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 Voigtlander’s semantic bidirectionalization, creating a new programming style that is able to bag benefits from both.

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

General Terms Languages

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

1. Introduction

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

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

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

\[
\text{data } L \ s \ v = L \left\{ \text{get} :: s \rightarrow v, \text{put} :: s \rightarrow v \rightarrow s \right\}
\]

\[
fst_S :: L \ (\ a, \ b \ ) \ a \ = \ L \left( \lambda (a, \_ \ ) \rightarrow a \right) \left( \lambda (\_, \ b \ ) \rightarrow (a, \ b) \right)
\]

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 $\text{fst}_S$, 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:

\[
\text{put } \ell \ s \ (\text{get } \ell \ s) = s \quad \text{(Acceptability)}
\]

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

for all $s$, $s'$ and $v$. Note that in this paper, we write $v = v'$ with the assumption that neither $v$ nor $v'$ 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 define well-behavedness [1, 9, 13]. A bidirectional transformation $L \text{ get } \text{put}$ is called well-behaved if it satisfies well-behavedness. The above example $\text{fst}_S$ 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 languages normally start with (at least mentally) constructing a get before trying to make it bidirectional.

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

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

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

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

The key idea of our proposal is to lift lenses of type $L \ (A_1, \ldots, A_n) \ B$ to lens functions of type $\forall s. \ L^+_s \ a \rightarrow \cdots \rightarrow L^+_s \ a_n \rightarrow \cdots \rightarrow L^+_s \ b$ where $L^+_1$ 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 composition and manipulation in an applicative style, and can be passed to higher-order functions directly. For example, we can write a bidirectional version of $\texttt{unlines}$, defined by

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

as below.

\[
\begin{align*}
\texttt{unlinesF} :: & \ [L^+_1 \ s \ \text{String}] \rightarrow L^+_1 \ s \ \text{String} \\
\texttt{unlinesF} \ [\ ] & = \texttt{new} \ "" \\
\texttt{unlinesF} \ (x : xs) & = \texttt{lilt2} \ \texttt{catLineL} \ (x, \ \texttt{unlinesF} \ xs)
\end{align*}
\]

where $\texttt{catLineL}$ is a lens version of $\forall x \ y \rightarrow x + \ "\texttt{n}" + y$. In the above, except for the noise of $\texttt{new}$ and $\texttt{lilt2}$, the definition is faithful to the original structure of $\texttt{unlines}$’ definition, in an applicable style. With the heavy-lifting done in defining the lens function $\texttt{unlinesF}$, a corresponding lens $\texttt{unlinesL} :: \ L \ [\text{String}] \ \text{String}$ is readily available through straightforward unlifting: $\texttt{unlinesL} = \texttt{unliftT} \ \texttt{unlinesF}$.

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 ⊥. 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 $A \rightarrow B \rightarrow \text{Maybe} \ A$, 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/haztk/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 "◦"$
\begin{align*}
(\hat{\circ}) :: & \ L \ b \ c \rightarrow L \ a \ b \rightarrow L \ a \ c \\
(L \ \texttt{get}_2 \ \texttt{put}_2) \ (\hat{\circ}) \ (L \ \texttt{get}_1 \ \texttt{put}_1) & = \\
L \ (\texttt{get}_2 \ ◦ \ \texttt{get}_1) \ (\lambda s \ v \rightarrow \texttt{put}_1 \ s \ (\texttt{put}_2 \ (\texttt{get}_1 \ s \ v)))
\end{align*}
$

we can compose $\texttt{fstL}$ to itself to obtain a lens that operates on nested pairs, as below.

\[
\begin{align*}
\texttt{fstTriL} & :: \ L \ ((a, b), c) \ a \\
\texttt{fstTriL} & = \texttt{fstL} \ ◦ \ \texttt{fstL}
\end{align*}
\]

Well-behavedness is preserved by such compositions: $\texttt{fstTriL}$ is well-behaved by construction assuming well-behaved $\texttt{fstL}$.

The composition operator "◦" has the identity lens $\texttt{idL}$ as its unit.

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

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 $\forall s. \ L s a \rightarrow L s b$
lift :: L a b → (∀s. L s a → L s b)

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 \( ∀s. L s A → L s B \) on updatable data. We call such lifted lenses *lens functions*.

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

\[ \text{unlift} :: (∀s. L s a → L s b) → L a b \]

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.

\[
\begin{align*}
\text{fstTri}_L & \triangleq ((a, b), c) \mapsto (\text{lift} \circ \text{fst}_L \circ \text{lift} \circ \text{fst}_L) \\
\text{fstTri}_L & = \text{unlift} \circ \lambda x → \text{twice} \circ (\text{lift} \circ \text{fst}_L) \circ x
\end{align*}
\]

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

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

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 \( \text{lift} \) is injective with \( \text{unlift} \) as its left inverse.

