Applicative Bidirectional Programming with Lenses

Kazutaka Matsuda
Tohoku University
kztk@ecei.tohoku.ac.jp

Meng Wang
University of Kent
m.w.wang@kent.ac.uk

Abstract

A bidirectional transformation is a pair of mappings between source and view data objects, one in each direction. When the view is modified, the source is updated accordingly with respect to some laws. One way to reduce the development and maintenance effort of bidirectional transformations is to have specialized languages in which the resulting programs are bidirectional by construction—giving rise to the paradigm of bidirectional programming.

In this paper, we develop a framework for *applicative-style* and *higher-order* bidirectional programming, in which we can write bidirectional transformations as unidirectional programs in standard functional languages, opening up access to the bundle of language features previously only available to conventional unidirectional languages. Our framework essentially bridges two very different approaches of bidirectional programming, namely the lens framework and Voigtländer's semantic bidirectionalization, creating a new programming style that is able to bag benefits from both.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Language*]: Languages Constructs and Features—Data types and structures, Polymorphism

General Terms Languages

Keywords Bidirectional Programming, Lens, Bidirectionalization, Free Theorem, Functional Programming, Haskell

1. Introduction

Bidirectionality is a reoccurring aspect of computing: transforming data from one format to another, and requiring a transformation in the opposite direction that is in some sense an inverse. The most well-known instance is the *view-update problem* [1, 6, 8, 13] from database design: a "view" represents a database computed from a source by a query, and the problem comes when translating an update of the view back to a "corresponding" update on the source.

But the problem is much more widely applicable than just to databases. It is central in the same way to most interactive programs, such as *desktop and web applications*: underlying data, perhaps represented in XML, is presented to the user in a more accessible format, edited in that format, and the edits translated back in terms of the underlying data [12, 16, 30]. Similarly for *model transformations*, playing a substantial role in software evolution: having transformed a high-level model into a lower-level implementation, for a variety of reasons one often needs to reverse engineer a revised high-level model from an updated implementation [42, 43].

Using terminologies originated from the lens framework [4, 9, 10], bidirectional transformations, coined *lenses*, can be represented as pairs of functions known as get of type $S \to V$ and put of type $S \to V \to S$. Function get extracts a view from a source, and put takes both an updated view and the original source as inputs to produce an updated source. An example definition of a bidirectional transformation in Haskell notations is

$$\begin{array}{l} \mathbf{data}\ L\ s\ v = L\ \{get :: s \to v, put :: s \to v \to s\} \\ fst_{\mathrm{L}} :: L\ (a, b)\ a \\ fst_{\mathrm{L}} = L\ (\lambda(a, \_) \to a)\ (\lambda(\_, b)\ a \to (a, b)) \end{array}$$

A value ℓ of type $L\ \ s\ \ v$ is a lens that has two function fields namely get and put, and the record syntax overloads the field names as access functions: $get\ \ \ell$ has type $s \to v$ and $put\ \ \ell$ has type $s \to v \to s$. The datatype is used in the definition of fst_{L} where the first element of a source pair is projected as the view, and may be updated to a new value.

Not all bidirectional transformations are considered "reasonable" ones. The following laws are generally required to establish bidirectionality:

$$put\ \ell\ s\ (get\ \ell\ s) = s \qquad (\textbf{Acceptability})$$

$$get\ \ell\ s' = v \text{ if } put\ \ell\ s\ v = s' \qquad (\textbf{Consistency})$$

for all s, s' and v. Note that in this paper, we write $e = e'$ with the assumption that neither e nor e' is undefined. Here **Consistency** (also known as the **PutGet** law [9]) roughly corresponds to right-invertibility, ensuring that all updates on a view are captured by the updated source; and **Acceptability** (also known as the **GetPut** law [9]), prohibits changes to the source if no update has been made on the view. Collectively, the two laws defines *well-behavedness* [1, 9, 13]. A bidirectional transformation $L\ get\ put$ is called *well-behaved* if it satisfies well-behavedness. The above example fst_{L} is a well-behaved bidirectional transformation.

By dint of hard effort, one can construct separately the forward transformation get and the corresponding backward transformation put. However, this is a significant duplication of work, because the two transformations are closely related. Moreover, it is prone to error, because they do really have to correspond with each other to be well-behaved. And, even worse, it introduces a maintenance issue, because changes to one transformation entail matching changes to the other. Therefore, a lot of work has gone into ways to reduce this duplication and the problems it causes; in particular, there has been a recent rise in linguistic approaches to streamlining bidirectional transformations [2, 4, 9–11, 14, 16, 20–22, 25, 27, 30, 33, 35, 36, 38–41].

Ideally, bidirectional programming should be as easy as usual unidirectional programming. For this to be possible, techniques of conventional languages such as *applicative-style* and *higher-order* programming need to be available in the bidirectional languages, so that existing programming idioms and abstraction methods can be ported over. It makes sense to at least allow programmers to treat functions as first-class objects and have them applied explicitly. It is also beneficial to be able to write bidirectional programs in the same style of their gets, as cultivated by traditional unidirectional programming programmers normally start with (at least mentally) constructing a get before trying to make it bidirectional.

However, existing bidirectional programming frameworks fall short of this goal by quite a distance. The lens bidirectional programming framework [2, 4, 9–11, 16, 25, 27, 30, 38, 39], the most influential of all, composes small lenses into larger ones by special lens combinators. The combinators preserve well-behavedness, and thus produce bidirectional programs that are correct by construction. Lenses are impressive in many ways: they are highly expressive and adaptable, and in many implementations a carefully crafted type system guarantees the totality of the bidirectional transformation. But at the same time, like many other combinator-based languages, lenses restrict programming to the point-free style, which may not be the most appropriate in all cases. We have learned from past experiences [23, 28] that a more convenient programming style does profoundly impact on the popularity of a language.

The researches on *bidirectionalization* [14, 20–22, 33, 35, 36, 38, 39, 41], which mechanically derives a suitable put from an existing get, share the same spirit with us to some extent. The gets can be programmed in a unidirectional language and passed in as objects to the bidirectionalization engine, which performs program analysis and the generation of puts. However, the existing bidirectionalization methods are whole program analyses; there is no better way to compose individually constructed bidirectional transformations.

In this paper, we develop a novel bidirectional programming framework:

- As lenses, it supports composition of user-constructed bidirectional transformations, and well-behavedness of the resulting bidirectional transformations is guaranteed by construction.
- As a bidirectionalization system, it allows users to write bidirectional transformations almost in the same way as that of gets, in an applicative and higher-order programming style.

The key idea of our proposal is to lift lenses of type $L\ (A_1,\ldots,A_n)\ B$ to *lens functions* of type

$$\forall s.\ L^{\mathrm{T}}\ s\ A_1 \to \cdots \to L^{\mathrm{T}}\ s\ A_n \to \ldots \to L^{\mathrm{T}}\ s\ B$$

where L^{T} is a type-constrained version of L (Sections 2 and 3). The n-tuple above is then generalized to data structures such as lists in Section 4. This function representation of lenses is open to manipulation in an applicative style, and can be passed to higher-order functions directly. For example, we can write a bidirectional version of $unlines$, defined by

$$\begin{array}{l} unlines :: [String] \to String \\ unlines\ [\,] \qquad\ = \texttt{""} \\ unlines\ (x : xs) = x +\!\!+\ \texttt{"\textbackslash n"} +\!\!+\ unlines\ xs \end{array}$$

as below.

$$\begin{array}{l} unlines_{\mathrm{F}} :: [L^{\mathrm{T}}\ s\ String] \to L^{\mathrm{T}}\ s\ String \\ unlines_{\mathrm{F}}\ [\,] \qquad\ = new\ \texttt{""} \\ unlines_{\mathrm{F}}\ (x : xs) = lift2\ catLine_{\mathrm{L}}\ (x, unlines_{\mathrm{F}}\ xs) \end{array}$$

where $catLine_{\mathrm{L}}$ is a lens version of $\lambda x\ y \to x +\!\!+\ \texttt{"\textbackslash n"} +\!\!+\ y$. In the above, except for the noise of new and $lift2$, the definition is faithful to the original structure of $unlines$' definition, in an applicative style. With the heavy-lifting done in defining the lens function $unlines_{\mathrm{F}}$, a corresponding lens $unlines_{\mathrm{L}} :: L\ [String]\ String$ is readily available through straightforward unlifting: $unlines_{\mathrm{L}} = unliftT\ unlines_{\mathrm{F}}$.

We demonstrate the expressiveness of our system through a realistic example of a bidirectional evaluator for a higher-order programming language (Section 5), followed by discussions of smooth integration of our framework with both lenses and bidirectionalization approaches (Section 6). We discuss related techniques in Section 7, in particularly making connection to semantic bidirectionalization [21, 22, 33, 41] and conclude in Section 8. An implementation of our idea is available from `https://hackage.haskell.org/package/app-lens`.

Notes on Proofs and Examples. Due to the space restriction, we omit many of the proofs in this paper, but note that some of the proofs are based on free theorems [34, 37]. To simplify the formal discussion, we assume that all functions except puts are total and no data structure contains $\perp$. To deal with the partiality of puts, we assume that a put function of type $A \to B \to A$ can be represented as a total function of type $A \to B \to Maybe\ A$, which upon termination will produce either a value $Just\ a$ or an error $Nothing$.

We strive to balance the practicality and clarity of examples. Very often we deliberately choose small but hopefully still illuminating examples aiming at directly demonstrating the and only the theoretical issue being addressed. In addition, we include in Section 5 a sizeable application and would like to refer interested readers to `https://bitbucket.org/kztk/app-lens` for examples ranging from some general list functions in Prelude to the specific problem of XML transformations.

2. Bidirectional Transformations as Functions

Conventionally, bidirectional transformations are represented directly as pairs of functions [9, 13, 14, 16, 20–22, 25, 33, 35, 36, 38–41] (see the datatype L defined in Section 1). In this paper, we use lenses to refer specifically bidirectional transformations in this representation.

Lenses can be constructed and reasoned compositionally. For example, with the composition operator "$\hat{\circ}$"

$$\begin{array}{l} (\hat{\circ}) :: L\ b\ c \to L\ a\ b \to L\ a\ c \\ (L\ get_2\ put_2)\ \hat{\circ}\ (L\ get_1\ put_1) = \\ \quad L\ (get_2 \circ get_1)\ (\lambda s\ v \to put_1\ s\ (put_2\ (get_1\ s)\ v)) \end{array}$$

we can compose fst_{L} to itself to obtain a lens that operates on nested pairs, as below.

$$\begin{array}{l} fstTri_{\mathrm{L}} :: L\ ((a,b),c)\ a \\ fstTri_{\mathrm{L}} = fst_{\mathrm{L}}\ \hat{\circ}\ fst_{\mathrm{L}} \end{array}$$

Well-behavedness is preserved by such compositions: $fstTri_{\mathrm{L}}$ is well-behaved by construction assuming well-behaved fst_{L}.

The composition operator "$\hat{\circ}$" has the identity lens id_{L} as its unit.

$$\begin{array}{l} id_{\mathrm{L}} :: L\ a\ a \\ id_{\mathrm{L}} = L\ id\ (\lambda\_\ v \to v) \end{array}$$

2.1 Basic Idea: A Functional Representation Inspired by Yoneda

Our goal is to develop a representation of bidirectional transformations such that we can apply them, pass them to higher-order functions and reason about well-behavedness compositionally.

Inspired by the Yoneda embedding in category theory [19], we lift lenses of type $L\ a\ b$ to polymorphic functions of type

$$\forall s.\ L\ s\ a \to L\ s\ b$$

by lens composition

$$lift :: L\ a\ b \rightarrow (\forall s.\, L\ s\ a \rightarrow L\ s\ b)$$
$$lift\ \ell = \lambda x \rightarrow \ell \mathbin{\hat{\circ}} x$$

Intuitively, a lens of type $L\ s\ A$ with the universally quantified type variable s can be seen as an updatable datum of type A, and a lens of type $L\ A\ B$ as a transformation of type $\forall s.\, L\ s\ A \rightarrow L\ s\ B$ on updatable data. We call such lifted lenses *lens functions*.

The lifting function *lift* is injective, and has the following left inverse.

$$unlift :: (\forall s.\, L\ s\ a \rightarrow L\ s\ b) \rightarrow L\ a\ b$$
$$unlift\ f = f\ id_{\mathrm{L}}$$

Since lens functions are normal functions, they can be composed and passed to higher-order functions in the usual way. For example, $fstTri_{\mathrm{L}}$ can now be defined with the usual function composition.

$$fstTri_{\mathrm{L}} :: L\ ((a, b), c)\ a$$
$$fstTri_{\mathrm{L}} = unlift\ (lift\ fst_{\mathrm{L}} \circ lift\ fst_{\mathrm{L}})$$

Alternatively in a more applicative style, we can use a higher-order function $twice :: (a \rightarrow a) \rightarrow a \rightarrow a$ as below.

$$fstTri_{\mathrm{L}} = unlift\ (\lambda x \rightarrow twice\ (lift\ fst_{\mathrm{L}})\ x)$$
$$\quad \mathbf{where}\ twice\ f\ x = f\ (f\ x)$$

Like many category-theory inspired isomorphisms, this functional representation of bidirectional transformations is not unknown [7]; but its formal properties and applications in practical programming have not been investigated before.

2.2 Formal Properties of Lens Functions

We reconfirm that *lift* is injective with *unlift* as its left inverse.

Proposition 1. *$unlift\ (lift\ \ell) = \ell$ for all lenses $\ell :: L\ A\ B$.* □

We say that a function f *preserves well-behavedness*, if $f\ \ell$ is well-behaved for any well-behaved lens ℓ. Functions *lift* and *unlift* have the following desirable properties.

