CORE

# Applicative Bidirectional Programming with Lenses 

Kazutaka Matsuda<br>Tohoku University<br>kztk@ecei.tohoku.ac.jp

Meng Wang<br>University of Kent<br>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 constructiongiving 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ICFP'15, August 31 - September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...\$15.00
http://dx.doi.org/10.1145/2784731.2784750
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 $S \rightarrow V$ and put of type $S \rightarrow V \rightarrow S$. 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

$$
\begin{aligned}
& \text { data } L s v=L\{g e t:: s \rightarrow v, p u t:: s \rightarrow v \rightarrow s\} \\
& f s t_{\mathrm{L}}:: L(a, b) a \\
& f s t_{\mathrm{L}}=L\left(\lambda\left(a,_{-}\right) \rightarrow a\right)(\lambda(-, b) a \rightarrow(a, b))
\end{aligned}
$$

A value $\ell$ of type $L s \quad v$ 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 $s \rightarrow v$ and put $\ell$ has type $s \rightarrow v \rightarrow s$. The datatype is used in the definition of $f s t_{\mathrm{L}}$ 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:

$$
\begin{aligned}
& \text { put } \ell s(\text { get } \ell s)=s \\
& \text { get } \ell s^{\prime}=v \text { if put } \ell s v=s^{\prime}
\end{aligned}
$$

(Acceptability)
for all $s, s^{\prime}$ and $v$. Note that in this paper, we write $e=e^{\prime}$ with the assumption that neither $e$ nor $e^{\prime}$ is undefined. Here Consistency (also known as the PutGet law [9]) roughly corresponds to rightinvertibility, 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 wellbehavedness $[1,9,13]$. A bidirectional transformation $L$ get put is called well-behaved if it satisfies well-behavedness. The above example $f_{s t}$ 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 $L\left(A_{1}, \ldots, A_{n}\right) B$ to lens functions of type

$$
\forall s . L^{\mathrm{T}} \text { s } A_{1} \rightarrow \cdots \rightarrow L^{\mathrm{T}} \text { s } A_{n} \rightarrow \ldots \rightarrow L^{\mathrm{T}} \text { s } B
$$

where $L^{\mathrm{T}}$ 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 higherorder functions directly. For example, we can write a bidirectional version of unlines, defined by

$$
\begin{aligned}
& \text { unlines }::[\text { String }] \rightarrow \text { String } \\
& \text { unlines }[]=" \mathrm{l} \\
& \text { unlines }(x: x s)=x+" \backslash \mathrm{n} "+\text { unlines } x s
\end{aligned}
$$

as below.

$$
\begin{aligned}
& \operatorname{unlines}_{\mathrm{F}}::\left[L^{\mathrm{T}} \text { s String }\right] \rightarrow L^{\mathrm{T}} \text { s String } \\
& \text { unlines }_{\mathrm{F}}[]=\text { new "" } \\
& \text { unlines }_{\mathrm{F}}(x: x s)=\text { lift2 catLine }_{\mathrm{L}}\left(x, \text { unlines }_{\mathrm{F}} x s\right)
\end{aligned}
$$

where catLine $_{\mathrm{L}}$ is a lens version of $\lambda x y \rightarrow x+$ " $\backslash \mathrm{n} "+y$. 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 $_{\mathrm{L}}=$ unlift $T$ 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 $A \rightarrow B \rightarrow A$ can be represented as a total function of type $A \rightarrow B \rightarrow$ 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 "ô"

$$
\begin{aligned}
& \text { (ô) :: Lbc L } a b \rightarrow L a c \\
& \left(L_{\text {get }}^{2} \text { put }{ }_{2}\right) \text { ô }\left(L_{\text {get }}^{1}{ }^{p u t_{1}}\right)= \\
& L\left(g e t_{2} \circ \text { get }_{1}\right)\left(\lambda s v \rightarrow p u t_{1} s\left(p_{2} t_{2}\left(\text { get }_{1} s\right) v\right)\right)
\end{aligned}
$$

we can compose $f s t_{\mathrm{L}}$ to itself to obtain a lens that operates on nested pairs, as below.

$$
\begin{aligned}
& f s t T r i_{\mathrm{L}}:: L((a, b), c) a \\
& f s t T r i_{\mathrm{L}}=\text { fst }_{\mathrm{L}} \hat{o} \text { fts }_{\mathrm{L}}
\end{aligned}
$$

Well-behavedness is preserved by such compositions: $f s t T r i_{\mathrm{L}}$ is well-behaved by construction assuming well-behaved $f s t_{\mathrm{L}}$.

The composition operator "ô" has the identity lens $i d_{\mathrm{L}}$ as its unit.

$$
\begin{aligned}
& i d_{\mathrm{L}}:: L a \operatorname{a} a \\
& i d_{\mathrm{L}}=L i d\left(\lambda_{-} v \rightarrow v\right)
\end{aligned}
$$

### 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

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

by lens composition

$$
\begin{aligned}
& \text { lift }:: L a b \rightarrow(\forall s . L s a \rightarrow L s b) \\
& \text { lift } \ell=\lambda x \rightarrow \ell \hat{o} x
\end{aligned}
$$

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.

$$
\begin{aligned}
& \text { unlift :: }(\forall s . L s a \rightarrow L s b) \rightarrow L a b \\
& \text { unlift } f=f \text { id } d_{\mathrm{L}}
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
& \text { fstTri }_{\mathrm{L}}=\text { unlift }\left(\lambda x \rightarrow \text { twice }\left(\text { lift } f s t_{\mathrm{L}}\right) x\right) \\
& \text { where twice } f x=f(f x)
\end{aligned}
$$

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 $\ell$. Functions lift and unlift have the following desirable properties.
Proposition 2. lift $\ell$ preserves well-behavedness if $\ell$ is wellbehaved.
Proposition 3. unlift $f$ is well-behaved if $f$ preserves wellbehavedness.

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

$$
\begin{array}{r}
f:: E q a \Rightarrow L s(\text { Maybe a }) \rightarrow L s(\text { Maybe a) } \\
f(L g p)=L g(\lambda s v \rightarrow \text { if } v=g s \text { then } s \\
\quad \text { else } p(p s \text { Nothing }) v)
\end{array}
$$

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

$$
\begin{aligned}
& \forall \ell .(\forall a b . L a b \rightarrow(\forall s \cdot \ell s a \rightarrow \ell s b)) \\
& \quad \rightarrow(\forall a b \cdot(\forall s . \ell s a \rightarrow \ell s b) \rightarrow L a b) \rightarrow \tau^{\prime}
\end{aligned}
$$

where $\tau^{\prime}=\tau[\ell / L]$ and $f=h$ lift unlift.
Essentially, the polymorphic $\ell$ 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 wellbehavedness 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 $f:: \forall s$. Ls $A \rightarrow L s B$ 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 $L$.

$$
\begin{aligned}
& \text { module AbstractLens ( } L_{\text {abs }}, \text { lift }_{\text {abs }} \text {, unlift }{ }_{\text {abs }} \text { ) where } \\
& \text { newtype } L_{\text {abs }} a b=L_{\text {abs }}\left\{u n L_{\text {abs }}:: L a b\right\} \\
& \text { lift }_{\text {abs }}:: L a b \rightarrow\left(\forall s . L_{\text {abs }} s a \rightarrow L_{\text {abs }} s b\right) \\
& \text { unlift }_{\text {abs }}::\left(\forall s . L_{\text {abs }} s a \rightarrow L_{\text {abs }} s b\right) \rightarrow L a b
\end{aligned}
$$

Outside the module AbstractLens, we can use lift $_{\text {abs }}$, unlift ${ }_{\text {abs }}$ and type $L_{\text {abs }}$ itself, but not the constructor of $L_{\text {abs }}$. 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 $f \ell=f i d_{\mathrm{L}} \hat{o} \ell$ 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 $B \mid f x$ ô $y=f(x \hat{o} y)\}$.