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

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

**Proposition 2.** \( \text{lift} \circ \ell \) preserves well-behavedness if \( \ell \) is well-behaved.

**Proposition 3.** \( \text{unlift} \circ f \) is well-behaved if \( f \) preserves well-behavedness.

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

\[
\begin{align*}
f & :: \text{Eq} a ⇒ L s (\text{Maybe} a) → L s (\text{Maybe} a) \\
f (L g p) & = L g (λs v → \text{if} \ v == \text{Just} \ s \ \text{then} \ s \ \text{else} \ p \ (s Nothing \ v))
\end{align*}
\]

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 \( ∀s. L s A → L s B \). 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 \( f :: τ \) if there is a polymorphic function \( h \) of type

\[ ∀τ. (∀a. L a b → (∀s. L s a → L s b)) \rightarrow (∀a. L s a → L s b) → \tau' \]

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

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 \( \text{lift} \)) 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 example, we can state that functions of type \( ∀s. L s A → L s B \) are well-behavedness preserving as follows.

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

```haskell
module AbstractLens (Labs, liftAbs, unliftAbs) where

newtype Labs a = Labs ((a, a) → Labs a)

liftAbs :: Labs a b = Labs (unLabs :: Labs ((a, a) → Labs a))

unliftAbs :: (∀s. Labs s a → Labs s b) → Labs a b

Outside the module AbstractLens, we can use liftAbs, unliftAbs and type Labs itself, but not the constructor of Labs. Thus the only way to access data of type \( L \) is through liftAbs and unliftAbs.

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

**Lemma 1.** Let \( f \) be a function of type \( ∀s. L s A → L s B \) in which \( L \) is abstract. Then \( f \text{lift} = \text{id} \circ f \) holds for all \( \ell :: L S A \).

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

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

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 \( ∀s. L s A → L s B \) 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

- \( \{L :: L A B\} \), and
- \( \{f :: ∀s. L s A → L s B | f \text{lift} \circ y = f \text{lift} \circ y\} \).

The condition \( f \text{lift} \circ y = f \text{lift} \circ y \) is required to make \( f \) a natural transformation between functors \( L (−) A \) and \( L (−) B \); here, the contravariant functor \( L (−) A \) maps a lens \( \ell \) of type \( L X Y \) to a function \( λy → y \circ \ell \) of type \( L Y A \). Note that \( f \text{lift} \circ y = f \text{lift} \circ y \) is equivalent to \( f = f \text{lift} \circ \text{id} \circ \circ \). 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
With the duplication lens, the above-mentioned function can be defined as below. For simplicity, we assume that \( L (a, b) \) is subtly distinct from \( L (b, a) \). One reason for this is with the definition of \( \text{fst} \text{Tri} L \), which consists of the composition of lifted \( \text{fst} s \). But more fundamentally it is the type of \( \text{lift} (L x y \rightarrow (\forall s. L s x \rightarrow L s y)) \), 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 unidirectional function \( \text{swap} a :: a \rightarrow (a, b) \). 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 \( \text{unlift} \_1 \), 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 \( (\equiv) \) represents observational equivalence.

\[ \text{dup} \_L :: L s (a, b) \rightarrow L s (a, b) \]
\[ \text{dup} \_L = L \lambda s \rightarrow (s, s) (\lambda \_ s t \rightarrow r s t) \]
\[ \text{where} r s t \mid s \equiv t \quad \text{This will cause a problem.} \]

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

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

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

\[ \text{lift} \_2 :: L (a, b) \rightarrow (\forall s. L s a, L s b) \rightarrow L s (a, b) \]
\[ \text{lift} \_2 \_L (x, y) \equiv \text{lift} \_L (x \otimes y) \]

It is tempting to have the following as the inverse for \( \text{lift} \_2 \).

\[ \text{unlift} \_2 :: (\forall s. L s a, L s b) \rightarrow L s (a, b) \]
\[ \text{unlift} \_2 \_L f = f \_L \_1 (\text{snd} \_L) \]

But \( \text{unlift} \_2 \circ \text{lift} \_2 \) does not result in identity:

\[ (\text{unlift} \_2 \circ \text{lift} \_2) \_L \]
\[ = (\text{definition unfolding & \( \beta \)-reduction}) \]
\[ \equiv \text{lift} \_L (\text{fst} \_1 \circ \text{snd} \_L) \]
\[ = (\text{unfolding \( \circ \)}) \]
\[ \equiv \text{lift} \_L (\text{fst} \_1 \circ \text{snd} \_L \circ \text{dup} \_L) \]
\[ = (\text{definition unfolding}) \]
\[ \equiv \text{lift} \_L \circ \text{block} \_L, \text{where} \]
\[ \text{block} \_L = L \lambda s \rightarrow (L \lambda s s \rightarrow \text{if} \_s s = s \_\text{then} \_s \ldots) \]