Proposition 2. *$lift\ \ \ell$ preserves well-behavedness if ℓ is well-behaved.* □

Proposition 3. *$unlift\ \ f$ is well-behaved if f preserves well-behavedness.* □

As it stands, the type L is open and it is possible to define lens functions through pattern-matching on the constructor. For example

$$f :: Eq\ a \Rightarrow L\ s\ (Maybe\ a) \rightarrow L\ s\ (Maybe\ a)$$
$$f\ (L\ g\ p) = L\ g\ (\lambda s\ v \rightarrow \mathbf{if}\ v == g\ s\ \mathbf{then}\ s$$
$$\qquad\qquad\qquad \mathbf{else}\ p\ (p\ s\ Nothing)\ v)$$

Here the input lens is pattern matched and the *get*/*put* components are used directly in constructing the output lens, which breaks encapsulation and blocks compositional reasoning of behaviors.

In our framework the intention is that all lens functions are constructed through lifting, which sees bidirectional transformations as atomic objects. Thus, we require that L is used as an "abstract type" in defining lens functions of type $\forall s.\, L\ s\ A \rightarrow L\ s\ B$. That is, we require the following conditions.

- L values must be constructed by lifting.
- L values must not be destructed.

This requirement is formally written as follows.

Definition 1 (Abstract Nature of L)**.** We say L *is abstract* in $f :: \tau$ if there is a polymorphic function h of type

$$\forall \ell.\, (\forall a\ b.\, L\ a\ b \rightarrow (\forall s.\, \ell\ s\ a \rightarrow \ell\ s\ b))$$
$$\quad \rightarrow (\forall a\ b.\, (\forall s.\, \ell\ s\ a \rightarrow \ell\ s\ b) \rightarrow L\ a\ b) \rightarrow \tau'$$

where $\tau' = \tau[\ell/L]$ and $f = h\ lift\ unlift$. □

Essentially, the polymorphic ℓ in h's type prevents us from using the constructor L directly, while the first functional argument of h (which is *lift*) provides the means to create L values.

Now the compositional reasoning of well-behavedness extends to lens functions; we can use a logical relation [31] to characterize well-behavedness for higher-order functions. As an instance, we can state that functions of type $\forall s.\, L\ s\ A \rightarrow L\ s\ B$ are well-behavedness preserving as follows.

Theorem 1. *Let $f :: \forall s.\, L\ s\ A \rightarrow L\ s\ B$ be a function in which L is abstract. Suppose that all applications of lift in the definition of f are to well-behaved lenses. Then, f preserves well-behavedness, and thus $unlift\ f$ is well-behaved.* □

2.3 Guaranteeing Abstraction

Theorem 1 requires the condition that L is abstract in f, which can be enforced by using abstract types through module systems. For example, in Haskell, we can define the following module to abstract L.

$$\mathbf{module}\ AbstractLens\ (L_{\mathrm{abs}}, lift_{\mathrm{abs}}, unlift_{\mathrm{abs}})\ \mathbf{where}$$
$$\mathbf{newtype}\ L_{\mathrm{abs}}\ a\ b = L_{\mathrm{abs}}\ \{unL_{\mathrm{abs}} :: L\ a\ b\}$$
$$lift_{\mathrm{abs}} :: L\ a\ b \rightarrow (\forall s.\, L_{\mathrm{abs}}\ s\ a \rightarrow L_{\mathrm{abs}}\ s\ b)$$
$$unlift_{\mathrm{abs}} :: (\forall s.\, L_{\mathrm{abs}}\ s\ a \rightarrow L_{\mathrm{abs}}\ s\ b) \rightarrow L\ a\ b$$

Outside the module $AbstractLens$, we can use $lift_{\mathrm{abs}}$, $unlift_{\mathrm{abs}}$ and type L_{abs} itself, but not the constructor of L_{abs}. Thus the only way to access data of type L is through $lift_{\mathrm{abs}}$ and $unlift_{\mathrm{abs}}$.

A consequence of having abstract L is that *lift* is now surjective (and *unlift* is now injective). We can prove the following property using the free theorems [34, 37].

Lemma 1. *Let f be a function of type $\forall s.\, L\ s\ A \rightarrow L\ s\ B$ in which L is abstract. Then $f\ \ell = f\ id_{\mathrm{L}} \mathbin{\hat{\circ}} \ell$ holds for all $\ell :: L\ S\ A$.* □

Correspondingly, we also have that *unlift* is injective on lens functions.

Theorem 2. *For any $f :: \forall s.\, L\ s\ A \rightarrow L\ s\ B$ in which L is abstract, $lift\ (unlift\ f) = f$ holds.* □

In the rest of this paper, we always assume abstract L unless specially mentioned otherwise.

2.4 Categorical Notes

As mentioned earlier, our idea of mapping $L\ A\ B$ to $\forall s.\, L\ s\ A \rightarrow L\ s\ B$ is based on the Yoneda lemma in category theory (Section III.2 in [19]). Since our purpose of this paper is not categorical formalization, we briefly introduce an analogue of the Yoneda lemma that is enough for our discussion.

Theorem 3 (An Analogue of the Yoneda Lemma (Section III.2 in [19]))**.** *A pair of functions $(lift, unlift)$ is a bijection between*

- *$\{\ell :: L\ A\ B\}$, and*
- *$\{f :: \forall s.\, L\ s\ A \rightarrow L\ s\ B \mid f\ x \mathbin{\hat{\circ}} y = f\ (x \mathbin{\hat{\circ}} y)\}$.* □

The condition $f\ x \mathbin{\hat{\circ}} y = f\ (x \mathbin{\hat{\circ}} y)$ is required to make f a natural transformation between functors $L\ (-)\ A$ and $L\ (-)\ B$; here, the contravariant functor $L\ (-)\ A$ maps a lens ℓ of type $L\ Y\ X$ to a function $(\lambda y \rightarrow y \mathbin{\hat{\circ}} \ell)$ of type $L\ X\ A \rightarrow L\ Y\ A$. Note that $f\ x \mathbin{\hat{\circ}} y = f\ (x \mathbin{\hat{\circ}} y)$ is equivalent to $f\ x = f\ id_{\mathrm{L}} \mathbin{\hat{\circ}} x$. Thus the naturality conditions imply Theorem 2.

In the above, we have implicitly considered the category of (possibly non-well-behaved) lenses, in which objects are types (sets in our setting) and morphisms from A to B are lenses of type $L\ A\ B$. This category of lenses is monoidal [15] but not closed [30], and thus has no higher-order functions. That is, there is no type $X\ B\ C$ such that there is a bijection between $L\ (A, B)\ C$ and $L\ A\ (X\ B\ C)$, which can be easily checked by comparing cardinalities. Our discussion does not conflict with this fact. What we state is that, for any s, $(L\ s\ A, L\ s\ B) \to L\ s\ C$ is isomorphic to $L\ s\ A \to (L\ s\ B \to L\ s\ C)$ via standard *curry* and *uncurry*; note that s is quantified globally.

Also note that $L\ s\ (-)$ is a functor that maps a lens ℓ to a function $\mathit{lift}\ \ell$. It is not difficult to check that $\mathit{lift}\ x \circ \mathit{lift}\ y = \mathit{lift}\ (x \mathbin{\hat{\circ}} y)$ and $\mathit{lift}\ (\mathit{id}_{\mathrm{L}} :: L\ A\ A) = (\mathit{id} :: L\ s\ A \to L\ s\ A)$.

3. Lifting n-ary Lenses and Flexible Duplication

So far we have presented a system that lifts lenses to functions, manipulates the functions, and then "unlifts" the results to construct composite lenses. One example is $\mathit{fstTri}_{\mathrm{L}}$ from Section 2 reproduced below.

$$\begin{aligned}&\mathit{fstTri}_{\mathrm{L}} :: L\ ((a, b), c)\ a\\ &\mathit{fstTri}_{\mathrm{L}} = \mathit{unlift}\ (\mathit{lift}\ \mathit{fst}_{\mathrm{L}} \circ \mathit{lift}\ \mathit{fst}_{\mathrm{L}})\end{aligned}$$

Astute readers may have already noticed the type $L\ ((a, b), c)\ a$ which is subtly distinct from $L\ (a, b, c)\ a$. One reason for this is with the definition of $\mathit{fstTri}_{\mathrm{L}}$, which consists of the composition of lifted $\mathit{fst}_{\mathrm{L}}$s. But more fundamentally it is the type of $\mathit{lift}\ (L\ x\ y \to (\forall s.\ L\ s\ x \to L\ s\ y))$, which treats x as a black box, that has prevented us from rearranging the tuple components.

Let's illustrate the issue with an even simpler example that goes directly to the heart of the problem.

$$\begin{aligned}&\mathit{swap}_{\mathrm{L}} :: L\ (a, b)\ (b, a)\\ &\mathit{swap}_{\mathrm{L}} = \ldots\end{aligned}$$

Following the programming pattern developed so far, we would like to construct this lens with the familiar unidirectional function $\mathit{swap} :: (a, b) \to (b, a)$. But since *lift* only produces *unary* functions of type $\forall s.\ L\ s\ A \to L\ s\ B$, despite the fact that A and B are actually pair types here, there is no way to compose *swap* with the resulting lens function.

In order to construct $\mathit{swap}_{\mathrm{L}}$ and many other lenses, including $\mathit{unlines}_{\mathrm{L}}$ in Section 1, a conversion of values of type $\forall s.\ (L\ s\ A_1, \ldots, L\ s\ A_n)$ to values of type $\forall s.\ L\ s\ (A_1, \ldots, A_n)$ is needed. In this section we look at how such a conversion can be defined for binary lenses, which can be easily extended to arbitrary n-ary cases.

3.1 Caveats of the Duplication Lens

To define a function of type $\forall s.\ (L\ s\ A, L\ s\ B) \to L\ s\ (A, B)$, we use the duplication lens $\mathit{dup}_{\mathrm{L}}$ (also known as *copy* elsewhere [9]) defined as below. For simplicity, we assume that $(==)$ represents observational equivalence.

$$\begin{aligned}&\mathit{dup}_{\mathrm{L}} :: \mathit{Eq}\ s \Rightarrow L\ s\ (s, s)\\ &\mathit{dup}_{\mathrm{L}} = L\ (\lambda s \to (s, s))\ (\lambda\_\ (s, t) \to r\ s\ t)\\ &\quad \mathbf{where}\ r\ s\ t \mid s == t = s \quad \text{-- } \textit{This will cause a problem.}\end{aligned}$$

With the duplication lens, the above-mentioned function can be defined as

$$\begin{aligned}&(\circledast) :: \mathit{Eq}\ s \Rightarrow L\ s\ a \to L\ s\ b \to L\ s\ (a, b)\\ &x \circledast y = (x \mathbin{\hat{\otimes}} y) \mathbin{\hat{\circ}} \mathit{dup}_{\mathrm{L}}\end{aligned}$$

where $(\hat{\otimes})$ is a lens combinator that combines two lenses applying to each component of a pair [9]:

$$\begin{aligned}&(\hat{\otimes}) :: L\ a\ a' \to L\ b\ b' \to L\ (a, b)\ (a', b')\\ &(L\ \mathit{get}_1\ \mathit{put}_1) \mathbin{\hat{\otimes}} (L\ \mathit{get}_2\ \mathit{put}_2) =\\ &\quad L\ (\lambda(a, b) \to (\mathit{get}_1\ a, \mathit{get}_2\ b))\\ &\quad\quad (\lambda(a, b)\ (a', b') \to (\mathit{put}_1\ a\ a', \mathit{put}_2\ b\ b'))\end{aligned}$$

We call $(\circledast)$ "split" in this paper. With $(\circledast)$ we can support the lifting of binary lenses as below.

$$\begin{aligned}&\mathit{lift2} :: L\ (a, b)\ c \to (\forall s.\ (L\ s\ a, L\ s\ b) \to L\ s\ c)\\ &\mathit{lift2}\ \ell\ (x, y) = \mathit{lift}\ \ell\ (x \circledast y)\end{aligned}$$

It is tempting to have the following as the inverse for *lift2*.

$$\begin{aligned}&\mathit{unlift2} :: (\forall s.\ (L\ s\ a, L\ s\ b) \to L\ s\ c) \to L\ (a, b)\ c\\ &\mathit{unlift2}\ f = f\ (\mathit{fst}_{\mathrm{L}}, \mathit{snd}_{\mathrm{L}})\end{aligned}$$

But $\mathit{unlift2} \circ \mathit{lift2}$ does not result in identity:

$$\begin{aligned}&(\mathit{unlift2} \circ \mathit{lift2})\ \ell\\ =&\ \{\ \text{definition unfolding \& } \beta\text{-reduction}\ \}\\ &\ell \mathbin{\hat{\circ}} (\mathit{fst}_{\mathrm{L}} \circledast \mathit{snd}_{\mathrm{L}})\\ =&\ \{\ \text{unfolding } (\circledast)\ \}\\ &\ell \mathbin{\hat{\circ}} (\mathit{fst}_{\mathrm{L}} \mathbin{\hat{\otimes}} \mathit{snd}_{\mathrm{L}}) \mathbin{\hat{\circ}} \mathit{dup}_{\mathrm{L}}\\ =&\ \{\ \text{definition unfolding}\ \}\\ &\ell \mathbin{\hat{\circ}} \mathit{block}_{\mathrm{L}}\ \mathbf{where}\\ &\quad \mathit{block}_{\mathrm{L}} = L\ \mathit{id}\ (\lambda s\ v \to \mathbf{if}\ s == v\ \mathbf{then}\ v\ \mathbf{else}\ \bot)\end{aligned}$$

Lens $\mathit{block}_{\mathrm{L}}$ is not a useful lens because it blocks any update to the view. Consequently any lenses composed with it become useless too.

