Citation for published version


DOI

Link to record in KAR

http://kar.kent.ac.uk/49084/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
Applicative Bidirectional Programming with Lenses

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

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

Abstract

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

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

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

General Terms Languages

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

1. Introduction

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

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

Using terminologies originated from the lens framework [4, 9, 10], bidirectional transformations, coined lenses, can be represented as pairs of functions known as get of type \( S \rightarrow V \) 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

\[
\text{data } L \ s \ v = L \{ \text{get} :: s \rightarrow v, \text{put} :: s \rightarrow v \rightarrow s \} \\
\text{fst}_L :: L \ (a, b) \ a \\
\text{fst}_L = L \ (\lambda (a, \_) \rightarrow a) \ (\lambda (_, b) \ a \rightarrow (a, b))
\]

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

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

\[
\text{put } \ell \ s \ (\text{get } \ell \ s) = s \quad (\text{Acceptability}) \\
\text{get } \ell' \ s' = v \text{ if } \text{put } \ell \ s \ v = s' \quad (\text{Consistency})
\]

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

By dint of hard effort, one can construct separately the forward transformation \( \text{get} \) and the corresponding backward transformation \( \text{put} \). 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 bidirectionalizations is guaranteed by construction.
- As a bidirectionalization system, it allows users to write bidirectional transformations almost in the same way as that of gets, in an applicative and higher-order programming style.

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

\[
\forall s. L^s s A_1 \to \cdots \to L^s s A_n \rightarrow \cdots \rightarrow L^s s B
\]

where \( L^s \) 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 transformations. 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.

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 bidirectionalizations is guaranteed by construction.
- As a bidirectionalization system, it allows users to write bidirectional transformations almost in the same way as that of gets, in an applicative and higher-order programming style.

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

\[
\forall s. L^s s A_1 \to \cdots \to L^s s A_n \rightarrow \cdots \rightarrow L^s s B
\]

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

\[
\begin{align*}
\text{unlines} & : [\text{String}] \rightarrow \text{String} \\
\text{unlines} [ ] & = "" \\
\text{unlines} (x : xs) & = x + \text{"}\text{\n}\text{"} + \text{unlines} xs
\end{align*}
\]

as below.

\[
\begin{align*}
\text{unlines}_F & : [L^s s \text{String}] \rightarrow L^s s \text{String} \\
\text{unlines}_F [ ] & = \text{new }"" \\
\text{unlines}_F (x : xs) & = \text{lft2 catLine}_L (x, \text{unlines}_F xs)
\end{align*}
\]

where \( \text{catLine}_L \) is a lens version of \( \text{catLine} y \rightarrow x + \text{"}\text{\n}\text{"} + y \). In the above, except for the noise of \( \text{new} \) and \( \text{lft2} \), the definition is faithful to the original structure of \( \text{unlines} \)'s definition, in an applicable style. With the heavy-lifting done in defining the lens function \( \text{unlines}_F \), a corresponding lens \( \text{unlines}_L : L[\text{String}] \text{String} \) is readily available through straightforward unlifting: \( \text{unlines}_L = \text{unlift}_L \text{unlines}_F \).

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 \( \text{puts} \) are total and no data structure contains \( \perp \). To deal with the partiality of \( \text{puts} \), we assume that a \( \text{put} \) function of type \( A \rightarrow B \rightarrow A \) can be represented as a total function of type \( A \rightarrow B \rightarrow \text{Maybe} A \), which upon termination will produce either a value \( \text{Just} \ a \) or an error \( \text{Nothing} \).

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, 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 \( \circ \)

\[
(\hat{\circ}) : L b c \rightarrow L a b \rightarrow L a c
\]

\[
(L \text{get}_2 \circ \text{put}_2) \hat{\circ} (L \text{get}_1 \circ \text{put}_1) =
L (\text{get}_2 \circ \text{get}_1) (\lambda s \rightarrow \text{put}_1 s (\text{put}_2 (\text{get}_1 s) v))
\]

we can compose \( \text{fst}_L \) to itself to obtain a lens that operates on nested pairs, as below.

\[
\begin{align*}
\text{fstTri}_L & : L ((a, b), c) a \\
\text{fstTri}_L & = \text{fst}_L \circ \text{fst}_L
\end{align*}
\]

Well-behavedness is preserved by such compositions: \( \text{fstTri}_L \) is well-behaved by construction assuming well-behaved \( \text{fst}_L \).

The composition operator \( \circ \) has the identity lens \( \text{id}_L \) as its unit.

\[
\begin{align*}
\text{id}_L & : L a a \\
\text{id}_L & = L \text{id} (\lambda v \rightarrow v)
\end{align*}
\]

#### 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
\[ \text{lift} :: L \ a \ b \to (\forall s. L \ s \ a \to L \ s \ b) \]
\[ \text{lift} \ \ell = \lambda x \to \ell \circ x \]

Intuitively, a lens of type \( L \ s \ A \) with the universally quantified type variable \( s \) can be seen as an updatable datum of type \( A \), and a lens of type \( L \ A \ B \) as a transformation of type \( \forall s. L \ s \ A \to L \ s \ B \) on updatable data. We call such lifted lenses \textit{lens functions}.

The lifting function \textit{lift} is injective, and has the following left inverse.

\[ \text{unlift} :: (\forall s. L \ s \ a \to L \ s \ b) \to L \ a \ b \]
\[ \text{unlift} \ f = f \ . \text{id}_a \]

Since lens functions are normal functions, they can be composed and passed to higher-order functions in the usual way. For example, \( \text{fstTri}_L \) can now be defined with the usual function composition.

\[ \text{fstTri}_L :: ((a, b), c) \to L \ a \ c \]
\[ \text{fstTri}_L = \text{unlift} \ (\text{lift} \ \text{fst}_L \circ \text{lift} \ \text{fst}_L) \]

Alternatively in a more applicable style, we can use a higher-order function \( \text{twice} :: (\forall x. a \to a) \to a \to a \) as below.

\[ \text{fstTri}_L = \text{unlift} \ (\lambda x \to \text{twice} \ (\text{lift} \ \text{fst}_L) \ x) \]

where \( \text{twice} \ f \ x = f \ (f \ x) \)

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

### 2.2 Formal Properties of Lens Functions

We reconfirm that \textit{lift} is injective with \textit{unlift} as its left inverse.

**Proposition 1.** \( \text{unlift} \ (\text{lift} \ \ell) = \ell \) for all lenses \( \ell :: L \ A \ B \).

We say that a function \( f \) \textit{preserves well-behavedness}, if \( f \) \( \ell \) is well-behaved for any well-behaved lens \( \ell \). Functions \textit{lift} and \textit{unlift} have the following desirable properties.