The condition $f x \hat{\circ} y=f(x \hat{o} y)$ is required to make $f$ a natural transformation between functors $L(-) A$ and $L(-) B$; here, the contravariant functor $L(-) A$ maps a lens $\ell$ of type $L Y X$ to a function $(\lambda y \rightarrow y$ ô $\ell$ ) of type $L X A \rightarrow L Y A$. Note that $f x$ ô $y=f(x$ ô $y)$ is equivalent to $f x=f i d_{\mathrm{L}} \hat{\circ} x$. 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 $L A\left(\begin{array}{ll}X & B\end{array}\right)$, which can be easily checked by comparing cardinalities. Our discussion does not conflict with this fact. What we state is that, for any $s,(L s A, L s B) \rightarrow L s C$ is isomorphic to $L$ s $A \rightarrow(L$ s $B \rightarrow L s C)$ via standard curry and uncurry; note that $s$ is quantified globally.

Also note that $L s(-)$ is a functor that maps a lens $\ell$ to a function lift $\ell$. It is not difficult to check that lift $x \circ$ lift $y=$ lift $(x \circ y)$ and lift $\left(i d_{\mathrm{L}}:: L A A\right)=(i d:: L s A \rightarrow L s A)$.

## 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 $f s t T r i_{\mathrm{L}}$ from Section 2 reproduced below.

$$
\begin{aligned}
& f s t T r i_{L_{\mathrm{L}}}:: L((a, b), c) a \\
& \text { fstTri } \left._{\mathrm{L}}=\text { unlift (lift fst } t_{\mathrm{L}} \circ \text { lift }^{\text {fst }}{ }_{\mathrm{L}}\right)
\end{aligned}
$$

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 $f s t T r i_{\mathrm{L}}$, which consists of the composition of
 $(\forall s . L s x \rightarrow L s y)$ ), 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.

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

Following the programming pattern developed so far, we would like to construct this lens with the familiar unidirectional function swap $::(a, b) \rightarrow(b, a)$. But since lift only produces unary functions of type $\forall s . L s A \rightarrow L s B$, 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 $s w a p_{\mathrm{L}}$ and many other lenses, including unlines $_{\mathrm{L}}$ in Section 1, a conversion of values of type $\forall s$. (Ls $A_{1}$, $\left.\ldots, L s A_{n}\right)$ to values of type $\forall s . L s\left(A_{1}, \ldots, A_{n}\right)$ 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 $\forall s$. $(L s A, L s B) \rightarrow L s(A, B)$, we use the duplication lens $d u p_{\mathrm{L}}$ (also known as copy elsewhere [9]) defined as below. For simplicity, we assume that $(==)$ represents observational equivalence.

```
\(d u p_{\mathrm{L}}:: E q s \Rightarrow L s(s, s)\)
\(d u p_{\mathrm{L}}=L(\lambda s \rightarrow(s, s))\left(\lambda_{-}(s, t) \rightarrow r s t\right)\)
    where \(r\) st \(\mid s==t=s\)-- This will cause a problem.
```

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

$$
\begin{aligned}
& (\circledast):: E q s \Rightarrow L s a \rightarrow L s b \rightarrow L s(a, b) \\
& x \circledast y=(x \hat{\otimes} y) \hat{\circ} \operatorname{dup}_{\mathrm{L}}
\end{aligned}
$$

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

$$
\begin{aligned}
& (\hat{\otimes}):: L a a^{\prime} \rightarrow L b b^{\prime} \rightarrow L(a, b)\left(a^{\prime}, b^{\prime}\right) \\
& \left(L_{\text {get }}^{1} \text { put } t_{1}\right) \hat{\otimes}\left(\text { L get }_{2} \text { put }_{2}\right)= \\
& L\left(\lambda(a, b) \rightarrow\left(\text { get }_{1} a, \text { get }_{2} b\right)\right) \\
& \quad\left(\lambda(a, b)\left(a^{\prime}, b^{\prime}\right) \rightarrow\left(\text { put }_{1} a a^{\prime}, \text { put }_{2} b b^{\prime}\right)\right)
\end{aligned}
$$

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

```
lift2 :: \(L(a, b) c \rightarrow(\forall s .(L s a, L s b) \rightarrow L s c)\)
lift2 \(\ell(x, y)=\) lift \(\ell(x \circledast y)\)
```

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

```
unlift2 :: (\foralls. (Ls a,Ls b) }->Lsc)->L(a,b)
unlift2 f}=f(fs\mp@subsup{t}{\textrm{L}}{},sn\mp@subsup{d}{\textrm{L}}{}
```

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

```
    (unlift2 ○ lift2) \(\ell\)
\(=\{\) definition unfolding \(\& \beta\)-reduction \(\}\)
    \(\ell o\left(f s t_{\mathrm{L}} \circledast s n d_{\mathrm{L}}\right)\)
\(=\{\) unfolding \((\circledast)\}\)
    \(\ell \hat{o}\left(f s t_{\mathrm{L}} \hat{\otimes} s n d_{\mathrm{L}}\right) \hat{o} d u p_{\mathrm{L}}\)
\(=\{\) definition unfolding \(\}\)
    \(\ell o ̂ b l o c k_{\mathrm{L}}\) where
        block \(_{\mathrm{L}}=L i d(\lambda s v \rightarrow\) if \(s==v\) then \(v\) else \(\perp)\)
```

Lens 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 unlift2 $\circ$ lift2 useless has its root in $d u p_{\mathrm{L}}$. If we look at the lens $d u p_{\mathrm{L}}$ 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 $d u p_{\mathrm{L}}$ is applied, there is more room for maneuver. Let us consider the lifting function lift2 again, and how put $d u p_{\mathrm{L}}$, which rejects the update above, works in the execution of put (unlift2 (lift2 id $\mathrm{L}_{\mathrm{L}}$ ).

```
    put \(\left(\right.\) unlift2 \(\left(\right.\) lift2 \(_{\text {id }}^{\mathrm{L}}\) )) \((1,2)(\underline{3}, \underline{4})\)
\(=\{\) simplification \(\}\)
    put \(\left(\left(f s t_{\mathrm{L}} \hat{\otimes} s n d_{\mathrm{L}}\right) \hat{o} d u p_{\mathrm{L}}\right)(1,2)(\underline{3}, \underline{4})\)
\(=\{\) definition unfolding \(\& \beta\)-reduction \(\}\)
```



```
\(=\{\beta\)-reduction \(\}\)
    put \(\operatorname{dup}_{\mathrm{L}}(1,2)((\underline{3}, 2),(1, \underline{4}))\)
```

The last call to put $d u p_{\mathrm{L}}$ 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: lift2 $i d B$ 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 $d u p_{\mathrm{L}}$ 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].

$$
\text { data Tag } a=U\{u n T a g:: a\} \mid O\{u n T a g:: a\}
$$

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 $d u p_{\mathrm{L}}$ below.

$$
\begin{aligned}
& d u p_{\mathrm{L}}:: \text { Poset } s \Rightarrow L s(s, s) \\
& d u p_{\mathrm{L}}=L(\lambda s \rightarrow(s, s))\left(\lambda_{-}(s, t) \rightarrow s \curlyvee t\right)
\end{aligned}
$$

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

```
class Poset s where ( }\curlyvee)::s->s->
```

We require that $(\gamma)$ must be associative, commutative and idempotent; but unlike a semilattice, $(\curlyvee)$ can be partial. Tagged elements and their (nested) pairs are ordered as follows.
instance $E q a \Rightarrow \operatorname{Poset}(T a g a)$ where

$$
\begin{aligned}
& (O s) \curlyvee(U t)=U t \\
& (U s) \curlyvee(O t)=U s \\
& (O s) \curlyvee(O t) \mid s==t=O s \\
& (U s) \curlyvee(U t) \mid s==t=U s
\end{aligned}
$$

instance (Poset $a$, Poset $b) \Rightarrow \operatorname{Poset}(a, b)$ where
$(a, b) \curlyvee\left(a^{\prime}, b^{\prime}\right)=\left(a \curlyvee a^{\prime}, b \curlyvee b^{\prime}\right)$
We also introduce the following type synonym for brevity. ${ }^{1}$

