ZU064-05-FPR main 22 February 2018 18:16

Applicative Bidirectional Programming

Mixing Lenses and Semantic Bidirectionalization

KAZUTAKA MATSUDA
Graduate School of Information Sciences, Tohoku University
and
MENG WANG
Department of Computer Science, University of Bristol

Abstract

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

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

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* (Bancilhon & Spyratos 1981; Dayal & Bernstein 1982; Fegaras 2010; Hegner 1990) 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 (Hayashi *et al.* 2007; Hu *et al.* 2004; Rajkumar *et al.* 2013). 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 (Xiong *et al.* 2007; Yu *et al.* 2012).

Using terminologies originated from the lens framework (Bohannon *et al.* 2008; Foster *et al.* 2007, 2008), bidirectional transformations, coined *lenses*, can be represented as pairs of functions known as *get* of type $S \to V$ and *put* of type $S \to V \to S$. Function *get* extracts a view from a source, and *put* takes both an updated view and the original source as inputs to produce an updated source. An example definition of a bidirectional transformation in Haskell notation is

$$\mathbf{data}\ \mathit{Lens}\ s\ v = \mathit{Lens}\ \{\mathit{get} :: s \to v, \mathit{put} :: s \to v \to s\}$$
$$\mathit{fst}_{\mathrm{L}} :: \mathit{Lens}\ (a,b)\ a$$
$$\mathit{fst}_{\mathrm{L}} = \mathit{Lens}\ (\lambda(a,\_) \to a)\ (\lambda(\_,b)\ a \to (a,b))$$

A value ℓ of type *Lens 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: $\mathit{get}\ \ell$ has type $s \to v$ and $\mathit{put}\ \ell$ has type $s \to v \to s$. The datatype is used in the definition of $\mathit{fst}_{\mathrm{L}}$ where the first element of a source pair is projected as the view, and may be updated to a new value.

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

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

for all s, s' and v. Note that in this paper, we write $e = e'$ with the assumption that neither e nor e' is undefined. Here **Consistency** (also known as the **PutGet** law (Foster *et al.* 2007)) 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 (Foster *et al.* 2007)), prohibits changes to the source if no update has been made on the view. Collectively, the two laws define *well-behavedness* (Bancilhon & Spyratos 1981; Foster *et al.* 2007; Hegner 1990). A bidirectional transformation *Lens get put* is called *well-behaved* if it satisfies well-behavedness. The above example $\mathit{fst}_{\mathrm{L}}$ 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 (Barbosa *et al.* 2010; Bohannon *et al.* 2008; Foster *et al.* 2007, 2008, 2010; Hidaka *et al.* 2010; Hu *et al.* 2004; Matsuda & Wang 2013, 2014; Matsuda *et al.* 2007; Mu *et al.* 2004; Pacheco *et al.* 2014b; Rajkumar *et al.* 2013; Voigtländer 2009a; Voigtländer *et al.* 2010, 2013; Wang & Najd 2014; Wang *et al.* 2010, 2011, 2013).

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. At the minimum, programmers shall be allowed to treat functions as first-class objects and have them applied explicitly. Moreover, it is beneficial to be able to write bidirectional programs in the same style of their *get*s, because 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 (Barbosa *et al.* 2010; Bohannon *et al.* 2008; Foster *et al.* 2007, 2008, 2010; Hu *et al.* 2004; Mu *et al.* 2004; Pacheco *et al.* 2014b; Rajkumar *et al.* 2013; Wang *et al.* 2010, 2013), 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 (McBride & Paterson 2008; Paterson 2001) that a more convenient programming style does profoundly impact on the popularity of a language.

Research on *bidirectionalization* (Hidaka *et al.* 2010; Matsuda & Wang 2013, 2014; Matsuda *et al.* 2007; Voigtländer 2009a; Voigtländer *et al.* 2010, 2013; Wang & Najd 2014; Wang *et al.* 2010, 2013), which mechanically derives a suitable *put* from an existing *get*, shares the same spirit with us to some extent. The *get*s can be programmed in a unidirectional language and passed in as objects to the bidirectionalization engine, which performs program analysis and then generation of *put*s. 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 *get*s, in an applicative and higher-order programming style.

The key idea of our proposal is to lift lenses of type $\mathit{Lens}\ (A_1,\dots,A_n)\ B$ to *lens functions* of type

$$\forall s.\, L\, s\, A_1 \to \cdots \to L\, s\, A_n \to \ldots \to L\, s\, B$$

where L is a type-constrained version of *Lens* (Sections 2 and 3). The n-ary tuple $(A_1,\dots,A_n)$ above is then generalized to data structures such as lists in Section 4. This function representation of lenses is open to manipulation in an applicative style, and can be passed to higher-order functions directly. For example, we can write a bidirectional version of *unlines*, defined by

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

as below.

$$
\begin{aligned}
&\mathit{unlines}_{\mathrm{F}} :: [L\ s\ \mathit{String}] \to L\ s\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{F}}\ [\,] \qquad\ \ = \mathit{new}\ \texttt{""} \\
&\mathit{unlines}_{\mathrm{F}}\ (x : xs) = \mathsf{lift}_2\ \mathit{catLine}_{\mathrm{L}}\ (x, \mathit{unlines}_{\mathrm{F}}\ xs)
\end{aligned}
$$

where $\mathit{catLine}_{\mathrm{L}}$ is a lens version of $\lambda x\ y \to x +\!\!+ \texttt{"\textbackslash n"} +\!\!+ y$. In the above, except for the noise of *new* and lift_2, the definition is faithful to the original structure of *unlines*' definition, in an applicative style. With the heavy-lifting done in defining the lens function $\mathit{unlines}_{\mathrm{F}}$, a corresponding lens $\mathit{unlines}_{\mathrm{L}} :: \mathit{Lens}\ [\mathit{String}]\ \mathit{String}$ is readily available through straightforward unlifting: $\mathit{unlines}_{\mathrm{L}} = \mathsf{unliftT}\ \mathit{unlines}_{\mathrm{F}}$. In the forward direction, lens $\mathit{unlines}_{\mathrm{L}}$ is the same as the unidirectional function *unlines*:

```
Main> get unlines_L ["a","b","c"]
"a\nb\nc\n"
```

In the backward direction, changes to the list elements in the view are put back to the source:

```
Main> put unlines_L ["a","b","c"] "AA\nBB\nCC\n"
["AA","BB","CC"]
```

With this definition, structural updates (i.e., changes to the length of the view list) are not allowed. For example, *put* $\mathit{unlines}_{\mathrm{L}}$ `["a","b","c"]` `"AA\nBB\n"` and *put* $\mathit{unlines}_{\mathrm{L}}$ `["a","b","c"]` `"AA\nBB\nCC\nDD\n"` result in exceptions. In Section 6, we explain that this restriction on updates is statically reflected in the type of $\mathit{unlines}_{\mathrm{F}}$, and may be relaxed at the cost of the simplicity of the definition.

In Section 5, we demonstrate the expressiveness of our core system through a realistic example (bidirectional evaluator for a higher-order programming language), and then extend the core system in two different dimensions, showing a smooth integration of our framework with both lenses and bidirectionalization approaches in Section 6. We deploy the extended system in the context of XML transformations (Section 7), before proving the correctness theorem (Section 8). We discuss related techniques in Section 9, in particular making connection to semantic bidirectionalization (Matsuda & Wang 2013, 2014; Voigtländer 2009a; Wang & Najd 2014), followed by conclusion in Section 10. An implementation of our idea is available from `https://hackage.haskell.org/package/app-lens`.

Notes on Proofs and Examples. To simplify the formal discussion, we assume that all functions except *put*s are total and no data structure contains $\perp$. To deal with the partiality of *put*s, we assume that a *put* function of type $A \to B \to A$ can be represented as a total function of type $A \to B \to \mathit{Maybe}\ A$, which upon termination will produce either a value *Just a* or an error *Nothing*.

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

A preliminary version of this paper appeared in ICFP'15 (Matsuda & Wang 2015), under the title "Applicative Bidirectional Programming with Lenses". The major differences to the preliminary version include proofs in Section 8 and Appendix A, more detailed discussion to Voigtländer's original bidirectionalization in Section 6.2, and an XML transformation example in Section 7 involving the extensions discussed in Section 6, together with the improvement of overall presentation and correction of technical errors in Section 3.

2 Bidirectional Transformations as Functions

Conventionally, bidirectional transformations are represented directly as pairs of functions (Foster *et al.* 2007; Hegner 1990; Hidaka *et al.* 2010; Hu *et al.* 2004; Matsuda & Wang 2013, 2014; Matsuda *et al.* 2007; Mu *et al.* 2004; Voigtländer 2009a; Voigtländer *et al.* 2010, 2013; Wang & Najd 2014; Wang *et al.* 2010, 2011, 2013) (see the datatype *Lens* defined in Section 1). In this paper, we use lenses to refer specifically to bidirectional transformations in this representation.

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

$$
\begin{array}{l}
(\hat{\circ}) :: \mathit{Lens}\ b\ c \to \mathit{Lens}\ a\ b \to \mathit{Lens}\ a\ c \\
(\mathit{Lens}\ \mathit{get}_2\ \mathit{put}_2)\ \hat{\circ}\ (\mathit{Lens}\ \mathit{get}_1\ \mathit{put}_1) = \\
\quad \mathit{Lens}\ (\mathit{get}_2 \circ \mathit{get}_1)\ (\lambda s\ v \to \mathit{put}_1\ s\ (\mathit{put}_2\ (\mathit{get}_1\ s)\ v))
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{fstTri}_{\mathrm{L}} :: \mathit{Lens}\ ((a,b),c)\ a \\
\mathit{fstTri}_{\mathrm{L}} = \mathit{fst}_{\mathrm{L}}\ \hat{\circ}\ \mathit{fst}_{\mathrm{L}}
\end{array}
$$

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

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

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

This means that the set of (both well-behaved and not-necessarily-well-behaved) lenses forms a category, where objects are types (sets in our setting), and morphisms from A to B are lenses of type $\mathit{Lens}\ A\ B$.

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 (Mac Lane 1998), we lift lenses of type *Lens a b* to polymorphic functions of type

$$\forall s.\,\mathit{Lens}\ s\ a \to \mathit{Lens}\ s\ b$$

by lens composition

$\mathsf{lift} :: \mathit{Lens}\ a\ b \to (\forall s.\mathit{Lens}\ s\ a \to \mathit{Lens}\ s\ b)$
$\mathsf{lift}\ \ell = \lambda x \to \ell \,\hat{\circ}\, x$

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

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

$\mathsf{unlift} :: (\forall s.\mathit{Lens}\ s\ a \to \mathit{Lens}\ s\ b) \to \mathit{Lens}\ a\ b$
$\mathsf{unlift}\ f = f\ \mathit{id}_{\mathrm{L}}$

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

$\mathit{fstTri}_{\mathrm{L}} :: \mathit{Lens}\ ((a,b),c)\ a$
$\mathit{fstTri}_{\mathrm{L}} = \mathsf{unlift}\ (\mathsf{lift}\ \mathit{fst}_{\mathrm{L}} \circ \mathsf{lift}\ \mathit{fst}_{\mathrm{L}})$

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

$\mathit{fstTri}_{\mathrm{L}} = \mathsf{unlift}\ (\lambda x \to \mathit{twice}\ (\mathsf{lift}\ \mathit{fst}_{\mathrm{L}})\ x)$
  where $\mathit{twice}\ f\ x = f\ (f\ x)$

Like many category-theory inspired isomorphisms, this functional representation of bidirectional transformations is not unknown (Ellis 2012); but its formal properties and applications in practical programming have not been investigated before.

