A clear picture of lens laws

—Functional Pearl—

Sebastian Fischer[1], Zhenjiang Hu[2], and Hugo Pacheco[3]

[1] Christian-Albrechts-Universität, Kiel, Germany
`sebf@informatik.uni-kiel.de`
[2] National Institute of Informatics, Tokyo, Japan
`hu@nii.ac.jp`
[3] Cornell University, Ithaca, NY, USA
`hpacheco@cs.cornell.edu`

Abstract. A lens is an optical device which refracts light. Properly adjusted, it can be used to project sharp images of objects onto a screen—a principle underlying photography as well as human vision. Striving for clarity, we shift our focus to lenses as abstractions for bidirectional programming. By means of standard mathematical terminology as well as intuitive properties of bidirectional programs, we observe different ways to characterize lenses and show exactly how their laws interact. Like proper adjustment of optical lenses is essential for taking clear pictures, proper organization of lens laws is essential for forming a clear picture of different lens classes. Incidentally, the process of understanding bidirectional lenses clearly is quite similar to the process of taking a good picture.
By showing that it is exactly the backward computation which defines lenses of a certain standard class, we provide an unusual perspective, as contemporary research tends to focus on the forward computation.

1 Scene selection

Select the scene to determine the topic of what will be seen in your picture.

A lens [3] is a program that can be run forwards and backwards. The forward computation extracts a *view* from a *source* and the backward computation updates a given source according to a given view producing a new source. Using Haskell notation [1], a lens consists of a view extraction function $get :: Source \to View$ and a source update function $put :: Source \to View \to Source$.

Not every combination of such functions is reasonable. Different algebraic laws between *get* and *put* have been proposed to define a hierarchy of lens classes. For the sake of clarity, we present all laws in terms of total functions. Some programmers find it easier to define partial functions for certain problems, and our results can be adjusted to this setting.

The smallest among the lens classes we investigate is that of *bijective lenses* where *get* and *put s* are bijective inverse functions of each other for all sources s.

$$\forall\ s, s' \quad put\ s\ (get\ s') = s' \qquad \textsf{StrongGetPut}$$
$$\forall\ s, v \quad get\ (put\ s\ v) = v \qquad \textsf{PutGet}$$

Bijective lenses require a one-to-one correspondence between the *Source* and *View* types which does not fit the asymmetric types of *get* and *put*. In fact, the source argument of *put* is not needed for bijective lenses because the *put* function builds the new source using only the view and discarding the input source.

In the broader class of *very well-behaved lenses*, the *get* function is allowed to discard information. Very well-behaved lenses satisfy the above PutGet law as well as the two following laws.

$$\forall\ s \quad put\ s\ (get\ s) = s \qquad \textsf{GetPut}$$
$$\forall\ s, v, v' \quad put\ (put\ s\ v')\ v = put\ s\ v \qquad \textsf{PutPut}$$

The GetPut law is a more permissive variant of the StrongGetPut law because it passes the same source values to *get* and *put*, allowing (and requiring) the *put* function to reconstruct information discarded by *get*. The PutPut law requires that source updates overwrite the effect of previous source updates.

That law is dropped in an even broader class of *well-behaved lenses* which satisfy only the PutGet and GetPut laws, requiring that source updates reflect changes in given views but do not perform any changes if views do not change.

The presented lens classes are closed under lens composition [3], i.e., for any two lenses in a class there is a composed lens (where the *get* function is the composition of the underlying *get* functions) that is itself in that class.

Example 1. As an example lens consider the following pair of functions.

$$\begin{aligned} &getFirst\ (x, \_) &&= x \\ &putFirst\ (\_, y)\ v &&= (v, y) \end{aligned}$$

Applying *putFirst* is analogous to changing the width of a picture without changing its height. We can verify the PutGet and GetPut laws as follows.

$$\begin{aligned} & getFirst\ (putFirst\ (x, y)\ v) \\ = \quad & \{ \text{ by definition of } putFirst \ \} \\ & getFirst\ (v, y) \\ = \quad & \{ \text{ by definition of } getFirst \ \} \\ & v \end{aligned}$$

$$\begin{aligned} & putFirst\ (x, y)\ (getFirst\ (x, y)) \\ = \quad & \{ \text{ by definition of } getFirst \ \} \\ & putFirst\ (x, y)\ x \\ = \quad & \{ \text{ by definition of } putFirst \ \} \\ & (x, y) \end{aligned}$$

Consequently, *getFirst* and *putFirst* form a well-behaved lens between an arbitrary type of pairs and the type of their first components. Specialized to the type $(\mathbb{R}^+, \mathbb{R}^+) \to \mathbb{R}^+$, the function *getFirst* also forms a well-behaved lens with a different *put* function:

$$\mathit{putScaled}\ (x, y)\ v = (v, v * y\ /\ x)$$

The function *putScaled* maintains the ratio of the components of the pair, which is analogous to resizing a picture without distorting it.