3.2 Flexible and Safe Duplication by Tagging

In the above, the equality comparison $s == v$ that makes $\mathit{unlift2} \circ \mathit{lift2}$ useless has its root in $\mathit{dup}_{\mathrm{L}}$. If we look at the lens $\mathit{dup}_{\mathrm{L}}$ in isolation, there seems to be no alternative. The two duplicated values have to remain equal for the bidirectional laws to hold. However, if we consider the context in which $\mathit{dup}_{\mathrm{L}}$ is applied, there is more room for maneuver. Let us consider the lifting function *lift2* again, and how $\mathit{put}\ \mathit{dup}_{\mathrm{L}}$, which rejects the update above, works in the execution of $\mathit{put}\ (\mathit{unlift2}\ (\mathit{lift2}\ \mathit{id}_{\mathrm{L}}))$.

$$\begin{aligned}&\mathit{put}\ (\mathit{unlift2}\ (\mathit{lift2}\ \mathit{id}_{\mathrm{L}}))\ (1, 2)\ (\underline{3}, \underline{4})\\ =&\ \{\ \text{simplification}\ \}\\ &\mathit{put}\ ((\mathit{fst}_{\mathrm{L}} \mathbin{\hat{\otimes}} \mathit{snd}_{\mathrm{L}}) \mathbin{\hat{\circ}} \mathit{dup}_{\mathrm{L}})\ (1, 2)\ (\underline{3}, \underline{4})\\ =&\ \{\ \text{definition unfolding \& } \beta\text{-reduction}\ \}\\ &\mathit{put}\ \mathit{dup}_{\mathrm{L}}\ (1, 2)\ (\mathit{put}\ \mathit{fst}_{\mathrm{L}}\ (1, 2)\ \underline{3}, \mathit{put}\ \mathit{snd}_{\mathrm{L}}\ (1, 2)\ \underline{4})\\ =&\ \{\ \beta\text{-reduction}\ \}\\ &\mathit{put}\ \mathit{dup}_{\mathrm{L}}\ (1, 2)\ ((\underline{3}, 2), (1, \underline{4}))\end{aligned}$$

The last call to $\mathit{put}\ \mathit{dup}_{\mathrm{L}}$ above will fail because $(\underline{3}, 2) \not\equiv (1, \underline{4})$. But if we look more carefully, there is no reason for this behavior: $\mathit{lift2}\ \mathit{idB}$ should be able to update the two elements of the pair independently. Indeed in the *put* execution above, relevant values to the view change as highlighted by underlining are only compared for equality with irrelevant values. That is to say, we should be able to relax the equality check in $\mathit{dup}_{\mathrm{L}}$ and update the old source $(1, 2)$ to $(\underline{3}, \underline{4})$ without violating bidirectional laws.

To achieve this, we tag the values according to their relevance to view updates [25].

$$\mathbf{data}\ \mathit{Tag}\ a = U\ \{\mathit{unTag} :: a\} \mid O\ \{\mathit{unTag} :: a\}$$

Tag U (representing Updated) means the tagged value *may be relevant* to the view update and O (representing Original) means the tagged value *must not be relevant* to the view update. The idea is that O-tagged values can be altered without violating the bidirectional laws, as the new $\mathit{dup}_{\mathrm{L}}$ below.

$$\begin{aligned}&\mathit{dup}_{\mathrm{L}} :: \mathit{Poset}\ s \Rightarrow L\ s\ (s, s)\\ &\mathit{dup}_{\mathrm{L}} = L\ (\lambda s \to (s, s))\ (\lambda\_\ (s, t) \to s \curlyvee t)\end{aligned}$$

Here, *Poset* is a type class for partially-ordered sets that has a method $(\curlyvee)$ (pronounced as "lub") to compute least upper bounds.

$$\mathbf{class}\ \mathit{Poset}\ s\ \mathbf{where}\ (\curlyvee) :: s \to s \to s$$

We require that $(\curlyvee)$ must be associative, commutative and idempotent; but unlike a semilattice, $(\curlyvee)$ can be partial. Tagged elements and their (nested) pairs are ordered as follows.

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Poset\ (Tag\ a)\ \textbf{where} \\
\quad (O\ s) \curlyvee (U\ t) \qquad\quad = U\ t \\
\quad (U\ s) \curlyvee (O\ t) \qquad\quad = U\ s \\
\quad (O\ s) \curlyvee (O\ t)\ |\ s \mathrel{==} t = O\ s \\
\quad (U\ s) \curlyvee (U\ t)\ |\ s \mathrel{==} t = U\ s \\
\textbf{instance}\ (Poset\ a, Poset\ b) \Rightarrow Poset\ (a, b)\ \textbf{where} \\
\quad (a, b) \curlyvee (a', b') = (a \curlyvee a', b \curlyvee b')
\end{array}
$$

We also introduce the following type synonym for brevity.[1]

$$\textbf{type}\ L^{\mathrm{T}}\ s\ a = Poset\ s \Rightarrow L\ s\ a$$

As we will show later, the move from L to L^{T} will have implications on well-behavedness.

Accordingly, we change the types of $(\circledast)$, *lift* and *lift2* as below.

$$
\begin{array}{l}
(\circledast) :: L^{\mathrm{T}}\ s\ a \to L^{\mathrm{T}}\ s\ b \to L^{\mathrm{T}}\ s\ (a, b) \\
lift :: L\ a\ b \to (\forall s.\, L^{\mathrm{T}}\ s\ a \to L^{\mathrm{T}}\ s\ b) \\
lift2 :: L\ (a, b)\ c \to (\forall s.\, (L^{\mathrm{T}}\ s\ a, L^{\mathrm{T}}\ s\ b) \to L^{\mathrm{T}}\ s\ c)
\end{array}
$$

And adapt the definitions of *unlift* and *unlift2* to properly handle the newly introduced tags.

$$
\begin{array}{l}
unlift :: Eq\ a \Rightarrow (\forall s.\, L^{\mathrm{T}}\ s\ a \to L^{\mathrm{T}}\ s\ b) \to L\ a\ b \\
unlift\ f = f\ id'_{\mathrm{L}} \mathbin{\hat{\circ}} tag_{\mathrm{L}} \\
id'_{\mathrm{L}}\ \ :: L^{\mathrm{T}}\ (Tag\ a)\ a \\
id'_{\mathrm{L}}\ \ = L\ unTag\ (const\ U) \\
tag_{\mathrm{L}} :: L\ a\ (Tag\ a) \\
tag_{\mathrm{L}} = L\ O\ (const\ unTag) \\
\\
unlift2 :: \quad (Eq\ a, Eq\ b) \Rightarrow \\
\qquad\qquad\quad (\forall s.\, (L^{\mathrm{T}}\ s\ a, L^{\mathrm{T}}\ s\ b) \to L^{\mathrm{T}}\ s\ c) \to L\ (a, b)\ c \\
unlift2\ f = f\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} tag2_{\mathrm{L}} \\
fst'_{\mathrm{L}} :: L^{\mathrm{T}}\ (Tag\ a, Tag\ b)\ a \\
fst'_{\mathrm{L}} = L\ (\lambda(a, \_) \to unTag\ a)\ (\lambda(\_, b)\ a \to (U\ a, b)) \\
snd'_{\mathrm{L}} :: L^{\mathrm{T}}\ (Tag\ a, Tag\ b)\ b \\
snd'_{\mathrm{L}} = L\ (\lambda(\_, b) \to unTag\ b)\ (\lambda(a, \_)\ b \to (a, U\ b)) \\
tag2_{\mathrm{L}} :: L\ (a, b)\ (Tag\ a, Tag\ b) \\
tag2_{\mathrm{L}} = L\ (\lambda(a, b) \to (O\ a, O\ b)) \\
\qquad\qquad\quad (\lambda\_\ (a, b) \to (unTag\ a, unTag\ b))
\end{array}
$$

We need to change *unlift* because it may be applied to functions calling *lift2* internally. In what follows, we only focus on *lift2* and *unlift2*, and expect the discussion straightforwardly extends to *lift* and the new *unlift*.

We can now show that the new *unlift2* is the left-inverse of *lift2*.

Proposition 4. *$unlift2\ \ (lift2\ \ \ell) = \ell$ holds for all lenses $\ell :: L\ (A, B)\ C$.*

Proof. We prove the statement with the following calculation.

$$
\begin{array}{l}
\quad unlift2\ (lift2\ \ell) \\
= \{\ \text{definition unfolding \& } \beta\text{-reduction}\ \} \\
\quad \ell \mathbin{\hat{\circ}} fst'_{\mathrm{L}} \circledast snd'_{\mathrm{L}} \mathbin{\hat{\circ}} tag2_{\mathrm{L}} \\
= \{\ \text{unfolding } (\circledast)\ \} \\
\quad \ell \mathbin{\hat{\circ}} (fst'_{\mathrm{L}} \mathbin{\hat{\otimes}} snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}} \mathbin{\hat{\circ}} tag2_{\mathrm{L}} \\
= \{\ (fst'_{\mathrm{L}} \mathbin{\hat{\otimes}} snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}} \mathbin{\hat{\circ}} tag2_{\mathrm{L}} = id_{\mathrm{L}} \text{ — (*)}\ \} \\
\quad \ell
\end{array}
$$

We prove the statement (*) by showing $get\ ((fst'_{\mathrm{L}} \mathbin{\hat{\otimes}} snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}} \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (a, b) = (a, b)$ and $put\ ((fst'_{\mathrm{L}} \mathbin{\hat{\otimes}} snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}} \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (a, b)\ (a', b') = (a', b')$. Since the former property is easy to prove, we only show the latter here.

$$
\begin{array}{l}
\quad put\ ((fst'_{\mathrm{L}} \mathbin{\hat{\otimes}} snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}} \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (a, b)\ (a', b') \\
= \{\ \text{definition unfolding \& } \beta\text{-reduction}\ \} \\
\quad put\ tag2_{\mathrm{L}}\ (a, b)\ \$ \\
\qquad put\ ((fst'_{\mathrm{L}} \mathbin{\hat{\otimes}} snd'_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}})\ (O\ a, O\ b)\ (a', b') \\
= \{\ \text{definition unfolding \& } \beta\text{-reduction}\ \} \\
\quad put\ tag2_{\mathrm{L}}\ (a, b)\ \$ \\
\qquad put\ dup_{\mathrm{L}}\ (O\ a, O\ b)\ \$ \\
\qquad\quad (put\ fst'_{\mathrm{L}}\ (O\ a, O\ b)\ a', put\ snd'_{\mathrm{L}}\ (O\ a, O\ b)\ b') \\
= \{\ \text{definitions of } fst'_{\mathrm{L}} \text{ and } snd'_{\mathrm{L}}\ \} \\
\quad put\ tag2_{\mathrm{L}}\ (a, b)\ \$ \\
\qquad put\ dup_{\mathrm{L}}\ (O\ a, O\ b)\ ((U\ a', O\ b), (O\ a, U\ b')) \\
= \{\ \text{definition of } dup_{\mathrm{L}}\ \} \\
\quad put\ tag2_{\mathrm{L}}\ (a, b)\ (U\ a', U\ b') \\
= \{\ \text{definition of } tag2_{\mathrm{L}}\ \} \\
\quad (a', b')
\end{array}
$$

Thus, we have proved that *lift2* is injective. ☐

We can recreate fst_{L} and snd_{L} with *unlift2*, which is rather reassuring.

Proposition 5. *$fst_{\mathrm{L}} = unlift2\ fst$ and $snd_{\mathrm{L}} = unlift2\ snd$.* ☐

Note that now *unlift* and *unlift2* are *no longer* injective (even with abstract L); there exist functions that are not equivalent but coincide after unlifting. An example of such is the pair $lift2\ fst_{\mathrm{L}}$ and *fst*: while unlifting both functions result in fst_{L}, they actually differ as $put\ (lift2\ fst_{\mathrm{L}}\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}}))\ (O\ a, O\ b)\ c = (U\ c, U\ b)$ and $put\ (fst\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}}))\ (O\ a, O\ b)\ c = (U\ c, O\ b)$. Intuitively, *fst* knows that the second argument is unused, while $lift2\ fst_{\mathrm{L}}$ does not because fst_{L} is treated as a black box by *lift2*. In other words, the relationship between the lifting/unlifting functions and the Yoneda Lemma discussed in Section 2 ceases to exist in this new context. Nevertheless, the counter-example scenario described here is contrived and will not affect practical programming in our framework.

Another side effect of this new development with tags is that the original bidirectional laws, i.e., the well-behavedness, are temporarily broken during the execution of *lift2* and *unlift2* by the new internal functions fst'_{L}, snd'_{L}, dup_{L} and $tag2_{\mathrm{L}}$. Consequently, we need a new theoretical development to establish the preservation of well-behavedness by the lifting/unlifting process.

3.3 Relevance-Aware Well-Behavedness

We have noted that the new internal functions dup_{L}, fst'_{L}, snd'_{L} and $tag2_{\mathrm{L}}$ are not well-behaved, for different reasons. For functions fst'_{L} and snd'_{L}, the difference from the original versions fst_{L} and snd_{L} is only in the additional wrapping/unwrapping that is needed to adapt to the existence of tags. As a result, as long as these functions are used in an appropriate context, the bidirectional laws are expected to hold. But for dup_{L} and $tag2_{\mathrm{L}}$, the new definitions are more defined in the sense that some originally failing executions of *put* are now intentionally turned into successful ones. For this change in semantics, we need to adapt the laws to allow temporary violations and yet still establish well-behavedness of the resulting bidirectional transformations in the end. For example, we still want $unlift2\ f$ to be well-behaved for any $f :: \forall s.\, (L^{\mathrm{T}}\ s\ A, L^{\mathrm{T}}\ s\ B) \to L^{\mathrm{T}}\ s\ C$, as long as the lifting functions are applied to well-behaved lenses.

3.3.1 Relevance-Ordering and Lawful Duplications

Central to the discussion in this and the previous subsections is the behavior of dup_{L}. To maintain safety, unequal values as duplications are only allowed if they have different tags (i,e,. one value must be

[1] Actually, we will have to use **newtype** for the code in this paper to pass GHC's type checking. We take a small deviation from GHC Haskell here in favor of brevity.

irrelevant to the update and can be discarded). We formalize such a property with the partial ordering between tagged values. Let us write $(\preceq)$ for the partial order induced from $\curlyvee$: that is, $s \preceq t$ if $s \curlyvee t$ is defined and equal to t. One can see that $(\preceq)$ is the reflexive closure of $O\ s \preceq U\ t$. We write $\uparrow s$ for a value obtained from s by replacing all O tags with U tags. Trivially, we have $s \preceq \uparrow s$. But there exists s' such that $s \preceq s'$ and $s' \neq \uparrow s$.

