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  
kzk@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

\[
\begin{align*}
\text{data } & L \ s \ v = L \{ \text{get} :: s \rightarrow v, \text{put} :: s \rightarrow v \rightarrow s \} \\
\text{fst}_L & :: (a, b) \rightarrow (a, b) \\
\text{fst}_L & = \lambda (a, b) . (a, b)
\end{align*}
\]

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:

\[
\begin{align*}
\text{put } & \ell \ s \ (\text{get } \ell \ s) = s \quad \text{(Acceptability)} \\
\text{get } & \ell \ s' \ v \text{ if put } \ell \ s \ v = s' \quad \text{(Consistency)}
\end{align*}
\]

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 define well-behavedness [1, 9, 13]. A bidirectional transformation \( L \) 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 get and the corresponding backward transformation put. However, this is a significant duplication of work, because the two transformations are closely related. Moreover, it is prone to error, because they do really have to correspond with each other to be well-behaved. And, even worse, it introduces a maintenance issue, because changes to one transformation entail matching changes to the other. Therefore, a lot of work has gone into ways to reduce this duplication and the problems it causes; in particular, there has been a recent rise in linguistic approaches to streamlining bidirectional transformations [2, 4, 9–11, 14, 16, 20–22, 25, 27, 30, 33, 35, 36, 38–41].
Ideally, bidirectional programming should be as easy as usual unidirectional programming. For this to be possible, techniques of conventional languages such as applicative-style and higher-order programming need to be available in the bidirectional languages, so that existing programming idioms and abstraction methods can be ported over. It makes sense to at least allow programmers to treat functions as first-class objects and have them applied explicitly. It is also beneficial to be able to write bidirectional programs in the same style of their gets, as cultivated by traditional unidirectional programming programmers normally start with (at least mentally) constructing a get before trying to make it bidirectional.

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

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

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

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

The key idea of our proposal is to lift lenses of type $L (A_1, \ldots, A_n) B$ to lens functions of type $\forall s. L^s s A_1 \to \cdots \to L^s s A_n \to \cdots \to L^s s B$ 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. No better way to compose individually constructed bidirectional transformations.

In particular, we develop a representation of 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 $\circ$

\[(\circ) :: L b c \to L a b \to L a c\]

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

can we compose $\text{fst}_{L_1}$ to itself to obtain a lens that operates on nested pairs, as below.

\[
\text{fstTri}_{L_1} :: L ((a, b), c) a
\]

\[\text{fstTri}_{L_1} = \text{fst}_{L_1} \circ \text{fst}_{L_1}\]

Well-behavedness is preserved by such compositions: $\text{fstTri}_{L_1}$ is well-behaved by construction assuming well-behaved $\text{fst}_{L_1}$.

The composition operator $\circ$ has the identity lens $\text{id}_{L_1}$ as its unit.

\[
\text{id}_{L_1} :: L a a
\]

\[\text{id}_{L_1} = \text{L id} (\lambda_{a,v} v \to v)\]

### 2.1 Basic Idea: A Functional Representation Inspired by Yoneda

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

Inspired by the Yoneda embedding in category theory [19], we lift lenses of type $L a b$ to polymorphic functions of type $\forall s. L s a \to L s b$
by lens composition
\[ \text{lif} :: L a b \to (\forall s. L s a \to L s b) \]
\[ \text{lif} \ell = \lambda x \to \ell \circ x \]

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{lif} is injective, and has the following left inverse.

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

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

\[ \text{fstTri}_L :: ((a, b), c) \to a \]
\[ \text{fstTri}_L = \text{unlif} (\text{lif} \circ \text{fst}_L \circ \text{lif} \circ \text{fst}_L) \]

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

\[ \text{fstTri}_L = \text{unlif} (\lambda x \to \text{twice} (\text{lif} \circ \text{fst}_L) x) \]
\[ \text{where twice} f x = f (f x) \]

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

### 2.2 Formal Properties of Lens Functions
We reconfirm that \textit{lif} is injective with \textit{unlif} as its left inverse.