Both *put* functions satisfy the PutPut law because the result of an update does not depend on previous updates.

Example 1 shows an inherent ambiguity in the definition of (very) well-behaved lenses based on *get* alone: the same *get* function can be combined with different *put* functions to form a valid lens. One cannot completely specify the behaviour of all lenses by only writing forward computations.

We select our scene to show certain laws of asymmetric bidirectional transformations and soon cast new light on the presented laws using old searchlights.

2 Camera setup

Set up the camera to meet technical requirements for capturing your picture.

For taking our picture, we remind ourselves of mathematical properties that we later use to describe implications of the lens laws.

Surjectivity is a property of functions which requires that every element in the codomain of a function is the image of some element in its domain. More formally, a function $f :: A \to B$ is surjective if and only if the following property holds.

$$\forall\ b :: B\ \exists\ a :: A \quad f\ a = b$$

Surjectivity of $f :: A \to B$, therefore, requires that A is *at least as big as* B.

Conversely, injectivity is a property of functions which requires different arguments to be mapped to different results. More formally, a function $f :: A \to B$ is injective if and only if the following property holds.

$$\forall\ a, a' :: A \quad f\ a = f\ a' \Rightarrow a = a'$$

Injectivity of $f :: A \to B$, therefore, requires that B is *at least as big as* A.

The technical requirements for taking our picture are standard properties of functions. We hope that describing lens laws based on established mathematical terminology helps to gain simple intuitions about lenses in different classes.

3 Perspective

Choose a perspective exposing elements of your picture you want to highlight.

The *put* functions are essential elements in our picture of lenses. We now highlight certain necessary conditions which must be satisfied for *put* functions in all lens classes discussed in this paper.

The GetPut law requires *put* to satisfy the following law.

$$\forall\ s\ \exists\ v \quad put\ s\ v = s \qquad \textsf{SourceStability}$$

Proof. To verify that GetPut implies SourceStability, let s be an arbitrary source value and $v = get\ s$ in the following calculation.

$$\begin{array}{ll} & put\ s\ v \\ = & \{ \text{ by definition of } v \text{ } \} \\ & put\ s\ (get\ s) \\ = & \{ \textsf{ GetPut } \} \\ & s \end{array}$$

SourceStability requires that every source is stable under a certain update. It implies that *put* is surjective in the following sense (equivalent to surjectivity of *uncurry put*.)

$$\forall\ s\ \exists\ s', v \quad put\ s'\ v = s \qquad \textsf{PutSurjectivity}$$

SourceStability is stronger than PutSurjectivity. For example, the *putShift* function defined below, which places the view in the first component of the source and moves the old first component to the second, satisfies the latter property but not the former, because there is no v such that $putShift\ (1,2)\ v = (1,2)$.

$$putShift\ (x, \_)\ v = (v, x)$$

Consequently, *putShift* cannot be part of a well-behaved lens.

PutGet also implies a necessary condition on well-behaved *put* functions.

$$\forall\ s, s', v, v' \quad put\ s\ v = put\ s'\ v' \Rightarrow v = v' \qquad \textsf{ViewDetermination}$$

Proof. We can use the PutGet law to conclude ViewDetermination as follows.

$$\begin{array}{ll} & put\ s\ v = put\ s'\ v' \\ \Rightarrow & \{ \text{ by applying } get \text{ to each side } \} \\ & get\ (put\ s\ v) = get\ (put\ s'\ v') \\ \Rightarrow & \{ \textsf{ PutGet } \} \\ & v = v' \end{array}$$

ViewDetermination requires that a view used to update a source can be determined from the result of an update, regardless of the original source. It implies the following injectivity property of *put* functions.

$$\forall\ s \quad put\ s \text{ is injective} \qquad \textsf{PutInjectivity}$$

ViewDetermination is stronger than PutInjectivity. For example, the *putSum* function defined below, which adds the source to the view to produce an updated source, satisfies the latter property but not the former, because $putSum\ 2\ 3 = putSum\ 1\ 4$ and $3 \neq 4$.

$$putSum\ m\ n = m + n$$

Consequently, *putSum* cannot be part of a well-behaved lens.

PutInjectivity confirms our previous intuition about the asymmetry of the *Source* and *View* types of a well-behaved lens: the *Source* type needs to be at least as big as the *View* type. From our examples, we already concluded that the opposite is not necessary: *get* may discard information that *put* is able to reconstruct based on a given source.

