
Publisher's PDF, also known as Version of record

License (if available):
CC BY

Link to published version (if available):
10.22152/programming-journal.org/2017/1/7
10.5281/zenodo.400437

Link to publication record in Explore Bristol Research
PDF-document

This is the final published version of the article (version of record). It first appeared online via AOSA at http://programming-journal.org/2017/1/7/. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research
General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/user-guides/explore-bristol-research/ebr-terms/
Profunctor Optics

Modular Data Accessors

Matthew Pickering\textsuperscript{a}, Jeremy Gibbons\textsuperscript{b}, and Nicolas Wu\textsuperscript{a}

\textsuperscript{a} University of Bristol
\textsuperscript{b} University of Oxford

Abstract Data accessors allow one to read and write components of a data structure, such as the fields of a record, the variants of a union, or the elements of a container. These data accessors are collectively known as \textit{optics}; they are fundamental to programs that manipulate complex data. Individual data accessors for simple data structures are easy to write, for example as pairs of ‘getter’ and ‘setter’ methods. However, it is not obvious how to combine data accessors, in such a way that data accessors for a compound data structure are composed out of smaller data accessors for the parts of that structure. Generally, one has to write a sequence of statements or declarations that navigate step by step through the data structure, accessing one level at a time—which is to say, data accessors are traditionally \textit{not} first-class citizens, combinable in their own right.

We present a framework for \textit{modular data access}, in which individual data accessors for simple data structures may be freely combined to obtain more complex data accessors for compound data structures. Data accessors become first-class citizens. The framework is based around the notion of \textit{profunctors}, a flexible generalization of functions. The language features required are higher-order functions (‘lambdas’ or ‘closures’), parametrized types (‘generics’ or ‘abstract types’) of higher kind, and some mechanism for separating interfaces from implementations (‘abstract classes’ or ‘modules’). We use Haskell as a vehicle in which to present our constructions, but other languages such as Scala that provide the necessary features should work just as well. We provide implementations of all our constructions, in the form of a literate program: the manuscript file for the paper is also the source code for the program, and the extracted code is available separately for evaluation. We also prove the essential properties, demonstrating that our profunctor-based representations are precisely equivalent to the more familiar concrete representations. Our results should pave the way to simpler ways of writing programs that access the components of compound data structures.

ACM CCS 2012

- Software and its engineering → Abstract data types; Patterns; Polymorphism;

Keywords lens, traversal, compositionality
1 Introduction

Modularity is at the heart of good engineering, since it encourages a separation of concerns whereby solutions to problems can be composed from solutions to subproblems. Compound data structures are inherently modular, and are key to software engineering. A key issue when dealing with compound data structures is in accessing their components—specifically, extracting those components, and modifying them. Since the data structures are compound, we should expect their data accessors to be modular, that is, for data accessors onto compound data structures to be assembled out of data accessors for the components of those structures.

There has been a recent flourishing of work on lenses [9] as one such mechanism for data access, bringing programming language techniques to bear on the so-called view–update problem [2]. In the original presentation, a lens onto a component of type \(A\) within a larger data structure of type \(S\) consists of a function \(\text{view} :: S \rightarrow A\) that extracts the component from its context, and a function \(\text{update} :: A \times S \rightarrow S\) that takes a new component of type \(A\) and an old data structure of type \(S\) and yields a new data structure with the component updated.

We can generalize this presentation, allowing the new component to have a different type \(B\) from the old one of type \(A\), and the new compound data structure correspondingly to have a different type \(T\) from the old one of type \(S\). This is a strict generalisation; the earlier definition can be retrieved by specialising the types to \(A = B\) and \(S = T\). The situation is illustrated in Figure 1(a); the \(\text{view}\) function is a simple arrow \(S \rightarrow A\), whereas the \(\text{update}\) function has type \(B \times S \rightarrow T\), so is illustrated by an ‘arrow’ with two tails and one head. We assemble these functions into a single entity representing the lens as a whole:

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

This definition declares that the type \(\text{Lens}\), parametrized by four type variables \(a, b, s, t\), has a single constructor, also called \(\text{Lens}\); a value of that type consists of the constructor applied to a record consisting of two fields, one called \(\text{view}\) of type \(s \rightarrow a\) and one called \(\text{update}\) of type \(b \times s \rightarrow t\). (We use an idealization of Haskell as a vehicle for our presentation. See Appendix A for a summary of Haskell notation, and an explanation of those idealizations.)

For example, there is a lens onto the left component of a pair:

\[
\pi_1 :: \text{Lens } a b (a \times c) (b \times c)
\]

\[
\pi_1 = \text{Lens } \text{view } \text{update where}
\]

\[
\text{view } (x, y) = x
\]

\[
\text{update } (x', (x, y)) = (x', y)
\]

That is, the value \(\pi_1\) is instance of the type \(\text{Lens}\) in which the third and fourth type parameters are instantiated to pair types; the value itself is obtained by applying the constructor \(\text{Lens}\) to the two functions \(\text{view}\) and \(\text{update}\), whose definitions are provided by the \(\text{where}\) clause. Thus, \(\pi_1\) is the lens whose \(\text{view}\) function extracts the first component of a pair, and whose \(\text{update}\) function overwrites that first component.
In the case of $\pi_1$, the element in focus—the left component of the pair—is directly and separately represented in the compound data structure, as the first component of the pair. The component in focus need not be so explicitly represented. For example, there is also a lens onto the sign of an integer (where, for simplicity, we say that the sign denotes whether the integer is non-negative, rather than being three-valued):

$$sign :: Lens \text{Bool} \text{Bool} \text{Integer} \text{Integer}$$

$$sign = Lens \text{view} \text{update} \text{where}$$

$$\text{view} \ x = (x \geq 0)$$

$$\text{update} \ (b, x) = \text{if} \ b \ \text{then} \ \text{abs} \ x \ \text{else} \ -\text{(abs} \ x)$$

Thus, $sign$ is a lens onto a boolean within an integer; the $\text{view}$ function extracts the sign, and the $\text{update}$ function enforces a new sign while preserving the absolute value. Note that $sign$ is a monomorphic lens, whereas $\pi_1$ was polymorphic: the boolean sign can be replaced only with another boolean, not with a value of a different type.

Analogously to lenses, one could consider compound data structures of several variants. Given a compound data structure of type $S$, one of whose possible variants is of type $A$, one can access that variant via a function $match :: S \to S + A$ that ‘downcasts’ to an $A$ if possible, and yields the original $S$ if not; conversely, one may update the data structure via a function $build :: A \to S$ that ‘upcasts’ a new $A$ to the compound type $S$. Such a pair of functions is formally dual to a lens (in the mathematical sense of ‘reversing all arrows’), and continuing the optical metaphor has been dubbed a prism; prisms are to sum datatypes as lenses are to product datatypes.

More generally, one can again allow the new component to be of a different type $B$ from the old one of type $A$, and the new compound structure correspondingly to have a different type $T$ from the old one of type $S$. The situation is illustrated in Figure 1(b); this time it is the $\text{build}$ function of type $B \to T$ that is a simple arrow, whereas $\text{match}$ has type $S \to T + A$ and performs a case analysis, so is illustrated by an ‘arrow’ with one tail and two heads. Again, we assemble these two functions together into a record, leading to the following declaration of Prism as a four-parameter type, constructed by applying the constructor (also called Prism) to a record consisting of two functions $\text{match}$ and $\text{build}$.

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

For example, there is a prism onto an optional value, downcasting to that value if present, for which upcasting guarantees its presence:
Profunctor Optics

the :: Prism a b (Maybe a) (Maybe b)
the = Prism match build where
    match (Just x) = Right x
    match Nothing = Left Nothing
    build x = Just x

An optional value of type Maybe A is either of the form Just x for some x :: A, or simply Nothing. The match field of the performs a case analysis on such a value, yielding a result of type Maybe B + A, namely Right x when the optional x is present and Left Nothing otherwise. The build function simply injects a new value x :: B into the option type Maybe B.

Less trivially, there is a prism to provide possible access to a floating-point number as a whole number, ‘downcasting’ a Double to an Integer whenever the fractional part is zero.

whole :: Prism Integer Integer Double Double
whole = Prism match build where
    match x |
         | f == 0  = Right n
         | otherwise = Left x
         where (n,f) = properFraction x
    build = fromIntegral

The fromIntegral function converts from Integer to Double, while properFraction x returns a pair (n,f) where n::Integer is the integer part of x and f::Double is the fractional part. (Of course, there are the usual caveats about floating-point precision.)

For any given kind of compound data structure, it is usually straightforward to provide such accessors. However, when it comes to composing data structures out of parts, the data accessors do not compose conveniently. For example, there is a lens onto the leftmost component A of a nested pair (A × B) × C; but it is very clumsy to write that lens in terms of the existing lens π₁ for non-nested pairs:

π₁₁ :: Lens a b ((a × c) × d) ((b × c) × d)
π₁₁ = Lens view update where
    Lens v u = π₁
    view     = v · v
    update (x',xyz) = u (xy',xyz) where
                    xy  = v xyz
                    xy' = u (x',xy)

(here, the first local definition matches the pattern Lens v u against π₁, thereby binding v and u to the two fields of π₁). In fact, for the update method it is clearer to resort instead to first principles, defining \(update(x',((x,y),z))=((x',y),z)\); this points to a failure of modularity in our abstraction.

The situation is even worse for heterogeneous composite data structures. For example, we might want to access the A component of a compound type \((1 + A) \times B\) built using both sums and products. We might further hope to be able to construct this

7-4
accessor onto the optional left-hand component by composing the prism the onto an optional value and the lens $\pi_1$ onto a left-hand component. However, the composite accessor is not a lens, because it cannot guarantee a view as an $A$; neither is it a prism, because it cannot build the composite data structure from an $A$ alone. We cannot even express this combination; our universe of accessors is not closed under the usual operations for composing data. The abstraction is clearly broken.

Lenses and prisms, and some other variations that we will see in Section 2, have collectively been called optics. In this paper, we present a different representation of optics, which fixes the broken abstraction for lenses, prisms, and the other optics. The representation is based on the notion of profunctors, a generalization of functions; we introduce and explain the ideas as we go along. We call this new representation profunctor optics. It provides accessors for composite data that are trivially composed from accessors for the parts of the data, using ordinary function composition: for example, lenses become easily combinable with lenses, and lenses become combinable at all with prisms—hence modular data accessors. Moreover, the profunctor representation reveals a lattice structure among varieties of optics—structure that remains hidden with the concrete representations.

The constructions we present are not in fact new; they are Haskell folklore, having been introduced by others in the form of Internet Relay Chat comments, blog posts, sketches of libraries, and so on. But they deserve to be better known; our main contribution is to write these constructions up in a clear and consistent manner.

The remainder of this paper is structured as follows. In Section 2, we introduce a common specialization of lenses and prisms (called adapters) and a common generalization (called traversals), both of which we introduce in this section. Section 3 introduces the notion of profunctor, the main technical device on which we depend. Section 4 revisits the four varieties of optic with new representations in terms of profunctors. Section 5 shows how the profunctor representation supports modular construction of accessors for compound data structures in ways that the concrete representations do not. Section 6 summarizes prior and related work, and Section 7 concludes.

The paper itself is a literate script; all the code is embedded in the paper, has been explicitly type-checked, and is available for experimentation [33]. We do not make essential use of any fancy Haskell features; all that is really needed are higher-order functions, higher-kindred parametrized types, and some mechanism for separating interfaces from implementations, all of which are available in other languages. For the benefit of non-Haskellers, Appendix A summarises the Haskell notation and standard functions that we use; Appendix B sketches an alternative implementation in Scala; and Appendix C formally states and proves equivalences between the concrete and profunctor optic representations.

## 2 Optics, concretely

We have already seen two distinct varieties of optic, namely lenses and prisms. It turns out that they have a common specialization, which we call adapters, and a common generalization, which we call traversals, both of which we introduce in this section.
2.1 Adapters

When the component being viewed through a lens is actually the whole of the structure, then the lens is essentially a pair of functions of types $S \rightarrow A$ and $B \rightarrow T$; there is no virtue in the update function taking the old $S$ value as an argument as well, because this will be completely overwritten. Dually, when the variant being accessed through a prism is actually the sole variant, the prism is again essentially a pair of functions of types $S \rightarrow A$ and $B \rightarrow T$; there is no virtue in the matching function having a fallback option, because the match will always succeed. This is illustrated in Figure 2. We introduce an abstraction for such a pair of functions, which we call an adapter.

\[
data Adapter a b s t = Adapter \{ from :: s \rightarrow a, to :: b \rightarrow t \}\]

It is often the case that \texttt{from} and \texttt{to} are in some sense each other’s inverses; but we will not attempt to enforce that property, nor shall we depend on it.

Although adapters look like rather trivial data accessors, they are very useful as ‘plumbing’ combinators, converting between representations. For example, we introduced earlier the composite optic $\pi_{11}$ to access the $A$ component buried within a nested $(A \times B) \times C$ tuple. Now suppose that we want to access the $A$ component in a different but isomorphic tuple type $A \times B \times C$ as well. We do not need a separate lens in order to allow this; it suffices to combine $\pi_{11}$ with the isomorphism

\[
\begin{align*}
\text{flatten} ::& \quad \text{Adapter} (a \times b \times c) (a' \times b' \times c') (a \times b) (a' \times b') (a' \times b' \times c') \\
&= \text{Adapter from to where} \\
&\text{from} ((x,y),z) = (x,y,z) \\
&\text{to} (x,y,z) = ((x,y),z)
\end{align*}
\]

which serves as an adapter between the two tuple types.

2.2 Traversal

A \textit{traversable} datatype is a container datatype (such as lists, or trees), in which the data structures a finite number of elements, and an ordering on the positions of those elements. Given such a traversable data structure, one can traverse it, visiting each of the elements in turn, in the given order. When the container is polymorphic, one may vary the type of the elements in the process; for example, turning a tree of integers into a tree of characters. Moreover, because the ordering on positions is explicit, one may safely apply an effectful operation to each element, for example performing I/O or manipulating some mutable variable; the traversal of the whole structure sequences the effects arising from the elements in an order determined by the positions of those elements [12, 23].