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

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

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

**Proposition 3.** \( \text{unlif} f \) \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
\[ f :: \text{Eq} a \Rightarrow L s (\text{Maybe} a) \to L s (\text{Maybe} a) \]
\[ f (L g p) = L g (\lambda s v \to \begin{cases} a \to a & \text{if } v =: g s \text{ then } a \\ p \to \text{Nothing} & \text{else } \end{cases}) v \]

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

In our framework the intention is that all lens functions are constructed through lifting, which sees bidirectional transformations as atomic objects. Thus, we require that \( L \) is used as an “abstract type” in defining lens functions of type \( \forall s. L s A \to L s B \). 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 \( \tau' = \tau[\ell / L] \) and \( f = h \comp \text{lif} \comp \text{unlif} \).

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{lif}) 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 \( \forall s. L s A \to L s B \) 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 \text{lif} in the definition of \( f \) are to well-behaved lenses. Then, \( f \) preserves well-behavedness, and thus \( \text{unlif} f \) is well-behaved.

### 2.3 Guaranteeing Abstraction

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

\begin{verbatim}
module AbstractLens (Labs, lifAbs, unlifAbs) where

newtype Labs a b = Labs { unLabs :: Labs a b }

lifAbs :: Labs a b \to (\forall s. Labs s a \to Labs s b)
unlifAbs :: (\forall s. Labs s a \to Labs s b) \to Labs a b

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

A consequence of having abstract \( L \) is that \text{lif} is now surjective (and \text{unlif} 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. If \( \ell = f \circ \text{id}_L \circ \ell \) holds for all \( \ell :: L S A \).

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

**Theorem 2.** For any \( f :: \forall s. L s A \to L s B \) in which \( L \) is abstract, \( \text{unlif} f = f \circ \text{id}_L \circ \text{unlif} \).

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

### 2.4 Categorical Notes

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

- \( \{ :: L A B \} \),
- \( \{ :: \forall s. L s A \to L s B \} \).

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

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

Also note that \(L s (\_\) is a functor that maps a lens \(\ell\) to a function \(lift \ell\). It is not difficult to check that \(\ell x \circ lift y = lift (x \circ y)\) and \(lift (id_L :: L A A) = (id :: L s A \to L s A)\).

3. Lifting \(n\)-ary Lenses and Flexible Duplication

So far we have presented a system that lifts lenses to functions, manipulates the functions, and then “unlifts” the results to construct composite lenses. One example is \texttt{fstTri}\(_L\) from Section 2 reproduced below.

\[
\text{\texttt{fstTri}\(_L\)} :: L ((a, b), c) a \\
\text{\texttt{fstTri}\(_L\)} = \text{\texttt{unlift}} \ (\text{\texttt{lift}} \ \text{\texttt{fst}}_L \circ \text{\texttt{lift}} \ \text{\texttt{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 \texttt{fstTri}\(_L\), which consists of the composition of lifted \texttt{fst}\(_L\)s. But more fundamentally it is the type of lift \((L x y \to (\forall s. L s x \to L s y))\), which treats \(x\) as a black box, that has prevented us from rearranging the tuple components.

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

\[
\text{\texttt{swap}}_1 :: L (a, b) (b, a) \\
\text{\texttt{swap}}_1 = \ldots
\]

Following the programming pattern developed so far, we would like to construct this lens with the familiar unidirectional function \texttt{swap}\(_2\)::\((a, b) \to (b, a)\). But since lift only produces unary functions of type \(\forall s. L s A \to L s B\), despite the fact that \(A\) and \(B\) are actually pair types here, there is no way to compose \texttt{swap} with the resulting lens function.

In order to construct \texttt{swap}\(_1\) and many other lenses, including \texttt{unlines1}, in Section 1, a conversion of values of type \(\forall s. (L s A_1, \ldots, L s A_n)\) to values of type \(\forall s. L s (A_1, \ldots, A_n)\) is needed. In this section we look at how such a conversion can be defined for binary lenses, which can be easily extended to arbitrary \(n\)-ary cases.