2.2 Formal Properties of Lens Functions

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

Proposition 1. $\mathsf{unlift}\ (\mathsf{lift}\ \ell) = \ell$ for all lenses $\ell :: \mathit{Lens}\ A\ B$. □

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

Proposition 2. $\mathsf{lift}\ \ell$ preserves well-behavedness if ℓ is well-behaved.

Proof
Immediate from the fact that $\hat{\circ}$ preserves well-behavedness (Foster *et al.* 2007). □

Proposition 3. $\mathsf{unlift}\ f$ is well-behaved if f preserves well-behavedness. □

As it stands, the type *Lens* is open and it is possible to define lens functions through pattern-matching on the constructor. This becomes a problem when we want to guarantee that $f :: \forall s.\mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B$ preserves well-behavedness. For example, the following f does not preserve well-behavedness.

$f :: \mathit{Lens}\ s\ \mathit{Int} \to \mathit{Lens}\ s\ \mathit{Int}$
$f\ (\mathit{Lens}\ g\ p) = \mathit{Lens}\ g\ (\lambda s\ \_ \to p\ s\ 3)$

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. Moreover, it is worth mentioning that lift is not surjective due to the exposure of *Lens*. The following f' is an example that lift cannot produce, i.e., lift $(\mathsf{unlift}\,f') \not\equiv f'$.

$$f' :: Eq\ a \Rightarrow Lens\ s\ (Maybe\ a) \to Lens\ s\ (Maybe\ a)$$
$$f'\ (Lens\ g\ p) = Lens\ g\ (\lambda s\ v \to \mathbf{if}\ v == g\ s\ \mathbf{then}\ s\ \mathbf{else}\ p\ (p\ s\ Nothing)\ v)$$

For example, for a well-behaved lens

$$\begin{array}{l}
\ell :: Lens\ (Maybe\ (Int,Int))\ (Maybe\ Int) \\
\ell = Lens\ g\ p \\
\quad \mathbf{where}\ g\ Nothing\ = Nothing \\
\qquad\quad g\ (Just\ s)\ \ = Just\ (fst\ s) \\
\qquad\quad p\ Nothing\ Nothing = Nothing \\
\qquad\quad p\ (Just\ s)\ \ (Just\ v)\ = Just\ (v, snd\ s)
\end{array}$$

we have $put\ (f'\ \ell)\ (Just\ (1,2))\ (Just\ 3) = \bot$ while $put\ (\mathsf{lift}\ (\mathsf{unlift} f')\ \ell)\ (Just\ (1,2))\ (Just\ 3) = Just\ (3,2)$.

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 *Lens* is used as an "abstract type" in defining lens functions of type $\forall s. Lens\ s\ A \to Lens\ s\ B$. That is, we require that lens values must be produced and consumed only by using lifted lens functions. This requirement is formally written as follows.

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

$$\begin{array}{l}
\forall \ell. (\forall a\ b. Lens\ a\ b \to (\forall s. \ell\ s\ a \to \ell\ s\ b)) \\
\quad \to (\forall a\ b. (\forall s. \ell\ s\ a \to \ell\ s\ b) \to Lens\ a\ b) \to \tau'
\end{array}$$

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

Essentially, the polymorphic ℓ in h's type prevents us from using the constructor *Lens* directly, while the first functional argument of h (which is lift) provides the (only) means to produce and consume *Lens* values. For example, for a function $\mathsf{lift}\,fst_{\mathrm{L}} :: Lens\ s\ (a,b) \to Lens\ s\ a$, we have a function $h\ lift'\ \_ = lift'\ fst_{\mathrm{L}} :: \forall \ell. (\forall a\ b. Lens\ a\ b \to (\forall s. \ell\ s\ a \to \ell\ s\ b)) \to (\forall a\ b. (\forall s. \ell\ s\ a \to \ell\ s\ b)) \to \ell\ s\ (a,b) \to \ell\ s\ a$ such that $\mathsf{lift}\,fst_{\mathrm{L}} = h$ lift unlift, and thus *Lens* is abstract in $\mathsf{lift}\,fst_{\mathrm{L}}$.

Now the compositional reasoning of well-behavedness extends to lens functions; we can use a logical relation (Reynolds 1983) to characterize well-behavedness for higher-order functions. As an instance, we can state that functions of type $\forall s. Lens\ s\ A \to Lens\ s\ B$ are well-behavedness preserving as follows.

Theorem 1. Let $f :: \forall s. Lens\ s\ A \to Lens\ s\ B$ be a function in which *Lens* is abstract. Suppose that only well-behaved lenses are passed to lift during evaluation. Then, f preserves well-behavedness, and thus unlift f is well-behaved. □

The functions $\mathsf{lift}\,fst_{\mathrm{L}} \circ \mathsf{lift}\,fst_{\mathrm{L}}$ and $twice\ (\mathsf{lift}\,fst_{\mathrm{L}})$ are examples of f in this theorem. Notice that we can use unlift in the definition of f; $\mathsf{lift}\ (\mathsf{unlift}\ (\mathsf{lift}\,fst_{\mathrm{L}}))$ is also a function in which *Lens* is abstract and has a type $\forall s. \mathit{Lens}\ s\ (A,B) \to \mathit{Lens}\ s\ A$. We shall omit the proof of Theorem 1 because it can be proved similarly to Theorem 4 and 6. The condition on lift in Theorem 1, which is also assumed in Theorems 4 and 6, essentially asserts that (the denotation of) lift only takes well-behaved lenses, which will be used in the proof of Theorem 6 in Section 8.

Another consequence of having abstract *Lens* is that lift is now surjective (and unlift is now injective).

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

Although this lemma is key to prove the bijectivity of $\mathsf{lift}/\mathsf{unlift}$ and ensures the naturality of f, which is mentioned in Yoneda lemma (Section 2.4), our system does not rely on the surjectivity of lifting functions for correctness: injectivity alone is sufficient. As a matter of fact, the bijectivity property does not hold when we extend lifting to n-ary lenses in Section 3. Therefore, we delay the largish proof of this lemma to Appendix A, so as not to disrupt the flow of the paper.

Theorem 2. For any $f :: \forall s. \mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B$ in which *Lens* is abstract, $\mathsf{lift}\ (\mathsf{unlift}\ f) = f$ holds. □

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

2.3 Guaranteeing Abstraction

Theorem 1 requires the condition that *Lens* 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 *Lens*.

$$
\begin{array}{l}
\textbf{module}\ \mathit{AbstractLens}\ (\mathit{Lens}_{\mathrm{abs}}, \mathit{lift}_{\mathrm{abs}}, \mathit{unlift}_{\mathrm{abs}})\ \textbf{where} \\
\textbf{newtype}\ \mathit{Lens}_{\mathrm{abs}}\ a\ b = \mathit{Lens}_{\mathrm{abs}}\ \{\mathit{unLens}_{\mathrm{abs}} :: \mathit{Lens}\ a\ b\} \\
\mathit{lift}_{\mathrm{abs}} :: \mathit{Lens}\ a\ b \to (\forall s. \mathit{Lens}_{\mathrm{abs}}\ s\ a \to \mathit{Lens}_{\mathrm{abs}}\ s\ b) \\
\mathit{lift}_{\mathrm{abs}}\ \ell = \lambda x \to \mathit{Lens}_{\mathrm{abs}}\ (\mathsf{lift}\ \ell\ (\mathit{unLens}_{\mathrm{abs}}\ x)) \\
\mathit{unlift}_{\mathrm{abs}} :: (\forall s. \mathit{Lens}_{\mathrm{abs}}\ s\ a \to \mathit{Lens}_{\mathrm{abs}}\ s\ b) \to \mathit{Lens}\ a\ b \\
\mathit{unlift}_{\mathrm{abs}}\ f = \mathsf{unlift}\ (\mathit{unLens}_{\mathrm{abs}} \circ f \circ \mathit{Lens}_{\mathrm{abs}})
\end{array}
$$

Outside the module *AbstractLens*, we can use $\mathit{lift}_{\mathrm{abs}}$, $\mathit{unlift}_{\mathrm{abs}}$ and type $\mathit{Lens}_{\mathrm{abs}}$ itself, but not the constructor of $\mathit{Lens}_{\mathrm{abs}}$. Thus the only way to access data of type *Lens* is through $\mathit{lift}_{\mathrm{abs}}$ and $\mathit{unlift}_{\mathrm{abs}}$.

2.4 Categorical Notes

As mentioned earlier, our idea of mapping $\mathit{Lens}\ A\ B$ to $\forall s. \mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B$ is based on the Yoneda lemma in category theory (Mac Lane 1998, Section III.2). 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 (Mac Lane 1998, Section III.2)). The pair of functions $(\mathsf{lift}, \mathsf{unlift})$ is a bijection between

- $\{\ell :: \mathit{Lens}\ A\ B\}$, and
- $\{f :: \forall s. \mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B \mid f\ x \hat{\circ} y = f\ (x \hat{\circ} y)\}$. □

The condition $f\ x \hat{\circ} y = f\ (x \hat{\circ} y)$ is required to make f a natural transformation between functors $\mathit{Lens}\ (-)\ A$ and $\mathit{Lens}\ (-)\ B$; here, the contravariant functor $\mathit{Lens}\ (-)\ A$ maps a lens ℓ of type $\mathit{Lens}\ Y\ X$ to a function $(\lambda y \to y \hat{\circ} \ell)$ of type $\mathit{Lens}\ X\ A \to \mathit{Lens}\ Y\ A$. Note that $f\ x \hat{\circ} y = f\ (x \hat{\circ} y)$ is equivalent to $f\ x = f\ \mathit{id}_{\mathrm{L}} \hat{\circ} x$. Thus the naturality condition implies Theorem 2 (through Lemma 1), and vice versa. That is, Theorem 3 is nothing but Proposition 1 and Theorem 2 put together.

