# Verified Profunctor Optics in Idris

Oliver Balfour

October 26, 2021

## Abstract

Optics are a commonly used design pattern in industrial functional programming. They are convenient combinators for reading and updating fields in composite data structures. Common implementations such as Edward Kmett's Haskell `lens` library are highly complex. We discuss profunctor optics, a modern formulation of optics which is more flexible than the more common van Laarhoven formulation. This report discusses the implementation and formal verification of profunctor optics in Idris, a dependently typed functional programming language and theorem prover. TODO summarise results, discussion and conclusion

# Contents

Introduction . . . 2
Background . . . 3
    Idris . . . 3
    Dependent Types . . . 3
    Propositions as Types . . . 3
    Proof Techniques . . . 4
    Limitations . . . 5
    Functors . . . 6
    Profunctors . . . 7
    Optics . . . 7
    Profunctor Optics . . . 8
Formally Verified Profunctor Optics . . . 8
Related Work . . . 8
Conclusion . . . 9
Appendix: Source Code . . . 11
    Simple Optics: PrimitiveOptics.idr . . . 11
    Morphisms: Morphism.idr . . . 12
    Verified Functors and Applicatives: VFunctor.idr . . . 13
    Verified Profunctors: VProfunctor.idr . . . 19
    Profunctor Optics: Main.idr . . . 22

# Introduction

The view-update problem is the problem of how to neatly read and write small components of large composite data structures (Foster et al. 2005). In imperative languages, objects are generally mutated in-place, circumventing the view-update problem altogether. Pure functional programming languages however are not afforded mutable variables, making the issue pernicious in industrial programs with highly complex data structures.

Optics are a pure functional solution to the view-update problem (Foster et al. 2005). Data structures representing components in real world systems frequently have dozens of fields and nested data structures with additional complexity. In a pure functional language, updating a field in a composite data type such as `Maybe (a, Bool)` requires boilerplate functions for every such composite type as in the below Idris code:

```
updateComplexType : (a -> b) -> Maybe (a, Bool) -> Maybe (b, Bool)
updateComplexType f (Just (x, y)) = Just (f x, y)
updateComplexType f Nothing = Nothing
```

As data structures become more complex, writing getters and setters becomes a tedious and bug-prone task. Optics are objects which represent a view into a data type which can be composed to create views into composite types, and used to view or update fields. Using the profunctor optics library discussed in this report, the above `updateComplexType` function may be defined as `updateComplexType = update (op . π₁)` where `op` is an optic for optional (`Maybe a`) types and `π₁` is a left projection optic for product types.

However, even in imperative languages there are often many benefits from using immutable objects. In JavaScript for example, there is an increasing trend towards pure functional state management for designing user interfaces, termed *declarative UI* (Steinberger 2021). Libraries such as Redux.js (Abramov et al. 2015) use an immutable state object with a group of actions that act on the state type whenever an event is triggered by user interactions. This presents numerous benefits such as simple control flow and undo/redo functionality. However, this requires a new state object after each event with perhaps a single field changed. The conventional approach in JavaScript is to use Immer.js (Weststrate et al. 2019), which rather than using pure optics exploits esoteric language features to emulate mutability on immutable objects. However, there is no fundamental reason why optics would not work equally well.

Profunctor optics are a very flexible and powerful encoding of optics, however they are highly complex, demonstrating a need for quality assurance.

In statically typed languages, types correspond with certain logical propositions and programs serve as proofs of those propositions (Wadler 2015). This insight is known as the Curry-Howard correspondence (Sørensen and Urzyczyn 2006) and it underpins theorem provers and formal verification. Dependent types are a feature of some type systems which allows types to depend on values. Dependent types allow programmers to encode first order logical propositions and equalities between expressions into the type system and prove many useful theorems and properties of their programs.

Idris is a dependently-typed functional programming language similar to Haskell which may be used as a theorem prover. This report discusses using Idris to both implement and formally verify a profunctor optics library. Dependent types are used to express and prove that the profunctor optics adhere to all relevant mathematical laws and desirable properties.

## Background

### Idris

Idris is a Haskell-like functional programming language with first-class support for dependent types. It is an actively developed experimental research language. Syntactically Idris and Haskell are almost identical, the most notable difference is that `:` is used to declare types and `::` is the list `cons` constructor. Additionally, types are first class citizens so functions may accept or return types (values may depend on types), a strict generalisation of Haskell which only allows types to depend on types (type constructors).

Idris additionally has linear types based on quantitative type theory which allow types to be annotated with requirements that they must be used exactly 0 or 1 times at runtime (Brady 2021). Idris also has implicit (inferred) arguments. Unlike Haskell, Idris does not possess type inference, as type inference is undecidable in general for dependent types with non-empty typing contexts (Dowek 1993).

Idris is unique in that it is a practical and simple functional programming language to understand given prerequisite Haskell experience, and it doubles up as a theorem prover. The type system is powerful enough to encode theorems about equalities between expressions and universal and existential quantifiers. This allows programmers to express and prove complex properties and invariants of their programs alongside their code, which makes languages like Idris a good candidate language for critical infrastructure and similar systems.

### Dependent Types

Dependent types are types that depend on values. For example, the Idris type `Vect 3 Int` is inhabited by vectors of precisely 3 integers. We say the type is indexed by the value `3`.

Some other languages have equivalent types such as `std::array<int, 3>` in C++. However, in C++, non-type template parameters (that is, values the type depends on) must be statically evaluated because generic types are monomorphised at compile time (ISO 2020, 14.1.4). This means template arguments cannot be non-trivial expressions as in Idris.

There are two main kinds of dependent types. $\Pi$ types generalise the `Vect 3 Int` example above. The type $\Pi x.Px$, which is expressed as `(x:a) -> P x` in Idris for some `P : (x:a) -> Type` is a function type where the codomain type depends on the value of the argument `x`. This allows functions to dynamically compute their return types in a type-safe manner. For instance, the `replicate` function in the Idris standard library has the type `replicate : (len : Nat) -> a -> Vect len a`, using a $\Pi$ type to construct a length `len` vector of copies of an object.

The other kind is $\Sigma$ types, which in Idris are known as dependent pairs. The type $\Sigma x.Px$ corresponds with the dependent pair `(x:a ** P x)` which is a pair of a value and a type where the type may depend on the value. Dependent pairs are outside the scope of this report.

As types can depend on values, Idris has an equality type `=` indexed by two values. It has one constructor `Refl : x = x` (reflexivity). An instance of `Refl : a = b` in some cases is obtainable using type rewriting rules discussed later, in which case the expressions `a` and `b` share the same normal form and are intensionally equal.

Dependent types are useful because they allow programmers to express more sophisticated types such as length indexed vectors, which allow programmers to write total matrix multiplication functions. Additionally, logical propositions correspond with types, and dependent types are expressive enough to allow a language to be used as a theorem prover and formally verify properties of programs.

### Propositions as Types

The Curry-Howard correspondence, also known as *Propositions as Types*, is the observation that propositions in a logic correspond with types in a language and proofs correspond with function definitions (Wadler 2015). This observation underpins theorem provers like Idris, Coq and Lean. The theorem statement or goal is encoded in a type signature. The function body is a proof of the goal. If the program is well-typed, the proof is correct.