Now we can define a variant of well-behavedness local to the U-tagged elements.

Definition 2 (Local Well-Behavedness)**.** A bidirectional transformation $\ell :: L^{\mathrm{T}}\ a\ b$ is called *locally well-behaved* if the following four conditions hold.

- (**Forward Tag-Irrelevance**) If $v = get\ \ell\ s$, then for all s' such that $\uparrow s' = \uparrow s$, $v = get\ \ell\ s'$ holds.
- (**Backward Inflation**) For all minimal (with respect to $\preceq$) s, if $put\ \ell\ s\ v$ succeeds as s', then $s \preceq s'$.
- (**Local Acceptability**) For all s, $s \preceq put\ \ell\ s\ (get\ \ell\ s) \preceq \uparrow s$.
- (**Local Consistency**) For all s and v, assuming $put\ \ell\ s\ v$ succeeds as s', then for all s'' with $s' \preceq s''$, $get\ \ell\ s'' = v$ holds. □

In the above, tags introduced for the flexible behavior of put must not affect the behavior of get: $\uparrow s' = \uparrow s$ means that s and s' are equal if tags are ignored. The property local-acceptability is similar to acceptability, except that O-tags are allowed to change to U-tags. The property local consistency is stronger than consistency in the sense that get must map all values sharing the same U-tagged elements with s' to the same view. The idea is that O-tagged elements in s' are not connected to the view v, and thus changing them will not affect v. A similar reasoning applies to backward inflation stating that source elements changed by put will have U-tags. Note that in this definition of local well-behavedness, tags are assumed to appear only in the sources. As a matter of fact, only dup_{L} and $tag2_{\mathrm{L}}/tag_{\mathrm{L}}$ introduce tagged views; but they are always precomposed when used, as shown in the following.

We have the following compositional properties for local well-behavedness.

Lemma 2. *The following properties hold for bidirectional transformations x and y with appropriate types.*

- *If x is well-behaved and y is locally well-behaved, then $lift\ x\ y$ is locally well-behaved.*
- *If x and y are locally well-behaved, $x \circledast y$ is locally well-behaved.*
- *If x and y are locally well-behaved, $x\ \hat{\circ}\ tag2_{\mathrm{L}}$ and $y\ \hat{\circ}\ tag_{\mathrm{L}}$ are well-behaved.*

Proof. We only prove the second property, which is the most nontrivial one among the three, although we would like to note that forward tag-irrelevance is used to prove the third property.

We first show local acceptability.

$$\begin{aligned}
& put\ ((x \mathbin{\hat{\otimes}} y)\ \hat{\circ}\ dup_{\mathrm{L}})\ s\ (get\ ((x \mathbin{\hat{\otimes}} y)\ \hat{\circ}\ dup_{\mathrm{L}})\ s) \\
=\ & \{\ \text{simplification}\ \} \\
& put\ dup_{\mathrm{L}}\ s\ (put\ (x \mathbin{\hat{\otimes}} y)\ (s,s)\ (get\ (x \mathbin{\hat{\otimes}} y)\ (s,s))) \\
=\ & \{\ \text{by the local acceptability of } x \mathbin{\hat{\otimes}} y\ \} \\
& put\ dup_{\mathrm{L}}\ s\ (s', s'') \quad \text{— } \textit{where}\ s \preceq s' \preceq \uparrow s,\ s \preceq s'' \preceq \uparrow s \\
=\ & \{\ \text{by the definition of } dup_{\mathrm{L}} \text{ and that } s' \curlyvee s'' \text{ is defined}\ \} \\
& s' \curlyvee s'' \preceq \uparrow s
\end{aligned}$$

Note that, since $s' \preceq \uparrow s$ and $s'' \preceq \uparrow s$, there is $s' \curlyvee s'' \preceq \uparrow s$.

Then, we prove local consistency. Assume that $put\ ((x \mathbin{\hat{\otimes}} y)\ \hat{\circ}\ dup_{\mathrm{L}})\ s\ (v_1, v_2)$ succeeds in s'. Then, by the following calculation, we have $s' = put\ x\ s\ v_1 \curlyvee put\ y\ s\ v_2$.

$$\begin{aligned}
& put\ ((x \mathbin{\hat{\otimes}} y)\ \hat{\circ}\ dup_{\mathrm{L}})\ s\ (v_1, v_2) \\
=\ & \{\ \text{simplification}\ \} \\
& put\ dup_{\mathrm{L}}\ s\ (put\ x\ s\ v_1, put\ y\ s\ v_2) \\
=\ & \{\ \text{definition unfolding}\ \} \\
& put\ x\ s\ v_1 \curlyvee put\ y\ s\ v_2
\end{aligned}$$

Let s'' be a source such that $s' \preceq s''$. Then, we prove $get\ ((x \mathbin{\hat{\otimes}} y)@dup_{\mathrm{L}})\ s'' = (v_1, v_2)$ as follows.

$$\begin{aligned}
& get\ ((x \mathbin{\hat{\otimes}} y)\ \hat{\circ}\ dup_{\mathrm{L}})\ s''\ (v_1, v_2) \\
=\ & \{\ \text{simplification}\ \} \\
& (get\ x\ s'', get\ y\ s'') \\
=\ & \{\ \text{the local consistency of } x \text{ and } y\ \} \\
& (v_1, v_2)
\end{aligned}$$

Note that we have $put\ x\ s\ v_1 \preceq s' \preceq s''$ and $put\ y\ s\ v_2 \preceq s' \preceq s''$ by the definition of $\curlyvee$.

Forward tag-irrelevance and backward inflation are straightforward. □

Corollary 1. *The following properties hold.*

- *$lift\ \ell :: \forall s.\, L^{\mathrm{T}}\ s\ A \to L^{\mathrm{T}}\ s\ B$ preserves local well-behavedness, if $\ell :: L^{\mathrm{T}}\ A\ B$ is well-behaved.*
- *$lift2\ \ell :: \forall s.\, (L^{\mathrm{T}}\ s\ A, L^{\mathrm{T}}\ s\ B) \to L^{\mathrm{T}}\ s\ C$ preserves local well-behavedness, if $\ell :: L^{\mathrm{T}}\ (A,B)\ C$ is well-behaved.* □

Similar to the case in Section 2, compositional reasoning of well-behavedness requires the lens type L^{T} to be abstract.

Definition 3 (Abstract Nature of L^{T})**.** We say L^{T} *is abstract* in $f :: \tau$ if there is a polymorphic function h of type

$$\begin{aligned}
& \forall \ell.\, (\forall a\ b.\, L^{\mathrm{T}}\ a\ b \to (\forall s.\, \ell\ s\ a \to \ell\ s\ b)) \\
& \quad \to (\forall a\ b.\, (\forall s.\, \ell\ s\ a \to \ell\ s\ b) \to L^{\mathrm{T}}\ a\ b) \\
& \quad \to (\forall s\ a\ b.\, \ell\ s\ a \to \ell\ s\ b \to \ell\ s\ (a,b)) \\
& \quad \to (\forall a\ b\ c.\, (\forall s.\, (\ell\ s\ a, \ell\ s\ b) \to \ell\ x\ c) \to L^{\mathrm{T}}\ (a,b)\ c) \\
& \quad \to \tau'
\end{aligned}$$

satisfying $f = h\ lift\ unlift\ (\circledast)\ unlift2$ and $\tau' = \tau[\ell/L^{\mathrm{T}}]$. □

Then, we obtain the following properties from the free theorems [34, 37].

Theorem 4. *Let f be a function of type $\forall s.\, (L^{\mathrm{T}}\ s\ A, L^{\mathrm{T}}\ s\ B) \to L^{\mathrm{T}}\ s\ C$ in which L^{T} is abstract. Then, $f\ (x,y)$ is locally well-behaved if x and y are also locally well-behaved, assuming that lift is applied only to well-behaved lenses.* □

Corollary 2. *Let f be a function of type $\forall s.\, (L^{\mathrm{T}}\ s\ A, L^{\mathrm{T}}\ s\ B) \to L^{\mathrm{T}}\ s\ C$ in which L^{T} is abstract. Then, $unlift2\ f$ is well-behaved, assuming that lift is applied only to well-behaved lenses.* □

Example 1 (swap)**.** The bidirectional version of $swap$ can be defined as follows.

$$\begin{aligned}
& swap_{\mathrm{L}} :: (Eq\ a, Eq\ b) \Rightarrow L\ (a,b)\ (b,a) \\
& swap_{\mathrm{L}} = unlift2\ (lift2\ id_{\mathrm{L}} \circ swap)
\end{aligned}$$

And it behaves as expected.

$$\begin{aligned}
& put\ swap_{\mathrm{L}}\ (1,2)\ (4,3) \\
=\ & \{\ \text{unfold definitions}\ \} \\
& put\ ((snd'_{\mathrm{L}} \mathbin{\hat{\otimes}} fst'_{\mathrm{L}})\ \hat{\circ}\ dup_{\mathrm{L}}\ \hat{\circ}\ tag2_{\mathrm{L}})\ (1,2)\ (4,3) \\
=\ & \{\ \text{simplifications}\ \} \\
& put\ tag2_{\mathrm{L}}\ (1,2)\ \$ \\
& \quad put\ dup_{\mathrm{L}}\ (O\ 1, O\ 2)\ \$ \\
& \quad\quad (put\ snd'_{\mathrm{L}}\ (O\ 1, O\ 2)\ 4, put\ fst'_{\mathrm{L}}\ (O\ 1, O\ 2)\ 3) \\
=\ & \{\ \text{definition of } fst'_{\mathrm{L}} \text{ and } snd'_{\mathrm{L}}\ \} \\
& put\ tag2_{\mathrm{L}}\ (1,2)\ \$ \\
& \quad\quad put\ dup_{\mathrm{L}}\ (O\ 1, O\ 2)\ ((O\ 1, U\ 4), (U\ 3, O\ 2))
\end{aligned}$$

$= \{$ definitions of dup_{L} and $tag2_{\mathrm{L}}$ $\}$
$(3, 4)$ $\square$

It is worth mentioning that $(\circledast)$ is the base for "splitting" and "lifting" tuples of arbitrary arity. For example, the triple case is as follows.

$$
\begin{aligned}
& split3 :: (L^{\mathrm{T}}\ s\ a, L^{\mathrm{T}}\ s\ b, L^{\mathrm{T}}\ s\ c) \to L^{\mathrm{T}}\ s\ (a, b, c) \\
& split3\ (x, y, z) = lift\ flattenL_{\mathrm{L}}\ ((x \circledast y) \circledast z) \\
& \quad \mathbf{where}\ flattenL_{\mathrm{L}} :: L\ ((a, b), c)\ (a, b, c) \\
& \qquad flattenL_{\mathrm{L}} = L\ (\lambda((x, y), z) \to (x, y, z)) \\
& \qquad\qquad (\lambda\_\ (x, y, z) \to ((x, y), z)) \\
& lift3\ \ell\ t = lift\ \ell\ (split3\ t)
\end{aligned}
$$

For the family of unlifting functions, we additionally need n-ary versions of projection and tagging functions, which are straightforward to define.

In the above definition of $split3$, we have decided to nest to the left in the intermediate step. This choice is not essential.

$$
\begin{aligned}
& split3'\ (x, y, z) = lift\ flattenR_{\mathrm{L}}\ (x \circledast (y \circledast z)) \\
& \quad \mathbf{where}\ flattenR_{\mathrm{L}} :: L\ ((a, b), c)\ (a, b, c) \\
& \qquad flattenR_{\mathrm{L}} = L\ (\lambda(x, (y, z)) \to (x, y, z)) \\
& \qquad\qquad (\lambda\_\ (x, y, z) \to (x, (y, z)))
\end{aligned}
$$

The two definitions $split3$ and $split3'$ coincide.

To complete the picture, the nullary lens function

$$
\begin{aligned}
& unit :: \forall s.\ L^{\mathrm{T}}\ s\ () \\
& unit = L\ (\lambda\_ \to ())\ (\lambda s\ ()\to s)
\end{aligned}
$$

is the unit for $(\circledast)$. Theoretically $(L^{\mathrm{T}}\ s\ (-), \circledast, unit)$ forms a lax monoidal functor (Section XI.2 in [19]) under certain conditions (see Section 3.4). Practically, $unit$ enables us to define the following combinator.

$$
\begin{aligned}
& new :: Eq\ a \Rightarrow a \to \forall s.\ L^{\mathrm{T}}\ s\ a \\
& new\ a = lift\ (L\ (const\ a)\ (\lambda\_\ a' \to check\ a\ a'))\ unit \\
& \quad \mathbf{where} \\
& \qquad check\ a\ a' = \mathbf{if}\ a == a'\ \mathbf{then}\ () \\
& \qquad\qquad \mathbf{else}\ error\ \texttt{"Update on constant"}
\end{aligned}
$$

Function new lifts ordinary values into the bidirectional transformation system; but since the values are not from any source, they are not updatable. Nevertheless, this ability to lift constant values is very useful in practice [21, 22], as we will see in the examples to come.

3.4 Categorical Notes

Recall that $L\ S\ (-)$ is a functor from the category of lenses to the category of sets and (total) functions, which maps $\ell :: L\ A\ B$ to $lift\ \ell :: L\ S\ A \to L\ S\ B$ for any S. In the case that S is tagged and thus partially ordered, $(L^{\mathrm{T}}\ S\ (-), \circledast, unit)$ forms a lax monoidal functor, under the following conditions.