Lenses \( \text{block} \_L \) 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 \( = \) that makes \( \text{unlift} \_2 \circ \text{lift} \_2 \) useless has its root in \( \text{dup} \_L \). If we look at the lens \( \text{dup} \_L \) 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 \( \text{dup} \_L \) is applied, there is more room for maneuver. Let us consider the lifting function \( \text{lift} \_2 \) again, and how \( \text{put} \_L \) works, which rejects the update above, works in the execution of \( \text{put} \_L (\text{unlift} \_2 \circ \text{lift} \_2 \_L) \).
instance Eq a ⇒ Poset (Tag a) where
  (O s) ∨ (U t) = U t
  (U s) ∨ (O t) = U s
  (O s) ∨ (O t) s ⊢ t = O s
  (U s) ∨ (U t) s ⊢ t = U s

instance (Poset a, Poset b) ⇒ Poset (a, b) where
  (a, b) ∨ (a', b') = (a ∨ a', b ∨ b')

We also introduce the following type synonym for brevity.¹

type L² s a = Poset s ⇒ L s a

As we will show later, the move from L to L² will have implications on well-behavedness.

Accordingly, we change the types of (⊕), lift and lift2 as below.

(⊕) : L² s a → L² s b → L² s (a, b)
lift : L a b → (∀ s. L² s a → L² s b)
lift2 : L (a, b) c → (∀ s. L² s a, L² s b) → L² s c

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

unlift :: Eq a ⇒ (∀ s. L² s a → L² s b) → L a b
unlift f = f idL² a ∘ tagL²

idL² a = L (Tag a) a

tagL² : L a (Tag a)
tagL² = L O (const unTag)

unlift2 :: (Eq a, Eq b) ⇒ (∀ s. L² s a, L² s b) → L² s c → L (a, b) c
unlift2 f = (fstL²L2 a b 

fstL²L2 : L² (Tag a, Tag b) a

sndL²L2 : L² (Tag a, Tag b) b

sndL²L2 = L (λ (a, b) → unTag a) (λ (a, b) → (a, b))

unlift2 f = (fstL²L2 a b, sndL²L2 a b, f (fstL²L2 a b, sndL²L2 a b))

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 ℓ) = ℓ holds for all lenses ℓ :: L (A, B) C.

Proof. We prove the statement with the following calculation.

unlift2 (lift2 ℓ) =
  = { definition unfolding & β-reduction }
  ℓ ∘ (fstL²L2, sndL²L2) ∘ tag2L²a
  = { unfolding (⊕) }
  ℓ ∘ (fstL²L2, sndL²L2) ∘ dupL² ∘ tag2L²a
  = { (fstL² ∘ sndL²) ∘ dupL ∘ tag2L = idL² } (ℓ)

We prove the statement (ℓ) by showing get ((fstL² ∘ sndL²) ∘ dupL ∘ tag2L) (a, b) = (a, b) and put ((fstL² ∘ sndL²) ∘ dupL ∘ tag2L) (a, b) = (a', b'). Since the former property is easy to prove, we only show the latter here.

put ((fstL² ∘ sndL²) ∘ dupL ∘ tag2L) (a, b) (a', b')
  = { definition unfolding & β-reduction }
  put tag2L² (a, b) $ (a, b', b')
  = { definition unfolding & β-reduction }
  put tag2L² (a, b) $ (O a, O b) (a', b')
  = { definitions of fstL² and sndL² }
  put tag2L² (a, b) $ (O a, O b) (O a, b) (O a, O b) b'
  = { definition of dupL² }
  put tag2L² (a, b) (O a', b) (O a, O b) b'
  = { definition of tag2L² }
  (a', b')

Thus, we have proved that lift2 is injective.

We can recreate γL² and γL² with unlift2, which is rather reassuring.

Proposition 5. fstL²L2 and sndL²L2 with unlift2, which is rather reassuring.

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 fstL²L2 and lift: while unlifting both functions result in liftL²L2, they actually differ as put (lift2 liftL²L2 (fstL²L2L (O a, O b) c = (O a, O b)) and put (fstL²L2L (O a, O b) c = (O a, O b)). Intuitively, lift knows that the second argument is unused, while lift2 liftL²L2 does not because liftL²L2 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 fstL²L2, sndL²L2, dupL² and tagL². 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 dupL²L², fstL²L2 and tagL²L² are not well-behaved, for different reasons. For functions fstL²L2 and sndL²L2, the difference from the original versions fstL² and sndL² 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 dupL² and tagL², 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 :: ∀ s. (L² s A, L² s B) → L² 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 dupL². 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 order between tagged values. Let us write \( \preceq \) for the partial order induced from \( \Upsilon \); that is, \( s \preceq t \) if \( s \nind t \) is defined and equal to \( t \). One can see that \( \preceq \) is the reflexive closure of \( O \subseteq \Upsilon t \). We write \( \Uparrow \) for a value obtained from \( s \) by replacing all \( O \)-tags with \( \Upsilon \)-tags. Trivially, we have \( s \Uparrow \preceq s \).

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

**Definition 2** (Local Well-Behavedness). A bidirectional transformation \( \ell :: \L_1 a \leadsto b \) is called **locally well-behaved** if the following four conditions hold.

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