Every consistent type system encodes some set of logical propositions. Dependent types are expressive enough that they can encode an intuitionistic or constructive logic complete with implications, conjunction, disjunction, negation, quantifiers and equalities.

In Idris, the type `a` is interpreted as a proposition $a$, where $a$ is true iff `a` as a type is inhabited. A proof of $a$ is simply an object of type `a`. The function type `a -> b` is interpreted as a logical implication $a \implies b$. Intuitively, if a total function of type `a -> b` exists then the existence of an `a` guarantees the existence of a `b`. Logical negation is encoded as `a -> Void` where `Void` is uninhabited.

The equality type is especially useful in conjunction with. If $a = b$ and a constructive proof of this exists then `a = b` is a singleton type, and if no proof exists it is uninhabited and thus false.

A is tabulated below. Σ types are encoded using a construct called dependent pairs, which is not discussed in this report. Π types are encoded with function types where the return type depends on the argument.

| Logic | Type Theory | Idris Type |
|---|---|---|
| $T$ | $\top$ | `()` |
| $F$ | $\bot$ | `Void` |
| $a \wedge b$ | $a \times b$ | `(a, b)` |
| $a \vee b$ | $a + b$ | `Either a b` |
| $a \Rightarrow b$ | $a \rightarrow b$ | `a -> b` |
| $\forall x.Px$ | $\Pi x.Px$ | `(x:a) -> P x` |
| $\exists x.Px$ | $\Sigma x.Px$ | `(x:a ** P x)` |
| $\neg p$ | $p \rightarrow \bot$ | `p -> Void` |
| $a = b$ | $a = b$ | `a = b` |

Table 1: Corresponding connectives and quantifiers. Note that the predicates in Idris are of the form `P : (x : a) -> Type` where `P x = ()` or `P x = Void`.

## Proof Techniques

Idris will reduce values in types to their normal form by applying function definitions. It will attempt to unify both sides of equality types as well by reducing either side until it coincides with the other. This allows proofs to skip many intermediate simplification steps. Idris will generally reduce values in types to their normal form, analogous to simplifying mathematical expressions. For instance `3 + 7 = 11` will be rewritten to `10 = 11` (which of course is uninhabited).

This allows us to write simple proofs as below, which are analogous to unit tests.

```
fact : Nat -> Nat
fact Z = 1
fact (S n) = (S n) * fact n

factTheorem : fact 5 = 120
factTheorem = Refl

factTheorem2 : (S n) * fact n = fact (S n)
factTheorem2 = Refl
```

The main proof techniques in Idris are structural induction, rewriting types and ex falso quodlibet.

Structural induction is the most common tool. This entails case splitting a theorem over each constructor and recursively invoking the theorem on smaller components of an inductively defined structure. If Idris can determine the theorem is total as the recursive calls eventually reach the base case, the proof will type check. Recursive calls are analogous to inductive hypotheses.

For example,

```
-- ∀ n : Nat. n + 0 = n
natPlusZeroId : (n : Nat) -> n + 0 = n
natPlusZeroId Z = Refl
natPlusZeroId (S n) = cong S (natPlusZeroId n)

-- ∀ xs : List a. xs ++ [] = xs
listConcatRightNilId : (xs : List a) -> xs ++ [] = xs
listConcatRightNilId [] = Refl
listConcatRightNilId (x::xs) = cong (x::) (listConcatRightNilId xs)
```

These proofs invoke a lemma in the Idris Prelude, `cong : (f:t->u) -> (a = b) -> (f a = f b)`, which is analogous to the rule $\forall f.\ a = b \implies f(a) = f(b)$ in mathematics.

Idris also provides a facility for rewriting the goal type using an equality. For example:

```
trans' : a = b -> b = c -> a = c
trans' p1 p2 =
  -- goal: a = c
  rewrite p1 in  -- replace `a` with `b` in `a = c`
    -- new goal: b = c
    p2
```

Rewriting can be convenient, however using a number of rewrites makes proofs difficult to follow. Prelude functions such as `trans`, `sym`, `cong` and `replace` can accomplish the same tasks with a more conventional proof structure.

As intuitionistic logics do not have the law of the excluded middle or double negation, proof by contradiction is not possible. Instead, ex falso quodlibet, the principle of explosion, must be used. In some cases a function has cases which are not possible but well-typed proofs must exist for those cases to satisfy the totality checker. In this case, rather than deriving a contradiction to show the state is not possible, the contradiction can be used with the function `void : Void -> a` to derive the proof goal.

Idris has holes like Haskell, which are placeholder expressions denoted `?hole_name`. There is a `:t hole_name` command in the Idris REPL which prints out the typing context and goal, much like other theorem provers like Coq. This is immensely useful in developing proofs.

## Limitations

Dependently typed theorem provers are intuitionistic in nature, which is strictly less powerful than classical logic. There exist theorems which can be proven with classical logic for which no constructive proof in Idris exists.

Double negation cancellation is not true in general, as there is no canonical map `((a -> Void) -> Void) -> a`. Existence statements cannot be proven without finding a witness to the proof, so a proof by contradiction that $\neg\forall x.P(x)$ does not imply $\exists x.\neg P(x)$. Instead, a dependent pair containing an explicit $x$ satisfying $\neg P(x)$ must be constructed, which may not be possible.

Additionally, there is a distinction between intensional and extensional equality of functions. In mathematics, the statements $f = g$ and $\forall x.f(x) = g(x)$ are equivalent. In Idris however, only the forward implication is true. Function equality is intensional, meaning functions are equal iff the normal form of their lambda expressions are $\alpha$-equivalent, so they are the equal up to renaming bound variables. In many cases extensionally equal functions are not intensionally equal, so the Idris equality type may not be helpful. It is possible to use the built-in `believe_me : a -> b` proof to introduce an extensionality axiom, however Idris cannot rewrite types if they invoke axioms as there essentially is no definition to substitute.

These limitations mean that many theorems of interest either cannot be proven or are much more difficult to prove in Idris. TODO examples

## Functors

Before discussing profunctors, we discuss categories, functors, applicative functors and monads. A category is a mathematical object which consists of a collection of objects and between any two objects a collection of arrows or morphisms (Mac Lane 1970). We will only discuss locally small categories so we may assume these collections of morphisms are sets, called Hom-sets. The only properties categories must have is an associative composition operation on morphisms and an identity morphism on each object. Categories are a useful abstraction as they generalise objects and structure preserving maps between them from many different fields. There is a category of sets where objects are sets and Hom-sets contain functions, $\mathrm{Hom}(A, B) = \{f : A \to B \text{ is a function}\}$. Morphism composition is function composition, and there exists an identity function on each set. In group theory, there is a category of groups where objects are groups and morphisms are group homomorphisms. Many other examples exist, for example partially ordered sets form categories where objects are elements and exactly one morphism exists between every ordered pair.

Notably, types and total functions in Idris form a category similar to the category of sets. For convenience, these categories are assumed the same.

Functors are structure preserving maps between categories (and thus morphisms in the category of categories). They consist of two components mapping objects and morphisms from the domain category to objects and morphisms in the codomain category. Functors respect identities $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ and composition $F(f \circ g) = F(f) \circ F(g)$ An endofunctor is a functor which maps into the same category.

In Idris, generic containers such as lists and trees are endofunctors. The type constructor `List : Type -> Type` is the component of the functor mapping objects, and the `map : (a -> b) -> (List a -> List b)` function is the component mapping morphisms. Additionally, the partially applied arrow type `a->` is a functor, the covariant Hom functor (and partially applied Hom profunctor).

