What You Needa Know about Yoneda

Profunctor Optics and the Yoneda Lemma (Functional Pearl)

GUILLAUME BOISSEAU, Department of Computer Science, University of Oxford, UK
JEREMY GIBBONS, Department of Computer Science, University of Oxford, UK

Profunctor optics are a neat and composable representation of bidirectional data accessors, including lenses, and their dual, prisms. The profunctor representation exploits higher-order functions and higher-kinded type constructor classes, but the relationship between this and the familiar representation in terms of 'getter' and 'setter' functions is not at all obvious. We derive the profunctor representation from the concrete representation, making the relationship clear. It turns out to be a fairly direct application of the Yoneda Lemma, arguably the most important result in category theory. We hope this derivation aids understanding of the profunctor representation. Conversely, it might also serve to provide some insight into the Yoneda Lemma.

CCS Concepts: • **Theory of computation** → **Program constructs**; **Categorical semantics**; *Type theory*; • **Software and its engineering** → **Abstract data types**; **Data types and structures**; **Patterns**; *Polymorphism*; *Semantics*;

Additional Key Words and Phrases: Lens, prism, optic, profunctors, composable references, Yoneda Lemma.

ACM Reference Format:
Guillaume Boisseau and Jeremy Gibbons. 2018. What You Needa Know about Yoneda: Profunctor Optics and the Yoneda Lemma (Functional Pearl). *Proc. ACM Program. Lang.* 2, ICFP, Article 84 (September 2018), 27 pages. https://doi.org/10.1145/3236779

1 INTRODUCTION

There is a saying in English that *you can tell a lot about a person by the company they keep*. Similarly, the Dutch say *zeg me wie uw vrienden zijn, dan zeg ik wie u bent*—"tell me who your friends are, and I will tell you who you are"; and the Japanese for 'human being' is 人間, constructed from the two kanji 人 ('nin', meaning 'person') and 間 ('gen', meaning 'between'), conveying the idea that "you as a human being only exist through your relations with others" [Matsumoto 2018].

The sentiment that we are defined by our relationships with others evidently transcends national boundaries. Indeed, it transcends human interaction altogether: one might say that it is the essence of the *Yoneda Lemma*, which has been called "arguably the most important result in category theory" [Riehl 2016, p57]. The formal statement

> **Yoneda Lemma:** For $\mathbb{C}$ a locally small category, $[\mathbb{C}, \mathbb{S}et](\mathbb{C}(A, -), F) \simeq F(A)$, naturally in $A \in \mathbb{C}$ and $F \in [\mathbb{C}, \mathbb{S}et]$.

of the result is quite formidably terse—we explain it gently in Section 2—but roughly speaking, it says that a certain kind of object (on the right of the isomorphism) is fully determined its relationships (on the left).

Authors' addresses: Guillaume Boisseau, guillaume.boisseau@cs.ox.ac.uk, Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; Jeremy Gibbons, jeremy.gibbons@cs.ox.ac.uk, Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/9-ART84
https://doi.org/10.1145/3236779

The high level of abstraction in the statement of the Yoneda Lemma means that it can be applied in many contexts, some of which are actually quite familiar. For example, it explains Cayley's Theorem for monoids (which is the trick that enables the use of an accumulating parameter, which can often turn a quadratic-time program into a linear-time one), proofs by 'indirect equality' (that is, $b \leqslant a$ if and only if $\forall c\,.\,(a \leqslant c) \Rightarrow (b \leqslant c)$, an important technique in program calculation), and the correctness of both closure conversion and translation to continuation-passing style. We discuss these applications in detail in Section 3.

More broadly, one might see shadows of Yoneda in the Euclidean axiom that there is nothing more to a geometrical point than the lines that meet there, and in the way that artificial intelligence can attribute its successes to the position that there is nothing more to the 'semantics' of an entity than syntactic references to other entities. Mazzola [2002, Chapter 9] identifies a whole *Yoneda Philosophy* in the arts: understanding a thing through its relationships with other things. Mazzola cites Paul Valéry's dictum that "c'est l'exécution du poème qui est le poème" [Valéry 1937] (that is, the essence of a poem is determined by its readings, public or private, and not just its text), Theodor Adorno's statement that "die Idee der Interpretation gehört zur Musik selber und ist ihr nicht akzidentiell" [Adorno 1956] (that is, performance is an essential aspect of a composition's identity), and the common experience of walking around a sculpture to see it from all angles in order fully to appreciate it.

Our goal in this paper is to use the Yoneda Lemma to derive *profunctor optics*, a surprisingly flexible higher-order datatype-generic representation of data accessors such as lenses and prisms. We define these notions properly in Section 2.3, but summarize briefly here for accessibility. Concretely, a *lens* of type $Lens\, S\, T\, A\, B$ provides access via a 'getter' and a 'setter' onto a component of type A within a composite product structure of type S, allowing that component to be replaced by one of type B, yielding a new composite of type T [Foster et al. 2005; Pickering et al. 2017]:

$$\textbf{data}\ Lens\ a\ b\ s\ t = Lens\,\{\, view :: s \rightarrow a,\ update :: s \times b \rightarrow t \,\}$$

The *Profunctor* class

$$\begin{array}{l}\textbf{class}\ Profunctor\ p\ \textbf{where} \\ \quad dimap :: (c \rightarrow a) \rightarrow (b \rightarrow d) \rightarrow p\ a\ b \rightarrow p\ c\ d\end{array}$$

represents 'transformer' type constructors P, such that a transformer of type $P\,A\,B$ 'consumes' A values and 'produces' B values. A *cartesian* (also called *strong*) profunctor is one that is in addition coherent with products:

$$\begin{array}{l}\textbf{class}\ Profunctor\ p \Rightarrow Cartesian\ p\ \textbf{where} \\ \quad second :: p\ a\ b \rightarrow p\ (c \times a)\ (c \times b)\end{array}$$

Now, a *profunctor lens* of type $LensP\, A\, B\, S\, T$ is a mechanism to lift a component-transformer of type $P\,A\,B$ to a composite-transformer of type $P\,S\,T$, for any cartesian profunctor P:

$$\textbf{type}\ LensP\ a\ b\ s\ t = \forall p\,.\, Cartesian\ p \Rightarrow p\ a\ b \rightarrow p\ s\ t$$

The somewhat surprising fact is that $Lens\, A\, B\, S\, T$ and $LensP\, A\, B\, S\, T$ are naturally isomorphic types. (To be precise, one usually imposes 'well-behavedness' or 'round-tripping' laws on lenses. These laws appear to be completely orthogonal to the choice of representation, and to the isomorphism, so we do not address them in this paper.)

There is a dual story for *prisms*, which provide access onto a component in a composite sum type, in terms of *cocartesian profunctors*. Lenses and prisms are particular kinds of *optic*. The profunctor representation *LensP* has a number of benefits over the concrete representation *Lens* [Pickering et al. 2017]. For one thing, the profunctor representation is in terms of ordinary functions, and so profunctor optics compose using ordinary function composition. For a second, the profunctor

representation readily supports *heterogeneous* optics, providing access onto components of a sum-of-products composite, which cannot be captured using the concrete representations.

We provide a new proof of the equivalence of the concrete and profunctor representations of optics. The proof depends heavily on the Yoneda Lemma. It can be expressed in great generality, applying to lenses, prisms, and other kinds of optic too. It is considerably simpler and more straightforward than previous proofs—both our own [Pickering et al. 2017] and others' [Milewski 2017b].

The remainder of this paper is structured as follows. In Section 2, we introduce the necessary background in category theory, and more precisely define profunctor optics. Section 3 describes a number of more or less familiar concrete applications of the Yoneda Lemma, in order to provide some intuition for what follows. The main content of the paper is Section 4, presenting our proof using the Yoneda Lemma that profunctor optics are equivalent to their concrete cousins. Section 5 concludes, with a summary, discussion, and thoughts for future work.

2 BACKGROUND

2.1 Notational Conventions

We conduct our proofs using categorical notations, for generality, but translate the ideas into a programming language for concreteness. We choose Haskell as that language, but the choice is not particularly important. We do use types as partial specifications; in order to express various abstractions at the type level, one does require generics of higher kind—so Scala would also suffice for this aspect, but Java would not. But static types are not essential, and the ideas would also work in a dynamically typed language.

Although we use Haskell as a programming notation, we make some simplifications. We ignore non-termination and partially defined values, treating Haskell types as sets and Haskell programs as total functions. We write '$A \times B$' for the Haskell pair type (A, B), and '$A + B$' for the sum type *Either A B*. We sometimes subscript methods of a type class, to give a hint as to which instance is being used; for example, we might write '$\mathit{fmap}_{\mathit{List}}$' to indicate the *List* instance of *fmap*. Where in Haskell one would write '`forall`' outside a datatype constructor to indicate what is effectively an existential quantification, we actually write '$\exists$'; for example,

$$\textbf{data}\ \mathit{Boolable} = \exists a\ .\ \mathit{Boolable}\ (a, a \to \mathit{Bool})$$

We write '$f : A \simeq B : g$' to indicate that $f : A \to B$ and $g : B \to A$ form an isomorphism.

2.2 Categorical Prerequisites

Categories. A *category* $\mathbb{C}$ consists of:

- a collection $|\mathbb{C}|$ of *objects*;
- a set $\mathbb{C}(A, B)$ of *arrows* from A to B for each pair of objects $A, B \in |\mathbb{C}|$;
- an *identity* arrow $id_A \in \mathbb{C}(A, A)$ for each object $A \in |\mathbb{C}|$;
- the *composition* $g \circ f \in \mathbb{C}(A, C)$ for each pair of compatible arrows $f \in \mathbb{C}(A, B)$ and $g \in \mathbb{C}(B, C)$,

such that

- composition is associative; and
- appropriate identity arrows serve as neutral elements for composition.

The sets $\mathbb{C}(A, B)$ are called *homsets*. (Technically, this is the definition of a *locally small* category; for cardinality reasons, sometimes the collection of arrows between a pair of objects can be too large to be a set. We gloss over such technicalities.)

One very familiar example is the category $\mathbb{S}et$, whose objects are sets, with the homset $\mathbb{S}et(A, B)$ consisting of the set of total functions from A to B; we can see the objects and arrows as representations of types and programs, respectively. But we will also draw inspiration (and later illustrations) from two other examples. Any preordered set $(A, \leqslant)$ induces a category $\mathbb{P}re(A, \leqslant)$, whose objects are the elements of A, with the homset $\mathbb{P}re(A, \leqslant)(a, b)$ consisting of a single element when $a \leqslant b$ and empty otherwise; reflexivity of $\leqslant$ accounts for the identity arrows, and transitivity of $\leqslant$ for composition. And any monoid $(M, \oplus, e)$ induces a category $\mathbb{M}on(M, \oplus, e)$, with a single object $*$, and sole homset $\mathbb{M}on(M, \oplus, e)(*, *)$ consisting of the elements of M; the neutral element e of the monoid forms the identity arrow, and composition is given by $\oplus$. Indeed, one can see category theory as a kind of common generalization of preorders and monoids.

Any category $\mathbb{C}$ has an *opposite* category $\mathbb{C}^{op}$, with the same objects but reversed arrows:

- the objects $|\mathbb{C}^{op}|$ are the objects $|\mathbb{C}|$;
- the arrows $\mathbb{C}^{op}(A, B)$ from A to B in $\mathbb{C}^{op}$ are the arrows $\mathbb{C}(B, A)$ from B to A in $\mathbb{C}$;
- composition is backwards.

Given two categories $\mathbb{C}, \mathbb{D}$, the *product category* $\mathbb{C} \times \mathbb{D}$ has pairs of objects and pairs of arrows:

- an object $(A, B) \in |\mathbb{C} \times \mathbb{D}|$ for each pair of objects $A \in |\mathbb{C}|$ and $B \in |\mathbb{D}|$;
- an arrow $(f, g) \in (\mathbb{C} \times \mathbb{D})((A, B), (C, D))$ for each pair of arrows $f \in \mathbb{C}(A, C)$ and $g \in \mathbb{D}(B, D)$;
- composition is pointwise.

Functors. Categories are evidently structured entities, so one should consider the structure-preserving mappings between categories, called *functors*. Formally, a functor $F : \mathbb{C} \to \mathbb{D}$ from category $\mathbb{C}$ to category $\mathbb{D}$ is a mapping from the objects and arrows of $\mathbb{C}$ to those of $\mathbb{D}$, respecting the structure:

- an object $F(A) \in |\mathbb{D}|$ for each object $A \in |\mathbb{C}|$;
- an arrow $F(f) \in \mathbb{D}(F(A), F(B))$ for each arrow $f \in \mathbb{C}(A, B)$,

such that

- $F(id_A) = id_{F(A)}$ for each object $A \in |\mathbb{C}|$;
- $F(g \circ f) = F(g) \circ F(f)$ for each pair f, g of compatible arrows.

One class of functors $F : \mathbb{S}et \to \mathbb{S}et$ on $\mathbb{S}et$ are the *container types*; for example, the functor $List : \mathbb{S}et \to \mathbb{S}et$ takes the set A of elements to the set $List(A)$ of finite lists of elements drawn from A, and acts on arrows as the 'map' operation, taking $f : A \to B$ to $List(f) : List(A) \to List(B)$. (A functor $F : \mathbb{C} \to \mathbb{C}$ from a category to itself is called an *endofunctor*.)

