ \} \\
& \alpha_I(id)_{Ja} \circ k_{a,a} \\
= & \quad \{ \text{ Proposition 4.5(a) } \} \\
& \varepsilon_{a,Ja} \circ k_{a,a}
\end{aligned}
$$

□

The previous theorem can be generalised to the case where we have an adjunction.

Theorem 4.10 (*Generalised lens representation*)
Let $\mathscr{E}$ and $\mathscr{F}$ be two small categories of Set-endofunctors, such that $\mathscr{E}$ and $\mathscr{F}$ are (strict) monoidal with respect to the identity functor I and functor composition $- \cdot -$, and $\mathscr{E}$ is a full subcategory. Let $(-)^* \dashv U : \mathscr{E} \rightharpoonup \mathscr{F}$, be an adjunction between them, such that U is strict monoidal. Then

1. $UR^{(K)*}$ is a parameterised comonad.
2. Given functors $J, K : \mathscr{P} \to Set$, then the family $k_{a,b} : Ja \to UR^{(K)*}_{a,b}(Jb)$ which form the $UR^{(K)*}$-coalgebras (J,k) are isomorphic to the family of ends

$$\int_{F:\mathscr{F}} (Ka \to UF(Kb)) \to Ja \to UF(Jb)$$

which satisfy the linearity and unity laws.

Proof
See Appendix 7, Proposition A.7. □

By considering the identity adjunction between $\mathscr{E}$ and itself, Theorem 4.7 can be recovered from this generalised version.

4.1 Implementing lenses in Haskell

The Lens representation theorem demonstrates that the coalgebra representation of lenses and the Van Laarhoven representation are isomorphic. Both representations can be implemented in Haskell.

```
  -- Parameterised store comonad
data PStore a b x = PStore (b → x) a

  -- Coalgebra representation of lenses
newtype KLens ja jb ka kb = KLens (ja → PStore ka kb jb)

  -- Van Laarhoven representation of lenses
type VLens ja jb ka kb = ∀f. Functor f ⇒ (ka → f kb) → ja → f jb
```

There are a few observations to make about this Haskell code. Firstly, neither the coalgebra laws nor the linearity and unity laws of the Van Laarhoven representation can be enforced by Haskell's type system, as it often happens when implementing algebraic structures such as monoids or monads. We have accordingly omitted

writing out the parameterised comonad operations of *PStore*. Secondly, rather than taking J and K as parameters, we take source and target types for each functor. By not explicitly using functors as parameters, we avoid **newtype** wrapping and unwrapping functions that would otherwise be needed. Consider the example of building a lens to access the first component of a pair.

```
fstLens          :: VLens a b (a, y) (b, y)
fstLens f (a, y) = (λb → (b, y)) `fmap` (f a)
```

Above we are constructing a *VLens* value but the argument applies equally well to a *KLens* value. The pair type is functorial in two arguments. For *fstLens*, we care about pairs being functorial with respect to the first position. If we were required to pass a J functor explicitly to *VLens*, we would need to add a wrapper around (a, b) to make it explicitly a functor of the first position. Furthermore, we are implicitly using the identity functor for the K functor. If we were required to pass a K functor explicitly to *VLens*, we would have to wrap and unwrap the *Identity* functor in Haskell in order to use the lens. Fortunately, all lens functionality can be implemented without explicitly mentioning the functor parameters.

The third thing to note about the *VLens* formulation is that we use a type alias rather than a **newtype**. This allows us to compose a lens of type *VLens ja jb ka kb* and another lens of type *VLens ka kb la lb* by simply using the standard function composition operator. There is another advantage that the type alias gives us, which we will see later.

The isomorphism between the two representations can be written out explicitly in Haskell.

```
instance Functor (PStore i j) where
  fmap f (PStore h x) = PStore (f ∘ h) x

kLens2VLens      :: KLens ja jb ka kb → VLens ja jb ka kb
kLens2VLens k f  = (λ(PStore h x) → h `fmap` f x) ∘ k

vLens2KLens      :: VLens ja jb ka kb → KLens ja jb ka kb
vLens2KLens v    = v (PStore id)
```

The generalised lens representation theorem gives us pairs of representations of various lens derivatives. Using pointed functors, i.e. using the free pointed functor generated by *PStore* in the case of the coalgebra representation, or quantifying over pointed functors in the case of the Van Laarhoven representation, gives us the notion of a partial lens (O'Connor *et al.*, 2013), also known as an affine traversal (Kmett, 2013).[3]

```
data FreePointedPStore a b x = Unit x
                             | FreePointedPStore (b → x) a
  -- coalgebra representation of partial lenses
newtype KPartialLens ja jb ka kb = KPartialLens (ja → FreePointedPStore ka kb jb)
```

[3] An affine traversal from A to B is so-called because it specifies an isomorphism between A and $F\ B$ for some affine container F, i.e. for some functor F where $F\ X \cong C_1 \times X + C_2$.

```
class Functor f ⇒ Pointed f where
  point :: a → f a
  -- Van Laarhoven representation of partial lenses
type VPartialLens ja jb ka kb = ∀f. Pointed f ⇒ (ka → f kb) → ja → f jb
```

A partial lens provides a reference to 0 or 1 occurrences of K within J. If we instead use applicative functors (Section 5.1), we get a reference to a sequence of 0 or more occurrences of K within J. This lens derivative is called a traversal.

```
data FreeApplicativePStore a b x =
    Unit x
  | FreeApplicativePStore (FreeApplicativePStore a b (b → x)) a
  -- coalgebra representation of traversals
newtype KTraversal ja jb ka kb = KTraversal (ja → FreeApplicativePStore ka kb jb)
  -- Van Laarhoven representation of traversals
type VTraversal ja jb ka kb = ∀f. Applicative f ⇒ (ka → f kb) → ja → f jb
```

The Haskell implementation of the isomorphism between *KPartialLens* and *VPartialLens* and the isomorphism between *KTraversal* and *VTraversal* is left as an exercise to the interested reader.

The second advantage of using a type synonym for the Van Laarhoven representation is that values of type *VLens* are values of type *VPartialLens* and *VTraversal*, while the values of type *KLens* need to be explicitly converted to *KPartialLens* and *KTraversal*. If Haskell's standard library were modified such that Pointed was a super class of Applicative, then values of type *VPartialLens* would be of type *VTraversal* as well.

5 The finiteness of traversals

In this section, we show another application of the representation theorem. We show that traversable functors are exactly the finitary containers. We first introduce the relevant definitions and then provide the proof.

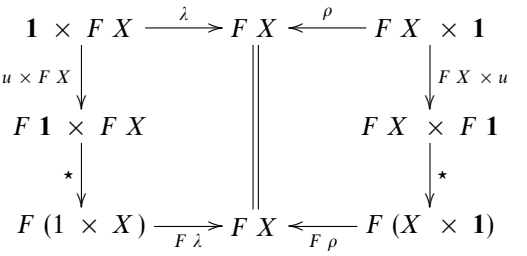
5.1 Applicative functors

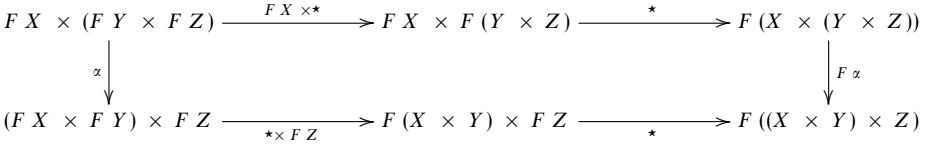
The cartesian product gives the category Set a monoidal structure $(\mathsf{Set}, \times, \mathbf{1}, \alpha, \lambda, \rho)$, where $\alpha_{X,Y,Z} : X \times (Y \times Z) \to (X \times Y) \times Z$, $\lambda_X : \mathbf{1} \times X \to X$, and $\rho_X : X \times \mathbf{1} \to X$ are natural isomorphisms expressing associativity of the product, left unit and right unit, respectively.