Monads are a subset of endofunctors equipped with two maps `η : a -> m a` and `μ : m (m a) -> m a` named pure/return and join respectively, where `m` is the monad. In functional programming, they can be used to encapsulate and compose side effecting functions in a type safe way (Moggi 1991). The `IO` type constructor is a monad representing side effects which allows side effects to be composed and enforce that functions emitting side effects have a return type containing `IO`. Lists form a monad where `pure x = [x]` and `join = concat` and function composition results in a list of all possible applications of functions to arguments.

Monads satisfy the following laws

1. ``∀ f,x. (return `join` f) x = f x`` (left identity law, tildes denote infix function calls)
2. ``∀ f,x. (f `join` return) x = f x`` (right identity law)
3. ``∀ f,g,h,x. f `join` (g `join` h) $ x = (f `join` g) `join` h $ x`` (associativity)

Monads naturally give rise to a category known as a Kleisli category (Mac Lane 1970 p.147). In this category, the objects are the same as the category of types, and the morphisms are each of the form `a -> m b` so $\mathrm{Hom}(\mathtt{a}, \mathtt{b}) = \{f : \mathtt{a\ ->\ m\ b}\}$. Morphism composition utilises the join operation, so $f \circ_m g = \mu \circ f \circ g$.

Applicative functors are a subset of endofunctors introduced by McBride and Paterson 2008 as a useful abstraction in functional programming intermediate between endofunctors and monads. They are endofunctors equipped with maps `pure : a -> f a` and `ap : f (a -> b) -> (f a -> f b)` (written `<*>` as an infix operator) for every applicative `f` satisfying the following properties

1. `∀ v. pure id <*> v = v` (identity law)
2. `∀ g,x. pure g <*> pure x = pure (g x)` (homomorphism law)
3. `∀ u,y. u <*> pure y = pure (\x => x y) <*> y` (interchange law)

4. `∀ u,v,w. ((pure (.) <*> u) <*> v) <*> w = u <*> (v <*> w)` (composition law)

It is worth mentioning applicative functors and monads briefly

## Profunctors

Profunctors are a generalisation of functors which relate to Hom-sets. A profunctor from category $C$ to $D$ formally is a functor $D^{op} \times C \to \textbf{Set}$. ( TODO what's a dual category, product category?)

In Idris, profunctors correspond to arrow-like types. For instance, '->' is a profunctor.

Profunctors give rise to a category in which they are the Hom profunctor (is this true in general?)

Intuitively, profunctors are type constructors that construct an function arrow like type.

Hom is a profunctor Hom functor is partially applied hom profunctor

-> is the Hom profunctor Kleisli is the Hom profunctor in the Kleisli category

There are two profunctors of interest: '->' (the standard Hom profunctor in the cat of Idris types) and '(a,b:Type)->(a->f b)' for functor/monad 'f' (the Hom profunctor in the Kleisli category). The latter allows us to accumulate side effects while using our optics

Dictionary / hash map is a profunctor (https://bartoszmilewski.com/2017/03/29/ends-and-coends/)

VProfunctor = verified not v-enriched

## Optics

Optics are data accessors that ease reading and writing into composite data structures. Industrial programs tend to have very complex and deeply nested data structures, which makes tasks like copying and updating a single field in a deeply nested data structure in a functional style very cumbersome. Optics are an elegant solution to this problem. They are objects which represent a view into a field of a data structure which can be composed for nested structures and used to view and update the field they model.

There are many encodings of optics. This report discusses simple algebraic data type optics, van Laarhoven optics (Laarhoven 2011) and profunctor optics (Pickering, Gibbons, and Wu 2017). Most established implementations such as the `lens` library in Haskell (Kmett 2012) use the van Laarhoven design, however these have many limitations. One of the principal benefits of optics is that they compose elegantly, however the van Laarhoven encoding makes a distinction between lenses (optics for product types) and prisms (optics for sum types) and does not allow composition of lenses and optics, so you cannot express an optic for the integer in a `Maybe (Integer, String)` type (Pickering, Gibbons, and Wu 2017).

However, profunctor optics generalise optics to work around these issues. As with many other functional programming design patterns, they are inspired by category theory, specifically the notion of a profunctor. Profunctor optics are generic over the typeclass of profunctors, so they allow choice of profunctor in which to use an optic. This allows programmers to not just use optics to view and update, but to also accumulate side effects and recover constructors for sum types in the process.

A major concern is that optics are very complicated. Formal verification of profunctor optics is thus a natural application of dependent types.

The simplest encoding of optics is to create typeclasses (interfaces in Idris) for lenses, prisms, traversals and adapters. Lenses are optics for product types that allow viewing and updating fields. Prisms are optics for sum types that allow pattern matching if a field is present and constructing a sum type from one of the components. Adapters and traversals are optics for isomorphic types and container types respectively and are not discussed in this report.

In the below encoding, `PrimitiveLens a b (a,c) (b,c)` means a view into the `a` in `(a,c)`. The other two type variables add a degree of freedom when updating tuples to change the type of the left element of the tuple.

```
record PrimitiveLens a b s t where
  constructor MkPrimLens
  view : s -> a
  update : (b, s) -> t

record PrimitivePrism a b s t where
  constructor MkPrimLens
  match : s -> Either t a
  build : b -> t

-- Left projection lens
_1 : PrimitiveLens a b (a,c) (b,c)
_1 = MkPrimLens fst update where
  update : (b, (a, c)) -> (b, c)
  update (x', (x, y)) = (x', y)
```

Then `view _1 (2, True) == 2`. However, there is no clear way to compose two lenses or two prisms using this encoding, and it is not possible to compose a lens and a prism using these two typeclasses.

A more powerful encoding is the van Laarhoven functor transformer lenses (Laarhoven 2011). These are parameterised over the functor typeclass, where different functors applied to the optics change how they behave. In Haskell,

```
type LaarhovenLens a b = forall f. Functor f => (b -> f b) -> (a -> f a)
```

Under this encoding, the above product projection lens would have type `LaarhovenLens (a,b) a`. It can be modified to have the additional degree of freedom in the simpler encoding.

The `Const a` functor `newtype Const a b = { unConst :: a }` stores a value of type `a`. Applied to the above definition, it produces a getter `view optic structure = optic (\x -> Const x) structure`. Likewise, the identity functor `newtype Id a = { unId :: a }` produces an update function `update optic field structure = optic (\x -> Id field) structure`.

These optics are simple functions and so support composition, however lenses and prisms are still mutually exclusive and cannot be composed. This leaves profunctor optics, which generalise van Laarhoven's functor transformer lenses and are flexible enough to support composition.

### Profunctor Optics

what are profunctor optics? why are they good? what are common/useful optics with examples? optics on type indexed data types??? correspondence with van Laarhoven Boisseau and Gibbons 2018 - this means even though they're harder to write you can map primitive ones to complex ones

## Formally Verified Profunctor Optics

> We can then have a penultimate section discussing a framework for formal verification of profunctor optics in Idris and discussing the structure and details of your solution (minimal use of source code here). Concretely, this section should properly motivate the formal verification of profunctor optics and then progress through your solution. Talk concepts and give formal examples but avoid using source code (you can refer to sections in the appendix where necessary though).

