Optics and Type Equivalences

Abstract

Bidirectional programming, lenses, prisms, and other optics have connections to reversible programming which have been explored from several perspectives, mostly by attempting to recover bidirectional transformations from unidirectional ones. We offer a novel and foundational perspective in which reversible programming is expressed using "type equivalences." This perspective offers several advantages: first, it is possible to construct sets of sound and complete type equivalences for certain collections of types; these correspond to canonical optic constructions. Second, using ideas inspired by category theory and homotopy type theory, it is possible to construct sound and complete "equivalences between equivalences" which provide the canonical laws for reasoning about lens and prism equivalences.

1 Introduction

The notion of lenses (and its generalizations to optics) is now established as one of the formalisms for bidirectional transformations [1]. These optics are generally studied in the context of conventional programming languages (e.g., Java, Haskell, etc.) which leaves untapped the richness of a dependently-typed language, especially one which directly supports programming with proof-relevant type equivalences. (See however the characterization of bidirectional transformations as proof-relevant bisimulations [1] for a closely related perspective.)

In this paper, we show that in the context of a programming language for proof-relevant type equivalences, the many constructions of optics and more importantly, their correctness and their laws, become simple consequences of the general properties of proof-relevant type equivalences. In particular, we formalize the intuitive, but informal, constructions and laws, in various sources [10, 18, 21].

We start in the next section with the conventional definition of lenses using a pair of *very well-behaved* set/get functions. That definition is only implicitly related to type equivalences via a hidden *constant-complement*. In order to expose the underlying type equivalence, we first reformulate the definition of lenses using an existential record that packages an unknown but fixed complement type. That definition, however, turns out to have weak proof-theoretic properties. We therefore introduce our final definition of lenses using the notion of *setoid* to formalize the correct equivalence relation on the source type of the lens. We present

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

a complete formalized proof in Agda that this final definition is sound and complete with respect to the conventional set/get definition.

With a formulation of lenses based on proof-relevant type-equivalences in hand, we aim to show that many variants of lenses, as well as other optics (prisms, etc.), are directly expressible, and more importantly, that their laws are immediately derivable. In order to do that, however, we first need, a language in which to express type equivalences as well as proofs between type equivalences. In previous work, we have established that if we restrict ourselves to finite types constructed from the empty type, the unit type, the sum type, and the product type, then it is possible to formulate a two-level language with the following properties. The programs at level-1 in the language are sound and complete type equivalences, and the programs at level-2 are sound and complete proofs of equivalences between the level-1 programs (see Sec. 3). This setting of finite types thus provides us with a framework in which to define canonical optics with their properties (see Sec. 4). (In the presence of richer types, lenses and their properties can still be expressed but we generally lose guarantees of completeness.) In Sec. 5, we show that the framework is robut and generalizes to prisms and other less common optics. We finish with a short discussion putting our work in context and conclude.

2 Lenses

A *lens* is a structure that mediates between a source S and view A. Typically a lens comes equipped with two functions *get* which projects a view from a source, and *set* which takes a source and a view and reconstructs an appropriate source with that view. A monomorphic interface for such lenses is shown below, including the commonly cited laws for the lens to be very well-behaved:

```
record GS-Lens {ℓs ℓa : Level} (S : Set ℓs) (A : Set ℓa) : Set (ℓs ⊔ ℓa) where
  field
    get    : S → A
    set    : S → A → S
    getput : (s : S) (a : A) → get (set s a) ≡ a
    putget : (s : S) → set s (get s) ≡ s
    putput : (s : S) (a a' : A) → set (set s a) a' ≡ set s a'
open GS-Lens
```

A common theme in the literature on lenses is that the function *get* discards some information from the source to create a view, and that this information can be explicitly represented using the *constant-complement* technique from the database literature. In other words, lenses can viewed as elements of $\exists\ C.S \simeq C \times A$ where $\simeq$ is type equivalence.

This observation is what connects lenses to type equivalences and hence to reversible programming. The main contribution of the paper is to exploit various canonical constructions and completness results in the world of reversible programming and export them to the world of bidirectional programming with lenses (and other optics).

Although correct in principle, a straightforward encoding of *constant-complement lenses* as $\Sigma\ C.S \simeq C \times A$ is not satisfactory: a GS-Lens does not reveal any sort of complement C; so the constant-complement lenses should not either. To do this, we should somehow hide our choice of C. We could use a variety of tricks to do this, but all would rely on features of Agda which do not have well-understood meta-theory. Instead, we will rely on *discipline* to not access the actual C. Note that because Set ℓ does not allow introspection, actually getting one's hands on this C still does not reveal very much!

We can use the formulation $\exists\ C.S \simeq C \times A$ as the basis for a first definitions of isomorphism-based lens. We make C implicit, so as to reduce the temptation to examine it. This formulation will not be entirely adequate, but is very close to our final definition.

```
record Lens₁ {ℓ : Level} (S : Set ℓ) (A : Set ℓ) : Set (suc ℓ) where
  constructor ∃-lens
```

```
  field
    {C} : Set ℓ
    iso : S ≃ (C × A)
```

Given an Lens_1, we can build a GS-Lens, so that this is certainly sound:

```
sound : {ℓ : Level} {S A : Set ℓ} → Lens₁ S A → GS-Lens S A
sound (∃-lens (f , qinv g α β)) = record
  { get = λ s → proj₂ (f s)
  ; set = λ s a → g (proj₁ (f s) , a)
  ; getput = λ s a → cong proj₂ (α _)
  ; putget = λ s → β s
  ; putput = λ s a a' → cong g (cong₂ _,_ (cong proj₁ (α _)) P.refl) }
```

It is important to notice that the conversion above only uses the iso part of the Lens_1.