3.1 Caveats of the Duplication Lens

To define a function of type \(\forall s. (L s A, L s B) \to L s (A, B)\), we use the duplication lens \texttt{dup}\(_1\) (also known as \texttt{copy} elsewhere [9]) defined as below. For simplicity, we assume that \(\texttt{zip}\) represents observational equivalence.

\[
\text{\texttt{dup}}_1 :: \text{\texttt{Eq}} s \Rightarrow L s (s, s) \\
\text{\texttt{dup}}_1 = L (\lambda s \to (s, s)) (\lambda (s, t) \to r s t) \\
\text{\texttt{where}} r s t | s = t = s \quad \text{-- This will cause a problem.}
\]

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

\[
(\otimes) :: \text{\texttt{Eq}} s \Rightarrow L s a \to L s b \to L s (a, b) \\
x \otimes y = (x \otimes y) \circ \text{\texttt{dup}}_1
\]

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

\[
(\otimes) :: L a a' \to L b b' \to L (a, b) (a', b') \\
(\text{\texttt{get}1 \text{\texttt{pat}1}} \otimes (\text{\texttt{get}2 \text{\texttt{pat}2}}) = \\
L (\lambda (a, b) \to (\text{\texttt{get}1} a, \text{\texttt{get}2} b)) \\
(\lambda (a, b) (a', b') \to (\text{\texttt{pat}1} a a', \text{\texttt{pat}2} b b'))
\]

We call \((\otimes)\) “split” in this paper. With \((\otimes)\) we can support the lifting of binary lenses as below.
instance Eq a ⇒ Poset (Tag a) where
(O s) γ (U t) = U i
(O s) γ (O t) = U s
(O s) γ (O t) | s :: t = O s
(O s) γ (U t) | s :: U s

instance (Poset a, Poset b) ⇒ Poset (a, b) where
(a, b) γ (a', b') = (a γ a', b γ 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.

(⊙) :: L¹ s a → L¹ s b → L¹ s (a, b)
lift :: L a b → (∀s. L¹ s a → L¹ s b)
lift2 :: L (a, b) c → (∀s. (L¹ s a, L¹ s b) → L¹ s c)

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

unlift :: Eq a ⇒ (∀s. L¹ s a → L¹ s b) → L a b
unlift f = f idL¹ ⊙ tagL¹

idL¹ = L (Tag a) a

unlift2 :: (Eq a, Eq b) ⇒ (∀s. (L¹ s a, L¹ s b) → L¹ s c) → L (a, b) c
unlift2 f = f (fstL¹ ⊙ sndL¹ ⊙ tag2L¹)

fstL¹ :: L¹ (Tag a, Tag b) a

fstL¹ = L (λ(a, _) → unTag a) (λ(_, b) → a) (U, a) b)
sndL¹ :: L¹ (Tag a, Tag b) b

sndL¹ = L (λ(_, b) → unTag b) (λ(a, _) → (a, U b))
tag2L¹ :: L (a, b) (Tag a, Tag b)
tag2L¹ = L (λ(a, b) → (O a, O 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 ○ fstL¹ ⊙ sndL¹ ○ tag2L¹ = [ unfolding (⊙) ] ℓ ○ (fstL¹ ⊙ sndL¹) ○ dupL ○ tag2L¹ = [ (fstL¹ ⊙ sndL¹) ○ dupL ○ tag2L¹ = idL¹ ] ℓ

We prove the statement (*) by showing get ((fstL¹ ⊙ sndL¹) ○ dupL ○ tag2L¹) (a, b) = (a, b) and put ((fstL¹ ⊙ sndL¹) ○ dupL ○ tag2L¹) (a, b) = (a', b'). Since the former property is easy to prove, we only show the latter here.

put ((fstL¹ ⊙ sndL¹) ○ dupL ○ tag2L¹) (a, b) (a', b') = [ definition unfolding & β-reduction ] put tag2L¹ (a, b) $ put dupL (O a, O b) $ put sndL¹ (O a, O a) (a', b') = [ definitions of fstL¹ and sndL¹ ] put tag2L¹ (a, b) $ put dupL (O a, O b) (U a', O b), (O a, O b') = [ definition of dupL ] put tag2L¹ (a, b) (U a', U b') = [ definition of tag2L¹ ] (a', b')

Thus, we have proved that lift2 is injective.