For any category $\mathbb{C}$ and objects $A, B \in |\mathbb{C}|$, the homset $\mathbb{C}(A, B)$ is a set of arrows; so $\mathbb{C}(A, -)$ is a function from $|\mathbb{C}|$ to $|\mathbb{S}et|$, taking object B to set $\mathbb{C}(A, B)$. This function on objects can be extended to a function on arrows too, taking $f \in \mathbb{C}(B, C)$ to $(f\circ) \in \mathbb{S}et(\mathbb{C}(A, B), \mathbb{C}(A, C))$, that is, the function that takes $g \in \mathbb{C}(A, B)$ to $f \circ g \in \mathbb{C}(A, C)$. This makes $\mathbb{C}(A, -)$ a functor from $\mathbb{C}$ to $\mathbb{S}et$, called the *homfunctor*. Similarly, $\mathbb{C}(-, B)$ is a functor, taking arrow f to $(\circ f)$; but this action on arrows is contravariant, so this homfunctor is a functor from $\mathbb{C}^{op}$ to $\mathbb{S}et$.

Functors $\mathbb{P}re(A, \leqslant) \to \mathbb{P}re(B, \sqsubseteq)$ correspond to monotonic functions between preordered sets, and functors $\mathbb{M}on(M, \oplus, e) \to \mathbb{M}on(N, \otimes, e')$ correspond to homomorphisms between monoids. A functor $F : \mathbb{M}on(M, \oplus, e) \to \mathbb{S}et$ takes the sole object $*$ to some set $S \in |\mathbb{S}et|$ and each arrow $m : * \to *$ to a function $S \to S$ in a way that respects $\oplus$ and e, so amounts to a *left action* of the monoid $(M, \oplus, e)$ on S. In particular, the homfunctor $\mathbb{M}on(M, \oplus, e)(*, -) : \mathbb{M}on(M, \oplus, e) \to \mathbb{S}et$ takes sole object $*$ to the set M, and each arrow $m : * \to *$ to the function $(m \oplus -) : M \to M$, so it is the left action of $(M, \oplus, e)$ on M itself.

It is a worthwhile exercise to verify all of these assertions.

Natural Transformations. Functors too are evidently structured entities, so one should consider the structure-preserving mappings between them, which are called *natural transformations*. Formally, given two functors $F, G : \mathbb{C} \to \mathbb{D}$ between the same two categories $\mathbb{C}$ and $\mathbb{D}$, a natural transformation $\phi : F \stackrel{.}{\to} G$ from F to G is a family of arrows in $\mathbb{D}$, indexed by objects in $\mathbb{C}$:

- an arrow $\phi_A \in \mathbb{D}(F(A), G(A))$ for each object $A \in |\mathbb{C}|$,

satisfying the *naturality condition*

- $\phi_B \circ F(f) = G(f) \circ \phi_A$ for each arrow $f \in \mathbb{C}(A, B)$.

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle\phi_A}\downarrow & & \downarrow{\scriptstyle\phi_B} \\ G(A) & \xrightarrow[G(f)]{} & G(B) \end{array}$$

The natural transformations between endofunctors on $\mathbb{S}et$ correspond to polymorphic functions. For example, $reverse : List \stackrel{.}{\to} List$ is a natural transformation from the list functor to itself; and if $Tree$ is the $\mathbb{S}et$ functor representing finite node-labelled binary trees of elements, then the function $inorder$ that computes the in-order traversal is a natural transformation $Tree \stackrel{.}{\to} List$. The naturality condition for $inorder$ states that mapping a function $f : A \to B$ over a $Tree(A)$ and then traversing the tree yields the same result as traversing and then mapping over the ensuing $List(A)$:

$$inorder_B \circ Tree(f) = List(f) \circ inorder_A$$

Given two functors $f, g : \mathbb{P}re(A, \leqslant) \to \mathbb{P}re(B, \sqsubseteq)$—that is, monotonic functions—between two preorders, a natural transformation $\phi : f \stackrel{.}{\to} g$ is a witness to the pointwise ordering $f \sqsubseteq g$ between the functions, in the sense that $\phi_a : f(a) \sqsubseteq g(a)$ for each $a \in A$.

For any two categories $\mathbb{C}, \mathbb{D}$, the *functor category* $[\mathbb{C}, \mathbb{D}]$ has as objects the functors from $\mathbb{C}$ to $\mathbb{D}$, and as arrows $[\mathbb{C}, \mathbb{D}](F, G)$ between objects $F, G \in |[\mathbb{C}, \mathbb{D}]|$ the natural transformations $F \stackrel{.}{\to} G$.

The Yoneda Lemma. The Yoneda Lemma states that for (locally small) category $\mathbb{C}$, there is a bijection

$$\lceil - \rceil^{A,F} : [\mathbb{C}, \mathbb{S}et](\mathbb{C}(A, -), F) \simeq F(A) : \lfloor - \rfloor^{A,F}$$

natural in object $A \in \mathbb{C}$ and functor $F \in [\mathbb{C}, \mathbb{S}et]$. Let us unpack what that says, following Leinster [2000]. The lemma asserts a bijection between two sets. Apart from category $\mathbb{C}$, the two free variables are the object $A \in \mathbb{C}$ and the functor $F : \mathbb{C} \to \mathbb{S}et$. In addition to functor F, also appearing is the homfunctor $\mathbb{C}(A, -)$, another functor $\mathbb{C} \to \mathbb{S}et$. On the left of the bijection is a homset in the functor category $[\mathbb{C}, \mathbb{S}et]$; that is, the set of natural transformations $\phi : \mathbb{C}(A, -) \stackrel{.}{\to} F$, namely families of functions $\phi_B : \mathbb{C}(A, B) \stackrel{.}{\to} F(B)$ in $\mathbb{S}et$ satisfying an appropriate naturality condition. On the right is simply the set $F(A)$.

The proof of the lemma amounts to constructing the two functions that form the bijection. From left to right, the function $\lceil - \rceil^{A,F}$ should take a natural transformation $\phi : \mathbb{C}(A, -) \stackrel{.}{\to} F$ and yield an element of the set $F(A)$. That is achieved by picking the Ath component $\phi_A : \mathbb{C}(A, A) \to F(A)$ of the family ϕ, and applying this to the identity arrow $id_A \in \mathbb{C}(A, A)$:

$$\lceil \phi \rceil^{A,F} = \phi_A(id_A)$$

From right to left, the function $\lfloor - \rfloor^{A,F}$ should take an element x of the set $F(A)$ and yield a natural transformation $\mathbb{C}(A, -) \stackrel{.}{\to} F$. We construct such a natural transformation ϕ from x as follows. For object $B \in |\mathbb{C}|$, the Bth component ϕ_B should be an arrow in $\mathbb{S}et$—that is, a function—from $\mathbb{C}(A, B)$ to $F(B)$. Given an $f \in \mathbb{C}(A, B)$, we can map f over x using the functorial action of F to yield a value of the correct type:

$$\lfloor x \rfloor^{A,F}_B(f) = F(f)(x)$$

It is worth noting the essence of this construction: in the $\lceil - \rceil$ direction, there is an 'application to the identity arrow', and in the $\lfloor - \rfloor$ direction, 'use of the functorial action'. We will see this essence also in the various applications of the Yoneda Lemma in the rest of the paper.

The two naturality conditions amount to the following. Given an arrow $g \in \mathbb{C}(A,C)$ and a natural transformation $\phi : F \stackrel{.}{\to} G$, it does not matter whether one uses the isomorphism $(\lceil - \rceil^{A,F}, \lfloor - \rfloor^{A,F})$ at object A and functor F, then transforms A and F to C and G using g and ϕ, or one transforms A, F to C, G using g, ϕ then uses the isomorphism $(\lceil - \rceil^{C,G}, \lfloor - \rfloor^{C,G})$ at object C and functor G. However, we will make no further explicit use of these naturality conditions, so we do not attempt to spell them out in greater detail.

To complete the proof, we would need to show that $\lfloor x \rfloor^{A,F}$ really is a natural transformation for any $x \in F(A)$, that $\lceil - \rceil^{A,F}$ and $\lfloor - \rfloor^{A,F}$ are each other's inverses, and that the two additional naturality conditions hold. The proofs are standard, so we direct the curious reader to the literature [Awodey 2006; Mac Lane 1971]. We present a number of illuminating examples in Section 3.

There is a dual version of the Yoneda Lemma, expressed in terms of the other homfunctor $\mathbb{C}(-,B)$ which is obtained by fixing the target rather than the source object of the homset. This homfunctor is contravariant, that is, a functor from $\mathbb{C}^{\mathrm{op}}$ to $\mathbb{S}et$; so we must also have $F : \mathbb{C}^{\mathrm{op}} \to \mathbb{S}et$. Then Yoneda states that

$$[\mathbb{C}^{\mathrm{op}}, \mathbb{S}et](\mathbb{C}(-,B), F) \simeq F(B)$$

(Equivalently, this contravariant version of the Yoneda Lemma is obtained immediately by considering the covariant version in the category $\mathbb{C}^{\mathrm{op}}$. So this is not a separate result, but an alternative presentation of the original result.)

The whole homfunctor $\mathbb{C}(-,=)$ of a category $\mathbb{C}$ is a functor $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{S}et$. Since it is a functor from a product category, we can curry it; doing so yields a functor $\mathbb{C}^{\mathrm{op}} \to [\mathbb{C}, \mathbb{S}et]$, sometimes written '$H^{\bullet}$'. Then we have:

Theorem 2.1 (Yoneda Embedding). *The functor $H^{\bullet} : \mathbb{C}^{\mathrm{op}} \to [\mathbb{C}, \mathbb{S}et]$ is full and faithful, and injective on objects.*

A functor is called *full* when it is surjective on each homset, and *faithful* when it is injective. The 'full and faithful' part of the theorem is a corollary of Yoneda, obtained by instantiating the functor F in the statement of the Yoneda Lemma also to a homfunctor $\mathbb{C}(B,-)$ to yield the equivalence

$$[\mathbb{C}, \mathbb{S}et](\mathbb{C}(A,-), \mathbb{C}(B,-)) \simeq \mathbb{C}(B,A)$$

for all objects $A, B \in |\mathbb{C}|$. Indeed, the mapping $\lfloor = \rfloor^{A,\mathbb{C}(B,-)}$ from right to left takes an arrow $f \in \mathbb{C}(B,A)$ to $\mathbb{C}(f,-) = (\circ f)$, and the Yoneda Lemma proves that this mapping forms an isomorphism. The 'injective on objects' part of the theorem is automatic, since by convention distinct homsets of a category are disjoint.

2.3 Profunctor Optics

As already summarized briefly in Section 1, a *lens* provides access onto component within a composite product data structure. The concrete representation $\mathit{Lens}\ A\ B\ S\ T$ of type-varying lenses consists of a pair of functions: a 'getter' $\mathit{view} :: S \to A$ which extracts a view of type A from a source of type S; and a 'setter' $\mathit{update} :: S \times B \to T$ which inserts an updated view of type B (possibly different from A) into an old source of type S to yield an updated source of type T (correspondingly, possibly different from S).

$$\mathbf{data}\ \mathit{Lens}\ a\ b\ s\ t = \mathit{Lens}\,\{\, \mathit{view} :: s \to a, \mathit{update} :: s \times b \to t \,\}$$

For example, there is an obvious lens onto the second component of a pair:

$$\mathit{sndLens} :: \mathit{Lens}\ a\ b\ (c \times a)\ (c \times b)$$
$$\mathit{sndLens} = \mathit{Lens}\ \mathit{vw}\ \mathit{up}\ \mathbf{where}$$

$$\begin{aligned} &vw\,(c,a) &&= a \\ &up\,((c,a),a') &&= (c,a') \end{aligned}$$

The view function vw extracts the second component, and the update function up replaces that component with a new one.

Dually, a *prism* provides access onto a component within a composite sum data structure. The concrete representation $Prism\,A\,B\,S\,T$ of type-varying prisms consists of a pair of functions: a 'getter' $match :: S \to T + A$ which 'downcasts' the source of type S to a view of type A, if possible; and a 'setter' $build :: B \to T$ which 'upcasts' an updated view of type B (possibly different from S) back to an updated source of type T (correspondingly, possibly different from S). When the getter cannot downcast the source to type A, then the source must inhabit another variant of the sum structure, and so it can be directly coerced by $match$ to the updated source type T.

$$\mathbf{data}\; Prism\;a\;b\;s\;t = Prism\,\{\,match :: s \to t + a,\; build :: b \to t\,\}$$

For example, there is a prism onto an optional value:

$$\begin{aligned} &the :: Prism\;a\;b\;(Maybe\;a)\;(Maybe\;b) \\ &the = Prism\;mt\;bd\;\mathbf{where} \\ &\quad mt\;(Just\;a) \;= Right\;a \\ &\quad mt\;Nothing = Left\;Nothing \\ &\quad bd\;b = Just\;b \end{aligned}$$

The match function mt extracts the optional value when it is present, and yields the complement (that is, $Nothing$) when it is absent; and the build function bd injects an updated value.

Lenses and prisms turn out to be divergent generalizations of a common specialization, called *adapters*, which provide access onto a value via a change of representation:

$$\mathbf{data}\; Adapter\;a\;b\;s\;t = Adapter\,\{from :: s \to a,\; to :: b \to t\,\}$$

For example, there is an adapter allowing one to act on a nested pair as if it were a triple:

$$\begin{aligned} &flatten :: Adapter\;(a \times b \times c)\;(a' \times b' \times c')\;((a \times b) \times c)\;((a' \times b') \times c') \\ &flatten = Adapter\;fro\;to\;\mathbf{where} \\ &\quad fro\;((a,b),c) = (a,b,c) \\ &\quad to\;(a,b,c) = ((a,b),c) \end{aligned}$$

(writing '$(a \times b \times c)$' for a type of triples). An adapter is equivalent to a lens in which the view completely determines the source, so that the *update* function need not be provided with the old source; and conversely, an adapter is equivalent to a prism in which there is only one variant, so that the *match* downcast always succeeds.