Definition 5.1 (*Applicative functor*)
An *applicative functor* is a functor $F : \mathsf{Set} \to \mathsf{Set}$ which is strong lax monoidal with respect to this monoidal structure. That is, it is equipped with a map and a natural transformation

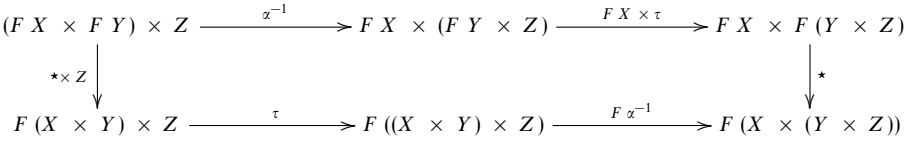
$$
\begin{array}{lcll}
u & : & \mathbf{1} \to F\,\mathbf{1} & \text{(monoidal unit)} \\
\star_{X,Y} & : & F\,X \times F\,Y \to F\,(X \times Y) & \text{(monoidal action)}
\end{array}
$$

such that



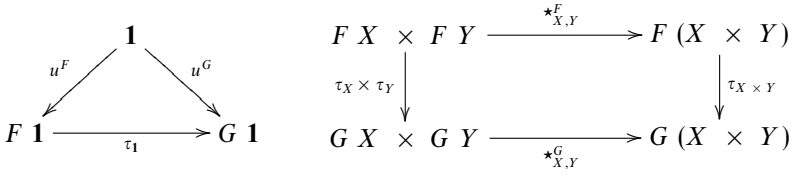


All Set functors are strong, but the strength $\tau : F\ X\ \times\ Y \to F\ (X\ \times\ Y)$ of an applicative functor F is required to be coherent with the monoidal action, i.e. the following diagram commutes.



Applicative functors may alternatively be given as a mapping of objects $F : |\mathsf{Set}| \to |\mathsf{Set}|$ equipped with two natural transformations $pure_X : X \to F\ X$ and $\circledast_{X,Y} : F\ (X \to Y) \times F\ X \to F\ Y$, together with some equations (see (McBride & Paterson, 2008) for details). This presentation is more useful for programming and therefore is the one chosen in Haskell. However, for our purposes, the presentation of applicative functors as monoidal functors is more convenient. This situation where one presentation is more apt for programming, and another presentation is better for formal reasoning also occurs with monads, where *bind* ($\gg\!=$) is preferred for programming and the multiplication (*join*) is preferred for formal reasoning.

Definition 5.2 (*Applicative morphism*)
Let F and G be applicative functors. An *applicative morphism* is a natural transformation $\tau : F \to G$ that respects the unit and multiplication. That is, a natural transformation τ such that the following diagrams commute.



Applicative functors and applicative morphisms form a strict monoidal category $\mathscr{A}$. The identity functor is an applicative functor, and the composition of applicative functors is an applicative functor. Hence, $\mathscr{A}$ has the structure of a strict monoidal category.

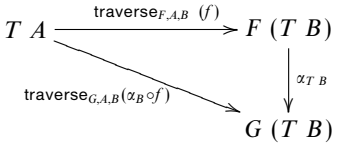
5.2 Traversable functors

McBride and Paterson (2008) characterise *traversable functors* as those equipped with a family of morphisms $\mathsf{traverse}_{F,A,B} : (A \to FB) \times TA \to F(TB)$, natural in an applicative functor F, and sets A and B (cf. the type synonym *VTraversable* from Section 4.1.) However, without further constraints this characterisation is too coarse. Hence, Jaskelioff and Rypáček (2012) proposed the following notion:

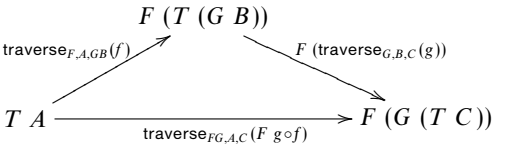
Definition 5.3 (*Traversable functor*)
A functor $T : Set \to Set$ is said to be *traversable* if there is a family of functions

$$\mathsf{traverse}_{F,A,B} : (A \to FB) \times TA \to F(TB)$$

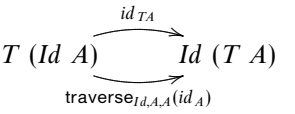
natural in F, A, and B that respects the monoidal structure of applicative functor composition. More concretely, for all applicative functors $F, G : Set \to Set$ and applicative morphisms $\alpha : F \to G$, the following diagrams should commute:



naturality



linearity



unity

5.3 Characterising traversable functors

Let $\mathscr{A}$ be the category of applicative functors and applicative morphisms. In order to prove that traversable functors are finitary containers, we first note that the forgetful functor U from the category of applicative functors $\mathscr{A}$ into the category of endofunctors has a left adjoint $(-)^*$ (Capriotti & Kaposi, 2014) and therefore we can apply Theorem 4.10 to any traversal which satisfies the linearity and unity laws. Hence, for every traversal on T

$$\mathsf{traverse}_{A,B} : \int_{F:\mathscr{A}} (A \to UFB) \to TA \to UF(TB)$$

there is a corresponding coalgebra

$$t_{A,B} : T\,A \to UR^*_{A,B}(T\,B)$$

where $R^*_{A,B}$ is the free applicative functor for $R_{A,B}$. The following proposition tells us what this free applicative functor looks like.

Proposition 5.4
The free applicative functor on $R_{A,B}$ is

$$R^*_{A,B}\,X = \Sigma\, n : \mathbb{N}.\ A^n \times (B^n \to X)$$

with action on morphisms $R^*_{A,B}(h)\,(n, as, f) = (n, as, h \circ f)$, and applicative structure

$$\begin{array}{rcl} u & : & R^*_{A,B}\,1 \\ u & = & (0, *, \lambda bs.*) \end{array}$$

$$\begin{array}{rcl} \star_{X,Y} & : & R^*_{A,B}\,X \times R^*_{A,B}\,Y \to R^*_{A,B}(X \times Y) \\ (n, as, f) \star (m, as', g) & = & (n+m,\ as +\!\!+ as',\ \lambda bs.(f\ (\mathsf{take}\ n\ bs), g\ (\mathsf{drop}\ n\ bs))) \end{array}$$

where we write X^n for vectors of length n, i.e. the n-fold product $\overbrace{X \times \cdots \times X}^{n \text{ times}}$, $+\!\!+$ for vector append, and $\mathsf{take}\ n$ and $\mathsf{drop}\ n$ for the functions that given a vector of size $n+m$ return the first n elements and the last m elements respectively.

The datatype *FreeApplicativePStore* given in Section 4.1 is a Haskell implementation of the free applicative functor on $R_{A,B}$, namely $R^*_{A,B}$.

Hence $R^*_{A,B}\,X$ consists of

1. a natural number, which we call the *dimension*,
2. a finite vector, which we call the *position*,
3. a function from a finite vector, which allows us to *peek* into new positions.

In order to make it easier to talk about the different components, we define projections: let $r = (n, i, g) : R^*_{A,B}\,X$, then $\mathsf{dim}\ r = n$, $\mathsf{pos}\ r = i$, and $\mathsf{peek}\ r = g$.

Theorem 4.10 tells us that UR^* is a parameterised comonad with the following counit and comultiplication operations.

$$\begin{array}{rcl} \varepsilon_{A,X} & : & UR^*_{A,A}\,X \to X \\ \varepsilon_{A,X}(n, as, f) & = & f\ as \end{array}$$

$$\begin{array}{rcl} \delta_{A,B,C,X} & : & UR^*_{A,C}\,X \to UR^*_{A,B}(UR^*_{B,C}\,X) \\ \delta_{A,B,C,X}(n, as, f) & = & (n,\ as,\ \lambda bs.(n, bs, f)) \end{array}$$

Furthermore, given a traversal of T, a coalgebra for UR^*, (T, t) is given by $t_{A,B} = \mathsf{traverse}_{A,B}\,\mathsf{wrap}_{A,B}$, where

$$\begin{array}{rcl} \mathsf{wrap}_{A,B} & : & A \to UR^*_{A,B}\,B \\ \mathsf{wrap}_{A,B}\ a & = & (1, a, id_b) \end{array}$$

In the other direction, given a coalgebra for UR^*, (T, t), we obtain a traversal for T:

$$\mathsf{traverse}_{A,B}\ f\ x = \mathbf{let}\ (n, as, g) = t\ x\ \mathbf{in}\ F(g)\ (\mathsf{collect}_n\ f\ as)$$

where $\mathsf{collect}_n\ f\ (x_1, \ldots, x_n) = f(x_1) \star \cdots \star f(x_n)$.

5.4 Finitary containers

A *finitary container* (Abbott *et al.*, 2003) is given by a set of shapes S, and an arity function $ar : S \to \mathbb{N}$. The *extension* of a finitary container (S, ar) is a functor $[\![S, ar]\!] : \mathsf{Set} \to \mathsf{Set}$ defined as follows.

$$[\![S, ar]\!]\ X = \Sigma\ s : S.\ X^{(ar\ s)}$$

Given an element of an extension of a finitary container $c = (s, xs) : \Sigma\ s : S.\ X^{(ar\ s)}$, we define projections $\mathsf{shape}\ c = s$, and $\mathsf{contents}\ c = xs$.