## Related Work

https://bartoszmilewski.com/2017/03/29/ends-and-coends/

http://www.mtm.ufsc.br/ ebatista/2016-2/maclanecat.pdf https://dl.acm.org/doi/pdf/10.1145/3236779

Research Rabbit

https://dl.acm.org/doi/pdf/10.1145/1040305.1040325

Existing research on profunctor optics

Much existing work is focused on the correspondence between the van Larrhoven and profunctor representations of lenses, prisms, adapters and traversals. Boisseau and Gibbons 2018 provides a proof of the correspondence with the Yoneda lemma, and previous work including Pickering, Gibbons, and Wu 2017 and Milewski 2017 provide proofs invoking more complex machinery such as Tambara modules and tensor products.

No prior work is known to have been done on formally verified profunctor optics.

Future research on profunctor optics for dependent types such as type indexed syntax trees could be very useful. This would enable programming languages written in Idris to use dependent types to verify all syntax trees are well typed and use optics to elegantly traverse, view and update subtrees.

## Conclusion

> Finally in the conclusion you can discuss what you've accomplished, what improvements could be made, and other related work in the field (i.e where the current boundaries of formally verified profunctor optics are if any - and general boundaries of profunctor optics research).

# Bibliography

Abramov, Dan et al. (2015). *Redux: A Predictable State Container for JS Apps.* URL: `https://redux.js.org/` (visited on 10/26/2021).

Boisseau, Guillaume and Jeremy Gibbons (2018). "What you needa know about Yoneda: Profunctor optics and the Yoneda Lemma (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 2.ICFP, pp. 1–27.

Brady, Edwin (2021). "Idris 2: Quantitative Type Theory in Practice". In: *arXiv preprint arXiv:2104.00480.*

Dowek, Gilles (1993). "The undecidability of typability in the $\lambda\Pi$-calculus". In: *International Conference on Typed Lambda Calculi and Applications.* Springer, pp. 139–145.

Foster, J Nathan et al. (2005). "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem". In: *ACM SIGPLAN Notices* 40.1, pp. 233–246.

ISO (Feb. 2020). *ISO/IEC 14882:2020 Information technology — Programming languages — C++.* Geneva, Switzerland: International Organization for Standardization. URL: `https://www.iso.org/standard/79358.html`.

Kmett, Edward (2012). *Lens: Lenses, Folds, and Traversals.* URL: `https://lens.github.io/` (visited on 10/26/2021).

Laarhoven, Twan van (2011). "Lenses: viewing and updating data structures in Haskell". Institute for Computing and Information Sciences, Radboud University Nijmegen. URL: `https://www.twanvl.nl/blog/news/2011-05-19-lenses-talk`.

Mac Lane, Saunders (1970). *Categories for the working mathematician.* 2nd ed. Springer-Verlag New York.

McBride, Conor and Ross Paterson (2008). "Applicative programming with effects". In: *Journal of functional programming* 18.1, pp. 1–13.

Milewski, Bartosz (2017). *Profunctor Optics: The Categorical View.* URL: `https://bartoszmilewski.com/2017/07/07/profunctor-optics-the-categorical-view/` (visited on 08/07/2021).

Moggi, Eugenio (1991). "Notions of computation and monads". In: *Information and computation* 93.1, pp. 55–92.

Pickering, Matthew, Jeremy Gibbons, and Nicolas Wu (2017). "Profunctor optics: Modular data accessors". In: *arXiv preprint arXiv:1703.10857.*

Sørensen, Morten Heine and Pawel Urzyczyn (2006). *Lectures on the Curry-Howard isomorphism.* Elsevier.

Steinberger, Peter (2021). *The Shift to Declarative UI.* URL: `https://increment.com/mobile/the-shift-to-declarative-ui/` (visited on 10/26/2021).

Wadler, Philip (2015). "Propositions as types". In: *Communications of the ACM* 58.12, pp. 75–84.

Weststrate, Michel et al. (2019). *Immer: Create the next immutable state tree by simply modifying the current tree.* URL: `https://github.com/immerjs/immer` (visited on 10/26/2021).

## Appendix: Source Code

Mirrored at https://github.com/OliverBalfour/ProfunctorOptics

### Simple Optics: PrimitiveOptics.idr

```
module Primitive.PrimitiveOptics

%default total

-- Primitive optics
-- Simpler to write than profunctor optics but they don't compose well
-- Solution: write primitive optics and map them to profunctor optics

public export
data PrimLens : Type -> Type -> Type -> Type -> Type where
  MkPrimLens
    :  (view : s -> a)
    -> (update : (b, s) -> t)
    -> PrimLens a b s t

public export
data PrimPrism : Type -> Type -> Type -> Type -> Type where
  MkPrimPrism
    :  (match : s -> Either t a)
    -> (build : b -> t)
    -> PrimPrism a b s t

public export
data PrimAdapter : Type -> Type -> Type -> Type -> Type where
  MkPrimAdapter
    :  (from : s -> a)
    -> (to : b -> t)
    -> PrimAdapter a b s t

-- Examples of simple optics

-- Product left/right projection lens
π₁ : PrimLens a b (a, c) (b, c)
π₁ = MkPrimLens fst update where
  update : (b, (a, c)) -> (b, c)
  update (x', (x, y)) = (x', y)
π₂ : PrimLens a b (c, a) (c, b)
π₂ = MkPrimLens snd update where
  update : (b, (c, a)) -> (c, b)
  update (x', (y, x)) = (y, x')

-- Sign of an integer lens
sgn : PrimLens Bool Bool Integer Integer
sgn = MkPrimLens signum chsgn where
  signum : Integer -> Bool
  signum x = x >= 0
  chsgn : (Bool, Integer) -> Integer
  chsgn (True,  x) =  abs x
  chsgn (False, x) = -abs x

-- Maybe prism
op' : PrimPrism a b (Maybe a) (Maybe b)
op' = MkPrimPrism match build where
  match : Maybe a -> Either (Maybe b) a
  match (Just x) = Right x
  match Nothing = Left Nothing
  build : b -> Maybe b
  build = Just

-- Adapter for the isomorphism (A x B) x C = A x (B x C)
prodAssoc : PrimAdapter ((a,b),c) ((a',b'),c') (a,(b,c)) (a',(b',c'))
prodAssoc = MkPrimAdapter (\(x,(y,z)) => ((x,y),z)) (\((x,y),z) => (x,(y,z)))
```

### Morphisms: Morphism.idr

```
module Category.Morphism

%default total

-- Derived from Data.Morphisms

-- Morphisms in the category of Idris types
-- This wrapper exists to help Idris unify types in some proofs
public export
record Morphism a b where
  constructor Mor
  applyMor : a -> b

infixr 1 ~>

public export
(~>) : Type -> Type -> Type
(~>) = Morphism

-- Morphisms in a Kleisli category
-- Functions of type a -> f b for a functor or monad f
public export
record KleisliMorphism (f : Type -> Type) a b where
  constructor Kleisli
  applyKleisli : a -> f b

infixr 1 ~~>

public export
(~~>) : {f : Type -> Type} -> Type -> Type -> Type
(~~>) = KleisliMorphism f

-- Helpers

public export
eta : (a -> b) -> (a -> b)
eta f = \x => f x
```

```

-- f = \x => f x
public export
ext : (f : a -> b) -> (eta f = f)
ext f = Refl

-- id . f = f
public export
idCompLeftId : (f : a -> b) -> (\x => x) . f = f
idCompLeftId f = ext f

-- Extensionality axiom: used sparingly, uses a back door in the type system
-- Idris cannot rewrite types using axioms so this must be avoided at all
 ↪  costs
public export
extensionality : {f, g : a -> b} -> ((x : a) -> f x = g x) -> f = g
extensionality {f} {g} prf = believe_me ()
```