In the above, tags introduced for the flexible behavior of \( put \) must not affect the behavior of \( get \); \( \Uparrow s' = \Uparrow s \) means that \( s \) and \( s' \) are equal if tags are ignored. The property local-acceptability is similar to acceptability, except that \( O \)-tags are allowed to change to \( \Upsilon \)-tags. The property local consistency is stronger than consistency in the sense that \( get \) must map all values sharing the same \( \Upsilon \)-tagged elements with \( s' \) to the same view. The idea is that \( O \)-tagged elements in \( s' \) are not connected to the view \( v \), and thus changing them will not affect \( v \). A similar reasoning applies to backward inflation stating that source elements changed by \( put \) will have \( \Upsilon \)-tags. Note that in this definition of local well-behavedness, tags are assumed to appear only in the sources. As a matter of fact, only \( dup_1 \) and \( tag_2 \) introduce tagged views; but they are always precomposed when used, as shown in the following.

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

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

- If \( x \) is well-behaved and \( y \) is locally well-behaved, then \( lift x \ y \) is locally well-behaved.
- If \( x \) and \( y \) are locally well-behaved, \( x \odot y \) is locally well-behaved.
- If \( x \) and \( y \) are locally well-behaved, \( x \odot tag_2 L \) and \( y \odot tag_2 L \) 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.

\[
put (((x \odot y) \odot dup_1) L) s (get ((x \odot y) \odot dup_1) L) s = \{
\text{ simplification }
\}
put dup_1 L (put (x \odot y) (s, s) (get (x \odot y) (s, s))) = \{
\text{ by the local acceptability of } x \odot y \}
p\ell (s, s'') \quad \text{where } s \preceq s' \preceq \Uparrow s, s \preceq s'' \preceq \Uparrow s
= \{
\text{ by the definition of } p\ell (s, s'') \text{ and that } s' \nind s'' \text{ is defined }
\}
\Uparrow s' \nind s'' \leq \Uparrow s
\]

Note that, since \( s' \preceq \Uparrow s \) and \( s'' \preceq \Uparrow s \), there is \( s' \nind s'' \preceq \Uparrow s \).

Now, we prove local consistency. Assume that \( put ((x \odot y) \odot dup_1) L) s (v_1, v_2) \) succeeds in \( s' \). Then, by the following calculation, we have \( s' = put x s v_1 \odot get y s v_2 \).

\[
\begin{align*}
put ((x \odot y) \odot dup_1) L) s (v_1, v_2) &= \{
\text{ simplification }
\}
put dup_1 L (put x s v_1, put y s v_2)
= \{
\text{ definition unfolding }
\}
put x s v_1 \odot get y s v_2
\end{align*}
\]

Let \( s'' \) be a source such that \( s' \preceq s'' \). Then, we prove \( get ((x \odot y) \odot dup_1) L) s'' = (v_1, v_2) \) as follows.

\[
\begin{align*}
\text{ get } ((x \odot y) \odot dup_1) L) s'' & (v_1, v_2) = \{
\text{ simplification }
\}
\text{ get } x s'' \odot get y s''
= \{
\text{ the local consistency of } x \text{ and } y
\}
(v_1, v_2)
\end{align*}
\]

Note that we have \( put x s v_1 \preceq s' \preceq s'' \) and \( put y s v_2 \preceq s' \preceq s'' \) by the definition of \( \Upsilon \).

Forward tag-irrelevance and backward inflation are straightforward.

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

- \( lift \ell :: \forall s. (L_1 s A \leadsto B) \) preserves local well-behavedness, if \( \ell :: L_1 A \rightarrow B \) is well-behaved.
- \( lift2 \ell :: \forall s. (L_1 s A \leadsto B) \rightarrow L_1 \) preserves local well-behavedness, if \( \ell :: L_1 (A, B) \rightarrow C \) 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 \( L \)). We say \( L_1 \) is abstract in \( f :: \tau \) if there is a polymorphic function \( h \) of type

\[
\forall \ell. (\forall a b. L_1 a b \rightarrow (\forall s. \ell s a \rightarrow s b)) \rightarrow (\forall a b. (\forall s. \ell s a \rightarrow s b)) \rightarrow (\forall s a b. \ell s a \rightarrow s b)
\]

satisfying \( f = h lift \text{ unlift2 } (\odot) \text{ unlift2 } 2 \quad \text{ and } \quad \tau' = \tau[\ell/L_1] \).

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

**Theorem 4.** Let \( f \) be a function of type \( \forall s. (L_1 s A \leadsto B) \rightarrow L_1 s C \) in which \( L_1 \) is abstract. Then, \( f (x, y) \) is locally well-behaved if \( x \) and \( y \) are also locally well-behaved, assuming that \( lift \) is applied only to well-behaved lenses.

**Corollary 2.** Let \( f \) be a function of type \( \forall s. (L_1 s A \leadsto B) \rightarrow L_1 s C \) in which \( L_1 \) is abstract. Then, \( \text{unlift2 } f \) 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.

\[
\text{swap}_L :: (\text{Eq } a \rightarrow \text{Eq } b) \Rightarrow L_1 (a, b) (b, a)
\]

\[
\text{swap}_L = \text{unlift2 } (\text{lift2 } \text{id}_L \circ \text{swap})
\]

And it behaves as expected.

\[
\begin{align*}
\text{put } \text{swap}_L L_1 (1, 2) (4, 3)
= \{
\text{ unfold definitions }
\}
\text{put } ((\text{snd}_L \odot \text{fst}_L) \odot \text{dup}_L \odot \text{tag}_2 L) (1, 2) (4, 3)
= \{
\text{ simplifications }
\}
\text{put } \text{tag}_2 L (1, 2)
\text{ put } \text{dup}_L L_1 (O 1, O 2) \text{ put } \text{snd}_L L_1 (O 1, O 2) \text{ put } \text{fst}_L L_1 O 1, O 2 3)
= \{
\text{ definition of } \text{fst}_L \text{ and } \text{snd}_L
\}
\text{put } \text{tag}_2 L (1, 2)
\text{ put } \text{dup}_L L_1 (O 1, O 2) \text{ put } \text{dup}_L L_1 (O 1, U 4, (U 3, O 2))