The *Functor* class in Haskell

$$\begin{aligned} &\mathbf{class}\; Functor\;f\;\mathbf{where} \\ &\quad fmap :: (a \to b) \to f\;a \to f\;b \\ &\qquad \text{-- law: } fmap\,(g \circ h) = fmap\,g \circ fmap\,h \\ &\qquad \text{-- law: } fmap\;id = id \end{aligned}$$

represents container types, in the sense that for a functor F, a value of type $F\,A$ 'contains' elements of type A, and so might in principle deliver values of that type. They are analogous to categorical endofunctors on $\mathbb{S}et$. The *Profunctor* class

$$\begin{aligned} &\mathbf{class}\; Profunctor\;p\;\mathbf{where} \\ &\quad dimap :: (c \to a) \to (b \to d) \to p\;a\;b \to p\;c\;d \end{aligned}$$

$$P\,A\,B \xrightarrow[\mathit{dimap\ lunit\ lunit'}]{\mathit{second}_1} P\,(1\times A)\,(1\times B)$$

$$\begin{array}{ccc} P\,A\,B & \xrightarrow[\mathit{second}_D]{} & P\,(D\times A)\,(D\times B) \\ \downarrow \mathit{second}_{C\times D} & & \downarrow \mathit{second}_C \\ P\,((C\times D)\times A)\,((C\times D)\times B) & \xleftarrow[\mathit{dimap\ assoc\ assoc'}]{} & P\,(C\times(D\times A))\,(C\times(D\times B)) \end{array}$$

Fig. 1. Cartesian profunctor laws

-- law: $\mathit{dimap}\,(f' \circ f)\,(g \circ g') = \mathit{dimap}\,f\,g \circ \mathit{dimap}\,f'\,g'$
-- law: $\mathit{dimap}\;\mathit{id}\;\mathit{id} = \mathit{id}$

generalizes this to 'transformer' types, which may both consume and produce values: for profunctor P, a transformer of type $P\,A\,B$ 'consumes' A values and 'produces' B values—note that the *dimap* operator which modifies the consumed and produced values is contravariant in the 'consumed' argument. The canonical *Profunctor* instance is the function type former, for which *dimap* is obtained simply by function composition:

instance *Profunctor* $(\rightarrow)$ **where**
  $\mathit{dimap}\,f\,g\,h = g \circ h \circ f$

A profunctor P is *cartesian* (or *strong*) if it is coherent with products, in the sense that there is an operator *second* that, for any type C, lifts a transformer of type $P\,A\,B$ to a transformer of type $P\,(C \times A)\,(C \times B)$ that acts on pairs, transforming As into Bs but passing the Cs around unchanged, respecting the isomorphisms

$\mathit{assoc} : (A \times B) \times C \simeq A \times (B \times C) : \mathit{assoc}'$
$\mathit{lunit} \;: 1 \times A \simeq A : \mathit{lunit}'$

of products:

class *Profunctor* $p \Rightarrow$ *Cartesian* p **where**
  $\mathit{second} :: p\,a\,b \rightarrow p\,(c \times a)\,(c \times b)$
    -- law: $\mathit{dimap}\;\mathit{assoc}\;\mathit{assoc}'\;(\mathit{second}_C\,(\mathit{second}_D\,h)) = \mathit{second}_{C\times D}\,h$
    -- law: $\mathit{dimap}\;\mathit{lunit}\;\mathit{lunit}'\;h = \mathit{second}_1\,h$

These laws are illustrated by the commutative diagrams in Figure 1.

The function type former is an instance:

instance *Cartesian* $(\rightarrow)$ **where**
  $\mathit{second}\,f\,(c, a) = (c, f\,a)$

(Given *second*, one can construct a symmetric operator *first* that transforms the first component of a pair.)

Dually, a profunctor P is *co-cartesian* (sometimes called *choice*) if it is coherent with the structure of sums

$\mathit{coassoc} : (A + B) + C \simeq A + (B + C) : \mathit{coassoc}'$
$\mathit{lzero} \quad : 0 + A \simeq A : \mathit{lzero}'$

in the same way:

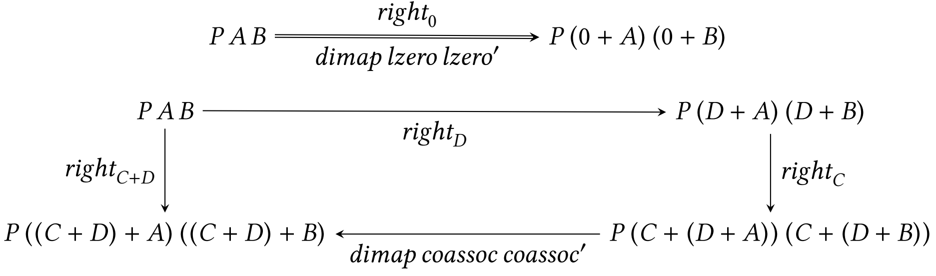

Fig. 2. Cocartesian profunctor laws

```
class Profunctor p ⇒ Cocartesian p where
  right :: p a b → p (c + a) (c + b)
    -- law: dimap coassoc coassoc′ (right_C (right_D h)) = right_{C+D} h
    -- law: dimap lzero lzero′ h = right_0 h
```

These laws are illustrated by similar commutative diagrams in Figure 2.

Again, the function type former is an instance:

```
instance Cocartesian (→) where
  right f (Left c)  = Left c
  right f (Right a) = Right (f a)
```

The profunctor representation $AdapterP\ A\ B\ S\ T$ of an adapter is as a function lifting a component-transformer $P\,A\,B$ to a composite-transformer $P\,S\,T$, that works *uniformly* for all profunctors P:

```
type AdapterP a b s t = ∀p . Profunctor p ⇒ p a b → p s t
```

For example, the profunctor representation *flattenP* of the concrete adapter *flatten* is:

```
flattenP :: AdapterP (a × b × c) (a′ × b′ × c′) ((a × b) × c) ((a′ × b′) × c′)
flattenP h = dimap f t h where
  f ((a, b), c) = (a, b, c)
  t (a, b, c)   = ((a, b), c)
```

which wraps its argument transformer in the two halves of the isomorphism.

Similarly, the profunctor representation $LensP\ A\ B\ S\ T$ of a lens is as a function lifting a component-transformer $P\,A\,B$ to a composite-transformer $P\,S\,T$, that works uniformly for all *cartesian* profunctors P:

```
type LensP a b s t = ∀p . Cartesian p ⇒ p a b → p s t
```

For example, the profunctor representation *sndLensP* of the concrete lens *sndLens* is:

```
sndLensP :: LensP a b (c × a) (c × b)
sndLensP h = second h
```

which is simply an application of the *second* method.

Dually, the profunctor representation $PrismP\ A\ B\ S\ T$ of a prism is as a function lifting a component-transformer $P\,A\,B$ to a composite-transformer $P\,S\,T$, that works uniformly for all *co-cartesian* profunctors P:

```
type PrismP a b s t = ∀p . Cocartesian p ⇒ p a b → p s t
```

For example, the profunctor representation *theP* of *the* is:

$$\begin{aligned}
&\mathit{theP} :: \mathit{PrismP}\ a\ b\ (\mathit{Maybe}\ a)\ (\mathit{Maybe}\ b) \\
&\mathit{theP}\ h = \mathit{dimap}\ \mathit{m2s}\ \mathit{s2m}\ (\mathit{right}\ h)\ \textbf{where} \\
&\quad \mathit{m2s}\ \mathit{Nothing} = \mathit{Left}\ () \\
&\quad \mathit{m2s}\ (\mathit{Just}\ x) = \mathit{Right}\ c \\
&\quad \mathit{s2m}\ (\mathit{Left}\ ()) = \mathit{Nothing} \\
&\quad \mathit{s2m}\ (\mathit{Right}\ x) = \mathit{Just}\ x
\end{aligned}$$

Given a transformer $h :: P\ A\ B$, we can lift it using *right* to a transformer of type $P\ (() + A)\ (() + B)$, and then use *dimap* to adapt this to one of type $P\ (\mathit{Maybe}\ A)\ (\mathit{Maybe}\ B)$ as required.

3 EXAMPLES OF YONEDA

The power of the Yoneda Lemma comes from its high level of abstraction; but that also makes it difficult to get to grips with at first. To provide some intuition, in this section we present a number of more or less familiar concrete applications of the Yoneda Lemma and Yoneda Embedding.

One way of gaining some intuition for an abstract result in category theory is to consider what it says in the two very special cases of preorder categories $\mathbb{P}re(A, \leqslant)$, where the homsets are all either empty or singletons, and monoid categories $\mathbb{M}on(M, \oplus, e)$, where there is only a single object; these are the two simplest kinds of category. As noted in Section 2, one can see category theory as the least common generalization of preorders and monoids. For example, one common route to gaining some intuition for the rather abstract notions of adjunction and monad is through their specialization to preorder categories, where they simplify to the perhaps more familiar Galois connections and to closure operators respectively.

3.1 Indirect Inequality

One important technique in the mathematics of program construction is the method of *proof by indirect inequality* [Feijen 2001; von Karger 2002], also called the *rule of indirect order* [Backhouse 2003]: for a preorder $(A, \leqslant)$,

$$(b \leqslant a) \Leftrightarrow (\forall c\,.\,(a \leqslant c) \Rightarrow (b \leqslant c))$$

The right-to-left direction is equivalent to reflexivity of $\leqslant$, and the left-to-right direction to transitivity [Dijkstra 1991]. In some situations, the right-hand side can be much easier to manipulate than the left. This equivalence is an instance of the Yoneda Embedding in the category $\mathbb{P}re(A, \leqslant)$. In this category, the homset $\mathbb{P}re(A, \leqslant)(b, a)$ is a 'thin set'—a singleton when $b \leqslant a$, and empty otherwise. The homfunctor $\mathbb{P}re(A, \leqslant)(a, -)$ is a 'thin function', taking an element $c \in A$ to a singleton set when $a \leqslant c$ and to the empty set otherwise. A natural transformation $\phi{:}\mathbb{P}re(A, \leqslant)(a, -) \stackrel{\cdot}{\to} \mathbb{P}re(A, \leqslant)(b, -)$ is a family of functions $\phi_c : \mathbb{P}re(A, \leqslant)(a, c) \to \mathbb{P}re(A, \leqslant)(b, c)$ between the thin sets. But the total functions between thin sets X and Y are highly constrained—indeed, they are completely determined, unless X is non-empty and Y is empty, in which case no such function can exist. So the family ϕ of functions is a witness to the fact that for each c, if $\mathbb{P}re(A, \leqslant)(a, c)$ is non-empty then so is $\mathbb{P}re(A, \leqslant)(b, c)$; that is, that $a \leqslant c$ implies $b \leqslant c$.

3.2 Indirect Equality

In a similar vein, Yoneda can yield elegant proofs of many equivalences, through the rule of *indirect equality* [Backhouse 2003; Feijen 2001]. One nice property of full and faithful functors is that they both *preserve and reflect isomorphisms*. Any functor preserves isomorphisms:

$$
\begin{array}{ll}
& F(f) \circ F(g) = id \\
\Leftrightarrow & [\![\;\; F \text{ is a functor} \;\;]\!] \\
& F(f \circ g) = F(id) \\
\Leftarrow & [\![\;\; \text{Leibniz} \;\;]\!] \\
& f \circ g = id
\end{array}
$$

That is, if f and g form an isomorphism, then so do $F(f)$ and $F(g)$. When F is faithful (injective on arrows), then the last step is an equivalence; in that case, when $F(f)$ and $F(g)$ form an isomorphism, so too do f and g. And when F is full (surjective on arrows), if $F(f)$ does have an inverse h, then h is in the range of F—there is a g such that $h = F(g)$. Thus, for a full and faithful functor F, the arrow f forms an isomorphism if and only if $F(f)$ does. The Yoneda Embedding provides a full and faithful functor; therefore we have

$$(\mathbb{C}(B,-) \simeq \mathbb{C}(A,-)) \Leftrightarrow (A \simeq B) \Leftrightarrow (\mathbb{C}(-,A) \simeq \mathbb{C}(-,B))$$

As Leinster [2000, §2.3] writes, if one regards $\mathbb{C}(-,A)(U) = \mathbb{C}(U,A)$ as 'A viewed from U', then this states that two objects A, B are the same if and only if they look the same from all viewpoints U; this is Mazzola's 'Yoneda Philosophy' as discussed in Section 1.