In a pure language such as Haskell, we express a class of effects as a datatype of effectful computations. The best known example of such a datatype is \textit{monads} [39].
It turns out that we do not need the full expressive power of monads for traversals; the more restrictive abstraction of applicative functors [23] suffices. The interfaces modelling functors and applicative functors are represented using type classes:

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

class Functor f => Applicative f where
  pure :: a -> f a
  (<>') :: f (a -> b) -> f a -> f b
```

These declarations introduce two classes `Functor` and `Applicative` of operations on types, with the latter a subclass of the former. For `F` to be a functor, one has to provide a function `fmap` of the declared type; one can think of the type `F A` denoting a certain kind of ‘containers of As’, such as lists, and of `fmap f` as applying `f` to each element of such a container. Similarly, one can think of applicative functor `F` as representing a certain class of effects, and the type `F A` as the type of ‘computations that may have effects of type `F` when run, and will yield a result of type `A'`; ordinary functions of type `A -> B` can be thought of as ‘effectful functions’ from `A` to `B`, having effects modelled by `F`. The `pure` operation lifts a plain value to a trivial computation that actually has no effects and simply returns the given value; alternatively, one can think of `pure` itself as an effectful version of the identity function. The operator `<>'` acts to combine computations: if `m :: F (A -> B)` is a computation that effectfully returns an `A -> B` function, and `n :: F A` similarly a computation that effectfully returns an `A`, then `m <>' n` is the composite computation that runs `m` to get an `A -> B` function and runs `n` to get an `A` argument, then applies the function to the argument to return a `B` result overall, incurring the effects of both `m` and `n`.

As an example of a computation type, consider stateful computations represented as state-transforming functions:

```
data State s a = State { run :: s -> a x s }
```

so that the computation that increments an integer counter and returns a given boolean value is captured by the definition

```
inc :: Bool -> State Integer Bool
inc b = State (λn -> (b, n + 1))
```

For any state type `S`, the type `State S` is an applicative functor:

```
instance Functor (State s) where
  fmap f m = State (λs -> let (x, s') = run m s in (f x, s'))

instance Applicative (State s) where
  pure x = State (λs -> (x, s))
  m <>' n = State (λs -> let (f, s') = run m s
                          (x, s'') = run n s'
                          in (f x, s''))
```
in which mapping applies a function to the returned result, a pure computation leaves the state unchanged, and sequential composition threads the state through the first then through the second computation.

Now, for an applicative functor \( F \), a traversal takes an effectful operation of type \( A \rightarrow FB \) on the elements of a container, and lifts this to an effectful computation of type \( S \rightarrow FT \) over the whole container, applying the operation to each element in turn. We say that container type \( S \) with elements of type \( A \) is traversable when there exist types \( B, T \) and a traversal function of type \( (A \rightarrow FB) \rightarrow (S \rightarrow FT) \) for each applicative functor \( F \) (which should satisfy some laws [12], not needed here).

For example, consider the datatype

\[
\text{data Tree a} = \text{Empty} \mid \text{Node (Tree a) a (Tree a)}
\]

of internally labelled binary trees, in which constructor \( \text{Empty} \) represents the empty tree and \( \text{Node t x u} \) the non-empty tree with root labelled \( x \) and children \( t, u \). This datatype is traversable; one of several possible orders of traversal is in-order:

\[
\text{inorder} :: \text{Applicative } f \Rightarrow (a \rightarrow f b) \rightarrow \text{Tree a} \rightarrow f (\text{Tree b})
\]

\[
\text{inorder m Empty} = \text{pure Empty}
\]

\[
\text{inorder m (Node t x u)} = ((\text{pure Node } \langle \star \rangle \text{ inorder m t}) \langle \star \rangle \text{ m x}) \langle \star \rangle \text{ inorder m u}
\]

which visits the root \( x \) after visiting the left child \( t \) and before visiting the right child \( u \). (The sequencing operation \( \langle \star \rangle \) is declared to associate to the left, like function application does, so the parentheses on the right-hand side of the second equation are redundant; we will henceforth omit them.) Thus, effectfully traversing each of the elements of the empty tree is pure, yielding simply the empty tree; and traversing a non-empty tree yields the (pure) assembly of the results of recursively traversing its left child, operating on the root label, and recursively traversing the right child, with the effects occurring in that order. For example, the computation

\[
\text{countOdd} :: \text{Integer} \rightarrow \text{State Integer Bool}
\]

\[
\text{countOdd n} = \text{if even n then pure False else inc True}
\]

increments a counter when the argument is odd and leaves it unchanged when the argument is even, and returns the parity of that argument; and so

\[
\text{inorder countOdd} :: \text{Tree Integer} \rightarrow \text{State Integer (Tree Bool)}
\]

performs an in-order traversal of a tree of integers, counting the odd ones, and returning their parities.

### 2.3 Traversals as concrete optics

Traversal can be seen as a generalisation of lenses and of prisms, providing access not just to a single component within a whole structure but onto an entire sequence of such components. Indeed, the type \( (A \rightarrow FB) \rightarrow (S \rightarrow FT) \) of witnesses to traversability of the container type \( S \) is almost equivalent to a pair of functions \( \text{contents :: S} \rightarrow A^n \) and
fill :: S × B^n → T, for some n being the number of elements in the container. The idea
is that contents yields the sequence of elements in the container, in the order specified
by the traversal, and fill takes an old container and a new sequence of elements and
updates the old container by replacing each of the elements with a new one. Roughly
speaking, for singleton containers (n = 1) this specialises both to lenses and to prisms.
However, a factorization into two functions contents and fill is not quite right, because
the appropriate value of the exponent n depends on the particular container in S,
and must match for applications of contents and fill: one can in general only refill
a container with precisely the same number of elements as it originally contained.
However, the dependence can be captured by tupling together the two functions and
using a common existentially quantified length: the traversable type S is equivalent to
∃n. A^n × (B^n → T). This fact is not obvious, but is well established [3, 14].

We can capture the result of that tupled pair of functions via the following datatype:

```haskell
data FunList a b t = Done t | More a (FunList a b (b → t))
```

This datatype was introduced by van Laarhoven [19]. It is a so-called nested datatype [4],
because in the More case a larger value of type FunList A B T is constructed not from
smaller values of the same type, but from a value of a different type FunList A B (B → T).
One may verify inductively that FunList A B T is isomorphic to ∃n. A^n × (B^n → T): the
Done case consists of simply a T, corresponding to n = 0; and the More case consists
of an A and an A^n × (B^n → (B → T)) for some n, and by isomorphisms of products and
function spaces we have A × (A^n × (B^n → (B → T))) ≃ A^{n+1} × (B^{n+1} → T).

The isomorphism between FunList A B T and T + (A × (FunList A B (B → T))) is
witnessed by the following two functions:

```haskell
out :: FunList a b t → t + (a, FunList a b (b → t))
out (Done t) = Left t
out (More x l) = Right (x, l)

inn :: t + (a, FunList a b (b → t)) → FunList a b t
inn (Left t) = Done t
inn (Right (x, l)) = More x l
```

Now, a traversal function of type (A → F B) → (S → F T) for each applicative functor F
yields an isomorphism S ≃ FunList A B T. In order to construct the transformation
from S to FunList A B T using such a traversal function, we require FunList A B to be an
applicative functor:

```haskell
instance Functor (FunList a b) where
  fmap f (Done t) = Done (f t)
  fmap f (More x l) = More x (fmap (f ·) l)
```

```haskell
instance Applicative (FunList a b) where
  pure = Done
  Done f φ l' = fmap f l'
  More x l φ l' = More x (fmap flip l φ l')
```

The actual definitions may appear obscure, but in essence they make FunLists a kind of
sequence, with three operations corresponding to mapping, the empty sequence, and
Profunctor Optics

concatenation. We also require an operation of type \( A \rightarrow \text{FunList} \ A \ B \ B \) on elements, which we call `single` as it parcels up an element as a singleton `FunList`:

\[
\text{single} :: a \rightarrow \text{FunList} \ a \ b \ b \\
\text{single} \ x = \text{More} \ x \ (\text{Done} \ \text{id})
\]

We can use `single` as the body of a traversal, instantiating the applicative functor \( F \) to `FunList \ A \ B \`. This traversal will construct a singleton \( \text{FunList} \) for each element of a container, then concatenate the singletons into one long `FunList`. In particular, this gives \( t \text{single} :: S \rightarrow \text{FunList} \ A \ B \ T \) as one half of the isomorphism \( S \simeq \text{FunList} \ A \ B \ T \). Conversely, we can retrieve the traversable container from the `FunList`:

\[
\text{fuse} :: \text{FunList} \ b \ b \ t \rightarrow t \\
\text{fuse} \ (\text{Done} \ t) = t \\
\text{fuse} \ (\text{More} \ x \ l) = \text{fuse} \ l \ x
\]

This motivates the following definition of concrete traversals:

\[
\text{data} \ \text{Traversal} \ a \ b \ s \ t = \text{Traversal} \ \{ \text{extract} :: s \rightarrow \text{FunList} \ a \ b \ t \}\n\]

The situation is illustrated in Figure 3. The upper collection of left-to-right arrows represents the `contents` aspect, extracting the sequence of \( A \) elements from a container; the lower collection of right-to-left arrows represents the `fill` aspect, generating a new container from a fresh sequence of \( B \) elements. However, it is more precise to think of these as one combined arrow `extract` from \( S \), generating both the \( A \s\) and the mapping back from the \( B \s\) to the \( T \).

As another example, `inorder single`:: `Tree a` \( \rightarrow \text{FunList} \ a \ b \ (\text{Tree} \ b) \) extracts the in-order sequence of elements from a tree, and moreover provides a mechanism to refill the tree with a new sequence of elements. This type matches the payload of a concrete traversal; so we can define concrete in-order traversal of a tree by:

\[
\text{inorderC} :: \text{Traversal} \ a \ b \ \text{(Tree} \ a) \ (\text{Tree} \ b) \\
\text{inorderC} = \text{Traversal} \ \text{(inorder single)}
\]

3 Profunctors

The key to the design of a modular abstraction for data accessors is to identify what they have in common. Any data accessor for a component of a data structure is ‘function-like’, in the sense that reading ‘consumes’ the component from the data structure and writing ‘produces’ an updated component to put back into the data structure. The type structure of such function-like things—henceforth `transformers`—is
technically known as a profunctor. Profunctors can be represented by the following type class:

```haskell
class Profunctor p where
dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'
```

This says that the two-place operation on types $P$ is a Profunctor if there is a suitable definition of the function $\text{dimap}$ with the given type. Think of $P A B$ as a type of `transformers that consume $A$s and produce $B$s', with different instantiations of $P$ corresponding to different notions of `function-like', as illustrated in Figure 4. One can think of $\text{dimap } f g h$ as 'wrapping' the transformer $h$ in a preprocessor $f$ and a postprocessor $g$. The crucial point is that a transformer is covariant in what it produces, but contravariant in what it consumes; hence the reversal of the arrow ($a' \to a$ rather than $a \to a'$) in the type of the preprocessor, the first argument of $\text{dimap}$. The term 'profunctor' comes from category theory, although much of the categorical structure gets lost in translation.

Instances of the Profunctor class should satisfy two laws about the interaction between $\text{dimap}$ and function composition:

1. $\text{dimap id id} = \text{id}$
2. $\text{dimap } (f' \cdot f) \cdot (g' \cdot g) = \text{dimap } f g \cdot \text{dimap } f' g'$

Note again the contravariance in the preprocessor argument in the second law.

The canonical example of transformers is, of course, functions themselves; and indeed, the function arrow $\to$ on types, for which $(\to) A B = A \to B$, is an instance:

```haskell
instance Profunctor (\to) where
dimap f g h = g \cdot h \cdot f
```

The reader should see that the contravariance in the first argument is necessary. It is instructive to verify for oneself that the definition of $\text{dimap}$ for function types does indeed satisfy the two profunctor laws.

Plain functions are, of course, not the only instantiation of the abstraction—if they were, the abstraction would not be very useful. Functions that return a result together with a boolean flag are another instance, as illustrated in Figure 4(b). So are functions that return a pair of results, as illustrated in Figure 4(c). This pattern generalizes to functions of the form $A \to F B$ for some functor $F$:

```haskell
data UpStar f a b = UpStar {unUpStar :: a \to f b}
```
Profunctor Optics

Any functor \( F \) lifts in this manner to a profunctor:

```haskell
instance Functor f ⇒ Profunctor (UpStar f) where
dimap f g (UpStar h) = UpStar (fmap g · h · f)
```

(Indeed, the construction dualizes, to functions of the form \( FA \to B \), and the two constructions may be combined for functions of the form \( FA \to GB \); but we do not need that generality for this paper.)

We will, however, have need of three refinements of the notion of profunctor, concerning its interaction with product and sum types. For the first, we say that a profunctor is cartesian if, informally, it can pass around some additional context in the form of a pair. This is represented by an additional method \( \text{first} \) that lifts a transformer of type \( PAB \) to one of type \( P(A \times C)(B \times C) \) for any type \( C \), passing through an additional contextual value of type \( C \):

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

For each instance \( P \), the method \( \text{first} \) should satisfy two additional laws, concerning coherence with product and the unit type:

\[
\text{dimap } \text{runit} \ \text{runit}' \ h = \text{first } h \\
\text{dimap } \text{assoc} \ \text{assoc}' (\text{first } (\text{first } h)) = \text{first } h
\]

(and symmetrically for \( \text{second} \)), where \( \text{runit} :: a \times 1 \to a \) and \( \text{runit}' :: a \to a \times 1 \) are witnesses to the unit type being a right unit of the cartesian product, and \( \text{assoc} :: a \times (b \times c) \to (a \times b) \times c \) and \( \text{assoc}' :: (a \times b) \times c \to a \times (b \times c) \) are witnesses to the associativity of product. (Note the typing in the unit law, which instantiates \( C \) to 1: instead of passing around trivial additional context, one may discard it then recreate it.) To be precise, one might call such profunctors cartesianly strong, because \( \text{first} \) acts as a categorical ‘strength’ with respect to cartesian product; we abbreviate this more precise term to simply ‘cartesian’. The function arrow is obviously cartesian:

```haskell
instance Cartesian (→) where
  first h = cross h id
  second h = cross id h
```

(where \( \text{cross } f \ g \ (x,y) = (f x, g y) \) applies two functions to a pair of arguments). So too are functions with structured results, as captured by \( \text{UpStar} \):

```haskell
instance Functor f ⇒ Cartesian (UpStar f) where
  first (UpStar unUpStar) = UpStar (rstrength · cross unUpStar id)
  second (UpStar unUpStar) = UpStar (lstrength · cross id unUpStar)
```

where the so-called ‘right strength’

\[
\text{rstrength} :: \text{Functor } f \Rightarrow ((f a) \times b) \to f (a \times b) \\
\text{rstrength } (fx,y) = \text{fmap } (\cdot , y) fx
\]
distributes copies of a $B$-value over an $F A$ structure, and symmetrically for left strength—here, the function $(.,y)$ takes $x$ to $(x,y)$. But it is not always so obvious how to thread the contextual values through a profunctor. In particular, there is no general construction for the dual case of functions with structured arguments. For example, when $F$ is the functor $Pair$ yielding pairs of elements, the dual case entails putting together a function of type $Pair A \to B$ with a $Pair (A \times C)$ to make a $B \times C$; there are two input $Cs$ from which to choose the output, with neither being canonical. Worse, when $F = Maybe$, there is not necessarily a $C$ in the input at all.

Similarly, there is a refinement of profunctors that can be lifted to act on sum types:

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

Informally, if $h :: P A B$ is a transformer of $A$s into $B$s, then $left h :: P (A + C) (B + C)$ acts on the $A$s in a sum type $A + C$, turning them into $B$s and leaving the $C$s alone.

For each instance $P$, the method $left$ should satisfy two additional laws, concerning coherence with sum and the empty type:

```
dimap rzero rzero' h = left h
    dimap coassoc' coassoc (left (left h)) = left h
```

(and symmetrically for $right$), where $rzero :: a + 0 \to a$ and $rzero' :: a \to a + 0$ are witnesses to the empty type $0$ being a right unit of sum, and $coassoc :: a + (b + c) \to (a + b) + c$ and $coassoc' :: (a + b) + c \to a + (b + c)$ are witnesses to the associativity of sum. (Again, note the typing of the zero law, instantiating $C = 0$: instead of lifting to a trivial sum, one may discard and then recreate the trivial missing information.) To be precise, one might call such profunctors co-cartesianly strong, because the methods act as left and right strengths for the co-Cartesian structure; but we will stick with the abbreviation ‘co-cartesian’.

The function arrow is obviously co-cartesian:

```
instance Cocartesian (→) where
    left h = plus h id
    right h = plus id h
```

(where $plus f g$ takes $Left x$ to $Left (f x)$ and $Right y$ to $Right (g y)$). So too are functions with structured results, provided that there is an injection $A \to FA$ of pure values into that structure. For convenience, we capture that requirement here as the method $pure$ of the type class $Applicative$, since we have introduced this already:

```
instance Applicative f ⇒ Cocartesian (UpStar f) where
    left (UpStar unUpStar) = UpStar (either (fmap Left ∙ unUpStar) (pure ∙ Right))
    right (UpStar unUpStar) = UpStar (either (pure ∙ Left) (fmap Right ∙ unUpStar))
```

However, that constraint is stronger than necessary, because we do not need here the $\langle\rangle$ method of the $Applicative$ class. (Again, there is no similar construction for functions with structured arguments.)
The third refinement is a class of profunctors that support a form of parallel composition (in the sense of ‘independent’ rather than ‘concurrent’):

```haskell
class Profunctor p ⇒ Monoidal p where
  par :: p a b → p c d → p (a × c) (b × d)
  empty :: p 1 1
```

Informally, if \( h :: P A B \) and \( k :: P C D \) are transformers of \( A \)s into \( B \)s and of \( C \)s into \( D \)s, respectively, then \( \text{par} \ h \ k \) transforms \( A \times C \) pairs into \( B \times D \) pairs by acting independently on each component of the pair; and \( \text{empty} \) is a trivial transformer of unit values into unit values.

For each \( \text{Monoidal} \) instance \( P \), the two operations \( \text{par} \) and \( \text{empty} \) should satisfy some laws concerning coherence with the product structure: they should form a monoid, up to monoidal isomorphisms on the value types:

\[
\begin{align*}
\text{dimap assoc assoc'} \ (\text{par} \ (\text{par} \ h \ j) \ k) &= \text{par} \ h \ (\text{par} \ j \ k) \\
\text{dimap runit runit'} \ h &= \text{par} \ h \ \text{empty} \\
\text{dimap lunit lunit'} \ h &= \text{par} \ \text{empty} \ h
\end{align*}
\]

where \( \text{lunit} :: 1 \times a → a \) and \( \text{lunit'} :: a → 1 \times a \) are the witnesses to the unit type being the left as well as the right unit of cartesian product.

The function arrow is obviously monoidal:

```haskell
instance Monoidal (→) where
  par = cross
  empty = id
```

For functions with structured results (that is, of the form \( A → F B \)) to be monoidal, it is necessary to be able to ‘zip’ together two \( F \)-structures. This can be done when \( F \) is an applicative functor:

```haskell
instance Applicative f ⇒ Monoidal (UpStar f) where
  empty = UpStar pure
  par h k = UpStar (pair (unUpStar h) (unUpStar k))
```

where for the definition of \( \text{par} \) we make use of the lifting

\[
\begin{align*}
\text{pair} :: \text{Applicative} \ f ⇒ (a → f b) → (c → f d) → (a, c) → f (b, d) \\
\text{pair} \ h \ k \ (x, y) &= \text{pure} (,) \ h \ x \ (,) \ k \ y
\end{align*}
\]

to applicative functors of the pairing function \( (,) \) defined by \( (,) \ x \ y = (x, y) \).

### 4 Optics in terms of profunctors

Plain data accessors might be modelled simply as transformers, values of some type that is an instance of the \( \text{Profunctor} \) type class as discussed above. However, such a
model will not address the problems of compositionality that motivated us in the first place. Instead, we represent data accessors as mappings between transformers:

\[
\text{type } \text{Optic } p \ a \ b \ s \ t = p \ a \ b \rightarrow p \ s \ t
\]

Informally, when \(S\) is a composite type with some component of type \(A\), and \(T\) similarly a composite type in which that component has type \(B\), and \(P\) is some notion of transformer, then we can think of a data accessor of type \(\text{Optic } P \ A \ B \ S \ T\) as lifting a component transformer of type \(P \ A \ B\) to a whole-structure transformer of type \(P \ S \ T\).

We will retrieve equivalents of our original definitions of lens, prism, and so on by placing various constraints on \(P\), starting with requiring \(P\) to be a \text{Profunctor}. Crucially, different varieties of optic all now have the same form—in particular, they are all simply functions—and so they will compose straightforwardly; they may involve different constraints on \(P\), but those constraints simply conjoin.

The situation is somewhat analogous to that with real, imaginary, and complex numbers. The concrete representations of optics are like having real numbers and imaginary numbers but not arbitrary complex numbers. One can combine two real numbers using addition to make a third, and combine two imaginary numbers to make a third; but one cannot combine a real number and an imaginary number with addition, because the result is in general neither real nor imaginary. But once one invents complex numbers, now arbitrary combinations by addition are expressible. Moreover, the complex numbers embed faithful representations of the real numbers and of the imaginary numbers; the space of complex numbers is strictly richer than the union of the spaces of real and of imaginary numbers. (The analogy only goes so far. Composition of concrete lenses is not easily defined, and only becomes straightforward in the profunctor representation. It is as if addition of real numbers becomes more easily expressed by passage through the complex numbers.)

### 4.1 Profunctor adapters

Recall the concrete representation of adapters from Section 2.1:

\[
\text{data } \text{Adapter } a \ b \ s \ t = \text{Adapter} \{ \text{from } :: s \rightarrow a, \text{to } :: b \rightarrow t \}
\]

The two methods \text{from} and \text{to} of an \text{Adapter } A \ B \ S \ T do not generally compose, specifically when types \(A\) and \(B\) differ. However, if we could somehow transform \(A\) into \(B\)s, then we could make the two methods fit together; and moreover, we would then be able to transform \(S\)s into \(T\)s in the same way. Which is to say, there is an obvious mapping that takes an \text{Adapter } A \ B \ S \ T and a \(P \ A \ B\) and yields a \(P \ S \ T\), provided that \(P\) is a profunctor. This motivates the following datatype:

\[
\text{type } \text{AdapterP } a \ b \ s \ t = \forall p. \text{Profunctor } p \Rightarrow \text{Optic } p \ a \ b \ s \ t
\]

That is, an optic of type \text{AdapterP } A \ B \ S \ T is simply a function from \(P \ A \ B\) to \(P \ S \ T\) that works polymorphically in the profunctor type \(P\). It will turn out, somewhat surprisingly, that \text{AdapterP } A \ B \ S \ T is precisely equivalent to \text{Adapter } A \ B \ S \ T (see Appendix C for the proof); we are therefore justified in using \text{AdapterP} as a profunctor representation of adapters.
The translations between the two representations are not difficult to construct. We have already hinted at the translation from the concrete representation Adapter to the profunctor representation AdapterP:

\[
\text{adapterC2P} :: \text{Adapter} \ a \ b \ s \ t \ \rightarrow \ \text{AdapterP} \ a \ b \ s \ t \\
\text{adapterC2P} \ (\text{Adapter} \ o \ i) = \text{dimap} \ o \ i
\]

This definition repays a little contemplation: given functions \(o :: S \rightarrow A\) and \(i :: B \rightarrow T\), then \(\text{dimap} \ o \ i\) has type \(P \ A \ B \rightarrow P \ S \ T\) for any profunctor \(P\), as required.

The translation in the opposite direction takes a little more effort: what can we do with an \(l\) of type \(\text{AdapterP} \ A \ B \ S \ T\)? This function has type \(P \ A \ B \rightarrow P \ S \ T\) for arbitrary profunctor \(P\); if we are to use it somehow to construct an \(\text{Adapter} \ A \ B \ S \ T\), then it had better be the case that \(\text{Adapter} \ A \ B\) is a profunctor, a suitable instantiation for \(P\). Happily, this is the case:

\[
\text{instance} \ \text{Profunctor} \ (\text{Adapter} \ a \ b) \ \text{where} \\
\quad \text{dimap} \ f \ g \ (\text{Adapter} \ o \ i) = \text{Adapter} \ (o \cdot f) \ (g \cdot i)
\]

Informally, this \(\text{dimap}\) wraps the pair of functions \(o :: S \rightarrow A\) and \(i :: B \rightarrow T\) in a preprocessor \(f :: S' \rightarrow S\) and postprocessor \(g :: T \rightarrow T'\):

\[
S' \xrightarrow{f} S \overset{o}{\longrightarrow} A \quad B \overset{i}{\longrightarrow} T \xrightarrow{g} T'
\]

to yield a pair of functions of types \(S' \rightarrow A\) and \(B \rightarrow T'\). It is straightforward to check that this definition satisfies the two profunctor laws.

Now, we construct the trivial concrete adapter \(\text{Adapter} \ id \ id\) of type \(\text{Adapter} \ A \ B \ A \ B\), and use the profunctor adapter to lift that to the desired concrete adapter:

\[
\text{adapterP2C} :: \text{AdapterP} \ a \ b \ s \ t \ \rightarrow \ \text{Adapter} \ a \ b \ s \ t \\
\text{adapterP2C} \ l = l \ (\text{Adapter} \ id \ id)
\]

Again, it is a worthwhile exercise to verify the types: function \(l\) is applicable at arbitrary profunctors \(P\), but we use it here only for the specific profunctor \(P = \text{Adapter} \ A \ B\); then \(l\) transforms a \(P \ A \ B\) into a \(P \ S \ T\).

Note the essential use of profunctors in the translation. For \(\text{adapterC2P}\), it is tempting to pick a simpler translation: given an \(\text{Adapter} \ A \ B \ S \ T\), which is a pair of functions of types \(S \rightarrow A\) and \(B \rightarrow T\), and another function of type \(A \rightarrow B\), then one can construct a function of type \(S \rightarrow T\); that is, one can translate from \(\text{Adapter} \ A \ B \ S \ T\) to the pure function type \((A \rightarrow B) \rightarrow (S \rightarrow T)\). But this translation loses information, because there is no obvious translation back from here to the profunctor representation—in particular, it provides no way of constructing anything other than a pure function.

The proof that \(\text{adapterC2P}\) and \(\text{adapterP2C}\) are each other’s inverses, and hence that \(\text{Adapter} \ A \ B \ S \ T\) and \(\text{AdapterP} \ A \ B \ S \ T\) are equivalent, can be found in Appendix C. For the remaining varieties of optic, we present the constructions and discussion in a bit less detail.
4.2 Profunctor lenses

Recall the concrete representation of lenses from Section 1:

```
data Lens a b s t = Lens {view :: s → a, update :: b × s → t}
```

The occurrence of the product type in the argument of update suggests that the analogue of Lens will have something to do with cartesian profunctors. Indeed, we define the profunctor representation of lenses as follows:

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

That is, a profunctor lens \(\text{LensP} A B S T\) lifts a transformer on components \(P A B\) to a transformer on structures \(P S T\), for arbitrary cartesian profunctor \(P\).

Concrete lenses are themselves cartesian profunctors:

```
instance Profunctor (Lens a b) where
dimap f g (Lens v u) = Lens (v · f) (g · u · cross id f)
```

```
instance Cartesian (Lens a b) where
  first (Lens v u) = Lens (v · first) (fork (u · cross id first) (snd · snd))
  second (Lens v u) = Lens (v · second) (fork (first · snd) (u · cross id snd))
```

(where \(\text{fork} f g x = (f x, g x)\) applies two functions to a common argument to return a pair). The translations back and forth make crucial use of the lifting to products. For the translation from the concrete representation to the profunctor representation, we need to translate a concrete lens \(\text{Lens} v u :: \text{Lens} A B S T\) to a profunctor optic, which is a function of type \(\text{Optic} P A B S T\) that has to work for an arbitrary cartesian profunctor \(P\). In other words, given \(v :: S → A\) and \(u :: B × S → T\) and a transformer \(h :: P A B\) for some cartesian profunctor \(P\), we have to construct another transformer of type \(P S T\). Now, \(\text{first} h\) has type \(P (A × C) (B × C)\) for any type \(C\), and in particular, for \(C = S\). Then it suffices to wrap this transformer in a preprocessor \(S → A × S\) and a postprocessor \(B × S → T\), both of which are easy to construct:

```
lensC2P :: Lens a b s t → LensP a b s t
lensC2P (Lens v u) = dimap (fork v id) u · first
```

For the translation in the opposite direction, we use the same approach as for adapters. We are given the profunctor lens, \(l :: \text{Optic} P A B S T\), which will work for arbitrary cartesian profunctor \(P\); we have to construct a concrete lens of type \(\text{Lens} A B S T\). We note that \(\text{Lens} A B\) is itself a cartesian profunctor, so \(l\) is applicable at the type \(P = \text{Lens} A B\). We therefore construct the trivial concrete lens \(\text{Lens id} fst :: \text{Lens} A B A B\), and lift it using \(l\) to a \(\text{Lens} A B S T\) as required:

```
lensP2C :: LensP a b s t → Lens a b s t
lensP2C l = l (Lens id fst)
```

These definitions may seem somewhat mysterious; and indeed, they are surprising—at least, they were to the authors. But it is not necessary to have a robust intuition for
how they work. The important points are first, that \textit{lensC2P} and \textit{lensP2C} are inverses, so the two representations are equivalent (see Appendix C for the proofs); and second, that the profunctor representation supports composition of optics, which we will see in Section 5.

\subsection{Profunctor prisms}

Recall the \textit{Concrete representation of prisms from Section 1:}

\begin{verbatim}
data Prism a b s t = Prism { match :: s → t + a, build :: b → t }
\end{verbatim}

Dually to lenses, the occurrence of the sum type for \textit{match} suggests that the analogue of \textit{Prism} will have something to do with co-cartesian profunctors. Indeed, we define:

\begin{verbatim}
type PrismP a b s t = ∀ p. Cocartesian p ⇒ Optic p a b s t
\end{verbatim}

That is, a profunctor prism \textit{PrismP A B S T} lifts a transformer \textit{P A B} on components to a transformer \textit{P S T} on structures, for arbitrary co-cartesian profunctor \textit{P}.

Concrete prisms are themselves co-cartesian profunctors:

\begin{verbatim}
instance Profunctor (Prism a b) where
dimap f g (Prism m b) = Prism (plus g id · m · f) (g · b)
instance Cocartesian (Prism a b) where
left (Prism m b) = Prism (either (plus Left id · m) (Left · Right)) (Left · b)
right (Prism m b) = Prism (either (Left · Left) (plus Right id · m)) (Right · b)
\end{verbatim}

Again dually to lenses, the translations back and forth make crucial use of the lifting to sums. For the translation from the concrete to the profunctor representation, given the two functions \textit{match} :: \textit{S} → \textit{T} + \textit{A} and \textit{build} :: \textit{B} → \textit{T} that constitute the concrete prism, and a transformer \textit{h} :: \textit{P A B} for some co-cartesian profunctor \textit{P}, we have to construct another transformer of type \textit{P S T}. Now, \textit{right \textit{h}} has type \textit{P (C + A) (C + B)} for any type \textit{C}, and in particular for \textit{C = T}. Then it suffices to wrap this transformer in a \textit{preprocessor} \textit{S} → \textit{T} + \textit{A} and a \textit{postprocessor} \textit{T + B} → \textit{T}, both of which are easy to construct:

\begin{verbatim}
prismC2P :: Prism a b s t → PrismP a b s t
prismC2P (Prism m b) = dimap m (either id b) · right
\end{verbatim}

For the translation in the opposite direction, we use the lifting approach again. We are given the profunctor \textit{lens} \textit{l} :: \textit{Optic P A B S T}, which will work for an arbitrary co-cartesian profunctor \textit{P}. We note that \textit{Prism A B} is itself a co-cartesian profunctor, so \textit{l} is applicable at the type \textit{P = Prism A B}. We therefore construct the trivial concrete prism \textit{Prism Right id} :: \textit{Prism A B A B}, and lift it using \textit{l} to a \textit{Prism A B S T} as required:

\begin{verbatim}
prismP2C :: PrismP a b s t → Prism a b s t
prismP2C l = l (Prism Right id)
\end{verbatim}

Again, the proof of the pudding is in the facts that these two translations are inverses, so that the representations are equivalent (Appendix C), and that the profunctor representation supports composition (Section 5).
4.4 Profunctor traversals

Recall the concrete representation of traversals from Section 2.3:

```haskell
data Traversal a b s t = Traversal { extract :: s → FunList a b t }
```

The key step in the profunctor representation of traversals is to identify a function `traverse` that lifts a transformation `k :: P A B` from `A` to `B` to act on each of the elements of a `FunList` in order:

```haskell
traverse :: (Cocartesian p, Monoidal p) ⇒ p a b → p (FunList a c t) (FunList b c t)
traverse k = dimap out inn (right (par k (traverse k)))
```

Informally, `traverse k` uses `out` to analyse the `FunList`, determining whether it is `Done` or consists of `More` applied to a head and a tail; in the latter case (the combinator `right` lifts a transformer to act on the right-hand component in a sum type), it applies `k` to the head and recursively calls `traverse k` on the tail; then it reassembles the results using `inn`. For this inductive definition to be well founded, it is necessary that the `FunList` is finite, and therefore that the structures being traversed are finite too; this is no additional limitation, because data structures supporting a well-behaved traversal are necessarily finite anyway.

Traversals may then be represented as optics, in precisely the same form as lenses and prisms only with a stronger type class constraint:

```haskell
type TraversalP a b s t = ∀ p. (Cartesian p, Cocartesian p, Monoidal p) ⇒ Optic p a b s t
```

This definition makes `TraversalP A B S T` isomorphic to our earlier more direct notion `Traversal A B S T` of traversals. For the translation from `Traversal` to `TraversalP`, we are given `Traversal h :: Traversal A B S T` and a transformer `k :: P A B` on elements, for some cartesian, co-cartesian, monoidal profunctor `P`, which we have to lift to a transformer `P S T` on containers. We use `traverse` to lift `k` to obtain a transformer on `FunList`, which we then sandwich between preprocessor `h :: S → FunList A B T` and postprocessor `fuse :: FunList B B T → T` to obtain a transformer on actual containers:

```haskell
traversalC2P :: Traversal a b s t → TraversalP a b s t
traversalC2P (Traversal h) k = dimap h fuse (traverse k)
```

In the opposite direction, we have an optic `l :: P A B → P S T`, applicable for an arbitrary cartesian, co-cartesian, monoidal profunctor `P`, and an effectful operation `m :: A → F B` on elements for some applicative functor `F`, which we have to lift to a traversal `S → F T` over the whole container. Fortunately, `Traversal A B` is itself a cartesian, co-cartesian, monoidal profunctor:

```haskell
instance Profunctor (Traversal a b) where
dimap f g (Traversal h) = Traversal (fmap g · h · f)
```

```haskell
instance Cartesian (Traversal a b) where
first (Traversal h) = Traversal (λ (s, c) → fmap (,) (h s))
```
Profunctor Optics

second (Traversal h) = Traversal (λ(c,s) → fmap (c,) (h s))

instance Cocartesian (Traversal a b) where
  left (Traversal h) = Traversal (either (fmap Left · h) (Done · Right))
  right (Traversal h) = Traversal (either (Done · Left) (fmap Right · h))

instance Monoidal (Traversal a b) where
  par (Traversal h) (Traversal k) = Traversal (pair h k)
  empty = Traversal pure

We can therefore instantiate P to Traversal A B; it suffices to take single::A → FunList A B B, for which Traversal single is a trivial concrete traversal Traversal A B A B, and use l to lift this to Traversal A B S T as required.

traversalP2C :: TraversalP a b s t → Traversal a b s t
traversalP2C l = l (Traversal single)

These two translations are each other’s inverses, as shown in Appendix C. (We need the Cartesian constraint for the proofs of equivalence, if not for these definitions.)

In order to apply a traversal, it is useful to define an additional combinator traverseOf that turns a profunctor traversal into the kind of traversing function originally described in Section 2.2:

traverseOf :: TraversalP a b s t → (∀f. Applicative f ⇒ (a → f b) → s → f t)
traverseOf p f = unUpStar (p (UpStar f))

5 Composing profunctor optics

Let us return now to the motivating examples from Section 1, where we saw that concrete representations of optics do not support composition well. Things are, happily, much better with the profunctor representation. For example, recall the concrete representation π₁ :: Lens a b (a × c) (b × c) of the lens onto the first component of a pair. It is straightforward to translate this lens into the profunctor representation, with πP₁ = lensC2P π₁ :: LensP a b (a × c) (b × c). It is instructive to simplify this definition to first principles; expanding the definitions, we conclude that

πP₁ :: Cartesian p ⇒ p a b → p (a × c) (b × c)
πP₁ = dimap (fork fst id) (cross id snd) · first

The point here is that the definition of the profunctor lens is not complicated; however, neither is it obvious. Crucially, though, this definition supports composition trivially: since profunctor lenses are nothing but functions, they compose using function composition. In particular, the lens onto the left-most component of a nested pair, which presented us with difficulties in Section 1, may be written directly in terms of πP₁:

πP₁₁ :: LensP a b ((a × c) × d) ((b × c) × d)
πP₁₁ = πP₁ ∘ πP₁
Similarly, recall the prism \( \text{the} :: \text{Prism} \; a \; b \) \((\text{Maybe} \; a) \; (\text{Maybe} \; b)\) onto an optional value. Its concrete representation can again be directly translated to the profunctor representation:

\[
\text{theP} :: \text{PrismP} \; a \; b \; (\text{Maybe} \; a) \; (\text{Maybe} \; b)
\]

\[
\text{theP} = \text{prismC2P} \; \text{the}
\]

or, unpacking the definitions,

\[
\text{theP} = \text{dimap} \; (\text{maybe} \; (\text{Left} \; \text{Nothing}) \; \text{Right}) \; (\text{either} \; \text{id} \; \text{Just}) \; \text{right}
\]

(where \(\text{maybe} :: b \rightarrow (a \rightarrow b) \rightarrow \text{Maybe} \; a \rightarrow b\) deconstructs an optional value). And again, it is a profunctor optic, so it is nothing but a function, and may be combined with other optics using familiar function composition. For example, we may obtain an optic onto the first component of an optional pair:

\[
\text{theP} \cdot \pi_1 :: (\text{Cartesian} \; p, \text{Cocartesian} \; p) \Rightarrow \text{Optic} \; p \; a \; b \; (\text{Maybe} \; (a \times c)) \; (\text{Maybe} \; (b \times c))
\]

Note that in a sense the optic is constructed inside out—\(\pi_1\) gives access to a component inside a pair, and \(\text{theP}\) gives access to this pair inside a \(\text{Maybe}\)—whereas the more natural naming is arguably of the form ‘first projection of the optional value’. By composing the optics in the opposite order, we obtain instead a composite optic onto the optional first component of a pair:

\[
\pi_1 \cdot \text{theP} :: (\text{Cartesian} \; p, \text{Cocartesian} \; p) \Rightarrow \text{Optic} \; p \; a \; b \; (\text{Maybe} \; a \times c) \; (\text{Maybe} \; b \times c)
\]

In both cases, we get the conjunction of the \text{Cartesian} constraint of lenses and the \text{Cocartesian} constraint of prisms. Neither combination is purely a lens or purely a prism; they are not expressible using the concrete representations.

We can act on the component in focus under the optic. Specifically, the function arrow \((\rightarrow)\) is a profunctor, and a \text{Cartesian}, \text{Cocartesian}, and \text{Monoidal} one to boot; so any optic may be applied to a plain function, and will modify the components in focus using that function. For example, to square the integer in the left-hand component of an optional pair, we have

\[
(\text{theP} \cdot \pi_1) \; \text{square} \; (\text{Just} \; (3, \text{True})) = \text{Just} \; (9, \text{True})
\]

Traversals fit neatly into the scheme too. Recall from Section 2.2 the concrete representation \(\text{inorderC} :: \text{Traversal} \; a \; b \; (\text{Tree} \; a) \; (\text{Tree} \; b)\) of the in-order traversal of an internally labelled binary tree. This is straightforwardly translated into the profunctor representation:

\[
\text{inorderP} :: \text{TraversalP} \; a \; b \; (\text{Tree} \; a) \; (\text{Tree} \; b)
\]

\[
\text{inorderP} = \text{traversalC2P} \; \text{inorderC}
\]

and may then be composed with other profunctor optics, using ordinary function composition. Thus, if the tree is labelled with pairs, and we want to traverse only the first components of the pairs, we can use:
Once we have given such an optic specifying how to access the elements in the tree, we can actually ‘apply’ the optic by turning it back into a traversal function using `traverseOf`. For example, applying it to the body function `countOdd` will yield a traversal of trees of pairs whose first components are integers, counting the odd ones and returning their parities:

\[
\text{traverseOf } (\text{inorderP } \cdot \pi_P 1) \text{ countOdd} :: \text{Tree } (\text{Integer } \times c) \rightarrow \text{State Integer } (\text{Tree } (\text{Bool } \times c))
\]

Similarly, we can compose `inorderP` with a prism, `the`, to count the odd integers in a tree which optionally contains values at its nodes:

\[
\text{traverseOf } (\text{inorderP } \cdot \text{the}) \text{ countOdd} :: \text{Tree } (\text{Maybe a}) \rightarrow \text{State Integer } (\text{Tree } (\text{Maybe a}))
\]

These examples highlight the common pattern of programming with optics. We compositionally describe which parts of the data we want to access, and separately specify the operation we want to perform. The type system ensures that we can construct only appropriate combinations.

As a slightly more extended example, consider a `Book` of contact details, consisting of a `Tree` of `Entries`, where each `Entry` has a `Name` and `Contact` details, the latter being either a `Phone` number or a `Skype` identifier:

```haskell
type Number = String
type ID = String
type Name = String
data Contact = Phone Number | Skype ID
data Entry = Entry Name Contact
type Book = Tree Entry
```

It is straightforward to define a prism to access the possible phone number in a `Contact`, and a lens to access the `Contact` in an `Entry”—for example, by defining a concrete prism and lens, respectively, and translating each to the profunctor representation:

```haskell
phone :: PrismP Number Number Contact Contact
phone = prismC2P (Prism m Phone) where
  m (Phone n) = Right n
  m (Skype s) = Left (Skype s)
contact :: LensP Contact Contact Entry Entry
contact = lensC2P (Lens v u) where
  v (Entry n c) = c
  u (c', Entry n c) = Entry n c'
```

These may be combined, in order to access the possible phone number in an `Entry`:

```haskell
contactPhone :: TraversalP Number Number Entry Entry
contactPhone = contact \cdot phone
```
and combined further with in-order traversal, in order to access all the phone numbers in a Book of contacts:

\[
\text{bookPhones :: } \text{TraversalP } \text{Number } \text{Number } \text{Book } \text{Book} \\
\text{bookPhones = inorderP \cdot contact \cdot phone}
\]

If we have a function \( \text{tidyNumber :: Number \rightarrow Number} \) to normalize a phone number, perhaps to add spaces, parentheses, and hyphens according to local custom, then we can tidy a whole Book by tidying each phone number:

\[
\text{tidyBook :: Book \rightarrow Book} \\
\text{tidyBook = bookPhones tidyNumber}
\]

If we have a function \( \text{output :: Number \rightarrow IO Number} \) in the IO monad of input–output actions, which prints out a phone number and returns a copy of it too, then we can print each of the numbers in turn:

\[
\text{printBook :: Book \rightarrow IO Book} \\
\text{printBook = traverseOf bookPhones output}
\]

There is an applicative functor \( \text{Const } m \) for any monoid \( m \), and in particular for the list monoid; using this, we can extract a list of all the phone numbers in a book of contacts:

\[
\text{listBookPhones :: Book \rightarrow } [\text{Number}] \\
\text{listBookPhones = getConst \cdot traverseOf bookPhones (Const \cdot (\lambda x \rightarrow [x]))}
\]

6 Related work

Lenses were introduced by Foster, Pierce et al. [9] as a model of bidirectional transformations. Their motivation was to take a linguistic approach to the so-called view–update problem in databases [2], the problem there being, given a computed view table (analogous to the view method of our Lens), to propagate a modified view back as a corresponding update to the original source tables (analogous to our update).

A basic criterion for soundness of a bidirectional transformation is that it satisfies a pair of round-trip laws. In our terminology, this criterion presupposes a monomorphic lens \( \text{Lens } A A S S \), in which the types do not change. Given \( \text{Lens } v u \) of this type, the first law is that \( v (u (a, s)) = a \); informally, that a modified view is faithfully stored, so that it may later be retrieved. The second is that \( u (v s, s) = s \); informally, that propagating an unmodified view does not change the source. Such a lens is said to be well behaved. (As we discuss in Section 7, well-behavedness is orthogonal to the question of whether or not the profunctor representation matches the concrete one.)

In the simple case, the source factorizes cleanly into the view and its ‘complement’, for example when the view is the left component of a pair; in such cases, more can be said. Specifically, \( \text{Lens } v u \) will satisfy a third law, that \( u (a', (u (a, s))) = u (a', s); \)
Profunctor Optics

informally, a later update completely overwrites an earlier one. In that case, the lens is said to be very well behaved; in the database community, this is called a constant complement situation, because the complement is untouched by an update. This special case received earlier attention from programming language researchers, for example in the work of Kagawa [15] on representing mutable cells in pure languages, and of Oles [29] on modelling block structure in Algol-like languages.

One can consider programming with optics as an application of datatype-generic programming [11], that is, the construction of programs that are parametrized by the shape of the data they manipulate. The parameter has traditionally been a functor; for us, it is a profunctor. In particular, there is significant related work on datatype-generic traversals. The essential structure of the generic traversal is already apparent in the work of Meertens [24]. Other approaches, including Lämmel's Scrap Your Boilerplate (SYB) [21], Mitchell's uniplate library [25], Bringert's Compos library [5], McBride and Paterson's Traversable class [23], and O'Connor's Multiplate [26], all provide traversal functions of similar forms to our definition of traverse. In recent years, attention has turned to the in-depth comparison and study of these different definitions [3, 12, 14].

The SYB approach [21] is of particular note due to the recognition of the need to combine an effectful traversal with an operation that focusses in turn on each element of a data structure. The SYB implementation of this is ad-hoc, using dynamic type checking, but in our framework we can write the same programs by composing a traversal with a lens. Thus, another way to view our work is as a generalisation of effectful traversals.

More recent work has explored the so-called van Laarhoven representation [20, 26] in terms of functions of type \((A \to F B) \to (S \to F T)\) for various functors \(F\), which is the predominant representation currently used in Haskell. This representation shares many properties with the profunctor representation we describe, but is slightly less elegant (it requires instantiation of the functor \(F\) even when it is not needed—for example, for simple adapters). Other representations of lenses have also been explored [22], but these appear to lack extension to other varieties of optic at all.

As far as we are aware, prisms have not previously been described in the literature, and are only folklore amongst functional programmers. Reid Barton is credited by Edward Kmett [16] with the observation that there should exist a 'dual' to lenses which works with sum types. The development of the idea was then led by Kmett, Elliott Hird and Shachaf Ben-Kiki whilst working on the lens library [17]. Prisms are an implementation of first-class patterns, unlike other proposals; for example, Tullsen [37] recognised the close connection between data constructors and pattern matching. Pattern matching is the method which is used to deconstruct values constructed by data constructors; first-class patterns should also be able to re-build the values that they match. Pattern synonyms [32] are also bidirectional in this sense, but are not first-class. Scala has the closely related extractor objects [8] which have two special methods unapply and apply which correspond to matching and building respectively.

What is unique to the framework we have described is the explicit connection between these different kinds of generic functions. This is further highlighted by the representation allowing us to seamlessly upcast and use more specific optics in places where a less powerful optic would suffice—for example, using a lens as a traversal.
Bringing together these different styles of datatype-generic programming makes it straightforward to construct heterogeneous composite data accessors, a use case that is not possible in each framework individually.

There are several nascent implementations of profunctor-based optics. The most well developed is the purescript library purescript-profunctor-optics [10], which provides indexed optics in addition to the optics that we have described. The optics library [7] is a Javascript proof of concept implementation. Russell O’Connor’s Haskell implementation mezzolens [27] was instrumental in our understanding.

## Discussion

We have drawn together a series of folklore developments that together lead to a modular framework for data accessors. This framework accommodates adapters, which provide access via a change of representation; lenses, which provide access to a component of a product structure, such as a field of a record; prisms, which provide access to a component of a sum structure, such as one variant from a union; and traversals, which provide access to a sequence of components, such as the elements in a container datatype. Collectively, these four varieties of data accessor are called optics. Crucially, the four varieties of optic have a similar representation, and this form is closed under composition; this allows us to combine different varieties of optic, such as a lens with a prism, which is not possible with more direct representations.

The particular representation we use is mappings between transformers, where transformers are represented in terms of profunctors, a generalization of functions:

\[ \forall p . \text{Profunctor } p \Rightarrow p \to s t \]

In other words, it is a representation using higher-order functions rather than more concrete datatypes. This choice of representation is the essential trick, both accommodating the wide variety of apparently distinct optics, and straightforwardly supporting combinations via function composition. That this representation is even adequate comes as quite a surprise—it is salutary to reflect on Christopher Strachey’s observation of half a century ago [34] that

many of the more interesting developments of programming and programming languages come from the unrestricted use of functions, and in particular of functions which have functions as results

and yet we are still finding new applications of higher-order functions.

It is interesting to note that the four representations we have chosen for the four different varieties of optic form a lattice, as shown in Figure 5: adapters are special kinds of lens and of prism, and lenses and prisms are each special kinds of traversal. They all have the higher-order functional form quoted above, differing only in the constraints imposed on the parameter \( p \). A combination of different varieties of optic is also of the same form, but collects all the constraints from the individual parts; that is, it forms an upper bound in the lattice. Thus, the combination of an adapter with a lens is another lens, and the combination of a traversal with anything is again a traversal.
Profunctor Optics

The lattice structure becomes apparent only in the profunctor representation, because heterogeneous combinations are not otherwise expressible. The combination of a lens and a prism is a traversal; but in fact, the combination needs only the Cartesian and Cocartesian constraints of lenses and prisms respectively, and not the additional Monoidal constraint of traversals, so does not use the full power of a traversal (indeed, such a combination necessarily targets at most one component of a structure, and so there is no need for sequencing of effectful operations). This means that there is a fifth point in the lattice, the least upper bound of lenses and prisms but strictly below traversals. It is as yet unclear to us whether that fifth point is a useful abstraction in its own right, or a mere artifact of our representation; this question calls for further work.

Curiously, our presentation does not depend at all on the ‘well-behavedness’ laws of lenses or their duals for prisms, nor on the two functions making up an adapter being each other’s inverses, nor on the laws of traversals [12]. The proofs of equivalence (in Appendix C) do not use them; the abstractions accommodate ill-behaved optics just as well as they do well-behaved ones. Identifying suitable well-behavedness laws for profunctor optics is another topic for future research. We conjecture that addressing this question will entail consideration of sequential composition of profunctors (taking a $P A B$ and a $P B C$ to a $P A C$). That might have other benefits too; in particular, given composition, one can define $\text{par}$ in terms of $\text{first}$ and $\text{second}$, which should simplify the assumptions we make.

We have taken the opportunity to judiciously rename some abstractions from existing libraries, in the interests of tidying up. The names $\text{first}$, $\text{second}$, $\text{left}$, and $\text{right}$ were already popular in the Haskell Arrow library, so Kmett’s Profunctor library [18] uses $\text{first}'$, $\text{second}'$, $\text{left}'$, $\text{right}'$ instead. What we have called an Adapter is conventionally called an Iso, despite there being no requirement for it actually to form an isomorphism. Our classes Cartesian and Cocartesian are conventionally (and asymmetrically) called Strong and Choice. The conventional ordering of type parameters for optics, for example from the mezzolens library [27], would be to have $\text{Lens } a b s t$ and so on; we have used the ordering $\text{Lens } a b s t$ instead, so that we can conveniently apply the type constructor just to the first two, as required for the profunctor instances in the translation functions in Section 4. Our $\text{Traversal}$ is not quite the same as the $\text{traverse}$ method of the $\text{Traversable}$ type class in the Haskell libraries, because that type class insists that the container datatype is polymorphic; ours allows us for example to access a packed $\text{ByteString}$ as a container of eight-bit $\text{Word8s}$, or an integer as a container of digits—traversal still makes sense for monomorphic containers, it just cannot change the types of the elements.
Although we have presented our constructions using Haskell as a vehicle, they do not really depend in any essential way on Haskell. All that seems to be required are higher-order functions (‘lambdas’ or ‘closures’), parametrized types (‘generics’ or ‘abstract types’) of higher kind (in particular, parametrization by profunctors), and some mechanism for separating interfaces from implementations (‘abstract classes’ or ‘modules’). We feel that Haskell allows clear expression using those features—the algorithmic language de nos jours—but it is possible to replicate our constructions in other languages such as Scala; we sketch an implementation in Appendix B.

Acknowledgements Our primary thanks are due to the various people who have contributed to the library development and online discussions whose outcomes we have made use of in this paper, especially Edward Kmett, Russell O’Connor, Twan van Laarhoven, Reid Barton, Shachaf Ben-Kiki, and Elliott Hird. We are grateful to Paul Levy for suggesting the names ‘cartesianly strong’, ‘co-cartesianly strong’, and ‘monoidal’, and to Sam Staton, Guillaume Boisseau, and especially to James McKinna for many helpful comments and encouraging discussions. This paper is a condensed version of the first author’s undergraduate thesis [31]. The work has been partially supported by the UK EPSRC-funded project A Theory of Least Change for Bidirectional Transformations (EP/K020919/1).

References


Profunctor Optics


[10] Phil Freeman. purescript-profunctor-lenses-2.2.0 library. URL: https://pursuit.purescript.org/packages/purescript-profunctor-lenses/2.2.0.


[27] Russell O’Connor. *mezzolens-o.0.0 library*. URL: https://hackage.haskell.org/package/mezzolens-0.0.0.


Profunctor Optics


A Haskell background

We have used an idealization of Haskell [30] throughout as a lingua franca, although our constructions do not really depend on anything other than higher-order functions and some notion of interface and implementation. For the reader unfamiliar with Haskell, we summarize here some conventions and useful standard operations. We also explain our idealizations, so even those familiar with Haskell should skim this section.

Haskell is a functional programming language, so of course revolves around functions. The type $A \rightarrow B$ consists of functions that take an argument of type $A$ and return a result of type $B$; for example, the function even of type $\text{Integer} \rightarrow \text{Bool}$ determines whether or not an Integer is even. Type declarations are written with a double colon:

\[
even :: \text{Integer} \rightarrow \text{Bool}
\]

\[
even n = (\mod n 2 == 0)
\]

The identity function is written $\text{id}$, and function composition with a centred dot, so that $(f \cdot g) x = f(g x)$.

By convention, functions are curried wherever possible; for example, rather than defining the modulus function to accept a pair of arguments:

\[
\text{mod} :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}
\]

we make it take two separate arguments:

\[
\text{mod} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}
\]

or equivalently, to take one argument and yield a function of the other:

\[
\text{mod} :: \text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})
\]

Functions may be polymorphic; so identity and composition have the following types:

\[
id :: a \rightarrow a
\]

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)
\]

Type variables (such as $a$ above) are written with a lowercase initial letter, and are implicitly universally quantified; specific types (such as $\text{Integer}$) are written with an uppercase initial letter. We therefore use uppercase identifiers (such as $A, B, C$) for specific types in examples in prose.

Pairs are written in parentheses, both as values and as types; for example, $(3, \text{True}) :: (\text{Integer}, \text{Bool})$. However, we have used $\times$ for pair types in the paper, writing $\text{Integer} \times \text{Bool}$ for $(\text{Integer}, \text{Bool})$. Two functions with a common source type may be combined to yield a pair of results, via the function fork:

\[
\text{fork} :: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b,c)
\]

\[
\text{fork } f \ g \ x = (f \ x, g \ x)
\]
and two functions may act in parallel on the components of a pair, via the function \( \text{cross} \):

\[
\text{cross} :: (a \to a') \to (b \to b') \to (a, b) \to (a', b')
\]

\[
\text{cross } f \ g (x, y) = (f \ x, g \ y)
\]

Since pairs have projections \( \text{fst} :: (a, b) \to a \) and \( \text{snd} :: (a, b) \to b \), the \( \text{cross} \) operation is in fact an instance of \( \text{fork} \):

\[
\text{cross } f \ g = \text{fork} (f \cdot \text{fst}) (g \cdot \text{snd})
\]

Dually, \( \text{sum} \) types correspond to 'variant' types in Pascal or unions in C. In Haskell, they are written with an algebraic datatype:

```
data Either a b = Left a | Right b
```

(In the paper, we have used the idealized notation \( A + B \) for \( \text{Either } A \ B \).) This declaration introduces a new two-parameter (polymorphic) datatype \( \text{Either} \), and two (polymorphic) constructors \( \text{Left} :: a \to \text{Either } a \ b \) and \( \text{Right} :: b \to \text{Either } a \ b \); so values of type \( \text{Either } A \ B \) are either of the form \( \text{Left } x \) with \( x :: A \), or of the form \( \text{Right } y \) with \( y :: B \). Two functions with a common target type may be combined to act on a \( \text{sum} \) type, via the function \( \text{either} \):

\[
\text{either} :: (a \to c) \to (b \to c) \to \text{Either } a \ b \to c
\]

\[
\text{either } f \ g (\text{Left } x) = f \ x
\]

\[
\text{either } f \ g (\text{Right } y) = g \ y
\]

As an instance of this, we can act independently on the two variants of a \( \text{sum} \) type, via the function \( \text{plus} \):

\[
\text{plus} :: (a \to a') \to (b \to b') \to \text{Either } a \ b \to \text{Either } a' \ b'
\]

\[
\text{plus } f \ g = \text{either } (f \cdot \text{Left}) (g \cdot \text{Right})
\]

A useful special case of \( \text{sum} \) types is \( \text{sum} \) with the unit type, which provides a representation of \text{optional} values:

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

It is sometimes convenient to model a datatype as a \text{record} with named fields; for example, had pairs not been built in, we might have defined instead