It sounds contradictory, but there are no higher-order lenses in a categorical sense. Recall that the set of (not-necessarily-well-behaved) lenses forms a category. This category of lenses is monoidal (Hofmann *et al.* 2011), but is believed to be not closed (Rajkumar *et al.* 2013) and have no higher-order lenses. Our discussion does not conflict with this fact. What we state is that, for any s, $(\mathit{Lens}\ s\ A, \mathit{Lens}\ s\ B) \to \mathit{Lens}\ s\ C$ is isomorphic to $\mathit{Lens}\ s\ A \to (\mathit{Lens}\ s\ B \to \mathit{Lens}\ s\ C)$, where s is quantified globally; the standard *curry* and *uncurry* are the required bijections.

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

3 Lifting n-ary Lenses and Flexible Duplication

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

$$\mathit{fstTri}_{\mathrm{L}} :: \mathit{Lens}\ ((a,b),c)\ a$$
$$\mathit{fstTri}_{\mathrm{L}} = \mathsf{unlift}\ (\mathsf{lift}\ \mathit{fst}_{\mathrm{L}} \circ \mathsf{lift}\ \mathit{fst}_{\mathrm{L}})$$

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

$$\mathit{swap}_{\mathrm{L}} :: \mathit{Lens}\ (a,b)\ (b,a)$$
$$\mathit{swap}_{\mathrm{L}} = \ldots$$

Following the programming pattern developed so far, we would like to construct this lens with the familiar unidirectional function $\mathit{swap} :: (a,b) \to (b,a)$. But since lift only produces *unary* functions of type $\forall s. \mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B$, despite the fact that A and B are actually pair types here, there is no way to compose *swap* with the resulting lens function. If we use the intuition developed in Section 2.1 that a lens of type $\mathit{Lens}\ s\ A$ represents an updatable datum of type A, lift treats a pair (indeed any data structure) as a single datum. What we really want here is a pair of functions $\mathsf{lift}_2 :: \mathit{Lens}\ (a,b)\ c \to (\forall s. (\mathit{Lens}\ s\ a, \mathit{Lens}\ s\ b) \to \mathit{Lens}\ s\ c)$ and $\mathsf{unlift}_2 :: (\forall s. (\mathit{Lens}\ s\ a, \mathit{Lens}\ s\ b) \to \mathit{Lens}\ s\ c) \to \mathit{Lens}\ (a,b)\ c$, which are able to go into the pair structure and create separate updatable data that can be manipulated by functions like *swap* as:

$$
\begin{aligned}
&\mathit{swap}_{\mathrm{L}} :: \mathit{Lens}\ (a,b)\ (b,a)\\
&\mathit{swap}_{\mathrm{L}} = \mathsf{unlift}_2\ (\mathsf{lift}_2\ \mathit{id}_{\mathrm{L}} \circ \mathit{swap})
\end{aligned}
$$

In this section, we will see how such a lift_2/unlift_2 pair is defined (with slightly different types for the reason that will be discussed in Section 3.1), and show how the idea of having lift_2/unlift_2 is related to *Applicative* in Haskell (McBride & Paterson 2008; Paterson 2012).

3.1 Caveats of the Duplication Lens

The key of binary lifting is the ability to split a pair and have separate lenses applied to each component. This is achieved via function $(\circledast)$, pronounced “split”.

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

where $(\hat{\otimes})$ is a lens combinator that combines two lenses applying to each component of a pair (Foster *et al.* 2007):

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

With $(\circledast)$ we can define the lifting of binary lenses as below.

$$
\begin{aligned}
&\mathsf{lift}_2 :: \mathit{Lens}\ (a,b)\ c \to (\forall s. (\mathit{Lens}\ s\ a, \mathit{Lens}\ s\ b) \to \mathit{Lens}\ s\ c)\\
&\mathsf{lift}_2\ \ell\ (x,y) = \mathsf{lift}\ \ell\ (x \circledast y)
\end{aligned}
$$

The class constraint $\mathit{Eq}\ s$ in the type of $(\circledast)$ comes from the use of duplication lens $\mathit{dup}_{\mathrm{L}}$ (also known as *copy* elsewhere (Foster *et al.* 2007)) defined as below. For simplicity, we assume that $(==)$ represents observational equivalence.

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

Despite being fitting type-wise, this definition of $\mathit{dup}_{\mathrm{L}}$ causes a serious execution issue. We would like to use the following definition as lift_2’s left inverse.

$$
\begin{aligned}
&\mathsf{unlift}_2 :: (\forall s. (\mathit{Lens}\ s\ a, \mathit{Lens}\ s\ b) \to \mathit{Lens}\ s\ c) \to \mathit{Lens}\ (a,b)\ c\\
&\mathsf{unlift}_2\ f = f\ (\mathit{fst}_{\mathrm{L}}, \mathit{snd}_{\mathrm{L}})
\end{aligned}
$$

But $\mathsf{unlift}_2 \circ \mathsf{lift}_2$ does not result in identity:

$$
\begin{aligned}
&\quad (\mathsf{unlift}_2 \circ \mathsf{lift}_2)\ \ell\\
&= \{\ \text{definition unfolding \& } \beta\text{-reduction}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \ell \mathbin{\hat{\circ}} (fst_{\mathrm{L}} \circledast snd_{\mathrm{L}}) \\
= {} & \{ \text{ unfolding } (\circledast) \ \} \\
& \ell \mathbin{\hat{\circ}} (fst_{\mathrm{L}} \mathbin{\hat{\otimes}} snd_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}} \\
= {} & \{ \text{ definition unfolding } \} \\
& \ell \mathbin{\hat{\circ}} block_{\mathrm{L}} \ \mathbf{where} \\
& \quad block_{\mathrm{L}} = Lens\ id\ (\lambda s\ v \to \mathbf{if}\ s == v\ \mathbf{then}\ v\ \mathbf{else}\ \bot)
\end{aligned}
$$

Lens $block_{\mathrm{L}}$ is not a useful lens because it blocks any update to the view. Consequently any lenses composed with it become useless too. The reason for the failure is that dup_{L} demands the duplicated copies to remain equal amid updates, which will not hold because the purpose of the duplication is to create separate updatable data.

3.2 Flexible and Safe Duplication by Tagging

If we look at the lens dup_{L} in isolation, there seems to be no way out. The two duplicated values have to remain equal for the bidirectional laws to hold. However, if we consider the context in which dup_{L} is applied, there is more room for maneuver. Let us consider the lifting function lift_2 again, and how $put\ dup_{\mathrm{L}}$, which rejects the update above, works in the execution of $put\ (\mathsf{unlift}_2\ (\mathsf{lift}_2\ id_{\mathrm{L}}))$.

$$
\begin{aligned}
& put\ (\mathsf{unlift}_2\ (\mathsf{lift}_2\ id_{\mathrm{L}}))\ (1,2)\ (\underline{3},\underline{4}) \\
= {} & \{ \text{ simplification } \} \\
& put\ ((fst_{\mathrm{L}} \mathbin{\hat{\otimes}} snd_{\mathrm{L}}) \mathbin{\hat{\circ}} dup_{\mathrm{L}})\ (1,2)\ (\underline{3},\underline{4}) \\
= {} & \{ \text{ definition unfolding \& } \beta\text{-reduction } \} \\
& put\ dup_{\mathrm{L}}\ (1,2)\ (put\ fst_{\mathrm{L}}\ (1,2)\ \underline{3}, put\ snd_{\mathrm{L}}\ (1,2)\ \underline{4}) \\
= {} & \{ \ \beta\text{-reduction } \} \\
& put\ dup_{\mathrm{L}}\ (1,2)\ ((\underline{3},2),(1,\underline{4}))
\end{aligned}
$$

The last call to $put\ dup_{\mathrm{L}}$ above will fail because $(\underline{3},2) \not\equiv (1,\underline{4})$. But if we look more carefully, there is no reason for this behavior: $\mathsf{lift}_2\ id_{\mathrm{L}}$ should be able to update the two elements of the pair independently. Indeed in the *put* execution above, relevant values to the view change as highlighted by underlining are only compared for equality with irrelevant values. That is to say, we should be able to relax the equality check in dup_{L} and update the old source $(1,2)$ to $(\underline{3},\underline{4})$ without violating bidirectional laws.

To achieve this, we tag the values according to their relevance to view updates (Mu *et al.* 2004).

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

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

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

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

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

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

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Poset}\ (\mathit{Tag}\ a)\ \textbf{where} \\
\quad (O\ \_) \curlyvee (U\ t) \qquad\quad = U\ t \\
\quad (U\ s) \curlyvee (O\ \_) \qquad\quad = U\ s \\
\quad (O\ s) \curlyvee (O\ t) \mid s == t = O\ s \quad \text{-- } \textit{The check } s == t \textit{ never fails.} \\
\quad (U\ s) \curlyvee (U\ t) \mid s == t = U\ s \quad \text{-- } \textit{In contrast, this check can fail.} \\
\textbf{instance}\ (\mathit{Poset}\ a, \mathit{Poset}\ b) \Rightarrow \mathit{Poset}\ (a,b)\ \textbf{where} \\
\quad (a,b) \curlyvee (a',b') = (a \curlyvee a', b \curlyvee b')
\end{array}
$$

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

$\textbf{type}\ L\ s\ a = \mathit{Poset}\ s \Rightarrow \mathit{Lens}\ s\ a$

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

Accordingly, we change the types of $(\circledast)$, lift and lift_2 as below (notice that due to the change of $\mathit{dup}_{\mathrm{L}}$ the behavior of lift_2 is changed accordingly)

$$
\begin{array}{l}
(\circledast) :: L\ s\ a \to L\ s\ b \to L\ s\ (a,b) \\
\mathsf{lift} :: \mathit{Lens}\ a\ b \to (\forall s. L\ s\ a \to L\ s\ b) \\
\mathsf{lift}_2 :: \mathit{Lens}\ (a,b)\ c \to (\forall s. (L\ s\ a, L\ s\ b) \to L\ s\ c)
\end{array}
$$

and adapt the definitions of unlift and unlift_2 to properly handle the newly introduced tags.