We can re-create fstL¹ and sndL¹ with unlift2, which is rather reassuring.

Proposition 5. fstL¹ = unlift2 fst and sndL¹ = unlift2 snd.

Note that now unlift and unlift2 are no longer injective (even with abstract L); there exist functions that are not equivalent but coincide after unlifting. An example of such is the pair lift2 fstL¹ and fSt: while unlifting both functions result in fSt, they actually differ as put (lift2 fstL¹ (fstL¹', sndL¹')) (O a, O b) c = (U c, U b) and put (fstL¹ (fstL¹', sndL¹')) (O a, O b) c = (U c, O b). Intuitively, fst knows that the second argument is unused, while lift2 fstL¹ does not because fstL¹ 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 fStL¹, sndL¹, dupL, and tag2L¹. 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 dupL, fStL¹, sndL¹, and tag2L¹ are not well-behaved, for different reasons. For functions fStL¹ and sndL¹, the difference from the original versions fStL¹ and sndL¹ 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 dupL and tag2L¹, 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. (L¹ s A, L¹ s B) → L¹ 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 dupL. 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 \(\gamma\); that is, \(s \preceq t\) if \(s \gamma t\) is defined and equal to \(t\). One can see that \((\preceq)\) is the reflexive closure of \(O s \preceq U t\). We write \(\uparrow s\) for a value obtained from \(s\) by replacing all \(O\) tags with \(U\) tags. Trivially, we have \(s \preceq \uparrow s\). But there exists \(s'\) such that \(s \preceq s'\) and \(s' \not\preceq \uparrow s\).

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

**Definition 2** (Local Well-Behavedness). A bidirectional transformation \(\ell : L^1 \to L^2\) 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 \(\ell s\) succeeds as \(s'\), then \(s \preceq s'\).
- **(Local Acceptability)** For all \(s\) and \(v\), assuming \(\ell s\) succeeds as \(s'\), then for all \(s''\) with \(s' \preceq s''\), \(\ell s'' = v\) holds.
- **(Local Consistency)** For all \(s\) and \(v\), assuming \(\ell s\) succeeds as \(s'\), then for all \(s''\) with \(s' \preceq s''\), \(\ell s'' = v\) holds.

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

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

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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{put}((x \otimes y) \circ \text{dup}_L) s ((x \otimes y) \circ \text{dup}_L) s)</td>
<td>[simplification]</td>
</tr>
<tr>
<td>(\text{put dup}_L s ((x \otimes y) s, s) (x \otimes y) (s, s))</td>
<td>[by the local acceptability of (x \otimes y)]</td>
</tr>
<tr>
<td>(\text{put dup}_L (s, s'')) (\rightarrow) where (s \preceq s' \preceq \uparrow s), (s \preceq s'' \preceq \uparrow s)</td>
<td>[by the definition of (\text{dup}_L) and that (s' \gamma s'') is defined]</td>
</tr>
</tbody>
</table>
| \(s' \gamma s'' \preceq \uparrow s\) | Note that, since \(s' \preceq \uparrow s\) and \(s'' \preceq \uparrow s\), there is \(s' \gamma s'' \preceq \uparrow s\). Then, we prove local consistency.

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

- \(\text{lift}\ell : \forall s. L^1 s A \rightarrow L^2 s B\) preserves local well-behavedness, if \(\ell : L^1 A B\) is well-behaved.
- \(\text{lift2}\ell : \forall s. (L^1 s A, L^2 s B) \rightarrow L^1 s C\) preserves local well-behavedness, if \(\ell : (L^1 A, B) C\) is well-behaved.

Similarly 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 \(\forall \ell. (\forall a b. L^1 a b) \rightarrow L^1\) such that \(L^1\) is abstract.

**Theorem 4.** Let \(f\) be a function of type \(\forall s. (L^1 s A, L^2 s B) \rightarrow L^1 s C\) in which \(L^1 s C\) is abstract. Then, \(f (x, y)\) is locally well-behaved if \(x\) and \(y\) are also locally well-behaved, assuming that \(\ell\) is applied only to well-behaved lenses.