As an example, lists are given by the finitary container $(\mathbb{N}, id_{\mathbb{N}})$, where the set of shapes indicates the length of the list. Therefore its extension is

$$[\![\mathbb{N}, id]\!]\ X = \Sigma\ n : \mathbb{N}.\ X^n.$$

Vectors of length n are given by the finitary container $(1, \lambda x.n)$. They have only one shape and have a fixed arity. Streams are containers (Abbott *et al.*, 2003) with exactly one shape, but are not finitary.

Lemma 5.5 (*Finitary containers are traversable*)
The extension of any finitary container (S, ar) is traversable with a canonical traversal given by:

$$\begin{array}{lcl} \mathsf{traverse}_{F,X,Y} & : & (X \to F\ Y) \times [\![S, ar]\!]\ X \ \to\ F\ [\![S, ar]\!]\ Y \\ \mathsf{traverse}_{F,X,Y}\ (f, (s, xs)) & = & F(\lambda c.\ (s, c))(\mathsf{collect}_{ar(s)}\ f\ xs) \end{array}$$

5.5 *Finitary containers from coalgebras*

For the first part of our proof, we already showed that every traversal is isomorphic to an UR^*-coalgebra. For the second part, we show that if (T, t) is a UR^*-coalgebra then T is a finitary container.

Theorem 5.6
Let $X : \mathsf{Set}$ and let (T, t) be a coalgebra for UR^*. That is, $T : \mathsf{Set} \to \mathsf{Set}$ is a functor and $t_{A,B} : T\ A \to UR^*_{a,b}\ (T\ B)$ is a family natural in A and dinatural in B such that certain laws hold (see Definition 4.2). Then $T\ X$ is isomorphic to the extension of the finitary container $[\![T1, \lambda s.\ \mathsf{dim}\ (t\ s)]\!]\ X$.

Proof
We define an isomorphism between $T\ X$ and $\Sigma\ s : T1.\ X^{(\mathsf{dim}\ (t\ s))}$.

Given a value $x : T\ X$, the contents of the resulting container are simply the position of $(t\ x)$. The shape of the resulting container is obtained by peeking into $(t\ x)$ at the trivial vector $*^n : 1^n$ where n is the dimension of $(t\ x)$. More formally, we define one direction of the isomorphism as

$$\begin{array}{l} \Phi\ \ : T\ X \to \Sigma\ s : T1.\ X^{(\mathsf{dim}\ (t\ s))} \\ \Phi\ x = \mathbf{let}\ (n, i, g) = t\ x\ \mathbf{in}\ (g\ (*^n), i) \end{array}$$

Given a value $(s, v) : \Sigma\ s : T1.\ X^{(\mathsf{dim}\ (t\ s))}$, we can create a $T\ X$ by peaking into $(t\ s)$ at v. More formally, the other direction of the isomorphism is defined as

$$\begin{array}{l} \Psi\ \ : \Sigma\ s : T1.\ X^{(\mathsf{dim}\ (t\ s))} \to T\ X \\ \Psi\ (s, v) = \mathsf{peek}\ (t\ s)\ v \end{array}$$

First, we prove that $\Psi\ (\Phi\ x) = x$.

$$
\begin{aligned}
& \Psi\ (\Phi\ x) \\
= & \quad \{\ \text{definition of } \Psi, \Phi\ \} \\
& \textbf{let}\ (n, i, g) = t\ x\ \textbf{in}\ \mathsf{peek}\ (t\ (g\ (*^n)))\ i \\
= & \quad \{\ \text{map on morphisms of } UR^*_{a,b}\ \} \\
& \textbf{let}\ (n, i, h) = UR^*_{a,b}\ (t)\ (t\ x)\ \textbf{in}\ \mathsf{peek}\ (h\ (*^n))\ i \\
= & \quad \{\ \text{comultiplication-coalgebra law}\ \} \\
& \textbf{let}\ (n, i, h) = \delta\ (t\ x)\ \textbf{in}\ \mathsf{peek}\ (h\ (*^n))\ i \\
= & \quad \{\ \text{definition of } \delta \text{ and } \mathsf{peek}\ \} \\
& \textbf{let}\ (n, i, g) = (t\ x)\ \textbf{in}\ g\ i \\
= & \quad \{\ \text{definition of } \epsilon\ \} \\
& \varepsilon\ (t\ x) \\
= & \quad \{\ \text{counit-coalgebra law}\ \} \\
& x
\end{aligned}
$$

Last, we prove that $\Phi\ (\Psi\ (s, v)) = (s, v)$.

$$
\begin{aligned}
& \Phi\ (\Psi\ (s, v)) \\
= & \quad \{\ \text{definition of } \Psi, \Phi, \text{and map on morphisms of } UR^*_{a,b}\ \} \\
& \textbf{let}\ \{(\_, \_, h) = UR^*_{a,b}\ t\ (t\ s); (n, i, g) = h\ v\}\ \textbf{in}\ (g\ (*^n), i) \\
= & \quad \{\ \text{comultiplication-coalgebra law}\ \} \\
& \textbf{let}\ \{(\_, \_, h) = \delta\ (t\ s); (n, i, g) = h\ v\}\ \textbf{in}\ (g\ (*^n), i) \\
= & \quad \{\ \text{definition of } \delta\ \} \\
& \textbf{let}\ (n, j, g) = t\ s\ \textbf{in}\ (g\ (*^n), v) \\
= & \quad \{\ j = (*^n) \text{ because } 1^n \text{ has a unique element}\ \} \\
& \textbf{let}\ (n, j, g) = t\ s\ \textbf{in}\ (g\ j, v) \\
= & \quad \{\ \text{definition of } \epsilon\ \} \\
& (\varepsilon\ (t\ s), v) \\
= & \quad \{\ \text{counit-coalgebra law}\ \} \\
& (s, v)
\end{aligned}
$$

□

Corollary 5.7
Let X : Set and T : Set → Set be a traversable functor. Then, $T\ X$ is isomorphic to the finitary container $[\![T1, \lambda s.\ \mathsf{dim}\ (\mathsf{traverse}\ \mathsf{wrap}\ s)]\!]\ X$.

Proof
Apply Theorem 5.6 with the UR^*-coalgebra $t = \mathsf{traverse}\ \mathsf{wrap}$. □

All that remains to show is that this isomorphism maps the traversal of T to the canonical traversal of the finitary container.

Theorem 5.8
Let T : Set → Set be a traversable functor and let Φ : $T\ X \to [\![T1, \lambda s.\ \mathsf{dim}\ (\mathsf{traverse}\ \mathsf{wrap}\ s)]\!]\ X$ be the isomorphism defined above. Let F be an arbitrary applicative functor and let $f : A \to F\ B$ and $x : T\ A$. Then, $F\ (\Phi)\ (\mathsf{traverse}\ f\ x) = \mathsf{traverse}\ f\ (\Phi\ x)$.

Proof
Before beginning, we prove two small lemmas. First that pos (traverse wrap x) = contents ($\Phi\ x$).

$$
\begin{array}{ll}
 & \mathsf{pos}\ (\mathsf{traverse}\ \mathsf{wrap}\ x) \\
= & \quad \{\ \text{definition of pos}\ \} \\
 & \textbf{let}\ (\_, i, \_) = \mathsf{traverse}\ \mathsf{wrap}\ x\ \textbf{in}\ i \\
= & \quad \{\ \text{definition of}\ \Phi\ \} \\
 & \mathsf{contents}\ (\Phi\ x)
\end{array}
$$

Second, we prove that Φ (peek (traverse wrap x) w) = (shape ($\Phi\ x$), w)

$$
\begin{array}{ll}
 & \Phi\ (\mathsf{peek}\ (\mathsf{traverse}\ \mathsf{wrap}\ x)\ w) \\
= & \quad \{\ \text{definition of peek}\ \} \\
 & \textbf{let}\ (\_, \_, g) = \mathsf{traverse}\ \mathsf{wrap}\ x\ \textbf{in}\ \Phi\ (g\ w) \\
= & \quad \{\ \text{definition of}\ \Phi\ \} \\
 & \textbf{let}\ \{(\_, \_, g) = \mathsf{traverse}\ \mathsf{wrap}\ x; (n, i, h) = \mathsf{traverse}\ \mathsf{wrap}\ (g\ w)\}\ \textbf{in}\ (h\ (*^n), i) \\
= & \quad \{\ \text{definition of}\ UR^*_{a,b}\ \} \\
 & \textbf{let}\ \{(\_, \_, g) = UR^*_{a,b}\ (\mathsf{traverse}\ \mathsf{wrap})\ (\mathsf{traverse}\ \mathsf{wrap}\ x); (n, i, h) = g\ w\}\ \textbf{in}\ (h\ (*^n), i) \\
= & \quad \{\ \text{coalgebra law for}\ \delta\ \} \\
 & \textbf{let}\ \{(\_, \_, g) = \delta\ (\mathsf{traverse}\ \mathsf{wrap}\ x); (n, i, h) = g\ w\}\ \textbf{in}\ (h\ (*^n), i) \\
= & \quad \{\ \text{definition of}\ \delta\ \} \\
 & \textbf{let}\ (\_, \_, g) = \mathsf{traverse}\ \mathsf{wrap}\ x\ \textbf{in}\ (g\ (*^n), w) \\
= & \quad \{\ \text{definition of}\ \Phi\ \} \\
 & (\mathsf{shape}\ (\Phi\ x), w)
\end{array}
$$

