Shattered lens

Oleg Grenrus
Well-Typed LLP
oleg.grenrus@iki.fi

Very well-behaved lenses do not describe a (virtual) field of a record as we might intuitively think. Updating a structure through a lens can cause changes in the other parts of the structure in a larger extent than what is visible through the getter part. That is evident by considering homotopic descriptions of lenses which also suggests that the setter part is the defining part of the lens. On the other hand prisms can be seen as "generalized constructors". Builder and matcher uniquely define each other.

1 Introduction

A *very well-behaved lens* [4] from a structure type S to a value type V is usually specified using two functions, a getter $f : S \to V$ and a setter $g : S \to V \to S$. These functions are required to satisfy three laws:

$$\mathsf{GetPut}\ f\ g \coloneqq \prod_{s:S} \prod_{v:V} f\ (g\ s\ v) = b \qquad \mathsf{PutGet}\ f\ g \coloneqq \prod_{s:S} g\ s\ (f\ s) = s$$
$$\mathsf{PutPut}\ g \coloneqq \prod_{s:S} \prod_{v\ v':B} g\ (g\ s\ v')\ v = g\ s\ v$$

The type of lens is then

$$\mathsf{Lens}\ S\ V \coloneqq \sum_{(f,g):(S\to V)\times(S\to V\to S)} \mathsf{GetPut}\ f\ g \times \mathsf{PutGet}\ f\ g \times \mathsf{PutPut}\ g$$

This formulation is problematic if we want to talk about the *equality* of lenses, for example to prove the associativity of lens composition. We need to show that law evidences are also equal which may not be easy as we do not have uniqueness of identity proofs because it is not consistent with univalence. Compare that with the definitions of *isomorphism* and *equivalence* in homotopy type theory. We will introduce the concepts needed in Appendix A using the standard reference for homotopy type theory: the "HoTT Book" [5].

Paolo Capriotti proposes the following definition for *higher lens* [1].

Definition 1.1. A *higher lens* is a dependent triple

$$\mathsf{HigherLens}\ S\ V \coloneqq \sum_{f:S\to V} \sum_{P:\|V\|\to\mathsf{Type}} \prod_{v:V} (\mathsf{fiber}_f\ v \simeq P\,|v|).$$

where $\mathsf{fiber}_f\ b \coloneqq \sum_{a:A}(fa = b)$ (Definition A.4).

This formulation is equivalent to to an *equivalence-lens* definition [3]:

$$\mathsf{ELens}\ S\ V \coloneqq \sum_{R:\mathsf{Type}} (S \simeq (V \times R)) \times (R \to \|V\|).$$

Submitted to:
MSFP2020

However, these definitions are not easy to work with. The restricting types are not *mere propositions* as isEquiv (Definition A.5) in case of equivalence. More concretely, the f in the HigherLens definition does not determine the lens completely. There is non-unique computational content for a setter "hidden" in the fiber equivalence. It is wrong to only say "f is a lens" without mentioning how the setter is defined.

We can illustrate this by a small example. See Appendix B for an analogous real-world example.

$$
\begin{aligned}
&f :: (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool} \\
&f = \mathsf{fst} \\
&g :: (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool} \to (\mathsf{Bool}, \mathsf{Bool}) \\
&g\ (x, r)\ y = (y, \mathsf{xor}\ y\ (\mathsf{xor}\ r\ x))
\end{aligned}
$$

It would be highly misleading to call the lens formed from f and g by the getter: fst.

The corresponding higher lens has $f = \mathsf{fst}$ and $Px = \mathsf{Bool}$ as the first and second fields of a triple. The third field, a dependent function $\prod_{v:\mathsf{Bool}} \mathsf{fiber}_{\mathsf{fst}}\ v \simeq \mathsf{Bool}$, determines the behaviour of the setter. In fact there are two values which produce the same behaviour, assigning id and not equivalences to different values of v. This means that can construct two observably different higher lenses which getters and setters are equal to f and g. This suggests that HigherLens is "bigger" than Lens (when S and V are h-sets).

The above invites one to look into the homotopic representations of lenses more closely.

2 Prisms

Before investigating lenses we look at prisms. Their theory turns out to be quite simple. Prisms are to coproducts what lenses are to products. A Prism $S\ V$ is usually specified using two functions, a builder $g : V \to S$ and a matcher $f : S \to \mathsf{Maybe}\ V$. Prisms can also be defined by requiring that the *builder* function is a *decidable embedding* i.e. a function whose fibers are decidable propositions.