We finally observe yet another necessary condition on *put* functions that is implied by the combination of the PutGet and GetPut laws.

$$\forall\ s, v \quad put\ (put\ s\ v)\ v = put\ s\ v \qquad \textsf{PutTwice}$$

Proof. $\quad put\ (put\ s\ v)\ v$

$= \quad \{\ \textsf{PutGet}\ \}$

$\quad put\ (put\ s\ v)\ (get\ (put\ s\ v))$

$= \quad \{\ \textsf{GetPut}\ \}$

$\quad put\ s\ v$

PutTwice is weaker than PutPut, because it only requires source updates to be independent of previous updates that were using the same view. For example, the *putChanges* function defined below places the given view in the first component of the source and counts the number of changes in the second, which is anologous to counting how often a picture (in the first component) has been resized. It satisfies PutTwice but not PutPut.

$$putChanges\ (x, c)\ v = (v, \mathbf{if}\ x == v\ \mathbf{then}\ c\ \mathbf{else}\ c + 1)$$

So, although *putChanges* forms a well-behaved lens with *getFirst* it cannot be part of a very well-behaved one.

The following proposition summarizes the properties of *put* functions that we observed in this section.

Proposition 1. *Every put function of a well-behaved lens satisfies SourceStability, ViewDetermination, PutSurjectivity, PutInjectivity, and PutTwice.*

We choose a perspective that highlights the *put* function by masking the *get* function in implications of the standard formulation of lens laws. Our focus will be justified by the central role of *put* which we are about to reveal.

4 Composition

Compose picture elements to convey your message, applying appropriate technique.

We now align our camera to put the described properties into a position that clarifies their role in our picture. We do not concern ourselves with the composition of lenses (which is orthogonal to our focus on *put* functions) but with the composition of lens laws to characterize well-behaved lenses using the necessary conditions on *put* functions presented in Section 3.

The following theorem shows that the *put* function in a well-behaved lens determines the corresponding *get* function and that every *put* function which satisfies the properties identified in Section 3 is part of a well-behaved lens.

Theorem 1. *Let $put :: Source \to View \to Source$ be a function that satisfies $\mathsf{SourceStability}$ and $\mathsf{ViewDetermination}$. Then there is a unique get function that forms a well-behaved lens with put.*

Proof. **Uniqueness** *Let $get_1, get_2 :: Source \to View$ be functions that form a well-behaved lens with put. Then the following equation holds for all sources s.*

$$
\begin{aligned}
& get_1\ s \\
= \quad & \{\ \text{by } \mathsf{GetPut} \text{ law for } get_2\ \} \\
& get_1\ (put\ s\ (get_2\ s)) \\
= \quad & \{\ \text{by } \mathsf{PutGet} \text{ law for } get_1\ \} \\
& get_2\ s
\end{aligned}
$$

Existence *Let s be a source value. Due to $\mathsf{SourceStability}$, there is a view v such that $put\ s\ v = s$. Because of $\mathsf{ViewDetermination}$ this view v is unique because for a view w with $put\ s\ w = s$ we have $put\ s\ v = put\ s\ w \Rightarrow v = w$. Hence, setting $get\ s = v$ for the v such that $put\ s\ v = s$ defines $get\ s$ uniquely.*
To conclude well-behavedness, we observe the GetPut and PutGet laws for this definition of get. For all sources s, the following equation, i.e., the GetPut law, follows from the definition of get based on $\mathsf{SourceStability}$.

$$
\begin{aligned}
& put\ s\ (get\ s) \\
= \quad & \{\ \text{by definition of } get\ \} \\
& put\ s\ v \quad \{\ \text{with } v \text{ such that } put\ s\ v = s\ \} \\
= \quad & \{\ put\ s\ v = s\ \} \\
& s
\end{aligned}
$$

For all sources s and views v, the following equation, i.e., the PutGet law, follows from the definition of get and $\mathsf{ViewDetermination}$.

$$
\begin{aligned}
& get\ (put\ s\ v) \\
= \quad & \{\ \text{by definition of } get\ \} \\
& v' \quad \{\ \text{with } v' \text{ such that } put\ (put\ s\ v)\ v' = put\ s\ v\ \} \\
= \quad & \{\ \mathsf{ViewDetermination}\ \} \\
& v
\end{aligned}
$$

Choosing a perspective that highlights properties of *put* functions, we observed that some of them can be composed to characterize well-behaved lenses. We used equational reasoning as our techique to show that certain properties of *put* functions uniquely determine *get* functions in well-behaved lenses. We neither delve into the question (raised by the non-constructive way of defining *get*) whether there is always a *computable get* function nor do we reflect on how a corresponding *get* function could be derived from *put*.

5 Post-editing

Apply changes during post-editing to refine your picture.

In order to increase the contrast in our picture, we investigate the internal structure of the properties of *put* functions presented in Section 3. The following theorem characterizes the conditions of Theorem 1 to provide an alternative way to specify the sufficient and necessary conditions on *put* functions in well-behaved lenses based on standard mathematical terminology.