Lastly, we prove our main result.

$$
\begin{array}{ll}
 & F\ (\Phi)\ (\mathsf{traverse}\ f\ x) \\
= & \quad \{\ \text{isomorphism in Theorem 4.10}\ \} \\
 & \textbf{let}\ (n, i, g) = \mathsf{traverse}\ \mathsf{wrap}\ x\ \textbf{in}\ F\ (\Phi)\ (F\ (g)\ (\mathsf{collect}_n\ f\ i)) \\
= & \quad \{\ \text{functors respect composition}\ \} \\
 & \textbf{let}\ (n, i, g) = \mathsf{traverse}\ \mathsf{wrap}\ x\ \textbf{in}\ F\ (\Phi \circ g)\ (\mathsf{collect}_n\ f\ i) \\
= & \quad \{\ \text{application of above two lemmas}\ \} \\
 & \textbf{let}\ (s, v) = \Phi\ x\ \textbf{in}\ F\ (\lambda c.\ (s, c))\ (\mathsf{collect}_n\ f\ v) \\
= & \quad \{\ \text{definition of canonical traverse for finitary containers}\ \} \\
 & \mathsf{traverse}\ f\ (\Phi\ x)
\end{array}
$$

□

The isomorphism between T and $[\![T1, \lambda s.\ \mathsf{dim}\ (\mathsf{traverse}\ \mathsf{wrap}\ s)]\!]$ must be natural by construction. However, naturality is also an immediate consequence of the preceding theorem because traversing with the identity functor I is equivalent to the mapping on morphisms of a traversable functor.

6 Implementing algebraic theories

As a last application of the representation theorem, we take a look at the case where we consider $\mathscr{M}$, the category of monads with monad homomorphisms. In this situation, the functor $(-)^* : \mathscr{E} \to \mathscr{M}$, maps any functor $F : \mathscr{E}$ to F^*, the free monad

on F, while the functor $U : \mathscr{M} \to \mathscr{E}$ forgets the monad structure. The representation theorem then states that

$$\int_{M \in \mathscr{M}} (A \to UM\,B) \to UM\,X \quad \cong \quad UR^*_{A,B}\,X \tag{6.1}$$

where, $R_{A,B}\,X = A \times (B \to X)$ is the parameterised store comonad.

In Haskell, we can write the isomorphism (6.1) as

$$\forall m.\ \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to m\ x \quad \cong \quad \mathit{Free}\ (\mathit{PStore}\ a\ b)\ x$$

where *PStore* (as given in Section 4.1) and the free monad construction are as follows:

```
newtype PStore a b x = PStore (b → x) a
data Free f x = Unit x | Branch (f (Free f x))
instance Functor f ⇒ Monad (Free f) where
  return          = Unit
  Pure x >>= f    = f x
  Branch xs >>= f = Branch (fmap (>>= f) xs)
```

This way of constructing a free monad from an arbitrary functor requires a recursive datatype. The isomorphism Equation (6.1), on the other hand, shows a non-recursive way of describing the free monad on functors of the form *PStore a b*.

While this result seems to be of limited applicability, we note that every signature of an algebraic operation with parameter a and arity b determines a functor of this form. Hence, the theorem tells us how to construct the free monad on a given signature of a single algebraic operation. Intuitively the type

$$\forall m.\ \mathsf{Monad}\ m \Rightarrow (a \to m\ b) \to m\ x$$

describes a monadic computation $m\ x$ in which the only source of impurity is the operation of type $a \to m\ b$ in the argument. This type can be implemented in Haskell in the following manner, where we have abstracted over the types of the argument operation.

```
newtype FreeOp primOp x = FreeOp {runOp :: ∀m. Monad m ⇒ primOp m → m x}
instance Monad (FreeOp primOp) where
  return x = FreeOp (const (return x))
  x >>= f  = FreeOp (λop → runOp x op >>= λa → runOp (f a) op)
```

Notice that the bind operation for *FreeOp* is not recursive, but is implemented in terms of the bind operation for an arbitrary abstract monad.

For example, exceptions in a type e can be given by a nullary operation *throw* with parameter e.[4]

```
type Exc e m = e → m ∅
```

[4] In order to avoid clutter, we sometimes use a type synonym where a real implementation would require a newtype, with its associated constructor and destructor.

where $\emptyset$ is the empty type, and hence *FreeOp* (*Exc e*) is the type of monadic computations which can throw an exception using the following operation:

```
throw   :: e → FreeOp (Exc e) ∅
throw e = FreeOp (λ_throw → _throw e)
```

We may model environments in r by an operation *ask* with parameter () and arity r.

```
type Env r m = () → m r
```

Hence, *FreeOp* (*Env r*) is the type of monadic computation which can read an environment using the following operation:

```
ask :: FreeOp (Env r) r
ask = FreeOp (λ_ask → _ask ())
```

More generally, we may want to consider algebraic theories with more than one operation. Following the same argument as before, but considering the N-ary representation theorem, we can construct the free monad on any signature of algebraic operations and express it by its *generic effects* (Plotkin & Power, 2003) by means of a polymorphic type.

For example, a simple teletype interface can be represented by the following functor (Swierstra, 2008):

```
data Teletype x = GetChar (Char → x)
                | PutChar Char x
```

The free monad generated by this *Teletype* functor produces a tree representing all the interactions with a teletype machine a user can have. The *Teletype* functor is isomorphic to a sum of instances of R

$$Teletype\ x \quad \cong \quad ((), Char \to x) + (Char, () \to x) \quad \cong \quad (R\ ()\ Char + R\ Char\ ())\ x$$

By the N-ary representation theorem, the free monad generated by *Teletype* is isomorphic to

$$\forall m.\ \mathsf{Monad}\ m \Rightarrow (() \to m\ Char) \to (Char \to m\ ()) \to m\ x$$

We define a type for representing teletype operations. In order to reuse our previous definition of *FreeOp* and to get names for each argument, we define the type as a record in which each field corresponds to an operation.

```
data TTOp m = TTOp { _ttGetChar :: m Char
                   , _ttPutChar :: Char → m ()
                   }
```

We obtain the free monad for *TTOp* and define operations on it that basically choose the corresponding field from the record.

```
type FreeTT = FreeOp TTOp
ttGetChar :: FreeTT Char
```

```
ttGetChar = FreeOp _ttGetChar
ttPutChar  :: Char → FreeTT ()
ttPutChar c = FreeOp (λpo → _ttPutChar po c)
```

Values of type *FreeTT* can easily be interpreted in *IO*, by providing operations of the appropriate type.

```
runTTIO :: FreeTT a → IO a
runTTIO = runOp ttOpIO
  where ttOpIO :: TTOp IO
        ttOpIO = TTOp { _ttGetChar = getChar
                      , _ttPutChar = putChar
                      }
```

Of course, the larger purpose is that *FreeTT* values can be interpreted in other ways, for example, by logging input, or for use in automated tests by replaying previously logged input. Furthermore, a *FreeOp* monad can easily be embedded into another *FreeOp* monad with a larger set of primitive commands, or interpreted into another *FreeOp* monad with a smaller, more primitive set of commands, providing a simple way of implementing handlers of algebraic effects (Plotkin & Pretnar, 2009). Hence, Theorem 3.2 might provide the basis for a simple implementation of an algebraic-effects library.

7 Related work

Traversable functors were introduced by McBride and Paterson (2008), generalising a notion of traversal by Moggi et al. (1999). The notion proposed was too coarse and Gibbons and Oliveira (2009) analysed several properties that should hold for all traversals. Based on some of these properties, Jaskelioff and Rypáček (2012) proposed a characterisation of traversable functors, and conjectured that they were isomorphic to finitary containers (Abbott *et al.*, 2003). The conjecture was proven correct by Bird et al. (2013) by a means of a change of representation. The proof of this same fact presented in Section 5 uses a similar change of representation and was found independently.

The representation of the free applicative functor on the parameterised store comonad, *R*, is a dependently typed version of Van Laarhoven's *FunList* data type (Van Laarhoven, 2009b). Van Laarhoven's applicative and parameterised comonad instances for this type have been translated to work on the dependently typed implementation. A particular case of the representation theorem has been conjectured by Van Laarhoven (2009c), and proved by O'Connor (2011). The proof of representation theorem for functors via the Yoneda lemma was discovered independently by Bartosz Milewski (2013).