### Verified Functors and Applicatives: VFunctor.idr

```
module Category.VFunctor

import Category.Morphism
import Data.Vect

%default total
%hide Applicative
%hide (<*>)
%hide (<$>)

infixl 4 <*>
infixl 4 <$>

-- Verified functors
-- Optics over functorial types can be verified in part using functor laws
public export
interface VFunctor (f : Type -> Type) where
  -- fmap maps functions
  fmap : (a -> b) -> (f a -> f b)
  -- fmap respects identity, F(id) = id
  fid : (x : f a)
    -> fmap (\x => x) x = x
  -- fmap respects composition, F(g . h) = F(g) . F(h)
  fcomp : (x : f a) -> (g : b -> c) -> (h : a -> b)
    -> fmap (g . h) x = (fmap g . fmap h) x
  -- Infix alias for fmap
  (<$>) : (a -> b) -> (f a -> f b)
  f <$> x = fmap f x
  infixSame : (g : a -> b) -> (x : f a) -> fmap g x = g <$> x

-- Verified applicative functors
public export
interface VFunctor f => VApplicative (f : Type -> Type) where
  -- pure (aka return, ⊠)
  ret : a -> f a
  -- ap
  (<*>) : f (a -> b) -> (f a -> f b)
  -- Identity law, pure id <*> v = v
  aid : (v : f a) -> ret (\x => x) <*> v = v
  -- Homomorphism law, pure g <*> pure x = pure (g x)
  ahom : (g : a -> b) -> (x : a)
    -> ret g <*> ret x = ret (g x)
  -- Interchange law, u <*> pure y = pure ($ y) <*> u
  aint : (u : f (a -> b)) -> (y : a)
    -> u <*> ret y = ret ($ y) <*> u
  -- Composition law, ((pure (.) <*> u) <*> v) <*> w = u <*> (v <*> w)
  acomp : (u : f (b -> c)) -> (v : f (a -> b)) -> (w : f a)
    -> ((ret (.) <*> u) <*> v) <*> w = u <*> (v <*> w)

-- Lists are functors
public export
implementation VFunctor List where
  -- fmap for lists is map
  fmap f [] = []
  fmap f (x::xs) = f x :: fmap f xs
  fid [] = Refl
  fid (x::xs) = cong (x::) (fid xs)
  fcomp [] g h = Refl
  fcomp (x::xs) g h = cong (g (h x) ::) (fcomp xs g h)
  infixSame f x = Refl

-- forall xs. xs ++ [] = xs
public export
nilRightId : (xs : List a) -> xs ++ [] = xs
nilRightId [] = Refl
nilRightId (x::xs) =
  let iH = nilRightId xs
  in rewrite iH in Refl

-- List concatenation is associative
public export
concatAssoc : (xs, ys, zs : List a) -> xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
concatAssoc [] ys zs = Refl
concatAssoc (x::xs) ys zs = cong (x::) (concatAssoc xs ys zs)

-- Lists are applicative functors
public export
implementation VApplicative List where
  -- pure makes a singleton list
  ret = (::[])
  -- ap applies a list of functions to a list of arguments
  -- [f1, ..., fn] <*> [x1, ..., xn] = [f1 x1, ..., f1 xn, f2 x1, ..., fn xn]
  (<*>) [] xs = []
  (<*>) (f::fs) xs = fmap f xs ++ (fs <*> xs)
  -- Laws
  aid [] = Refl
  aid (x::xs) =
    let iH = aid xs
```

```
      shed : (fmap (\x => x) xs = xs) = fid xs
      prf : ((fmap (\y => y) xs ++ []) = xs)
      prf = rewrite shed in rewrite nilRightId xs in Refl
  in cong (x::) prf
ahom g x = Refl
aint [] y = Refl
aint (u::us) y =
  let iH = aint us y
  in cong (u y::) iH
acomp us vs ws =
  let elimNil : (((fmap (.) us ++ []) <*> vs) <*> ws = ((fmap (.) us) <*>
   ↪  vs) <*> ws)
      elimNil = cong (\x => (x <*> vs) <*> ws) (nilRightId (fmap (.) us))
  in rewrite elimNil in case us of
    -- Goal: ((fmap (.) <*> us) vs) <*> ws = us <*> (vs <*> ws)
    [] => Refl
    (u::us') => let iH = acomp us' vs ws in let
      l1 : List (a -> c)
      l1 = fmap ((.) u) vs
      l2 : List (a -> c)
      l2 = (fmap (.) us') <*> vs
      step : ((l1 ++ l2) <*> ws = (l1 <*> ws) ++ (l2 <*> ws))
      step = concatDist l1 l2 ws
      elimNil2 : (fmap u (vs <*> ws) ++ (<*>) ((fmap (.) us' ++ []) <*> vs)
       ↪  ws = fmap u (vs <*> ws) ++ (((fmap (.) us') <*> vs) <*> ws))
      elimNil2 = cong (\x => fmap u (vs <*> ws) ++ (<*>) (x <*> vs) ws)
       ↪  (nilRightId (fmap (.) us'))
      prf : ((l1 ++ l2) <*> ws = fmap u (vs <*> ws) ++ (us' <*> (vs <*>
       ↪  ws)))
      prf = rewrite step in rewrite sym iH in rewrite elimNil2 in
        cong (++ (((fmap (.) us') <*> vs) <*> ws)) (
          -- Goal: ((fmap ((.) u) vs) <*> ws = fmap u (vs <*> ws))
          case vs of
            [] => Refl
            (v::vs') => let
              iH2 = acomp us' vs' ws
              step2 : ((<*>) (fmap ((.) u) vs') ws = fmap u (vs' <*> ws))
              step2 = apLemma u vs' ws
              step3 : (fmap (u . v) ws ++ (<*>) (fmap ((.) u) vs') ws = fmap
               ↪  (u . v) ws ++ fmap u (vs' <*> ws))
              step3 = cong (fmap (u . v) ws ++) step2
              step4 : (fmap (u . v) ws ++ fmap u (vs' <*> ws) = fmap u (fmap
               ↪  v ws) ++ fmap u (vs' <*> ws))
              step4 = rewrite fcomp ws u v in Refl
              step5 : (fmap u (fmap v ws) ++ fmap u (vs' <*> ws) = fmap u
               ↪  (fmap v ws ++ (vs' <*> ws)))
              step5 = fmapHom u (fmap v ws) (vs' <*> ws)
              final : (fmap (u . v) ws ++ ((fmap ((.) u) vs') <*> ws) = fmap
               ↪  u (fmap v ws ++ (vs' <*> ws)))
              final = (step3 `trans` step4) `trans` step5
              in final
          )
      in prf
  where
```

```
      -- Lemmas
      -- Empty xs gives empty fs <*> xs
      apRightNil : (fs : List (p -> q)) -> fs <*> [] = []
      apRightNil [] = Refl
      apRightNil (f::fs) = apRightNil fs
      -- (<*>) distributes over (++)
      concatDist : (as, bs : List (p -> q)) -> (xs : List p)
          -> (as ++ bs) <*> xs = (as <*> xs) ++ (bs <*> xs)
      concatDist [] bs xs = Refl
      concatDist (a::as) bs xs = rewrite concatDist as bs xs in
        concatAssoc (fmap a xs) (as <*> xs) (bs <*> xs)
      -- fmap is a monoid homomorphism over the (List a, (++), []) monoid
      fmapHom : (m : p -> q) -> (as, bs : List p)
        -> fmap m as ++ fmap m bs = fmap m (as ++ bs)
      fmapHom m [] bs = Refl
      fmapHom m (a::as) bs = rewrite fmapHom m as bs in Refl
      -- Function composition can be done before or after (<*>)
      apLemma : (m : q -> r) -> (as : List (p -> q)) -> (bs : List p)
        -> ((fmap ((.) m) as) <*> bs = fmap m (as <*> bs))
      apLemma m [] bs = Refl
      apLemma m (a::as) bs =
        let iH = apLemma m as bs
        in rewrite sym (fmapHom m (fmap a bs) (as <*> bs))
        in rewrite sym iH
        in rewrite fcomp bs m a
        in Refl

-- Maybe is an applicative functor
public export
implementation VFunctor Maybe where
  -- fmap maps over Just values
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
  fid (Just x) = Refl
  fid Nothing = Refl
  fcomp (Just x) g h = Refl
  fcomp Nothing g h = Refl
  infixSame f x = Refl

public export
implementation VApplicative Maybe where
  ret = Just
  -- ap returns a Just value iff it's possible to do so
  (<*>) (Just f) (Just x) = Just (f x)
  (<*>) _ _ = Nothing
  aid (Just x) = Refl
  aid Nothing = Refl
  ahom g x = Refl
  aint (Just f) y = Refl
  aint Nothing y = Refl
  acomp (Just u) (Just v) (Just w) = Refl
  acomp Nothing _ _ = Refl
  acomp (Just u) Nothing _ = Refl
  acomp (Just u) (Just v) Nothing = Refl
```

```idris

-- Either a (partially applied sum type) is an applicative functor
-- over the second type variable
public export
implementation {a:Type} -> VFunctor (Either a) where
  fmap f (Left x) = Left x
  fmap f (Right x) = Right (f x)
  fid (Left x) = Refl
  fid (Right x) = Refl
  fcomp (Left x) g h = Refl
  fcomp (Right x) g h = Refl
  infixSame f x = Refl

public export
implementation {a:Type} -> VApplicative (Either a) where
  ret = Right
  -- same as VApplicative Maybe, Left x is treated as Nothing and Right x
  -- as Just x
  (<*>) (Right f) (Right x) = Right (f x)
  (<*>) (Left x) y = Left x
  (<*>) _ (Left x) = Left x
  aid (Left x) = Refl
  aid (Right x) = Refl
  ahom g x = Refl
  aint (Left x) y = Refl
  aint (Right x) y = Refl
  acomp (Right u) (Right v) (Right w) = Refl
  acomp (Left _) _ _ = Refl
  acomp (Right u) (Left x) _ = Refl
  acomp (Right u) (Right v) (Left x) = Refl

-- Partially applied product type is a functor
-- over the second type variable
-- (a,) is only an applicative if a is a monoid (omitted)
public export
implementation {a:Type} -> VFunctor (a,) where
  fmap f (x, y) = (x, f y)
  fid (x, y) = Refl
  fcomp (x, y) g h = Refl
  infixSame f x = Refl

-- Morphism a = Hom(a, -) is an applicative functor,
-- the covariant Hom functor
public export
implementation {a:Type} -> VFunctor (Morphism a) where
  -- fmap is function composition
  -- The Mor wrapper is only present to help Idris unify types in proofs
  fmap f (Mor g) = Mor (f . g)
  fid (Mor f) = cong Mor (sym (ext f))
  fcomp (Mor f) g h = Refl
  infixSame f x = Refl

public export
implementation {a:Type} -> VApplicative (Morphism a) where
```

```
  ret x = Mor (const x)
  (<*>) (Mor f) (Mor g) = Mor (\x => f x (g x))
  aid (Mor x) = Refl
  ahom g x = Refl
  aint (Mor f) y = Refl
  acomp (Mor u) (Mor v) (Mor w) = Refl

plusZeroRightId : (n : Nat) -> n + 0 = n
plusZeroRightId Z = Refl
plusZeroRightId (S n) = rewrite plusZeroRightId n in Refl

vectPlusZero : {n : Nat} -> Vect (plus n 0) a -> Vect n a
vectPlusZero xs = replace {p = \prf => Vect prf a} (plusZeroRightId n) xs

-- As with lists, length indexed vectors are functors
public export
implementation {n:Nat} -> VFunctor (Vect n) where
  fmap f [] = []
  fmap f (x::xs) = f x :: fmap f xs
  fid [] = Refl
  fid (x::xs) = cong (x::) (fid xs)
  fcomp [] g h = Refl
  fcomp (x::xs) g h = cong (g (h x) ::) (fcomp xs g h)
  infixSame f x = Refl

-- Binary trees are functors
public export
data BTree : Type -> Type where
  Null : BTree a
  Node : BTree a -> a -> BTree a -> BTree a

public export
implementation VFunctor BTree where
  -- fmap maps f recursively over the values in every node
  fmap f Null = Null
  fmap f (Node l x r) = Node (fmap f l) (f x) (fmap f r)
  fid Null = Refl
  fid (Node l x r) =
    let iH1 = fid l
        iH2 = fid r
    in rewrite iH1
    in rewrite iH2
    in Refl
  fcomp Null g h = Refl
  fcomp (Node l x r) g h =
    let iH1 = fcomp l g h
        iH2 = fcomp r g h
    in rewrite iH1
    in rewrite iH2
    in Refl
  infixSame f x = Refl

-- Rose trees are functors
public export
```

```
data RTree : Type -> Type where
  Leaf : a -> RTree a
  Branch : List (RTree a) -> RTree a

-- These are for VFunctor RTree but had to be pulled out so `branches`
-- could be used in a proof about fmap as well as fmap
mutual
  branches : (a -> b) -> List (RTree a) -> List (RTree b)
  branches f [] = []
  branches f (b::bs) = fmapRTree f b :: branches f bs

  fmapRTree : (a -> b) -> (RTree a) -> (RTree b)
  fmapRTree f (Leaf x) = Leaf (f x)
  fmapRTree f (Branch bs) = Branch (branches f bs)

public export
implementation VFunctor RTree where
  fmap = fmapRTree
  fid (Leaf x) = Refl
  fid (Branch bs) = cong Branch (prf bs) where
    prf : (bs : List (RTree a)) -> branches (\x => x) bs = bs
    prf [] = Refl
    prf (b::bs) = rewrite prf bs in cong (::bs) (fid b)
  fcomp (Leaf x) g h = Refl
  fcomp (Branch bs) g h = cong Branch (prf bs g h) where
    prf : (bs : List (RTree a)) -> (g : b -> c) -> (h : a -> b)
      -> (branches (g . h) bs = branches g (branches h bs))
    prf [] g h = Refl
    prf (b::bs) g h = rewrite prf bs g h
      in cong (:: branches g (branches h bs)) (fcomp b g h)
  infixSame f x = Refl
```