\end{align*}
\]
It is worth mentioning that (⊗) is the base for “splitting” and “lifting” tuples of arbitrary arity. For example, the triple case is as follows.

\[
\text{split3 } :: (L^3 s a, L^3 s b, L^3 s c) \to L^3 s (a, b, c)
\]

\[
\text{split3 } (x, y, z) = \text{lift flattenL}_3 ((x \otimes y) \otimes z)
\]

\[
\begin{align*}
\text{where } & \text{flattenL}_3 :: (L (a, b, c), (a, b, c) \to L (\lambda((x, y), z) \to (x, y, z))) \\
\text{flattenL}_3 &= L (\lambda((x, y), z) \to (x, y, z))
\end{align*}
\]

\[
\text{lift3 } \ell \cdot t = \text{lift } \ell (\text{split3 } t)
\]

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.

\[
\text{split3}' (x, y, z) = \text{lift flattenR}_3 (x \otimes (y \otimes z))
\]

\[
\begin{align*}
\text{where } & \text{flattenR}_3 :: (L (a, b, c), (a, b, c) \to L (\lambda((x, y), z) \to (x, y, z))) \\
\text{flattenR}_3 &= L (\lambda((x, y), z) \to (x, y, z))
\end{align*}
\]

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

To complete the picture, the nullary lens function

\[
\begin{align*}
\text{unit} & :: \forall s. L^0 s () \\
\text{unit} &= L (\lambda s (\to s))
\end{align*}
\]

is the unit for (⊗). Theoretically, \((L^0 s (-), \otimes, \text{unit})\) 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.

\[
\text{new} :: \text{Eq } a \Rightarrow a \to \forall s. L^0 s a
\]

\[
\text{new } a = \text{lift } (L (\text{const } a) (\lambda a' \to \text{check } a a'))
\]

\[
\begin{align*}
\text{where } & \text{check } a a' = \text{if } a :: a' \text{ then } () \text{ else error "Update on constant"}
\end{align*}
\]

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 \(A S (-)\) is a functor from the category of lenses to the category of sets and (total) functions, which maps \(\ell :: L A B\) to \(\text{lift } \ell :: L S A \to L S B\) for any \(S\). In the case that \(S\) is tagged and thus partially ordered, \((L^0 S (-), \otimes, \text{unit})\) forms a lax monoidal functor, under the following conditions.

- \(\otimes\) must be natural, i.e., \((\text{lift } f x) \otimes (\text{lift } g y) = \text{lift } (f \otimes g) (x \otimes y)\) for all \(f, g, x, y\) with appropriate types.
- \(\text{split3}\) and \(\text{split3}'\) coincide.
- \(\text{lift } \text{elimUnitL}_3 (\text{unit} @ x) = x\) must hold where \(\text{elimUnitL}_3 :: L () a\) 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 \(s \forall \text{ put } x s v = \text{ put } x s v\), 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 \(L^0 S A\) as a quotient set of \(L S A\) by the equivalence relation \(\equiv\) defined as \(x \equiv y\) if \(\text{get } x = \text{ get } y\), \(\text{put } x s = \text{ put } y s\) for all minimal \(s\). This equivalence is preserved by manipulations of \(L^0\)-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 \(x \otimes z \equiv y \otimes w\).
- \(x \equiv y\) implies \(x \circ \text{tagL} = y \circ \text{tagL}\) (or \(x \circ \text{tag2L} = y \circ \text{tag2L}\)).

Note that the above three cases cover the only ways to construct/describe \(L^0 f\) when \(L^0\) is abstract. The third condition says that this “coarse” equivalence \((\equiv)\) on \(L^0\) can be “sharpened” to the usual extensional equality \((\approx)\) by \(\text{tagL}\) and \(\text{tag2L}\) 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 \textit{map} to a lens function \(\text{lift } \ell\), we will obtain a function of type \(\text{map } (\text{lift } \ell) :: [L^0 s A] \to [L^0 s B]\). But what we really would like is a lens of type \(L [A] [B]\). The way to achieve this is to internally treat length-\(n\) lists as \(n\)-ary tuples. This treatment effectively restricts us to in-place updates of views (i.e., no change is allowed to the list structure); we will revisit this issue in more detail in Section 6.1.

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

\[
\begin{align*}
\text{listsequence}_{\text{list}} & :: [L^0 s a] \to L^0 s [a] \\
\text{listsequence}_{\text{list}} [] &= \text{lift nilL } \text{unit} \\
\text{listsequence}_{\text{list}} (x : xs) &= \text{lift2 consL} (x, \text{listsequence}_{\text{list}} xs) \\
\text{nilL}_3 &= L (\lambda (\to [])) (\lambda []) \to (\to [])) \\
\text{consL}_3 &= L (\lambda (a, as) \to (a : as)) (\lambda (a' : as') \to (a' : as'))
\end{align*}
\]

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

\[
\begin{align*}
\text{listLift}_{\text{list}} & :: [L [a] b] \to \forall s. [L^0 s a] \to L^0 s b \\
\text{listLift } \ell \text{xs} &= \text{lift } \ell (\text{listsequence}_{\text{list}} \text{xs})
\end{align*}
\]

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

**Instance**

\[
\begin{align*}
\text{Poset } a \Rightarrow \text{Poset } [a] \text{ where } \\
xs \forall ys \equiv \text{if length } xs \neq \text{length } ys \text{ then zipWith (\circ \circ) xs ys } \\
\quad \text{else } \bot & - - \text{Unreachable in our framework}