- $(\circledast)$ must be natural, i.e., $(lift\ f\ x) \circledast (lift\ g\ y) = lift\ (f \hat{\otimes} g)\ (x \circledast y)$ for all f, g, x and y with appropriate types.
- $split3$ and $split3'$ coincide.
- $lift\ elimUnitL_{\mathrm{L}}\ (unit \circledast x) = x$ must hold where $elimUnitL_{\mathrm{L}} :: L\ ((), a)\ a$ is the bidirectional version of elimination of $()$, and so does its symmetric version.

Intuitively, the second and the third conditions state that the mapping must respect the monoid structure of products, with the former concerning associativity and the latter concerning the identity elements. The first and second conditions above hold without any additional assumptions, whereas the third condition, which reduces to $s \curlyvee put\ x\ s\ v = put\ x\ s\ v$, is not necessarily true if s is not minimal (if s is minimal, this property holds by backward inflation). Recall that minimality of s implies that s can only have O-tags. To get around this restriction, we take $L^{\mathrm{T}}\ S\ A$ as a quotient set of $L\ S\ A$ by the equivalence relation $\equiv$ defined as $x \equiv y$ if $get\ x = get\ y \wedge put\ x\ s = put\ y\ s$ for all minimal s. This equivalence is preserved by manipulations of L^{T}-data; that is, the following holds for x, y, z and w with appropriate types.

- $x \equiv y$ implies $lift\ \ell\ x \equiv lift\ \ell\ y$ for any well-behaved lens ℓ.
- $x \equiv y$ and $z \equiv w$ implies $x \circledast z \equiv y \circledast w$.
- $x \equiv y$ implies $x\ \hat{\circ}\ tag_{\mathrm{L}} = y\ \hat{\circ}\ tag_{\mathrm{L}}$ (or $x\ \hat{\circ}\ tag2_{\mathrm{L}} = y\ \hat{\circ}\ tag2_{\mathrm{L}}$).

Note that the above three cases cover the only ways to construct/destruct L^{T} in f when L^{T} is abstract. The third condition says that this "coarse" equivalence $(\equiv)$ on L^{T} can be "sharpened" to the usual extensional equality $(=)$ by tag_{L} and $tag2_{\mathrm{L}}$ in the unlifting functions.

It is known that an $Applicative$ functor in Haskell corresponds to a monoidal functors [29]. However, we cannot use an $Applicative$-like interface because there is no exponentials in lenses [30]. Nevertheless, the same spirit of applicative-style programming centering around lambda abstractions and function applications is shared in our framework.

4. Going Generic

In this section, we make the ideas developed in previous sections practical by extending the technique to lists and other data structures.

4.1 Unlifting Functions on Lists

We have looked at how unlifting works for n-nary tuples in Section 3. And we now see how the idea can be extended to lists. As a typical usage scenario, if we apply map to a lens function $lift\ \ell$, we will obtain a function of type $map\ (lift\ \ell) :: [L^{\mathrm{T}}\ s\ A] \to [L^{\mathrm{T}}\ s\ B]$. But what we really would like is a lens of type $L\ [A]\ [B]$. The way to achieve this is to internally treat length-n lists as n-ary tuples. This treatment effectively restricts us to in-place updates of views (i.e., no change is allowed to the list structure); we will revisit this issue in more detail in Section 6.1.

First, we can "split" lists by repeated pair-splitting, as follows.

$$
\begin{aligned}
& lsequence_{\mathrm{list}} :: [L^{\mathrm{T}}\ s\ a] \to L^{\mathrm{T}}\ s\ [a] \\
& lsequence_{\mathrm{list}}\ [\,] \qquad = lift\ nil_{\mathrm{L}}\ unit \\
& lsequence_{\mathrm{list}}\ (x : xs) = lift2\ cons_{\mathrm{L}}\ (x, lsequence_{\mathrm{list}}\ xs) \\
& nil_{\mathrm{L}} = L\ (\lambda() \to [\,])\ (\lambda()\ [\,] \to ()) \\
& cons_{\mathrm{L}} = L\ (\lambda(a, as) \to (a : as)) \\
& \qquad\qquad (\lambda\_\ (a' : as') \to (a', as'))
\end{aligned}
$$

The name of this function is inspired by $sequence$ in Haskell. Then the lifting function is defined straightforwardly.

$$
\begin{aligned}
& lift_{\mathrm{list}} :: L\ [a]\ b \to \forall s.\ [L^{\mathrm{T}}\ s\ a] \to L^{\mathrm{T}}\ s\ b \\
& lift_{\mathrm{list}}\ \ell\ xs = lift\ \ell\ (lsequence_{\mathrm{list}}\ xs)
\end{aligned}
$$

Tagged lists form an instance of $Poset$.

$$
\begin{aligned}
& \mathbf{instance}\ Poset\ a \Rightarrow Poset\ [a]\ \mathbf{where} \\
& \quad xs \curlyvee ys = \mathbf{if}\ length\ xs == length\ ys \\
& \qquad\qquad \mathbf{then}\ zipWith\ (\curlyvee)\ xs\ ys \\
& \qquad\qquad \mathbf{else}\ \bot\ \ \texttt{--}\ \textit{Unreachable in our framework}
\end{aligned}
$$

Note that the requirement that xs and ys must has the same shape is made explicit above, though it is automatically enforced by the abstract use of L^{T} in lifted functions.

The definition of $unlift_{\mathrm{list}}$ is a bit more involved. What we need to do is to turn every element of the source list into a projection lens and apply the lens function f.

$$
\begin{aligned}
&\mathit{unlift}_{\mathrm{list}} :: \forall a\ b.\ \mathit{Eq}\ a \Rightarrow \\
&\qquad\qquad (\forall s.\, [L^{\mathrm{T}}\ s\ a] \to L^{\mathrm{T}}\ s\ b) \to L\ [a]\ b \\
&\mathit{unlift}_{\mathrm{list}}\ f = L\ (\lambda s \to \mathit{get}\ (\mathit{mkLens}\ s)\ s) \\
&\qquad\qquad (\lambda s \to \mathit{put}\ (\mathit{mkLens}\ s)\ s) \\
&\quad \mathbf{where} \\
&\quad\ \mathit{mkLens}\ s = f\ (\mathit{projs}\ (\mathit{length}\ s))\ \hat{\circ}\ \mathit{tagList}_{\mathrm{L}} \\
&\quad\ \mathit{tagList}_{\mathrm{L}} = L\ (\mathit{map}\ O)\ (\lambda\_\ \mathit{ys} \to \mathit{map}\ \mathit{unTag}\ \mathit{ys}) \\
&\quad\ \mathit{projs}\ n = \mathit{map}\ \mathit{proj}_{\mathrm{L}}\ [0 \mathinner{..} n-1] \\
&\quad\ \mathit{proj}_{\mathrm{L}} :: \mathit{Int} \to L^{\mathrm{T}}\ [\mathit{Tag}\ a]\ a \\
&\quad\ \mathit{proj}_{\mathrm{L}}\ i = \ L\ (\lambda \mathit{xs} \to \mathit{unTag}\ (\mathit{xs}\ !!\ i)) \\
&\qquad\qquad (\lambda \mathit{as}\ a \to \mathit{update}\ i\ (U\ a)\ \mathit{as})
\end{aligned}
$$

Giving that the need to inspect the length of the source leads to the separated definitions of get and put in the above, there might be worry that we may lose the guarantee of well-behaveness of the resulting lens. But this is not a problem here since the length of the source list is an invariant of the resulting lens. Similar to $\mathit{lift2}$, $\mathit{lift}_{\mathrm{list}}$ is an injection with $\mathit{unlift}_{\mathrm{list}}$ as its left inverse.

Example 2 (Bidirectional tail)**.** Let us consider the function tail.

$$
\begin{aligned}
&\mathit{tail} :: [a] \to [a] \\
&\mathit{tail}\ (x : \mathit{xs}) = \mathit{xs}
\end{aligned}
$$

A bidirectional version of tail is easily constructed by using $\mathit{lsequence}_{\mathrm{list}}$ and $\mathit{unlift}_{\mathrm{list}}$ as follows.

$$
\begin{aligned}
&\mathit{tail}_{\mathrm{L}} :: \mathit{Eq}\ a \Rightarrow L\ [a]\ [a] \\
&\mathit{tail}_{\mathrm{L}} = \mathit{unlift}_{\mathrm{list}}\ (\mathit{lsequence}_{\mathrm{list}} \circ \mathit{tail})
\end{aligned}
$$

The obtained lens $\mathit{tail}_{\mathrm{L}}$ supports all in-place updates, such as $\mathit{put}\ \mathit{tail}_{\mathrm{L}}\ [\texttt{"a"},\texttt{"b"},\texttt{"c"}]\ [\texttt{"B"},\texttt{"C"}] = [\texttt{"a"},\texttt{"B"},\texttt{"C"}]$. In contrast, any change on list length will be rejected; specifically $\mathit{nil}_{\mathrm{L}}$ or $\mathit{cons}_{\mathrm{L}}$ in $\mathit{lsequence}_{\mathrm{list}}$ throws an error. □

Example 3 (Bidirectional $\mathit{unlines}$)**.** Let us consider a bidirectional version of $\mathit{unlines} :: [\mathit{String}] \to \mathit{String}$ that concatenate lines, after appending a terminating newline to each. For example, $\mathit{unlines}\ [\texttt{"ab"},\texttt{"c"}] = \texttt{"ab\textbackslash nc\textbackslash n"}$. In conventional unidirectional programming, one can implement $\mathit{unlines}$ as follows.

$$
\begin{aligned}
&\mathit{unlines}\ [\,] \qquad\quad = \texttt{""} \\
&\mathit{unlines}\ (x : \mathit{xs}) = \mathit{catLine}\ x\ (\mathit{unlines}\ \mathit{xs}) \\
&\mathit{catLine}\ x\ y = x +\!\!+ \texttt{"\textbackslash n"} +\!\!+ y
\end{aligned}
$$

To construct a bidirectional version of $\mathit{unlines}$, we first need a bidirectional version of $\mathit{catLine}$.

$$
\begin{aligned}
&\mathit{catLine}_{\mathrm{L}} :: L\ (\mathit{String}, \mathit{String})\ \mathit{String} \\
&\mathit{catLine}_{\mathrm{L}} = \\
&\ \ L\ (\lambda(s,t) \to s +\!\!+ \texttt{"\textbackslash n"} +\!\!+ t) \\
&\quad (\lambda(s,t)\ u \to \mathbf{let}\ n = \mathit{length}\ (\mathit{filter}\ (== \texttt{'\textbackslash n'})\ s) \\
&\qquad\qquad\qquad\quad\ \ i = \mathit{elemIndices}\ \texttt{'\textbackslash n'}\ u\ !!\ n \\
&\qquad\qquad\qquad\quad\ \ (s', t') = \mathit{splitAt}\ i\ u \\
&\qquad\qquad\qquad\ \mathbf{in}\ (s', \mathit{tail}\ t'))
\end{aligned}
$$

Here, $\mathit{elemIndices}$ and $\mathit{splitAt}$ are functions from `Data.List`: $\mathit{elemIndices}\ c\ s$ returns the indices of all elements that are equal to c; $\mathit{splitAt}\ i\ x$ returns a tuple where the first element is x's prefix of length i and the second element is the remainder of the list. Intuitively, $\mathit{put}\ \mathit{catLine}_{\mathrm{L}}\ (s,t)\ u$ splits u into s' and $\texttt{"\textbackslash n"} +\!\!+ t'$ so that s' contains the same number of newlines as the original s. For example, $\mathit{put}\ \mathit{catLine}_{\mathrm{L}}\ (\texttt{"a\textbackslash nbc"},\texttt{"de"})\ \texttt{"A\textbackslash nB\textbackslash nC"} = (\texttt{"A\textbackslash nB"},\texttt{"C"})$.

Then, construction of a bidirectional version $\mathit{unlines}_{\mathrm{L}}$ of $\mathit{unlines}$ is straightforward; we only need to replace `""` with $\mathit{new}\ \texttt{""}$ and $\mathit{catLine}$ with $\mathit{lift2}\ \mathit{catLine}_{\mathrm{L}}$, and to apply $\mathit{unlift}_{\mathrm{list}}$ to obtain a lens.

$$
\begin{aligned}
&\mathit{unlines}_{\mathrm{L}} :: L\ [\mathit{String}]\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{L}} = \mathit{unlift}_{\mathrm{list}}\ \mathit{unlines}_{\mathrm{F}} \\
&\mathit{unlines}_{\mathrm{F}} :: \forall s.\, [L^{\mathrm{T}}\ s\ \mathit{String}] \to L^{\mathrm{T}}\ s\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{F}}\ [\,] \qquad\quad = \mathit{new}\ \texttt{""} \\
&\mathit{unlines}_{\mathrm{F}}\ (x : \mathit{xs}) = \mathit{lift2}\ \mathit{catLine}_{\mathrm{L}}\ (x, \mathit{unlines}_{\mathrm{F}}\ \mathit{xs})
\end{aligned}
$$

As one can see, $\mathit{unlines}_{\mathrm{F}}$ is written in the same applicative style as $\mathit{unlines}$. The construction principle is: if the original function handles data that one would like update bidirectionally (e.g., String in this case), replace the all manipulations (e.g., $\mathit{catLine}$ and `""`) of the data with the corresponding bidirectional versions (e.g., $\mathit{lift2}\ \mathit{catLine}_{\mathrm{L}}$ and $\mathit{new}\ \texttt{""}$).

Lens $\mathit{unlines}_{\mathrm{L}}$ accepts updates that do not change the original formatting of the view (i.e., the same number of lines and an empty last line). For example, we have $\mathit{put}\ \mathit{unlines}_{\mathrm{L}}\ [\texttt{"a"},\texttt{"b"},\texttt{"c"}]\ \texttt{"AA\textbackslash nBB\textbackslash nCC\textbackslash n"} = [\texttt{"AA"},\texttt{"BB"},\texttt{"CC"}]$, but $\mathit{put}\ \mathit{unlines}_{\mathrm{L}}\ [\texttt{"a"},\texttt{"b"},\texttt{"c"}]\ \texttt{"AA\textbackslash nBB\textbackslash n"} = \bot$ and $\mathit{put}\ \mathit{unlines}_{\mathrm{L}}\ [\texttt{"a"},\texttt{"b"},\texttt{"c"}]\ \texttt{"AA\textbackslash nBB\textbackslash nCC\textbackslash nD"} = \bot$.