**Proposition 2.** \( \text{lift} \ \ell \) \textit{is well-behaved}.

**Proposition 3.** \( \text{unlift} \ f \) \textit{is well-behaved}.

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

\[ f :: \text{Eq} \ a \Rightarrow L \ s \ (\text{Maybe} \ a) \to L \ s \ (\text{Maybe} \ a) \]
\[ f \ (L \ g \ p) = L \ g \ (\lambda x \to \text{if} \ v = g \ s \text{ then } s \text{ else } p \ (p \ s \text{ Nothing} \ v)) \]

Here the input lens is pattern matched and the \textit{get}/\textit{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 \( \forall s. L \ s \ A \to L \ s \ B \). 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 \textit{abstract} in \( f :: \tau \) if there is a polymorphic function \( h \) of type

\[ \forall \ell. (\forall a. L \ a \ b \to (\forall s. \ell \ s a \to \ell \ s b)) \to (\forall a. (\forall s. \ell \ s a \to \ell \ s b) \to L \ a \ b) \to \tau' \]

where \( \tau' = \tau[\ell/\ell] \) and \( f = \text{h lift unlift} \).

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 \textit{lift}) provides the means to create \( L \) values.

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

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

\[ \text{module} \ \text{AbstractLens} \ (L\text{abs}, \text{lift}\text{abs}, \text{unlift}\text{abs}) \text{ where} \]
\[ \text{newtype} L\text{abs} \ a \ b = L\text{abs} \ (\text{unlift}\text{abs} :: L \ a \ b) \]
\[ \text{lift}\text{abs} :: L \ a \ b \to (\forall s. L\text{abs} \ s \ a \to L\text{abs} \ s \ b) \]
\[ \text{unlift}\text{abs} :: (\forall s. L\text{abs} \ s \ a \to L\text{abs} \ s \ b) \to L \ a \ b \]

Outside the module \textit{AbstractLens}, we can use \textit{lift}\textit{abs}, \textit{unlift}\textit{abs} and type \textit{Labs} itself, but not the constructor of \( L\text{abs} \). Thus the only way to access data of type \( L \) is through \textit{lift}\textit{abs} and \textit{unlift}\textit{abs}.

A consequence of having abstract \( L \) is that \textit{lift} is now surjective (and \textit{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 \( \forall s. L \ s \ A \to L \ s \ B \) in which \( L \) is abstract. Then \( f \ \ell = f \ . \text{id}_L \circ \ell \) holds for all \( \ell :: L \ S \ A \).

Correspondingly, we also have that \textit{unlift} is injective on lens functions.

**Theorem 2.** For any \( f :: \forall s. L \ s \ A \to L \ s \ B \) in which \( L \) is abstract, \( \text{lift} \ (\text{unlift} \ f) = f \ . \text{holds} \).

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

#### 2.4 Categorical Notes

As mentioned earlier, our idea of mapping \( L \ A \ B \) to \( \forall s. L \ s \ A \to L \ s \ B \) 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 \( (\text{lift}, \text{unlift}) \) is a bijection between

- \( \{ :: L \ A \ B \} \), and
- \( \{ :: \forall s. L \ s \ A \to L \ s \ B \mid f \ . \text{x o y} = f \ . (f \ . \text{o y}) \}. \)

The condition \( f \ . \text{x o y} = f \ . (f \ . \text{o y}) \) 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 \( \text{L Y X} \) to a function \( (\lambda y \to \ell \ . \text{o y}) \) of type \( \text{L X} \to \text{L Y} \). Note that \( f \ . \text{x o y} = f \ . (f \ . \text{o y}) \) is equivalent to \( f \ . x \) = \( f \ . \text{id}_L \circ x \). 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 $X \times B \times C$ such that there is a bijection between $L (A, B \times C)$ and $L (X \times B \times C)$, which can be easily checked by comparing cardinalities. Our discussion does not conflict with this fact. What we state is that, for any $s$, $(L s a, L s B) \rightarrow L s C$ 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 function that maps a lens $\ell$ to a function $\ell$. It is not difficult to check that $\ell x \circ \ell y = \ell (x \circ y)$ and $\ell (\text{id}_t :: L A A) = (\text{id} :: L s A \rightarrow L s A)$.

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 $\text{fstTri}_L$ from Section 2 reproduced below.

$$
\text{fstTri}_L :: L ((a, b), c) a
\text{fstTri}_L = \text{unlift} (\text{lift} \text{fst}_L \circ \text{lift} \text{fst}_L)
$$

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 $\text{fstTri}_L$, which consists of the composition of lifted $\text{fst}_L$s. But more fundamentally it is the type of $\ell (L x y \rightarrow (\forall s. L s x \rightarrow L s y))$, 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.

$$
\text{swap}_L :: L (a, b) (b, a)
\text{swap}_L \ldots
$$

Following the programming pattern developed so far, we would like to construct this lens with the familiar unidiirectional function $\text{swap}: (a, b) \rightarrow (b, a)$. 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 $\text{swap}$ with the resulting lens function.

In order to construct $\text{swap}_L$ and many other lenses, including $\text{unlines}_1$ in Section 1, a conversion of values of type $\forall s. L s A_1, \ldots, L s A_n$ to values of type $\forall s. L s (A_1, \ldots, A_n)$ is needed. In this section we look at how such a conversion can be defined for binary lenses, which can be easily extended to arbitrary $n$-ary cases.

3.1 Caveats of the Duplication Lens

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

$$
\text{dup}_L :: Eq s \Rightarrow L s (s, s)
\text{dup}_L = L (\lambda s \rightarrow (s, s)) (\lambda_\_ (s, t) \rightarrow r s t)
\text{where } r \text{ s } t \ni s = t \Rightarrow \_ \_ This \ will \ cause \ a \ problem.
$$

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

$$
\text{(\otimes)} :: Eq s \Rightarrow L s a \rightarrow L s b \rightarrow L s (a, b)
\varepsilon \otimes \varepsilon = (\varepsilon \otimes \varepsilon) \circ \text{dup}_L
$$

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

$$
\text{\text{(\otimes)}} :: L a a' \rightarrow L b b' \rightarrow L (a, b) (a', b')
(L \text{get}_1 \text{put}_1) \otimes (L \text{get}_2 \text{put}_2) =
L (\lambda (a, b) \rightarrow (\text{get}_1 a, \text{get}_2 b))
(\lambda (a', b') \rightarrow (\text{put}_1 a\', \text{put}_2 b'))
$$

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

$$
\text{lift2} :: L (a, b) c \rightarrow (\forall s. L s a, L s b) \rightarrow L s c
\text{lift2} \ell (x, y) = \ell (x \otimes y)
$$

It is tempting to have the following as the inverse for $\text{lift2}$.

$$
\text{unlift2} :: (\forall s. L s a, L s b) \rightarrow L s c \rightarrow L (a, b) c
\text{unlift2} f = f (\text{fst}_L, \text{snd}_L)
$$

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

$$
(\text{unlift2} \circ \text{lift2}) \ell
= [\text{definition unfolding} & \beta\text{-reduction}]
\ell \circ (\text{fst}_L \otimes \text{snd}_L)
= [\text{unfolding} (\otimes)]
(\ell \otimes \text{block}_L, \text{where})
$$

The last call to put $\text{dup}_L$ above will fail because $\exists (2, 3) \neq (1, 4)$. But if we look more carefully, there is no reason for this behavior: $(\text{lift2} \ od) \text{dup}_L$ 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 $\text{dup}_L$ and update the old source $(1, 2)$ to $(3, 4)$ without violating bidirectional laws.