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

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

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

And it behaves as expected.

\[\text{put}\text{swap}_L (1, 2) (4, 3)\]

\[= \{ \text{unfold definitions} \}\]

\[\text{put}\text{dup}_L (O 1, O 2)\]

\[= \{ \text{simplifications} \}\]

\[\text{put}\text{tag2}_L (1, 2) \circ (\text{snd}_L (O 1, O 2) 4, \text{fst}_L (O 1, O 2) 3)\]

\[= \{ \text{definition of}\text{tag2}_L \}\]

\[= \text{dup}_L (O 1, O 2) \circ (O 1, U 4, U 3, O 2)\]
It is worth mentioning that $\odot$ is the base for “splitting” and “lifting” tuples of arbitrary arity. For example, the triple case is as follows.

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

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

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

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 split3, we have decided to nest to the left in the intermediate step. This choice is not essential.

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

\[
\text{where } \text{flattenR}_L = L (\lambda ((a, b), c) (a, b, c))
\]

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

The two definitions split3 and split3’ coincide.

To complete the picture, the null lens function

\[
\text{unit} :: \forall s. L^3 s ()
\]

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

is the unit for $\odot$. Theoretically, $(L^3 s) (\odot) \text{unit}$ forms a lax monoidal functor (Section XI.2 in [19]) under certain conditions (see Section 3.4). Practically, unit enables us to define the following combinator.

\[
\text{new :: } \text{Eq } a \Rightarrow a \rightarrow \forall s. L^3 s a
\]

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

\[
\text{where }
\]

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

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

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

### 3.4 Categorical Notes

Recall that $L \ S$ is a functor from the category of lenses to the category of sets and (total) functions, which maps $\ell :: L A B$ to $\text{lift } \ell :: L S A \rightarrow L S B$ for any $S$. In the case that $S$ is tagged and thus partially ordered, $(L^3 S) (\odot), \text{unit}$ forms a lax monoidal functor, under the following conditions.

- $\odot$ must be natural, i.e., $(\text{lift } f x) \odot (\text{lift } g y) = \text{lift } (f \odot g) (x @ y)$ for all $f, g, x$ and $y$ with appropriate types.

- $\text{split3}$ and $\text{split3’}$ coincide.

- $\text{lift } \text{elimUnitL}_L (\text{unit} @ x) = x$ must hold where $\text{elimUnitL}_L :: L () \rightarrow a$ 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 $s \gamma \text{put } x s v = \text{put } x s v$, 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 $L^3 S A$ by the equivalence relation $\equiv$ defined as $x \equiv y$ if $\text{get } x = \text{get } y \land \text{put } x s = \text{put } y s$ for all minimal $s$. This equivalence is preserved by manipulations of $L^3$-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 $x \equiv z$ implies $x \equiv z \equiv w$.

- $x \equiv y$ implies $x \odot \text{tagL} = y \odot \text{tagL}$ (or $x \odot \text{tag2L} = y \odot \text{tag2L}$).

Note that the above three cases cover the only ways to construct/destruct $L^3$ in $f$ when $L^3$ is abstract. The third condition says that this “coarse” equivalence ($\equiv$) on $L^3$ can be “sharpened” to the usual extensional equality ($\Rightarrow$) by $\text{tagL}$ and $\text{tag2L}$ in the unlifting functions.

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

### 4. Going Generic

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

#### 4.1 Unlifting Functions on Lists

We have looked at how unlifting works for n-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 map to a lens function $\text{lift } \ell$, we will obtain a function of type $\text{map } (\text{lift } \ell) :: [L^3 s A] \rightarrow [L^3 s B]$. 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.

\[
\text{isequenceL} :: [L^3 s a] \rightarrow L^3 s [a]
\]

\[
\text{isequenceL} [] = \text{nilL } \text{unit}
\]

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

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

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

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

\[
\text{liftL } :: L [a] b \rightarrow \forall s. [L^3 s a] \rightarrow L^3 s b
\]

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

Tagged lists form an instance of Poset.