\end{align*}
\]

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 \(L^0\) in lifted functions.

The definition of \(\text{unlift3}_{\text{list}}\) 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. \text{Eq} \ a \Rightarrow (\text{List}\ a \rightarrow \text{List}\ s) \rightarrow \text{List}\ a \rightarrow \text{List}\ s

unlift \text{list} \ f = L (\lambda s \rightarrow \text{get} (\text{mkLens} s) s) (\lambda s \rightarrow \text{put} (\text{mkLens} s) s)

where

\text{mkLens} s = f (\text{projs} (\text{length} s)) \odot \text{tagList} _L

\text{tagList} _L = L (\text{map} O) (\lambda y \rightarrow \text{map} \text{unTag} y)

\text{projs} n = \text{map} \text{proj} _L [0..n-1]

\text{proj} _L \ i = L (\lambda x \rightarrow \text{unTag} (xs \uplus i)) (\lambda a \rightarrow \text{update} i (U a) as)

Giving that we need to inspect the length of the source leads to the
separated definitions of \text{get} and \text{put} in the above, there might be
worry that we may lose the guarantee of well-behavness 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 \text{lift2},
\text{lift\text{list}} is an injection with \text{unlift\text{list}} as its left inverse.

Example 2 (Bidirectional \text{tail}). Let us consider the function \text{tail}.

tail :: [a] \rightarrow [a]
tail (x : xs) = xs

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

\text{tail} _L :: \text{Eq} \ a \Rightarrow L [a] [a]
\text{tail} _L = \text{unlift\text{list}} (\text{sequence\text{list}} \circ \text{tail})

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

Example 3 (Bidirectional \text{unlines}). Let us consider a bidirec-
tional version of \text{unlines} :: [\text{String}] \rightarrow \text{String} that concatenate
lines, after appending a terminating newline to each. For example,
\text{unlines} ["a\nbc", "de"] = "a\nbc\n\n de". In conventional unidirectional
programming, one can implement \text{unlines} as follows.

\text{unlines} [\ ] = ""\n\text{unlines} (x : xs) = \text{catLine} x \ (\text{unlines} xs)
\text{catLine} x y = x \ + \ "\n" \ + \ y

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

\text{catLine} _L :: L (\text{String}, \text{String}) \text{String}
\text{catLine} _L =
L (\lambda s, t \rightarrow s \ + \ "\n" \ + \ t)
(\lambda s, t \rightarrow \text{let} n = \text{length} (\text{filter} (xz \rightarrow \text{length} xz) s)
 i = \text{elemIndices} n? \ n \ !! \ n
 (s', t') = \text{splitAt} i u
 (\text{in} (s', t'))

Here, \text{elemIndices} and \text{splitAt} are functions from \text{Data.List}.
\text{elemIndices} \ c s returns the indices of the elements that are equal
to \ c; \text{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, \text{put catLine} _L (s, t) u \ splits \ u \ \text{into} \ s' \ \text{and} \ "\n" \ + \ t'
so that \ s' contains the same number of newlines as the original \ s.
For example, \text{put catLine} _L ("a\nbc", "de") "A\nB\n\n C" =
("A\nB", "C").

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

\text{unlines} _L :: L [\text{String}] \text{String}
\text{unlines} _L = \text{unlift\text{list}} \text{unlines}

\text{unlines} _F :: \forall s. [L^1 s \text{String}] \rightarrow L^1 s \text{String}
\text{unlines} _F [] = \text{new} **
\text{unlines} _F (x : xs) = \text{lift2} \text{catLine} _L (x, \text{unlines} _F xs)

As one can see, \text{unlines} _F is written in the same applicative style
as \text{unlines}. The construction principle is: if the original function
handles data that one would like update bidirectionally (e.g., \text{String}
in this case), replace the all manipulations (e.g., \text{catLine} and \text{new})
with the data with the corresponding bidirectional versions (e.g.,
\text{lift2} \text{catLine} _L and \text{new} **).

\text{Lens unlines} _L accepts updates that do not change the original
formatting of the view (i.e., the same number of lines and an
last line). For example, we have \text{put unlines} _L ["a", "b", "c"]
"A\nB\nC" = ["A", "B", "C"] , but \text{put unlines} _L
["a", "b", "c"] "A\nB\nC" = \perp and \text{put unlines} _L ["a", "b", "c"] "A\nB\nC\nD" = \perp.

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

\text{unlines} = \text{foldr} \text{catLine} **

The same coding principle for constructing bidirectional versions
applies.

\text{unlines} _L :: L [\text{String}] \text{String}
\text{unlines} _L = \text{unlift\text{list}} \text{unlines}

\text{unlines} _F :: \forall s. [L^1 s \text{String}] \rightarrow L^1 s \text{String}
\text{unlines} _F = \text{foldr} (\text{lift2} \text{catLine} _L) (\text{new} **)

The new \text{unlines} _F is again in the same applicative style as the
new \text{unlines}, where the unidirectional function \text{foldr} is applied to
normal functions and lens functions alike.

For readers familiar with the literature of bidirectional transfor-
mation, this restriction to in-place updates is very similar to that