The representation theorems applied to the case where the structured functors are monads (as in Section 6) yields isomorphisms analogous to the ones presented by Bauer *et al.,* (2013). However, our proof is based on a categorical model, while

theirs is based on a parametric model. Also, as opposed to us, they do not explore the connection with algebraic effects.

Bernardy *et al.,* (2010) use a representation theorem to transform polymorphic properties of a certain shape into monomorphic properties, which are easier and more efficient to test. This suggests that another application for the representation theorems in this article is to facilitate the testing of polymorphic properties.

Acknowledgments

Jaskelioff is funded by ANPCyT PICT 2009-15. Many thanks go to Edward Kmett who assisted the authors with the isomorphism between *KLens* and *VLens*, and to Exequiel Rivas, Jeremy Gibbons, and the anonymous referees for helping us improve the presentation of the paper. We also thank Shachaf Ben-Kiki for explaining why affine traversals are called so, and Gabor Greif for finding some typos.

References

Abbott, M., Altenkirch, T. & Ghani, N. (2003) Categories of containers. In Proceedings of Foundation s of Software Science and Computation Structures. pp. 23–38.

Atkey, R. (2009a) Algebras for parameterised monads. In Proceedings of the Algebra and Coalgebra in Computer Science, 3rd International Conference, CALCO 2009, Udine, Italy (September 7–10, 2009), Kurz, A., Lenisa, M. & Tarlecki, A. (eds), vol. 5728. Lecture Notes in Computer Science, Springer, pp. 3–17.

Atkey, R. (2009b) Parameterised notions of computation. *J. Funct. Program.* **19**(3 & 4), 335–376.

Awodey, S. (2006) *Category Theory*. USA: Oxford University Press.

Bainbridge, E. S., Freyd, P. J., Scedrov, A. & Scott, P. J. (1990) Functorial polymorphism. *Theor. Comput. Sci.* **70**(1), 35–64.

Bauer, A., Hofmann, M. & Karbyshev, A. (2013) On monadic parametricity of second-order functionals. In *Foundations of Software Science and Computation Structures*, Pfenning, F. (ed), vol. 7794. Lecture Notes in Computer Science, . Berlin Heidelberg: Springer, pp. 225–240.

Bernardy, J.-P., Jansson, P. & Claessen, K. (2010) Testing polymorphic properties. In *Programming Languages and Systems*, Gordon, A. D. (ed), vol. 6012. Lecture Notes in Computer Science. Berlin Heidelberg: Springer, pp. 125–144.

Bird, R. & de Moor, O. (1997) *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall.

Bird, R., Gibbons, J., Mehner, S., Voigtländer, J. & Schrijvers, T. (2013) Understanding idiomatic traversals backwards and forwards. In Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. Haskell '13. New York, NY, USA: ACM, pp. 25–36.

Capriotti, P. & Kaposi, A. 2014 (April) Free applicative functors. In Proceedings of the 5th Workshop on Mathematically Structured Functional Programming. MSFP '14, pp. 2–30.

Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. *Sigplan Not.* **41**(1), 206–217.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. & Schmitt, A. (2007) Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* **29**(3). (Article 17).

Gibbons, J. & Johnson, M. (2012) Relating algebraic and coalgebraic descriptions of lenses. **49** (Bidirectional Transformations 2012). Berlin, Germany.

Gibbons, J. & Oliveira, B. C. D. S. (2009) The essence of the iterator pattern. *J. Funct. Program.*, **19**, 377–402.

Jaskelioff, M. & Rypacek, O. (2012) An investigation of the laws of traversals. In Proceedings of the 4th Workshop on Mathematically Structured Functional Programming. EPTCS, Chapman, J. & Levy, P. B. (eds), vol. 76, pp. 40–49.

Kmett, E. 2013 (October) *Lens-4.0: Lenses, Folds and Traversals.* Available at: http://ekmett.github.io/lens/Control-Lens-Traversal.html.

Mac Lane, S. (1971) *Categories for the Working Mathematician.* Graduate Texts in Mathematics, no. 5, 2nd ed., Springer-Verlag, 1998.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(01), 1–13.

Milewski, B. 2013 (October) *Lenses, Stores, and Yoneda.* Available at: http://bartoszmilewski.com/2013/10/08/lenses-stores-and-yoneda.

Moggi, E., Bellè, G. & Jay, C. B. (1999) Monads, shapely functors and traversals. *Electron. Notes Theor. Comput. Sci.* **29**, 187–208.

O'Connor, R. 2010 (Nov.) *Lenses are Exactly the Coalgebras for the Store Comonad.* Available at: http://r6research.livejournal.com/23705.html.

O'Connor, R. (2011) Functor is to lens as applicative is to biplate: Introducing multiplate. *Corr*, **abs/1103.2841v1**.

O'Connor, R., Kmett, E. A. & Morris, T. 2013 (October) *Data-lens-2.10.4: Haskell 98 lenses.* Available at: http://hackage.haskell.org/package/data-lens-2.10.4/docs/Data-Lens-Partial-Common.html.

Plotkin, G. & Power, J. (2003) Algebraic operations and generic effects. *Appl. Categ. Struct.* **11**(1), 69–94.

Plotkin, G. & Pretnar, M. (2009) Handlers of algebraic effects. In *Programming Languages and Systems*, Castagna, G. (ed), vol. 5502. Lecture Notes in Computer Science, . Berlin Heidelberg: Springer.

Reynolds, J. C. & Plotkin, G. D. (1993) On functors expressible in the polymorphic typed lambda calculus. *Inform. Comput.* **105**(1), 1–29.

Swierstra, W. (2008) Data types à la carte. *J. Funct. Program.* **18**(4), 423–436.

Van Laarhoven, T. 2009a (Aug.) *CPS Based Functional References.* Available at: http://twanvl.nl/blog/haskell/cps-functional-references.

Van Laarhoven, T. 2009b (Apr.) *A Non-Regular Data Type Challenge.* Available at: http://twanvl.nl/blog/haskell/non-regular1.

Van Laarhoven, T. 2009c (Apr.) *Where Do I Get My Non-Regular Types?* Available at: http://twanvl.nl/blog/haskell/non-regular2.

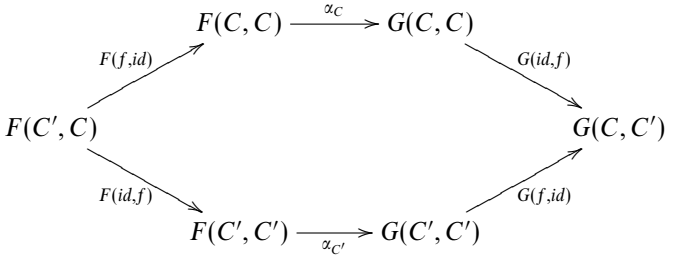
Appendix A: Ends

Ends are a special type of limit. The limit for a functor $F : \mathscr{C} \to \mathscr{D}$ is a universal natural transformation $K_D \to F$ (the universal cone to F) from the functor which is constantly D, for a $D \in \mathscr{D}$, into the functor F. The end for a functor $F : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ arises as a *dinatural* transformation $K_D \to F$ (the universal wedge).

Definition A.1

A *dinatural transformation* $\alpha : F \to G$ between functors $F, G : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ is a family of morphisms of the form $\alpha_C : F(C, C) \to G(C, C)$, such that for every

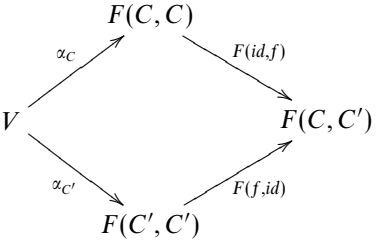
morphism $f : C \to C'$ the following diagram commutes.



Differently from natural transformations, dinatural transformations are not closed under composition.

Definition A.2

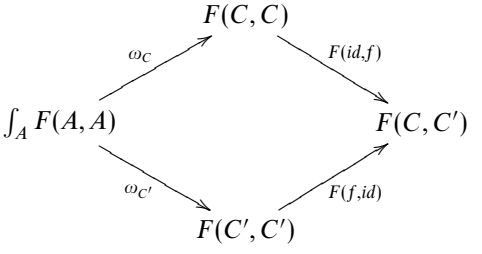
A *wedge* from an object $V \in \mathscr{D}$ to a functor $F : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ is a dinatural transformation from the constant functor $K_V : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ to F. Explicitly, an object V together with a family of morphisms $\alpha_X : V \to F(X, X)$ such that for each $f : C \to C'$ the following diagram commutes.



Whereas a limit is a final cone, an *end* is a final wedge.