For a preorder $(A, \leqslant)$, this equivalence manifests itself as the rule of indirect equality that forms a significant tool in Backhouse's textbook [Backhouse 2003]:

$$(b = a) \Leftrightarrow (\forall c \,.\, (a \leqslant c) \Leftrightarrow (b \leqslant c))$$

For another example, following Awodey [2006, §8.4], the category $\mathbb{S}et$ supports *currying*—a function from a product type is equivalent to a function to an exponential, that is, a function of one argument that yields a function of the other argument:

$$\mathbb{S}et(A \times C, B) \simeq \mathbb{S}et(A, B^C)$$

and the sum type constructor satisfies the universal property that a function from a sum type is equivalent to a pair of functions with a common target:

$$\mathbb{S}et(A + B, C) \simeq \mathbb{S}et(A, C) \times \mathbb{S}et(B, C)$$

From these two properties we can calculate:

$$
\begin{array}{ll}
& \mathbb{S}et((A + B) \times C, X) \\
\simeq & [\![\;\; \text{currying} \;\;]\!] \\
& \mathbb{S}et(A + B, X^C) \\
\simeq & [\![\;\; \text{sum} \;\;]\!] \\
& \mathbb{S}et(A, X^C) \times \mathbb{S}et(B, X^C) \\
\simeq & [\![\;\; \text{currying} \;\;]\!] \\
& \mathbb{S}et(A \times C, X) \times \mathbb{S}et(B \times C, X) \\
\simeq & [\![\;\; \text{sum} \;\;]\!] \\
& \mathbb{S}et((A \times C) + (B \times C), X)
\end{array}
$$

and so by the rule of indirect equality we can conclude that

$$(A + B) \times C \simeq (A \times C) + (B \times C)$$

(As an aside for the categorically wise: the only fact about product and exponential that we used here is that $- \times C$ is left adjoint to $-^C$; the same calculation works to show that $L\,(A + B) \simeq L\,A + L\,B$ for any left adjoint L. Indeed, it works in any category $\mathbb{C}$, for any colimit in place of $+$. As a result, we get a proof of the slogan 'left adjoints preserve colimits'. Dually, right adjoints preserve limits.)

3.3 Cayley's Theorem

Cayley's Theorem states that every group $(G, \oplus, e, (-)^{-1})$ is isomorphic to a group of endomorphisms, a subgroup of the symmetric group acting on G: in one direction, each element $g \in G$ is taken to the endomorphism $(g \oplus -)$; in the opposite direction, the endomorphism f is taken to the element $f(e)$. Cayley's Theorem is an instance of the Yoneda Embedding. But groups are not very familiar to functional programmers, so we will illustrate with the slight generalization to *Cayley's Theorem for Monoids*: every monoid $(M, \oplus, e)$ is isomorphic to a monoid of endomorphisms, sometimes called a *transformation monoid*, a submonoid of the 'full transformation monoid' of M consisting of all functions $M \to M$. The construction and proof is entirely analogous, except that it omits discussion of the inverse operation of a group.

Recall that a monoid $(M, \oplus, e)$ can be represented as a category $\mathbb{M}on(M, \oplus, e)$, with a single object $*$, and with sole homset $\mathbb{M}on(M, \oplus, e)(*, *)$ consisting of the elements of M, with $\oplus$ serving as composition and e as the identity arrow. The functor $H^{\bullet} : \mathbb{M}on(M, \oplus, e)^{\mathrm{op}} \to [\mathbb{M}on(M, \oplus, e), \mathbb{S}et]$ takes the sole object $*$ to the homfunctor $\mathbb{M}on(M, \oplus, e)(*, -)$, and each arrow $m : * \to *$ to the function $(m \oplus -) : M \to M$. Then the Yoneda Embedding in this category states that this mapping is a bijection; again, in one direction, each element $m \in M$ is taken to the endomorphism $(m \oplus -)$ (that is, the functorial action of the homfunctor), and in the opposite direction, an endomorphism f is taken to $f(e)$ (that is, application to the identity arrow).

This construction is common in functional programming. It's the essence of the accumulating parameter technique [Bird 1984], and of 'the concatenate vanishes' [Wadler 1987]. For example, Hughes' 'novel representation of lists' [Hughes 1986] amounts to Cayley's Theorem, substituting the monoid of endomorphisms $([A] \to [A], (\circ), id)$ for the monoid of lists $([A], +\!\!+, [\,])$, representing each list $xs :: [A]$ by the endomorphism $(xs +\!\!+) :: [A] \to [A]$; with this data representation, the formerly quadratic-time *reverse* function becomes linear-time.

The analogous construction on a preorder $(A, \leqslant)$ of Cayley's Theorem on monoids is essentially the rule of indirect inequality discussed in Section 3.1. If we define the upwards closure $\uparrow a$ of an element $a \in A$ as $\{c \in A \mid a \leqslant c\}$, then indirect inequality can be stated as follows:

$$\begin{aligned} (b \leqslant a) &\Leftrightarrow (\forall c \,.\, (c \in \uparrow a) \Rightarrow (c \in \uparrow b)) \\ &\Leftrightarrow \uparrow a \subseteq \uparrow b \end{aligned}$$

This gives the representation of the preordered set $(A, \leqslant)$ by its *upward-closed sets*. Thus, a preorder on any kind of carrier is equivalent to one on sets using inclusion, just as a monoid on any kind of carrier is equivalent to one on functions using composition.

As Awodey [2006, §8.4] observes, it is often the case that the Yoneda Embedding is into a space with better structural properties, thereby simplifying reasoning. For example, the representation of a rational number as an upward-closed set under the usual ordering is a *Dedekind Cut*, and this mapping embeds the rationals into the real line, which admits solutions to more equations; a similar phenomenon happens for other instances of the Yoneda Embedding. (Formally, the functor category $[\mathbb{C}, \mathbb{S}et]$ has various kinds of limit, even when $\mathbb{C}$ does not.)

3.4 Universal and Existential

The Yoneda Lemma can be almost directly rendered into Haskell, via the following type declaration:

$$\textbf{data}\ \mathit{Yo}\, f\ a = \mathit{Yo}\,\{\mathit{unYo} :: \forall r \,.\, (a \to r) \to f\ r\}$$

This declares a container datatype $\mathit{Yo}\, f$ for every f, with constructor *Yo* and deconstructor *unYo*; the body of a data structure of type $\mathit{Yo}\, f\ a$ is a polymorphic function of type $\forall r \,.\, (a \to r) \to f\ r$. The Yoneda Lemma states that $\mathit{Yo}\, f\ a \simeq f\ a$ when f is a functor; the two halves of the isomorphism are given by the following two functions:

$$fromYo :: Yo\,f\;a \to f\;a$$
$$fromYo\;y = unYo\;y\;id$$

$$toYo :: Functor\,f \Rightarrow f\;a \to Yo\,f\;a$$
$$toYo\;x = Yo\,(\lambda h \to fmap\,h\,x)$$

(Note again the essence: ‘application to the identity arrow’ in one direction, and ‘use of the functorial action’ in the other.) Informally, a value of type $Yo\,f\;a$ can take an arbitrary function of type $a \to r$, for any r, and yield a value of type $f\;r$; so it must in some sense have an $f\;a$ stored inside [Piponi 2006].

In the special case that f is the identity functor Id, we get $Yo\,Id\;a$, which is essentially a function of type $\forall r\,.\,(a \to r) \to r$, the continuation-passing representation of the type a [Hinze and James 2010; Stay 2008]. Or perhaps more precisely, the CPS transformation of a function of type $b \to a$ is a function of type $b \to Yo\,Id\;a$; and the correctness of the CPS transformation boils down to this special case of the Yoneda Lemma.

This rendering of the Yoneda Lemma into Haskell uses a universal type quantification. There is a complementary rendering, sometimes called ‘co-Yoneda’, using an existential quantification instead. To discover it, consider the following calculation with types:

$$
\begin{array}{ll}
 & \forall a\,.\,f\;a \to (\forall r\,.\,(a \to r) \to g\;r) \\
\simeq & [\![\ \text{universal property of universal quantification}\ ]\!] \\
 & \forall a\,.\,\forall r\,.\,f\;a \to (a \to r) \to g\;r \\
\simeq & [\![\ \text{uncurrying}\ ]\!] \\
 & \forall a\,.\,\forall r\,.\,(f\;a \times (a \to r)) \to g\;r \\
\simeq & [\![\ \text{swapping the two quantifiers}\ ]\!] \\
 & \forall r\,.\,\forall a\,.\,(f\;a \times (a \to r)) \to g\;r \\
\simeq & [\![\ \text{universal property of existential quantification}\ ]\!] \\
 & \forall r\,.\,(\exists a\,.\,(f\;a \times (a \to r))) \to g\;r
\end{array}
$$

Note that in the final step, it is the contravariant position of the function type that is being distributed over the quantifier, so the universal quantification turns into an existential. Let us define a datatype capturing the source type of the final function type above:

$$\mathbf{data}\;CoYo\,f\;r = \exists a\,.\,CoYo\,\{\,unCoYo :: f\;a \times (a \to r)\,\}$$

Informally, this is an abstract datatype, representing an $f\;r$ abstraction in terms of an $f\;a$ structure for some hidden type a, together with a function $a \to r$.

The first line of the type calculation is (modulo the constructor Yo) the type of natural transformations $f \mathrel{\dot{\to}} Yo\,g$, and now the last line is (again, modulo the constructor) the type $CoYo\,f \mathrel{\dot{\to}} g$; so provided that f and g are functors, we have:

$$
\begin{array}{ll}
 & f \mathrel{\dot{\to}} g \\
\simeq & [\![\ \text{Yoneda Lemma: } g \simeq Yo\,g \text{ for functor } g\ ]\!] \\
 & f \mathrel{\dot{\to}} Yo\,g \\
\simeq & [\![\ \text{calculation above}\ ]\!] \\
 & CoYo\,f \mathrel{\dot{\to}} g
\end{array}
$$

and so, by indirect equality, it follows that $f \simeq CoYo\,f$. The components of this isomorphism between the abstraction f and the representation $CoYo\,f$ are as follows:

$$fromCoYo :: Functor\,f \Rightarrow CoYo\,f\;b \to f\;b$$
$$fromCoYo\,(CoYo\,(x, h)) = fmap\,h\,x$$

```
toCoYo :: f b → CoYo f b
toCoYo y = CoYo (y, id)
```

Even if f is not a functor, *CoYo f* is—to 'map' a function $g :: b \to c$ over a *CoYo f b*, it suffices to compose g with the function inside:

```
instance Functor (CoYo f) where
  fmap g (CoYo (x, h)) = CoYo (x, g ∘ h)
```

This is sometimes called 'the co-Yoneda trick' [Manzyuk 2013]. It can be used to make a generalized algebraic datatype a functor when the type parameter is used only as an index, so cannot be varied freely; for example, Gibbons [2016] uses it to enable **do** notation on a GADT representing monomorphic typed commands. Even when the f parameter is already a functor, this change of representation can be an efficiency-improving transformation, because it turns a traversal over each of the elements of the payload x into a simple composition with one function h [Avramenko 2017]—that is, it implements the *map fusion* rule for free. New [2017] shows that the closure conversion of a function of type $a \to b$ yields one of type $a \to \exists e\,.\, e \times (e \to b)$, the result of which is a closure with hidden environment type e. That is, the closure conversion of function type $a \to b$ is $a \to$ *CoYo Id b*; New describes closure conversion itself as a kind of dual to CPS, both being applications of the Yoneda Lemma for the identity functor.

3.5 Representable Functors

A *representation* of a functor $F : \mathbb{C} \to \mathbb{S}et$ is an object $X \in |\mathbb{C}|$ together with a natural isomorphism $t : \mathbb{C}(X, -) \stackrel{\cdot}{\to} F$; and functor F is called *representable* if it has such a representation. It is then a corollary of the Yoneda Lemma that a representation of F is given equivalently by $X \in |\mathbb{C}|$ and a *universal element* $ps \in F(X)$ such that, for any $A \in |\mathbb{C}|$ and $x \in F(A)$, there exists a unique arrow $f \in \mathbb{C}(X, A)$ such that $F(f)(ps) = x$. That is, there is a one-to-one correspondence between pairs (X, t) and pairs (X, ps).

This result explains part of the design of Hancock's notion of *Naperian* functor [Gibbons 2017; Hancock 2005], expressed in Haskell as the type class

```
class Functor f ⇒ Naperian f where
  type Log f
  lookup    :: f a → (Log f → a)   -- each other's...
  tabulate  :: (Log f → a) → f a   -- ...inverses
  positions :: f (Log f)
  tabulate h = fmap h positions
  positions  = tabulate id
```

The idea here is that a functor f is Naperian if there is a type *Log f* of 'positions', such that $f\ a \simeq Log\ f \to a$. For example, the functor *Pair* is Naperian, with *Log Pair* = *Bool*, because there are two positions in a pair. The two halves of the isomorphism are given by *lookup* and *tabulate*. But there is an alternative presentation given instead by *lookup* and *positions*, since *tabulate* and *positions* are interdefinable; often *positions* is easier to define, but *tabulate* more convenient to use. For pairs, *positions* = (*True*, *False*) and *tabulate f* = (*f True*, *f False*). The *tabulate* version corresponds to the presentation in terms of representable functors, and the *positions* version to that in terms of universal elements. The Yoneda Lemma shows that the two presentations are equivalent. Again, the construction involves 'application to the identity arrow' in one direction, and 'use of the functorial action' in the other.

The reader should by now have some intuition as to the structure of instances of the Yoneda Lemma, and in particular of the recurring pattern of 'application to the identity arrow' and 'use of the functorial action'.

4 PROFUNCTOR OPTICS

We now exploit the power of the Yoneda Lemma in order to understand the isomorphisms between profunctor optics and their concrete analogues. The key result is the Double Yoneda Embedding, presented in Section 4.1. We apply the Double Yoneda Embedding to adapters in Section 4.2, to lenses in Section 4.3, and to prisms in Section 4.4. The proofs are all very similar; so in Section 4.5 we present an abstraction that covers all three specific cases, and use it in Sections 4.6 and 4.7 to cover more optics.