The question is, is this *complete*, in the sense that we can also go in the other direction? To achieve this, we must manufacture an appropriate constant complement. We certainly know what S contains all the necessary information for this but is in some sense "too big". But it is instructive to see what happens when we try.

Roughly speaking the forward part of the isomorphism is forced: given an $s : S$, there is only one way to get an A, and that is via get. To get an S back, there are two choices: either use s itself, or call set; the laws of GS-Lens say that either will actually work. In the backwards direction of the isomorphism, the laws help in narrowing down the choices: basically, we want the $s' : S$ where get s′ ≡ a, and so we again use set for the purpose:

```
incomplete : {ℓ : Level} {S A : Set ℓ} → GS-Lens S A → Lens₁ S A
incomplete {ℓ} {S} {A} l =
  ∃-lens ((λ s → s , get l s) ,
          qinv (λ { (s , a) → set l s a })
               (λ { (s , a) → cong₂ _,_ hole (getput l s a)})
                 λ s → putget l s)
```

That almost gets us there. The one hole we can't fill says

```
  where
    hole : {s : S} {a : A} → set l s a ≡ s
    hole = {!!}
```

But that will only ever happen if get s was already a (by putget).

Of course, we already knew this would happen: S is too big. Basically, it is too big by exactly the inverse image of A by get.

Thus our next move is to make that part of S not matter. In other words, rather than using the *type* S as a proxy, we want to use a Setoid where $s, t : S$ will be regarded as the same if they only differ in their A component. It is convenient to also define a function lens that lifts type isomorphisms (which work over propositional equality) to the Setoid setting.

```
record ∃-Lens {a s : Level} (S : Set s) (A : Set a) : Set (suc (a ⊔ s)) where
  constructor ll
  field
    {C} : Setoid s a
    iso : Inverse (P.setoid S) (C ×S (P.setoid A))

lens : {ℓ : Level} {S A C : Set ℓ} → S ≃ (C × A) → ∃-Lens S A
```

```
lens {C = C} (f , qinv g α β) = ll {C = P.setoid C} (record
  { to = record { _⟨$⟩_ = f ; cong = λ { P.refl → P.refl , P.refl} }
  ; from = record { _⟨$⟩_ = g ; cong = λ { (P.refl , P.refl) → P.refl } } -- η for × crucial
  ; inverse-of = record
    { left-inverse-of = β
    ; right-inverse-of = λ { (c , a) → (cong proj₁ (a _)) , cong proj₂ (a _)}
    }
  })
```

One important aspect of the proof is that not only are both laws α and β for the isomorphism used, but η for pairs is also crucial.

The soundness proof is then essentially identical to the previous one:

```
sound′ : {ℓ : Level} {S A : Set ℓ} → ∃-Lens S A → GS-Lens S A
sound′ {S = S} {A} (ll len) =
  let f = to len ⟨$⟩_
      g = from len ⟨$⟩_
      a = right-inverse-of len
      β = left-inverse-of len in
  record
  { get = λ s → proj₂ (f s)
  ; set = λ s a → g (proj₁ (f s) , a)
  ; getput = λ s a → proj₂ (a (proj₁ (f s) , a))
  ; putget = β
  ; putput = λ s a _ → Π.cong (from len) (proj₁ (a (proj₁ (f s) , a)) , P.refl) }
```

And now the completeness proof goes through. Key is to create an equivalence relation $\approx$ between sources s, t which makes them "the same" if they only differ in their A component.

```
complete : {ℓ : Level} {S A : Set ℓ} → GS-Lens S A → ∃-Lens S A
complete {ℓ} {S} {A} l = ll
  {C = record
    { Carrier = S
    ; _≈_ = λ s t → ∀ (a : A) → set l s a ≡ set l t a
    ; isEquivalence = record { refl = λ _ → P.refl ; sym = λ i≈j a → P.sym (i≈j a)
                             ; trans = λ i≈j j≈k a → P.trans (i≈j a) (j≈k a) } }}
    (record
      { to = record { _⟨$⟩_ = λ s → s , get l s ; cong = λ { refl → (λ _ → P.refl) , P.refl } }
      ; from = record { _⟨$⟩_ = λ {(s , a) → set l s a} ; cong = λ { {_ , a₁} (≈ , P.refl) → ≈ a₁ } }
      ; inverse-of = record
          { left-inverse-of = putget l
          ; right-inverse-of = λ { (s , a) → (λ a' → putput l s a a') , getput l s a}
          }
      })
```

Grenrus [10] gives a completely different type which also works — but the completeness proofs requires both proof irrelevance and function extensionality (crucially), while our proof works in a much simpler setting.

$$
\begin{array}{rcl}
a & = & a \\
\\
0 + a & = & a \\
a + b & = & b + a \\
a + b + c & = & a + b + c \\
\\
1 \cdot a & = & a \\
a \cdot b & = & b \cdot a \\
a \cdot b \cdot c & = & a \cdot b \cdot c \\
\\
0 \cdot a & = & 0 \\
a + b \cdot c & = & a \cdot c + b \cdot c
\end{array}
\qquad
\begin{array}{rcl}
A & \simeq & A \\
\\
\bot \uplus A & \simeq & A \\
A \uplus B & \simeq & B \uplus A \\
A \uplus B \uplus C & \simeq & A \uplus B \uplus C \\
\\
\top * A & \simeq & A \\
A * B & \simeq & B * A \\
A * B * C & \simeq & A * B * C \\
\\
\bot * A & \simeq & \bot \\
A \uplus B * C & \simeq & A * C \uplus B * C
\end{array}
$$

Figure 1. Semiring equalities and type isomorphisms.