Example 4 ($\mathit{unlines}$ defined by foldr)**.** Another common way to implement $\mathit{unlines}$ is to use foldr, as below.

$$
\mathit{unlines} = \mathit{foldr}\ \mathit{catLine}\ \texttt{""}
$$

The same coding principle for constructing bidirectional versions applies.

$$
\begin{aligned}
&\mathit{unlines}_{\mathrm{L}} :: L\ [\mathit{String}]\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{L}} = \mathit{unlift}_{\mathrm{list}}\ \mathit{unlines}_{\mathrm{F}} \\
&\mathit{unlines}_{\mathrm{F}} :: \forall s.\, [L^{\mathrm{T}}\ s\ \mathit{String}] \to L^{\mathrm{T}}\ s\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{F}} = \mathit{foldr}\ (\mathit{lift2}\ \mathit{catLine}_{\mathrm{L}})\ (\mathit{new}\ \texttt{""})
\end{aligned}
$$

The new $\mathit{unlines}_{\mathrm{F}}$ is again in the same applicative style as the new $\mathit{unlines}$, where the unidirectional function foldr is applied to normal functions and lens functions alike. □

For readers familiar with the literature of bidirectional transformation, this restriction to in-place updates is very similar to that in semantic bidirectionalization [21, 33, 41]. We will discuss the connection in Section 7.1.

4.2 Datatype-Generic Unlifting Functions

The treatment of lists is an instance of the general case of container-like datatypes. We can view any container with n elements as an n-tuple, only to have list length replaced by the more general container shape. In this section, we define a generic version of our technique that works for many datatypes.

Specifically, we use the datatype-generic function $\mathit{traverse}$, which can be found in `Data.Traversable`, to give data-type generic lifting and unlifting functions.

$$
\begin{aligned}
&\mathit{traverse} :: (\mathit{Traversable}\ t, \mathit{Applicative}\ f) \\
&\qquad\qquad \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)
\end{aligned}
$$

We use $\mathit{traverse}$ to define two functions that are able to extract data from the structure holding them ($\mathit{contents}$), and redecorate an "empty" structures with given data (fill). [2]

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{Const}\ a\ b = \mathit{Const}\ \{\mathit{getConst} :: a\} \\
&\mathit{contents} :: \mathit{Traversable}\ t \Rightarrow t\ a \to [a] \\
&\mathit{contents}\ t = \mathit{getConst}\ (\mathit{traverse}\ (\lambda x \to \mathit{Const}\ [x])\ t)
\end{aligned}
$$

[2] In GHC, the function $\mathit{contents}$ is called toList, which is defined in `Data.Foldable` (Every $\mathit{Traversable}$ instance is also an instance of $\mathit{Foldable}$). We use the name $\mathit{contents}$ to emphasize the function's role of extracting contents from structures [3].

$$
\begin{array}{l}
\mathit{fill} :: \mathit{Traversable}\ t \Rightarrow t\ b \rightarrow [a] \rightarrow t\ a \\
\mathit{fill}\ t\ \ell = \mathit{evalState}\ (\mathit{traverse}\ \mathit{next}\ t)\ \ell \\
\quad \textbf{where} \\
\qquad \mathit{next}\ \_ = \textbf{do}\ (a : x) \leftarrow \mathit{Control.Monad.State.get} \\
\qquad\qquad\qquad\qquad \mathit{Control.Monad.State.put}\ x \\
\qquad\qquad\qquad\qquad \mathit{return}\ a
\end{array}
$$

Here, $\mathit{Const}\ a\ b$ is an instance of the Haskell $\mathit{Functor}$ that ignores its argument b. It becomes an instance of $\mathit{Applicative}$ if a is an instance of Monoid. We qualified the state monad operations get and put to distinguish them from the get and put as bidirectional transformations.

For many datatypes such as lists and trees, instances of $\mathit{Traversable}$ are straightforward to define to the extend of being systematically derivable [23]. The instances of $\mathit{Traversable}$ must satisfy certain laws [3]; and for such lawful instances, we have

$$\mathit{fill}\ (\mathit{fmap}\ f\ t)\ (\mathit{contents}\ t) = t \qquad \textbf{(FillContents)}$$

$$\mathit{contents}\ (\mathit{fill}\ t\ \mathit{xs}) = \mathit{xs}\ \text{ if }\ \mathit{length}\ \mathit{xs} = \mathit{length}\ (\mathit{contents}\ t) \qquad \textbf{(ContentsFill)}$$

for any f and t, which are needed to established the correctness of our generic algorithm. Note that every $\mathit{Traversable}$ instance is also an instance of $\mathit{Functor}$.

We can now define a generic $\mathit{lsequence}$ function as follows.

$$
\begin{array}{l}
\mathit{lsequence} :: (\mathit{Eq}\ a, \mathit{Eq}\ (t\ ()), \mathit{Traversable}\ t) \Rightarrow \\
\qquad\qquad t\ (L^{\mathrm{T}}\ s\ a) \rightarrow L^{\mathrm{T}}\ s\ (t\ a) \\
\mathit{lsequence}\ t = \\
\quad \mathit{lift}\ (\mathit{fill}_{\mathrm{L}}\ (\mathit{shape}\ t))\ (\mathit{lsequence}_{\mathrm{list}}\ (\mathit{contents}\ t)) \\
\quad \textbf{where} \\
\qquad \mathit{fill}_{\mathrm{L}}\ s = L\ (\lambda \mathit{xs} \rightarrow \mathit{fill}\ s\ \mathit{xs})\ (\lambda\_\ t \rightarrow \mathit{contents}'\ s\ t) \\
\qquad \mathit{contents}'\ s\ t = \textbf{if}\ \mathit{shape}\ t == s \\
\qquad\qquad\qquad\qquad \textbf{then}\ \mathit{contents}\ t \\
\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{error}\ \texttt{"Shape Mismatch"}
\end{array}
$$

Here, shape computes the shape of a structure by replacing elements with units, i.e., $\mathit{shape}\ t = \mathit{fmap}\ (\lambda\_ \rightarrow ())\ t$. Also, we can make a Poset instance as follows.[3]

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Poset}\ a, \mathit{Eq}\ (t\ ()), \mathit{Traversable}\ t) \Rightarrow \\
\qquad\qquad \mathit{Poset}\ (t\ a)\ \textbf{where} \\
\quad t_1 \curlyvee t_2 = \textbf{if}\ \mathit{shape}\ t_1 == \mathit{shape}\ t_2 \\
\qquad\qquad \textbf{then}\ \mathit{fill}\ t_1\ (\mathit{contents}\ t_1 \curlyvee \mathit{contents}\ t_2) \\
\qquad\qquad \textbf{else}\ \bot\ \ \text{-- Unreachable, in our framework}
\end{array}
$$

Following the example of lists, we have a generic unlifting function with length replaced by shape.

$$
\begin{array}{l}
\mathit{unliftT} :: (\mathit{Eq}\ (t\ ()), \mathit{Eq}\ a, \mathit{Traversable}\ t) \Rightarrow \\
\qquad\qquad (\forall s.\ t\ (L^{\mathrm{T}}\ s\ a) \rightarrow L^{\mathrm{T}}\ s\ b) \rightarrow L\ (t\ a)\ b \\
\mathit{unliftT}\ f = L\ (\lambda s \rightarrow \mathit{get}\ (\mathit{mkLens}\ s)\ s) \\
\qquad\qquad\qquad (\lambda s \rightarrow \mathit{put}\ (\mathit{mkLens}\ s)\ s) \\
\quad \textbf{where} \\
\qquad \mathit{mkLens}\ s = f\ (\mathit{projTs}\ (\mathit{shape}\ s))\ \hat{\circ}\ \mathit{tagT}_{\mathrm{L}} \\
\qquad \mathit{tagT}_{\mathrm{L}} = L\ (\mathit{fmap}\ O)\ (\mathit{const}\ \$\ \mathit{fmap}\ \mathit{unTag}) \\
\qquad \mathit{projTs}\ \mathit{sh} = \\
\qquad\quad \textbf{let}\ n = \mathit{length}\ (\mathit{contents}\ \mathit{sh}) \\
\qquad\quad \textbf{in}\ \mathit{fill}\ \mathit{sh}\ [\mathit{projT}_{\mathrm{L}}\ i\ \mathit{sh} \mid i \leftarrow [0\,..\,n-1]] \\
\qquad \mathit{projT}_{\mathrm{L}}\ i\ \mathit{sh} = \\
\qquad\quad L\ (\mathit{unTag} \circ (!!i) \circ \mathit{contents}) \\
\qquad\qquad (\lambda s\ v \rightarrow \mathit{fill}\ \mathit{sh}\ (\mathit{update}\ i\ (U\ v)\ (\mathit{contents}\ s)))
\end{array}
$$

[3] This definition actually overlaps with that for pairs. So we either need to have "wrapper" type constructors, or enable `OverlappingInstances`.

Here, $\mathit{projT}_{\mathrm{L}}\ i\ t$ is a bidirectional transformation that extracts the ith element in t with the tag erased. Similarly to $\mathit{unlift}_{\mathrm{list}}$, the shape of the source is an invariant of the derived lens.

5. An Application: Bidirectional Evaluation

In this section, we demonstrate the expressiveness of our framework by defining a bidirectional evaluator in it. As we will see in a larger scale, programming in our framework is very similar to what it is in conventional unidirectional languages, distinguishing us from the others.

An evaluator can be seen as a mapping from an environment to a value of a given expression. A bidirectional evaluator [14] additionally takes the same expression but maps an updated value of the expression back to an updated environment, so that evaluating the expression under the updated environment results in the value.

Consider the following syntax for a higher-order call-by-value language.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp} = \mathit{ENum}\ \mathit{Int} \mid \mathit{EInc}\ \mathit{Exp} \\
\qquad\qquad \mid\ \mathit{EVar}\ \mathit{String} \mid \mathit{EApp}\ \mathit{Exp}\ \mathit{Exp} \\
\qquad\qquad \mid\ \mathit{EFun}\ \mathit{String}\ \mathit{Exp}\ \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ \mathit{Val}\ a = \mathit{VNum}\ a \\
\qquad\qquad \mid\ \mathit{VFun}\ \mathit{String}\ \mathit{Exp}\ (\mathit{Env}\ a)\ \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ \mathit{Env}\ a = \mathit{Env}\ [(\mathit{String}, \mathit{Val}\ a)]\ \textbf{deriving}\ \mathit{Eq}
\end{array}
$$

This definition is standard, except that the type of values is parameterized to accommodate both $\mathit{Val}\ (L^{\mathrm{T}}\ s\ \mathit{Int})$ and $\mathit{Val}\ \mathit{Int}$ for updatable and ordinary integers, and so does the type of environments. It is not difficult to make Val and Env instances of $\mathit{Traversable}$.

We only consider well-typed expressions. Using our framework, writing a bidirectional evaluator is almost as easy as writing the usual unidirectional one.

$$
\begin{array}{lcl}
\mathit{eval} :: \mathit{Env}\ (L^{\mathrm{T}}\ s\ \mathit{Int}) \rightarrow \mathit{Exp} \rightarrow \mathit{Val}\ (L^{\mathrm{T}}\ s\ \mathit{Int}) & & \\
\mathit{eval}\ \mathit{env}\ (\mathit{ENum}\ n) & = & \mathit{VNum}\ (\mathit{new}\ n) \\
\mathit{eval}\ \mathit{env}\ (\mathit{EInc}\ e) & = & \textbf{let}\ \mathit{VNum}\ v = \mathit{eval}\ \mathit{env}\ e \\
 & & \textbf{in}\ \mathit{VNum}\ (\mathit{lift}\ \mathit{inc}_{\mathrm{L}}\ v) \\
\mathit{eval}\ \mathit{env}\ (\mathit{EVar}\ x) & = & \mathit{lkup}\ x\ \mathit{env} \\
\mathit{eval}\ \mathit{env}\ (\mathit{EApp}\ e_1\ e_2) & = & \textbf{let}\ \mathit{VFun}\ x\ e'\ (\mathit{Env}\ \mathit{env}') = \\
 & & \qquad \mathit{eval}\ \mathit{env}\ e_1 \\
 & & \quad\ \ v_2 = \mathit{eval}\ \mathit{env}\ e_2 \\
 & & \textbf{in}\ \mathit{eval}\ (\mathit{Env}\ ((x, v_2) : \mathit{env}'))\ e' \\
\mathit{eval}\ \mathit{env}\ (\mathit{EFun}\ x\ e) & = & \mathit{VFun}\ x\ e\ \mathit{env}
\end{array}
$$

Here, $\mathit{inc}_{\mathrm{L}} :: L\ \mathit{Int}\ \mathit{Int}$ is a bidirectional version of $(+1)$ that can be defined as follows.

$$\mathit{inc}_{\mathrm{L}} = L\ (+1)\ (\lambda\_\ x \rightarrow x - 1)$$

and $\mathit{lkup} :: \mathit{String} \rightarrow \mathit{Env}\ a \rightarrow a$ is a lookup function.