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

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

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

```
(*) :: LL
lift :: L a b -> (\foralls. L L
lift2 :: L (a,b) c->(\foralls. (L'T}sa,\mp@subsup{L}{}{\textrm{T}}sb)->\mp@subsup{L}{}{\textrm{T}}sc
```

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

```
unlift :: Eq \(a \Rightarrow\left(\forall s . L^{\mathrm{T}} s a \rightarrow L^{\mathrm{T}} s b\right) \rightarrow L a b\)
unlift \(f=f i d_{\mathrm{L}}^{\prime} \hat{o} \operatorname{tag}_{\mathrm{L}}\)
\(i d_{\mathrm{L}}^{\prime}:: L^{\mathrm{T}}(\) Tag \(a) a\)
\(i d_{\mathrm{L}}^{\prime}=L\) unTag (const \(U\) )
\(t a g_{\mathrm{L}}:: L a(T a g a)\)
\(t a g_{\mathrm{L}}=L O\) (const unTag)
unlift2 :: \(\quad(E q a, E q b) \Rightarrow\)
    \(\left(\forall s .\left(L^{\mathrm{T}} s a, L^{\mathrm{T}} s b\right) \rightarrow L^{\mathrm{T}} s c\right) \rightarrow L(a, b) c\)
unlift2 \(f=f\left(f s t_{\mathrm{L}}^{\prime}, s n d_{\mathrm{L}}^{\prime}\right) \hat{\circ} \operatorname{tag}^{\mathrm{L}}{ }_{\mathrm{L}}\)
\(f s t_{\mathrm{L}}^{\prime}:: L^{\mathrm{T}}(\) Tag \(a\), Tag b) \(a\)
\(f s t_{\mathrm{L}}^{\prime}=L\left(\lambda\left(a,{ }_{-}\right) \rightarrow \operatorname{unTag} a\right)\left(\lambda\left({ }_{-}, b\right) a \rightarrow(U a, b)\right)\)
\(s n d_{\mathrm{L}}^{\prime}:: L^{\mathrm{T}}(\operatorname{Tag} a, \operatorname{Tag} b) b\)
\(s n d_{\mathrm{L}}^{\prime}=L(\lambda(-, b) \rightarrow u n T a g b)\left(\lambda\left(a,{ }_{-}\right) b \rightarrow(a, U b)\right)\)
\(\operatorname{tag}_{\mathrm{L}}:: L(a, b)(\) Tag \(a\), Tag \(b)\)
\({\operatorname{tag} 2_{\mathrm{L}}}=L(\lambda(a, b) \rightarrow(O a, O b))\)
    \(\left(\lambda_{-}(a, b) \rightarrow(\right.\) unTag \(a\), unTag \(\left.b)\right)\)
```

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.

```
unlift2 (lift2 \(\ell\) )
\(=\{\) definition unfolding \(\& \beta\)-reduction \(\}\)
    \(\ell \hat{\circ} f s t_{\mathrm{L}}^{\prime} \circledast s n d_{\mathrm{L}}^{\prime}\) ô \(\operatorname{tag}_{\mathrm{L}}\)
\(=\{\) unfolding \((\circledast)\}\)
    \(\ell \hat{o}\left(f s t_{\mathrm{L}}^{\prime} \hat{\otimes} s n d_{\mathrm{L}}^{\prime}\right) \hat{o} d u p_{\mathrm{L}} \hat{o} \operatorname{tag} 2_{\mathrm{L}}\)
\(=\left\{\left(f s t_{\mathrm{L}}^{\prime} \hat{\otimes} s n d_{\mathrm{L}}^{\prime}\right) \hat{\circ} \operatorname{dup}_{\mathrm{L}} \hat{o}{\operatorname{tag} \mathscr{2}_{\mathrm{L}}}=i d_{\mathrm{L}}-\left(^{*}\right)\right\}\)
```

We prove the statement $\left(^{*}\right)$ by showing get $\left(\left(f s t_{\mathrm{L}}^{\prime} \hat{\otimes} s n d_{\mathrm{L}}^{\prime}\right) \hat{o}\right.$ $\left.d u p_{\mathrm{L}} \hat{o} \operatorname{tag} 2_{\mathrm{L}}\right)(a, b)=(a, b)$ and $p u t\left(\left(f s t_{\mathrm{L}}^{\prime} \hat{\otimes} s n d_{\mathrm{L}}^{\prime}\right) \hat{o} d u p_{\mathrm{L}} \hat{\circ}\right.$

[^0]$\left.\operatorname{tag}_{\mathrm{L}}^{\mathrm{L}}\right)(a, b)\left(a^{\prime}, b^{\prime}\right)=\left(a^{\prime}, b^{\prime}\right)$. Since the former property is easy to prove, we only show the latter here.

```
        put \(\left(\left(f s t_{\mathrm{L}}^{\prime} \hat{\otimes} s n d_{\mathrm{L}}^{\prime}\right) \hat{\circ} d u p_{\mathrm{L}} \hat{\circ} \operatorname{tag} 2_{\mathrm{L}}\right)(a, b)\left(a^{\prime}, b^{\prime}\right)\)
\(=\{\) definition unfolding \& \(\beta\)-reduction \(\}\)
    put \(\operatorname{tag}_{2_{\mathrm{L}}}(a, b) \$\)
        put \(\left(\left(f s t_{\mathrm{L}}^{\prime} \hat{\otimes} s n d_{\mathrm{L}}^{\prime}\right) \hat{o} d u p_{\mathrm{L}}\right)(O a, O b)\left(a^{\prime}, b^{\prime}\right)\)
\(=\{\) definition unfolding \& \(\beta\)-reduction \(\}\)
        put \(\operatorname{tag}_{\mathrm{L}}(a, b) \$\)
            put dup \(\mathrm{L}_{\mathrm{L}}(\mathrm{O} a, O b) \$\)
            (put \(f s t_{\mathrm{L}}^{\prime}(O a, O b) a^{\prime}\), put \(\left.s n d_{\mathrm{L}}^{\prime}(O a, O b) b^{\prime}\right)\)
\(=\left\{\right.\) definitions of \(f s t_{\mathrm{L}}^{\prime}\) and \(\left.s n d_{\mathrm{L}}^{\prime}\right\}\)
        put tag2 \({ }_{\mathrm{L}}(a, b) \$\)
            \({\operatorname{put~} \operatorname{dup}_{\mathrm{L}}}(O a, O b)\left(\left(U a^{\prime}, O b\right),\left(O a, U b^{\prime}\right)\right)\)
\(=\left\{\right.\) definition of \(\left.d u p_{\mathrm{L}}\right\}\)
        put tag2 \({ }_{\mathrm{L}}(a, b)\left(U a^{\prime}, U b^{\prime}\right)\)
\(=\left\{\right.\) definition of \(\left.\operatorname{tag} 2_{\mathrm{L}}\right\}\)
        ( \(\left.a^{\prime}, b^{\prime}\right)\)
```

Thus, we have proved that lift2 is injective.
We can recreate $f s t_{\mathrm{L}}$ and $s n d_{\mathrm{L}}$ with unlift2, which is rather reassuring.
Proposition 5. $f s t_{\mathrm{L}}=$ unlift2 $f s t$ and $s n d_{\mathrm{L}}=$ 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 $f s t_{\mathrm{L}}$ and $f s t$ : while unlifting both functions result in $f_{s t_{\mathrm{L}}}$, they actually differ as put $\left(l i f t 2 f_{s} t_{\mathrm{L}}\left(f s t_{\mathrm{L}}^{\prime}, s n d_{\mathrm{L}}^{\prime}\right)\right)(O a, O b) c=(U c, U b)$ and put $\left(f s t\left(f s t_{\mathrm{L}}^{\prime}, s n d_{\mathrm{L}}^{\prime}\right)\right)(O a, O b) c=(U c, O b)$. Intuitively, $f s t$ knows that the second argument is unused, while lift2 fst $_{\mathrm{L}}$ does not because $f s t_{\mathrm{L}}$ 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 $f s t_{\mathrm{L}}^{\prime}, s n d_{\mathrm{L}}^{\prime}, d u p_{\mathrm{L}}$ and $\operatorname{tag}{\underset{\mathrm{L}}{\mathrm{L}}}$. 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 $d u p_{\mathrm{L}}, f s t_{\mathrm{L}}^{\prime}, s n d_{\mathrm{L}}^{\prime}$ and $\operatorname{tag} \mathcal{Z}_{\mathrm{L}}$ are not well-behaved, for different reasons. For functions $f s t_{\mathrm{L}}^{\prime}$ and $s n d_{\mathrm{L}}^{\prime}$, the difference from the original versions $f s t_{\mathrm{L}}$ and $s n d_{\mathrm{L}}$ 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 $d u p_{\mathrm{L}}$ and $\operatorname{tag} 2_{\mathrm{L}}$, 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 $f:: \forall s .\left(L^{\mathrm{T}} s A, L^{\mathrm{T}} s B\right) \rightarrow L^{\mathrm{T}} s C$, 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 $d u p_{\mathrm{L}}$. To maintain safety, unequal values as duplications are only allowed if they have different tags (i,e,. one value must be
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 $s^{\prime}$ such that $s \preceq s^{\prime}$ and $s^{\prime} \neq \uparrow s$.