**Instance**

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

\[
\text{xs } \gamma \text{ys } \equiv \text{if length } xs \ z z \text{length } ys
\]

\[
\text{then zipWith } (\gamma) \ xs \ ys
\]

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

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 $L^3$ in lifted functions.

The definition of unliftlL 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 $f$.

68
unlift_list :: ∀a b. Eq a ⇒ 
(\vs. [L \; \ell \; s \; a] \rightarrow L' \; \ell \; s \; b) \rightarrow L \; [a] \; b
unlift_list f = L (λs → get (mkLens s) s) 
(λs → put (mkLens s) s)
where
mkLens s = f (projs (length s)) • tagList_L

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 besequence_list and unlift_list as follows.

tailL :: Eq a ⇒ L [a] [a]
tailL = unfoldr (λxs → unfoldr f (xs!!i))

The obtained lens tailL supports all in-place updates, such as put tailL ["a", "b", "c"] ["AA", "BB", "CC"] = ["a", "b", "c"]. In contrast, any change on list length will be rejected; specifically nilL or constL in besequence_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 ["ab", "c"] = "ab\nnc\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(x, t) → s ++ "\n" ++ t)

Here, elemIndices and splitAt are functions from Data.List: elemIndices c s returns the indices of all elements that are equal to c; splitAt i x returns a tuple where the first element is x’s prefix of length i and the second element is the remainder of the list. Intuitively, put 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") "AA\nBB\nCc" = ("AA\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 :: ∀s. [L' s String] → L' s String
unlinesL [] = new **
unlinesL (x : xs) = lift2 catLinel (x, unlines xs)

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 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 empty last 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\n\n" = \n and put unlinesL ["a", "b", "c"] "AA\nBB\n\n" = \n. Lenses like unlinesL allow one to set the view in a way that guarantees correct bidirectional transformation when called from outside.

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

unlines = foldr catLine **

The same coding principle for constructing bidirectional versions applies.

unlinesL :: L [String] String
unlinesL = unlift_list unlinesF

unlinesF :: ∀s. [L' s String] → L' s String
unlinesF [] = foldr (lift2 catLinel) (new **)

The new unlinesF 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-General 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 (contents), and redecorate an “empty” structures with given data (fill).

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

contents :: Traversable t ⇒ t a → [a]

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

2 In GHC, the function contents is called toList, which is defined in Data.Foldable (Every Foldable 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 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 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
(contents (fill t xs) = xs if length xs = length (contents t))

(FillContents)

(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)) → L' s t (t a)
lsequence t = lift (filll _ (shape t)) (lsequence_last (contents t))

where

filll s (lambda t) (contents' 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) ⇒ (∀s. t (L' s)) → L (t a) b
unliftT f = L (λs → get (mkLens s) s)

where

mkLens s = f (projTs (shape s)) ◦ tagT,
tagT_L = L (fmap O (const $ fmap unTag))

projTs sh = let n = length (contents sh)
in fill [projT_L i sh | i ← [0..n - 1]]
projT_L i sh = L (λv → fill sh (update i (U v) (contents s)))

3 This definition actually overlaps with that for pairs. So we either need to have “wrapper” type constructors, or enable OverlappingInstances.

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 | EIrc Exp
    | EVar String | EApp Exp Exp
    | EFun String Exp deriving Eq

data Val a = VNum a
    | VFun String Exp (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 (ENum n) = VNum (new n)
eval env (EIrc 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
                          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.

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 = evalT (λenv → liftT idL, $ 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 @@ twice @@ twice @@ twice @@ inc @@ x

where

twice = EFun "f" $ EFun "x"  
EVar "f" @@ (EVar "f" @@ EVar "x")
x = EVar "x"
inc = EFun "x" $ EIrc (EVar "x")
An advantage of the original lens combinators [9] (that operate
we can run the bidirectional evaluator as follows, with
env we have presented so far is the ability to accept shape changes to
are inserted to the view, making the list lengths different. We can
something that has not been achieved in bidirectional-transformation
research so far.