A lens $\mathit{eval}_{\mathrm{L}} :: \mathit{Exp} \rightarrow L\ (\mathit{Env}\ \mathit{Int})\ (\mathit{Val}\ \mathit{Int})$ naturally arises from eval.

$$
\begin{array}{l}
\mathit{eval}_{\mathrm{L}} :: \mathit{Exp} \rightarrow L\ (\mathit{Env}\ \mathit{Int})\ (\mathit{Val}\ \mathit{Int}) \\
\mathit{eval}_{\mathrm{L}}\ e = \mathit{unliftT}\ (\lambda \mathit{env} \rightarrow \mathit{liftT}\ \mathit{id}_{\mathrm{L}}\ \$\ \mathit{eval}\ \mathit{env}\ e)
\end{array}
$$

As an example, let's consider the following expression which essentially computes $x + 65536$ by using a higher-order function twice in the object language.

$$
\begin{array}{l}
\mathit{expr} = \mathit{twice}\ \texttt{@@}\ \mathit{twice}\ \texttt{@@}\ \mathit{twice}\ \texttt{@@}\ \mathit{twice}\ \texttt{@@}\ \mathit{inc}\ \texttt{@@}\ x \\
\quad \textbf{where} \\
\qquad \mathit{twice} = \mathit{EFun}\ \texttt{"f"}\ \$\ \mathit{EFun}\ \texttt{"x"}\ \$ \\
\qquad\qquad\qquad \mathit{EVar}\ \texttt{"f"}\ \texttt{@@}\ (\mathit{EVar}\ \texttt{"f"}\ \texttt{@@}\ \mathit{EVar}\ \texttt{"x"}) \\
\qquad x\ \ \ = \mathit{EVar}\ \texttt{"x"} \\
\qquad \mathit{inc} = \mathit{EFun}\ \texttt{"x"}\ \$\ \mathit{EInc}\ (\mathit{EVar}\ \texttt{"x"})
\end{array}
$$

$$\textbf{infixl } 9 \texttt{@@} \quad \texttt{--}\ \textit{@@ is left associative}$$
$$(\texttt{@@}) = EApp$$

For easy reading, we translate the above expression to Haskell syntax.

$$expr = ((((twice\ twice)\ twice)\ twice)\ inc)\ x$$
$$\quad \textbf{where}\ twice\ f\ x = f\ (f\ x); inc\ x = x + 1$$

Now giving an environment that binds the free variables x and y, we can run the bidirectional evaluator as follows, with $env_0 = Env\ [(\texttt{"x"}, VNum\ 3)]$.

```
Main> get (eval_L expr) env_0
VNum 65539
Main> put (eval_L expr) env_0 (VNum 65536)
Env [("x", VNum 0)]
```

As a remark, this seemingly innocent implementation of $eval_{\rm L}$ is actually highly non-trivial. It essentially defines compositional (or modular) bidirectionalization [20, 21, 33, 41] of programs that are *monomorphic* in type and use *higher-order* functions in definition—something that has not been achieved in bidirectional-transformation research so far.

6. Extensions

In this section, we extend our framework in two dimensions: allowing shape changes via lifting lens combinators, and allowing $(L^{\rm T}\ s\ A)$-values to be inspected during forward transformations following our previous work [21, 22].

6.1 Lifting Lens-Combinators

An advantage of the original lens combinators [9] (that operate directly on the non-functional representation of lenses) over what we have presented so far is the ability to accept shape changes to views. We argue that our framework is general enough to easily incorporate such lens combinators.

Since we already know how to lift/unlift lenses, it only takes some plumbing to be able to handle lens combinators, which are simply functions over lenses. For example, for combinators of type $L\ A\ B \to L\ C\ D$ we have

$$liftC :: Eq\ a \Rightarrow (L\ a\ b \to L\ c\ d) \to$$
$$\qquad (\forall s.\, L^{\rm T}\ s\ a \to L^{\rm T}\ s\ b) \to (\forall t.\, L^{\rm T}\ t\ c \to L^{\rm T}\ t\ d)$$
$$liftC\ c\ f = lift\ (c\ (unlift\ f))$$

To draw an analogy to parametric higher-order abstract syntax [5], the polymorphic arguments of the lifted combinators represent closed expressions; for example, a program like $\lambda x \to \ldots c\ (\ldots x \ldots) \ldots$ does not type-check when c is a lifted combinator.

As an example, let us consider the following lens combinator $mapDefault_{\rm C}$.

$$mapDefault_{\rm C} :: a \to L\ a\ b \to L\ [a]\ [b]$$
$$mapDefault_{\rm C}\ d\ \ell = L\ (map\ (get\ \ell))\ (\lambda s\ v \to go\ s\ v)$$
$$\quad \textbf{where}\ go\ ss\ \ \ [\,]\ \ \ \ \ \ \ = [\,]$$
$$\qquad go\ [\,]\ \ \ \ (v : vs) = put\ \ell\ d\ v : go\ [\,]\ vs$$
$$\qquad go\ (s : ss)\ (v : vs) = put\ \ell\ s\ v : go\ ss\ vs$$

When given a lens on elements, $mapDefault_{\rm C}\ d$ turns it into a lens on lists. The default value d is used when new elements are inserted to the view, making the list lengths different. We can incorporate this behavior into our framework. For example, we can use $mapDefault_{\rm C}$ as the following, which in the forward direction is essentially $map\ (uncurry\ (+))$.

$$mapAdd_{\rm L} :: L\ [(Int, Int)]\ [Int]$$
$$mapAdd_{\rm L} = unlift\ mapAdd_{\rm F}$$
$$mapAdd_{\rm F}\ xs = map_{\rm F}\ (0,0)\ (lift\ addL)\ xs$$
$$map_{\rm F}\ d = liftC\ (mapDefault_{\rm C}\ d)$$
$$addL = L\ (\lambda(x,y) \to x + y)\ (\lambda(x,\_)\ v \to (x, v - x))$$

This lens $mapAdd_{\rm L}$ constructed in our framework handles shape changes without any trouble.

```
Main> put mapAdd_L [(1,1),(2,2)] [3,5]
[(1,2),(2,3)]
Main> put mapAdd_L [(1,1),(2,2)] [3]
[(1,2)]
Main> put mapAdd_L [(1,1),(2,2)] [3,5,7]
[(1,2),(2,3),(0,7)]
```

The trick is that the expression $map_{\rm F}\ (0,0)\ (lift\ addL)$ has type $\forall s.\, L^{\rm T}\ s\ [(Int, Int)] \to L^{\rm T}\ s\ [Int]$, where the list occurs inside $L^{\rm T}\ s$, contrasting to $map\ (lift\ addL)$'s type $\forall s.\, [L^{\rm T}\ s\ (Int, Int)] \to [L^{\rm T}\ s\ Int]$. Intuitively, the type constructor $L^{\rm T}\ s$ can be seen as an updatability annotation; $L^{\rm T}\ s\ [(Int, Int)]$ means that the list itself is updatable, whereas $[L^{\rm T}\ s\ (Int, Int)]$ means that only the elements are updatable. Here is the trade-off: the former has better updatability at the cost of a special lifted lens combinator; the latter has less updatability but simply uses the usual map directly. Our framework enables programmers to choose either style, or anywhere in between freely.

This position-based approach used in $mapDefault_{\rm C}$ is not the only way to resolve shape descrepencies. We can also match elements according to keys [2, 11]. As an example, let us consider a variant of the map combinator.

$$mapByKey_{\rm C} :: Eq\ k \Rightarrow a \to L\ a\ b \to L\ [(k,a)]\ [(k,b)]$$
$$mapByKey_{\rm C}\ d\ \ell = L\ (map\ (\lambda(k,s) \to (k, get\ \ell\ s)))$$
$$\qquad\qquad (\lambda s\ v \to go\ s\ v)$$
$$\quad \textbf{where}\ go\ ss\ [\,] \qquad = [\,]$$
$$\qquad go\ ss\ ((k,v) : vs) =$$
$$\qquad\quad \textbf{case}\ lookup\ k\ ss\ \textbf{of}$$
$$\qquad\qquad Nothing \to (k, put\ \ell\ d\ v) : go\ ss\ vs$$
$$\qquad\qquad Just\ s\ \ \to (k, put\ \ell\ s\ v) : go\ (del\ k\ ss)\ vs$$
$$\qquad del\ k\ [\,] \qquad\qquad = [\,]$$
$$\qquad del\ k\ ((k', s) : ss)\ \mid k == k' \quad = ss$$
$$\qquad\qquad\qquad \mid otherwise = (k', s) : del\ k\ ss$$

Lenses constructed with $mapByKey_{\rm C}$ match with keys instead of positions.

$$mapAddByKey_{\rm L} :: Eq\ k \Rightarrow L\ [(k, (Int, Int))]\ [(k, Int)]$$
$$mapAddByKey_{\rm L} = unlift\ mapAddByKey_{\rm F}$$
$$mapAddByKey_{\rm F}\ xs = mapByKey_{\rm F}\ (0,0)\ (lift\ addL)\ xs$$
$$mapByKey_{\rm F}\ d = liftC\ (mapByKey_{\rm C}\ d)$$

Let s be $[(\texttt{"A"}, (1,1)), (\texttt{"B"}, (2,2))]$. Then, the obtained lens works as follows.

```
Main> put mapAddByKey_L s [("B",5),("A",3)]
[("B",(2,3)),("A",(1,2))]
Main> put mapAddByKey_L s [("A",3)]
[("A",(1,2))]
Main> put mapAddByKey_L s [("B",5),("C",7),("A",3)]
[("B",(2,3)),("C",(0,7)),("A",(1,2))]
```

6.2 Observations of Lifted Values

So far we have programmed bidirectional transformations ranging from polymorphic to monomorphic functions. For example, $unlines$ is monomorphic because its base case returns a String constant, which is nicely handled in our framework by the function new. At the same time, it is also obvious that the creation of constant values is not the only cause of a transformation being monomorphic [21, 22]. For example, let us consider the following toy program.[4]

$$bad\ (x,y) = \textbf{if}\ x == new\ 0\ \textbf{then}\ (x,y)\ \textbf{else}\ (x, new\ 1)$$

In this program, the behavior of the transformation depends on the "observation" made to a value that may potentially be updated in the view. Then the naively obtained lens $bad_{\mathrm{L}} = unlift2\ (lift2\ id_{\mathrm{L}} \circ bad)$ would violate well-behavedness, as $put\ bad_{\mathrm{L}}\ (0,2)\ (1,2) = (1,2)$ but $get\ bad_{\mathrm{L}}\ (1,2) = (1,1)$.

Our previous work [21, 22] tackles this problem by using a monad to record observations, and to enforce that the recorded observation results remain unchanged while executing put. The same technique can be used in our framework, and actually in a much simpler way due to our new compositional formalization.

$$\textbf{newtype}\ R\ s\ a = R\ (Poset\ s \Rightarrow s \rightarrow (a, s \rightarrow Bool))$$

We can see that $R\ A\ B$ represents gets with restricted source updates: taking a source $s :: A$, it returns a view of type B together with a constraint of type $A \rightarrow Bool$ which must remain satisfied amid updates of s. Formally, giving $R\ m :: R\ A\ B$, for any s, if $(\_, p) = m\ s$ then we have: (1) $p\ s = True$; (2) $p\ s' = True$ implies $m\ s = m\ s'$ for any s'. It is not difficult to make $R\ s$ an instance of $Monad$—it is a composition of $Reader$ and $Writer$ monads. We only show the definition of $(\gg\!=)$.

$$\begin{aligned} R\ m \gg\!= f = R\ \$\ \lambda s \rightarrow\ & \textbf{let}\ (x, c_1) = m\ s \\ & \qquad (y, c_2) = \textbf{let}\ R\ k = f\ x\ \textbf{in}\ k\ s \\ & \textbf{in}\ (y, \lambda s \rightarrow c_1\ s \wedge c_2\ s) \end{aligned}$$

Then, we define a function that produces R values, and a version of unlifting that enforces the observations gathered.

$$\begin{aligned}
& observe :: Eq\ w \Rightarrow L^{\mathrm{T}}\ s\ w \rightarrow R\ s\ w \\
& observe\ \ell = R\ (\lambda s \rightarrow \textbf{let}\ w = get\ \ell\ s \\
& \qquad\qquad\qquad\qquad \textbf{in}\ (w, \lambda s' \rightarrow get\ \ell\ s' == w)) \\
& unliftM2 :: (Eq\ a, Eq\ b) \Rightarrow \\
& \qquad (\forall s. (L^{\mathrm{T}}\ s\ a, L^{\mathrm{T}}\ s\ b) \rightarrow R\ s\ (L^{\mathrm{T}}\ s\ c)) \\
& \qquad \rightarrow L\ (a,b)\ c \\
& unliftM2\ f = L\ (\lambda s \rightarrow get\ (mkLens\ f\ s)\ s) \\
& \qquad\qquad\qquad (\lambda s \rightarrow put\ (mkLens\ f\ s)\ s) \\
& \quad \textbf{where} \\
& \quad\quad mkLens\ f\ s = \\
& \quad\quad\quad \textbf{let}\ (\ell, p) = \textbf{let}\ R\ m = f\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}}) \\
& \quad\quad\quad\qquad\qquad\quad \textbf{in}\ m\ (get\ tag2_{\mathrm{L}}\ s) \\
& \quad\quad\quad\quad \ell' = \ell\ \hat{\circ}\ tag2_{\mathrm{L}} \\
& \quad\quad\quad\quad put'\ s\ v = \textbf{let}\ s' = put\ \ell'\ s\ v \\
& \quad\quad\quad\quad\qquad\quad \textbf{in if}\ p\ (get\ tag2_{\mathrm{L}}\ s')\ \textbf{then}\ s'\ \textbf{else}\ \bot \\
& \quad\quad\quad \textbf{in}\ L\ (get\ \ell')\ put'
\end{aligned}$$

Although we define the get and put components of the resulting lens separately in $unliftM2$, well-behavedness is guaranteed as long as R and L^{T} are used abstractly in f. Note that, similarly to $unliftM2$, we can define $unliftM$ and $unliftMT$, as monadic versions of $unlift$ and $unliftT$.