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 $s^{\prime}$ such that $\uparrow s^{\prime}=\uparrow s, v=$ get $\ell s^{\prime}$ holds.
- (Backward Inflation) For all minimal (with respect to $\preceq$ ) $s$, if put $\ell s v$ succeeds as $s^{\prime}$, then $s \preceq s^{\prime}$.
- (Local Acceptability) For all $s, s \preceq p u t \ell s($ get $\ell s) \preceq \uparrow s$.
- (Local Consistency) For all $s$ and $v$, assuming put $\ell s v$ succeeds as $s^{\prime}$, then for all $s^{\prime \prime}$ with $s^{\prime} \preceq s^{\prime \prime}$, get $\ell s^{\prime \prime}=v$ holds.

In the above, tags introduced for the flexible behavior of put must not affect the behavior of get: $\uparrow s^{\prime}=\uparrow s$ means that $s$ and $s^{\prime}$ 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 $s^{\prime}$ to the same view. The idea is that $O$-tagged elements in $s^{\prime}$ 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 $d u p_{\mathrm{L}}$ and $\operatorname{tag} \mathcal{2}_{\mathrm{L}} / t a g_{\mathrm{L}}$ introduce tagged views; but they are always precomposed when used, as shown in the following.

We have the following compositional properties for local wellbehavedness.

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 wellbehaved.
- If $x$ and $y$ are locally well-behaved, $x$ ô $\operatorname{tag} 2_{\mathrm{L}}$ and $y$ ô $\operatorname{tag}_{\mathrm{L}}$ 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.

$$
\begin{aligned}
& \text { put }\left((x \hat{\otimes} y) \hat{o} d u p_{\mathrm{L}}\right) s\left(\text { get }\left((x \hat{\otimes} y) \hat{\circ} d u p_{\mathrm{L}}\right) s\right) \\
= & \{\operatorname{simplification~}\} \quad \text { put dup } s(\text { put }(x \hat{\otimes} y)(s, s)(\operatorname{get}(x \hat{\otimes} y)(s, s))) \\
= & \{\text { by the local acceptability of } x \hat{\otimes} y\} \\
& \text { put dup } \operatorname{dup}_{\mathrm{L}} s\left(s^{\prime}, s^{\prime \prime}\right)-\text { where } s \preceq s^{\prime} \preceq \uparrow s, s \preceq s^{\prime \prime} \preceq \uparrow s \\
= & \left\{\text { by the definition of } d u p_{\mathrm{L}} \text { and that } s^{\prime} \curlyvee s^{\prime \prime} \text { is defined }\right\} \\
& s^{\prime} \curlyvee s^{\prime \prime} \preceq \uparrow s
\end{aligned}
$$

Note that, since $s^{\prime} \preceq \uparrow s$ and $s^{\prime \prime} \preceq \uparrow s$, there is $s^{\prime} \curlyvee s^{\prime \prime} \preceq \uparrow s$.
Then, we prove local consistency. Assume that put $((x \dot{\otimes} y) \hat{o}$ $\left.d u p_{\mathrm{L}}\right) s\left(v_{1}, v_{2}\right)$ succeeds in $s^{\prime}$. Then, by the following calculation, we have $s^{\prime}=$ put $x \leqslant v_{1} \curlyvee$ put $y s v_{2}$.

```
    put \(\left((x \hat{\otimes} y) \hat{o} d u p_{\mathrm{L}}\right) s\left(v_{1}, v_{2}\right)\)
\(=\{\) simplification \(\}\)
    put \(\operatorname{dup}_{\mathrm{L}} s\) (put \(x\) s \(v_{1}\), put \(y s v_{2}\) )
\(=\{\) definition unfolding \(\}\)
    put \(x s v_{1} \curlyvee\) put \(y s v_{2}\)
```

Let $s^{\prime \prime}$ be a source such that $s^{\prime} \preceq s^{\prime \prime}$. Then, we prove get $((x \hat{\otimes}$ $\left.y) @ d u p_{\mathrm{L}}\right) s^{\prime \prime}=\left(v_{1}, v_{2}\right)$ as follows.

```
        get ((x\hat{\otimes}y)\hat{o}du\mp@subsup{p}{\textrm{L}}{})\mp@subsup{s}{}{\prime\prime}(\mp@subsup{v}{1}{},\mp@subsup{v}{2}{})
    = { simplification }
        (get x s '', get y s')
={ the local consistency of }x\mathrm{ and }y
    (vi, v2)
```

Note that we have put $x s v_{1} \preceq s^{\prime} \preceq s^{\prime \prime}$ and put y $s v_{2} \preceq s^{\prime} \preceq$ $s^{\prime \prime}$ by the definition of $\curlyvee$.

Forward tag-irrelevance and backward inflation are straightforward.

## Corollary 1. The following properties hold.

- lift $\ell:: \forall s . L^{\mathrm{T}} s A \rightarrow L^{\mathrm{T}} s$ B preserves local well-behavedness, if $\ell:: L^{\mathrm{T}} A B$ is well-behaved.
- lift2 $\ell:: \forall s$. $\left(L^{\mathrm{T}}\right.$ s $\left.A, L^{\mathrm{T}} s B\right) \rightarrow L^{\mathrm{T}}$ 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 $L^{\mathrm{T}}$ to be abstract.
Definition 3 (Abstract Nature of $L^{\mathrm{T}}$ ). We say $L^{\mathrm{T}}$ is abstract in $f:: \tau$ if there is a polymorphic function $h$ of type

$$
\begin{aligned}
& \forall \ell .\left(\forall a b . L^{\mathrm{T}} a b \rightarrow(\forall s . \ell s a \rightarrow \ell s b)\right) \\
& \quad \rightarrow\left(\forall a b \cdot(\forall s \cdot \ell s a \rightarrow \ell s b) \rightarrow L^{\mathrm{T}} a b\right) \\
& \quad \rightarrow(\forall s a b . \ell s a \rightarrow \ell s b \rightarrow \ell s(a, b)) \\
& \quad \rightarrow\left(\forall a b c \cdot(\forall s .(\ell s a, \ell s b) \rightarrow \ell x c) \rightarrow L^{\mathrm{T}}(a, b) c\right) \\
& \rightarrow \tau^{\prime}
\end{aligned}
$$

satisfying $f=h$ lift unlift $(\circledast)$ unlift2 and $\tau^{\prime}=\tau\left[\ell / L^{\mathrm{T}}\right]$.
Then, we obtain the following properties from the free theorems [34, 37].
Theorem 4. Let $f$ be a function of type $\forall s .\left(L^{\mathrm{T}} s A, L^{\mathrm{T}} s B\right) \rightarrow$ $L^{\mathrm{T}} s C$ in which $L^{\mathrm{T}}$ is abstract. Then, $f(x, y)$ is locally wellbehaved 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 $\forall s .\left(L^{\mathrm{T}} s A, L^{\mathrm{T}} s B\right) \rightarrow$ $L^{\mathrm{T}} s C$ in which $L^{\mathrm{T}}$ 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.