4.1 Double Yoneda

The key insight to understanding the profunctor representation of optics is the following general result, which we dub the *Double Yoneda Embedding*; the name is ours, but the result is due to Milewski [2017a,b].

LEMMA 4.1 (DOUBLE YONEDA EMBEDDING). *For arbitrary $A, B \in \mathbb{C}$, we have the following isomorphism:*

$$\mathbb{C}(A, B) \simeq [[\mathbb{C}, \mathbb{S}et], \mathbb{S}et](-(A), -(B))$$

The isomorphism is natural in A and B, and respects composition and identity.

PROOF. For an object $A \in \mathbb{C}$, the operation of applying a functor $F : \mathbb{C} \to \mathbb{S}et$ to A, denoted $-(A)$, is indeed a functor from $[\mathbb{C}, \mathbb{S}et]$ to $\mathbb{S}et$: it maps $F \in [\mathbb{C}, \mathbb{S}et]$ to $F(A) \in \mathbb{S}et$, and $\phi : F \stackrel{.}{\to} G$ to ϕ_A.

Now, the Yoneda Lemma in $\mathbb{C}$ states that:

$$F(X) \simeq [\mathbb{C}, \mathbb{S}et](\mathbb{C}(X, =), F)$$

(naturally in F and X). Therefore the functor $-(X)$ is isomorphic to $[\mathbb{C}, \mathbb{S}et](\mathbb{C}(X, =), -)$, naturally in X. Then using the Yoneda Embedding twice, we get the following:

$$\begin{array}{ll} & [[\mathbb{C}, \mathbb{S}et], \mathbb{S}et](-(A), -(B)) \\ \simeq & [\![\text{ as noted above: } -(X) \simeq [\mathbb{C}, \mathbb{S}et](\mathbb{C}(X, =), -) \ ]\!] \\ & [[\mathbb{C}, \mathbb{S}et], \mathbb{S}et]([\mathbb{C}, \mathbb{S}et](\mathbb{C}(A, =), -), [\mathbb{C}, \mathbb{S}et](\mathbb{C}(B, =), -)) \\ \simeq & [\![\text{ Yoneda Embedding in } [\mathbb{C}, \mathbb{S}et] \ ]\!] \\ & [\mathbb{C}, \mathbb{S}et](\mathbb{C}(B, =), \mathbb{C}(A, =)) \\ \simeq & [\![\text{ Yoneda Embedding in } \mathbb{C} \ ]\!] \\ & \mathbb{C}(A, B) \end{array}$$

All the isomorphisms used are natural in the relevant variables, hence the overall isomorphism is natural in A and B. From the proof of the isomorphism we extract one of its components:

$$\begin{array}{l} \eta : \mathbb{C}(A, B) \to [[\mathbb{C}, \mathbb{S}et], \mathbb{S}et](-(A), -(B)) \\ (\eta\, f)_F := F(f) \end{array}$$

Because the index F is a functor, η clearly respects composition and identity; and therefore so too does its inverse. Therefore the isomorphism respects composition and identity. □

For the particular case of Haskell endofunctors $\mathbb{S}et \to \mathbb{S}et$, this result can be expressed as follows:

$$(a \to b) \simeq \forall f \,.\, \mathit{Functor}\, f \Rightarrow f\ a \to f\ b$$

Informally, this says that if we know nothing about f other than that it is a functor, the only way to go from $f\ a$ to $f\ b$ is to *fmap* a function $a \to b$:

$$fromFun :: (a \to b) \to (\forall f\ .\ Functor\ f \Rightarrow f\ a \to f\ b)$$
$$fromFun\ f = fmap\ f$$

and conversely, if we can go from $f\ a$ to $f\ b$ for any functor f, we can in particular do so for the identity functor:

$$toFun :: (\forall f\ .\ Functor\ f \Rightarrow f\ a \to f\ b) \to (a \to b)$$
$$toFun\ h = unId \circ h \circ Id$$

and moreover that these two transformations are each other's inverses. Note again the recurring pattern: 'use of the functorial action' and 'application to the identity' (that is, to the identity functor).

4.2 Adapters

We now show that concrete adapters and profunctor adapters are equivalent. The construction in this section is also originally due to Milewski [2017a,b].

Recall the datatype of concrete adapters:

$$\textbf{data}\ Adapter\ a\ b\ s\ t = Adapter\ \{from :: s \to a, to :: b \to t\}$$

In categorical language, this corresponds to a pair of arrows $\mathbb{C}(S, A) \times \mathbb{C}(B, T)$, that is, an arrow $(\mathbb{C}^{op} \times \mathbb{C})((A, B), (S, T))$. Adapters with matching types can be composed, and there is an identity adapter *Adapter id id*, in both cases adopting the structure of $\mathbb{C}^{op} \times \mathbb{C}$. We therefore introduce the synonym $\mathbb{A}da$ for the category $\mathbb{C}^{op} \times \mathbb{C}$, and think of it as the category whose arrows are adapters.

Likewise, the profunctor representation of adapters was defined as follows:

$$\textbf{type}\ AdapterP\ a\ b\ s\ t = \forall p\ .\ Profunctor\ p \Rightarrow p\ a\ b \to p\ s\ t$$

Categorically, a profunctor P is a functor $P : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{S}et$. It takes an object (A, B) in $\mathbb{C}^{op} \times \mathbb{C}$ to the set $P(A, B)$, and an arrow $(\mathbb{C}^{op} \times \mathbb{C})((A, B), (C, D))$ (that is, a pair of arrows $\mathbb{C}(C, A) \times \mathbb{C}(B, D)$) to a function $P(A, B) \to P(C, D)$ in $\mathbb{S}et$. Thus, the action on arrows corresponds precisely to the *dimap* method of the *Profunctor* type class, and the functor laws on $\mathbb{C}^{op} \times \mathbb{C}$ precisely to the *dimap* laws in the type class. We therefore define the category $\mathbb{P}rof$ (over a base category $\mathbb{C}$) to be the functor category $[\mathbb{C}^{op} \times \mathbb{C}, \mathbb{S}et]$.

Now, in categorical language, the universal quantification in the definition of *AdapterP* corresponds to a natural transformation. We therefore define the category $\mathbb{A}daP$ whose arrows are profunctor adapters (over a base category $\mathbb{C}$): the objects (A, B) are those of $\mathbb{C}^{op} \times \mathbb{C}$, as before, but the arrows are natural transformations:

$$\begin{aligned} \mathbb{A}daP((A, B), (S, T)) &:= [\mathbb{P}rof, \mathbb{S}et](-(A, B), -(S, T)) \\ &= [[\mathbb{C}^{op} \times \mathbb{C}, \mathbb{S}et], \mathbb{S}et](-(A, B), -(S, T)) \end{aligned}$$

Then the Double Yoneda Embedding states:

$$(\mathbb{C}^{op} \times \mathbb{C})((A, B), (S, T)) \simeq [[\mathbb{C}^{op} \times \mathbb{C}, \mathbb{S}et], \mathbb{S}et](-(A, B), -(S, T))$$

that is,

$$\mathbb{A}da((A, B), (S, T)) \simeq \mathbb{A}daP((A, B), (S, T))$$

On the left are arrows in $\mathbb{A}da$, which are concrete adapters; on the right are arrows in $\mathbb{A}daP$, which are profunctor adapters; and the Double Yoneda Embedding tells us that these are in one-to-one correspondence (moreover, with nice naturality and functoriality properties).

4.3 Lenses

So much for adapters; what about lenses? Here, we take a different approach from Milewski [2017a,b]. Recall the definition of a concrete lens:

$$\mathbf{data}\; Lens\; a\; b\; s\; t = Lens\,\{\, view :: s \to a,\; update :: s \times b \to t \,\}$$

Translated into categorical language, this again corresponds to a pair of arrows, but now also involving a product. We therefore introduce a category $\mathbb{L}ens$ (over a base category $\mathbb{C}$), whose objects are those of $\mathbb{C}^{op} \times \mathbb{C}$ as before, but whose arrows are concrete lenses:

$$\mathbb{L}ens((A, B), (S, T)) := \mathbb{C}(S, A) \times \mathbb{C}(S \times B, T)$$

Using the co-Yoneda Lemma, we can derive a more symmetrical characterization of the arrows:

$$\begin{aligned}
& \mathbb{C}(S, A) \times \mathbb{C}(S \times B, T) \\
\simeq\; & [\![\text{ co-Yoneda (see below) }]\!] \\
& \mathbb{C}(S, A) \times \exists C \,.\, \mathbb{C}(S, C) \times \mathbb{C}(C \times B, T) \\
\simeq\; & [\![\text{ first component is independent of the quantification }]\!] \\
& \exists C \,.\, \mathbb{C}(S, A) \times \mathbb{C}(S, C) \times \mathbb{C}(C \times B, T) \\
\simeq\; & [\![\text{ universal property of products }]\!] \\
& \exists C \,.\, \mathbb{C}(S, C \times A) \times \mathbb{C}(C \times B, T)
\end{aligned}$$

Thus $\mathbb{L}ens((A, B), (S, T)) \simeq \exists C \,.\, \mathbb{C}(S, C \times A) \times \mathbb{C}(C \times B, T)$. In Haskell, we might write:

$$\mathbf{data}\; LensC\; a\; b\; s\; t = \exists c \,.\, LensC\; (s \to c \times a, c \times b \to t)$$

Informally, a source S is factorized into a view A and its complement C, such that a new view B can be combined with the complement C to make a new source T. (But note that this is not the *constant complement* encoding of lenses [Bancilhon and Spyratos 1981; Foster et al. 2005]: there is no requirement that this pair of functions be in any sense inverses.)

The first step in the calculation uses the co-Yoneda Lemma from Section 3.4:

$$F(S) \simeq (\exists C \,.\, \mathbb{C}(C, S) \times F(C))$$

but in the opposite category: for functor $F : \mathbb{C}^{op} \to \mathbb{S}et$,

$$F(S) \simeq (\exists C \,.\, \mathbb{C}^{op}(C, S) \times F(C)) \simeq (\exists C \,.\, \mathbb{C}(S, C) \times F(C))$$

In the calculation, F is $\mathbb{C}(- \times B, T)$, which is indeed a functor $\mathbb{C}^{op} \to \mathbb{S}et$.

In this symmetric form, it is easier to see that $\mathbb{L}ens$ does indeed form a category. The identity lens on object (A, B) is $(lunit_A, lunit'_B)$, where the complement chosen is the unit type 1, the neutral element of product; composition works by taking the product of complements:

$$\begin{aligned}
(S \to C \times A) \times (A \to D \times X) &\mapsto (S \to (C \times D) \times X) \\
(C \times B \to T) \times (D \times Y \to B) &\mapsto ((C \times D) \times Y \to T)
\end{aligned}$$

Cartesian product forms a monoid only up to isomorphism; but those isomorphisms get quotiented out by the existential quantification, so composition of lenses forms a monoid up to equivalence of values of existential types.

The approach we took for adapters in Section 4.2 involved profunctors, which are functors $\mathbb{A}da \to \mathbb{S}et$. By analogy, let us investigate functors $P : \mathbb{L}ens \to \mathbb{S}et$. Such a functor takes an object (A, B) of $\mathbb{L}ens$ to a set $P(A, B)$, and an arrow $\mathbb{L}ens((A, B), (S, T))$ (that is, a concrete lens) to a function $P(A, B) \to P(S, T)$ in $\mathbb{S}et$. Let us name that action on arrows:

$$mapLens \in \forall A\, B\, S\, T \,.\, \mathbb{S}et(\mathbb{L}ens((A, B), (S, T)), \mathbb{S}et(P(A, B), P(S, T)))$$

Then we calculate with the type of *mapLens*:

$$
\begin{aligned}
& \forall A\,B\,S\,T\,.\,\mathbb{S}et(\mathbb{L}ens((A,B),(S,T)),\mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\ \text{symmetric characterization of arrows in } \mathbb{L}ens\ ]\!] \\
& \forall A\,B\,S\,T\,.\,\mathbb{S}et((\exists C\,.\,\mathbb{C}(S,C\times A)\times\mathbb{C}(C\times B,T)),\mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\ \text{universal property of existential quantification}\ ]\!] \\
& \forall A\,B\,C\,S\,T\,.\,\mathbb{S}et(\mathbb{C}(S,C\times A)\times\mathbb{C}(C\times B,T),\mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\ \text{arrows in } \mathbb{C}^{op}\times\mathbb{C}\ ]\!] \\
& \forall A\,B\,C\,S\,T\,.\,\mathbb{S}et((\mathbb{C}^{op}\times\mathbb{C})((C\times A,C\times B),(S,T)),\mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\ \text{natural transformations in } \mathbb{S}et \text{ as polymorphic functions}\ ]\!] \\
& \forall A\,B\,C\,.\,[\mathbb{C}^{op}\times\mathbb{C},\mathbb{S}et]((\mathbb{C}^{op}\times\mathbb{C})((C\times A,C\times B),-),\mathbb{S}et(P(A,B),P(-))) \\
\simeq\ & [\![\ \text{Yoneda Lemma in } \mathbb{C}^{op}\times\mathbb{C}\ ]\!] \\
& \forall A\,B\,C\,.\,\mathbb{S}et(P(A,B),P(C\times A,C\times B))
\end{aligned}
$$

(To be precise, for these calculations to be valid, the functors mentioned must be functorial in arrows of $\mathbb{C}^{op}\times\mathbb{C}$ and not only in arrows of *Lens*. For this to hold, we need an identity-on-objects functor $Lift_{\mathbb{L}ens} : \mathbb{C}^{op}\times\mathbb{C} \to \mathbb{L}ens$, the details of which will be spelled out in Section 4.5.)