3 Proof-relevant type equivalences

Our principal means of building lenses, lens, takes as input a *type equivalence*. These are called *proof relevant* because different witnesses (proofs) of an equivalence are not assumed to be the same. For example, there are two non-equivalent ways to prove that $A \times A \simeq A \times A$, namely the identity and "swap".

Our starting point will be a basic type theory with the empty type ($\bot$), the unit type ($\top$), the sum type ($\uplus$), and the product ($*$) type. But rather than focusing on *functions* between these types, we will instead look at *equivalences*.

3.1 Type Equivalences

The Curry-Howard correspondence teaches that logical expressions form a commutative semiring – and leads us to expect that types too form a commutative semiring. And indeed they do – at least up to *type isomorphism*. The reader unfamiliar with these can find a leisurely introduction in [5]. We will furthermore assume that the reader is already familiar with the basic definitions around *type equivalences*. That types, with $(\bot, \top, \uplus, *)$ interpreted as $(0, 1, +, \times)$ and strict equality replaced with equivalence $\simeq$ form a commutative semiring is a basic result of type theory.

However, we might be misled by the Curry-Howard correspondence: In logic, it is true that $A \vee A = A$ and $A \wedge A = A$. However, neither $A \uplus A$ nor $A * A$ are equivalent to A. They are however *equi-inhabited*. This is a fancy way of saying

$$A \uplus A \text{ is inhabited} \qquad \Leftrightarrow \qquad A \text{ is inhabited}$$

The above is the real *essence* of the Curry-Howard correspondence. In other words, classical Curry-Howard tells us about *logical equivalence* of types; there are indeed functions $f : A \uplus A \to A$ and $g : A \to A \uplus A$; however, they are not inverses.

The generators for our type isomorphisms will exactly be those of a semiring — we place them side-by-side in Fig. 1. Each is also named — the details can be found both in [4, 6] and in the online repository http://github.com/JacquesCarette/pi-dual (in file `Univalence/TypeEquiv.agda`). There, a programming language named Π is created to denote type isomorphisms.

This set of isomorphisms is known to be sound and complete [7, 8] for isomorphisms of finite types. Furthermore, it is also universal for hardware combinational circuits [12].

3.2 Examples

We can express a 3-bit word reversal operation as follows:

$$reverse : word_3 \leftrightarrow word_3$$
$$reverse = swap_\times \odot swap_\times \otimes id\leftrightarrow \odot assocr_\times$$

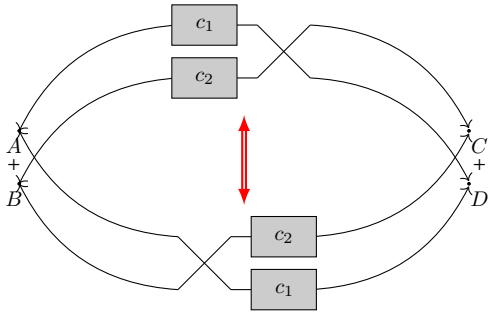
We can check that *reverse* does the right thing by applying it to a value v_1, v_2, v_3 and writing out the full reduction, which can be visualized as:

$$\begin{array}{rl} & v_1, v_2, v_3 \\ swap_\times & v_2, v_3, v_1 \\ swap_\times \otimes id\!\leftrightarrow & v_3, v_2, v_1 \\ assocr_\times & v_3, v_2, v_1 \end{array}$$

There are several universal primitives for conventional (irreversible) hardware circuits, such as *nand* and *fanout*. In the case of reversible hardware circuits, the canonical universal primitive is the Toffoli gate [20]. The Toffoli gate takes three boolean inputs: if the first two inputs are *true* then the third bit is negated. The Toffoli gate, and its simple cousin the *cnot* gate, are both expressible in the programming language Π.

3.3 Equivalences between Equivalences

Just as types can be shown equivalent, type isomorphisms also induce a "higher dimensional" set of equivalences. To illustrate, consider two equivalences that both map between the types $A + B$ and $C + D$:



The top path is $c_1 \oplus c_2 \odot swap_+$ and the bottom path $swap_+ \odot c_2 \oplus c_1$. These are equivalent – and in fact denote the same permutation. And, of course, not all programs between the same types are equivalent. The simplest example are the two automorphisms of $1 + 1$, namely $id\!\leftrightarrow$ and $swap_+$.

The language of type isomorphisms and equivalences between them has a strong connection to *univalent universes* in HoTT [4]. Based on this connection, we refer to the types as being at level-0, to the equivalences between types as being at level-1, and to the equivalences between equivalences of types (i.e., this subsection) as being at level-2.

The basic type equivalences were defined by using all the proof terms of commutative semirings. What we need now is to understand how *proofs* of algebraic identities should be considered equivalent. Classical algebra does not help, as proofs are not considered first-class citizens. However, another route is available to us: since the work of Hofmann and Streicher [11], we know that one can model types as *groupoids*. The additional structure comes from explicitly modeling the "identity types": instead of regarding all terms which witness the equality of (say) a and b of type A as being indistinguishable, we posit that there may in fact be many, i.e. proof relevance.