$$
\begin{aligned}
& \operatorname{swap}_{\mathrm{L}}::(E q a, E q b) \Rightarrow L(a, b)(b, a) \\
& \left.\operatorname{swap}_{\mathrm{L}}=\text { unlift2 }^{(l i f t 2} \text { id }_{\mathrm{L}} \circ \text { swap }\right)
\end{aligned}
$$

And it behaves as expected.

```
    put \(\operatorname{swap}_{\mathrm{L}}(1,2)(4,3)\)
\(=\{\) unfold definitions \(\}\)
    put \(\left(\left(s n d_{\mathrm{L}}^{\prime} \hat{\otimes} f s t_{\mathrm{L}}^{\prime}\right) \hat{o} d u p_{\mathrm{L}} \hat{\circ} \operatorname{tag}_{\mathrm{L}}\right)(1,2)(4,3)\)
\(=\{\) simplifications \(\}\)
    put tag2 \({ }_{\mathrm{L}}(1,2) \$\)
        put dup \(p_{\mathrm{L}}\left(\begin{array}{ll}O & 1, O\end{array}\right) \$\)
        \(\left(\right.\) put snd \(d_{\mathrm{L}}^{\prime}(O 1, O 2) 4\), put fst \(\left.t_{\mathrm{L}}^{\prime}(O 1, O 2) 3\right)\)
\(=\left\{\right.\) definition of \(f s t_{\mathrm{L}}^{\prime}\) and \(\left.s n d_{\mathrm{L}}^{\prime}\right\}\)
    put tag2 \({ }_{\mathrm{L}}(1,2) \$\)
        put \(\operatorname{dup}_{\mathrm{L}}(O 1, O 2)((O 1, U 4),(U 3, O 2))\)
```

```
={definitions of dup }\mp@subsup{\textrm{L}}{\textrm{L}}{}\mathrm{ and tag2 }\mp@subsup{\mathcal{L}}{\textrm{L}}{}
    (3,4)
```

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.

$$
\begin{aligned}
& \text { split3 }::\left(L^{\mathrm{T}} s a, L^{\mathrm{T}} s b, L^{\mathrm{T}} s c\right) \rightarrow L^{\mathrm{T}} s(a, b, c) \\
& \text { split3 }(x, y, z)=\text { lift flatten } L_{\mathrm{L}}((x \circledast y) \circledast z) \\
& \text { where } \text { flatten }_{\mathrm{L}}:: L((a, b), c)(a, b, c) \\
& \text { flatten } L_{\mathrm{L}}=L(\lambda((x, y), z) \rightarrow(x, y, z)) \\
&\left(\lambda_{-}(x, y, z) \rightarrow((x, y), z)\right) \\
& \text { lift3 } \ell t= \text { lift } \ell(\text { split3 } t)
\end{aligned}
$$

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.

$$
\begin{array}{r}
\text { split3 }^{\prime}(x, y, z)=\text { lift flatten } R_{\mathrm{L}}(x \circledast(y \circledast z)) \\
\text { where flatten } R_{\mathrm{L}}:: L((a, b), c)(a, b, c) \\
\text { flatten }_{\mathrm{L}}=L(\lambda(x,(y, z)) \rightarrow(x, y, z)) \\
\quad\left(\lambda_{-}(x, y, z) \rightarrow(x,(y, z))\right)
\end{array}
$$

The two definitions split3 and split3' coincide.
To complete the picture, the nullary lens function

$$
\begin{aligned}
& \text { unit }:: \forall s . L^{\mathrm{T}} s() \\
& \text { unit }=L\left(\lambda_{-} \rightarrow()\right)(\lambda s() \rightarrow s)
\end{aligned}
$$

is the unit for $(\circledast)$. Theoretically $\left(L^{\mathrm{T}} s(-), \circledast\right.$, 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.

$$
\begin{aligned}
& \text { new :: Eq } a \Rightarrow a \rightarrow \forall s . L^{\mathrm{T}} s a \\
& \text { new } a=\text { lift }\left(L(\text { const } a)\left(\lambda_{-} a^{\prime} \rightarrow \text { check } a a^{\prime}\right)\right) \text { unit } \\
& \text { where } \\
& \text { check } a a^{\prime}= \\
& \text { if } a==a^{\prime} \text { then () } \\
& \quad \text { else error "Update on constant" }
\end{aligned}
$$

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 $\ell:: L S A \rightarrow L S B$ for any $S$. In the case that $S$ is tagged and thus partially ordered, $\left(L^{\mathrm{T}} S(-), \circledast\right.$, unit) forms a lax monoidal functor, under the following conditions.

- $(*)$ must be natural, i.e., (lift $f x) \circledast($ lift $g y)=\operatorname{lift}(f \hat{\otimes}$ $g)(x \circledast y)$ for all $f, g, x$ and $y$ with appropriate types.
- split3 and split3 ${ }^{\prime}$ coincide.
- lift elimUnitL $L_{\mathrm{L}}(u n i t \circledast x)=x$ must hold where elimUnit $L_{\mathrm{L}}:$ : $L((), a) a$ 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 $x s=p u t y s$ for all minimal $s$. This equivalence is preserved by manipulations of $L^{\mathrm{T}}$-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 $\ell$.
- $x \equiv y$ and $z \equiv w$ implies $x \circledast z \equiv y \circledast w$.
- $x \equiv y$ implies $x$ o $\operatorname{tag} g_{\mathrm{L}}=y$ o $\operatorname{tag} \mathrm{L}_{\mathrm{L}}\left(\right.$ or $x$ o $\operatorname{tag} \mathscr{2}_{\mathrm{L}}=y$ ô $\left.\operatorname{tag} \mathscr{2}_{\mathrm{L}}\right)$.

Note that the above three cases cover the only ways to construct/destruct $L^{\mathrm{T}}$ in $f$ when $L^{\mathrm{T}}$ is abstract. The third condition says that this "coarse" equivalence ( $\equiv$ ) on $L^{\mathrm{T}}$ can be "sharpened" to the usual extensional equality $(=)$ by $\operatorname{tag} \mathrm{L}_{\mathrm{L}}$ and $\operatorname{tag} 2_{\mathrm{L}}$ in the unlifting functions.

It is known that an Applicative functor in Haskell corresponds to a monoidal functors [29]. However, we cannot use an Applicativelike 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 $\ell$ ) :: $\left[\begin{array}{lll}L^{\mathrm{T}} & s & A\end{array}\right] \rightarrow\left[\begin{array}{lll}L^{\mathrm{T}} & s & B\end{array}\right]$. 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.

$$
\begin{aligned}
& \text { lsequence }_{\text {list }}::\left[\begin{array}{lll}
L^{\mathrm{T}} & s & a
\end{array}\right] \rightarrow L^{\mathrm{T}} s[a] \\
& \text { lsequence }_{\text {list }}[]=\text { lift nil }_{\mathrm{L}} \text { unit } \\
& \text { lsequence }_{\text {list }}(x: x s)=\text { lift2 }^{\text {cons }}{ }_{\mathrm{L}}\left(x, \text { lsequence }_{\text {list }} x s\right) \\
& n i l_{\mathrm{L}}=L(\lambda() \rightarrow[])(\lambda()[] \rightarrow()) \\
& \operatorname{cons}_{\mathrm{L}}=L(\lambda(a, a s) \rightarrow(a: a s)) \\
& \left(\lambda_{-}\left(a^{\prime}: a s^{\prime}\right) \rightarrow\left(a^{\prime}, a s^{\prime}\right)\right)
\end{aligned}
$$

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

$$
\begin{aligned}
& \text { lift }_{\text {list }}:: L[a] b \rightarrow \forall s .\left[\begin{array}{lll}
L^{\mathrm{T}} & s & a
\end{array}\right] \rightarrow L^{\mathrm{T}} s \\
& \text { lift }_{\text {list }} \ell x s=\text { lift } \ell\left(\text { lsequence }_{\text {list }} x s\right)
\end{aligned}
$$

Tagged lists form an instance of Poset.

$$
\begin{aligned}
& \text { instance Poset } a \Rightarrow \text { Poset }[a] \text { where } \\
& x s \curlyvee y s= \\
& \text { if length xs }=\text { length ys } \\
& \text { then zipWith }(\curlyvee) \text { xs ys } \\
& \text { else } \perp-- \text { Unreachable in our framework }
\end{aligned}
$$

Note that the requirement that $x s$ and $y s$ must has the same shape is made explicit above, though it is automatically enforced by the abstract use of $L^{\mathrm{T}}$ in lifted functions.