The final type in the calculation above is precisely the type of the *second* method of a Cartesian profunctor:

```
class Profunctor p ⇒ Cartesian p where
  second :: p a b → p (c × a) (c × b)
```

Moreover, the property that *mapLens* respects composition and identity of arrows in $\mathbb{L}ens$ maps precisely onto the property that *second* respects the monoidal structure of products and the unit type (this will be worked out in detail in Section 4.5). That is, Cartesian profunctors are precisely the functors $\mathbb{L}ens \to \mathbb{S}et$.

Profunctor lenses were defined in terms of Cartesian profunctors:

```
type LensP a b s t = ∀p . Cartesian p ⇒ p a b → p s t
```

As with adapters, the universal quantification corresponds categorically to a natural transformation, between the functors $-(A,B)$ and $-(S,T)$ from $[\mathbb{L}ens,\mathbb{S}et]$ to $\mathbb{S}et$. We therefore define the category $\mathbb{L}ensP$, with objects from $\mathbb{C}^{op}\times\mathbb{C}$, and profunctor lenses (natural transformations) as arrows:

$$\mathbb{L}ensP((A,B),(S,T)) := [[\mathbb{L}ens,\mathbb{S}et],\mathbb{S}et](-(A,B),-(S,T))$$

The Double Yoneda Embedding in category $\mathbb{L}ens$ states:

$$\mathbb{L}ens((A,B),(S,T)) \simeq [[\mathbb{L}ens,\mathbb{S}et],\mathbb{S}et](-(A,B),-(S,T))$$

which immediately shows that

$$\mathbb{L}ens((A,B),(S,T)) \simeq \mathbb{L}ensP((A,B),(S,T))$$

On the left are arrows in $\mathbb{L}ens$, which are concrete lenses; on the right are arrows in $\mathbb{L}ensP$, which are profunctor lenses; and the Double Yoneda Embedding tells us that these are in one-to-one correspondence (again, with nice naturality properties).

4.4 Prisms

Prisms are to sums as lenses are to products; the development is entirely dual. Concrete prisms were defined as follows:

```
data Prism a b s t = Prism { match :: s → t + a, build :: b → t }
```

Categorically, we represent concrete prisms (over a base category $\mathbb{C}$) as arrows in a category $\mathbb{P}rism$, whose objects are those of $\mathbb{C}^{op}\times\mathbb{C}$ and whose arrows are pairs of base arrows:

$$\mathbb{P}rism((A,B),(S,T)) = \mathbb{C}(S,T+A) \times \mathbb{C}(B,T)$$

Using co-Yoneda again, we can derive an equivalent but more symmetric characterization:

$$\begin{aligned}
& \mathbb{C}(S,T+A) \times \mathbb{C}(B,T) \\
\simeq\ & [\![\text{ co-Yoneda }]\!] \\
& \exists C\,.\,\mathbb{C}(S,C+A) \times \mathbb{C}(C,T) \times \mathbb{C}(B,T) \\
\simeq\ & [\![\text{ universal property of sum }]\!] \\
& \exists C\,.\,\mathbb{C}(S,C+A) \times \mathbb{C}(C+B,T)
\end{aligned}$$

This time, the co-Yoneda Lemma is used covariantly, with the functor being $\mathbb{C}(S,-+A)$. Thus, $\mathbb{P}rism((A,B),(S,T)) \simeq \exists C\,.\,\mathbb{C}(S,C+A) \times \mathbb{C}(C+B,T)$. Informally, a prism downcasts a source S into either a view A or its complement C, in such a way that either the complement or a new view B may be upcast to a new source T.

As with lenses, from the monoidal structure of sum we derive a composition operator and identity that make $\mathbb{P}rism$ into a category (as well as a functor $Lift_{\mathbb{P}rism} : \mathbb{C}^{op} \times \mathbb{C} \to \mathbb{P}rism$; we postpone the details to Section 4.5, where we consider the general case having seen a couple of specific instances).

A functor $P : \mathbb{P}rism \to \mathbb{S}et$ takes objects (A,B) to sets $P(A,B)$, and has action on arrows given by:

$$mapPrism \in \forall A\,B\,S\,T\,.\,\mathbb{S}et(\mathbb{P}rism((A,B),(S,T)), \mathbb{S}et(P(A,B),P(S,T)))$$

Using Yoneda, we can massage the type of *mapPrism*:

$$\begin{aligned}
& \forall A\,B\,S\,T\,.\,\mathbb{S}et(\mathbb{P}rism((A,B),(S,T)), \mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\text{ symmetric characterization of arrows in } \mathbb{P}rism \text{ }]\!] \\
& \forall A\,B\,C\,S\,T\,.\,\mathbb{S}et(\mathbb{C}(S,C+A) \times \mathbb{C}(C+B,T), \mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\text{ arrows in } \mathbb{C}^{op} \times \mathbb{C} \text{ }]\!] \\
& \forall A\,B\,C\,S\,T\,.\,\mathbb{S}et((\mathbb{C}^{op} \times \mathbb{C})((C+A,C+B),(S,T)), \mathbb{S}et(P(A,B),P(S,T))) \\
\simeq\ & [\![\text{ natural transformations in } \mathbb{S}et \text{ as polymorphic functions }]\!] \\
& \forall A\,B\,C\,.\,[\mathbb{C}^{op} \times \mathbb{C}, \mathbb{S}et]((\mathbb{C}^{op} \times \mathbb{C})((C+A,C+B),-), \mathbb{S}et(P(A,B),P(-))) \\
\simeq\ & [\![\text{ Yoneda Lemma in } \mathbb{C}^{op} \times \mathbb{C} \text{ }]\!] \\
& \forall A\,B\,C\,.\,\mathbb{S}et(P(A,B),P(C+A,C+B))
\end{aligned}$$

This is precisely the type of the *right* method of a co-Cartesian profunctor:

```
class Profunctor p ⇒ Cocartesian p where
  right :: p a b → p (c + a) (c + b)
```

Moreover, the property that *mapPrism* respects composition and identity of arrows in $\mathbb{P}rism$ maps precisely onto the property that *right* respects the monoidal structure of sums and the empty type (see details in Section 4.5). That is, co-Cartesian profunctors are precisely the functors $\mathbb{P}rism \to \mathbb{S}et$.

Profunctor prisms are defined in terms of co-Cartesian profunctors:

```
type PrismP a b s t = ∀p . Cocartesian p ⇒ p a b → p s t
```

We define the category $\mathbb{P}rismP$, with objects from $\mathbb{C}^{op} \times \mathbb{C}$, and profunctor prisms (natural transformations) as arrows:

$$\mathbb{P}rismP((A,B),(S,T)) := [[\mathbb{P}rism, \mathbb{S}et], \mathbb{S}et](-(A,B), -(S,T))$$

The Double Yoneda Embedding in category $\mathbb{P}rism$ states:

$$\mathbb{P}rism((A,B),(S,T)) \simeq [[\mathbb{P}rism, \mathbb{S}et], \mathbb{S}et](-(A,B), -(S,T))$$

which immediately shows that

$$\mathbb{P}rism((A, B), (S, T)) \simeq \mathbb{P}rismP((A, B), (S, T))$$

—that is, concrete prisms are equivalent to profunctor prisms.

4.5 Arbitrary Profunctor Optics

The proofs in Sections 4.3 and 4.4 both look very similar: they rely on a particular monoidal structure that induces both an associated structure on profunctors and an associated category whose arrows are characterized by existential quantification. We can define those generically as follows.

Given a category $\mathbb{D}$ of profunctors over $\mathbb{C}$ with some additional structure, we define the category $\mathbb{P}rofOptic(\mathbb{D})$ as follows:

$$\mathbb{P}rofOptic(\mathbb{D})(X, Y) := [\mathbb{D}, \mathbb{S}et](-(X), -(Y))$$

This corresponds to a profunctor optic of type:

$$\textbf{type}\ \mathbb{P}rofOptic_{\mathbb{D}}\ a\ b\ s\ t = \forall p\,.\ p \in \mathbb{D} \Rightarrow p\ a\ b \rightarrow p\ s\ t$$

Define a *shape* as a family Σ of functors in $[\mathbb{C}, \mathbb{C}]$ that (up to isomorphism) is closed under composition and contains the identity functor (i.e. is a 'monoid up to isomorphism' under composition). Given a shape Σ, we can define a corresponding concrete existential optic:

$$\mathbb{E}xOptic_{\Sigma}((A, B), (S, T)) = \exists F \in \Sigma\,.\ \mathbb{C}(S, F(A)) \times \mathbb{C}(F(B), T)$$

a subclass of profunctors that respect the shape:

$$\begin{aligned}&\textbf{class}\ Profunctor\ p \Rightarrow Shaped_{\Sigma}\ p\ \textbf{where}\\ &\quad shape_{\Sigma} :: f \in \Sigma \Rightarrow p\ a\ b \rightarrow p\ (f\ a)\ (f\ b)\end{aligned}$$

and a corresponding profunctor existential optic:

$$\textbf{type}\ \mathbb{E}xOpticP_{\Sigma} = \mathbb{P}rofOptic_{Shaped_{\Sigma}}$$

(The notation '$f \in \Sigma$' is intended to suggest that the family Σ of functors be thought of as a type constructor class, and f an instance of this class.)

The laws imposed on $shape_{\Sigma}$ are:

$$\begin{aligned}&shape_{\Sigma\,Id} \simeq id\\ &shape_{\Sigma\,f} \circ shape_{\Sigma\,g} \simeq shape_{\Sigma\,(f \circ g)}\end{aligned}$$

(Those equalities are to be understood modulo appropriate wrapping/unwrapping of constructors).

Then we have the following theorem.

Theorem 4.2 (Representation theorem for profunctor optics). *Profunctor existential optics are isomorphic to concrete existential optics:*

$$\mathbb{E}xOpticP_{\Sigma} \simeq \mathbb{E}xOptic_{\Sigma}$$

Proof. As with lenses and prisms, we introduce a category $\mathbb{E}xOptic_{\Sigma}$ whose objects are those of $\mathbb{C}^{op} \times \mathbb{C}$ and whose arrows are concrete existential optics:

$$\mathbb{E}xOptic_{\Sigma}((A, B), (S, T)) = \exists F \in \Sigma\,.\ \mathbb{C}(S, F(A)) \times \mathbb{C}(F(B), T)$$

The identity and composition of arrows are inherited from the identity and composition on functors in Σ in a similar way as they were for lenses:

$$\begin{aligned}&id_{(A,B)} := \mathbb{E}xOptic_{Id}\ id_A\ id_B\\ &\mathbb{E}xOptic_F\ to\ fro \circ \mathbb{E}xOptic_G\ to'\ fro' := \mathbb{E}xOptic_{F \circ G}\ (F(to') \circ to)\ (fro \circ F(fro'))\end{aligned}$$

Using the identity functor we also build an identity-on-objects functor $Lift_{\mathbb{E}xOptic_{\Sigma}} : \mathbb{C}^{op} \times \mathbb{C} \rightarrow \mathbb{E}xOptic_{\Sigma}$ that takes an arrow from $\mathbb{C}(S, A) \times \mathbb{C}(B, T)$ to $\mathbb{C}(S, Id\ A) \times \mathbb{C}(Id\ B, T)$.

A functor $P : \mathbb{E}xOptic_{\Sigma} \to Set$ takes objects (A, B) to $P(A, B)$, and has action on arrows given by:

$$mapExOptic_{\Sigma} \in \forall A\, B\, S\, T\, .\, \mathbb{S}et(\mathbb{E}xOptic_{\Sigma}((A, B), (S, T)), \mathbb{S}et(P(A, B), P(S, T)))$$

By precomposing with $Lift_{\mathbb{E}xOptic_{\Sigma}}$, P moreover acquires the structure of a profunctor $\mathbb{C}^{op} \times \mathbb{C} \to Set$ with the same mapping on objects. This allows us to use the Yoneda Lemma to calculate with the type of $mapExOptic_{\Sigma}$:

$$\begin{array}{ll}
& \forall A\, B\, S\, T\, .\, \mathbb{S}et(\mathbb{E}xOptic_{\Sigma}((A, B), (S, T)), \mathbb{S}et(P(A, B), P(S, T))) \\
\simeq & [\![\text{ arrows in } \mathbb{E}xOptic_{\Sigma}]\!] \\
& \forall A\, B\, S\, T\, .\, \mathbb{S}et(\exists F \in \Sigma\, .\, \mathbb{C}(S, F(A)) \times \mathbb{C}(F(B), T), \mathbb{S}et(P(A, B), P(S, T))) \\
\simeq & [\![\text{ universal property of existential quantification }]\!] \\
& \forall A\, B\, S\, T\, .\, \forall F \in \Sigma\, .\, \mathbb{S}et(\mathbb{C}(S, F(A)) \times \mathbb{C}(F(B), T), \mathbb{S}et(P(A, B), P(S, T))) \\
\simeq & [\![\text{ arrows in } \mathbb{C}^{op} \times \mathbb{C}]\!] \\
& \forall A\, B\, S\, T\, .\, \forall F \in \Sigma\, .\, \mathbb{S}et((\mathbb{C}^{op} \times \mathbb{C})((F(A), F(B)), (S, T)), \mathbb{S}et(P(A, B), P(S, T))) \\
\simeq & [\![\text{ natural transformations in } \mathbb{S}et \text{ as polymorphic functions }]\!] \\
& \forall A\, B\, .\, \forall F \in \Sigma\, .\, [\mathbb{C}^{op} \times \mathbb{C}, \mathbb{S}et]((\mathbb{C}^{op} \times \mathbb{C})((F(A), F(B)), -), \mathbb{S}et(P(A, B), P(-))) \\
\simeq & [\![\text{ Yoneda Lemma in } \mathbb{C}^{op} \times \mathbb{C}]\!] \\
& \forall A\, B\, .\, \forall F \in \Sigma\, .\, \mathbb{S}et(P(A, B), P(F(A), F(B)))
\end{array}$$

—the latter type being the type of $shape_{\Sigma}$.