Thus, rather than looking at (untyped) commutative semirings, we should look at a *typed* version. This process frequently goes by the moniker of "categorification." We want a categorical algebra, where the basic objects are groupoids (to model our types), and where there is a natural notion of $+$ and $*$. At first, we hit what seems like a serious stumbling block: the category of all groupoids, Groupoid, have neither co-products nor products. However, we don't want to work internally in Groupoid– we want operations *on* groupoids. In other words, we want something akin to symmetric monoidal categories, but with two interacting monoidal structures. Luckily, this already exists: the categorical analog to (commutative) semirings are (symmetric) Rig Categories [14, 16]. This straightforwardly generalizes to symmetric Rig Groupoids.

How does this help? Coherence conditions! Symmetric monoidal categories, to start somewhere simple, do not just introduce natural transformations like the associator α and the left and right unitors (λ and ρ

respectively), but also coherence conditions that these must satisfy. Looking, for example, at just the additive fragment (i.e. with just 0, 1 and + for the types, $\odot$ and $\oplus$ as combinators, and only the terms so expressible), the sub-language would correspond, denotationally, to exactly symmetric monoidal groupoids. And here we have *equations between equations*, aka commutative diagrams. Transporting these coherence conditions, for example those that express that various transformations are *natural*, to our setting gives a list of equations between isomorphisms. Furthermore, all the natural transformations that arise are in fact natural *isomorphisms* – and thus reversible.

We can in fact prove that all the coherence conditions of symmetric Rig Groupoids holds for the groupoid interpretation of types [6]. This is somewhat tedious given the sheer numbers involved, but when properly formulated, relatively straightforward, up to a couple of tricky cases.

But why are these particular coherence laws? Are they all necessary? Conversely are they, in some appropriate sense, sufficient? This is the so-called *coherence problem*. Mac Lane, in his farewell address as President of the American Mathematical Society [17] gives a good introduction and overview of such problems. A more modern interpretation (which can nevertheless be read into Mac Lane's own exposition) would read as follows: given a set of equalities on abstract words, regarded as a rewrite system, and two means of rewriting a word in that language to another, is there some suitable notion of canonical form that expresses the essential uniqueness of the non-trivial rewrites? Note how this word-and-rewrite problem is essentially independent of the eventual interpretation. But one must take some care, as there are obvious degenerate cases (involving "trivial" equations involving 0 or 1) which lead to non-uniqueness. The landmark results, first by Kelly-Mac Lane [13] for closed symmetric monoidal categories, then (independently) Laplaza and Kelly [14, 16] for symmetric Rig Categories, is that indeed there are sound and complete coherence conditions that insure that all the "obvious" equalities between different abstract words in these systems give rise to commutative diagrams. The "obvious" equalities come from *syzygies* or *critical pairs* of the system of equations. The problem is far from trivial — Fiore et al. [9] document some publications where the coherence set is in fact incorrect. They furthermore give a quite general algorithm to derive such coherence conditions.

4 Exploring the Lens landscape

Given that we have a sound and complete set of primitive type equivalences (and combinators), we can explore what this means for actually programming lenses. Many papers have explored the most general settings for lenses, we will instead look inside the implementations. This will reveal the *inner structure* of lenses, rather than focusing on their macro structure.

4.1 Simple Lenses

Let's explore the simplest lenses first. For a GS-Lens, the simplest is when get is the identity, which forces the rest:

```
module _ (A B D E : Set) where
  open ∃-Lens

  AA-gs-lens : GS-Lens A A
  AA-gs-lens = record { get = id ; set = λ _ → id
    ; getput = λ _ _ → P.refl ; putget = λ _ → P.refl ; putput = λ _ _ _ → P.refl }
```

What does that correspond to as a ∃-Lens? Here, we can easily guess the complement by solving the equation $A \simeq C \times A$ for C: C must be $\top$. But then the ∃-Lens isn't quite as simple as above:

```
AA-∃-lens : ∃-Lens A A
AA-∃-lens = lens uniti⋆equiv
```

where uniti⋆equiv has type $A \simeq \top \times A$. In other words, as the complement is not actually present in A, it must be introduced. uniti⋆equiv names the "multiplicative unit introduction equivalence". From here on, we will not expand on the names, trusting that they can be guessed by the reader.

What about in the other direction, what is the ∃-Lens whose underlying isomorphism is the identity?

```
BAA-∃-lens : ∃-Lens (B × A) A
BAA-∃-lens = lens id≃
```

Since our definition of ∃-Lens is right-biased (we are looking for isomorphisms of shape $S \simeq C \times A$), the above lens extracts the A on the right. Of course, there is another lens which switches the roles of A and B — and this leaves a trace on the isomorphism:

```
BAB-∃-lens : ∃-Lens (B × A) B
BAB-∃-lens = lens swap⋆equiv
```

Thus, looking at type equivalences, which ones return a type of shape $C \times A$? We have already seen uniti⋆l, id⟷ and swap⋆ arise. That leaves four: assocl⋆, factorz, factor and ×≃. These occur as follows:

```
DBA-lens : ∃-Lens (D × (B × A)) A
DBA-lens = lens assocl⋆equiv

⊥-lens : ∃-Lens ⊥ A
⊥-lens = lens factorzequiv

⊎-lens : ∃-Lens ((D × A) ⊎ (B × A)) A
⊎-lens = lens factorequiv

⊗-lens : (E ≃ B) → (D ≃ A) → ∃-Lens (E × D) A
⊗-lens iso₁ iso₂ = lens (iso₁ ×≃ iso₂)
```

The first is a basic administrative "reshaping". The second takes a bit more thought, but is easily explained: if we promise an impossible source, it is easy to promise to return something arbitrary in return!