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

$$
data \text{Tag a} = U \{ \text{unTag : a} \} | O \{ \text{unTag : a} \}
data \text{Tag} U (\text{representing Updated}) \text{means the tagged value may be relevant to the view update and O (representing Original) means the tagged value must not be relevant to the view update. The idea is that O-tagged values can be altered without violating the bidirectional laws, as the new dup}_L \text{below.}
$$

$$
\text{dup}_L :: \text{Poset} s \Rightarrow L s (s, s)
\text{dup}_L = L (\lambda s \rightarrow (s, s)) (\lambda_\_ (s, t) \rightarrow s \_ t)
$$

Here, $\text{Poset}$ is a type class for partially-ordered sets that has a method $(\gamma)$ (pronounced as “lub”) to compute least upper bounds.

$$
\text{class Poset s where } (\gamma) :: s \rightarrow s \rightarrow s
$$

We require that $(\gamma)$ must be associative, commutative and idempotent; but unlike a semilattice, $(\gamma)$ can be partial. Tagged elements and their (nested) pairs are ordered as follows.
We also introduce the following type synonym for brevity.\footnote{Actually, we will have to use \texttt{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.}

\textbf{instance} \hspace{1mm} \texttt{Eq a ⇒ Poset \{Tag a\}} \hspace{1mm} \texttt{where}

\begin{align*}
(O \ s) \ γ \ (U \ t) &= U \ 1 \\
(U \ s) \ γ \ (O \ t) &= U \ s \\
(O \ s) \ γ \ (O \ t) \mid s \ ι t &= O \ s \\
(U \ s) \ γ \ (U \ t) \mid s \ ι t &= U \ s
\end{align*}

\textbf{instance} \hspace{1mm} \texttt{(Poset a, Poset b) ⇒ Poset \{a, b\}} \hspace{1mm} \texttt{where}

\begin{align*}
(a, b) \ γ \ (a', b') &= (a \ γ \ a', b \ γ b')
\end{align*}

We prove the statement \texttt{(*)} by showing \texttt{get \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(a, b) = (a, b)} \hspace{1mm} \text{and} \hspace{1mm} \texttt{put \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(a, b) = (a', b')}. Since the former property is easy to prove, we only show the latter here.

\begin{align*}
\texttt{put \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(a, b) = (a', b')} \\
= \{ \text{definition unfolding & \ β-reduction} \}
\end{align*}

\textbf{Proposition 5.} \hspace{1mm} \texttt{fst'_{\ L} = \{definition unfolding & \ β-reduction\}} \hspace{1mm} \texttt{\texttt{snd'_{\ L} = \{definition unfolding & \ β-reduction\}}}

\noindent Note that now \texttt{unlift} and \texttt{unlift2} are no longer injective (even with abstract \texttt{L}); there exist functions that are not equivalent but coincide after unlifting. An example of such is the pair \texttt{lift2 \ fst'_{\ L}} and \texttt{fst\Lift\Lift}; while unlifting both functions result in \texttt{fst'_{\ L}}, they actually differ as \texttt{put \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(O a, O b) c = (U c, U b)} \hspace{1mm} \text{and} \hspace{1mm} \texttt{put \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(O a, O b) c = (U c, O b)}. Intuitively, \texttt{fst} knows that the second argument is unused, while \texttt{lift2 \ fst'_{\ L}} does not because \texttt{fst'_{\ L}} is treated as a black box by \texttt{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 \texttt{lift2} and \texttt{unlift2} by the new internal functions \texttt{fst'_{\ L}}, \texttt{snd'_{\ L}}, \texttt{dup'_{\ L}}, and \texttt{tag2_{\ 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 \texttt{dup'_{\ L}}, \texttt{fst'_{\ L}}, \texttt{snd'_{\ L}} and \texttt{tag2_{\ L}} are not well-behaved, for different reasons. For functions \texttt{fst'_{\ L}} and \texttt{snd'_{\ L}}, the difference from the original versions \texttt{fst\Lift\Lift} and \texttt{snd\Lift\Lift} 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 \texttt{dup'_{\ L}} and \texttt{tag2_{\ L}}, the new definitions are more clearly defined in the sense that some originally failing executions of \texttt{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 \texttt{unlift2 \ f} to be well-behaved for any \texttt{f} \hspace{1mm} \text{as} \hspace{1mm} \texttt{\{definition unfolding & \ β-reduction\}} \hspace{1mm} \text{applied to functions calling \texttt{lift2} internally. In what follows, we only focus on \texttt{lift2} and \texttt{unlift2}, and expect the discussion straightforwardly extends to \texttt{lift} and the new \texttt{unlift}.}

Finally, we can now show that the new \texttt{unlift2} is the left-inverse of \texttt{lift2}.

\textbf{Proposition 4.} \hspace{1mm} \texttt{unlift2 \ (lift2 \ ℓ) = ℓ} \hspace{1mm} \text{holds for all lenses} \hspace{1mm} \texttt{ℓ \:: L \ (A, B)} \hspace{1mm} \text{C}.