Definition 2.1. A function $f : A \to B$ is an *embedding* if for every $a : A$ its fibers are mere propositions:

$$\mathsf{hasPropFibers}\ f :\equiv \prod_{a:A} \mathsf{isProp}\ (\mathsf{fiber}_f\ b).$$

Definition 2.2. A type P is *decidable* if P or $\neg P$:

$$\mathsf{Dec}\, P = P + \neg P$$

Definition 2.3. A function $g : V \to S$ is a *prism* when a proof of isPrism g can be constructed

$$
\begin{aligned}
\mathsf{isPrism}\ g &:\equiv \mathsf{hasPropFibers}\ g \times \prod_{s:S} \mathsf{Dec}\,(\mathsf{fiber}_g\ s) \\
\mathsf{Prism}\ S\ V &:\equiv \sum_{g:V\to S} \mathsf{isPrism}\ g
\end{aligned}
$$

.

isPrism is a mere proposition because the product of propositions is one, hasPropFibers is a mere proposition and Dec P is a mere proposition if P is. As a corollary, it is easy to show that composition of prisms is associative.

Contrary to the lens case, saying "a function g is a prism" is justified. The matching function is uniquely defined. To match on a value use the $\prod_{s:S} \mathsf{Dec}\,(\mathsf{fiber}_g\ s)$ part. You will either get a fiber which contains a value of type V, or a proof that there cannot be one.

3 Lenses

We will try to understand lenses by considering *fibers* of a getter. We start with a *barely behaving lens* which satisfies GetPut law. It can be further strengthened in few ways by imposing addition restrictions on the fiber mapping.

- If we require the mapping to be an equivalence, we do not get a previously known variant of lenses. We call such lenses *weak*.
- If we require the mapping to not change the fiber when $v_1 = v_2$, we get a *well-behaved lens* which also satisfies the PutGet law.
- The HigherLens (Definition 1.1) can be thought as requiring the mapping to go through a single (or more correctly: coherently constant) type $P\,|v|$.

Definition 3.1. A function $f : S \to V$ is a *barely behaving lens* if it comes together with a value of following type:

$$\mathsf{hasBareLens}\ f \coloneqq \prod_{v_1:V} \prod_{v_2:V} (\mathsf{fiber}_f\ v_1 \to \mathsf{fiber}_f\ v_2)$$

Definition 3.2. A function $f : S \to V$ is a *weak lens* if it comes together with a value of following type:

$$\mathsf{hasWeakLens}\ f \coloneqq \prod_{v_1:V} \prod_{v_2:V} (\mathsf{fiber}_f\ v_1 \simeq \mathsf{fiber}_f\ v_2)$$

Definition 3.3. A function $f : S \to V$ is a *well-behaving lens* (the name is due to [4]) if it is a barely behaving lens with a proof $p : \mathsf{hasBareLens}\ f$ and additionally the following type is inhabited:

$$\prod_{v:V} \prod_{x:\mathsf{fiber}_f\ b} p\ v\ v\ x = x$$

While the definitions above might be useful in various applications, none of them definitions is a mere proposition. There does not seem to be a simple way to strenghten HigherLens further.

For prisms we have found an elegant approach of building on top of a known concept: an embedding. We can try whether related *surjection* can be used to characterize lens. It is surprising at first that none of the lens representations allow to show that the getter is a surjection. We end up needing an assumption like $\|S + \neg S\|$ which is not a huge requirement, but a requirement nevertheless.

On the other hand, the setter part when thought as a function $S \times V \to S$ can be shown to be a surjection when the PutGet law holds, i.e. for well-behaved lenses and stronger variants. Alternatively we take the setter to be the primary building block of lenses:

Definition 3.4. A function $g : S \to V \to S$ has getter when

$$\mathsf{hasGetter}\ g \coloneqq \prod_{s:S} \mathsf{isContr}\ (\mathsf{fiber}_{g\,s}\ s)$$

hasGetter is a mere proposition, and lets us show the PutGet law. Interestingly to show the GetPut law we need a simpler variant of the PutPut law, essentially asking that the setter is idempotent:

Definition 3.5.

$$\mathsf{idempotentSetter}\ g \coloneqq \prod_{s:S} \prod_{v:V} g\ (g\ s\ v)\ v = g\ s\ v$$

The above suggests that the setter is the defining part of a lens. The compatible getter is uniquely determined. Yet, we do not yet know what a “higher setter“ would look like. That is a topic for further work.