The ⊎-lens is interesting, because it allows us to see a constant complement in a type which itself is not a product – it is, however, equivalent to one. The last uses the full power of equivalences, to see an A where, a priori, one does not seem to exist at all.

Lastly, we also have lens composition:

```
∘-lens : ∃-Lens D B → ∃-Lens B A → ∃-Lens D A
∘-lens l₁ l₂ = ll ((×-assoc ∘F (idF ×-inverse ∃-Lens.iso l₂)) ∘F ∃-Lens.iso l₁)
```

The above gives us our first *lens program* consisting of a composition of four more basic equivalences. However, it is "lower level" as we can only extract Setoid-based equivalences from a ∃-Lens. The necessary code is quite straightforward (and available in the literate Agda source of this paper).

4.2 Unusual lenses

It is possible to create lenses for things which are not "in" a type at all — an example is most instructive. For completeness, both GS-Lens and ∃-Lens will be given.

Let us consider a type `Colour` with exactly 3 inhabitants,

```
module _ {A : Set} where
  data Colour : Set where red green blue : Colour
```

First, a ∃-Lens built "by hand":

```
∃-Colour-in-A+A+A : ∃-Lens (A ⊎ A ⊎ A) Colour
∃-Colour-in-A+A+A = lens eq
  where
    f : A ⊎ A ⊎ A → A × Colour
    f (inj₁ x) = x , red
    f (inj₂ (inj₁ x)) = x , green
    f (inj₂ (inj₂ x)) = x , blue
    g : A × Colour → A ⊎ A ⊎ A
    g (a , red) = inj₁ a
    g (a , green) = inj₂ (inj₁ a)
    g (a , blue) = inj₂ (inj₂ a)
    eq : (A ⊎ A ⊎ A) ≃ (A × Colour)
    eq = f , qinv g (λ { (a , red) → refl ; (a , green) → refl ; (a , blue) → refl})
                    λ { (inj₁ x) → refl ; (inj₂ (inj₁ x)) → refl ; (inj₂ (inj₂ y)) → refl}
```

The equivalence is not too painful to establish. But let's do the same for the `GS-Lens`:

```
GS-Colour-in-A+A+A : GS-Lens (A ⊎ A ⊎ A) Colour
GS-Colour-in-A+A+A = record
  { get = λ { (inj₁ x) → red ; (inj₂ (inj₁ x)) → green ; (inj₂ (inj₂ y)) → blue}
  ; set = λ { (inj₁ x) red → inj₁ x ; (inj₁ x) green → inj₂ (inj₁ x) ; (inj₁ x) blue → inj₂ (inj₂ x)
            ; (inj₂ (inj₁ x)) red → inj₁ x ; (inj₂ (inj₁ x)) green → inj₂ (inj₁ x) ; (inj₂ (inj₁ x)) blue → inj₂ (inj₂ x)
            ; (inj₂ (inj₂ y)) red → inj₁ y ; (inj₂ (inj₂ y)) green → inj₂ (inj₁ y) ; (inj₂ (inj₂ y)) blue → inj₂ (inj₂ y)}
  ; getput = λ { (inj₁ x) red → refl ; (inj₁ x) green → refl ; (inj₁ x) blue → refl
               ; (inj₂ (inj₁ x)) red → refl ; (inj₂ (inj₁ x)) green → refl ; (inj₂ (inj₁ x)) blue → refl
               ; (inj₂ (inj₂ y)) red → refl ; (inj₂ (inj₂ y)) green → refl ; (inj₂ (inj₂ y)) blue → refl}
  ; putget = λ { (inj₁ x) → refl ; (inj₂ (inj₁ x)) → refl ; (inj₂ (inj₂ y)) → refl}
  ; putput = λ { (inj₁ x) red red → refl ; (inj₁ x) green red → refl ; (inj₁ x) blue red → refl
               ; (inj₁ x) red green → refl ; (inj₁ x) green green → refl ; (inj₁ x) blue green → refl
               ; (inj₁ x) red blue → refl ; (inj₁ x) green blue → refl ; (inj₁ x) blue blue → refl

               ; (inj₂ (inj₁ x)) red red → refl ; (inj₂ (inj₁ x)) green red → refl ; (inj₂ (inj₁ x)) blue red → refl
               ; (inj₂ (inj₁ x)) red green → refl ; (inj₂ (inj₁ x)) green green → refl ; (inj₂ (inj₁ x)) blue green → refl
               ; (inj₂ (inj₁ x)) red blue → refl ; (inj₂ (inj₁ x)) green blue → refl ; (inj₂ (inj₁ x)) blue blue → refl

               ; (inj₂ (inj₂ y)) red red → refl ; (inj₂ (inj₂ y)) green red → refl ; (inj₂ (inj₂ y)) blue red → refl
               ; (inj₂ (inj₂ y)) red green → refl ; (inj₂ (inj₂ y)) green green → refl ; (inj₂ (inj₂ y)) blue green → refl
               ; (inj₂ (inj₂ y)) red blue → refl ; (inj₂ (inj₂ y)) green blue → refl ; (inj₂ (inj₂ y)) blue blue → refl}
  }
```

Note how the ∃-Lens is linear in the size of the enumerated type, including the proofs, whilst GS-Lens is quadratic for the function size, and cubic in the proof size! Naturally in a tactic-based theorem provers, the proof for putput would likely have hidden this; this is misleading as the tactics nevertheless generate this large term, as it is what needs to be type-checked.