\textbf{Proof.} We prove the statement with the following calculation.

\begin{align*}
\texttt{unlift2 \ (lift2 \ ℓ) & = \{ definition unfolding & \ β-reduction \}}
\end{align*}

\begin{align*}
\ell \ ◦ \ \texttt{fst'_{\ L}} \ ◦ \ \texttt{snd'_{\ L}} \ ◦ \ \texttt{tag2_{\ L}}
\end{align*}

\begin{align*}
\texttt{ℓ \ ◦ \ \texttt{fst'_{\ L}} \ ◦ \ \texttt{snd'_{\ L}} \ ◦ \ \texttt{dup'_{\ L}} \ ◦ \ \texttt{tag2_{\ L}}}
\end{align*}

\begin{align*}
\texttt{ℓ \ ◦ \ \texttt{fst'_{\ L}} \ ◦ \ \texttt{snd'_{\ L}} \ ◦ \ \texttt{dup'_{\ L}} \ ◦ \ \texttt{tag2_{\ L}} = id_{\ L} \hspace{1mm} \\texttt{(*)}}
\end{align*}

\texttt{ℓ}

We prove the statement \texttt{(*)} by showing \texttt{get \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(a, b) = (a, b)} \hspace{1mm} \text{and} \hspace{1mm} \texttt{put \{(fst'_{\ L} \ ◦ \ snd'_{\ L}) \ ◦ \ dup'_{\ L} \ ◦ \ tag2_{\ L}\}} \hspace{1mm} \texttt{(a, b) = (a', b')}.
irrelevant to the update and can be discarded). We formalize such a property with the partial order between tagged values. Let us write \( (\leq) \) for the partial order induced from \( Y \); that is, \( s \leq t \) if \( s \Downarrow t \) is defined and equal to \( t \). One can see that \( (\leq) \) is the reflexive closure of \( O \leq U \). We write \( \uparrow s \) for a value obtained from \( s \) by replacing all \( O \)-tags with \( U \)-tags. Trivially, we have \( s \leq \uparrow s \). But there exists \( s' \) such that \( s \leq s' \) and \( s' \neq \uparrow s \).

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

**Definition 2** (Local Well-Behavedness). A bidirectional transformation \( \ell : L^1 \rightarrow b \) is called locally well-behaved if the following four conditions hold.

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

In the above, tags introduced for the flexible behavior of \( \ell \) must not affect the behavior of \( g: \uparrow s' = \uparrow s \) means that \( s \) and \( s' \) are equal if tags are ignored. The property local-acceptability is similar to acceptability, except that \( \ell \)-tags are allowed to change to \( U \)-tags. The property local consistency is stronger than consistency in the sense that \( \ell \) must map all values sharing the same \( U \)-tagged elements with \( s' \) to the same view. The idea is that \( \ell \)-tagged elements in \( s' \) are not connected to the view \( v \), and thus changing them will not affect \( v \). A similar reasoning applies to backward inflation stating that source elements changed by \( \ell \) will have \( \ell \)-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 \( \ell s \) and \( \ell s \) introduce tagged views; but they are always precomposed when used, as shown in the following.

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

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

- If \( x \) is well-behaved and \( y \) is locally well-behaved, then \( x \pi y \) is locally well-behaved.
- If \( x \) and \( y \) are locally well-behaved, \( x \pi y \) is locally well-behaved.
- If \( x \) and \( y \) are locally well-behaved, \( x \pi y \) is locally well-behaved.

**Proof.** We only prove the second property, which is the most non-trivial 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.

First, show local acceptability.

\[
\begin{align*}
\text{put } (x \pi y) \circ \text{dup}_L & \text{ s (get } (x \pi y) \circ \text{dup}_L \text{ s) } & = \text{[simplification]} \\
\text{put } \text{dup}_L & \text{ s (put } x \pi y \text{ s, s) (get } (x \pi y) \text{ s) } & = \text{[local acceptability]} \\
\text{put } \text{dup}_L & \text{ s (s, s') } & = \text{[definition of } \text{dup}_L \text{ and that } s' \preceq s \text{ is defined]} \\
\end{align*}
\]

Then, we prove local consistency. Assume that \( \text{put } (x \pi y) \circ \text{dup}_L \text{ s (v_1, v_2) succeeds in } s' \). Then, by the following calculation, we have \( s' = \text{put } x \pi v_1 \uparrow y \text{ put } y \pi v_2 \).

\[
\begin{align*}
\text{put } ((x \pi y) \circ \text{dup}_L) & \text{ s (v_1, v_2) } & = \text{[simplification]} \\
\text{put } \text{dup}_L & \text{ s (put } x \pi v_1, y \pi v_2 \text{) } & = \text{[definition unfolding]} \\
\text{put } x \pi v_1 & \uparrow y \text{ put } y \pi v_2 \text{ \[\]}
\end{align*}
\]

Let \( s'' \) be a source such that \( s' \preceq s'' \). Then, we prove \( ((x \pi y) \circ \text{dup}_L) s'' = (v_1, v_2) \) as follows.

\[
\begin{align*}
\text{get } ((x \pi y) \circ \text{dup}_L) s'' & \text{ (v_1, v_2) } & = \text{[simplification]} \\
\text{get } x \pi s'' & \text{ get } y \pi s'' & = \text{[the local consistency of } x \text{ and } y \text{]}
\end{align*}
\]

Note that we have \( \text{put } x \pi v_1 \preceq s' \preceq s'' \) and \( \text{put } y \pi v_2 \preceq s' \preceq s'' \) by the definition of \( \gamma \).

Forward tag-irrelevance and backward inflation are straightforward.

**Corollary 1.** The following properties hold.

- \( \ell \ell L^1 \rightarrow L^1 \rightarrow A B \) preserves local well-behavedness, if \( \ell : L^1 \rightarrow A B \) is well-behaved.
- \( \ell f L^1 \rightarrow L^1 \rightarrow L^1 \rightarrow C \) preserves local well-behavedness, if \( \ell : L^1 \rightarrow (A, B) \rightarrow C \) is well-behaved.

Similar to the case in Section 2, compositional reasoning of well-behavedness requires the lens type \( L^1 \) to be abstract.

**Definition 3** (Abstract Nature of \( L^1 \)). We say \( L^1 \) is abstract in \( f :: \tau \) if there is a polymorphic function \( h \) of type

\[
\forall (\forall a b. L^1 a b \rightarrow (\forall a. L^1 a b \rightarrow L^1 a b)) \rightarrow (\forall a b. L^1 a b \rightarrow L^1 a b)
\]

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

**Theorem 4.** Let \( f \) be a function of type \( \forall (\forall a b. L^1 a b \rightarrow L^1 a b) \rightarrow L^1 a b \). Then, \( f (x, y) \) is locally well-behaved if \( x \) and \( y \) are also locally well-behaved, assuming that lift is applied only to well-behaved lenses.

**Corollary 2.** Let \( f \) be a function of type \( \forall (\forall a b. L^1 a b \rightarrow L^1 a b) \rightarrow L^1 a b \). Then, \( f \) is well-behaved, assuming that lift is applied only to well-behaved lenses.