We can now sprinkle $observe$ at where observations happens, and use $unliftM$ to guard against changes to them.

$$\begin{aligned}
& good\ (x,y) = fmap\ (lift2\ id_{\mathrm{L}})\ \$\ \textbf{do} \\
& \quad b \leftarrow liftO2\ (==)\ x\ (new\ 0) \\
& \quad return\ (\textbf{if}\ b\ \textbf{then}\ (x,y)\ \textbf{else}\ (x, new\ 1))
\end{aligned}$$

Here, $liftO2$ is defined as follows.

$$\begin{aligned}
& liftO2 :: Eq\ w \Rightarrow \\
& \qquad\qquad (a \rightarrow b \rightarrow w) \rightarrow L^{\mathrm{T}}\ s\ a \rightarrow L^{\mathrm{T}}\ s\ b \rightarrow R\ s\ w \\
& liftO2\ p\ x\ y = liftO\ (uncurry\ p)\ (x \circledast y) \\
& liftO :: Eq\ w \Rightarrow (a \rightarrow w) \rightarrow L^{\mathrm{T}}\ s\ a \rightarrow R\ s\ w \\
& liftO\ p\ x = observe\ (lift\ (L\ p\ unused)\ x) \\
& \quad \textbf{where}\ unused\ s\ v \mid v == p\ s = s
\end{aligned}$$

Then the obtained lens $good_{\mathrm{L}} = unliftM2\ \ good$ successfully rejects illegal updates, as $put\ good_{\mathrm{L}}\ (0,2)\ (1,2) = \bot$.

One might have noticed that the definition of $good$ is in the *Monadic style*—not applicative in the sense of [23]. This is necessary for handling observations, as the effect of $(R\ s)$ must depend on the value in it [18].

Due to space restriction, we refer interested readers to our previous work [21, 22] for practical examples of bidirectional transformations with observations.

7. Related Work and Discussions

In this section, we discuss related techniques to our paper, making connections to a couple of notable bidirectional programming approaches, namely semantic bidirectionalization and the van Laarhoven representation of lenses.

7.1 Semantic Bidirectionalization

An alternative way of building bidirectional transformations other than lenses is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as bidirectionalization [20]. Different flavors of bidirectionalization have been proposed: syntactic [20], semantic [21, 22, 33, 41], and a combination of the two [35, 36]. Syntactic bidirectionalization inspects a forward function definition written in a somehow restricted syntactic representation and synthesizes a definition for the backward version. Semantic bidirectionalization on the other hand treats a polymorphic get as a semantic object, applying the function independently to a collection of unique identifiers, and the free theorems arising from parametricity states that whatever happens to those identifiers happens in the same way to any other inputs—this information is sufficient to construct the backward transformation.

Our framework can be viewed as a more general form of semantic bidirectionalization. For example, giving a function of type $\forall a. [a] \rightarrow [a]$, a bidirectionalization engine in the style of [33] can be straightforwardly implemented in our framework as follows.

$$\begin{aligned}
& bff :: (\forall a. [a] \rightarrow [a]) \rightarrow (Eq\ a \Rightarrow L\ [a]\ [a]) \\
& bff\ f = unlift_{\mathrm{list}}\ (lsequence_{\mathrm{list}} \circ f)
\end{aligned}$$

Replacing $unlift_{\mathrm{list}}$ and $lsequence_{\mathrm{list}}$ with $unliftT$ and $lsequence$, we also obtain the datatype generic version [33].

With the addition of $observe$ and the monadic unlifting functions, we are also able to cover extensions of semantic bidirectionalization [21, 22] in a simpler and more fundamental way. For example, $liftO2$ (and other n-ary observations-lifting functions) has to be a primitive previously [21, 22], but can now be derived from $observe$, $lift$ and $(\circledast)$ in our framework.

Our work's unique ability of combining lenses and semantic bidirectionalization results in more applicability and control than those offered by bidirectionalization alone: user-defined lenses on base types can now be passed to higher-order functions. For example, Q5 of Use Case "STRING" in XML Query Use Case (`http://www.w3.org/TR/xquery-use-cases`) which involves concatenation of strings in the transformation, can be handled by our technique, but not previously with bidirectionalization [21, 22, 33, 41]. We believe that with the proposal in this paper, all queries in XML Query Use Case can now be bidirectionalized. In a sense we are a step forward to the best of both worlds: gaining convenience in programming without losing expressiveness.

[4] This code actually does not type check as $(==)$ on $(L^{\mathrm{T}}\ s\ Int)$-values depends on a source and has to be implemented monadically. But we do not fix this program as it is meant to be a non-solution that will be discarded.

The handling of observation in this paper follows the idea of our previous work [21, 22] to record only the observations that actually happened, not those that may. The latter approach used in [33, 41] has the advantage of not requiring a monad, but at the same time not applicable to monomorphic transformations, as the set of the possible observation results is generally infinite.

7.2 Functional Representation of Bidirectional Transformations

There exists another functional representation of lenses known as the van Laarhoven representation [26, 32]. This representation, adopted by the Haskell library `lens`, encodes bidirectional transformations of type $L\ A\ B$ as functions of the following type.

$$\forall f.\, \mathit{Functor}\ f \Rightarrow (B \to f\ B) \to (A \to f\ A)$$

Intuitively, we can read $A \to f\ A$ as updates on A and a lens in this representation maps updates on B (view) to updates on A (source), resulting in a "put-back based" style of programming [27]. The van Laarhoven representation also has its root in the Yoneda Lemma [17, 24]; unlike ours which applies the Yoneda Lemma to $L\ (-)\ V$, they apply the Yoneda Lemma to a functor $(V, V \to (-))$. Note that the lens type $L\ S\ V$ is isomorphic to the type $S \to (V, V \to S)$.

Compared to our approach, the van Laarhoven representation is rather inconvenient for applicative-style programming. It cannot be used to derive a put when a get is already given, as in bidirectionalization [20–22, 33, 35, 36, 41] and the classical view update problem [1, 6, 8, 13], especially in a higher-order setting. In the van Laarhoven representation, a bidirectional transformation $\ell :: L\ A\ B$, which has $get\ \ell :: A \to B$, is represented as a function from some B structure to some A structure. This difference in direction poses a significant challenge for higher-order programs, because structures of abstractions and applications are not preserved by inverting the direction of $\to$. In contrast, our construction of put from get is straightforward; replacing base type operations with the lifted bidirectional versions is suffice as shown in the $unlines_{\mathrm{L}}$ and $eval_{\mathrm{L}}$ examples (monadification is only needed when supporting observations). Moreover, the van Laarhoven representation does not extend well to data structures: n-ary functions in the representation do not correspond to n-ary lenses. As a result, the van Laarhoven representation itself is not useful to write bidirectional programs such as $unlines_{\mathrm{L}}$ and $eval_{\mathrm{L}}$. Actually as far as we are aware, higher-order programming with the van Laarhoven representation has not been investigated before.

By using the Yoneda embedding, we can also express $L\ A\ B$ as functions of type $\forall v.\, L\ B\ v \to L\ A\ v$. It is worth mentioning that $L\ (-)\ V$ also forms a lax monoidal functor under some conditions [30]; for example, V must be a monoid. However, although their requirement fits well for their purpose of constructing HTML pages with forms, we cannot assume such a suitable monoid structure for a general V. Moreover, similarly to the van Laarhoven representation, this representation cannot be used to derive a put from a get.

8. Conclusion

We have proposed a novel framework of applicative bidirectional programming, which features the strengths of lens [4, 9, 10] and semantics bidirectionalization [21, 22, 33, 41]. In our framework, one can construct bidirectional transformations in an applicative style, almost in the same way as in a usual functional language. The well-behavedness of the resulting bidirectional transformations are guaranteed by construction. As a result, complex bidirectional programs can be now designed and implemented with reasonable efforts.

A future step will be to extend the current ability of handling shape updates. It is important to relax the restriction that only closed expressions can be unlifted to enable more practical programming. A possible solution to this problem would be to abstract certain kind of containers in addition to base-type values, which is likely to lead to a more fine-grained treatment of lens combinators and shape updates.

Acknowledgments

We would like to thank Shin-ya Katsumata, Makoto Hamana, Kazuyuki Asada, and Patrik Jansson for their helpful comments on categorical discussions in this paper. Especially, Shin-ya Katsumata and Makoto Hamana pointed out the relationship from a preliminary version of our method to the Yoneda lemma. We also the anonymous reviewers of this paper for their helpful comments.

This work is partially supported by JSPS KAKENHI Grant Numbers 24700020, 25540001, 15H02681, and 15K15966, and the Grand-Challenging Project on the "Linguistic Foundation for Bidirectional Model Transformation" of the National Institute of Informatics, The work is partly done when the first author was at the University of Tokyo, Japan, and when the second author was at Chalmers University of Technology, partially funded by the Swedish Foundation for Strategic Research through the the Resource Aware Functional Programming (RAW FP) Project.

References

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[2] D. M. J. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In P. Hudak and S. Weirich, editors, *ICFP*, pages 193–204. ACM, 2010.

[3] R. S. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers. Understanding idiomatic traversals backwards and forwards. In C. chieh Shan, editor, *Haskell*, pages 25–36. ACM, 2013.

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In G. C. Necula and P. Wadler, editors, *POPL*, pages 407–419. ACM, 2008.

[5] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In J. Hook and P. Thiemann, editors, *ICFP*, pages 143–156. ACM, 2008.

[6] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[7] T. Ellis. Category and lenses, Sep 2012. Blog post: `http://web.jaguarpaw.co.uk/~tom/blog/posts/2012-09-30-category-and-lenses.html`.

[8] L. Fegaras. Propagating updates through XML views using lineage tracing. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *ICDE*, pages 309–320. IEEE, 2010.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[10] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In J. Hook and P. Thiemann, editors, *ICFP*, pages 383–396. ACM, 2008.

[11] N. Foster, K. Matsuda, and J. Voigtländer. Three complementary approaches to bidirectional programming. In J. Gibbons, editor, *SSGIP*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer, 2010.

[12] Y. Hayashi, D. Liu, K. Emoto, K. Matsuda, Z. Hu, and M. Takeichi. A web service architecture for bidirectional XML updating. In G. Dong, X. Lin, W. Wang, Y. Yang, and J. X. Yu, editors, *APWeb/WAIM*, volume

4505 of *Lecture Notes in Computer Science*, pages 721–732. Springer, 2007.

[13] S. J. Hegner. Foundations of canonical update support for closed database views. In S. Abiteboul and P. C. Kanellakis, editors, *ICDT*, volume 470 of *Lecture Notes in Computer Science*, pages 422–436. Springer, 1990.

[14] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In P. Hudak and S. Weirich, editors, *ICFP*, pages 205–216. ACM, 2010.

[15] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In T. Ball and M. Sagiv, editors, *POPL*, pages 371–384. ACM, 2011.

[16] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In N. Heintze and P. Sestoft, editors, *PEPM*, pages 178–189. ACM, 2004.

[17] M. Jaskelioff and R. O'Connor. A representation theorem for second-order functionals. *CoRR*, abs/1402.1699, 2014.

[18] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electr. Notes Theor. Comput. Sci.*, 229(5):97–117, 2011.

[19] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Matheematics*. Springer, second edition edition, 1998.

[20] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In R. Hinze and N. Ramsey, editors, *ICFP*, pages 47–58. ACM, 2007.

[21] K. Matsuda and M. Wang. Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem. In R. Peña and T. Schrijvers, editors, *PPDP*, pages 297–308. ACM, 2013.

[22] K. Matsuda and M. Wang. "Bidirectionalization for free" for monomorphic transformations. *Science of Computer Programming*, 2014. DOI: `10.1016/j.scico.2014.07.008`.

[23] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[24] B. Milewski. Lenses, Stores, and Yoneda, Oct 2013. blog post: `http://bartoszmilewski.com/2013/10/08/lenses-stores-and-yoneda/`.

[25] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bidirectional updating. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2004.

[26] R. O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR*, abs/1103.2841, 2011. Accepted in WGP'11, but not included in its proceedings.

[27] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In W.-N. Chin and J. Hage, editors, *PEPM*, pages 39–50. ACM, 2014.

[28] R. Paterson. A new notation for arrows. In B. C. Pierce, editor, *ICFP*, pages 229–240. ACM, 2001.

[29] R. Paterson. Constructing applicative functors. In J. Gibbons and P. Nogueira, editors, *MPC*, volume 7342 of *Lecture Notes in Computer Science*, pages 300–323. Springer, 2012.

[30] R. Rajkumar, S. Lindley, N. Foster, and J. Cheney. Lenses for web data. In Preliminary Proceedings of Second International Workshop on Bidirectional Transformations (BX 2013), 2013.

[31] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. Mason, editor, *Information Processing*, pages 513–523. Elsevier Science Publishers B.V. (North-Holland), 1983.

[32] T. van Laarhoven. Cps based functional references, Jul 2009. blog post: `http://www.twanvl.nl/blog/haskell/cps-functional-references`.

[33] J. Voigtländer. Bidirectionalization for free! (pearl). In Z. Shao and B. C. Pierce, editors, *POPL*, pages 165–176. ACM, 2009.

[34] J. Voigtländer. Free theorems involving type constructor classes: functional pearl. In G. Hutton and A. P. Tolmach, editors, *ICFP*, pages 173–184. ACM, 2009.

[35] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In P. Hudak and S. Weirich, editors, *ICFP*, pages 181–192. ACM, 2010.

[36] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *J. Funct. Program.*, 23(5):515–551, 2013.

[37] P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

[38] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement: Blending pattern matching with data abstraction. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *MPC*, volume 6120 of *Lecture Notes in Computer Science*, pages 397–425. Springer, 2010.

[39] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Refactoring pattern matching. *Sci. Comput. Program.*, 78(11):2216–2242, 2013.

[40] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 392–403. ACM, 2011.

[41] M. Wang and S. Najd. Semantic bidirectionalization revisited. In W.-N. Chin and J. Hage, editors, *PEPM*, pages 51–62. ACM, 2014.

[42] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 164–173. ACM, 2007.

[43] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *ICSE*, pages 540–550. IEEE, 2012.