To work out the correspondence between the functor laws of $mapExOptic_{\Sigma}$ and the laws of $shape_{\Sigma}$, we extract from the proof above the relationship between those two maps. The main step of the proof is the application of the Yoneda Lemma, which gives us an idea of the components of the isomorphism: the Yoneda isomorphism in one direction involves applying to the identity, and the other direction involves mapping an arrow through a functorial action. Formally, $shape_{\Sigma}$ and $mapExOptic_{\Sigma}$ are thus related as follows:

$$\begin{array}{l}
shape_{\Sigma\, F} = mapExOptic_{\Sigma}\, (\mathbb{E}xOptic_{F}\; id_{F(A)}\; id_{F(B)}) \\
mapExOptic_{\Sigma}\, (\mathbb{E}xOptic_{F}\; to\; fro) = dimap\; to\; fro \circ shape_{\Sigma\, F}
\end{array}$$

The interaction between $mapExOptic_{\Sigma}$ and composition of arrows is linked with the interaction between $shape_{\Sigma}$ and composition of functors:

$$\begin{array}{ll}
& mapExOptic_{\Sigma}\, (\mathbb{E}xOptic_{F}\; to\; fro) \circ mapExOptic_{\Sigma}\, (\mathbb{E}xOptic_{G}\; to'\; fro') \\
= & [\![\; mapExOptic_{\Sigma} \text{ and } shape_{\Sigma} \;]\!] \\
& dimap\; to\; fro \circ shape_{\Sigma\, F} \circ dimap\; to'\; fro' \circ shape_{\Sigma\, G} \\
= & [\![\text{ by naturality, } shape_{\Sigma\, F} \circ dimap\; \alpha\; \beta = dimap\; F(\alpha)\; F(\beta) \circ shape_{\Sigma\, F} \;]\!] \\
& dimap\; to\; fro \circ dimap\; (F(to'))\; (F(fro')) \circ shape_{\Sigma\, F} \circ shape_{\Sigma\, G} \\
= & [\![\; dimap \text{ laws }]\!] \\
& dimap\; (F(to') \circ to)\; (fro \circ F(fro')) \circ shape_{\Sigma\, F} \circ shape_{\Sigma\, G}
\end{array}$$

and

$$\begin{array}{ll}
& mapExOptic_{\Sigma}\, (\mathbb{E}xOptic_{F}\; to\; fro \circ \mathbb{E}xOptic_{G}\; to'\; fro') \\
= & [\![\text{ composition in } \mathbb{E}xOptic_{\Sigma} \;]\!] \\
& mapExOptic_{\Sigma}\, (\mathbb{E}xOptic_{F \circ G}\; (F(to') \circ to)\; (fro \circ F(fro'))) \\
= & [\![\; mapExOptic_{\Sigma} \text{ and } shape_{\Sigma} \;]\!] \\
& dimap\; (F(to') \circ to)\; (fro \circ F(fro')) \circ shape_{\Sigma\, (F \circ G)}
\end{array}$$

From this we see that $shape_{\Sigma}$ respects composition of functors ($shape_{\Sigma\, F} \circ shape_{\Sigma\, G} \simeq shape_{\Sigma\, (F \circ G)}$) if and only if $mapExOptic_{\Sigma}$ respects composition of arrows ($mapExOptic_{\Sigma}\; l \circ mapExOptic_{\Sigma}\; l' =$

$\mathit{mapExOptic}_\Sigma\ (l \circ l')$). A similar correspondence applies with relation to the identity of those compositions:

$$\mathit{shape}_{\Sigma\ Id\ A\ B} = \mathit{mapExOptic}_\Sigma\ (\mathbb{E}\mathit{xOptic}_{Id}\ id_A\ id_B) = \mathit{mapExOptic}_\Sigma\ id_{(A,B)}$$

Therefore shape_Σ respects its laws if and only if $\mathit{mapExOptic}_\Sigma$ respects the functor laws. Thus functors $P : \mathbb{E}\mathit{xOptic}_\Sigma \to \mathbb{S}\mathit{et}$ are in one-to-one correspondence with profunctors $P \in \mathit{Shaped}_\Sigma$. Using the Double Yoneda Embedding, we deduce:

$$\mathbb{E}\mathit{xOptic}_\Sigma((A,B),(S,T)) \simeq \mathbb{E}\mathit{xOpticP}_\Sigma((A,B),(S,T))$$

as required.

□

The general form of $\mathbb{E}\mathit{xOpticP}_\Sigma$ captures most known profunctor optics. For lenses, the appropriate shape is $\Sigma_\times := \{(C \times -) \mid C \in \mathbb{C}\}$. By Cayley's Theorem, this forms a monoid isomorphic to the monoid $(\mathbb{C}, (\times), 1)$. Then, for that particular shape, the type of $\mathit{shape}_{\Sigma_\times}$ is the same as the type of *second* from the Cartesian typeclass, and the appropriate laws correspond as well. Thus the profunctors that respect shape $\Sigma_\times$ are precisely Cartesian profunctors, and profunctor lenses are isomorphic to $\mathbb{E}\mathit{xOpticP}_{\Sigma_\times}$. By the representation theorem, we recover the isomorphism between profunctor lenses and $\mathbb{E}\mathit{xOptic}_{\Sigma_\times} \simeq \mathbb{L}\mathit{ens}$ that we derived in Section 4.3. Similarly, prisms are an instance of this scheme for the shape $\Sigma_+ := \{(C + -) \mid C \in \mathbb{C}\}$, and we recover the isomorphism from Section 4.4. Finally, adapters correspond to the trivial shape $\Sigma_1 := \{\mathit{Id}\}$.

4.6 Traversals and More

Equipped with this representation theorem, we can now easily derive the isomorphism between profunctor optics and their concrete analogue for other known optics. For example, a common optic used for container-like values is called *traversal*. The *Traversable* class in the Haskell libraries is defined as follows:

```
class Functor t ⇒ Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
```

Informally, a finite container type T is traversable when one can iterate over the elements of a structure $T\ A$, in a fixed order, using an effectful operation of type $A \to F\ B$ for some applicative functor F, effectfully yielding a new $T\ B$ structure [Gibbons and dos Santos Oliveira 2009; McBride and Paterson 2008]. The family of traversable functors is a valid shape, since the identity functor is traversable and traversable functors compose. We can therefore define concrete traversals as follows [O'Connor 2014]:

$$\mathbb{T}\mathit{raversal} = \mathbb{E}\mathit{xOptic}_{\mathit{Traversable}}$$

Moreover, we immediately get the corresponding subclass $\mathit{Traversing} = \mathit{Shaped}_{\mathit{Traversable}}$ of profunctors that respect traversable, the associated profunctor optic $\mathbb{E}\mathit{xOpticP}_{\mathit{Traversing}}$, and a proof that the concrete and profunctor representations are isomorphic. This associated profunctor optic appears in profunctor optics libraries such as mezzolens [O'Connor 2015b].

We can also apply the representation theorem to some lesser-known optics, such as grates [O'Connor 2015a]. Concrete grates are defined by the rather un-enlightening type:

```
data Grate a b s t = ((s → a) → b) → t
```

O'Connor presents this type alongside an equivalent profunctor encoding, in terms of closed profunctors:

```
data GrateP a b s t = ∀p . Closed p ⇒ p a b → p s t
```

Closed profunctors are profunctors that can lift through functors of the form $(C \to -)$ for some type C—that is, through Naperian functors [Hancock 2005], or *representable functors* as discussed in Section 3.5. A Naperian functor F is a container of fixed shape: a value of type $F\,A$ contains exactly one element for each position of type C. Just as *Traversing* profunctors can lift through *Traversable* containers, *Closed* profunctors can lift through fixed-shape containers:

```
class Profunctor p ⇒ Closed p where
  closed :: p a b → p (c → a) (c → b)
```

Typical examples of Naperian functors are the identity functor, which is isomorphic to $(1 \to -)$, and pairs, isomorphic to $(Bool \to -)$. Naperian functors are also closed under composition, as can be shown using the currying isomorphism $C \to (D \to -) \simeq (C \times D) \to -$; therefore Naperian functors form a valid shape, and we have $GrateP = \mathbb{P}rofOptic_{Closed} \simeq \mathbb{E}xOpticP_{Naperian}$. Using the representation theorem, we can derive a third isomorphic form for grates: $\mathbb{E}xOptic_{Naperian}$. Spelling out its definition, we get:

$$\mathbb{E}xOptic_{Naperian}((A,B),(S,T)) = \exists C\,.\,\mathbb{C}(S, C \to A) \times \mathbb{C}(C \to B, T)$$

Using the co-Yoneda lemma, we recover the concrete representation of grates:

$$\begin{aligned}
& \exists C\,.\,\mathbb{C}(S, C \to A) \times \mathbb{C}(C \to B, T) \\
\simeq\ & [\![\text{ currying }]\!] \\
& \exists C\,.\,\mathbb{C}(S \times C, A) \times \mathbb{C}(C \to B, T) \\
\simeq\ & [\![\text{ currying }]\!] \\
& \exists C\,.\,\mathbb{C}(C, S \to A) \times \mathbb{C}(C \to B, T) \\
\simeq\ & [\![\text{ co-Yoneda lemma }]\!] \\
& \mathbb{C}((S \to A) \to B, T) \\
\simeq\ & [\![\text{ arrows in } \mathbb{G}rate]\!] \\
& \mathbb{G}rate((A,B),(S,T))
\end{aligned}$$

This existential form comes in useful for deriving properties of grates from properties of Naperian functors. Being of a fixed shape, different containers for the same Naperian functor can be zipped together. For example, three copies of such a container can be assembled into one copy containing triples:

```
zipWith3Naperian :: (a × a × a → b) → (c → a) × (c → a) × (c → a) → (c → b)
zipWith3Naperian op (f, g, h) c = op (f c, g c, h c)
```

As O'Connor [2015a] explains, a grate can be used to perform a similar zipping. Using the existential encoding, it becomes apparent how this property on Naperian functors is lifted to work with grates:

```
zipWith3Grate :: (∃c . (s → (c → a)) × ((c → b) → t)) → (a × a × a → b) → (s × s × s → t)
zipWith3Grate (to, fro) op = fro ∘ zipWith3Naperian op ∘ map3 to
  where map3 :: (a → b) → a × a × a → b × b × b
        map3 f (x, y, z) = (f x, f y, f z)
```

4.7 Composing Profunctor Optics

The main advantage of profunctor optics over concrete ones is their compositionality [Pickering et al. 2017]. They are just polymorphic functions, so may be combined using ordinary function composition; and the constraints for different kinds of optic simply conjoin, so even mixed compositions work fine — thus forming a semilattice of those optics. In contrast, concrete optics do not

compose nearly so well; in particular, mixed combinations, such as a lens with a prism, are not even expressible.

To put it another way, a profunctor adapter is automatically also a profunctor lens and a profunctor prism: if it is applicable for all profunctors P, then it is certainly applicable for all Cartesian profunctors, and for all co-Cartesian ones. Less obviously, a lens is also a traversal, because the shapes $(C\times)$ for lenses are instances of *Traversable*; and similarly for prisms, because $(C+)$ is also *Traversable*.

This in particular means that if we compose a lens and a traversal, by subsumption the resulting composition will be a traversal. What happens if we compose a lens and a prism? We know that the resulting optic will be a traversal, but it actually has a more precise type: it is the profunctor optic $\mathbb{P}rofOptic(Cartesian \wedge Cocartesian)$ for the intersection of the two typeclasses *Cartesian* and *Cocartesian*. Such optics are called affine traversals [Grenrus 2017]. We might capture this intersection via the following typeclass:

$$\textbf{class}\ (Cartesian\ p, Cocartesian\ p) \Rightarrow AffineTraversing\ p$$

This typeclass has just two members, inherited from the parent classes:

$$\begin{aligned} &second :: p\ a\ b \rightarrow p\ (c \times a)\ (c \times b) \\ &right\quad :: p\ a\ b \rightarrow p\ (c + a)\ (c + b) \end{aligned}$$

This is equivalent to having a sole member:

$$rightsecond :: p\ a\ b \rightarrow p\ (c + d \times a)\ (c + d \times b)$$

that respects the relevant monoidal structure (in one direction, compose the two methods; in the other direction, instantiate one of the extra types to be the appropriate neutral element).

Following the route taken earlier, we define a category

$$\mathbb{A}ffineTraversal((A, B), (S, T)) = \exists C, D\,.\,\mathbb{C}(S, C + D \times A) \times \mathbb{C}(C + D \times B, T)$$

and deduce from the representation theorem that:

$$\mathbb{P}rofOptic(Cartesian \wedge Cocartesian) \simeq \mathbb{A}ffineTraversal$$

When $\mathbb{C}$ is closed, co-Yoneda implies:

$$\mathbb{A}ffineTraversal((A, B), (S, T)) \simeq \mathbb{C}(S, T + A \times (B \rightarrow T))$$

Such optics manipulate, as expected, structures made from both sums and products.