```
data Pair a b = MkPair {fst :: a, snd :: b}
```

This declaration introduces a new two-parameter datatype \( \text{Pair} \), a constructor \( \text{MkPair} :: a \to b \to \text{Pair } a \ b \), and two field extractors \( \text{fst} :: \text{Pair } a \ b \to a \) and \( \text{snd} :: \text{Pair } a \ b \to b \). The namespaces for types (such as \( \text{Pair} \)) and values (such as \( \text{MkPair} \)) are disjoint, and it is idiomatic Haskell to name the constructor \( \text{Pair} \) rather than \( \text{MkPair} \).

Algebraic datatypes find most use for recursive datatypes. For example, a polymorphic datatype of internally labelled binary trees may be defined by
data Tree a = Empty | Node (Tree a) a (Tree a)

which has a constant Empty representing the empty tree, and a ternary constructor
Node assembling a non-empty tree from two children and a root label. Haskell provides
a built-in polymorphic datatype [a] of lists; but if that had not been provided, we
could have defined instead

data List a = Nil | Cons a (List a)

Interfaces and implementations are typically modelled in Haskell by means of type
classes. A type class represents a set of types, characterized by their common support
for a particular collection of methods. For example, the standard type classes Eq and
Ord denote the classes of types that support equality and ordering functions:

class Eq a where
  (==) :: a → a → Bool

class Eq a ⇒ Ord a where
  (≤) :: a → a → Bool

This declaration states that a type A is a member of the type class Eq if it supports
an equality test (==) :: A → A → Bool; and an Eq type A is a fortiori an Ord type if
it also supports the comparison (≤) :: A → A → Bool. Most of the standard types are
instances of these classes. New instances may be declared by providing a definition of
the necessary methods; for example, we could specify equality on optional values,
given equality on their elements, by:

instance Eq a ⇒ Eq (Maybe a) where
  Just x == Just y = (x == y)
  Nothing == Nothing = True
  _ == _ = False

(where the underscore is a wild-card pattern). To be precise, one typically declares
laws that instance methods should satisfy (such as being an equivalence relation for
Eq, and a preorder for Ord); but Haskell provides no way to state such laws other than
in comments. One can think of the type class as representing an interface, and the
instances as implementations of that interface.

Type classes may be used for type constructors as well as for concrete types. For ex-
ample, we make extensive use of the type class Functor, which represents polymorphic
container types. Informally, the type constructor F represents a polymorphic container
if a value of type FA contains elements of type A, on which one may operate with a
function of type A → B, to yield a new value of type F B. Formally:

class Functor f where
  fmap :: (a → b) → f a → f b

For example, the Maybe type constructor is a functor:

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
which should satisfy the following two laws:

\[
\begin{align*}
\text{fmap (f \cdot g)} &= \text{fmap f \cdot fmap g} \\
\text{fmap id} &= \text{id}
\end{align*}
\]

An important result about polymorphic functions between functors is known colloquially as *theorems for free* [40]. This result entails that a function \( h \) of type \( \forall a . F a \to G a \), for particular \( F \) and \( G \) that are instances of \textit{Functor}, but crucially for all values of the type parameter \( a \), satisfies the property

\[
\text{h \cdot fmap}_F f = \text{fmap}_G f \cdot h
\]

where the occurrences of \( \text{fmap} \) have been annotated to indicate that on the left of the equation it is the instance for \( F \) and on the right it is the one for \( G \), as may be easily determined from the type of \( h \). We call this property ‘the free theorem of the type of \( h \)’, or sometimes just ‘the free theorem of \( h \)’ when we have a particular definition of \( h \) in mind. The point is that it does not matter what that definition is, provided that it has the stated type. For example, consider a function \( h :: [a] \to \text{Maybe} \, a \). It does not matter whether this returns the head of the list and \text{Nothing} for the empty list, or the last element of the list similarly, or the fourth element when the list has odd length at least 13 and \text{Nothing} for even-length and shorter lists; it still necessarily satisfies the same property. (To be precise, the theorem may need a side condition when \( h \) is allowed to be a partial function; but in this paper, we restrict attention to total functions.)

Free theorems also apply to types involving constraints [38], essentially by following the translation from type classes to dictionary-passing style—the functions being mapped must preserve the methods of the class. For example, the function \( \text{nub} :: \text{Eq} \, a \Rightarrow [a] \to [a] \) removes duplicates from a list. The free theorem for a function \( h \) of type \( [a] \to [a] \) would state that \( h \cdot \text{fmap}_F f = \text{fmap}_F f \cdot h \) for any \( f \), which is plainly false for \( h = \text{nub} \). Of course, \( \text{nub} \) has a more constrained than \( h \); the expected theorem restricts the claim to those \( f \) that preserve equality—that is, such that \( (f \, x = f \, y) \Leftrightarrow (x = y) \). This is in fact precisely the theorem that arises for free from the type of the function \( \text{nubBy} :: (a \to a \to \text{Bool}) \to [a] \to [a] \), which is essentially the dictionary-passing translation of \( \text{nub} \).

**B Alternative implementation**

In this appendix, we sketch a parallel implementation in Scala of the concepts introduced in the paper. This exercise demonstrates that the ideas are not limited to Haskell. However, the constructions do require support for higher-kindred types (specifically, for instances of the \textit{Profunctor} class), which is not available in languages such as Java and C#.

We require Scala’s support for higher-kindred types:

\[
\text{import} \, \text{scala.language.higherKinds}
\]
Abstractions that are represented with type classes in Haskell are traditionally implemented using traits—a variant of classes allowing flexible mixin composition—in Scala. The following definition introduces the `Profunctor` abstraction, here a two-parameter type operation \( p \). For some \( p \) to instantiate the `Profunctor` abstraction, it must implement the `dimap` method. The type of `dimap` is parametrized by four type variables \( a, b, c, d \). The method itself takes two functions \( f \) and \( g \) of types \( c \Rightarrow a \) and \( b \Rightarrow d \) respectively, and an input transformation \( h : p[a, b] \), which it lifts to an output transformation of type \( p[c, d] \).

```
trait Profunctor[p[_, _]]{
  def dimap[a, b, c, d](f : c ⇒ a)(g : b ⇒ d)(h : p[a, b]) : p[c, d]
}
```

Similarly, the `Cartesian`, `Cocartesian`, and `Monoidal` specializations each involve implementing one or two methods. The Scala libraries provide a generic class `Tuple2` of pairs (in fact, the language also supports the Haskell syntax \((a, b)\) as a shorthand for `Tuple2[a, b]`), a generic class `Either` precisely matching Haskell’s `Either`, and a class `Unit` with a single inhabitant corresponding to Haskell’s \((\_ \_\_)\).

```
trait Cartesian[p[_, _]] extends Profunctor[p]{
  def first[a, b, c](h : p[a, b]) : p[Tuple2[a, c], Tuple2[b, c]]
}

trait Cocartesian[p[_, _]] extends Profunctor[p]{
  def right[a, b, c](h : p[a, b]) : p[Either[c, a], Either[c, b]]
}

trait Monoidal[p[_, _]] extends Profunctor[p]{
  def par[a, b, c, d](h : p[a, b])(k : p[c, d]) : p[Tuple2[a, c], Tuple2[b, d]]
  def empty : p[Unit, Unit]
}
```

The four varieties of optic are each represented as abstract classes, each with a single method called `apply` as the operative content. What is represented with a type class constraint in Haskell turns up as an `implicit` parameter on the `apply` methods in Scala; when there is a unique binding of the appropriate type in scope at the call site, the actual parameter may be omitted and this binding will be used in its place. In the case of `Traversals`, the implicit parameter is required to implement the mixin composition of three traits, analogous to having three type class constraints in Haskell.

```
abstract class Adapter[p[_, _], a, b, s, t]{
  def apply(h : p[a, b])(implicit prof : Profunctor[p]) : p[s, t]
}

abstract class Lens[p[_, _], a, b, s, t]{
  def apply(h : p[a, b])(implicit prof : Cartesian[p]) : p[s, t]
}