$$
\begin{array}{l}
\mathsf{unlift} :: \mathit{Eq}\ a \Rightarrow (\forall s. L\ s\ a \to L\ s\ b) \to \mathit{Lens}\ a\ b \\
\mathsf{unlift}\ f = f\ \mathit{id}'_{\mathrm{L}} \mathbin{\hat{\circ}} \mathit{tag}_{\mathrm{L}} \\
\mathit{id}'_{\mathrm{L}}\ :: L\ (\mathit{Tag}\ a)\ a \\
\mathit{id}'_{\mathrm{L}}\ = \mathit{Lens}\ \mathit{unTag}\ (\mathit{const}\ U) \\
\mathit{tag}_{\mathrm{L}} :: \mathit{Lens}\ a\ (\mathit{Tag}\ a) \\
\mathit{tag}_{\mathrm{L}} = \mathit{Lens}\ O\ (\mathit{const}\ \mathit{unTag}) \\
\\
\mathsf{unlift}_2 :: (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow (\forall s. (L\ s\ a, L\ s\ b) \to L\ s\ c) \to \mathit{Lens}\ (a,b)\ c \\
\mathsf{unlift}_2\ f = f\ (\mathit{fst}'_{\mathrm{L}}, \mathit{snd}'_{\mathrm{L}}) \mathbin{\hat{\circ}} \mathit{tag2}_{\mathrm{L}} \\
\mathit{fst}'_{\mathrm{L}} :: L\ (\mathit{Tag}\ a, \mathit{Tag}\ b)\ a \\
\mathit{fst}'_{\mathrm{L}} = \mathit{Lens}\ (\lambda(a,\_) \to \mathit{unTag}\ a)\ (\lambda(\_,b)\ a \to (U\ a, b)) \\
\mathit{snd}'_{\mathrm{L}} :: L\ (\mathit{Tag}\ a, \mathit{Tag}\ b)\ b \\
\mathit{snd}'_{\mathrm{L}} = \mathit{Lens}\ (\lambda(\_,b) \to \mathit{unTag}\ b)\ (\lambda(a,\_)\ b \to (a, U\ b)) \\
\mathit{tag2}_{\mathrm{L}} :: \mathit{Lens}\ (a,b)\ (\mathit{Tag}\ a, \mathit{Tag}\ b) \\
\mathit{tag2}_{\mathrm{L}} = \mathit{tag}_{\mathrm{L}} \mathbin{\hat{\otimes}} \mathit{tag}_{\mathrm{L}}
\end{array}
$$

We need to change unlift, though no duplication is needed in the unary case, because the function may be applied to functions calling lift_2 internally. The definitions are a bit

[1] Actually, we will have to use **newtype** for the code in this paper to pass GHC 7.8.3's type checking. We take a small deviation from GHC Haskell here in favor of brevity.

involved; but a key property is that tags are automatically introduced and eliminated by unlifts, an internal mechanism that is completely invisible to programmers.

We can now show that the new unlift_2 is the left-inverse of lift_2 (a similar property holds for lift/unlift); notice that, as we will discuss later, lift_2 is *not* a left-inverse of unlift_2 in contrast.

Proposition 4. $\mathsf{unlift}_2\ (\mathsf{lift}_2\ \ell) = \ell$ holds for all lenses $\ell :: \mathit{Lens}\ (A,B)\ C$.

Proof
We prove the statement with the following calculation.

$$
\begin{array}{ll}
& \mathsf{unlift}_2\ (\mathsf{lift}_2\ \ell) \\
= & \{ \text{ definition unfolding \& } \beta\text{-reduction } \} \\
& \ell \,\hat{\circ}\, (\mathit{fst}'_{\mathrm{L}} \circledast \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}} \\
= & \{ \text{ unfolding } (\circledast) \ \} \\
& \ell \,\hat{\circ}\, (\mathit{fst}'_{\mathrm{L}} \,\hat{\otimes}\, \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{dup}_{\mathrm{L}} \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}} \\
= & \{\ (\mathit{fst}'_{\mathrm{L}} \,\hat{\otimes}\, \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{dup}_{\mathrm{L}} \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}} = \mathit{id}_{\mathrm{L}} \text{ — (1) } \} \\
& \ell
\end{array}
$$

We prove the statement labeled (1) by showing $\mathit{get}\ ((\mathit{fst}'_{\mathrm{L}} \,\hat{\otimes}\, \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{dup}_{\mathrm{L}} \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}})\ (a,b) = (a,b)$ and $\mathit{put}\ ((\mathit{fst}'_{\mathrm{L}} \,\hat{\otimes}\, \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{dup}_{\mathrm{L}} \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}})\ (a,b)\ (a',b') = (a',b')$. Since the former property is easy to prove, we only show the latter here.

$$
\begin{array}{ll}
& \mathit{put}\ ((\mathit{fst}'_{\mathrm{L}} \,\hat{\otimes}\, \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{dup}_{\mathrm{L}} \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}})\ (a,b)\ (a',b') \\
= & \{ \text{ definition unfolding \& } \beta\text{-reduction } \} \\
& \mathit{put}\ \mathit{tag2}_{\mathrm{L}}\ (a,b)\ \$\ \mathit{put}\ ((\mathit{fst}'_{\mathrm{L}} \,\hat{\otimes}\, \mathit{snd}'_{\mathrm{L}}) \,\hat{\circ}\, \mathit{dup}_{\mathrm{L}})\ (O\ a, O\ b)\ (a',b') \\
= & \{ \text{ definition unfolding \& } \beta\text{-reduction } \} \\
& \mathit{put}\ \mathit{tag2}_{\mathrm{L}}\ (a,b)\ \$\ \mathit{put}\ \mathit{dup}_{\mathrm{L}}\ (O\ a, O\ b)\ \$\ (\mathit{put}\ \mathit{fst}'_{\mathrm{L}}\ (O\ a, O\ b)\ a', \mathit{put}\ \mathit{snd}'_{\mathrm{L}}\ (O\ a, O\ b)\ b') \\
= & \{ \text{ definitions of } \mathit{fst}'_{\mathrm{L}} \text{ and } \mathit{snd}'_{\mathrm{L}} \ \} \\
& \mathit{put}\ \mathit{tag2}_{\mathrm{L}}\ (a,b)\ \$\ \mathit{put}\ \mathit{dup}_{\mathrm{L}}\ (O\ a, O\ b)\ ((U\ a', O\ b), (O\ a, U\ b')) \\
= & \{ \text{ definition of } \mathit{dup}_{\mathrm{L}} \ \} \\
& \mathit{put}\ \mathit{tag2}_{\mathrm{L}}\ (a,b)\ (U\ a', U\ b') \\
= & \{ \text{ definition of } \mathit{tag2}_{\mathrm{L}} \ \} \\
& (\mathit{put}\ \mathit{tag}_{\mathrm{L}}\ a\ (U\ a'), \mathit{put}\ \mathit{tag}_{\mathrm{L}}\ b\ (U\ b')) \\
= & \{ \text{ definition of } \mathit{tag}_{\mathrm{L}} \ \} \\
& (a',b')
\end{array}
$$

Thus, we have proved that lift_2 is injective. □

We can recreate $\mathit{fst}_{\mathrm{L}}$ and $\mathit{snd}_{\mathrm{L}}$ with unlift_2, which is rather reassuring.

Proposition 5. $\mathit{fst}_{\mathrm{L}} = \mathsf{unlift}_2\,\mathit{fst}$ and $\mathit{snd}_{\mathrm{L}} = \mathsf{unlift}_2\ \mathit{snd}$. □

Example 1 (swap). Lens $\mathit{swap}_{\mathrm{L}}$ as seen in the beginning of this section can be defined as follows

$$
\begin{array}{l}
\mathit{swap}_{\mathrm{L}} :: (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow \mathit{Lens}\ (a,b)\ (b,a) \\
\mathit{swap}_{\mathrm{L}} = \mathsf{unlift}_2\ (\mathsf{lift}_2\ \mathit{id}_{\mathrm{L}} \circ \mathit{swap})
\end{array}
$$

and it behaves as expected.

$$
\begin{aligned}
& \quad \mathit{put}\ \mathit{swap}_{\mathrm{L}}\ (1,2)\ (4,3) \\
= & \ \{ \text{ unfold definitions } \} \\
& \quad \mathit{put}\ ((\mathit{snd}'_{\mathrm{L}}\ \hat{\otimes}\ \mathit{fst}'_{\mathrm{L}})\ \hat{\circ}\ \mathit{dup}_{\mathrm{L}}\ \hat{\circ}\ \mathit{tag2}_{\mathrm{L}})\ (1,2)\ (4,3) \\
= & \ \{ \text{ simplifications } \} \\
& \quad \mathit{put}\ \mathit{tag2}_{\mathrm{L}}\ (1,2)\ \$\ \mathit{put}\ \mathit{dup}_{\mathrm{L}}\ (O\ 1, O\ 2)\ \$\ (\mathit{put}\ \mathit{snd}'_{\mathrm{L}}\ (O\ 1, O\ 2)\ 4, \mathit{put}\ \mathit{fst}'_{\mathrm{L}}\ (O\ 1, O\ 2)\ 3) \\
= & \ \{ \text{ definition of } \mathit{fst}'_{\mathrm{L}} \text{ and } \mathit{snd}'_{\mathrm{L}} \} \\
& \quad \mathit{put}\ \mathit{tag2}_{\mathrm{L}}\ (1,2)\ \$\ \mathit{put}\ \mathit{dup}_{\mathrm{L}}\ (O\ 1, O\ 2)\ ((O\ 1, U\ 4), (U\ 3, O\ 2)) \\
= & \ \{ \text{ definitions of } \mathit{dup}_{\mathrm{L}} \text{ and } \mathit{tag2}_{\mathrm{L}} \} \\
& \quad (3,4) \qquad \square
\end{aligned}
$$

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

$$
\begin{aligned}
& \mathsf{split}_3 :: (L\ s\ a, L\ s\ b, L\ s\ c) \to L\ s\ (a,b,c) \\
& \mathsf{split}_3\ (x,y,z) = \mathsf{lift}\ \mathit{flattenL}_{\mathrm{L}}\ ((x \circledast y) \circledast z) \\
& \quad \mathbf{where}\ \mathit{flattenL}_{\mathrm{L}} :: \mathit{Lens}\ ((a,b),c)\ (a,b,c) \\
& \qquad\quad \mathit{flattenL}_{\mathrm{L}} = \mathit{Lens}\ (\lambda((x,y),z) \to (x,y,z)) \\
& \qquad\qquad\qquad\qquad (\lambda\_\ (x,y,z) \to ((x,y),z)) \\
& \mathsf{lift}_3\ \ell\ t = \mathsf{lift}\ \ell\ (\mathsf{split}_3\ t)
\end{aligned}
$$

For unlifts, we additionally need n-ary versions of projection and tagging functions. But they are straightforward to define.

In the above definition of split_3, we have decided to nest to the left in the intermediate step. This choice is not essential.

$$
\begin{aligned}
& \mathsf{split}'_3\ (x,y,z) = \mathsf{lift}\ \mathit{flattenR}_{\mathrm{L}}\ (x \circledast (y \circledast z)) \\
& \quad \mathbf{where}\ \mathit{flattenR}_{\mathrm{L}} :: \mathit{Lens}\ (a,(b,c))\ (a,b,c) \\
& \qquad\quad \mathit{flattenR}_{\mathrm{L}} = \mathit{Lens}\ (\lambda(x,(y,z)) \to (x,y,z)) \\
& \qquad\qquad\qquad\qquad (\lambda\_\ (x,y,z) \to (x,(y,z)))
\end{aligned}
$$

The two definitions split_3 and split'_3 coincide. That is, ($\circledast$) is associative up to isomorphism.