**Example 1** (swap). The bidirectional version of \( \text{swap} \) can be defined as follows.

\[
\begin{align*}
\text{swap}_L & \text{ := (Eq a, Eq b) } \Rightarrow (a, b) \text{ (b, a)} \\
\text{swap}_L & \text{ = } \text{swap}_L \circ (\text{id} \circ \text{swap})
\end{align*}
\]

And it behaves as expected.

\[
\begin{align*}
\text{put } \text{swap}_L & (1, 2) (4, 3) & = \text{[unfold definitions]} \\
\text{put } (\text{snd}_L \circ \text{fst}_L) & \circ \text{dup}_L \circ \text{tag}_L (1, 2) (4, 3) & = \text{[simplifications]} \\
\text{put } \text{tag}_L & (1, 2) & = \text{[definition of } \text{fst}_L \text{ and } \text{snd}_L \text{]} \\
\text{put } \text{dup}_L & (O 1, O 2) & = \text{[definition of } \text{fst}_L \text{ and } \text{snd}_L \text{]} \\
\text{put } \text{dup}_L & (O 1, O 2) = \text{[definition of } \text{fst}_L \text{ and } \text{snd}_L \text{]} \\
\text{put } \text{dup}_L & (O 1, O 2) = (O 1, U 4), (U 3, O 2))
\end{align*}
\]
\[\text{lift} \ell t = \text{lift} \ell (\text{split3 } t)\]

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 \(\text{split3}\), we have decided to nest to the left in the intermediate step. This choice is not essential.

\[\text{split3}' (x, y, z) = \text{lift} \text{flattenR}_L (x \circ (y \circ z))\]

where \(\text{flattenR}_L = L (\lambda((x, y), z) \rightarrow (x, y, z))\)

\[(\lambda_\circ (x, y, z) \rightarrow ((x, y), z))\]

The two definitions \(\text{split3}\) and \(\text{split3}'\) coincide.

To complete the picture, the nullary lens function

\[\text{unit} :: \forall s. L^s s\]

\[\text{unit} = L (\lambda_\circ \rightarrow () (\lambda s (s \rightarrow s))\]

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

\[\text{new} :: \exists s. L^s a \rightarrow \forall s. L^s s\]

\[\text{new } a = \text{lift} (L (\text{const } a) (\lambda_\circ a' \rightarrow \text{check } a a')) \text{ unit}\]

where

\[\text{check } a a' = \text{if } a :: a' \text{ then } ()\]

\[\text{else } \text{error "Update on constant"}\]

Function \(\text{new}\) 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 \(\text{lift} \ell : L S A \rightarrow L S B\) for any \(S\). In the case that \(S\) is tagged and thus partially ordered, \((L^s S (-), \circ, \text{unit})\) forms a lax monoidal functor, under the following conditions.

- \((\circ)\) must be natural, i.e., \((\text{lift } f x) \circ (\text{lift } g y) = \text{lift} (f \circ g) (x \circ y)\) for all \(f, g, x, y\) with appropriate types.
- \(\text{split3}\) and \(\text{split3}'\) coincide.
- \(\text{lift} \text{elimUnitL}_L (\text{unit}@x) = x\) must hold where \(\text{elimUnitL}_L :: L ((), a)\) 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 \(s \Rightarrow\) put \(x\) \(s \ s\ v = \text{put} x\ s\ s\ v\), 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 \(L^s S A\) by the equivalence relation \(\equiv\) defined as \(x \equiv y\) if \(\text{get } x = \text{get } y\) or \(\text{put } x\ s = \text{put } y\ s\) for all minimal \(s\). This equivalence is preserved by manipulations of \(L^s\)-data; that is, the following holds for \(x, y, z\) and \(w\) with appropriate types.

- \(x \equiv y\) implies \(\text{lift} \ell x \equiv\) \(\text{lift} \ell y\) for any well-behaved lens \(\ell\).
- \(x \equiv y\) and \(z \equiv w\) implies \(x \circ z \equiv y \circ w\).
- \(x \equiv y\) implies \(x \circ \text{tag} L = y \circ \text{tag} L\) (or \(x \circ \text{tag}2 L = y \circ \text{tag}2 L\)).

Note that the above three cases cover the only ways to construct/destruct \(L^s\) in \(L^s\) when \(L^s\) is abstract. The third condition says that this “coarse” equivalence \((\equiv)\) on \(L^s\) can be “sharpened” to the usual extensional equality \((\approx)\) by \(\text{tag}_L\) and \(\text{tag}2_L\) in the unlifting functions.

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

4. Going Generic

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

4.1 Unlifting Functions on Lists

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

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

\[\text{isquenceL} :: [L^s s a] \rightarrow L^s s a\]

\[\text{isquenceL} [] = \text{nilL}\]

\[\text{isquenceL} (x : xs) = \text{lift2 } \text{consL} (x, \text{isquenceL} xs)\]

\[\text{nilL} = L (\lambda ([]) \rightarrow []) (\lambda [] \rightarrow ())\]

\[\text{consL} = L (\lambda (a, as) \rightarrow (a : as)) (\lambda (a', as') \rightarrow (a', as'))\]

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

\[\text{liftL} :: L [a] b \rightarrow \forall s. [L^s s a] \rightarrow L^s s b\]

\[\text{liftL } \ell xs = \text{lift} \ell (\text{isquenceL} xs)\]

Tagged lists form an instance of \textit{Poset}.

\[\text{instance } \text{Poset} a \Rightarrow \text{Poset } [a] \text{ where}\]

\[\text{xs } \Rightarrow\ ys \equiv \begin{cases}\text{if } \text{length } xs = \text{length } ys \\
\text{then } \text{zipWith } (\Rightarrow) \text{ xs ys} \end{cases}\]

\[\text{else } \perp \text{ - Unreachable in our framework}\]

Note that the requirement that \(xs\) and \(ys\) must have the same shape is made explicit above, though it is automatically enforced by the abstract use of \(L^s\) in lifted functions.

The definition of \(\text{unliftL}\) 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 \(f\).
The obtained lens and catLine after appending a terminating newline to each. For example, contrast, any change on list length will be rejected; specifically lift2, lift_list is an injection with unlift_list as its left inverse.

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

tail :: [a] → [a]
tail (x : xs) = xs

A bidirectional version of tail is easily constructed by using sequence_list and unlift_list as follows.

tailL :: Eq a ⇒ L [a] [a]
tailL = unlift_list (sequence_list ∘ tail)

The obtained lens tailL supports all in-place updates, such as put tailL ["a", "b", "c"] ["B", "C"] = ["a", "b", "C*"]. In contrast, any change on list length will be rejected; specifically nilL or consL in sequence_list throws an error.