The definition of unlift $_{\text {list }}$ 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 $f$.

```
unlift \(_{\text {list }}:: \forall a b . E q a \Rightarrow\)
            \(\left(\forall s .\left[L^{\mathrm{T}} s a\right] \rightarrow L^{\mathrm{T}} s b\right) \rightarrow L[a] b\)
unlift \(_{\text {list }} f=L(\lambda s \rightarrow \operatorname{get}(\) mkLens \(s) s)\)
            \((\lambda s \rightarrow\) put \((m k L e n s s) s)\)
where
    mkLens \(s=f(\) projs \((\) length \())\) ô tagList \(_{\mathrm{L}}\)
    tagList \(_{\mathrm{L}}=L(\) map \(O)\left(\lambda_{-}\right.\)ys \(\rightarrow\) map unTag ys \()\)
    projs \(n=\) map \(^{\operatorname{proj}}{ }_{\mathrm{L}}[0 \ldots n-1]\)
    \(\operatorname{proj}_{\mathrm{L}}::\) Int \(\rightarrow L^{\mathrm{T}}[\) Tag a] \(a\)
    \(\operatorname{proj}_{\mathrm{L}} i=L(\lambda x s \rightarrow \operatorname{unTag}(x s!!i))\)
            ( \(\lambda\) as \(a \rightarrow\) update \(i\left(\begin{array}{ll}U & a\end{array}\right)\) as)
```

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 lift2, lift $_{\text {list }}$ is an injection with unlift $_{\text {list }}$ as its left inverse.
Example 2 (Bidirectional tail). Let us consider the function tail.

$$
\begin{aligned}
& \text { tail }::[a] \rightarrow[a] \\
& \text { tail }(x: x s)=x s
\end{aligned}
$$

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

$$
\left.\begin{array}{l}
\operatorname{tail}_{\mathrm{L}}:: E q a \Rightarrow L[a][a] \\
\text { tail }_{\mathrm{L}}=\text { unlift }_{\text {list }}
\end{array} \text { (lsequence }_{\text {list }} \circ \text { tail }^{2}\right) .
$$

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

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

```
unlines [] = ""
unlines (x:xs)=catLine x (unlines xs)
catLine x y = x # "\n" # y
```

To construct a bidirectional version of unlines, we first need a bidirectional version of catLine.

```
catLine \(_{\mathrm{L}}:: L\) (String, String) String
catLine \(_{\mathrm{L}}=\)
    \(L(\lambda(s, t) \rightarrow s+\) " \(\backslash \mathrm{n} "+t)\)
        \((\lambda(s, t) u \rightarrow\) let \(n=\) length \((\) filter \((==, \backslash \mathrm{n} \prime) s)\)
                \(i=\) elemIndices \(, \backslash \mathrm{n}, u!!n\)
                \(\left(s^{\prime}, t^{\prime}\right)=\) splitAt \(i u\)
            in \(\left(s^{\prime}\right.\), tail \(\left.\left.t^{\prime}\right)\right)\)
```

Here, elemIndices and splitAt are functions from Data.List: elemIndices $c s$ returns the indices of all elements that are equal to $c$; 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, put catLine ${ }_{\mathrm{L}}(s, t) u$ splits $u$ into $s^{\prime}$ and " $\backslash \mathrm{n} "+t^{\prime}$ so that $s^{\prime}$ contains the same number of newlines as the original s. For example, put catLine $\mathrm{L}_{\mathrm{L}}($ "a $\backslash \mathrm{nbc}$ ", "de") "A $\mathrm{nB} \backslash \mathrm{nC} "=$ ("A\nB", "C").

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

```
unlines \(_{\mathrm{L}}:: L\) [String \(]\) String
unlines \(_{\mathrm{L}}=\) unlift \(_{\text {list }}\) unlines \(_{\mathrm{F}}\)
unlines \(_{\mathrm{F}}:: \forall s .\left[L^{\mathrm{T}} s\right.\) String \(] \rightarrow L^{\mathrm{T}}\) s String
unlines \(_{\mathrm{F}}[] \quad=\) new ""
unlines \(_{\mathrm{F}}(x: x s)=\) lift2 \(^{\text {catLine }_{\mathrm{L}}}\left(x\right.\), unlines \(\left._{\mathrm{F}} x s\right)\)
```

As one can see, unlines ${ }_{F}$ is written in the same applicative style as 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., catLine and "") of the data with the corresponding bidirectional versions (e.g., lift2 catLine ${ }_{\mathrm{L}}$ and new "").

Lens 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 put unlines $\mathrm{s}_{\mathrm{L}}$ ["a", "b", "c"] "AA $\mathrm{nBB} \backslash \mathrm{nCC} \backslash \mathrm{n} "=[" \mathrm{AA} ", ~ " \mathrm{BB} ", ~ " C C "]$, but put unlines $\mathrm{L}_{\mathrm{L}}$ ["a", "b", "c"] "AA\nBB\n" $=\perp$ and put unlines ${ }_{\mathrm{L}}[$ "a", "b", "c"] "AA $\mathrm{nBB} \backslash \mathrm{nCC} \backslash \mathrm{nD} "=\perp$.

Example 4 (unlines defined by foldr). Another common way to implement unlines is to use foldr, as below.

$$
\text { unlines }=\text { foldr catLine "" }
$$

The same coding principle for constructing bidirectional versions applies.

$$
\begin{aligned}
& \text { unlines }_{\mathrm{L}}:: L[\text { String }] \text { String } \\
& \text { unlines }_{\mathrm{L}}=\text { unlift }_{1 \text { list }} \text { unlineses } \\
& \text { unlines }_{\mathrm{F}}:: \forall s .\left[L^{\mathrm{T}} \text { s String }\right] \rightarrow L^{\mathrm{T}} \text { s String } \\
& \text { unlines }_{\mathrm{F}}=\text { foldr }\left(\text { lift2 catLine }_{\mathrm{L}}\right)(\text { new } " \mathrm{c})
\end{aligned}
$$

The new unlines $_{\mathrm{F}}$ is again in the same applicative style as the new 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 containerlike 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 traverse, which can be found in Data.Traversable, to give data-type generic lifting and unlifting functions.

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

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

```
newtype Const \(a b=\) Const \(\{\) getConst \(:: a\}\)
contents :: Traversable \(t \Rightarrow t a \rightarrow[a]\)
contents \(t=\) getConst (traverse \((\lambda x \rightarrow\) Const \([x]) t)\)
```

[^1]```
fill:: Traversable t=>tb b [a] ->t a
fill t \ell = evalState (traverse next t) \ell
    where
        next _ = do ( }a:x)\leftarrow\mathrm{ Control.Monad.State.get
                Control.Monad.State.put x
                        return a
```

Here, Const $a b$ is an instance of the Haskell Functor that ignores its argument $\mathbf{b}$. It becomes an instance of 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 Traversable are straightforward to define to the extend of being systematically derivable [23]. The instances of Traversable must satisfy certain laws [3]; and for such lawful instances, we have

$$
\begin{aligned}
& \text { fill }(\text { fmap } f t)(\text { contents } t)=t \quad \text { (FillContents) } \\
& \text { contents }(\text { fill } t x s)=x s \text { if length } x s=\text { length }(\text { contents } t)
\end{aligned}
$$

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

We can now define a generic lsequence function as follows.