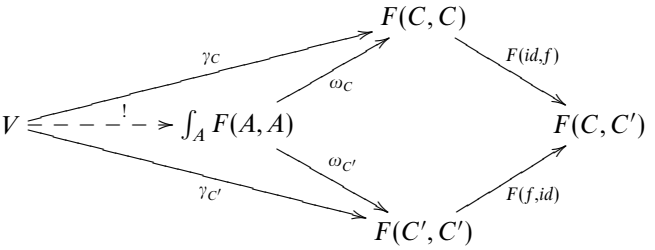
Definition A.3

The *end* of a functor $F : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ is a final wedge for F. Explicitly, it is an object $\int_A F(A, A) \in \mathscr{D}$ together with a family of morphisms $\omega_C : \int_A F(A, A) \to F(C, C)$ such that the diagram



commutes for each $f : C \to C'$, and such that for every wedge from $V \in \mathscr{D}$, given by a family of morphisms $\gamma_c : V \to F(C, C)$ such that $F(id, f) \circ \gamma_c = F(f, id) \circ \gamma'_c$ for every $f : C \to C'$, there exists a unique morphism $! : V \to \int_A F(A, A)$ such that the

following diagram commutes.



Remark A.4
When $\mathscr{C}$ is small and $\mathscr{D}$ is small-complete, an end over a functor $\mathscr{C} \times \mathscr{C}^{\mathrm{op}} \to \mathscr{D}$ can be reduced to an ordinary limit (Mac Lane, 1971). As a consequence, the Hom functor preserves ends: for every $D \in \mathscr{D}$,

$$D \xrightarrow{\mathscr{D}} \int_A F(A,A) \quad = \quad \int_A D \xrightarrow{\mathscr{D}} F(A,A).$$

Appendix B: Generalised Lens Representation Theorem

For all the propositions below, assume we have two small monoidal categories of endofunctors, $(\mathscr{E}, I, \cdot, \alpha, \lambda, \rho)$ and $(\mathscr{F}, I, \cdot, \alpha', \lambda', \rho')$, $\mathscr{E}$ is a subcategory of endofunctors over a base category $\mathscr{C}$, and $\mathscr{F}$ is a subcategory of endofunctors over a base category $\mathscr{D}$, and where the monoidal operation is composition of endofunctors (written $F \cdot G$) and with the identity functor, I, as the identity. Also assume we have an adjunction $(-)^* \dashv U : \mathscr{E} \rightharpoonup \mathscr{F}$, such that U is strict monoidal[5] (i.e. $U\,I = I$, $U(F \cdot G) = UF \cdot UG$, $U\lambda'_X = \lambda_{UX}$, etc.).

To reduce notational clutter, in this section we work directly with natural transformations. Rather that writing the counit of a parameterised comonad as a family of arrows $\varepsilon_{a,X} : C_{a,a}X \to X$ as we did in Section 4, we will write it as a family of natural transformations, $\varepsilon_a : C_{a,a} \to I$. Similarly, instead of writing comultiplication as $\delta_{a,b,c,X} : C_{a,c}X \to C_{a,b}(C_{b,c}X)$ we will write $\delta_{a,b,c} : C_{a,c} \to C_{a,b} \cdot C_{b,c}$, and so forth.

Proposition A.5
Let $(C, \varepsilon^C, \delta^C)$ be a $\mathscr{P}$-parameterised comonad on $\mathscr{C}$, such that for every $a, b : \mathscr{P}$, we have an endofunctor $C_{a,b} : \mathscr{E}$. Then $(C^*, \varepsilon^{C^*}, \delta^{C^*})$ is a $\mathscr{P}$-parameterised comonad on $\mathscr{D}$ where

$$\begin{array}{lcl} \varepsilon^{C^*}_a & : & C^*_{a,a} \to I \\ \varepsilon^{C^*}_a & = & \lceil \varepsilon^C_a \rceil \end{array}$$

$$\begin{array}{lcl} \delta^{C^*}_{a,b,c} & : & C^*_{a,c} \to C^*_{a,b} \cdot C^*_{b,c} \\ \delta^{C^*}_{a,b,c} & = & \lceil (\eta_{C_{a,b}} \cdot \eta_{C_{b,c}}) \circ \delta^C_{a,b,c} \rceil \end{array}$$

[5] These propositions still hold under the assumption that U is a strong monoidal functor. In order to avoid excessive notation, we use the simplifying assumption that U is strict.

The tensor $\cdot$ in the term corresponds to horizontal composition of natural transformations.

Proof
The first parameterised comonad law is

$$\lambda_{C_{a,b}} \circ (\varepsilon_a^C \cdot id) \circ \delta^C_{a,a,b} = id : C_{a,b} \xrightarrow{\mathscr{E}} C_{a,b}$$

We check that

$$\lambda'_{C^*_{a,b}} \circ (\varepsilon_a^{C^*} \cdot id) \circ \delta^{C^*}_{a,a,b} = id : C^*_{a,b} \xrightarrow{\mathscr{F}} C^*_{a,b}$$

$$\begin{aligned}
& \lambda'_{C^*_{a,b}} \circ (\varepsilon_a^{C^*} \cdot id) \circ \delta^{C^*}_{a,a,b} \\
= \quad & \{ \text{ Definition of } \delta^{C^*} \ \} \\
& \lambda'_{C^*_{a,b}} \circ (\varepsilon_a^{C^*} \cdot id) \circ \lceil (\eta_{C_{a,a}} \cdot \eta_{C_{a,b}}) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \text{ Equation (2.6) } \} \\
& \lceil U\lambda'_{C^*_{a,b}} \circ U(\varepsilon_a^{C^*} \cdot id) \circ (\eta_{C_{a,a}} \cdot \eta_{C_{a,b}}) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \ U \text{ is strict monoidal. } \} \\
& \lceil \lambda_{UC^*_{a,b}} \circ (U\, \varepsilon_a^{C^*} \cdot id) \circ (\eta_{C_{a,a}} \cdot \eta_{C_{a,b}}) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \ \text{Bifunctor } \cdot, \text{ definition of } \varepsilon^{C^*} \ \} \\
& \lceil \lambda_{UC^*_{a,b}} \circ ((U \lceil \varepsilon_a^C \rceil \circ \eta_{C_{a,a}}) \cdot \eta_{C_{a,b}}) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \ \text{Equation (2.8) } \} \\
& \lceil \lambda_{UC^*_{a,b}} \circ (\lfloor \lceil \varepsilon_a^C \rceil \rfloor \cdot \eta_{C_{a,b}}) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \ \text{isomorphism } \} \\
& \lceil \lambda_{UC^*_{a,b}} \circ (\varepsilon_a^C \cdot \eta_{C_{a,b}}) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \ \text{naturality of } \lambda \ \} \\
& \lceil \eta_{C_{a,b}} \circ \lambda_{C_{a,b}} \circ (\varepsilon_a^C \cdot id) \circ \delta^C_{a,a,b} \rceil \\
= \quad & \{ \ \text{first parameterised comonad law } \} \\
& \lceil \eta_{C_{a,b}} \rceil \\
= \quad & \{ \ \text{Equation (2.7) } \} \\
& \lceil \lfloor id \rfloor \rceil \\
= \quad & \{ \ \text{isomorphism } \} \\
& id
\end{aligned}$$

For the second parameterised comonad law, we proceed in a similar way to the first.

The third parameterised comonad law states

$$\alpha_{C_{a,b},C_{b,c},C_{c,d}} \circ (\delta^C_{a,b,c} \cdot id) \circ \delta^C_{a,c,d} = (id \cdot \delta^C_{b,c,d}) \circ \delta^C_{a,b,d} : C_{a,d} \xrightarrow{\mathscr{E}} C_{a,b} \cdot (C_{b,c} \cdot C_{c,d})$$