6. Extensions

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

6.1 Lifting Lens-Combinators

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

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

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

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

As an example, let us consider the following lens combinator
\( \text{mapDefault}_C \).

\[
\text{mapDefault}_C :: a \rightarrow L a b \rightarrow L [a] [b] \\
\text{mapDefault}_C \; d \; \ell = L (\text{map} (\text{get} \; \ell)) \; (\lambda s \rightarrow \text{go} \; s \; v)
\]

where \( \text{go} \; s \; s' = \text{put} \; \ell \; d \; v : \text{go} \; s' \; v \)

When given a lens on elements, \( \text{mapDefault}_C \) 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 \( \text{map} \; (\text{uncurry} \; (+)) \).

\[
\text{mapAdd}_L :: L [(\text{Int}, \text{Int})] \rightarrow L [(\text{Int}, \text{Int})] \\
\text{mapAdd}_L = \text{unlift} \; \text{mapAdd}_F
\]

This lens \( \text{mapAdd}_L \) constructed in our framework handles shape changes without any trouble.

\[
\text{Main} \Rightarrow \text{put} \; \text{mapAdd}_L \; [(1,1), (2,2)] \; [3, 5] \\
\text{Main} \Rightarrow \text{put} \; \text{mapAdd}_L \; [(1,1), (2,2)] \; [3, 5] \\
\text{Main} \Rightarrow \text{put} \; \text{mapAdd}_L \; [(1,1), (2,2)] \; [3, 5, 7] \\
\text{Main} \Rightarrow \text{put} \; \text{mapAdd}_L \; [(1,1), (2,2)] \; [(0,7)]
\]

The trick is that the expression \( \text{map}_F \; (0,0) \; (\text{lift} \; \text{addL}) \)
has type \( \forall s. L' s Int \rightarrow L' s (\text{Int}) \), where the
list occurs inside \( L' s \) and, contrasting to \( \text{map} \; \text{(lift} \; \text{addL})' \)'s type
\( \forall s. L' s (\text{Int}, \text{Int}) \rightarrow L' s \text{Int} \). Intuitively, the type constructor
\( L' s \) can be seen as an updatability annotation; \( L' s (\text{Int}, \text{Int}) \)
means that the list itself is updatable, whereas \( L' s \) 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.

6.2 Observations of Lifted Values

So far we have programmed bidirectional transformations ranging
from polymorphic to monomorphic functions. For example, \( \text{unlines} \)
is monomorphic because its base case returns a String constant,
which is nicely handled in our framework by the function \( \text{new} \). At
the same time, it is also obvious that the creation of constant values is

\[
\text{mapAdd}_F \; x = \text{map}_F \; (0,0) \; (\text{lift} \; \text{addL}) \; x \\
\text{map}_F \; d = \text{lift} \; C \; (\text{mapDefault}_C \; d)
\]

\[
\text{addL} = L (\lambda (x, y) \rightarrow x + y) \; \lambda (x, v) \rightarrow (x, v - x)
\]
not the only cause of a transformation being monomorphic [21, 22]. For example, let us consider the following toy program.

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

In this program, the behavior of the transformation depends on the ‘observation’ made to a value that may potentially be updated in the view. Then the naively obtained lens \(bad_1 = \text{unlift} \circ (\text{lift} \ 2 \ id_L \circ \text{bad})\) 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 put. The same technique can be used in our framework, and actually in a much simpler way due to our new compositional formalization.

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

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

\[
R \ m \gg= f = R \ \xi s \to (x, c_1) = m \ s \quad (y, c_2) = \text{let } R \ k = f \ x \ \text{in} \ k \ s \quad \text{in} \ (y, \xi s \to c_1 \land c_2 \ s)
\]

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

\[
\text{observe :: } Eq \ w \Rightarrow L^T \ s \ w \to R \ s \ w \\
\text{observe } \xi = R \ (\lambda s \to \text{let } w = \text{get } \xi s \ \text{in} \ (w, \lambda s' \to \text{get } \xi s' \ w))
\]

\[
\text{unliftM2 :: } (\text{Eq } a, \text{Eq } b) \Rightarrow (\forall s. (L^T \ s \ a, L^T \ s \ b) \to R \ s \ (L^T \ s \ c)) \to L \ (a, b) \ c
\]