```
lsequence :: (Eq a, Eq ( \(t())\), Traversable \(t) \Rightarrow\)
    \(t\left(L^{\mathrm{T}} s a\right) \rightarrow L^{\mathrm{T}} s(t a)\)
lsequence \(t=\)
    lift \(\left(\right.\) fill \(_{\mathrm{L}}(\) shape \(\left.t)\right)\left(\right.\) lsequence \(_{\text {list }}(\) contents \(\left.t)\right)\)
    where
        \(f_{i l l_{\mathrm{L}}} s=L(\lambda x s \rightarrow\) fill \(s x s)\left(\lambda_{-} t \rightarrow\right.\) contents \(\left.^{\prime} s t\right)\)
        contents \({ }^{\prime} s t=\) if shape \(t=s\)
                        then contents \(t\)
                            else error "Shape Mismatch"
```

Here, shape computes the shape of a structure by replacing elements with units, i.e., shape $t=\operatorname{fmap}\left(\lambda_{-} \rightarrow()\right) t$. Also, we can make a Poset instance as follows. ${ }^{3}$

```
instance (Poset a, Eq ( \(t()\) ), Traversable \(t) \Rightarrow\)
    Poset ( \(t\) a) where
    \(t_{1} \curlyvee t_{2}=\) if shape \(t_{1}==\) shape \(t_{2}\)
        then fill \(t_{1}\) (contents \(t_{1} \curlyvee\) contents \(t_{2}\) )
        else \(\perp\)-- Unreachable, in our framework
```

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

```
unliftT :: (Eq (t ()), Eq a, Traversable t) =>
    (\foralls.t(LT
unliftT f}=L(\lambdas->\operatorname{get}(mkLens s) s
            (\lambdas-> put (mkLens s) s)
where
    mkLens s=f(projTs (shape s))ô tagT T
    tagT}\mp@subsup{T}{\textrm{L}}{}=L(fmap O)(const $ fmap unTag)
    projTs sh=
        let n= length (contents sh)
        in fill sh [projT }\mp@subsup{\textrm{L}}{\textrm{L}}{}\mathrm{ i sh | i}[[0\ldotsn-1]
    projT}\mp@subsup{T}{\textrm{L}}{}\mathrm{ i sh =
        L(unTag\circ(!!i) ○ contents)
            (\lambdasv-> fill sh (update i (U v)(contents s)))
```

[^2]Here, $\operatorname{proj}_{\mathrm{L}} i t$ is a bidirectional transformation that extracts the $i$ th element in $t$ with the tag erased. Similarly to $u n l i f t ~_{\text {list }}$, 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.

$$
\begin{aligned}
& \text { data Exp } \begin{aligned}
& \text { ENum Int } \mid \text { EInc Exp } \\
& \text { EVar String } \mid \text { EApp Exp Exp } \\
& \text { EFun String Exp deriving Eq }
\end{aligned} \\
& \text { data Val } a=\text { VNum a } \\
& \\
& \text { data Env } a=\text { VFun String Exp }(\text { Env a) deriving Eq }[(\text { String, Val a)] deriving Eq }
\end{aligned}
$$

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

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

```
eval :: Env \(\left(L^{\mathrm{T}}\right.\) s Int \() \rightarrow \operatorname{Exp} \rightarrow \operatorname{Val}\left(L^{\mathrm{T}}\right.\) s Int \()\)
eval env (ENum n) \(=\operatorname{VNum}(\) new \(n)\)
eval env (EInc e) = let VNum \(v=\) eval env \(e\)
        in VNum (lift inc \(\mathrm{L}_{\mathrm{L}} v\) )
eval env \((E \operatorname{Var} x)=\) lkup \(x\) env
eval env \(\left(E A p p e_{1} e_{2}\right)=\) let VFun \(x e^{\prime}(E n v e n v ')=\)
                                    eval env \(e_{1}\)
        \(v_{2}=\) eval env \(e_{2}\)
        in eval \(\left(\operatorname{Env}\left(\left(x, v_{2}\right): e n v^{\prime}\right)\right) e^{\prime}\)
eval env \((\) EFun \(x e)=\) VFun \(x\) e env
```

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

$$
i n c_{\mathrm{L}}=L(+1)\left(\lambda_{-} x \rightarrow x-1\right)
$$

and lkup :: String $\rightarrow$ Env $a \rightarrow a$ is a lookup function. A lens eval ${ }_{\mathrm{L}}:: \operatorname{Exp} \rightarrow L$ (Env Int) (Val Int) naturally arises from eval.

$$
\begin{aligned}
& \text { eval }_{\mathrm{L}}:: \text { Exp } \rightarrow L \text { (Env Int) (Val Int) } \\
& \text { eval }{ }_{\mathrm{L}} e=\text { unlift } T\left(\lambda e n v \rightarrow \text { lift }^{\text {i }} \text { id }_{\mathrm{L}} \$\right. \text { eval env e) }
\end{aligned}
$$

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.

```
expr = twice @@ twice @@ twice @@ twice @@ inc @@ x
    where
        twice = EFun "f"$ EFun "x" $
            EVar "f"@@(EVar "f"@@ EVar "x")
    x = EVar "x"
    inc = EFun "x" $ EInc (EVar "x")
```

```
infixl 9 @@ -- @@ is left associative
\((@)=E A p p\)
```

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

```
expr \(=((((\) twice twice \()\) twice \()\) twice \()\) inc \() x\)
    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 $e n v_{0}=$ Env [("x", VNum 3)].

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

As a remark, this seemingly innocent implementation of eval ${ }_{\mathrm{L}}$ 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 definitionsomething 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 $\left(L^{\mathrm{T}} s A\right)$-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 $L A B \rightarrow L C D$ we have

$$
\begin{aligned}
& \text { liftC }:: E q a \Rightarrow(L a b \rightarrow L \quad c d) \rightarrow \\
& \quad\left(\forall s . L^{\mathrm{T}} s a \rightarrow L^{\mathrm{T}} s \quad b\right) \rightarrow\left(\forall t . L^{\mathrm{T}} t c \rightarrow L^{\mathrm{T}} t d\right) \\
& \text { liftC } c f=\operatorname{lift}(c(\text { unlift } f))
\end{aligned}
$$

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 $\lambda x \rightarrow$ $\ldots c(\ldots x \ldots) \ldots$ does not type-check when $c$ is a lifted combinator.

As an example, let us consider the following lens combinator mapDefault $_{\mathrm{C}}$.

$$
\begin{aligned}
& \text { mapDefault }_{\mathrm{C}}:: a \rightarrow L \text { a } b \rightarrow L[a][b] \\
& \text { mapDefault }_{\mathrm{C}} d \ell=L(\text { map }
\end{aligned} \begin{aligned}
& \text { (get } \ell))(\lambda s v \rightarrow \text { go } s v) \\
\text { where } g o s s \quad[] & =[] \\
\text { go }[] \quad(v: v s) & =\text { put } \ell v: \text { go }[] \text { vs } \\
\text { go }(s: s s)(v: v s) & =\text { put } \ell v: g o \text { ss vs }
\end{aligned}
$$

When given a lens on elements, mapDefault ${ }_{\mathrm{C}} d$ 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 ${ }_{\mathrm{C}}$ as the following, which in the forward direction is essentially map (uncurry $(+)$ ).

```
mapAdd L :: L[(Int, Int)] [ Int]
mapAddd
```

```
\(\operatorname{mapAdd}_{\mathrm{F}} x s=\operatorname{map}_{\mathrm{F}}(0,0)(\) lift addL \()\) xs
map \(_{\mathrm{F}} d=\) liftC ( mapDefault \(_{\mathrm{C}} d\) )
\(a d d L=L(\lambda(x, y) \rightarrow x+y)\left(\lambda\left(x,{ }_{-}\right) v \rightarrow(x, v-x)\right)\)
```