Theorem 2. *For* $put :: Source \to View \to Source$*, the following conjunctions are equivalent.*

1. $\mathsf{SourceStability} \wedge \mathsf{ViewDetermination}$
2. $\mathsf{PutSurjectivity} \wedge \mathsf{PutInjectivity} \wedge \mathsf{PutTwice}$

Proof. $1 \Rightarrow 2$ *Because of Theorem 1, put is part of a well-behaved lens. Hence, it satisfies all conditions summarized in Proposition 1.*

$2 \Rightarrow 1$ *First, we conclude* SourceStability *from* PutSurjectivity *and* PutTwice*.*

$$\begin{array}{ll} & s \\ = & \{\ \text{by } \mathsf{PutSurjectivity}\text{, choosing } s' \text{ and } v \text{ with } put\ s'\ v = s\ \} \\ & put\ s'\ v \\ = & \{\ \mathsf{PutTwice}\ \} \\ & put\ (put\ s'\ v)\ v \\ = & \{\ put\ s'\ v = s\ \} \\ & put\ s\ v \end{array}$$

Now, we conclude ViewDetermination *from* PutInjectivity *and* PutTwice*.*

$$\begin{array}{ll} & put\ s\ v = put\ s'\ v' \\ \Rightarrow & \{\ \text{by applying } \mathsf{PutTwice} \text{ on each side}\ \} \\ & put\ (put\ s\ v)\ v = put\ (put\ s'\ v')\ v' \\ \Rightarrow & \{\ put\ s\ v = put\ s'\ v'\ \} \\ & put\ (put\ s\ v)\ v = put\ (put\ s\ v)\ v' \\ \Rightarrow & \{\ \text{by injectivity of } put\ (put\ s\ v)\ \} \\ & v = v' \end{array}$$

Theorem 2 provides an alternative characterization of *put* functions that are part of well-behaved lenses. It shows how the PutTwice law closes the gap between the combination of SourceStability with ViewDetermination and the weaker combination of PutSurjectivity with PutInjectivity.

By incorporating PutPut into each of the conjunctions describing well-behaved *put* functions, we can also characterize very well-behaved lenses based on *put*. The difference that makes well-behaved lenses *very* well-behaved is by definition already expressed using only the *put* function.

After post editing, our picture shows how simple mathematical properties of functions play together to give rise to established laws for asymmetric bidirectional transformations.

6 Perception

Focus on essentials to let others perceive the message of your picture.

Our picture provides a novel perspective on bidirectional lenses. While there may be in general more than one *put* function that can be combined with a given *get* function to form a (very) well-behaved lens, our view on the lens laws emphasizes that *put* functions determine corresponding *get* functions uniquely in all lens classes discussed in this paper. While the idea of *put* determining *get* may be folklore in the circle of lens specialists, we are the first to put it in focus. Our contribution, therefore, is to a lesser extent based on directly applicable technical novelty than it aims to clarify formal properties of lenses in simple terms to gain insights into an already established theory.

Theorem 1 provides a concise characterization of well-behaved lenses only based on the *put* function. Theorem 2 provides an alternative, more elegant, *put*-based characterization in terms of standard mathematical terminology. As far as we know, there has not been a peer-reviewed publication of a *put*-based lens characterization.

Foster gives a similar characterization in his PhD thesis [2] based on properties we present in Section 3. He does not use one of the conjunctions characterized in Theorem 2 but a redundant and, therefore, less revealing combination. We believe that the clarification of the role of PutTwice in our Theorem 2 is novel.

The contemporary focus on *get* functions is incomplete, because a *put*-based characterization shows that it is *exactly* the *put* function that programmers need to define in order to specify (very) well-behaved lenses completely. Similar to how lens combinators allow to express compositions of bidirectional programs in a single definition, defining *put* and deriving *get* would allow to specify primitive lenses without giving a separate implementation for each direction.

Implementations of lens combinators in Haskell such as the `lens`[4] or the `fclabels`[5] packages usually require very well-behaved implementations of both

[4] `http://hackage.haskell.org/package/lens`

[5] `http://hackage.haskell.org/package/fclabels`

directions of primitive lenses. Deriving the *get* function automatically from *put* functions would be beneficial especially for supporting more complex synchronization strategies, because programmers would have to maintain only one function (not two corresponding ones) for each strategy. Our observations show that a *put*-based approach supports even less restrictive lens combinators that only require well-behavedness.

Acknowledgements

We appreciate the help of Janis Voigtländer, Jeremy Gibbons, Alcino Cunha, Nikita Danilenko, Insa Stucke, and José Nuno Oliveira who have looked at drafts of our picture.

References

1. Bird, R.S.: Introduction to Functional Programming Using Haskell. Prentice-Hall (1998), `http://www.cs.ox.ac.uk/publications/books/functional/`
2. Foster, J.: Bidirectional Programming Languages. Ph.D. thesis, University of Pennsylvania (December 2009)
3. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (2007)