Let us prove that

$$\alpha'_{C^*_{a,b},C^*_{b,c},C^*_{c,d}} \circ (\delta^{C^*}_{a,b,c} \cdot id) \circ \delta^{C^*}_{a,c,d} = (id \cdot \delta^{C^*}_{b,c,d}) \circ \delta^{C^*}_{a,b,d} : C^*_{a,d} \xrightarrow{\mathscr{F}} C^*_{a,b} \cdot (C^*_{b,c} \cdot C^*_{c,d})$$

$$
\begin{aligned}
& \alpha' \circ (\delta^{C^*}_{a,b,c} \cdot id) \circ \delta^{C^*}_{a,c,d} \\
= \quad & \{ \text{ Definition of } \delta^{C^*} \ \} \\
& \alpha' \circ (\delta^{C^*}_{a,b,c} \cdot id) \circ \lceil (\eta_{C_{a,c}} \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \text{ Equation (2.6), } U \text{ strict monoidal } \} \\
& \lceil \alpha \circ (U\delta^{C^*}_{a,b,c} \cdot id) \circ (\eta_{C_{a,c}} \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \cdot \text{ bifunctor } \} \\
& \lceil \alpha \circ ((U\delta^{C^*}_{a,b,c} \circ \eta_{C_{a,c}}) \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \text{ Equation (2.8) } \} \\
& \lceil \alpha \circ (\lfloor \delta^{C^*}_{a,b,c} \rfloor \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \text{ Definition of } \delta^{C^*} \ \} \\
& \lceil \alpha \circ (\lfloor \lceil (\eta_{C_{a,b}} \cdot \eta_{C_{b,c}}) \circ \delta^{C}_{a,b,c} \rceil \rfloor \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \text{ isomorphism } \} \\
& \lceil \alpha \circ ((\eta_{C_{a,b}} \cdot \eta_{C_{b,c}}) \circ \delta^{C}_{a,b,c}) \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \cdot \text{ bifunctor } \} \\
& \lceil \alpha \circ ((\eta_{C_{a,b}} \cdot \eta_{C_{b,c}}) \cdot \eta_{C_{c,d}}) \circ (\delta^{C}_{a,b,c} \cdot id) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \text{ naturality of } \alpha \ \} \\
& \lceil ((\eta_{C_{a,b}} \cdot (\eta_{C_{b,c}} \cdot \eta_{C_{c,d}})) \circ \alpha \circ (\delta^{C}_{a,b,c} \cdot id) \circ \delta^{C}_{a,c,d} \rceil \\
= \quad & \{ \text{ third parameterised comonad law } \} \\
& \lceil ((\eta_{C_{a,b}} \cdot (\eta_{C_{b,c}} \cdot \eta_{C_{c,d}})) \circ (id \cdot \delta^{C}_{b,c,d}) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \cdot \text{ bifunctor } \} \\
& \lceil (\eta_{C_{a,b}} \cdot ((\eta_{C_{b,c}} \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{b,c,d})) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \text{ isomorphism } \} \\
& \lceil (\eta_{C_{a,b}} \cdot \lfloor \lceil (\eta_{C_{b,c}} \cdot \eta_{C_{c,d}}) \circ \delta^{C}_{b,c,d} \rceil \rfloor) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \text{ Definition of } \delta^{C^*} \ \} \\
& \lceil (\eta_{C_{a,b}} \cdot \lfloor \delta^{C^*}_{b,c,d} \rfloor) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \text{ Equation (2.8) } \} \\
& \lceil (\eta_{C_{a,b}} \cdot (U\delta^{C^*}_{b,c,d} \circ \eta_{C_{b,d}})) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \cdot \text{ bifunctor } \} \\
& \lceil (id \cdot U\delta^{C^*}_{b,c,d}) \circ (\eta_{C_{a,b}} \cdot \eta_{C_{b,d}}) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \text{ Equation (2.6), } U \text{ strict monoidal } \} \\
& (id \cdot \delta^{C^*}_{b,c,d}) \circ \lceil (\eta_{C_{a,b}} \cdot \eta_{C_{b,d}}) \circ \delta^{C}_{a,b,d} \rceil \\
= \quad & \{ \text{ Definition of } \delta^{C^*} \ \} \\
& (id \cdot \delta^{C^*}_{b,c,d}) \circ \delta^{C^*}_{a,b,d}
\end{aligned}
$$

□

Proposition A.6
Let $(D, \varepsilon^D, \delta^D)$ be a $\mathscr{P}$-parameterised comonad on $\mathscr{D}$, such that for every $a, b : \mathscr{P}$, we have an endofunctor $D_{a,b} : \mathscr{F}$. Then $(U\,D, \varepsilon^{U\,D}, \delta^{U\,D})$ is a $\mathscr{P}$-parameterised comonad on $\mathscr{C}$ where

$$\begin{array}{lcl} \varepsilon_a^{U\,D} & : & U\,D_{a,a} \to I \\ \varepsilon_a^{U\,D} & = & U\varepsilon_a^D \end{array}$$

$$\begin{array}{lcl} \delta_{a,b,c}^{U\,D} & : & U\,D_{a,c} \to U\,D_{a,b} \cdot U\,D_{b,c} \\ \delta_{a,b,c}^{U\,D} & = & U\delta_{a,b,c}^D \end{array}$$

Proof
The laws of a parameterised comonad follow directly from the fact that U is a strict monoidal functor. □

Proposition A.7 (*Generalised lens representation (Theorem 4.10)*
Given a functor $K : \mathscr{P} \to Set$, define $R_{a,b}^{(K)} X = Ka \times (Kb \to X) : \mathscr{P} \times \mathscr{P}^{\mathrm{op}} \times Set \to Set$ as the parameterised comonad with counit $\varepsilon^{R^{(K)}}$ and comultiplication $\delta^{R^{(K)}}$ as defined in Example 4.4. Assume that $R_{a,b}^{(K)} : \mathscr{E}$ for every a and b. Then

1. $UR^{(K)*}$ is a parameterised comonad and
2. given a functor $J : \mathscr{P} \to Set$, then the families $k_{a,b} : Ja \to UR_{a,b}^{(K)*}(Jb)$ which form the $UR^{(K)*}$-coalgebras (J, k) are isomorphic to the families of ends

$$\int_{F:\mathscr{F}} (Ka \to UF(Kb)) \to Ja \to UF(Jb)$$

which satisfy the linearity and unity laws.

Proof
The previous two propositions entail that $UR^{(K)*}$ is a parameterised comonad with the following counit and comultiplication.

$$\begin{array}{lcl} \varepsilon_a^{UR^{(K)*}} & : & UR_{a,a}^{(K)*} \to I \\ \varepsilon_a^{UR^{(K)*}} & = & U\lceil \varepsilon_a^{R^{(K)}} \rceil \end{array}$$

$$\begin{array}{lcl} \delta_{a,b,c}^{UR^{(K)*}} & : & UR_{a,c}^{(K)*} \to UR_{a,b}^{(K)*} \cdot UR_{b,c}^{(K)*} \\ \delta_{a,b,c}^{UR^{(K)*}} & = & U\lceil (\eta_{R_{a,b}^{(K)}} \cdot \eta_{R_{b,c}^{(K)}}) \circ \delta_{a,b,c}^{R^{(K)}} \rceil \end{array}$$

The unary representation theorem (Theorem 3.1) entails the isomorphism

$$Ja \to UR_{a,b}^{(K)*}(Jb) \quad \cong \quad \int_{F:\mathscr{F}} (Ka \to UF(Kb)) \to Ja \to UF(Jb)$$

witnessed by the following functions

$$\begin{array}{ll} \gamma & : (\textstyle\int_F (Ka \to UF(Kb)) \to Ja \to UF(Jb)) \to (Ja \to UR_{a,b}^{(K)*}(Jb)) \\ \gamma(h) & = h_{R_{a,b}^{(K)*}} (\alpha^{-1}_{UR_{a,b}^{(K)*}} (\eta_{R_{a,b}^{(K)}})) \end{array}$$

$$\begin{array}{ll} \gamma^{-1} & : (Ja \to UR_{a,b}^{(K)*}(Jb)) \to \textstyle\int_F (Ka \to UF(Kb)) \to (Ja \to UF(Jb)) \\ \gamma^{-1}(k) = \tau \quad \text{where} & \tau_F \quad : (Ka \to UF(Kb)) \to Ja \to UF(Jb) \\ & \tau_F(g) = U\lceil \alpha_{UF}(g) \rceil_{Jb} \circ k \end{array}$$

All that remains is to show that $k_{a,b}$ satisfies the coalgebra laws if and only if $\gamma^{-1}(k_{a,b})$ satisfies the linearity and unity laws.