To complete the picture, the nullary lens function

$$
\begin{aligned}
& \mathsf{unit} :: \forall s. L\ s\ () \\
& \mathsf{unit} = \mathit{Lens}\ (\lambda\_ \to ())\ (\lambda s\ () \to s)
\end{aligned}
$$

is the unit for ($\circledast$). Theoretically $(L\ s\ (-), \circledast, \mathsf{unit})$ forms a lax monoidal functor (Mac Lane 1998, Section XI.2) under certain conditions (see Section 3.4). Practically, unit enables us to define the following combinator.

$$
\begin{aligned}
& \mathit{new} :: \mathit{Eq}\ a \Rightarrow a \to \forall s. L\ s\ a \\
& \mathit{new}\ a = \mathsf{lift}\ (\mathit{Lens}\ (\mathit{const}\ a)\ (\lambda\_\ a' \to \mathit{check}\ a\ a'))\ \mathsf{unit} \\
& \quad \mathbf{where} \\
& \qquad \mathit{check}\ a\ a' = \mathbf{if}\ a == a'\ \mathbf{then}\ () \\
& \qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{error}\ \texttt{"Update on constant"}
\end{aligned}
$$

Function *new* lifts ordinary values into the bidirectional transformation system; but since the values are not from any source, they are not updatable. Nevertheless, this ability to lift constant values is very useful in practice (Matsuda & Wang 2013, 2014), as we will see in the examples to come.

Note that now unlifts are *no longer* injective (even with abstract *Lens*); there exist functions that are not equivalent but coincide after unlifting. An example of such is the pair $\mathsf{lift}_2\, \mathit{fst}_\mathrm{L}$ and fst: while unlifting both functions results in fst_L, they actually differ as $\mathit{put}\ (\mathsf{lift}_2\, \mathit{fst}_\mathrm{L}\ (\mathit{fst}'_\mathrm{L}, \mathit{snd}'_\mathrm{L}))\ (O\ a, O\ b)\ c = (U\ c, U\ b)$ and $\mathit{put}\ (\mathit{fst}\ (\mathit{fst}'_\mathrm{L}, \mathit{snd}'_\mathrm{L}))\ (O\ a, O\ b)\ c = (U\ c, O\ b)$. Intuitively, fst knows that the second argument is unused, while $\mathsf{lift}_2\, \mathit{fst}_\mathrm{L}$ does not because fst_L is treated as a black box by lift_2. 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 lift_2 and unlift_2 by the new internal functions fst'_L, snd'_L, dup_L and $\mathit{tag2}_\mathrm{L}$. Consequently, we need a new theoretical development to establish the preservation of well-behavedness by the lifting/unlifting process.

3.3 Relevance-Aware Well-Behavedness

We have noted that the new internal functions dup_L, fst'_L, snd'_L and $\mathit{tag2}_\mathrm{L}$ are not well-behaved, for different reasons. For functions fst'_L and snd'_L, the difference from the original versions fst_L and snd_L is only in the additional wrapping/unwrapping that is required due to the introduction of tags. As a result, as long as these functions are used in an appropriate context, the bidirectional laws are expected to hold. But for dup_L and $\mathit{tag2}_\mathrm{L}$, the new definitions are more defined in the sense that some originally failing executions of *put* are now intentionally turned into successful ones. For this change in semantics, we need to adapt the laws to allow temporary violations and yet still establish well-behavedness of the resulting bidirectional transformations in the end. For example, we still want $\mathsf{unlift}_2\, f$ to be well-behaved for any $f :: \forall s. (L\ s\ A, L\ s\ B) \to L\ s\ C$, as long as the lifting functions are applied to well-behaved lenses.

3.3.1 Relevance-Ordering and Lawful Duplications

Central to the discussion in this and the previous subsections is the behavior of dup_L. To maintain safety, unequal values as duplications are only allowed if they have different tags (i.e., one value must be irrelevant to the update and can be discarded). We formalize such a property with the partial ordering between tagged values. Let us write $(\preceq)$ for the partial order induced from $\curlyvee$: that is, $s \preceq t$ if $s \curlyvee t$ is defined and equal to t. One can see that $(\preceq)$ is the reflexive closure of $O\ s \preceq U\ t$. The definition of $(\preceq)$ is extended to (n-ary) containers element-wise; for example, $(s_1, s_2) \preceq (t_1, t_2)$ if and only if $s_1 \preceq t_1$ and $s_2 \preceq t_2$. Nesting of tags is not allowed. We write $\uparrow\! s$ for a value obtained from s by replacing all O tags with U tags. Trivially, we have $s \preceq \uparrow\! s$. But there exists s' such that $s \preceq s'$ and $s' \neq \uparrow\! s$, unless s contains only U tags.

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

Definition 2 (Local Well-Behavedness). Let A be a type associated with $(\preceq)$. A bidirectional transformation $\ell :: L\,A\,B$ is called *locally well-behaved* if the following four conditions hold.

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

In the above, tags introduced for the flexible behavior of *put* must not affect the behavior of *get*: $\uparrow s' = \uparrow s$ means that s and s' are equal if tags are ignored. The property backward inflation states that *put* puts U-tags to all the updated elements, and thus O-tagged elements in the *put* result are kept unchanged, which will be used to show the naturality of $(\circledast)$ in Section 3.4. The property local-acceptability is similar to acceptability, except that O-tags are allowed to change to U-tags. The property local consistency is stronger than consistency in the sense that *get* must map all values sharing the same U-tagged elements with s' to the same view. The idea is that O-tagged elements in s' are not connected to the view v, and thus changing them will not affect v. In this sense, O-tagged values must not be relevant to the view. A similar reasoning applies to backward inflation stating that source elements changed by *put* will have U-tags. Note that in this definition of local well-behavedness, tags are assumed to appear only in the sources. As a matter of fact, only dup_{L} and $tag2_{\mathrm{L}}/tag_{\mathrm{L}}$ introduce tagged views (and actually they are not locally well-behaved); but they are always precomposed when used, as shown in the definitions of lift_2 and unlift_2.

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

Proof

We only prove the second and third properties because it is straightforward to prove the first property.

The second property. We first show local acceptability.

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

Note that, since $s' \preceq \uparrow s$ and $s'' \preceq \uparrow s$, it follows that $s' \curlyvee s'' \preceq \uparrow s$.

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

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

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

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

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

Forward tag-irrelevance and backward inflation are straightforward.

The third property. First, we prove acceptability.

$$\begin{array}{ll}
& put\ (x \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (s_1, s_2)\ (get\ (x \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (s_1, s_2)) \\
= & \{ \text{ unfolding } \hat{\circ} \ \} \\
& put\ tag2_{\mathrm{L}}\ (s_1, s_2)\ (put\ x\ (get\ tag2_{\mathrm{L}}\ (s_1, s_2))\ (get\ x\ (get\ tag2_{\mathrm{L}}\ (s_1, s_2)))) \\
= & \{ \text{ unfolding the definition of } get\ tag2_{\mathrm{L}} \ \} \\
& put\ tag2_{\mathrm{L}}\ (s_1, s_2)\ (put\ x\ (O\ s_1, O\ s_2)\ (get\ x\ (O\ s_1, O\ s_2))) \\
= & \{ \text{ by the local acceptability of } x \ \} \\
& put\ tag2_{\mathrm{L}}\ (s_1, s_2)\ (tag\ s_1, tag\ s_2)\ \textbf{where}\ tag = O \vee tag = U \\
= & \{ \text{ unfolding the definition of } put\ tag2_{\mathrm{L}} \ \} \\
& (s_1, s_2)
\end{array}$$

The proof of the acceptability of $y \mathbin{\hat{\circ}} tag_{\mathrm{L}}$ is similar.

Next, we prove consistency. Assume that $(s'_1, s'_2) = put\ (x \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (s_1, s_2)\ v$. Then, it must be the case when there are tag_1 and tag_2 such that $(tag_1\ s'_1, tag_2\ s'_2) = put\ x\ (O\ s_1, O\ s_2)\ v$ where tag_i is either O or U for $i = 1, 2$. Here, we have $(O\ s'_1, O\ s'_2) \preceq (tag_1\ s'_1, tag_2\ s'_2)$. Then, we have:

$$\begin{array}{ll}
& get\ (x \mathbin{\hat{\circ}} tag2_{\mathrm{L}})\ (s'_1, s'_2) \\
= & \{ \text{ unfolding } \hat{\circ} \ \} \\
& get\ x\ (get\ tag2_{\mathrm{L}}\ (s'_1, s'_2)) \\
= & \{ \text{ unfolding the definition of } get\ tag2_{\mathrm{L}}\ (s'_1, s'_2) \ \} \\
& get\ x\ (O\ s'_1, O\ s'_2) \\
= & \{ \text{ the forward tag-irrelevance of } x \ \} \\
& get\ x\ (tag_1\ s'_1, tag_2\ s'_2) \\
= & \{ \text{ the local consistency of } x \text{ and } (tag_1\ s'_1, tag_2\ s'_2) = put\ x\ (O\ s_1, O\ s_2)\ v \ \} \\
& v
\end{array}$$

The proof of the consistency of $y \mathbin{\hat{\circ}} \mathit{tag}_{\mathrm{L}}$ is similar. □

Corollary 1. The following properties hold.

- $\mathsf{lift}\ \ell :: \forall s. L\ s\ A \to L\ s\ B$ preserves local well-behavedness, if $\ell :: \mathit{Lens}\ A\ B$ is well-behaved.
- $\mathsf{lift}_2\ \ell :: \forall s. (L\ s\ A, L\ s\ B) \to L\ s\ C$ preserves local well-behavedness, if $\ell :: \mathit{Lens}\ (A,B)\ C$ is well-behaved. □

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

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

$$\begin{aligned}&\forall \ell. (\forall a\ b. \mathit{Lens}\ a\ b \to (\forall s. \ell\ s\ a \to \ell\ s\ b))\\ &\quad \to (\forall a\ b. (\forall s. \ell\ s\ a \to \ell\ s\ b) \to \mathit{Lens}\ a\ b)\\ &\quad \to (\forall s. \ell\ s\ ())\\ &\quad \to (\forall s\ a\ b. \ell\ s\ a \to \ell\ s\ b \to \ell\ s\ (a,b))\\ &\quad \to (\forall a\ b\ c. (\forall s. (\ell\ s\ a, \ell\ s\ b) \to \ell\ s\ c) \to \mathit{Lens}\ (a,b)\ c)\\ &\quad \to \tau'\end{aligned}$$

satisfying $f = h\ \mathsf{lift}\ \mathsf{unlift}\ \mathsf{unit}\ (\circledast)\ \mathsf{unlift}_2$ and $\tau' = \tau[\ell/L]$. □

Then, we obtain the following properties from the free theorems (Voigtländer 2009b; Wadler 1989).

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

We omit the proof because we will prove the more involved version, Theorem 6, in Section 8.