5 CONCLUSION

5.1 Summary

We have presented a new and much simpler proof than any previously published of the correspondence between concrete optics (lenses, prisms, traversals, and so on) and their profunctor variants. The proof makes essential use of the Yoneda Lemma and its corollary the Yoneda Embedding; we hope to have inspired the reader to absorb this beautiful and powerful result, and hope also to have provided some intuition and guidance that will facilitate that outcome.

5.2 Related Work

The correspondence between concrete and profunctor optics is not new. It was well known to people such as Kmett and O'Connor, who have written optic libraries [Kmett 2018; O'Connor 2015b], and publicly discussed their properties online in blogs and chat channels. A formal presentation of the correspondence was first published by Pickering et al. [2017]; however, their proof made no use of the Yoneda Lemma, and instead went back to first principles in terms of free theorems—and as

a result, their proofs are more complicated than ours. An online talk and blog post by Milewski [2017a,b] present a proof that does use Yoneda, and in particular introduced the Double Yoneda Embedding (Lemma 4.1), of which we make crucial use. Our presentation follows his as far as profunctor adapters—the construction in Section 4.2 is also originally his—but then we diverge; specifically, Milewski makes use of a somewhat complicated adjunction in order to deal with lenses and prisms, introducing 'free Tambara modules for a tensor', whereas we think our approach is much simpler.

Matsuda and Wang [2015] present an approach to bidirectional programming in terms of the Yoneda Embedding of concrete (two-parameter) lenses into the higher-order space of functions between lenses. However, they do not make use of the isomorphism in the Yoneda Lemma for equational reasoning with lenses, or for proving equivalence with concrete lenses, reverting again to free theorems. Moreover, they only consider lenses, not prisms and other optics. Jaskelioff and O'Connor [2015] do make thorough use of the equational consequences of Yoneda for reasoning about a class of second-order functions, including some optics under the alternative van Laarhoven representation [van Laarhoven 2009], but they do not address the more convenient profunctor representation.

5.3 Discussion

One should note that there are (at least) three different kinds of dual to the Yoneda Lemma, as it was stated formally in Section 1. Duality in one sense arises from currying the homfunctor, and fixing one of the two arguments; the main version we have stated fixes the contravariant argument, leaving a covariant homfunctor. There is also a contravariant version of the Yoneda Lemma, arising from fixing instead the covariant argument; but as discussed in Section 2, this is not a separate result, because it is equivalent to the covariant case in the opposite category. Duality in a second sense arises as in Section 3.4: a straightforward rendering into Haskell of the natural transformations in the statement of the Yoneda Lemma uses a universal quantification, but there is a related presentation using existential quantification. The existential presentation is sometimes called 'co-Yoneda' [Manzyuk 2013; New 2017]; but it is essentially a consequence of the same Yoneda Lemma, not a separate result. Duality also arises in a third sense: both the covariant and the contravariant readings of the Yoneda Lemma involve natural transformations *from* a homfunctor *to* another functor, but one might ask instead about natural transformations *to* a homfunctor. This has been discussed in passing by Mac Lane [1971, Exercise III.2.3], attributed by him to Kan and also called 'co-Yoneda'; this really is a separate result, but it seems much less familiar, and less obviously useful.

The reader familiar with the literature on lenses will have noticed that we did not discuss *well-behavedness* [Foster et al. 2005], sometimes called *Put–Get* and *Get–Put* or *correctness* and *hippocraticness* [Stevens 2010]: informally, if one puts a view and then immediately retrieves it, one obtains the view that was put (the view was correctly stored), and conversely, if one gets a view from a source and immediately puts it back, the source does not change (no unnecessary harm was done in restoring consistency). In fact, well-behavedness is an orthogonal issue to the profunctor representation: profunctor optics may be well-behaved or ill-behaved, just as concrete optics may [Pickering et al. 2017]. In particular, note that the existential encoding $\exists C \,.\, \mathbb{C}(S, C \times A) \times \mathbb{C}(C \times B, T)$ of a concrete lens in Section 4.3 does not imply 'constant complement' [Bancilhon and Spyratos 1981; Foster et al. 2005], which would imply well-behavedness: although this encoding entails two functions that relate a source S to a view A and a complement C, it does not follow that these two functions are each other's inverses, so we do not get hippocraticness. We leave for future work the question of precisely how the correctness properties of lenses and other optics manifest themselves in the profunctor representation.

More categorically-inclined readers will have recognized the construction we used to make a category out of a homset with appropriate compositionality properties. Such a homset corresponds precisely to a monad in the bicategory $\mathbb{P}rof$ of profunctors. A standard property of those monads is that they can be identified with a category equipped with an identity-on-objects functor [nLab contributors 2018]. In our case, the constructed categories are $\mathbb{L}ens$, $\mathbb{P}rism$, etc., and the associated functors are $Lift_{\mathbb{L}ens}$, $Lift_{\mathbb{P}rism}$, etc. Note that we manipulate two different kinds of profunctors: profunctors over $\mathbb{C}$, which are the ones mentioned explicitly, for which we consider additional properties such as being Cartesian; and profunctors over $\mathbb{C}^{\mathrm{op}} \times \mathbb{C}$, which are the homsets and are monads in $\mathbb{P}rof$. As such, we do not encounter profunctors that would be both monads and strong, as described by Asada [2010]. Further exploration of the links between optics and monads in $\mathbb{P}rof$ is left as a topic for future research.

ACKNOWLEDGMENTS

Profunctor optics were introduced by Edward Kmett, Elliott Hird, Shachaf Ben-Kiki and others in the Haskell `lens` and `profunctor` libraries [Kmett 2015, 2018], and in some blog posts by Russell O'Connor describing his `mezzolens` library [O'Connor 2015b]; that history is documented in our earlier paper [Pickering et al. 2017]. The inspiration for this particular paper came from a talk by Bartosz Milewski [Milewski 2017a,b], and the results in Sections 4.1 and 4.2 are due to him. We are grateful to members of the *Algebra of Programming* group at Oxford, especially Ohad Kammar and Sam Staton, and to Ichiro Hasuo and Thorsten Wißmann at NII, for helpful comments and discussion; and to Tara Stubbs, for help in tracking down the Valéry quotation. The first author is funded by the UK EPSRC, and the paper was mostly written while the second author was a Visiting Professor at the National Institute of Informatics in Tokyo; we thank both EPSRC and NII for their support.

REFERENCES

Theodor Adorno. 1956. Fragment über Musik und Sprache. Reprinted in *Gesammelte Schriften, Band 16: Musikalische Schriften I–III*, Suhrkamp Verlag, 1978, p251–256.

Kazuyuki Asada. 2010. Arrows are Strong Monads. In *Mathematically Structured Functional Programming*. ACM. https://doi.org/10.1145/1863597.1863607

Alexey Avramenko. 2017. Yoneda and Coyoneda Trick. (April 2017). https://medium.com/@olxc/yoneda-and-coyoneda-trick-f5a0321aeba4.

Steve Awodey. 2006. *Category Theory*. Oxford University Press.

Roland Backhouse. 2003. *Program Construction: Calculating Implementations from Specifcations*. Wiley.

F. Bancilhon and N. Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 557–575. https://doi.org/10.1145/319628.319634

Richard S. Bird. 1984. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems* 6, 4 (Oct. 1984), 487–504. https://doi.org/10.1145/1780.1781 See also [Bird 1985].

Richard S. Bird. 1985. Addendum to "The Promotion and Accumulation Strategies in Transformational Programming". *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 490–492.

Edsger W. Dijkstra. 1991. Why Preorders Are Beautiful. (June 1991). EWD1102; available from http://www.cs.utexas.edu/~EWD/ewd11xx/EWD1102.PDF.

Wim Feijen. 2001. The Joy of Formula Manipulation. *Inform. Process. Lett.* 77 (2001), 89–96. https://doi.org/10.1016/S0020-0190(00)00195-2

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2005. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View Update Problem. In *Principles of Programming Languages*. ACM Press, 233–246. https://doi.org/10.1145/1040305.1040325

Jeremy Gibbons. 2016. Free Delivery (Functional Pearl). In *Haskell Symposium*. ACM, 45–50. https://doi.org/10.1145/2976002.2976005

Jeremy Gibbons. 2017. APLicative Programming with Naperian Functors. In *European Symposium on Programming (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer Berlin Heidelberg, 556–583. https://doi.org/10.1007/978-3-662-54434-1_21

Jeremy Gibbons and Bruno César dos Santos Oliveira. 2009. The Essence of the Iterator Pattern. *Journal of Functional Programming* 19, 3,4 (2009), 377–402. https://doi.org/10.1017/S0956796809007291

Oleg Grenrus. 2017. Affine Traversal. http://oleg.fi/gists/posts/2017-03-20-affine-traversal.html

Peter Hancock. 2005. What is a Naperian Container? (June 2005). http://sneezy.cs.nott.ac.uk/containers/blog/?p=14.

Ralf Hinze and Daniel W. H. James. 2010. Reason Isomorphically!. In *Workshop on Generic Programming*, Bruno C. d. S. Oliveira and Marcin Zalewski (Eds.). ACM, 85–96. https://doi.org/10.1145/1863495.1863507

R. John Muir Hughes. 1986. A Novel Representation of Lists and Its Application to the Function 'Reverse'. *Inform. Process. Lett.* 22 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

Mauro Jaskelioff and Russell O'Connor. 2015. A Representation Theorem for Second-Order Functionals. *Journal of Functtional Programming* 25 (2015). https://doi.org/10.1017/S0956796815000088

Edward Kmett. 2015. `profunctor` library, Version 5. https://hackage.haskell.org/package/profunctors-5

Edward Kmett. 2018. `lens` library, Version 4.16. https://hackage.haskell.org/package/lens-4.16

Tom Leinster. 2000. The Yoneda Lemma: What's It All About? (Oct. 2000). http://www.maths.ed.ac.uk/~tl/categories/yoneda.ps.

Saunders Mac Lane. 1971. *Categories for the Working Mathematician.* Springer-Verlag.

Oleksandr Manzyuk. 2013. Co-Yoneda Lemma. (Jan. 2013). https://oleksandrmanzyuk.wordpress.com/2013/01/18/co-yoneda-lemma/.

Kazutaka Matsuda and Meng Wang. 2015. Applicative Bidirectional Programming with Lenses. In *International Conference on Functional Programming*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 62–74. https://doi.org/10.1145/2784731.2784750

Shoukei Matsumoto. 2018. I Clean, Therefore I Am. *The Guardian* (5th January 2018). https://www.theguardian.com/commentisfree/2018/jan/05/buddhist-monk-cleaning-good-for-you.

Guerino Mazzola. 2002. *The Topos of Music.* Birkhäuser.

Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Bartosz Milewski. 2017a. Profunctor Optics: The Categorical Approach. https://www.youtube.com/watch?v=l1FCXUi6Vlw. Keynote at *Lambda World* conference, Cadiz.

Bartosz Milewski. 2017b. Profunctor Optics: The Categorical View. (July 2017). https://bartoszmilewski.com/2017/07/07/profunctor-optics-the-categorical-view/.

Max New. 2017. Closure Conversion as CoYoneda. (Aug. 2017). http://prl.ccs.neu.edu/blog/2017/08/28/closure-conversion-as-coyoneda/.

nLab contributors. 2018. monad in nLab. https://ncatlab.org/nlab/revision/monad/72#other_examples [Online; accessed 16-March-2018].

Russell O'Connor. 2014. Mainline Profunctor Hierarchy for Optics. http://r6research.livejournal.com/27476.html

Russell O'Connor. 2015a. Grate: A new kind of Optic. https://r6research.livejournal.com/28050.html

Russell O'Connor. 2015b. `mezzolens` library, Version 0. https://hackage.haskell.org/package/mezzolens-0

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1, 2 (2017), Article 7. https://doi.org/10.22152/programming-journal.org/2017/1/7

Dan Piponi. 2006. Reverse Engineering Machines with the Yoneda Lemma. (Nov. 2006). http://blog.sigfpe.com/2006/11/yoneda-lemma.html.

Emily Riehl. 2016. *Category Theory in Context.* Dover Publications. Available from http://www.math.jhu.edu/~eriehl/context.pdf.

Mike Stay. 2008. The Continuation Passing Transform and the Yoneda Embedding. (Jan. 2008). https://golem.ph.utexas.edu/category/2008/01/the_continuation_passing_trans.html.

Perdita Stevens. 2010. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and System Modeling* 9, 1 (2010), 7–20. https://doi.org/10.1007/s10270-008-0109-9

Paul Valéry. 1937. Leçon Inaugurale du Cours de Poétique du Collège de France. Reprinted in *Variété V*, Gallimard, 1944, p295–322.

Twan van Laarhoven. 2009. CPS-Based Functional References. (July 2009). http://www.twanvl.nl/blog/haskell/cps-functional-references

Burghard von Karger. 2002. Temporal Algebra. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction (Lecture Notes in Computer Science)*, Roland Backhouse, Roy Crole, and Jeremy Gibbons (Eds.), Vol. 2297. Springer-Verlag, 310–386. https://doi.org/10.1007/3-540-47797-7_9

Philip Wadler. 1987. The Concatenate Vanishes. (Dec. 1987). University of Glasgow. Revised November 1989.