abstract class Prism[p[_, _], a, b, s, t]{
  def apply(h : p[a, b])(implicit prof : Cocartesian[p]) : p[s, t]
}
```
Profunctor Optics

}\nabstract class Traversal[p[_,_],a,b,s,t] {\n  def apply(h : p[a,b]) {\n    (implicit prof : Cartesian[p] with Cocartesian[p] with Monoidal[p]) : p[s,t] \n  }\n}\n
Here is a concrete class corresponding to the lens \( \pi_1 \) we used in the paper, for access to the left-hand component of a pair. It is defined by extending a suitable type instantiation of the \textit{Lens} trait, and providing an implementation of the \textit{apply} method. The three subsidiary definitions \textit{view}, \textit{update}, and \textit{fork} are local to \textit{apply}; the final line of the definition of \textit{apply} is the actual body. The value is obtained by applying of \textit{dimap} to \textit{first}, both of which are methods of the \textit{Cartesian} profunctor \textit{prof}.

\begin{verbatim}
class PiOne[p[_,_],a,b,c] extends Lens[p,a,b,Tuple2[a,c],Tuple2[b,c]] {
  def apply(f : p[a,b])(implicit prof : Cartesian[p]) : p[Tuple2[a,c],Tuple2[b,c]] = {
    def view[a,c] : (Tuple2[a,c] ⇒ a) = (xy ⇒ xy._1);
    def update[a,b,c] : (Tuple2[b,b,Tuple2[a,c]] ⇒ Tuple2[b,c]) = (zxy ⇒ newTuple2(zxy._1, (zxy._2 - 2));
    def fork[a,b,c] (f : a ⇒ b)(g : a ⇒ c) : (a ⇒ Tuple2[b,c]) = (x ⇒ newTuple2(f(x), g(x));
    prof.dimap(fork(view[a,c])(identity))(update[a,b,c])(prof.first(f))
  }
}
\end{verbatim}

Similarly, here is a class corresponding to the prism \textit{the} from the body of the paper, for access to the payload of an optional value. The Scala libraries provide a class \textit{Option} corresponding to Haskell’s \textit{Maybe} type constructor. The local functions \textit{match} and \textit{either} are defined using case analysis over one argument. (Scala uses \texttt{match} as a keyword, so we have taken a liberty in using that word as an identifier.)

\begin{verbatim}
class The[p[_,_],a,b] extends Prism[p,a,b,Option[a],Option[b]] {
  def apply(f : p[a,b])(implicit prof : Cocartesian[p]) : p[Option[a],Option[b]] = {
    def match[a,b] : (Option[a] ⇒ Either[Option[b],a]) = {
      case Some(a) ⇒ Right(a)
      case None ⇒ Left(None)
    };
    def build[b] : (b ⇒ Option[b]) = (b ⇒ Some(b));
    def either[a,b,c](f : a ⇒ c)(g : b ⇒ c) : (Either[a,b] ⇒ c) = {
      case Left(a) ⇒ f(a)
      case Right(b) ⇒ g(b)
    };
    prof.dimap(match[a,b])(either(identity[Option[b]])(build[b]))(prof.right(f))
  }
}\n\end{verbatim}
As well as Haskell, there are implementations of variations of these ideas in Javascript [1, 7] and Purescript [10]. There is a Scala library cats that implements profunctors but has not been extended to optics [6], and a Scala library monocle of optics that does not use profunctors [35]; there are some initial experiments towards a Scala implementation of profunctor optics [36] that does not yet seem to be complete. There is also a Java library called Functional Java [13] implementing optics without using profunctors, so without their modularity benefits.

C. Proofs of equivalence

We formalize here the statements made earlier about the equivalences between the concrete and profunctor representations of the various kinds of optic, and provide proofs of those equivalences.

Theorem 1. The functions adapterC2P and adapterP2C are each other’s inverses, and so the types Adapter A B S T and AdapterP A B S T are isomorphic for all type parameters A, B, S, T. □

Theorem 2. The functions lensC2P and lensP2C are each other’s inverses, and so the types Lens A B S T and LensP A B S T are isomorphic for all type parameters A, B, S, T. □

Theorem 3. The functions prismC2P and prismP2C are each other’s inverses, and so the types Prism A B S T and PrismP A B S T are isomorphic for all type parameters A, B, S, T. □

Theorem 4. The functions traversalC2P and traversalP2C are each other’s inverses, and so the types Traversal A B S T and TraversalP A B S T are isomorphic for all type parameters A, B, S, T. □

C.1 Adapters

One of the key ingredients in the proofs of the theorems is the notion of profunctor morphism.

Definition 5. A polymorphic function \( \phi : \forall a b. P a b \to Q a b \), for given profunctors \( P, Q \) but for all types \( a, b \), is a ‘profunctor morphism from \( P \) to \( Q \)’ if

\[
\text{dimap}_Q f g \cdot \phi = \phi \cdot \text{dimap}_P f g
\]

for all functions \( f, g \), where we have annotated the two occurrences of \( \text{dimap} \) to indicate which instances they are. □

In particular, the translation function \( \text{adapterC2P} \) from Section 4.1, when applied to a given transformer of type \( P A B \) as its second argument, yields a profunctor morphism from \( \text{Adapter} A B \) to \( P \). To make this precise, consider

\[
\text{flip adapterC2P} :: \text{Profunctor} p \Rightarrow p a b \to \text{Adapter} a b s t \to p s t
\]
Profunctor Optics

which is like $adapterC2P$ but takes its arguments in a different order, since $flip \, f \, x \, y = f \, y \, x$.

Then we may state:

Lemma 6. For given $k :: P \, A \, B$ for some profunctor $P$ and types $A,B$, the function $flip \, adapterC2P \, k :: Adapter \, A \, B \, s \, t \rightarrow P \, s \, t$ is a profunctor morphism from $Adapter \, A \, B$ to $P$.

Proof. We have:

\[
\begin{align*}
\dimap \, f \, g \, (flip \, adapterC2P \, k \, (\text{Adapter } o \, i)) &= \left[ \left[ \text{flip} \right] \right] \\
\dimap \, f \, g \, (adapterC2P \, (\text{Adapter } o \, i) \, k) &= \left[ \left[ adapterC2P \right] \right] \\
\dimap \, f \, g \, (\dimap \, o \, i \, k) &= \left[ \left[ \text{dimap composition} \right] \right] \\
\dimap \, (o \cdot f) \, (g \cdot i) \, k &= \left[ \left[ adapterC2P \right] \right] \\
adapterC2P \, (\text{Adapter } (o \cdot f) \, (g \cdot i)) \, k &= \left[ \left[ \text{dimap for Adapter} \right] \right] \\
adapterC2P \, (\dimap \, f \, g \, (\text{Adapter } o \, i)) \, k &= \left[ \left[ \text{flip} \right] \right] \\
\text{flip} \, adapterC2P \, k \, (\dimap \, f \, g \, (\text{Adapter } o \, i))
\end{align*}
\]

and so $\dimap \, f \, g \cdot flip \, adapterC2P \, k \, = \, flip \, adapterC2P \, k \cdot \dimap \, f \, g$ as required.

The second key ingredient is the free theorem of profunctor optics, obtained by instantiating Wadler’s ‘theorems for free’ at the appropriate type [38, 40]:

Lemma 7. For any $l$ of type $\forall \, p \cdot \text{Profunctor } p \Rightarrow \text{Optic } p \, A \, B \, S \, T$ and any profunctor morphism $\phi$ from profunctor $P$ to profunctor $Q$, we have $l \cdot \phi = \phi \cdot l$.

Both sides are of type $P \, A \, B \rightarrow Q \, S \, T$; the free theorem states that lifting the $P \, A \, B$ to $P \, S \, T$ and then translating to $Q \, S \, T$ coincides with first translating to $Q \, A \, B$ and then lifting to $Q \, S \, T$.

Proof (of Theorem 1). One direction is quite straightforward:

\[
\begin{align*}
\text{adapterP2C} \, (\text{adapterC2P} \, (\text{Adapter } o \, i)) &= \left[ \left[ \text{adapterC2P} \right] \right] \\
\text{adapterP2C} \, (\dimap \, o \, i) &= \left[ \left[ \text{adapterP2C} \right] \right] \\
\dimap \, o \, i \, (\text{Adapter } id \, id) &= \left[ \left[ \text{dimap for Adapter} \right] \right] \\
\text{Adapter} \, (\text{id} \cdot o) \, (i \cdot id) &= \left[ \left[ \text{identity} \right] \right] \\
\text{Adapter} \, o \, i
\end{align*}
\]
as required. For the other direction, we need to use Lemma 6:

\[
\text{adapterC2P (adapterP2C \ l) \ k}
\]

= \[
\text{adapterP2C (l (Adapter id id)) \ k}
\]

= \[
\text{flip adapterC2P k (l (Adapter id id))}
\]

= \[
\text{free theorem of l, and Lemma /six.oldstyle}
\]

\[
l (\text{flip adapterC2P k (Adapter id id))}
\]

= \[
\text{flip adapterC2P (Adapter id id) \ k}
\]

= \[
\text{adapterC2P (dimap id id k)}
\]

= \[
\text{dimap identity}
\]

\[
l k
\]

so \(\text{adapterC2P} \cdot \text{adapterP2C} = \text{id}\) as required.

\[\heartsuit\]

C.2 Lenses

Lemma 8. The ‘free theorem’ \([40]\) of first is that

\[
dimap id h k = dimap g id l \Rightarrow dimap (\cross (h,f)) (\first k) = dimap (\cross (g,f)) \text{id} (\first l)
\]

\[\square\]

Lemma 9. For given \(k :: P A B\) for some cartesian profunctor \(P\) and types \(A,B\), the function \(\text{flip lensC2P} k :: \text{Lens } A B s t \rightarrow P s t\) is a profunctor morphism from \(\text{Lens } A B\) to \(P\).

Proof. We have:

\[
dimap f g (\text{flip lensC2P} k (\text{Lens } v u))
\]

= \[
\text{flip}
\]

\[
dimap f g (\text{lensC2P} (\text{Lens } v u) \ k)
\]

= \[
\text{dimap composition}
\]

\[
dimap (\text{fork} (v, id)) u (\first k)
\]

= \[
\text{products and fork}
\]

\[
dimap (\cross (id,f) \cdot \text{fork} (v \cdot f, id)) (g \cdot u) (\first k)
\]

= \[
\text{dimap composition}
\]

\[
dimap (\text{fork} (v \cdot f, id)) (g \cdot u) (\dimap \text{id} (\cross (id,f)) (\first k))
\]

= \[
\text{free theorem of first (Lemma 8), with } g = \text{id}, h = \text{id}, \text{ and } k = l
\]

\[
dimap (\text{fork} (v \cdot f, id)) (g \cdot u) (\dimap \text{cross} (id,f)) (\first k)
\]

= \[
\text{dimap composition}
\]

\[
dimap (\text{fork} (v \cdot f, id)) (g \cdot u \cdot \cross (id,f)) (\first k)
\]
Profunctor Optics

\[ \text{lensC2P} (\text{Lens} (v \cdot f) (g \cdot u \cdot \text{cross} (id, f))) \]

\[ \text{dimap for Lens} \]

\[ \text{lensC2P} (\text{dimap} f g (\text{Lens} v u)) \]

\[ \text{flip} \]

\[ \text{flip lensC2P} k (\text{dimap} f g (\text{Lens} v u)) \]

So \( \text{dimap} f g \cdot \text{flip lensC2P} k = \text{flip lensC2P} k \cdot \text{dimap} f g \) as required.

We also need the following observation, due to O’Connor [28].

**Lemma 10.**

\[ \text{dimap} (\text{fork} (id, id)) \text{fst} \cdot \text{first} = \text{id} \]

**Proof.** We have:

\[ \text{dimap} (\text{fork} (id, id)) \text{fst} \cdot \text{first} \]

\[ \text{dimap composition} \]

\[ \text{dimap} (\text{fork} (id, id)) \text{fst} \cdot \text{dimap id} (\text{cross} (id, \text{const} ())) \cdot \text{first} \]

\[ \text{free theorem of first} \]

\[ \text{dimap} (\text{fork} (id, id)) \text{fst} \cdot \text{dimap} (\text{cross} (id, \text{const} ())) \text{id} \cdot \text{first} \]

\[ \text{dimap composition} \]

\[ \text{dimap} (\text{cross} (id, \text{const} ()) \cdot \text{fork} (id, id)) \text{fst} \cdot \text{first} \]

\[ \text{products} \]

\[ \text{dimap} ((), ()) \text{fst} \cdot \text{first} \]

\[ \text{coherence of first with unit type: dimap} \text{fst} ((),()) = \text{first} \]

\[ \text{dimap} ((), ()) \text{fst} \cdot \text{dimap} \text{fst} ((),()) \]

\[ \text{dimap composition} \]

\[ \text{dimap} (\text{fst} \cdot ((),())) (\text{fst} \cdot ((),())) \]

\[ \text{products} \]

\[ \text{dimap id id} \]

\[ \text{dimap identity} \]

\[ \text{id} \]

as required.

**Proof (of Theorem 2).** As with adapters, one direction is quite straightforward:

\[ \text{lensP2C} (\text{lensC2P} (\text{Lens} v u)) \]

\[ \text{lensP2C} (\text{dimap} (\text{fork} (v, id)) u \cdot \text{first}) \]

\[ \text{dimap} (\text{fork} (v, id)) u \cdot \text{first} (\text{Lens} id \text{fst}) \]
For the other direction, we have:

\[ \text{lensC2P} (\text{lensP2C} \ l) \ k = \text{lensP2C} (\text{lensC2P} (\text{lensP2C} \ l) \ k) = \text{flip} \ \text{lensC2P} \ k \ (\text{lensP2C} \ l) = \text{flip} \ \text{lensC2P} \ (\text{lensP2C} \ l) \ k = \text{lensC2P} \ \text{lensC2P} \ (\text{lensP2C} \ l) \ k = \text{l} \ \text{lensC2P} \ (\text{lensP2C} \ l) \ k = \text{l} \ \text{lensC2P} \ \text{lensP2C} \ (\text{lensP2C} \ l) \ k = \text{l} \ k \]

so \( \text{lensC2P} \cdot \text{lensP2C} = \text{id} \) as required.

\section*{C.3 Prisms}

**Lemma 11.** The free theorem of right is that

\[ \text{dimap id } h \ k = \text{dimap g id } l \ \Rightarrow \ \text{dimap dimap} \ (\text{plus} \ (f, h)) \ (\text{right } k) = \text{dimap dimap} \ (\text{plus} \ (f, g)) \ (\text{right } l) \]

\[ \square \]

**Lemma 12.** For given \( k :: \text{P} \ A \ B \) for some co-cartesian profunctor \( P \) and types \( A, B \), the function \( \text{flip prismC2P} \ k :: \text{Prism} \ A \ B \ s \ t \rightarrow \text{P} \ s \ t \) is a profunctor morphism from \( \text{Prism} \ A \ B \) to \( P \).

\[ \square \]

**Proof.** We have:

\[ \text{dimap f g} \ \text{(flip prismC2P} \ k \ (\text{PrismC} \ m \ b)) = [\text{flip}] \ \text{dimap f g} \ \text{(prismC2P} \ (\text{PrismC} \ m \ b) \ k) = [\text{prismC2P}] \ \text{dimap f g} \ \text{(dimap m} \ (\text{either id} \ b) \ (\text{right k})) = [\text{dimap composition}] \]
**Profunctor Optics**

\[
\text{dimap } (m \cdot f) (g \cdot \text{either id } b) (\text{right } k) \\
= \begin{array}{l}
\text{sums and either } \\
\text{dimap composition} \\
\text{dimap } (m \cdot f) (\text{either id } (g \cdot b) \cdot \text{plus } g \cdot \text{id}) (\text{right } k) \\
= \begin{array}{l}
\text{free theorem of right (Lemma 11), with } g = \text{id}, h = \text{id}, \text{ and } k = l \\
\text{dimap composition} \\
\text{dimap } (m \cdot f) (\text{either id } (g \cdot b)) (\text{dimap } (\text{plus } g \cdot \text{id}) (\text{right } k)) \\
= \begin{array}{l}
\text{dimap composition} \\
\text{dimap } (\text{plus } g \cdot \text{id} \cdot m \cdot f) (\text{either id } (g \cdot b)) (\text{right } k) \\
= \begin{array}{l}
\text{prismC2P} \\
\text{prismC2P } (\text{PrismC } (\text{plus } g \cdot \text{id} \cdot m \cdot f) (g \cdot b)) \text{ } k \\
= \begin{array}{l}
\text{flip} \\
\text{flip prismC2P } k \text{ } (\text{dimap } f \text{ } g \text{ } (\text{PrismC } m \text{ } b)) \\
\text{So dimap } f \text{ } g \cdot \text{flip prismC2P } k = \text{flip prismC2P } k \cdot \text{dimap } f \text{ } g \text{ as required.}\end{array}
\end{array}
\end{array}
\]

We also need the following result:

**Lemma 13.**

\[
\text{dimap } \text{Right } (\text{either id id}) \cdot \text{right} = \text{id}
\]

\[\square\]

**Proof.** Writing \(\text{absurd} :: 0 \rightarrow a\) for the unique function from the empty type to any other, we have:

\[
\text{dimap } \text{Right } (\text{either id id}) \cdot \text{right} \\
= \begin{array}{l}
\text{sums} \\
\text{dimap } \text{composition} \\
\text{dimap } (\text{plus } \text{absurd } \text{id} \cdot \text{Right}) (\text{either id id}) \cdot \text{right} \\
= \begin{array}{l}
\text{dimap } \text{composition} \\
\text{coherence of right with empty type: dimap } (\text{either absurd } \text{id}) \text{Right} = \text{right} \\
\text{dimap } (\text{either id id} \cdot \text{plus absurd id} \cdot \text{Right}) \\
= \begin{array}{l}
\text{sums} \\
\text{dimap id id} \\
\text{id} \\
\end{array}
\end{array}
\end{array}
\]

as required.
Proof (of Theorem 3). As with adapters, one direction is quite straightforward:

\[
\begin{align*}
prismP2C (prismC2P (PrismC m b)) &= \ \ \\
&= \ [ \ [ prismP2C ] ] \\
prismP2C (dimap m (either id b) \cdot right) &= \ [ \ [ prismP2C ] ] \\
(dimap m (either id b) \cdot right) (PrismC Right id) &= \ [ \ [ \text{composition} ] ] \\
dimap m (either id b) (right (PrismC Right id)) &= \ [ \ [ \text{right for PrismC} ] ] \\
dimap m (either id b) (PrismC (either (Left \cdot Left) (plus Right id \cdot Right)) Right) &= \ [ \ [ \text{dimap for PrismC} ] ] \\
PrismC (plus (either id b) \cdot \cdot \cdot (plus Right id \cdot Right) \cdot m) (either id b \cdot Right) &= \ [ \ [ \text{sums} ] ] \\
PrismC m b
\end{align*}
\]

For the other direction, we have:

\[
\begin{align*}
prismC2P (prismP2C l) k &= \ \ \\
&= \ [ \ [ prismP2C ] ] \\
prismC2P (l (PrismC Right id)) k &= \ [ \ [ \text{flip} ] ] \\
flip prismC2P k (l (PrismC Right id)) &= \ [ \ [ \text{free theorem of l, and Lemma 12} ] ] \\
l (flip prismC2P k (PrismC Right id)) &= \ [ \ [ \text{flip} ] ] \\
l (prismC2P (PrismC Right id) k) &= \ [ \ [ \text{Lemma 13} ] ] \\
l (dimap Right (either id id) (right k)) &= l k
\end{align*}
\]

so \( prismC2P \cdot prismP2C = id \) as required.

\[\heartsuit\]

C.4 Traversals

This proof depends on the fact that \textit{FunLists} are traversable:

\[
\begin{align*}
\text{travFunList} :: \text{Applicative } f \Rightarrow (a \rightarrow f b) \rightarrow \text{FunList } a c t \rightarrow f (\text{FunList } b c t) \\
\text{travFunList } f \text{ (Done } t) &= \text{pure } (\text{Done } t) \\
\text{travFunList } f \text{ (More } x \text{ l) } &= \text{pure } \langle*\rangle f x \langle*\rangle \text{ travFunList } f \text{ l}