Proposition 6. $\mathit{fst}'_{\mathrm{L}}$ and $\mathit{snd}'_{\mathrm{L}}$ are locally well-behaved. □

Corollary 2. Let f be a function of type $\forall s. (L\ s\ A, L\ s\ B) \to L\ s\ C$ in which L is abstract. Then, $\mathsf{unlift}_2\ f$ is well-behaved, assuming that only well-behaved lenses are passed to lift during evaluation.

3.4 Categorical Notes

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

- $(\circledast)$ must be natural, i.e., $(\mathsf{lift}\ f\ x) \circledast (\mathsf{lift}\ g\ y) = \mathsf{lift}\ (f \mathbin{\hat{\otimes}} g)\ (x \circledast y)$ for all f, g, x and y with appropriate types.
- split_3 and split'_3 coincide.
- $\mathsf{lift}\ \mathit{elimUnitL}_{\mathrm{L}}\ (\mathsf{unit} \circledast x) = x$ must hold where $\mathit{elimUnitL}_{\mathrm{L}} :: \mathit{Lens}\ ((),a)\ a$ is the bidirectional version of elimination of $()$, and so does its symmetric version.

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

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

Note that the above three cases cover the only ways to construct/destruct L in f when L is abstract. The third condition says that this "coarse" equivalence ($\equiv$) on L can be "sharpened" to the usual extensional equality ($=$) by tag_{L} and $tag2_{\mathrm{L}}$ in the unlifting functions. Thus, quotienting L with $\equiv$, the three conditions hold, and thus we have the following theorem.

Theorem 5. $(L\ S\ (-), \circledast, \mathsf{unit})$ forms a lax monoidal functor. □

The fact that our framework forms a lax monoidal functor may suggest a connection to Haskell's *Applicative* class (McBride & Paterson 2008; Paterson 2012), which shares the same mathematical structure. It is known that *Applicative* is exactly an endo lax monoidal functor (with strength) on the category of Haskell functions (Paterson 2012). However, it is not possible to structure our code with the *Applicative* class, because our functor is not endo and there are (believed to be) no exponentials in the category of lenses (Rajkumar *et al.* 2013). Nevertheless, one may consider the following classes similar to those in Rajkumar *et al.* (2013) (unlike their type classes we consider covariant functors instead of contravariant ones).

```
class LFunctor f where
  lift :: Lens a b → (f a → f b)
class LFunctor f ⇒ LMonoidal f where
  unit :: f ()
  (⊛) :: f a → f b → f (a,b)
```

The laws for *LMonoidal* have already been discussed in Section 3.4. These classes have the following instance declarations.

```
instance LFunctor (Lens s) where
  lift ℓ x = ℓ ô x
instance Poset s ⇒ LMonoidal (Lens s) where
  unit  = Lens (λ_ → ()) (λs () → s)
  x ⊛ y = (x ⊗̂ y) ô dup_L
```

We then can define lift_2 as:

$$\mathsf{lift}_2 :: \mathit{Lens}\ (a,b)\ c \to (\forall f.\, \mathit{LMonoidal}\, f \Rightarrow (f\ a, f\ b) \to f\ c)$$
$$\mathsf{lift}_2\ \ell = \lambda (x,y) \to \mathsf{lift}\ \ell\ (x \circledast y)$$

Now unlift and unlift_2 have the types $\mathit{Eq}\ a \Rightarrow (\forall f.\, \mathit{LMonoidal}\, f \Rightarrow f\ a \to f\ b) \to \mathit{Lens}\ a\ b$ and $(\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow (\forall f.\, \mathit{LMonoidal}\, f \Rightarrow (f\ a, f\ b) \to f\ c) \to \mathit{Lens}\ (a,b)\ c$, respectively, while their implementations are kept unchanged. Haskell programmers may prefer this class-based interface, but it is more of a matter of taste.

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, when we apply *map* to a lens function $\mathsf{lift}\ \ell$, we will obtain a function of type $\mathit{map}\ (\mathsf{lift}\ \ell) :: [L\ s\ A] \to [L\ s\ B]$. But what we really want is a lens of type $\mathit{Lens}\ [A]\ [B]$. 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.

$$\begin{array}{l}
\mathit{lsequence}_{\mathrm{list}} :: [L\ s\ a] \to L\ s\ [a] \\
\mathit{lsequence}_{\mathrm{list}}\ [\,] \qquad\quad = \mathsf{lift}\ \mathit{nil}_{\mathrm{L}}\ \mathsf{unit} \\
\mathit{lsequence}_{\mathrm{list}}\ (x : xs) = \mathsf{lift}_2\ \mathit{cons}_{\mathrm{L}}\ (x, \mathit{lsequence}_{\mathrm{list}}\ xs) \\
\mathit{nil}_{\mathrm{L}} = \mathit{Lens}\ (\lambda () \to [\,])\ (\lambda ()\ [\,] \to ()) \\
\mathit{cons}_{\mathrm{L}} = \mathit{Lens}\ (\lambda (a, as) \to (a : as))\ (\lambda _\ (a' : as') \to (a', as'))
\end{array}$$

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

$$\begin{array}{l}
\mathsf{lift}_{\mathrm{list}} :: \mathit{Lens}\ [a]\ b \to \forall s.\, [L\ s\ a] \to L\ s\ b \\
\mathsf{lift}_{\mathrm{list}}\ \ell\ xs = \mathsf{lift}\ \ell\ (\mathit{lsequence}_{\mathrm{list}}\ xs)
\end{array}$$

Notice that we have $\mathsf{lift}_{\mathrm{list}}\ \mathit{id}_{\mathrm{L}} = \mathit{lsequence}_{\mathrm{list}}$.

Tagged lists form an instance of *Poset*.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Poset}\ a \Rightarrow \mathit{Poset}\ [a]\ \textbf{where} \\
\quad xs \curlyvee ys = \textbf{if}\ \mathit{length}\ xs == \mathit{length}\ ys\ \textbf{then}\ \mathit{zipWith}\ (\curlyvee)\ xs\ ys \\
\qquad\qquad\quad \textbf{else} \qquad\qquad\qquad\qquad\quad \bot \quad \text{-- } \textit{Unreachable in our framework}
\end{array}$$

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* in lifted functions.

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

$$
\begin{array}{l}
\mathsf{unlift}_{\mathrm{list}} :: \forall a\, b.\, Eq\ a \Rightarrow (\forall s.\,[L\ s\ a] \to L\ s\ b) \to Lens\ [a]\ b \\
\mathsf{unlift}_{\mathrm{list}}\, f = Lens\ (\lambda s \to get\ (mkLens\ s)\ s)\ (\lambda s \to put\ (mkLens\ s)\ s) \\
\quad \mathbf{where} \\
\qquad mkLens\ s = f\ (projs\ (length\ s)) \mathbin{\hat{\circ}} tagList_{\mathrm{L}} \\
\qquad tagList_{\mathrm{L}} = Lens\ (map\ O)\ (\lambda\_\ ys \to map\ unTag\ ys) \\
\qquad projs\ n = map\ proj_{\mathrm{L}}\ [0\,..\,n-1] \\
\qquad proj_{\mathrm{L}} :: Int \to L\ [Tag\ a]\ a \\
\qquad proj_{\mathrm{L}}\ i = Lens\ (\lambda xs \to unTag\ (xs\,!!\,i))\ (\lambda as\ a \to update\ i\ (U\ a)\ as) \\
\qquad update :: Int \to a \to [a] \to [a] \\
\qquad update\ 0\ v\ (\_ : xs) = v : xs \\
\qquad update\ i\ v\ (x : xs) \ \ = x : update\ (i-1)\ v\ xs
\end{array}
$$

Given that the need to inspect the length of the source leads to the separate definitions of *get* and *put* in the above, there might be worry that we may lose the guarantee of well-behavedness 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 lift_2, $\mathsf{lift}_{\mathrm{list}}$ is an injection with $\mathsf{unlift}_{\mathrm{list}}$ as its left inverse.

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

$$
\begin{array}{l}
tail :: [a] \to [a] \\
tail\ (x : xs) = xs
\end{array}
$$

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

$$
\begin{array}{l}
tail_{\mathrm{L}} :: Eq\ a \Rightarrow Lens\ [a]\ [a] \\
tail_{\mathrm{L}} = \mathsf{unlift}_{\mathrm{list}}\ (lsequence_{\mathrm{list}} \circ tail)
\end{array}
$$

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

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

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

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

$$
\begin{array}{l}
catLine_{\mathrm{L}} :: Lens\ (String, String)\ String \\
catLine_{\mathrm{L}} = \\
\quad Lens\ (\lambda (s,t) \to s +\!\!+ \texttt{"\\n"} +\!\!+ t) \\
\qquad\quad (\lambda (s,t)\ u \to \mathbf{let}\ n = length\ (filter\ (==\ \texttt{'\\n'})\ s)
\end{array}
$$

$$
\begin{aligned}
&\quad i = \mathit{elemIndices}\ \texttt{'\textbackslash n'}\ u\,!!\,n \\
&\quad (s', t') = \mathit{splitAt}\ i\ u \\
&\mathbf{in}\ (s', \mathit{tail}\ t'))
\end{aligned}
$$

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, $\mathit{put}\ \mathit{catLine}_{\mathrm{L}}\ (s,t)\ u$ splits u into s' and $\texttt{"\textbackslash n"} +\!\!+\ t'$ so that s' contains the same number of newlines as the original s. For example, $\mathit{put}\ \mathit{catLine}_{\mathrm{L}}\ (\texttt{"a\textbackslash nbc"}, \texttt{"de"})\ \texttt{"A\textbackslash nB\textbackslash nC"} = (\texttt{"A\textbackslash nB"}, \texttt{"C"})$.

Then, construction of a bidirectional version $\mathit{unlines}_{\mathrm{L}}$ of *unlines* is straightforward; we only need to replace `""` with *new* `""` and *catLine* with $\mathsf{lift}_2\ \mathit{catLine}_{\mathrm{L}}$, and to apply $\mathsf{unlift}_{\mathsf{list}}$ to obtain a lens.

$$
\begin{aligned}
&\mathit{unlines}_{\mathrm{L}} :: \mathit{Lens}\ [\mathit{String}]\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{L}} = \mathsf{unlift}_{\mathsf{list}}\ \mathit{unlines}_{\mathrm{F}} \\
&\mathit{unlines}_{\mathrm{F}} :: \forall s.\,[L\ s\ \mathit{String}] \to L\ s\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{F}}\ [\,] \qquad\ \ = \mathit{new}\ \texttt{""} \\
&\mathit{unlines}_{\mathrm{F}}\ (x : xs) = \mathsf{lift}_2\ \mathit{catLine}_{\mathrm{L}}\ (x, \mathit{unlines}_{\mathrm{F}}\ xs)
\end{aligned}
$$

As one can see, $\mathit{unlines}_{\mathrm{F}}$ is written in the same applicative style as *unlines*. The construction principle is: if the original function handles data that one would like to update bidirectionally (e.g., *String* in this case), replace all manipulations (e.g., *catLine* and `""`) of the data with the corresponding bidirectional versions (e.g., $\mathsf{lift}_2\ \mathit{catLine}_{\mathrm{L}}$ and *new* `""`).