\[
\text{unliftM2 } f = L \ (\lambda s \to \text{get} \ (\text{mkLens } f \ s) \ s) \quad (\lambda s \to \text{put} \ (\text{mkLens } f \ s) \ s)
\]

where

\[
\text{mkLens } f \ s = \text{let } (\ell, p) = \text{let } R \ m = f \ (\text{fst}_L, \text{snd}_L) \quad \text{in} \ m \ (\text{get } \text{tag}_2_L, s) \\
\ell' = \ell \land \text{tag}_2_L \\
\text{put' } s' v = \text{put } s' \ v \quad \text{if } p \ (\text{get } \text{tag}_2_L, s') \text{ then } s' \text{ else } \bot
\]

\text{in } L \ (\text{get } \ell') \ \text{put'}
\]

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

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

\[
\text{good } (x, y) = \text{fmap} \ (\text{lift} 2 \ id_L) \ \text{do} \\
\text{b } \leftarrow \text{liftO2 } (\xi) \ x \ (\text{new } 0) \\
\text{return } \left(\begin{array}{l}
\text{if } b \text{ then } (x, y) \text{ else } (x, \text{new } 1)
\end{array}\right)
\]

Here, \text{liftO2} is defined as follows.

\[\text{liftO2 :: } Eq \ w \Rightarrow (a \to b \to w) \to L^T \ s \ a \to L^T \ s \ b \to R \ s \ w\]

\[
\text{liftO2 } p \ x y = (l \to w) \ (\text{uncurry } p) \ (x \odot y)
\]

\[
\text{liftO } :: Eq \ w \Rightarrow (a \to w) \to L^T \ s \ a \to R \ s \ w
\]

\[
\text{liftO } p \ x = \text{observe} \ (l \ (p \ \text{unused}) \ x)
\]

\text{where unused } s v \ | v \ p = s = 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 \text{good} is in the \text{Monad style}—not applicable 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 \text{get} as a semantic object, applying the function independently to a collection of unique identifiers, and the free theorems arising from parametricity states that whatever happens to those identifiers happens in the same way to any other inputs—this information is sufficient to construct the backward transformation.

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

\[
\text{bf :: } (\forall a. [a] \to [a]) \to (\lambda a \Rightarrow L \ a \ a)
\]

\[
\text{bf } f = \text{unliftL} \ (\lambda s \to \text{list } (\lambda s \to \text{list } f))
\]

Replacing \text{unliftL} and \text{isquenceL} with \text{unliftT} and \text{isquence}, we also obtain the datatype generic version [33].

With the addition of \text{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, \text{liftO2} (and other \(n\)-ary observations-lifting functions) has to be a primitive previously [21, 22], but can now be derived from \text{observe}, \text{lift} and \((\odot)\) 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 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:

\[
\text{forall } f \Rightarrow (B \to f \ B) \to (A \to f \ A)
\]

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

Compared to our approach, the van Laarhoven representation is rather inconvenient for applicative-style programming. It cannot be used to derive a \( \text{put} \) 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 \( f \colon L \ A \ B \), which has \( \text{get} f \colon A \to B \), is represented as a function from some \( B \) structure to some \( A \) structure. This difference in direction poses a significant challenge for higher-order programs, because structures of abstractions and applications are not preserved by inverting the direction of \( \to \). In contrast, our construction of \( \text{put} \) from \( \text{get} \) is straightforward; replacing base type operations with the lifted bidirectional versions is suffice as shown in the \( \text{unlines}_L \) and \( \text{eval}_L \) examples (monadification is only needed when supporting observations). Moreover, the van Laarhoven representation does not extend well to data structures: \( n \)-ary functions in the representation do not correspond to \( n \)-ary lenses. As a result, the van Laarhoven representation itself is not useful to write bidirectional programs such as \( \text{unlines}_L \) and \( \text{eval}_L \). Actually as far as we are aware, higher-order programming with the van Laarhoven representation has not been investigated before.

By using the Yoneda embedding, we can also express \( L \ A \ B \) as functions of type \( \forall V. L \ B \ v \to L \ A \ v \). It is worth mentioning that \( L \ (\_ \to V) \) 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 \( \text{put} \) 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