This lens $\operatorname{mapA}_{2} d_{\mathrm{L}}$ constructed in our framework handles shape changes without any trouble.

```
Main> put mapAdd d [(1,1), (2, 2)] [3,5]
[(1,2),(2,3)]
Main> put mapAddd 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 $\operatorname{map}_{\mathrm{F}}(0,0)$ (lift addL) has type $\forall s . L^{\mathrm{T}} s[($ Int, Int $)] \rightarrow L^{\mathrm{T}} s$ [Int], where the list occurs inside $L^{\mathrm{T}} s$, contrasting to map (lift addL)'s type $\forall s .\left[L^{\mathrm{T}} s(\right.$ Int, Int $\left.)\right] \rightarrow\left[L^{\mathrm{T}} s\right.$ Int $]$. Intuitively, the type constructor $L^{\mathrm{T}} s$ can be seen as an updatability annotation; $L^{\mathrm{T}} s[($ Int, Int $)]$ means that the list itself is updatable, whereas $\left[L^{\mathrm{T}} s(\right.$ 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 ${ }_{\mathrm{C}}$ 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 \(_{\mathrm{C}}:: E q k \Rightarrow a \rightarrow L a b \rightarrow L[(k, a)][(k, b)]\)
\(\operatorname{mapByKey}_{\mathrm{C}} d \ell=L(\operatorname{map}(\lambda(k, s) \rightarrow(k\), get \(\ell s)))\)
    \((\lambda s v \rightarrow g o s v)\)
    where go ss [] \(=[]\)
    go ss \(((k, v): v s)=\)
        case lookup \(k\) ss of
            Nothing \(\rightarrow(k\), put \(\ell d v)\) : go ss vs
            Just \(s \rightarrow(k, p u t \ell s v):\) go (del \(k\) ss \() v s\)
        del \(k[] \quad=[]\)
        del \(k\left(\left(k^{\prime}, s\right): s s\right) \mid k==k^{\prime}=s s\)
        \(\mid\) otherwise \(=\left(k^{\prime}, s\right):\) del \(k\) ss
```

Lenses constructed with mapByKey ${ }_{\mathrm{C}}$ match with keys instead of positions.

$$
\begin{aligned}
& \operatorname{mapAddByKey}_{\mathrm{L}}:: \text { Eq } k \Rightarrow L[(k,(\text { Int, Int }))][(k, \text { Int })] \\
& \text { mapAddByKey }_{\mathrm{L}}=\text { unlift mapAddByKey } \\
& \text { mapAddByKey }_{\mathrm{F}} x s=\operatorname{mapByKey}_{\mathrm{F}}(0,0)(\text { lift addL }) x s \\
& \text { mapByKey }_{\mathrm{F}} d=\text { liftC }\left(\text { mapByKey }_{\mathrm{C}} d\right)
\end{aligned}
$$

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

```
Main> put mapAddByKey 
[("B", (2,3)),("A", (1, 2))]
Main> put mapAddByKey }\mp@subsup{\textrm{L}}{}{s}\mathrm{ [("A",3)]
[("A", (1, 2))]
Main> put mapAddByKey }\mp@subsup{)}{\textrm{L}}{}\mathrm{ [("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. ${ }^{4}$

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

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 $b a d_{\mathrm{L}}=$ unlift2 (lift2 $i d_{\mathrm{L}} \circ$ bad) would violate well-behavedness, as put bad $\mathrm{L}_{\mathrm{L}}(0,2)(1,2)=$ $(1,2)$ but get $^{\text {bad }} \mathrm{L}(1,2)=(1,1)$.

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.

```
newtype R s a = R(Poset s=>s->(a,s->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) $p s=T r u e$; (2) $p s^{\prime}=$ True implies $m s=m s^{\prime}$ for any $s^{\prime}$. 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 ( $\gg$ ).

$$
\begin{array}{r}
R m \gg f=R \$ \lambda s \rightarrow \operatorname{let}\left(x, c_{1}\right)=m s \\
\left(y, c_{2}\right)=\operatorname{let} R k=f x \text { in } k s \\
\operatorname{in}\left(y, \lambda s \rightarrow c_{1} s \wedge c_{2} s\right)
\end{array}
$$

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

```
observe :: \(E q w \Rightarrow L^{\mathrm{T}} \mathrm{s} w \rightarrow R s w\)
observe \(\ell=R(\lambda s \rightarrow\) let \(w=\) get \(\ell s\)
        in \(\left(w, \lambda s^{\prime} \rightarrow\right.\) get \(\left.\left.\ell s^{\prime}==w\right)\right)\)
unliftM2 \(::(E q a, E q b) \Rightarrow\)
    \(\left(\forall s .\left(L^{\mathrm{T}} s a, L^{\mathrm{T}} s b\right) \rightarrow R s\left(L^{\mathrm{T}} s c\right)\right)\)
    \(\rightarrow L(a, b) c\)
unliftM2 \(f=L(\lambda s \rightarrow\) get \((m k L e n s f s) s)\)
                \((\lambda s \rightarrow\) put \((m k L e n s f s) s)\)
where
    \(m k L e n s f s=\)
        let \((\ell, p)=\operatorname{let} R m=f\left(f s t_{\mathrm{L}}^{\prime}, s n d_{\mathrm{L}}^{\prime}\right)\)
                in \(m\left(\right.\) get \(\left.\operatorname{tag}_{2}{ }_{\mathrm{L}} s\right)\)
            \(\ell^{\prime}=\ell \hat{o} \operatorname{tag}_{\mathrm{L}} \mathrm{L}^{2}\)
            put's \(v=\operatorname{let} s^{\prime}=p u t \ell^{\prime} s v\)
                        in if \(p\left(\right.\) get \(\left.\operatorname{tag}^{2}{ }_{\mathrm{L}} s^{\prime}\right)\) then \(s^{\prime}\) else \(\perp\)
        in \(L\left(\right.\) get \(\left.\ell^{\prime}\right) p u t^{\prime}\)
```

Although we define the get and put components of the resulting lens separately in unliftM2, well-behavedness is guaranteed as long as $R$ and $L^{\mathrm{T}}$ 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.

$$
\begin{aligned}
& \operatorname{good}(x, y)=\text { fmap }\left(\text { lift2 } i d_{\mathrm{L}}\right) \$ \text { do } \\
& b \leftarrow \operatorname{liftO2}(==) x(\text { new } 0) \\
& \text { return }(\text { if } b \text { then }(x, y) \text { else }(x, \text { new } 1))
\end{aligned}
$$

Here, liftO2 is defined as follows.

[^3]```
liftO2 :: Eq w \(\Rightarrow\)
\[
(a \rightarrow b \rightarrow w) \rightarrow L^{\mathrm{T}} s a \rightarrow L^{\mathrm{T}} s b \rightarrow R s w
\]
liftO2 \(p x y=\) liftO \((\) uncurry \(p)(x \circledast y)\)
liftO \(:: E q w \Rightarrow(a \rightarrow w) \rightarrow L^{\mathrm{T}}\) s \(a \rightarrow R s w\)
liftO \(p x=\) observe \((\) lift \((L p\) unused \() x)\)
where unused \(s v \mid v==p s=s\)
```

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

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.

$$
\begin{aligned}
& \text { bff }::(\forall a .[a] \rightarrow[a]) \rightarrow(E q a \Rightarrow L[a][a]) \\
& \text { bff } f=\text { unlift }_{\text {list }}\left(\text { lsequence }_{\text {list }} \circ f\right)
\end{aligned}
$$

Replacing unlift $_{\text {list }}$ and lsequence $_{\text {list }}$ 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.

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.

$$
\forall f . \text { Functor } f \Rightarrow(B \rightarrow f B) \rightarrow(A \rightarrow f A)
$$

Intuitively, we can read $A \rightarrow f A$ 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 $(V, V \rightarrow$ $(-))$. Note that the lens type $L S V$ is isomorphic to the type $S \rightarrow(V, V \rightarrow S)$.

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 $\ell:: A \rightarrow B$, 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 $\rightarrow$. 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 $\mathrm{L}_{\mathrm{L}}$. 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 $\forall v . L B v \rightarrow L A v$. It is worth mentioning that $L(-) \quad V$ 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):381416, 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 secondorder 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, $I C F P$, 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 164173. 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.


[^0]:    ${ }^{1}$ 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.

[^1]:    ${ }^{2}$ In GHC, the function contents is called toList, which is defined in Data.Foldable (Every Traversable instance is also an instance of Foldable). We use the name contents to emphasize the function's role of extracting contents from structures [3].

[^2]:    ${ }^{3}$ This definition actually overlaps with that for pairs. So we either need to have "wrapper" type constructors, or enable OverlappingInstances.

[^3]:    ${ }^{4}$ 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.