## Verified Profunctors: VProfunctor.idr

```
module Category.VProfunctor

import Category.VFunctor
import Category.Morphism

%default total
%hide Applicative

-- Verified profunctors
public export
interface VProfunctor (p : Type -> Type -> Type) where
  -- dimap maps two morphisms over a profunctor
  -- p(a,-) is a covariant functor, p(-,a) is contravariant
  dimap : (a -> b) -> (c -> d) -> p b c -> p a d

  -- Identity law, dimap id id = id
  pid : {a, b : Type} -> (x : p a b) -> dimap (\x => x) (\x => x) x = x
  -- Composition law, dimap (f' . f) (g . g') = dimap f g . dimap f' g'
  pcomp
    : {a, b, c, d, e, t : Type}
    -> (x : p a b)
    -> (f' : c -> a) -> (f  : d -> c)
    -> (g :  e -> t) -> (g' : b -> e)
    -> dimap (f' . f) (g . g') x = (dimap f g . dimap f' g') x

-- Profunctors for product and sum types, and monoidal profunctors

-- Cartesianly strong profunctors preserve product types
public export
interface VProfunctor p => Cartesian p where
  first  : p a b -> p (a, c) (b, c)
  second : p a b -> p (c, a) (c, b)

-- Co-Cartesianly strong profunctors preserve sum types
public export
interface VProfunctor p => Cocartesian p where
  left  : p a b -> p (Either a c) (Either b c)
  right : p a b -> p (Either c a) (Either c b)

-- Profunctors with monoid object structure
public export
interface VProfunctor p => Monoidal p where
  par   : p a b -> p c d -> p (a, c) (b, d)
  empty : p () ()

-- Profunctor implementations

-- Hom(-,-) profunctor, the canonical profunctor
public export
implementation VProfunctor Morphism where
  dimap f g (Mor h) = Mor (g . h . f)
  pid (Mor f) = cong Mor (sym (ext f))
  pcomp (Mor x) f' f g g' = Refl

public export
implementation Cartesian Morphism where
  first (Mor f) = Mor (\(a, c) => (f a, c))
  second (Mor f) = Mor (\(c, a) => (c, f a))

public export
implementation Cocartesian Morphism where
  left (Mor f) = Mor (\case
    Left a => Left (f a)
    Right c => Right c)
  right (Mor f) = Mor (\case
    Left c => Left c
    Right a => Right (f a))

public export
implementation Monoidal Morphism where
  par (Mor f) (Mor g) = Mor (\(x, y) => (f x, g y))
  empty = Mor (const ())

-- Hom profunctor in the Kleisli category
-- This is the category of monadic types `m a` with Kleisli composition
-- f . g = \x => join (f (g x)), where join : m (m a) -> m a
-- We only require a functor for convenience
public export
implementation {k : Type -> Type} -> VFunctor k => VProfunctor
 ↪  (KleisliMorphism k) where
  dimap f g (Kleisli h) = Kleisli (fmap g . h . f)
  -- This proof reduces to `fmap (\x => x) . f = f` for `f : a -> k b`
  -- We can't make `fid` intensional, ie `fid : fmap (\x => x) = id`,
  -- because we need something to pattern match on to prove fid, so we must
  ↪  use
  -- extensionality here
  pid (Kleisli f) = cong Kleisli (extensionality (\x => fid (f x)))
  pcomp (Kleisli u) f' f g g' = cong Kleisli (extensionality (\x =>
    fcomp (u (f' (f x))) g g'))

-- Const profunctor, Const r a is isomorphic to Hom((), a)
-- This profunctor allows us to use our optics as constructors
-- eg: op {p=Const} (MkConst 3) == MkConst (Just 3)
public export
record Const r a where
  constructor MkConst  -- MkConst : a -> Const r a
  unConst : a          -- unConst : Const r a -> a

public export
implementation VProfunctor Const where
  dimap f g (MkConst x) = MkConst (g x)
  pid (MkConst x) = Refl
  pcomp (MkConst x) f' f g g' = Refl

public export
implementation Cocartesian Const where
  left (MkConst x) = MkConst (Left x)
  right (MkConst x) = MkConst (Right x)

public export
implementation Monoidal Const where
  par (MkConst x) (MkConst y) = MkConst (x, y)
  empty = MkConst ()

-- `Forget r` profunctor
-- Allows us to use our profunctor optics as getters
-- eg: unForget (π₁ {p=Forget Int} (MkForget (\x => x))) (3, True) == 3
-- Inspired by PureScript's profunctor-lenses:
-- https://github.com/purescript-contrib/purescript-profunctor-lenses/
public export
record Forget r a b where
  constructor MkForget  -- MkForget : (a -> r) -> Forget r a b
  unForget : a -> r     -- unForget : Forget r a b -> (a -> r)

public export
implementation {r : Type} -> VProfunctor (Forget r) where
  dimap f g (MkForget h) = MkForget (h . f)
  pid (MkForget x) = Refl
  pcomp (MkForget x) f' f g g' = Refl

public export
implementation {r : Type} -> Cartesian (Forget r) where
  first (MkForget f) = MkForget (\(x, y) => f x)
  second (MkForget f) = MkForget (\(x, y) => f y)
```