Example 3 (Bidirectional unlines). Let us consider a bidirectional version of unlines :: [String] → String that concatenate lines, after appending a terminating newline to each. For example, unlines ["a", "b", "c"] = "ab\nbc\n\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.

catLineL :: L (String, String) String
catLineL = L (λ(s, t) → s ++ "\n" ++ t)
(λ(s, t) → let n = length (filter (xs → \n) s)
i = elemIndices (\n) n ?!! n
(s', t') = splitAt i u
in (s', t'))

Here, elemIndices and splitAt are functions from Data.List: elemIndices c s returns the indices of the 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 catLineL (s, t) u splits u into s’ and "\n" ++ t’ so that s’ contains the same number of newlines as the original s. For example, put catLineL ("a\nbc", "de") "\nB\nC" = ("\nB", "C"). Then, construction of a bidirectional version unlinesL of unlines is straightforward; we only need to replace "" with new "" and catLine with lift2 catLineL, and to apply unlift_list to obtain a lens.

unlinesL :: ∀a b. Eq a ⇒ (L [a] → L [b]) → L [a] b
unlinesL f = L (λs → get (mkLens s) s)
(λs → put (mkLens s) s)

where

mkLens s = f (projs (length s)) ∙ tagListL

unliftListL = L (map O) (λλs → map unTag ys)
projs n = map projL [0..n - 1]
projL i = L (λxs → unTag (xs !! i))
(λas a → update i (U a) as)

As one can see, unlinesL 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 new **) with the data with the corresponding bidirectional versions (e.g., lift2 catLineL and new **).

Lens unlinesL accepts updates that do not change the original formatting of the view (i.e., the same number of lines and an end line). For example, we have put unlinesL ["a", "b", "c"] "AA\nBB\nCC\n\n" = ["AA", "BB", "CC"], but put unlinesL ["a", "b", "c"] "AA\nBB\nCC\n\n" = ⊥ and put unlinesL ["a", "b", "c"] "AA\nBB\nCC\nD" = ⊥.

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

unlines = foldl catLine **

The same coding principle for constructing bidirectional versions applies.

unlinesL :: L [String] String
unlinesL = unlift_list unlinesL
unlinesL [] = []
unlinesL (x : xs) = foldr (lift2 catLineL) (new **) (unlinesL x) (unlinesL xs)

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

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

traverse :: (Traversable t, Applicative f)
⇒ (a → f b) → t a → f (t b)

We use traverse to define two functions that are able to extract data from the structure holding them (content), and redecorate an “empty” structures with given data (fill). ²

newtype Const a b = Const { getConst :: a }

newtype = Const a b = Const { getConst :: a }
contents :: Traversable t ⇒ t a → a

contents = getConst (traverse (λx → Const [x]) t)

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].
fill :: Traversable t ⇒ t b → [a] → t a
fill t tf = evalState (traverse next t) tf
where
  next _ = do (a : x) ← 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 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 extent of being
systematically derivable [23]. The instances of Traversable must
satisfy certain laws [3]; and for such lawful instances, we have

fill (fmap f t) (contents t) = t
          (FillContents)
contents (fill t xs) = xs if length xs = length (contents t)
          (ContentsFill)

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

We can now define a generic isquence function as follows.

isquence :: (Eq a, Eq (t ()), Traversable t) ⇒
t (L' s a) → L' s (t a)
isquence t =
  lift (fill (shape t) (isquence t))
where
  fill 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 = fmap (Λ_ → ()) t. Also, we can make a
Poset instance as follows.3

instance (Poset a, Eq (t ()), Traversable t) ⇒
  Poset (t a)
where
t1 ∋ t2 = if shape t1 @= shape t2
  then fill t1 (contents t1 ∋ contents t2)
  else ⊥ -- 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) ⇒
  (ys, t (L' s a) → L' s b) → L (t a) b
unliftT f = L (λs → get (mkLens s) s)
          (λs → put (mkLens s) s)
where
  mkLens s = f (projTs (shape s)) ◦ tagT_L
  tagT_L = L (fmap O) (const $ fmap unTag)
  projTs sh =
    let n = length (contents sh)
        in fill sh [projT_L i sh | i ← [0..n - 1]]
  projT_L i sh =
    L (unTag ◦ (!!) i ◦ contents)
          (λs v → fill sh (update i (U v) (contents s)))

Here, projT_L i t is a bidirectional transformation that extracts the
ith element in t with the tag erased. Similarly to unliftT, 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.

data Exp = ENum Int | EInc Exp
          | EVar Var | EApp Exp Exp
          | EFun String Exp deriving Eq

data Val a = VNum a
          | VFun String Exp deriving Eq

data Env a = Env [(String, Val a)] deriving Eq

This definition is standard, except that the type of values is pa-
parameterized to accommodate both Val (L' s Int) and Val Int
for updatable and ordinary integers, and so does the type of en-
vironments. 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 :: Exp (L' s Int) → Exp → Val (L' s Int)
eval env (ENum n) = VNum (new n)
eval env (EInc e) = let VNum v = eval env e
                   in VNum (lift incl v)
eval env (EVar x) = lkup x env
eval env (EApp e1 e2) = let VFun x e' (Env env') =
                         eval env e1
                         v2 = eval env e2
                         in eval (Env ((x, v2) : env')) e'
eval env (EFun x e) = VFun x e env

Here, incl :: L Int Int is a bidirectional version of (+1) that can
be defined as follows.

incL = L (+1) (λ_ x → x - 1)

and lkup :: String → Env a → a is a lookup function.
A lens evalL :: Exp → L (Env Int) (Val Int) naturally arises
from eval.

evalL :: Exp → L (Env Int) (Val Int)
evalL e = unliftT (λenv → liftT id $ eval env e)

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 5 twice 2 twice 0 twice 0 twice 0 inc 0 x
where
twice = EFun "f" ⨪ EFun "x" ⨪
EVar "f" ⨪ (EVar "f" ⨪ EVar "x")
x = EVar "x"
inc = EFun "x" ⨪ EInc (EVar "x")
An advantage of the original lens combinators [9] (that operate
env
T
o draw an analogy to parametric higher-order abstract syn-
views. W e argue that our framework is general enough to easily
some plumbing to be able to handle lens combinators, which are
incorporate this behavior into our framework. For example, we can
simply functions over lenses. For example, for combinators of type
map
inc
6. Extensions
In this section, we extend our framework in two dimensions: al-
owing shape changes via lifting lens combinators, and allowing
(L\textquotesingle s \ A)-values to be inspected during forward transformations
following our previous work [21, 22].