in semantic bidirecational [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 \text{traverse},
which can be found in \text{Data.Traversable}, to give data-type
generic lifting and unlifting functions.

\text{traverse} :: (\text{Traversable} t, \text{Applicative} f)
\Rightarrow (a \rightarrow f \ b) \rightarrow t \ a \rightarrow f \ (t \ b)

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

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

\text{2} In \text{GHC}, the function \text{content} is called \text{toList}, which is defined in
\text{Data.Foldable} (\text{Every} \text{Traversable} instance is also an instance of
\text{Foldable}). We use the name \text{content} to emphasize the function's role
of extracting contents from structures [3].
fill :: Traversable t ⇒ t b → [a] → t a
fill t ℓ = evalState (traverse next ℓ) t

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 Monad. 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 an instance of Functor. For such data types, 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 lsequence function as follows.

lsequence :: (Eq a, Eq (t ()), Traversable t) ⇒ t (L' s a) → L' s (t a)
lsequence t = lift (fillL (shape t)) (lsequence Last contents t)
where
fillL s t = L (λxs → fill s xs) (λα t → contents' s t)

contents' 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.⁴

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 unfolding function with length replaced by shape.

unliftT :: (Eq (t ()), Eq a, Traversable t) ⇒ (∀s. t (L' s a) → L' s b) → L (t a) b
unliftT f = L (λxs → get (mkLens s) s)

(λα → put (mkLens s) s)

where
mkLens s = f (projTs (shape s)) ∘ tagTLₗ
projTs sh = λn = length (contents sh)
in fill sh [projTLₗ i sh | i ← [0‥n - 1]]
projTLₗ i sh = L (λv → fill sh (update i (U v) (contents s)))

Here, projTLₗ 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 = EVal Int | ENum Exp
          | EVar String | EApp Exp Exp
          | EFun String Exp deriving Eq

data Val a = VNum a
          | VFun String (Env a) deriving Eq

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

This definition is standard, except that the type of values is parameterized to accommodate both Val (L' 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 (L' s Int) → Exp → Val (L' s Int)
eval env (EVal n) = VNum (new n)
eval env (EInt 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 e x) = VFun x e env

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

inclL = 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 (λ 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 ℓ twice ℓ twice ℓ twice ℓ twice ℓ inc ℓ x

where
twice = EFun "f" (EFun "x" (EFun "f" (EFun "x" (EFun "x"))))
x = EVar "x"
inc = EFun "x" (EInc (EVar "x"))

³ This definition actually overlaps with that for pairs. So we either need to have “wrapper” type constructors, or enable OverlappingInstances.
mapDefault :: L [(Int, Int)] [Int]
mapAddL = unfold mapDefault

mapAddL xs = mapAdd (0, 0) (lift addL) xs
mapAddL d = liftC (mapDefaultC d) addL

When given a lens on elements, mapDefaultC 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 mapDefaultC as the following, which in the forward direction is essentially map (uncurry (+)).

mapAddL :: L [(Int, Int)] [Int]
mapAddL = unfold mapAdd

An advantage of the original lens combinators [9] (that operate we have presented so far is the ability to accept shape changes to c . . . λ . . .)

mapDefault incorporate this behavior into our framework. For example, we can 6. Extensions

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

6.1 Lifting Lens-Combinators

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

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

\[
\text{liftC} :: \forall s. L s a \rightarrow (L' s b) \rightarrow (\forall t. L' t c \rightarrow L' t d) \\
\text{liftC} c f = lift (c \ (\text{unlift} f))
\]

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

As an example, let us consider the following lens combinator mapDefaultC.

\[
\text{mapDefaultC} :: a \rightarrow L a b \rightarrow L [a] [b] \\
\text{mapDefaultC} \ d \ e \ = \ L \ (\text{map} \ (\text{get} \ e)) \ (\lambda s \rightarrow \text{go} \ s \ v)
\]

\[
\text{where} \ \\
\text{go} \ [] \ = \ [] \\
\text{go} \ [x:xs] \ = \ \text{put} \ e \ d \ v \ : \ \text{go} \ [] \ vs \\
\text{go} \ (s:ss) \ = \ \text{put} \ e \ s \ v \ : \ \text{go} \ ss \ vs
\]

When given a lens on elements, mapDefaultC 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 mapDefaultC as the following, which in the forward direction is essentially map (uncurry (+)).

\[
\text{mapAddL} :: L [(Int, Int)] [Int] \\
\text{mapAddL} = \text{unfold} \ \text{mapAdd}
\]

\[
\text{mapAddL} \ [s_1, \ldots, s_n] \ x \ = \ \text{mapAdd} \ [s_1, \ldots, s_n] \ x
\]

\[
\text{mapAddL} \ (s_1, \ldots, s_n) \ x \ = \ \text{mapAdd} \ (s_1, \ldots, s_n) \ x
\]

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.³