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

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

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

The same coding principle for constructing bidirectional versions applies.

$$
\begin{aligned}
&\mathit{unlines}_{\mathrm{L}} :: \mathit{Lens}\ [\mathit{String}]\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{L}} = \mathsf{unlift}_{\mathsf{list}}\ \mathit{unlines}_{\mathrm{F}} \\
&\mathit{unlines}_{\mathrm{F}} :: \forall s.\,[L\ s\ \mathit{String}] \to L\ s\ \mathit{String} \\
&\mathit{unlines}_{\mathrm{F}} = \mathit{foldr}\ (\mathit{curry}\ (\mathsf{lift}_2\ \mathit{catLine}_{\mathrm{L}}))\ (\mathit{new}\ \texttt{""})
\end{aligned}
$$

The new $\mathit{unlines}_{\mathrm{F}}$ is again in the same applicative style as the new *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 (Matsuda & Wang 2013; Voigtländer 2009a; Wang & Najd 2014). We will discuss the connection in Section 9.1.

4.2 Datatype-Generic Unlifting Functions

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

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

$$traverse :: (Traversable\ t, Applicative\ f) \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)$$

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

$$\begin{array}{l}
\mathbf{newtype}\ Const\ a\ b = Const\ \{getConst :: a\} \\
contents :: Traversable\ t \Rightarrow t\ a \to [a] \\
contents\ t = getConst\ (traverse\ (\lambda x \to Const\ [x])\ t) \\
fill :: Traversable\ t \Rightarrow t\ b \to [a] \to t\ a \\
fill\ t\ \ell = evalState\ (traverse\ next\ t)\ \ell \\
\quad \mathbf{where}\ next\ _ = \mathbf{do}\ (a:x) \leftarrow Control.Monad.State.get \\
\qquad\qquad\qquad\qquad\quad Control.Monad.State.put\ x \\
\qquad\qquad\qquad\qquad\quad return\ a
\end{array}$$

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 (McBride & Paterson 2008). The instances of *Traversable* must satisfy certain laws (Bird *et al.* 2013); and for such lawful instances, we have

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

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

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

We can now define a generic *lsequence* function as follows.

$$\begin{array}{l}
lsequence :: (Eq\ a, Eq\ (t\ ()), Traversable\ t) \Rightarrow t\ (L\ s\ a) \to L\ s\ (t\ a) \\
lsequence\ t = \mathsf{lift}\ (fill_{\mathrm{L}}\ (shape\ t))\ (lsequence_{\mathrm{list}}\ (contents\ t)) \\
\quad \mathbf{where} \\
\qquad fill_{\mathrm{L}}\ s = Lens\ (\lambda xs \to fill\ s\ xs)\ (\lambda_\ t \to contents'\ s\ t) \\
\qquad contents'\ s\ t = \mathbf{if}\ shape\ t == s\ \mathbf{then}\ contents\ t \\
\qquad\qquad\qquad\qquad\ \mathbf{else}\qquad\quad error\ \texttt{"Shape Mismatch"}
\end{array}$$

[2] In GHC, the function *contents* is called *toList*, which is defined in `Data.Foldable` (Every *Traversable* instance is also an instance of *Foldable*). We use the name *contents* to emphasize the function's role of extracting contents from structures (Bird *et al.* 2013).

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

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

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

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

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

5 An Application: Bidirectional Evaluation

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

An evaluator can be seen as a mapping from an environment to a value of a given expression. A bidirectional evaluator (Hidaka *et al.* 2010) additionally takes the same expression but maps an updated value of the expression back to an updated environment, so that evaluating the expression under the updated environment results in the value.

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

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

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*.

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

Using our framework, writing a bidirectional evaluator is almost as easy as writing the usual unidirectional one.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eval} :: \mathit{Env}\ (L\ s\ \mathit{Int}) \to \mathit{Exp} \to \mathit{Val}\ (L\ s\ \mathit{Int})} \\
\mathit{eval}\ \mathit{env}\ (\mathit{ENum}\ n) & = & \mathit{VNum}\ (\mathit{new}\ n) \\
\mathit{eval}\ \mathit{env}\ (\mathit{EInc}\ e) & = & \mathbf{let}\ \mathit{VNum}\ v = \mathit{eval}\ \mathit{env}\ e \\
 & & \mathbf{in}\ \mathit{VNum}\ (\mathsf{lift}\ \mathit{inc}_{\mathrm{L}}\ v) \\
\mathit{eval}\ \mathit{env}\ (\mathit{EVar}\ x) & = & \mathit{lkup}\ x\ \mathit{env} \\
\mathit{eval}\ \mathit{env}\ (\mathit{EApp}\ e_1\ e_2) & = & \mathbf{let}\ \mathit{VFun}\ x\ e'\ (\mathit{Env}\ \mathit{env}') = \mathit{eval}\ \mathit{env}\ e_1 \\
 & & \qquad v_2 = \mathit{eval}\ \mathit{env}\ e_2 \\
 & & \mathbf{in}\ \mathit{eval}\ (\mathit{Env}\ ((x, v_2) : \mathit{env}'))\ e' \\
\mathit{eval}\ \mathit{env}\ (\mathit{EFun}\ x\ e) & = & \mathit{VFun}\ x\ e\ \mathit{env}
\end{array}
$$

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

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

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

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

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

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

```
expr = twice @@ twice @@ twice @@ twice @@ inc @@ x
  where
    twice = EFun "f" $ EFun "x" $ EVar "f" @@ (EVar "f" @@ EVar "x")
    x     = EVar "x"
    inc   = EFun "x" $ EInc (EVar "x")
infixl 9 @@   -- @@ is left associative
(@@) = EApp
```

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

```
expr = ((((twice twice) twice) twice) inc) x
  where twice f x = f (f x)
        inc x = x + 1
```

Now giving an environment that binds the free variable x, we can run the bidirectional evaluator as follows, with $\mathit{env}_0 = \mathit{Env}\ [(\texttt{"x"}, \mathit{VNum}\ 3)]$.

```
Main> get (evalL expr) env0
VNum 65539
Main> put (evalL expr) env0 (VNum 65536)
Env [("x", VNum 0)]
```

As a remark, this seemingly innocent implementation of $\mathit{eval}_{\mathrm{L}}$ is actually highly non-trivial. It essentially defines compositional (or modular) bidirectionalization (Matsuda & Wang 2013; Matsuda *et al.* 2007; Voigtländer 2009a; Wang & Najd 2014) of programs that are *monomorphic* in type and use *higher-order* functions in definition—something that has not been achieved in bidirectional-transformation research so far.

6 Extensions

In this section, we extend our framework in two dimensions: allowing shape changes via lifting lens combinators, and allowing $(L\ s\ A)$-values to be inspected during forward transformations following our previous work (Matsuda & Wang 2013, 2014).

6.1 Lifting Lens-Combinators

An advantage of the original lens combinators (Foster *et al.* 2007) (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 $Lens\ A\ B \to Lens\ C\ D$ we have

$$\begin{array}{l}
\mathsf{liftC} :: Eq\ a \Rightarrow (Lens\ a\ b \to Lens\ c\ d) \to (\forall s. L\ s\ a \to L\ s\ b) \to (\forall t. L\ t\ c \to L\ t\ d) \\
\mathsf{liftC}\ c\ f = \mathsf{lift}\ (c\ (\mathsf{unlift}\ f))
\end{array}$$

Using the analogy to higher-order abstract syntax (Church 1940; Huet & Lang 1978; Miller & Nadathur 1987; Pfenning & Elliott 1988), the polymorphic arguments of the lifted combinators represent closed expressions; for example, a program like $\lambda x \to \ldots c\ (\ldots x \ldots) \ldots$ does not type-check when c is a lifted combinator.

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

$$\begin{array}{l}
mapDefault_{\mathrm{C}} :: a \to Lens\ a\ b \to Lens\ [a]\ [b] \\
mapDefault_{\mathrm{C}}\ d\ \ell = Lens\ (map\ (get\ \ell))\ (\lambda s\ v \to go\ s\ v) \\
\quad \textbf{where}\ go\ ss \quad\ [\,] \qquad\ \ = [\,] \\
\qquad\qquad\ go\ [\,] \quad\ (v : vs) = put\ \ell\ d\ v : go\ [\,]\ vs \\
\qquad\qquad\ go\ (s : ss)\ (v : vs) = put\ \ell\ s\ v : go\ ss\ vs
\end{array}$$

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

$$\begin{array}{l}
mapAdd_{\mathrm{L}} :: Lens\ [(Int,Int)]\ [Int] \\
mapAdd_{\mathrm{L}} = \mathsf{unlift}\ mapAdd_{\mathrm{F}} \\
mapAdd_{\mathrm{F}} :: \forall t. L\ t\ [(Int,Int)] \to L\ t\ [Int] \\
mapAdd_{\mathrm{F}}\ xs = map_{\mathrm{F}}\ (0,0)\ (\mathsf{lift}\ addL)\ xs \\
map_{\mathrm{F}} :: Eq\ a \Rightarrow a \to (\forall s. L\ s\ a \to L\ s\ b) \to (\forall t. L\ t\ [a] \to L\ t\ [b]) \\
map_{\mathrm{F}}\ d = \mathsf{liftC}\ (mapDefault_{\mathrm{C}}\ d) \\
addL :: Lens\ (Int,Int)\ Int \\
addL = Lens\ (\lambda(x,y) \to x+y)\ (\lambda(x,_)\ v \to (x, v-x))
\end{array}$$

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

$$\begin{array}{l}
\texttt{Main>}\ put\ mapAdd_{\mathrm{L}}\ [(1,1),(2,2)]\ [3,5] \\
[(1,2),(2,3)] \\
\texttt{Main>}\ put\ mapAdd_{\mathrm{L}}\ [(1,1),(2,2)]\ [3] \\
[(1,2)] \\
\texttt{Main>}\ put\ mapAdd_{\mathrm{L}}\ [(1,1),(2,2)]\ [3,5,7] \\
[(1,2),(2,3),(0,7)]
\end{array}$$

The trick is that the expression $map_{\mathrm{F}}\ (0,0)\ (\mathsf{lift}\ addL)$ has type $\forall s. L\ s\ [(Int,Int)] \to L\ s\ [Int]$, where the list occurs inside $L\ s$, contrasting to $map\ (\mathsf{lift}\ addL)$'s type $\forall s. [L\ s\ (Int,Int)] \to [L\ s\ Int]$. Intuitively, the type constructor $L\ s$ can be seen as an updatability annotation; $L\ s\ [(Int,Int)]$ means that the list itself is updatable, whereas $[L\ s\ (Int,Int)]$ means that only the elements are updatable. Here is the trade-off: the former has better updatability at the cost of a special lifted lens combinator; the latter has less updatability but simply uses the usual *map* directly. Our framework enables programmers to choose either style, or anywhere in between freely.