References

[1] Paolo Capriotti: *Higher Lenses*. https://homotopytypetheory.org/2014/04/29/higher-lenses/. Accessed: 2020-03-15.

[2] Koen Claessen & John Hughes (2000): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP 00, Association for Computing Machinery, New York, NY, USA, pp. 268–279, doi:10.1145/351240.351266.

[3] Nils Anders Danielsson: *Dependent Lenses*. http://www.cse.chalmers.se/~nad/publications/danielsson-dependent-lenses.pdf. Accessed: 2020-03-15.

[4] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce & Alan Schmitt (2007): *Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem*. *ACM Trans. Program. Lang. Syst.* 29(3), doi:10.1145/1232420.1232424.

[5] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[6] Andrea Vezzosi, Anders Mörtberg & Andreas Abel (2019): *Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types*. *Proc. ACM Program. Lang.* 3(ICFP), doi:10.1145/3341691.

A Homotopy Type Theory overview

As we mentioned in the introduction, the "HoTT Book" [5] is the standard reference for homotopy type theory. The functional programming language AGDA has been recently extended into CUBICAL AGDA [6], giving computation interpretation to the homotopy type theory. The statements in this paper are translated from CUBICAL AGDA code.

Homotopy type theory is based on a dependent type theory with

- dependent function types $(a \mapsto \ldots) : \prod_{a:A} B(a)$;
- dependent pair types $\langle a;b\rangle : \Sigma_{a:A} B(a)$ (we name projections fst and snd);
- universe types we call just Type as we omit level issues for brevity,
- non-dependent product types $\langle a;b\rangle : A \times B$ (with projections named proj_1 and proj_2);
- empty type $\bot$, and negation $\neg A := A \to \bot$;
- coproduct types $A + B$; with constructors inl and inr[1];
- Booleans $\mathsf{true} : \mathsf{Bool}$; $\mathsf{false} : \mathsf{Bool}$
- ...

Definition A.1 (Identity types)**.** There is a type family of identity types $\mathsf{Id}_A : A \to A \to \mathsf{Type}$. It is also called the path space of A. We use a standard notation $a =_A b$ or just $a = b$ to denote that two elements of the same type $a, b : A$ are equal. The introduction rule is a dependent function

$$\mathsf{refl} : \prod_{a:A} (a =_A a).$$

Definition A.2 (Homotopy levels)**.** Types can be classified according to their *homotopy-level*. Homotopy levels are cumulative.

[1] in AGDA code the $A \uplus B$ is used to avoid clash with addition on natural numbers

- A type is *contractible* when it has an element to which all others elements are equal

$$\mathsf{isContr}\ A \coloneqq \sum_{x:A} \prod_{y:A} (x = y).$$

- A type is *mere proposition* when all its elements are equal:

$$\mathsf{isProp}\ A \coloneqq \prod_{x\ y:A} (x = y)$$

  Mere propositions values a *proof irrelevant*: they are all equal if they exist. However, not as types Coq in Prop universe, values of mere proposition types can be eliminated into non-proposition types too. $\mathsf{isContr}\ A$ and $\mathsf{isProp}\ A$ are itself mere propositions.

- A type is *h-set* when its path space is mere proposition.

$$\mathsf{isSet}\ A \coloneqq \prod_{x\ y:A} \mathsf{isProp}\ (x = y)$$

  For h-sets the *uniqueness of identity proofs* holds.

Definition A.3. It is possible to truncate any type to be a mere proposition, using *propositional truncation*

$$|a| : \|A\|$$

The downside of using propositional truncation is that values of type $\|A\|$ can be eliminated only with mere proposition motives.