Profunctor Optics: Main.idr

```
module Main

import Category.VProfunctor
import Category.VFunctor
import Category.Morphism
import Primitive.PrimitiveOptics
import Data.Vect

%default total
%hide Prelude.Interfaces.(<*>)
%hide Prelude.Interfaces.(<$>)

infixr 0 ~>

-- Profunctor optic types

Optic : (Type -> Type -> Type) -> Type -> Type -> Type -> (Type -> Type)
Optic p a b s t = p a b -> p s t

Adapter : Type -> Type -> Type -> Type -> Type
Adapter a b s t = {p : Type -> Type -> Type} -> VProfunctor p => Optic p a b s
 ↪  t

Lens : Type -> Type -> Type -> Type -> Type
Lens a b s t = {p : Type -> Type -> Type} -> Cartesian p => Optic p a b s t

Prism : Type -> Type -> Type -> Type -> Type
Prism a b s t = {p : Type -> Type -> Type} -> Cocartesian p => Optic p a b s t

LensPrism : Type -> Type -> Type -> Type -> Type
LensPrism a b s t = {p : Type -> Type -> Type}
  -> (Cartesian p, Cocartesian p)
  => Optic p a b s t

Traversal : Type -> Type -> Type -> Type -> Type
Traversal a b s t = {p : Type -> Type -> Type}
  -> (Cartesian p, Cocartesian p, Monoidal p)
  => Optic p a b s t

-- Product type optics

-- π₁ : {p : Type -> Type -> Type} -> Cartesian p => p a b -> p (a, c) (b, c)
π₁ : Lens a b (a, c) (b, c)
π₁ = first