\end{align*}
\]

Indeed, the following lemma shows that \textit{FunLists} are in a sense the archetypical traversable containers: an application of \textit{traverse} in the \textit{FunList} applicative functor can be expressed in terms of \textit{travFunList}.

Lemma 14. For any \( h :: A \rightarrow \text{FunList } B C D \), we have
**Profunctor Optics**

\[
\text{traverse} (\text{Traversal } h) = \text{Traversal} (\text{travFunList } h)
\]

**Proof.** We prove this by structural induction over \textit{FunLists}. Both cases start in the same way, so we capture the common reasoning first. Suppose that \( h : S \to \text{FunList } A B T \), and let \( k = \text{extract} (\text{traverse} (\text{Traversal } h)) \) so that \( \text{traverse} (\text{Traversal } h) = \text{Traversal } k \). Then

\[
\text{traverse} (\text{Traversal } h) = \text{dimap out inn (right (\text{par} (\text{Traversal } h) (\text{Traversal } k))))} = \text{dimap out inn (right (\text{Traversal} (\text{pair } h k))))} = \text{dimap for Traversal} \]

\[
\text{Traversal} (\text{fmap inn \cdot either (Done \cdot Left) (fmap Right \cdot pair h k) \cdot out})
\]

so

\[
\text{extract} (\text{traverse} (\text{Traversal } h)) = \text{fmap inn \cdot either (Done \cdot Left) (fmap Right \cdot pair h k) \cdot out}
\]

We now prove that

\[
\text{extract} (\text{traverse} (\text{Traversal } h)) = \text{travFunList } h
\]

by structural induction over the \textit{FunList} argument. Again, let \( k = \text{extract} (\text{traverse} (\text{Traversal } h)) \).

For the base case \textit{Done } t, we have:

\[
\text{extract} (\text{traverse} (\text{Traversal } h)) (\text{Done } t) = \text{fmap inn \cdot either (Done \cdot Left) (fmap Right \cdot pair h k) \cdot out (Done t)})
\]

\[
= \text{fmap inn \cdot either (Done \cdot Left) (fmap Right \cdot pair h k) (Left t))}
\]

\[
= \text{fmap inn (Done (Left t))}
\]

\[
= \text{fmap for FunList \text{ Done (Left t))}
\]

\[
= \text{out \text{ Done (Done t) }}
\]

\[
= \text{pure for FunList \text{ pure (Done t) }}
\]

\[
= \text{travFunList \text{ travFunList } h (Done t)}
\]

For the inductive step \textit{More } x \ l, we assume the inductive hypothesis

\[
\text{extract} (\text{traverse} (\text{Traversal } h)) l = \text{travFunList } h l
\]
and then calculate:

\[
\begin{align*}
ex (\text{traverse} (\text{Traversal} \ h)) (\text{More} \ x \ l) \\
= & \left[ \left[ \text{first steps (above)} \right] \right] \\
\text{fmap inn} \ (\text{either} (\text{Done} \cdot \text{Left}) \ (\text{fmap Right} \cdot \text{pair} \ h \ k) \ (\text{out} \ (\text{More} \ x \ l))) \\
= & \left[ \left[ \text{out} \right] \right] \\
\text{fmap inn} \ (\text{either} (\text{Done} \cdot \text{Left}) \ (\text{fmap Right} \cdot \text{pair} \ h \ k) \ (\text{Right} \ (x, l))) \\
= & \left[ \left[ \text{sums} \right] \right] \\
\text{fmap inn} \ (\text{fmap Right} \ (\text{pair} \ h \ k \ (x, l))) \\
= & \left[ \left[ \text{pair} \right] \right] \\
\text{fmap inn} \ (\text{fmap Right} \ ((\text{pure} \ (\cdot \cdot \cdot \ h \ x \ (\cdot \cdot \cdot \ k \ l)))) \\
= & \left[ \left[ \text{inn} \right] \right] \\
\text{pure More} \ (\cdot \cdot \cdot \ h \ x \ (\cdot \cdot \cdot \ k \ l) \\
= & \left[ \left[ \text{inductive hypothesis} \right] \right] \\
\text{pure More} \ (\cdot \cdot \cdot \ h \ x \ (\cdot \cdot \cdot \ \text{travFunList} \ h \ l) \\
= & \left[ \left[ \text{travFunList} \right] \right] \\
\text{travFunList} \ h \ (\text{More} \ x \ l)
\end{align*}
\]

which completes the proof.

Next, we show that traversal of the empty FunList is essentially the identity transformer, constructed from empty using first and the left unit isomorphism for products:

\[
\text{id} :: \text{FunList} \\
\text{id} = \text{dimap} \ \text{lunit} \ \text{lunit} \ (\text{first empty})
\]

Lemma 15.

\[
\text{dimap} \ (\text{const} \ (\text{Done} \ t)) \ \text{id} \ (\text{traverse} \ k) = \text{dimap} \ \text{id} \ (\text{const} \ (\text{Done} \ t)) \ \text{id}
\]

(where \text{const} :: a \rightarrow b \rightarrow a yields a constant function).

Proof. We have:

\[
\begin{align*}
\text{dimap} \ (\text{const} \ (\text{Done} \ t)) \ \text{id} \ (\text{traverse} \ k) \\
= & \left[ \left[ \text{traverse} \right] \right] \\
\text{dimap} \ (\text{const} \ (\text{Done} \ t)) \ \text{id} \ (\text{dimap} \ \text{out} \ \text{inn} \ (\text{right} \ (\text{par} \ k \ (\text{traverse} \ k)))) \\
= & \left[ \left[ \text{dimap composition} \right] \right] \\
\text{dimap} \ \text{id} \ \text{inn} \ (\text{dimap} \ (\text{out} \cdot \text{const} \ (\text{Done} \ t)) \ \text{id} \ (\text{right} \ (\text{par} \ k \ (\text{traverse} \ k)))) \\
= & \left[ \left[ \text{out} \ (\text{Done} \ t) = \text{Left} \ t \right] \right] \\
\text{dimap} \ \text{id} \ \text{inn} \ (\text{dimap} \ (\text{const} \ (\text{Left} \ t)) \ \text{id} \ (\text{right} \ (\text{par} \ k \ (\text{traverse} \ k)))) \\
= & \left[ \left[ \text{property of right: dimap} \ \text{Left} \ \text{id} \ (\text{right} \ f) = \text{dimap} \ \text{id} \ \text{Left} \ \text{id} \ \text{identity} \right] \right] \\
\text{dimap} \ \text{id} \ \text{inn} \ (\text{dimap} \ (\text{const} \ t) \ \text{id} \ (\text{dimap} \ \text{id} \ \text{Left} \ \text{id} \ \text{identity})) \\
= & \left[ \left[ \text{dimap composition} \right] \right] \\
\text{dimap} \ (\text{const} \ t) \ (\text{inn} \cdot \text{Left}) \ \text{id} \ \text{identity} \\
= & \left[ \left[ \text{free theorem of identity: dimap} \ f \ \text{id} \ \text{id} \ = \ \text{dimap} \ \text{id} \ f \ \text{id} \ \text{identity} \right] \right] \\
\text{dimap} \ \text{id} \ (\text{inn} \cdot \text{Left} \cdot \text{const} \ t) \ \text{id} \ \text{identity}
\end{align*}
\]
Profunctor Optics

\[
\dimap \text{id} (\text{const} \ (\text{Done} \ t)) \ \text{id}
\]
as required.

Similarly, traversal of a singleton \text{FunList} is essentially a single application of the traversal body.

**Lemma 16.** The free theorem of \text{par} is that

\[
\dimap (\text{cross} \ (f, f')) (\text{cross} \ (g, g')) (\text{par} \ h \ h') = \text{par} (\dimap g \ h) (\dimap f' \ g' \ h')
\]

\[\square\]

**Lemma 17.**

\[
\dimap \text{single} \ \text{id} \ (\text{traverse} \ k) = \dimap \text{id} \ \text{single} \ k
\]

\[\square\]

**Proof.** We have:

\[
dimap \text{single} \ \text{id} \ (\text{traverse} \ k)
\]
\[
= \begin{array}{l}
\text{traverse} \\
\text{dimap composition}
\end{array}
\]
\[
dimap \text{id} \ \text{inn} \ (\text{dimap \ out \ \text{inn} \ \text{right} \ \text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{dimap composition}
\end{array}
\]
\[
dimap \text{id} \ \text{inn} \ (\text{dimap \ \text{out} \ \text{single} \ \text{id} \ \text{right} \ \text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{out} \ \text{(single} \ x = \ \text{Right} \ (x, \ \text{Done} \ id))
\end{array}
\]
\[
dimap \text{id} \ \text{inn} \ (\text{dimap} \ \lambda x \ \rightarrow \ \text{Right} \ (x, \ \text{Done} \ id)) \ \text{id} \ \text{right} \ \text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{dimap \ composition}
\end{array}
\]
\[
dimap \text{id} \ \text{inn} \ (\text{dimap} \ \\lambda x \ \rightarrow \ \text{Right} \ (x, \ \text{Done} \ id)) \ \text{(either} \ \text{id} \ \text{right} \ \text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{dimap \ composition}
\end{array}
\]
\[
dimap \text{id} \ \text{inn} \ (\text{dimap} \ , \ \text{Done} \ id) \ \text{Right} \ (\text{dimap} \ \text{Right} \ (\text{either} \ \text{id}) \ (\text{right} \ \text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{Lemma} \ 13
\end{array}
\]
\[
dimap \text{id} \ \text{inn} \ (\text{dimap} \ , \ \text{Done} \ id) \ \text{Right} \ (\text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{dimap \ composition}
\end{array}
\]
\[
dimap \ (\text{par} \ k (\text{traverse} \ k))
\]
\[
= \begin{array}{l}
\text{products, dimap \ composition}
\end{array}
\]
\[
dimap \ \text{runUnit} \ \text{(inn} \ \text{Right}) \ (\text{dimap} \ \text{cross} \ (\text{id}, \ \text{const} \ (\text{Done} \ id))) \ \text{id} \ \text{par} \ k \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{free \ theorem \ of \ par \ (Lemma \ 16)}
\end{array}
\]
\[
dimap \ \text{runUnit} \ \text{(inn} \ \text{Right}) \ (\text{par} \ k (\text{dimap} \ \text{const} \ (\text{Done} \ id))) \ \text{id} \ (\text{traverse} \ k)))
\]
\[
= \begin{array}{l}
\text{Lemma} \ 15
\end{array}
\]
\[
dimap \ \text{runUnit} \ \text{(inn} \ \text{Right}) \ (\text{par} \ k (\text{dimap} \ \text{const} \ (\text{Done} \ id))) \ \text{identity})
\]
\[
= \begin{array}{l}
\text{free \ theorem \ of \ par \ (Lemma \ 16) \ again}
\end{array}
\]
\[
dimap \ \text{runUnit} \ \text{(inn} \ \text{Right}) \ (\text{dimap} \ (\text{cross} \ (\text{id}, \ \text{const} \ (\text{Done} \ id))) \ \text{(par} \ k \ \text{identity}))
\]
\[
= \begin{array}{l}
\text{dimap \ composition}
\end{array}
\]
\[
dimap \ (\text{inn} \ \text{Right} \ \text{cross} \ (\text{id}, \ \text{const} \ (\text{Done} \ id))) \ (\text{dimap} \ \text{runUnit} \ \text{id} \ (\text{par} \ k \ \text{identity}))
\]
\[
= \begin{array}{l}
\text{identity} :: \text{P} \ 1 \ 1 \ = \ \text{empty} \ \text{see \ below}
\end{array}
\]

\[7-46\]
dimap id (inn · Right · cross (id, const (Done id))) (dimap runit’ id (par k empty))
= (par and empty)
dimap id (inn · Right · cross (id, const (Done id))) (dimap id runit’ k)
= (dimap composition)
dimap id (inn · Right · cross (id, const (Done id)) · runit’) k
= (inn · Right · cross (id, const (Done id)) · runit’ = single)
dimap id single k

Here, we used the fact that identity at the unit type is simply empty:

\[ \text{identity}_1 = \text{dimap identity} \]
\[ \text{dimap lunit’ lunit (first}_1 \text{ empty)} \]
\[ \text{dimap lunit’ lunit (dimap runit runit’ empty)} \]
\[ \text{dimap composition} \]
\[ \text{dimap (lunit’ · runit) (lunit · runit’)} \text{ empty} \]
\[ \text{dimap id id empty} \]
\[ \text{dimap identity} \]

\[ \text{empty} \]

where we have written identity$_1$ for identity :: P 1 1 and first$_1$ for first :: P A B \(\rightarrow\) P (A \times 1) (B \times 1) as mnemonics for their more specialized types.

The next lemma states that traversal of a FunList using a body that constructs a singleton makes a structure of singletons, and unpacking each of these singletons yields the original FunList. To prove this, we need the free theorem of \(\langle*\rangle\):

Lemma 18. The free theorem of \(\langle*\rangle\) is that

\[ \text{fmap } (h \cdot) \text{ fs } = \text{fmap } (\cdot k) \text{ gs } \Rightarrow \text{fmap f } (\text{fs } \langle*\rangle \text{ xs }) = \text{gs } \langle*\rangle \text{ fmap } k \text{ xs} \]

\[ \square \]

Lemma 19.

\[ \text{fmap fuse } \cdot \text{travFunList single } = \text{id} \]

\[ \square \]

Proof. We proceed by structural induction over the FunList argument. For the base case Done t, we have:

\[ \text{fmap fuse } (\text{travFunList single } (\text{Done } t)) \]
\[ = (\text{travFunList}) \]
\[ \text{fmap fuse } (\text{pure } (\text{Done } t)) \]
\[ = (\text{pure } \text{for } \text{FunList}) \]
Profunctor Optics

\[
\begin{align*}
&fmap \ fuse (\text{Done} (\text{Done} t)) \\
&= \begin{array}{l}
[ [ \text{fmap for FunList} ]]
\end{array}
\text{Done} (\text{fuse} (\text{Done} t)) \\
&= \begin{array}{l}
[ [ \text{fuse} ]]
\end{array}
\text{Done} t
\end{align*}
\]

as required. For the inductive case \(\text{More} x l\), we assume the inductive hypothesis that

\[
\begin{align*}
&fmap \ fuse (\text{travFunList single} l) = l
\end{align*}
\]

and then calculate:

\[
\begin{align*}
&fmap \ fuse (\text{travFunList single} (\text{More} x l)) \\
&= \begin{array}{l}
[ [ \text{travFunList} ]]
\end{array}
\text{fmap} \ fuse (\text{pure More} (\text{pure} x) \text{travFunList single} l) \\
&= \begin{array}{l}
[ [ \text{pure and \<\}\ for \text{FunList} ]]
\end{array}
\text{fmap} \ fuse (\text{More} x (\text{Done More}) (\text{pure} x) \text{travFunList single} l) \\
&= \begin{array}{l}
[ [ \text{free theorem of \<\}) \ (\text{Lemma 18}), \ with \ k = \text{id} ]]
\end{array}
\text{fmap} (\text{fuse}\cdot) (\text{More} x (\text{Done More})) (\text{pure} x) \text{travFunList single} l \\
&= \begin{array}{l}
[ [ \text{fmap for FunList} ]]
\end{array}
\text{More} x (\text{Done} ((\text{fuse}\cdot) \cdot \text{More})) (\text{pure} x) \text{travFunList single} l \\
&= \begin{array}{l}
[ [ \text{little sublemma (see below) } ]]
\end{array}
\text{More} x (\text{Done} (\text{flip fuse})) (\text{pure} x) \text{travFunList single} l \\
&= \begin{array}{l}
[ [ \text{More} x (\text{Done} (\text{flip f})) \ (\text{pure} x) l = \text{More} x (\text{fmap f l}) ]]
\end{array}
\text{More} x (\text{fmap fuse} (\text{travFunList single} l)) \\
&= \begin{array}{l}
[ [ \text{induction} ]]
\end{array}
\text{More} x l
\end{align*}
\]

which completes the proof. We used a little sublemma that

\[
(\text{fuse}\cdot) \cdot \text{More} = \text{flip fuse}
\]

whose justification is simply a matter of expanding definitions.

Then we need to show that the translation from concrete to profunctor traversals is a profunctor morphism, as with the other three classes of optic.

**Lemma 20.** For given \(k :: P A B\) for some cartesian, co-cartesian, monoidal profunctor \(P\) and types \(A, B\), the function \(\text{flip traversalC2P} k :: \text{Traversal A B s t} \to P s t\) is a profunctor morphism from \(\text{Traversal A B}\) to \(P\). □

**Proof.** We have:

\[
\begin{align*}
&\text{dimap f g} (\text{flip traversalC2P} k (\text{Traversal h})) \\
&= \begin{array}{l}
[ [ \text{flip} ]]
\end{array}
\text{dimap f g} (\text{traversalC2P} (\text{Traversal h}) k)
\end{align*}
\]
\[ \dimap f g \cdot \text{dimap composition} \cdot \text{dimap} (h \cdot f) \cdot \text{dimap} \text{traverse} k \]

==

\[ \text{dimap} (h \cdot f) \cdot \text{dimap} \text{traverse} k \]

==

\[ \text{dimap for Traverse} \cdot \text{dimap} \text{traverse} \]

==

\[ \text{Lemma 14: traversal on FunLists} \cdot \text{traverse} \]

==

\[ \text{dimap for Traverse} \cdot \text{Traverse} \]

==

\[ \text{Lemma 19: traversal with singletons} \cdot \text{Traverse} \]

In the middle, we used a specialization of the free theorem of traverse to

\[ \dimap id (fmap g) \cdot \text{traverse} = \dimap (fmap g) \cdot id \cdot \text{traverse} \]

Thus

\[ \dimap f g \cdot \text{flip traversal} = \text{flip traversal} \cdot \dimap f g \]

as required.

\[ \heartsuit \]

Finally, we can proceed with the proof that the concrete and profunctor representations of traversals are equivalent.

**Proof (of Theorem 4).** As with the earlier proofs, one direction is fairly straightforward:

\[ \text{traversalP2C} \cdot \text{traversalC2P} \cdot \text{traverse} \]

==

\[ \text{traversalP2C} \cdot \text{traversalC2P} \cdot \text{traverse} \]

==

\[ \text{Lemma 14: traversal on FunLists} \cdot \text{traverse} \]

==

\[ \text{Lemma 19: traversal with singletons} \cdot \text{Traverse} \]

For the other direction, we have:
\textbf{Profunctor Optics}

\[
\text{traversalC2P (traversalP2C l) k} \\
= \bigl[ \, \text{traversalP2C} \bigr] \\
\text{traversalC2P (l (Traversal single)) k} \\
= \bigl[ \, \text{flip} \, \bigr] \\
\text{flip traversalC2P k (l (Traversal single))} \\
= \bigl[ \, \text{Lemma 20: flip traversalC2P k is a profunctor morphism} \bigr] \\
l (\text{flip traversalC2P k (Traversal single)}) \\
= \bigl[ \, \text{flip} \, \bigr] \\
l (\text{traversalC2P (Traversal single) k}) \\
= \bigl[ \, \text{Lemma 17: traversal of a singleton} \bigr] \\
l (\text{dimap single fuse (traverse k)}) \\
= \bigl[ \, \text{Lemma 17: traversal of a singleton} \bigr] \\
l (\text{dimap id fuse (dimap id single k)}) \\
= \bigl[ \, \text{dimap composition} \bigr] \\
l (\text{dimap id (fuse \cdot single) k}) \\
= \bigl[ \, \text{fuse \cdot single = id} \bigr] \\
l (\text{dimap id id k}) \\
= \bigl[ \, \text{dimap identity} \bigr] \\
l k
\]

which completes the proof.

\textcopyright
About the authors

Matthew Pickering is a PhD student at the University of Bristol. Contact him at matthew.pickering@bristol.ac.uk.

Jeremy Gibbons is Professor of Computing at the University of Oxford. Contact him at jeremy.gibbons@cs.ox.ac.uk.

Nicolas Wu is a Lecturer in Computer Science at the University of Bristol. Contact him at nicolas.wu@bristol.ac.uk.