6.1 Lifting Lens-Combinators
An advantage of the original lens combinators [9] (that operate
directly on the non-functional representation of lenses) over what
we have presented so far is the ability to accept shape changes to
views. W e 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{align*}
\text{lifC} :: & \text{Eq } a \Rightarrow (L a \rightarrow L c) \rightarrow \\
& \forall s, L^t s a \rightarrow L^t s b) \rightarrow \forall t, L^t t c \rightarrow L^t t d) \\
\text{lifC } c f &= \text{lif } (c \text{ (unlif } f))
\end{align*}
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
\text{mapDefault}C.
mapDefaultC :: a \rightarrow L a \rightarrow L [a] [b]
where go ss [] = []
go [] (v : vs) = put \ell d v : go [] vs
\begin{align*}
go (s : ss) (v : vs) &= \text{put } \ell s v : go ss vs
\end{align*}
When given a lens on objects, \text{mapDefault}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 \text{mapDefault}C as the following, which in the forward direction
is essentially \(\text{map (uncurry (+))}\).
mapAddL :: L [(Int, Int)] [Int]
mapAddL = \text{unlif} \text{mapAddV}
mapAddV xs = mapV (0, 0) (\ell \text{ addL} l) xs
mapV d = \text{lifC} (\text{mapDefault}C d)
addL = L (\lambda (x, y) \rightarrow x + y) (\lambda (x, \_ v) \rightarrow (x, v - x))
This lens \text{mapAddL} constructed in our framework handles shape
changes without any trouble.
Main> \text{put mapAddL} [(1, 1), (2, 2)] [3, 5]
Main> \text{put mapAddL} [(1, 1), (2, 2)] [3]
Main> \text{put mapAddL} [(1, 1), (2, 2)] [3, 5, 7]
Main> \text{put mapAddL} [(1, 2), (2, 3), (0, 7)]

The trick is that the expression \(\text{mapV} (0, 0) (\ell \text{ addL} l)\)
has type \(\forall s, L^s s [\text{Int, Int}] \rightarrow L^s s [\text{Int}]\), where the
list occurs inside \(L^s s\), contrasting to \(\text{map (lift addL)}\)’s type
\(\forall s, L^s s [\text{Int, Int}] \rightarrow L^s s [\text{Int}]\). Intuitively, the type construc-
tor \(L^s s\) can be seen as an updatability annotation; \(L^s s [\text{Int, Int}]\)
means that the list itself is updatable, whereas \([L^s s [\text{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
\text{map} directly. Our framework enables programmers to choose either
style, or anywhere in between freely.

This position-based approach used in \text{mapDefault}C is not the only
way to resolve shape dependencies. We can also match elements according to keys [2, 11]. As an example, let us consider a
variant of the map combinator.
mapByKeyC :: Eq k \Rightarrow (\lambda s \rightarrow (k, get s)) (\lambda s \rightarrow go s v)
where go ss [] = []
go ss (k : v : vs) =
\begin{align*}
case \text{lookup k ss of}
& \text{Nothing } \rightarrow (k, put \ell d v : go ss vs)
& \text{Just } s \rightarrow (k, put \ell s v : go (del k ss) vs)
& \text{del k []} \equiv []
& \text{del k (k' : ss) | } k \equiv k' \equiv ss
& \text{otherwise } = (k', s) : \text{del ss}
\end{align*}
Lenses constructed with \text{mapByKeyC} match with keys instead
of positions.
mapAddByKeyC :: Eq k \Rightarrow L [(k, \text{Int, Int})] [(k, \text{Int})]
mapAddByKeyC d \ell = L (\lambda (k, s) \rightarrow (k, get s \ell)) (\lambda s v \rightarrow go s v)
\begin{align*}
\text{where go ss []} &= []
go ss (k : v : vs) &= \text{put } \ell d v : go [] vs
\end{align*}
\begin{align*}
go (s : ss) (v : vs) &= \text{put } \ell s v : go ss vs
\end{align*}
mapAddV xs = mapV (0, 0) (\ell \text{ addL} l) xs
mapV d = \text{lifC} (\text{mapDefault}C d)
addL = L (\lambda (x, y) \rightarrow x + y) (\lambda (x, _ v) \rightarrow (x, v - x))
This lens \text{mapAddL} constructed in our framework handles shape
changes without any trouble.
Main> \text{put mapAddL} [(1, 1), (2, 2)] [3, 5]
Main> \text{put mapAddL} [(1, 1), (2, 2)] [3]
Main> \text{put mapAddL} [(1, 1), (2, 2)] [3, 5, 7]
Main> \text{put mapAddL} [(1, 2), (2, 3), (0, 7)]

The trick is that the expression \(\text{mapV} (0, 0) (\ell \text{ addL} l)\)
has type \(\forall s, L^s s [\text{Int, Int}] \rightarrow L^s s [\text{Int}]\), where the
list occurs inside \(L^s s\), contrasting to \(\text{map (lift addL)}\)’s type
\(\forall s, L^s s [\text{Int, Int}] \rightarrow L^s s [\text{Int}]\). Intuitively, the type construc-
tor \(L^s s\) can be seen as an updatability annotation; \(L^s s [\text{Int, Int}]\)
means that the list itself is updatable, whereas \([L^s s [\text{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
\text{map} directly. Our framework enables programmers to choose either
style, or anywhere in between freely.

This position-based approach used in \text{mapDefault}C is not the only
way to resolve shape dependencies. We can also match elements according to keys [2, 11]. As an example, let us consider a
variant of the map combinator.
mapByKeyC :: Eq k \Rightarrow (\lambda s \rightarrow (k, get s)) (\lambda s \rightarrow go s v)
where go ss [] = []
go ss (k : v : vs) =
\begin{align*}
case \text{lookup k ss of}
& \text{Nothing } \rightarrow (k, put \ell d v : go ss vs)
& \text{Just } s \rightarrow (k, put \ell s v : go (del k ss) vs)
& \text{del k []} \equiv []
& \text{del k (k' : ss) | } k \equiv k' \equiv ss
& \text{otherwise } = (k', s) : \text{del ss}
\end{align*}
Lenses constructed with \text{mapByKeyC} match with keys instead
of positions.
mapAddByKeyC :: Eq k \Rightarrow L [(k, \text{Int, Int})] [(k, \text{Int})]
mapAddByKeyC d \ell = L (\lambda (k, s) \rightarrow (k, get s \ell)) (\lambda s v \rightarrow go s v)
\begin{align*}
\text{where go ss []} &= []
go ss (k : v : vs) &= \text{put } \ell d v : go [] vs
\end{align*}
\begin{align*}
go (s : ss) (v : vs) &= \text{put } \ell s v : go ss vs
\end{align*}
mapAddV xs = mapV (0, 0) (\ell \text{ addL} l) xs
mapV d = \text{lifC} (\text{mapDefault}C d)
addL = L (\lambda (x, y) \rightarrow x + y) (\lambda (x, _ v) \rightarrow (x, v - x))
not the only cause of a transformation being monomorphic [21, 22]. For example, let us consider the following toy program.\footnote{This code actually does not type check as \( \texttt{(x z)} \) on \( (\text{List} s \ \textbf{Int}) \)-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.}

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

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 \( \text{bad}_1 = \text{unlift2} \ (\text{lift2} \ \text{idL} \ \circ \ \text{bad}) \) would violate well-behavedness, as \( \text{put bad}_1 \ (0, 2) \ (1, 2) = (1, 2) \) but \( \text{get bad}_1 \ (1, 2) = (1, 1) \).