```plaintext
bad (x, y) = if x :: new 0 then (x, y) else (x, new 1)
```

In this program, the behavior of the transformation depends on the ‘observation’ made to a value that may potentially be updated in the view. Then the naively obtained lens `badL = unlift2 (lift2 idL o bad)` would violate well-behavedness, as `put badL (0, 2) (1, 2) = (1, 2)` but `get badL (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.

```plaintext
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` `Bool` which must remain satisfied amid updates of `s`. Formally, giving `R m :: R A B`, for any `s`, if `(~ p) = m s` then we have: (1) `p s = True`; (2) `s p' = True` implies `m s = m s'` for any `s'`. It is not difficult to make `R s` an instance of `Monad`—it is a composition of `Reader` and `Writer` monads. We only show the definition of `(=⇒)`.

```plaintext
R m ⇒⇒ f = R $ λs → let (x, c1) = m s (y, c2) = let R k = f x in k s in (y, λs → c1 s ∧ c2 s)
```

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

```plaintext
observe :: Eq w ⇒ L' s w → R s w observe ℓ = R (λs → let w = get ℓ s in (w, λs' → get ℓ s' ⇨ w))
```

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 `∀ a. [s] → [a]`, a bidirectionalization engine in the style of [33] can be straightforwardly implemented in our framework as follows.

```plaintext
bff :: (∀ a. [s] → [a]) → (Eq a ⇒ L [a] [a])
bff f = unlift (sequence (s inserts f))
```

Replacing `unlift_list` and `sequence_list` with `unliftT` and `sequence`, we also obtain the datatype generic version [33].

With the addition of `observe` and the monadic unfolding 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 `(⊙)` 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.

³This code actually does not type check as `mkLens` on `(L' 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.
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:

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

Intuitively, we can read \( A \rightarrow f \ A \) as updates on \( A \) and \( f \) 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 \( L \ (\rightarrow \) \( V \)), they apply the Yoneda Lemma to a functor \( (V, V \rightarrow (\rightarrow)) \). 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` and `eval`, 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 `unlines` and `eval`. 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 \( L \ (\rightarrow \ V \) 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 unified 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