But the deeper points is that $A \uplus A \uplus A$ does not "contain" a Colour, and yet we can create a lens to get and set it. The GS-Lens view makes this quite mysterious but, in our opinion, the ∃-Lens makes it clear that any type that we can see *up to isomorphism* can be focused on.

In a way, a "better" explanation of ∃-Colour-in-A+A+A is to remark that the types $\top \uplus \top \uplus \top$ (which we'll call $\mathbb{3}$) and Colour are isomorphic, which leads to the chains of isomorphisms $A \uplus A \uplus A \simeq A \times \mathbb{3} \simeq A \times$ Colour. This is a strength of the combinator-based approach to type isomorphisms.

An interesting interpretation of $A \uplus A \uplus A \simeq A \times$ Colour is that we can freely move tagging of data A with *finite information* between type-level tags and value-level tags at will.

4.3 Lenses from reversible circuits

Consider the following lens, built from a generalized `cnot` gate:

```
gcnot-equiv : {A B C : Set} → ((A ⊎ B) × (C ⊎ C)) ≃ ((A ⊎ B) × (C ⊎ C))
gcnot-equiv = factorequiv • id≃ ⊎≃ (id≃ ×≃ swap₊equiv) • distequiv

gcnot-lens : {A B C : Set} → ∃-Lens ((A ⊎ B) × (C ⊎ C)) (C ⊎ C)
gcnot-lens {A} {B} = lens gcnot-equiv
```

The above lens is rather unusual in that it dynamically chooses between passing the $C \uplus C$ value through as-is or swapped, depending on the first parameter. The corresponding GS-Lens would be considerably more complex to write (and prove correct).

The same can be done with a (generalized) Toffoli gate, which ends up being controlled by the conjunction of two values instead of just one, but otherwise introduces no new ideas.

There are quite a few ways to witness the equivalence using an isomorphism:

$$E = A \uplus B \times C \uplus C \simeq A \uplus B \times C \uplus C$$

Recall from Sec. 3 that the level-2 programs are equivalences between isomorphisms. Indeed, these equivalences can be used to show the equivalence of different implementations of gcnot-lens that use different ways of establishing E. More generally the level-2 equivalences can be used to simplify, optimize, and reason about lens programs.

4.4 Completeness

The Π language is *complete* for equivalences, in the sense that any two type which can be written as a sum-of-products over arbitrarily many variables are equivalent if and only if there is a term of Π which witnesses this equivalence. In other words,

Theorem 4.1. *Suppose S and A are two types belonging to the language of the semiring of types $T\left[x_1, \ldots, x_n\right]$ over n variables. If $\exists C.S \simeq C \times A$ is inhabited, then there is a term of Π witnessing the equivalence.*

5 Optics

Lenses have been generalized – and in keeping with the theme, have been named *optics*. The immediate dual to lens is *prism*, which we will dig into a little. We will follow this by general remarks on other optics, as the precise development follows a clear pattern.

5.1 Prism

Prisms are dual to lenses in that they arise from exchanging product ($\times$) with coproduct ($\uplus$). In other words, a prism is $\exists C.S \simeq C \uplus A$, giving us Prism_1 (straightforward definition elided). We can mimick the definitions used for lens for all.

But let us instead take this opportunity to do a rational reconstruction of the usual interface to a prism. Suppose that we have prism $\exists C.S \simeq C \uplus A$ in hand, then:

- Given just an S, what can we get? Running the isomorphism forward, we can obtain a $C \uplus A$ – but that is unsatisfactory as C is supposed to be hidden. We can however *squash* C to obtain a Maybe$A$.

- Given just an A? We can run the isomorphism backwards to get an S.
- Given both an A and an S does not provide any new opportunities.

It is common to describe prisms in terms of *pattern matching*. This is adequate when the isomorphism embedded in a prism is a "refocusing" of a member of a sum type – but less so with a non-trivial isomorphism. The formulation as $\exists C.S \simeq C \uplus A$ instead suggests that a prism is a *partition* view of S; we will thus choose to use belongs and inject as field names, rather than (respectively) preview and review as is common in Haskell implementations. As with the fields of the interface, we can reconstruct the laws by attempting various (legal) compositions. Putting all of this together, we get:

```
record BI-Prism {ℓs ℓa : Level} (S : Set ℓs) (A : Set ℓa) : Set (ℓs ⊔ ℓa) where
  field
    belongs : S → Maybe A
    inject  : A → S
    belongsinject : (a : A) → belongs (inject a) ≡ just a
    belongs≡just→inject : (s : S) → (a : A) → (belongs s ≡ just a → inject a ≡ s)
```

From this, we can again prove soundness:

```
module _ {ℓ : Level} (S A : Set ℓ) where
  prism-sound₁ : Prism₁ S A → BI-Prism S A
  prism-sound₁ (prism₁ (f , qinv g a β) ) = record
    { belongs = λ s → [ const nothing , just ]′ (f s)
    ; inject = g ∘ inj₂
    ; belongsinject = λ _ → cong ([ _ , _ ]′) (a _)
    ; belongs≡just→inject = refine
    }
    where
      refine : (t : S) → (a : A) → [ const nothing , just ]′ (f t) ≡ just a → g (inj₂ a) ≡ t
      refine s b pf with f s | inspect f s
      refine s b () | inj₁ x | _
      refine s _ refl | inj₂ y | [ eq ] = trans (cong g (sym eq)) (β s)
```

where injectivity of constructors is used in a crucial way. The combinator [_,_]′ for $\uplus$ is akin to Haskell's `either`. The details of the refine implementation rely on *injectivity of constructors* to reject impossible cases.