First, we prove two lemmas:

Lemma A.8
For all $F, G : \mathscr{F}$ and $f : Ka \to UF(Kb)$ and $g : Kb \to UG(Kc)$, we have that

$$\gamma^{-1}(k_{a,c})_{F\cdot G}(UFg \circ f) \quad = \quad U(\lceil \alpha_{UF}(f) \rceil \cdot \lceil \alpha_{UG}(g) \rceil)_{Jc} \circ \delta^{UR^{(K)*}}_{a,b,c,Jc} \circ k_{a,c}$$

Proof

$$\begin{aligned}
& \gamma^{-1}(k_{a,c})_{F\cdot G}(UFg \circ f) \\
= \quad & \{ \text{ Definition of } \gamma^{-1} \ \} \\
& U\lceil \alpha_{UF\cdot UG}(UFg \circ f) \rceil_{Jc} \circ k_{a,c} \\
= \quad & \{ \text{ Proposition 4.5(b) } \} \\
& U\lceil (\alpha_{UF}(f) \cdot \alpha_{UG}(g)) \circ \delta^{R^{(K)}}_{a,b,c} \rceil_{Jc} \circ k_{a,c} \\
= \quad & \{ \text{ isomorphism } \} \\
& U\lceil (\lfloor \lceil \alpha_{UF}(f) \rceil \rfloor \cdot \lfloor \lceil \alpha_{UG}(g) \rceil \rfloor) \circ \delta^{R^{(K)}}_{a,b,c} \rceil_{Jc} \circ k_{a,c} \\
= \quad & \{ \text{ Equation (2.8) } \} \\
& U\lceil ((U\lceil \alpha_{UF}(f) \rceil \circ \eta_{R^{(K)}_{a,b}}) \cdot (U\lceil \alpha_{UG}(g) \rceil \circ \eta_{R^{(K)}_{b,c}})) \circ \delta^{R^{(K)}}_{a,b,c} \rceil_{Jc} \circ k_{a,c} \\
= \quad & \{ \cdot \text{ bifunctor and U is strict } \} \\
& U\lceil U(\lceil \alpha_{UF}(f) \rceil \cdot \lceil \alpha_{UG}(g) \rceil) \circ (\eta_{R^{(K)}_{a,b}} \cdot \eta_{R^{(K)}_{b,c}}) \circ \delta^{R^{(K)}}_{a,b,c} \rceil_{Jc} \circ k_{a,c} \\
= \quad & \{ \text{ Equation (2.6) and U is strict } \} \\
& (U(\lceil \alpha_{UF}(f) \rceil \cdot \lceil \alpha_{UG}(g) \rceil) \circ U\lceil (\eta_{R^{(K)}_{a,b}} \cdot \eta_{R^{(K)}_{b,c}}) \circ \delta^{R^{(K)}}_{a,b,c} \rceil)_{Jc} \circ k_{a,c} \\
= \quad & \{ \text{ Definition of } \delta^{UR^{(K)*}} \ \} \\
& U(\lceil \alpha_{UF}(f) \rceil \cdot \lceil \alpha_{UG}(g) \rceil)_{Jc} \circ \delta^{UR^{(K)*}}_{a,b,c,Jc} \circ k_{a,c}
\end{aligned}$$

□

We note that Lemma 4.8 follows from Lemma A.8 by considering the identity adjunction between $\mathscr{E}$ and itself.

Lemma A.9
For all $F, G : \mathscr{F}$ and $f : Ka \to UF(Kb)$ and $g : Kb \to UG(Kc)$, we have that

$$UF(\gamma^{-1}(k_{b,c})_G(g)) \circ \gamma^{-1}(k_{a,b})_F(f) \quad = \quad U(\lceil \alpha_{UF}(f) \rceil \cdot \lceil \alpha_{UG}(g) \rceil)_{Jc} \circ UR^{(K)*}_{a,b}(k_{b,c}) \circ k_{a,b}$$

Proof

$$\begin{aligned}
& UF(\gamma^{-1}(k_{b,c})_G(g)) \circ \gamma^{-1}(k_{a,b})_F(f) \\
= \quad & \{ \text{ Definition of } \gamma^{-1} \ \} \\
& UF(U\lceil \alpha_{UG}(g) \rceil_{Jc} \circ k_{b,c}) \circ U\lceil \alpha_{UF}(f) \rceil_{Jb} \circ k_{a,b}
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{\ UF \text{ is a functor } \} \\
&\quad UF(U\lceil\alpha_{UG}(g)\rceil_{Jc}) \circ UF(k_{b,c}) \circ U\lceil\alpha_{UF}(f)\rceil_{Jb} \circ k_{a,b} \\
&= \quad \{\ U\lceil\alpha_{UF}(f)\rceil \text{ is natural } \} \\
&\quad UF(U\lceil\alpha_{UG}(g)\rceil_{Jc}) \circ U\lceil\alpha_{UF}(f)\rceil_{UR^{(K)*}_{b,c}(Jc)} \circ UR^{(K)*}_{a,b}(k_{b,c}) \circ k_{a,b} \\
&= \quad \{\ \text{Definition of } \cdot\ \} \\
&\quad (U\lceil\alpha_{UF}(f)\rceil \cdot U\lceil\alpha_{UG}(g)\rceil)_{Jc} \circ UR^{(K)*}_{a,b}(k_{b,c}) \circ k_{a,b} \\
&= \quad \{\ U \text{ is strict } \} \\
&\quad U(\lceil\alpha_{UF}(f)\rceil \cdot \lceil\alpha_{UG}(g)\rceil)_{Jc} \circ UR^{(K)*}_{a,b}(k_{b,c}) \circ k_{a,b}
\end{aligned}
$$

□

We note that Lemma 4.9 follows from Lemma A.9 by considering the identity adjunction between $\mathscr{E}$ and itself.

The linearity law for the image of γ^{-1} states

$$\forall F, G, f, g. \gamma^{-1}(k_{a,c})_{F \cdot G}(UFg \circ f) = UF(\gamma^{-1}(k_{b,c})_G(g)) \circ \gamma^{-1}(k_{a,b})_F(f)$$

By the previous two lemmas, this linearity law is equivalent to stating that $\forall F, G, f, g$

$$U(\lceil\alpha_{UF}(f)\rceil \cdot \lceil\alpha_{UG}(g)\rceil)_{Jc} \circ \delta^{UR^{(K)*}}_{a,b,c,Jc} \circ k_{a,c} = U(\lceil\alpha_{UF}(f)\rceil \cdot \lceil\alpha_{UG}(g)\rceil)_{Jc} \circ UR^{(K)*}_{a,b}(k_{b,c}) \circ k_{a,b}.$$

With this reformulation, we see that the comultiplication-coalgebra law,

$$\delta^{UR^{(K)*}}_{a,b,c,Jc} \circ k_{a,c} \quad = \quad UR^{(K)*}_{a,b}(k_{b,c}) \circ k_{a,b}$$

trivially implies the linearity law. To derive the comultiplication-coalgebra law from the linearity law, consider the instance where $F = R^{(K)}_{a,b}$, $f = \alpha^{-1}_{UR^{(K)*}_{a,b}}(\eta_{R^{(K)}_{a,b}})$, $G = R^{(K)}_{b,c}$, and $g = \alpha^{-1}_{UR^{(K)*}_{b,c}}(\eta_{R^{(K)}_{b,c}})$. In this case we have

$$
\begin{aligned}
&\quad U(\lceil\alpha_{UF}(f)\rceil \cdot \lceil\alpha_{UG}(g)\rceil) \\
&= \quad \{\ \text{definition of } f \text{ and } g\ \} \\
&\quad U(\lceil\alpha_{UR^{(K)*}_{a,b}}(\alpha^{-1}_{UR^{(K)*}_{a,b}}(\eta_{R^{(K)}_{a,b}}))\rceil \cdot \lceil\alpha_{UR^{(K)*}_{b,c}}(\alpha^{-1}_{UR^{(K)*}_{b,c}}(\eta_{R^{(K)}_{b,c}}))\rceil) \\
&= \quad \{\ \text{isomorphism } \} \\
&\quad U(\lceil\eta_{R^{(K)}_{a,b}}\rceil \cdot \lceil\eta_{R^{(K)}_{b,c}}\rceil) \\
&= \quad \{\ \text{Equation (2.7)} \} \\
&\quad U(\lceil\lfloor id\rfloor\rceil \cdot \lceil\lfloor id\rfloor\rceil) \\
&= \quad \{\ \text{isomorphism } \} \\
&\quad U(id \cdot id) \\
&= \quad \{\ \text{identity } \} \\
&\quad id
\end{aligned}
$$

and then the comultiplication-coalgebra law follows.

The unity law for the image of γ^{-1} states

$$\gamma^{-1}(k_{a,a})_I(id) \quad = \quad id : Ja \to Ja$$

The counit-coalgebra law states

$$\varepsilon^{UR^{(K)*}} \circ k_{a,a} \quad = \quad id : Ja \to Ja$$

Therefore, in order to show that these laws are equivalent, it suffices to prove the following.

$$\gamma^{-1}(k_{a,a})_I(id) \quad = \quad \varepsilon^{UR^{(K)*}} \circ k_{a,a}$$

$\gamma^{-1}(k_{a,a})_I(id)$

$=$ { definition of γ^{-1} }

$U(\lceil \alpha_I(id) \rceil)(Ja) \circ k_{a,a}$

$=$ { Proposition 4.5 }

$U(\lceil \varepsilon_a^{R^{(K)}} \rceil)(Ja) \circ k_{a,a}$

$=$ { Definition of $\varepsilon^{UR^{(K)*}}$ }

$\varepsilon^{UR^{(K)*}} \circ k_{a,a}$

□