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 \texttt{put}. The same technique can be used in our framework, and actually in a much simpler way due to our new compositional formalization.

\[
\text{newtype} \ R \ s \ a = R \ (\text{Poset} \ s \ \Rightarrow \ a \ (s \ \rightarrow \ \text{Bool}))
\]

We can see that \( R \ A \ B \) represents \texttt{gets} with restricted source updates: taking a source \( s : A \), it returns a view of type \( B \) together with a constraint of type \( A \ \text{to} \ Bool \) 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) \( p \ s = \text{True} \); (2) \( p \ s' = \text{True} \) implies \( m \ s = m \ s' \) for any \( s' \). It is not difficult to make \( R \ s \) an instance of \texttt{Monad}—it is a composition of \texttt{Reader} and \texttt{Writer} monads. We only show the definition of \( (\_ \_ = \_ \)).

\[
R \ m \ \Rightarrow \ f = R \ \$_{} \ \lambda s \rightarrow \ (x, c_1) = m \ s \ ((y, c_2) = \text{let} \ R \ k = f \ x \ \text{in} \ k \ s \ \text{in} \ (y, \lambda s \rightarrow c_1 \ s \ \land \ c_2 \ s))
\]

Then, we define a function that produces \( R \) values, and a version of \texttt{unlifting} that enforces the observations gathered.

\[
\text{observe} :: \ (Eq \ w \Rightarrow \text{List} s \ w \rightarrow R \ s \ w) \\
\text{observe} \ \ell = R \ (\lambda s \rightarrow \text{let} \ w = \text{get} \ \ell \ s \ \text{in} \ (w, \lambda s' \rightarrow \text{get} \ \ell \ s' \ z w))
\]

\[\text{unliftM2} :: ((Eq \ a, \text{Eq} \ b) \Rightarrow (\forall s. (\text{List} s \ a, \text{List} s \ b) \rightarrow R \ s \ (\text{List} s \ c)) \rightarrow \text{List} (a, \text{b} \text{c})) \rightarrow \text{List} (\text{mkLenses} \ f \ s) \ s \ (\lambda a \rightarrow \text{get} \ (\text{mkLenses} \ f \ s) \ s)
\]

where

\[
\text{mkLenses} \ f \ s = \text{let} \ (\ell, p) = \text{let} \ R \ m = f \ (\text{fst} \ L', \ \text{snd} \ L') \ \text{in} \ m \ (\text{get tag2L} \ s) \ \\
\ell' = \ell \ \circ \ \text{tag2L} \ s \ \text{put} \ s' \ v \ = \text{put} \ s' \ v \ \text{in} \ (p \ \text{get tag2L} \ s') \ \text{then} \ s' \ \text{else} \ \bot \ \text{in} \ L \ (\text{get} \ \ell') \ \text{put'}
\]

Although we define the \texttt{get} and \texttt{put} components of the resulting lens separately in \texttt{unliftM2}, well-behavedness is guaranteed as long as \( R \) and \( L' \) are used abstractly in \( f \). Note that, similarly to \texttt{unliftM2}, we can define \texttt{unliftM} and \texttt{unliftMT}, as monadic versions of \texttt{unlift} and \texttt{unlift2}.

We can now sprinkle \texttt{observe} at where observations happens, and use \texttt{unliftM} to guard against changes to them.

\[
\text{good} \ (x, y) = \text{fmap} \ (\lambda x. y) \ (\text{new 0}) \ \text{do} \ b \leftarrow \text{lift02} \ (\texttt{z} x \ (\text{new 0})) \ \text{return} \ (\text{if} \ b \ \text{then} \ (x, y) \ \text{else} \ (x, \text{new 1}))
\]

Here, \texttt{lift02} is defined as follows.

\[
lift02 :: \ (a \rightarrow b \rightarrow w) \rightarrow \text{List} s \ a \rightarrow \text{List} s \ b \rightarrow R \ s \ w
\]

\[
lift02 \ p \ x \ y = \text{lift0} \ ((\text{uncurry} \ p) \ (x \ \odot \ y))
\]

\[
lift0 :: \ (a \rightarrow w) \rightarrow \text{List} s \ a \rightarrow R \ s \ w
\]

\[
lift0 \ p \ x = \text{observe} \ (\text{lift} \ (L \ p \ \text{unused} \ x))
\]

\[\text{where} \ \text{unused} \ s \ v \ = \text{v} \ \text{p} \ s = \text{s}
\]

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

One might have noticed that the definition of \texttt{good} is in the \textit{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 \texttt{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 \( \forall s. [a] \rightarrow [a] \), a bidirectionalization engine in the style of [33] can be straightforwardly implemented in our framework as follows.

\[
\text{bff} :: \ (\forall s. [a] \rightarrow [a]) \rightarrow (Eq \ a \Rightarrow L \ [a] \ [a])
\]

\[
\text{bff} \ f = \text{unliftList} \ (\text{sequenceList} \ \odot \ f)
\]

Replacing \texttt{unliftList} and \texttt{sequenceList} with \texttt{unliftT} and \texttt{sequenceT}, we also obtain the datatype generic version [33].

With the addition of \texttt{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, \texttt{lift02} (and other \( n \)-ary observations-lifting functions) has to be a primitive previously [21, 22], but can now be derived from \texttt{observe}, \texttt{lift} and \( (\_ \_ = \_ \_ \ ) \) 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, QS of Use Case “STRING” in XML Query Use Case (\( \text{http://w3.org/TR/xquery-use-cases} \)) 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:

\[ \text{Functor } f \Rightarrow (B \rightarrow f B) \rightarrow (A \rightarrow f A) \]

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 \( \text{put} \) when a \( \text{get} \) 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 \( \text{get} \ell : A \rightarrow B \), 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 \( \text{put} \) from \( \text{get} \) is straightforward; replacing base type operations with the lifted bidirectional versions is suffice as shown in the \( \text{unlines}_{\ell} \) and \( \text{eval}_{\ell} \) examples. 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 \( \text{unlines}_{\ell} \) and \( \text{eval}_{\ell} \). Actually as far as we are aware, higher-order programming with the van Laarhoven representation has not been investigated before.

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 \( L (-) V \) also forms a lax monoidal functor under some conditions [30]; for example, \( V \) must be a monoid. 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 \( \text{put} \) from a \( \text{get} \).

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 combiners 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