But, as with lens, this is not quite complete. We thus upgrade the definition in the same way:

```
record ∃-Prism {ℓ : Level} (S : Set ℓ) (A : Set ℓ) : Set (suc ℓ) where
  constructor ∃-prism
  field
    {C} : Setoid ℓ ℓ
    iso : Inverse (P.setoid S) (C ⊎S (P.setoid A))

prism : {ℓ : Level} {S A C : Set ℓ} → S ≃ (C ⊎ A) → ∃-Prism S A
prism {S = S} {A} {C} (f , qinv g a β) = ∃-prism {C = P.setoid C} (record
  { to = record { _⟨$⟩_ = f ; cong = cong-f }
  ; from = record { _⟨$⟩_ = g ; cong = cong-g }
  ; inverse-of = record
    { left-inverse-of = β
    ; right-inverse-of = λ { (inj₁ x) → Setoid.reflexive Z (a (inj₁ x))
```

```
                    ; (inj₂ y) → Setoid.reflexive Z (a (inj₂ y))}
  }
})
where
  Z = P.setoid C ⊎S P.setoid A
  cong-f : {i j : S} → i ≡ j → (P.setoid C ≈S P.setoid A) (f i) (f j)
  cong-f {i} {.i} refl with f i
  cong-f {i} {.i} refl | inj₁ x = refl
  cong-f {i} {.i} refl | inj₂ y = refl
  cong-g : {i j : C ⊎ A} → (P.setoid C ≈S P.setoid A) i j → g i ≡ g j
  cong-g {inj₁ x} {inj₁ .x} refl = refl
  cong-g {inj₁ x} {inj₂ y} (lift ())
  cong-g {inj₂ y} {inj₁ x} (lift ())
  cong-g {inj₂ y} {inj₂ .y} refl = refl
```

The principal reason for including all of this code is to show that there are rather substantial differences in the details. Where η for products was crucial before, here injectivity of inj_1 and inj_2 play a similar role.

From this, we can then prove a new soundness result as well as a completeness result. The full details are omitted as they are quite lengthy[1]. The main component is the computation of the "other" component, corresponding roughly to $S - A$, which is the Setoid with Carrier $\Sigma S \lambda s \to \mathsf{belongs} bis \equiv \mathsf{nothing}$ and equivalence on the first field. This is roughly equivalent to what Grenrus showed [10], but without the need for proof irrelevance in the meta-theory, as we build it in to our Setoid instead.

Note that there is one more way, again equivalent, of defining a prism: rather than using $\mathsf{Maybe} A$, use $S \uplus A$ and replace belongs with $\lambda s \to \mathsf{Data.Sum.mapconst} sid fs$.

5.2 Other Optics

A number of additional optics have been put forth. Their salient properties can be contrasted in the following table which lists the relation between the source and the view in each case:

Equality	$S = A$
Iso	$S \simeq A$
Lens	$\exists C. S \simeq C \times A$
Prism	$\exists C. S \simeq C + A$
Achroma	$\exists C. S \simeq \top \uplus C \times A$
Affine	$\exists C, D. S \simeq D \uplus C \times A$
Grate	$\exists I. S \simeq I \to A$
Setter	$\exists F : Set \to Set. S \simeq FA$

Figure 2. A variety of optics

The names used are as found in various bits of the literature [2, 10, 15, 19]. A line is drawn when the language is extended. Equality is sometimes called Adapter: it merely witnesses equi-inhabitation of S and A without any requirements that the witnessing functions are in any way related. Iso, short for Isomorphism, is exactly type equivalence. Then we have Lens and Prism, as well as two new ones: Achroma [2] and Affine [15]. This latter construction is the most general. Using it, we obtain:

- Lens when $D = \bot$,
- Prism when $C = \top$,

[1] Though available, in full, in the source already cited.

- Achroma when $D = \bot$ and $C = \top \uplus C'$,
- Iso when $D = \bot$ and $C = \top$.

Specializing C to $\top$ in Affine does not give anything useful, as it ignores A and just says that S is isomorphic to *something*, which is a tautology (as D can be taken to be S itself). Strangely the table is obviously missing one case, which is when $D = \top$.

Grate [19] moves us to a rather different world, one that involves function types. And Setter is more general still, where all we know is that S is isomorphic to some *container* of As.

6 Discussion

6.1 Categorical approaches

So why all the complications with `Profunctor` (see e.g. [2])? Basically, that is mostly Haskell-isms: by relying on *Free Theorems*, one can get the type system to reject a lot of ill-formed lenses, though, of course, not all. Optics, in Agda and using equivalences turn out to be *simpler*, not harder!