Definition A.4. A *fiber* of a map $f : A \to B$ is the type of its pre-image values:

$$\mathsf{fiber}_f\ b \coloneqq \sum_{a:A} (fa = b)$$

Definition A.5. A function $f : A \to B$ is an *equivalence* if all its fibers are contractible:

$$\mathsf{isEquiv}\ f \coloneqq \prod_{b:B} \mathsf{isContr}\ (\mathsf{fiber}_f\ b)$$

$\mathsf{isEquiv}\ f$ is a mere proposition which implies that there is the *unique* inverse function:

$$f^{-1}\ \mathit{isEquiv\text{-}f}\ b \coloneqq \mathsf{fst}\ (\mathit{isEquiv\text{-}f}\ b).$$

Using isEquiv we define equivalence of types as:

$$A \simeq B \coloneqq \sum_{f:A\to B} \mathsf{isEquiv}\ f$$

Theorem A.6. Using the just defined equivalence we can state the *univalence axiom*:

$$\mathsf{univalence} \coloneqq (A \simeq B) \simeq (A = B)$$

Though we call it an axiom, in cubical type theories like in Cubical Agda it does compute. We will not use the axiom directly, but it is nevertheless important to mention. For an example note that $\mathsf{Bool} = \mathsf{Bool}$ type is not contractible, there are also other values than refl.

Theorem A.7. Given a type A and a type family $B : A \to \mathsf{Type}$, consider the dependent functions $f, g : \prod_{a:A} B\ a$. The following *functional extensionality* equivalence is a theorem in homotopy type theory:

$$\mathsf{funExt} \coloneqq (f = g) \simeq \prod_{a:A} (f\ a =_{B\ a} g\ a).$$

B Encrypt-decrypt example

This example will be written in HASKELL. Let us develop a stream cipher implementation. We will represent encryption keys by an initial state of the random number generator.

```
newtype Key = Key (Maybe Gen)
```

The generator is wrapped in Maybe, because we want to represent a special *zero key*:

```
zeroKey :: Key
zeroKey = Key Nothing
```

The behavior is best explained by the encryption code. We have a function to generate random Word8 (8-byte unsigned integer) from a Maybe Gen:

```
nextW8 :: Maybe Gen → (Word8, Maybe Gen)
nextW8 Nothing = (0, Nothing)
nextW8 (Just g) = ...
```

where the Just branch uses the RNG g. Using nextW8; we can *encrypt* a plain text message:

```
encrypt :: Key → [Word8] → [Word8]
encrypt _          []     = []
encrypt (Key g) (x : xs) = xor w x : encrypt (Key g') xs where (w, g') = nextW8 g
```

Because of the properties of xor we can use the same function to decrypt cipher text, in other words: decrypt = encrypt. Next let us a define a data structure which carries a key with a cipher text encrypted with that key.

```
data Bundled = Bundle Key [Word8]
```

We can bundle a *plain text* with the zeroKey:

```
plain :: [Word8] → Bundled
plain = Bundle zeroKey
```

Now the key part. We construct the key lens using a lens constructor which takes getter and setter functions. When setting a new key, we first decrypt the cipher text with an old key and then re-encrypt with the new one:

```
key :: Lens Bundled Key
key = lens
  (λ(Bundle k _)    → k)
  (λ(Bundle k x) k' → Bundle k' (encrypt k' (decrypt k x)))
```

And let us also have an ordinary projection lens for the cipher text part:

```
ciphertext :: Lens Bundled [Word8]
ciphertext = lens
```

$$(\lambda(\mathsf{Bundle}\ \_\ x) \quad \to x)$$
$$(\lambda(\mathsf{Bundle}\ k\ \_)\ x \to \mathsf{Bundle}\ k\ x)$$

If we not take $\mathsf{msg} = [108, 101, 110, 115]$ (which stands for `"lens"` in Latin1 encoding); then we can encrypt it using lens primitives as:

$$\mathsf{secret} = \mathsf{view}\ \mathsf{ciphertext}\ (\mathsf{set}\ \mathsf{key}\ \mathsf{anotherKey}\ (\mathsf{plain}\ \mathsf{msg}))$$

secret could then be e.g. $[113, 107, 89, 46]$ standing for `"qkY."`. But we can also decrypt that message using the same procedure!

$$\mathsf{msg}' = \mathsf{view}\ \mathsf{ciphertext}\ (\mathsf{set}\ \mathsf{key}\ \mathsf{zeroKey}\ (\mathsf{Bundle}\ \mathsf{anotherKey}\ \mathsf{secret}))$$

The result msg' will be equal to the original msg!

Both lenses used here – key and ciphertext – are lawful, one can verify that using e.g. QUICKCHECK library [2], It will not find any counterexamples. We think this is a cool example usage of lens but it feels *morally wrong*.