```

```
π₂ : Lens a b (c, a) (c, b)
π₂ = second

-- Optional type optics

-- op : {p : Type -> Type -> Type} -> Cocartesian p => p a b -> p (Maybe a)
 ↪  (Maybe b)
op : Prism a b (Maybe a) (Maybe b)
op = dimap (maybe (Left Nothing) Right) (either id Just) . right

-- Sum/coproduct type optics

leftP : Prism a b (Either a c) (Either b c)
leftP = left

rightP : Prism a b (Either c a) (Either c b)
rightP = right

-- Example of composition of optics

op_π₁ : LensPrism a b (Maybe (a, c)) (Maybe (b, c))
op_π₁ = op . π₁

-- Map primitive optics to profunctor optics

prismFromPrim : PrimPrism a b s t -> Prism a b s t
prismFromPrim (MkPrimPrism m b) = dimap m (either id b) . right

-- Complex data structures

-- This type is from van Laarhoven
-- https://twanvl.nl/blog/haskell/non-regular1
-- FunList a b t is isomorphic to ∃n. a^n × (b^n -> t)
-- which is equivalent to the type of a traversable (Pickering et. al. 2018)
-- It allows us to write optics for lists and trees
data FunList : Type -> Type -> Type -> Type where
  Done : t -> FunList a b t
  More : a -> FunList a b (b -> t) -> FunList a b t

out : FunList a b t -> Either t (a, FunList a b (b -> t))
out (Done t) = Left t
out (More x l) = Right (x, l)

inn : Either t (a, FunList a b (b -> t)) -> FunList a b t
inn (Left t) = Done t
inn (Right (x, l)) = More x l

implementation {a : Type} -> {b : Type} -> VFunctor (FunList a b) where
  fmap f (Done t) = Done (f t)
  fmap f (More x l) = More x (fmap (f .) l)
  fid (Done t) = Refl
  fid (More x l) = cong (More x) (fid l)
  fcomp (Done t) g h = Refl
  fcomp (More x l) g h = cong (More x) (fcomp l (g .) (h .))
  infixSame f x = Refl

implementation {a : Type} -> {b : Type} -> VApplicative (FunList a b) where
  ret = Done
  Done f <*> l = fmap f l
  More x l <*> l2 = assert_total More x (fmap flip l <*> l2)
  aid (Done t) = Refl
  aid (More x l) = cong (More x) (aid l)
  ahom g x = Refl
  aint u y = believe_me () -- todo
  acomp u v w = believe_me ()

single : a -> FunList a b b
single x = More x (Done id)

fuse : FunList b b t -> t
fuse (Done t) = t
fuse (More x l) = fuse l x

traverse : {p : Type -> Type -> Type} -> (Cocartesian p, Monoidal p)
  => p a b
  -> p (FunList a c t) (FunList b c t)
traverse k = assert_total dimap out inn (right (par k (traverse k)))

makeTraversal : (s -> FunList a b t) -> Traversal a b s t
makeTraversal h k = dimap h fuse (traverse k)

-- Binary tree traversals

inorder' : {f : Type -> Type} -> VApplicative f
  => (a -> f b)
  -> BTree a -> f (BTree b)
inorder' m Null = ret Null
inorder' m (Node l x r) = Node <$> inorder' m l <*> m x <*> inorder' m r

inorder : {a, b : Type} -> Traversal a b (BTree a) (BTree b)
inorder = makeTraversal (inorder' single)

preorder' : {f : Type -> Type} -> VApplicative f
  => (a -> f b)
  -> BTree a -> f (BTree b)
preorder' m Null = ret Null
preorder' m (Node l x r) =
  (\mid, left, right => Node left mid right) <$>
    m x <*> preorder' m l <*> preorder' m r

preorder : {a, b : Type} -> Traversal a b (BTree a) (BTree b)
preorder = makeTraversal (preorder' single)

postorder' : {f : Type -> Type} -> VApplicative f
  => (a -> f b)
  -> BTree a -> f (BTree b)
postorder' m Null = ret Null
postorder' m (Node l x r) =
  (\left, right, mid => Node left mid right) <$>
    postorder' m l <*> postorder' m r <*> m x

postorder : {a, b : Type} -> Traversal a b (BTree a) (BTree b)
postorder = makeTraversal (postorder' single)

-- List traversals

listTraverse' : {f : Type -> Type} -> VApplicative f
  => (a -> f b)
  -> List a -> f (List b)
listTraverse' g [] = ret []
listTraverse' g (x::xs) = (::) <$> g x <*> listTraverse' g xs

listTraverse : {a, b : Type} -> Traversal a b (List a) (List b)
listTraverse = makeTraversal (listTraverse' single)

-- PrimPrism a b forms a Cocartesian profunctor

-- Definitions and lemmas from the Either bifunctor for `VProfunctor
 ↪  (PrimPrism a b)`
bimapEither : (a -> c) -> (b -> d) -> Either a b -> Either c d
bimapEither f g (Left x) = Left (f x)
bimapEither f g (Right x) = Right (g x)

bimapId : (z : Either a b) -> bimapEither (\x => x) (\x => x) z = z
bimapId (Left y) = Refl
bimapId (Right y) = Refl

bimapLemma : (g :  e -> t) -> (g' : b -> e) -> (x' : Either b a)
  -> bimapEither (g . g') (\x => x) x' = bimapEither g (\x => x) (bimapEither
   ↪  g' (\x => x) x')
bimapLemma g g' (Left x) = Refl
bimapLemma g g' (Right x) = Refl

public export
implementation {a : Type} -> {b : Type} -> VProfunctor (PrimPrism a b) where
  dimap f g (MkPrimPrism m b) = MkPrimPrism (bimapEither g id . m . f) (g . b)
  pid (MkPrimPrism m b) = cong (`MkPrimPrism` b)
    (extensionality (\x => bimapId (m x)))
  pcomp (MkPrimPrism m b) f' f g g' = cong (`MkPrimPrism` (\x => g (g' (b
   ↪  x))))
    (extensionality (\x => bimapLemma g g' (m (f' (f x)))))

public export
implementation {a : Type} -> {b : Type} -> Cocartesian (PrimPrism a b) where
  left (MkPrimPrism m b) = MkPrimPrism (either (bimapEither Left id . m) (Left
   ↪  . Right)) (Left . b)
  right (MkPrimPrism m b) = MkPrimPrism (either (Left . Left) (bimapEither
   ↪  Right id . m)) (Right . b)

-- Helpful combinators

-- `Forget r` profunctor optics operate as getters
```

```
view : {a : Type} -> Lens a b s t -> s -> a
view optic x = unForget (optic {p=Forget a} (MkForget (\x => x))) x

-- Morphism profunctor optics operate as setters
update : Optic Morphism a b s t -> (a -> b) -> (s -> t)
update optic f x = applyMor (optic (Mor f)) x

-- Const profunctor optics recovers sum type constructors
build : Prism a b s t -> b -> t
build optic x = unConst (optic {p=Const} (MkConst x))

-- Unit tests (if these fail we get type errors)
-- These are provided as examples of how to use these profunctor optics in
↪  practice

test1 : update (Main.op . π₁) (\x => x * x) (Just (3, True)) = Just (9, True)
test1 = Refl

test2 : view π₁ (3, True) = 3
test2 = Refl

test3 : build Main.op 3 = Just 3
test3 = Refl

-- view π₁ = fst (extensionally)
forgetLeftProjection : (x : r) -> (y : b)
  -> fst (x, y) = view π₁ (x, y)
forgetLeftProjection x y = Refl

-- build op = Just (extensionally)
constBuildsMaybe : (x : a)
  -> Just x = build Main.op x
constBuildsMaybe x = Refl
```