Another thread is via the Yoneda lemma [3]. Of course, one can see this here too: the existentials correspond to a co-end, and the isomorphisms are exactly what is in the Hom-set. But we get more mileage from looking "under the hood" to see the fundamental **programming language** underlying Optics, rather than jumping to abstractions too early.

6.2 Geometry of types

The type equivalence view brings to the fore a certain "geometric" picture of types: A Lens is exactly a *cartesian factoring*, where a type can be seen, via an isomorphism, as a cartesian product. That the complement does not depend on A is an integral part of it being "cartesian".

By the same reasoning, a Prism is the identification of a type as being *partitionable* into two pieces.

A reasonable picture is to view A as a sort of *curved* 2-dimensional type, while $C \times A$ is the cartesian, straightened "rectangular" version. C doesn't depend on A, which is why the name *constant complement* is quite apt. In other words, a Lens is a *change of coordinates* that allows one to see A as a cartesian projection. Similarly, a Prism is a *change of coordinates* that shuffles all of A "to the right".

6.3 Optimizing Optics programs

Unlike general programs, which are fiendish to optimize, equivalences, written in the language in this paper, are rather different: there is a sound and complete set of equations that hold for those which are "sufficiently polymorphic" (the technical details can be found in [9, 16]). Most of those equations can be oriented (by size) to produce an optimizer. Of course, more general approaches are needed if wants a *global* optimizer — there is much room for future research in this area. The complete system is rather daunting: over 200 equations!

7 Conclusion

We have shown that the approach to optics via type equivalences, in a dependently typed language, is quite enlightening. Surprisingly, it is no more difficult than dealing with optics in other languages. In fact, the error messages we got from Agda were considerably clearer than when doing `Profunctor`-optics in Haskell.

The connection to reversible programming (and thence to quantum computing) is intriguing; while it has been known for a long time, it appears to have not been sufficiently investigated. Perhaps what was missing was the right dependently-typed setting to bring forth the deep connections.

References

[1] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. 2018. *Introduction to Bidirectional Transformations*. Springer, Cham. https://doi.org/10.1007/978-3-319-79108-1_1

[2] Guillaume Boisseau. 2017. *Understanding Profunctor Optics: a representation theorem.* Master's thesis. University of Oxford.

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

[4] Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. 2018. From Reversible Programs to Univalent Universes and Back. *Electronic Notes in Theoretical Computer Science* 336 (2018), 5 – 25. https://doi.org/10.1016/j.entcs.2018.03.013 The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

[5] Jacques Carette, Roshan P. James, and Amr Sabry. 2018. Embracing the Laws of Physics: Three Reversible Models of Computation. (2018). arXiv:1811.03678.

[6] Jacques Carette and Amr Sabry. 2016. *ESOP 2016*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Computing with Semirings and Weak Rig Groupoids, 123–148.

[7] Marcelo Fiore. 2004. Isomorphisms of generic recursive polynomial types. In *POPL*. ACM, 77–88.

[8] M. P. Fiore, R. Di Cosmo, and V. Balat. 2006. Remarks on Isomorphisms in Typed Calculi with Empty and Sum Types. *Annals of Pure and Applied Logic* 141, 1-2 (2006), 35–50.

[9] Thomas M. Fiore, Po Hu, and Igor Kriz. 2008. Laplaza sets, or how to select coherence diagrams for pseudo algebras. *Advances in Mathematics* 218, 6 (2008), 1705 – 1722. https://doi.org/10.1016/j.aim.2007.05.001

[10] Oleg Grenrus. [n. d.]. Finding correct (lens) laws. http://oleg.fi/gists/posts/2018-12-12-find-correct-laws.html Accessed: December 14, 2018.

[11] Martin Hofmann and Thomas Streicher. 1996. The Groupoid Interpretation of Type Theory. In *Venice Festschrift*. 83–111.

[12] Roshan P. James and Amr Sabry. 2012. Information effects. In *POPL*. ACM, 73–84.

[13] G.M. Kelly and Saunders Mac Lane. 1971. Coherence in closed categories. *Journal of Pure and Applied Algebra* 1, 1 (1971), 97 – 140. https://doi.org/10.1016/0022-4049(71)90013-2

[14] G. M. Kelly. 1974. Coherence theorems for lax algebras and for distributive laws. In *Category Seminar*, Gregory M. Kelly (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 281–375.

[15] Edward Kmett. 2013. lens-4.0: Lenses, folds and traversals. (2013). http://ekmett.github.io/lens/Control-Lens-Traversal.html.

[16] Miguel L. Laplaza. 1972. Coherence for distributivity. In *Coherence in Categories*, G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane (Eds.). Lecture Notes in Mathematics, Vol. 281. Springer Verlag, Berlin, 29–65. https://doi.org/10.1007/BFb0059555

[17] Saunders Mac Lane. 1976. Topology and Logic as a Source of Algebra. *Bull. Amer. Math. Soc.* 82 (1976), 1–40.

[18] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017. Synthesizing Bijective Lenses. *Proc. ACM Program. Lang.* 2, POPL, Article 1 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158089

[19] Russel O'Connor. 2015. Grate: A new kind of Optic. (2015). https://r6research.livejournal.com/28050.html.

[20] Tommaso Toffoli. 1980. Reversible Computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 632–644.

[21] Twan van Laarhoven. 2011. Isomorphic Lenses. (2011). Miltner:2017:SBL:3177123.3158089.