This position-based approach used in $mapDefault_{\mathrm{C}}$ is not the only way to resolve shape discrepancies. We can also match elements according to keys (Barbosa *et al.* 2010; Foster *et al.* 2010). As an example, let us consider a variant of the map combinator.

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

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

$$\begin{array}{l}
mapAddByKey_{\mathrm{L}} :: Eq\ k \Rightarrow Lens\ [(k,(Int,Int))]\ [(k,Int)] \\
mapAddByKey_{\mathrm{L}} = \mathsf{unlift}\ mapAddByKey_{\mathrm{F}} \\
mapAddByKey_{\mathrm{F}} :: Eq\ k \Rightarrow \forall t. L\ t\ [(k,(Int,Int))] \to L\ t\ [(k,Int)] \\
mapAddByKey_{\mathrm{F}}\ xs = mapByKey_{\mathrm{F}}\ (0,0)\ (\mathsf{lift}\ addL)\ xs \\
mapByKey_{\mathrm{F}} :: (Eq\ k, Eq\ a) \Rightarrow \\
\qquad\qquad a \to (\forall s. L\ s\ a \to L\ s\ b) \to (\forall t. L\ t\ [(k,a)] \to L\ t\ [(k,b)]) \\
mapByKey_{\mathrm{F}}\ d = \mathsf{liftC}\ (mapByKey_{\mathrm{C}}\ d)
\end{array}$$

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

$$\begin{array}{l}
\texttt{Main>}\ put\ mapAddByKey_{\mathrm{L}}\ s\ [(\texttt{"B"},5),(\texttt{"A"},3)] \\
[(\texttt{"B"},(2,3)),(\texttt{"A"},(1,2))] \\
\texttt{Main>}\ put\ mapAddByKey_{\mathrm{L}}\ s\ [(\texttt{"A"},3)] \\
[(\texttt{"A"},(1,2))]
\end{array}$$

$\texttt{Main>}$ $\mathit{put}\ \mathit{mapAddByKey}_{\mathrm{L}}\ s\ [(\texttt{"B"},5),(\texttt{"C"},7),(\texttt{"A"},3)]$
$[(\texttt{"B"},(2,3)),(\texttt{"C"},(0,7)),(\texttt{"A"},(1,2))]$

6.2 Observations of Lifted Values

So far we have programmed bidirectional transformations ranging from polymorphic to monomorphic functions. For example, *unlines* is monomorphic because its base case returns a String constant, which is nicely handled in our framework by the function *new*. At the same time, it is also obvious that the creation of constant values is not the only cause of a transformation being monomorphic (Matsuda & Wang 2013, 2014). For example, let us consider the following toy program.[4]

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

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 $\mathit{bad}_{\mathrm{L}} = \mathsf{unlift}_2\ (\mathsf{lift}_2\ \mathit{id}_{\mathrm{L}} \circ \mathit{bad})$ would violate well-behavedness, as $\mathit{put}\ \mathit{bad}_{\mathrm{L}}\ (0,2)\ (1,2) = (1,2)$ but $\mathit{get}\ \mathit{bad}_{\mathrm{L}}\ (1,2) = (1,1)$.

Our previous work (Matsuda & Wang 2013, 2014) 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 simpler way due to our new compositional formalization.

$$\mathbf{newtype}\ R\ a\ b = R\ (\mathit{Poset}\ a \Rightarrow a \to (b, a \to \mathit{Bool}))$$

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

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

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

$$\begin{aligned} &\mathsf{observe} :: \mathit{Eq}\ w \Rightarrow L\ s\ w \to R\ s\ w \\ &\mathsf{observe}\ x = R\ (\lambda s \to \mathbf{let}\ w = \mathit{get}\ x\ s\ \mathbf{in}\ (w, \lambda s' \to \mathit{get}\ x\ s' == w)) \\ &\mathsf{unliftM}_2 :: (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow (\forall s.\ (L\ s\ a, L\ s\ b) \to R\ s\ (L\ s\ c)) \to \mathit{Lens}\ (a,b)\ c \\ &\mathsf{unliftM}_2\ f = \mathit{Lens}\ (\lambda s \to \mathit{get}\ (\mathit{mkLens}\ f\ s)\ s)\ (\lambda s \to \mathit{put}\ (\mathit{mkLens}\ f\ s)\ s) \\ &\quad \mathbf{where} \end{aligned}$$

[4] This code actually does not type check as $(==)$ on $(L\ s\ \mathit{Int})$-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.

$$
\begin{array}{l}
\mathit{mkLens}\, f\; s = \\
\quad \mathbf{let}\; (\ell, p) = \mathbf{let}\; R\; m = f\; (\mathit{fst}'_{\mathrm{L}}, \mathit{snd}'_{\mathrm{L}})\; \mathbf{in}\; m\; (\mathit{get}\; \mathit{tag2}_{\mathrm{L}}\; s) \\
\qquad \ell' = \ell \,\hat{\circ}\, \mathit{tag2}_{\mathrm{L}} \\
\qquad \mathit{put}'\; s\; v = \mathbf{let}\; s' = \mathit{put}\; \ell'\; s\; v \\
\qquad\qquad\qquad\quad \mathbf{in}\; \mathbf{if}\; p\; (\mathit{get}\; \mathit{tag2}_{\mathrm{L}}\; s')\; \mathbf{then}\; s' \\
\qquad\qquad\qquad\qquad \mathbf{else}\; \mathit{error}\; \texttt{"Changed Observation"} \\
\quad \mathbf{in}\; \mathit{Lens}\; (\mathit{get}\; \ell')\; \mathit{put}'
\end{array}
$$

Although we define the *get* and *put* components of the resulting lens separately in $\mathsf{unliftM}_2$, well-behavedness is guaranteed as long as R and L are used abstractly in f, where this abstract nature of R and L are formalized as follows.

Definition 4 (Abstract Nature of L and R). We say *L and R are abstract* in $f :: \tau$ if there is a polymorphic function h of type

$$
\begin{array}{l}
\forall \ell\; r.\, (\forall a\, b.\, \mathit{Lens}\; a\; b \to (\forall s.\, \ell\; s\; a \to \ell\; s\; b)) \\
\quad \to (\forall a\, b.\, (\forall s.\, \ell\; s\; a \to \ell\; s\; b) \to \mathit{Lens}\; a\; b) \\
\quad \to (\forall s.\, \ell\; s\; ()) \\
\quad \to (\forall s\; a\; b.\, \ell\; s\; a \to \ell\; s\; b \to \ell\; s\; (a, b)) \\
\quad \to (\forall a\; b\; c.\, (\forall s.\, (\ell\; s\; a, \ell\; s\; b) \to \ell\; s\; c) \to \mathit{Lens}\; (a, b)\; c) \\
\quad \to (\forall s\; w.\, \mathit{Eq}\; w \Rightarrow \ell\; s\; w \to r\; s\; w) \\
\quad \to (\forall a\; b.\, (\forall s.\, \ell\; s\; a \to r\; s\; (\ell\; s\; b)) \to \mathit{Lens}\; a\; b) \\
\quad \to (\forall a\; b\; c.\, (\forall s.\, (\ell\; s\; a, \ell\; s\; b) \to r\; s\; (\ell\; s\; c)) \to \mathit{Lens}\; (a, b)\; c) \\
\quad \to \tau'
\end{array}
$$

satisfying $f = h\; \mathsf{lift}\; \mathsf{unlift}\; \mathsf{unit}\; (\circledast)\; \mathsf{unlift}_2\; \mathsf{observe}\; \mathsf{unliftM}\; \mathsf{unliftM}_2$ and $\tau' = \tau[\ell/L, r/R]$. □

Note that, similarly to $\mathsf{unliftM}_2$, we can define $\mathsf{unliftM}$ and $\mathsf{unliftMT}$, as monadic versions of unlift and $\mathsf{unliftT}$. Formally, we have the following theorem.

Theorem 6. Let f be a function of type $\forall s.\, (L\; s\; A, L\; s\; B) \to R\; s\; (L\; s\; C)$ in which L and R are abstract. Then $\mathsf{unliftM}_2\, f$ is well-behaved, if all the following conditions hold.

- only well-behaved lenses are passed to lift during evaluation,
- w in $\mathsf{observe} :: \mathit{Eq}\; w \Rightarrow L\; s\; w \to R\; s\; w$ is only instantiated to types W such that $(==)$ on W coincides with the semantic (observational) equality.

We postpone the proof till Section 8.

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

$$
\begin{array}{l}
\mathit{good} :: \forall s.\, (L\; s\; \mathit{Int}, L\; s\; \mathit{Int}) \to R\; s\; (L\; s\; (\mathit{Int}, \mathit{Int})) \\
\mathit{good}\; (x, y) = \mathbf{do}\; b \leftarrow \mathsf{liftO}_2\; (==)\; x\; (\mathit{new}\; 0) \\
\qquad\qquad\qquad\quad \mathit{return}\; (\mathbf{if}\; b\; \mathbf{then}\; x \circledast y\; \mathbf{else}\; x \circledast \mathit{new}\; 1)
\end{array}
$$

Here, liftO_2 is defined as follows.

$$
\begin{array}{l}
\mathsf{liftO}_2 :: \mathit{Eq}\; w \Rightarrow (a \to b \to w) \to L\; s\; a \to L\; s\; b \to R\; s\; w \\
\mathsf{liftO}_2\; p\; x\; y = \mathsf{liftO}\; (\mathit{uncurry}\; p)\; (x \circledast y) \\
\mathsf{liftO} :: \mathit{Eq}\; w \Rightarrow (a \to w) \to L\; s\; a \to R\; s\; w
\end{array}
$$

$\mathsf{liftO}\ p\ x = \mathsf{observe}\ (\mathsf{lift}\ (\mathit{Lens}\ p\ \mathit{unused})\ x)$
  where $\mathit{unused}\ s\ v \mid v == p\ s = s$

Then the obtained lens $\mathit{good}_{\mathrm{L}} = \mathsf{unliftM}_2\ \mathit{good}$ successfully rejects illegal updates, as $\mathit{put}\ \mathit{good}_{\mathrm{L}}\ (0,2)\ (1,2) = \bot$. Note that *unused* is unused as it stands in our framework; recall that $\mathsf{observe}\ x$ only uses the *get* component of x.

One might have noticed that the definition of *good* is in the *Monadic style*—not applicative in the sense of (McBride & Paterson 2008). This is necessary for handling observations, as the effect of $(R\ s)$ can depend on the value in it (Lindley *et al.* 2011).

Example 5 (*nub*). As a slightly involved example, let us consider a bidirectional version of *nub*, which removes duplicate elements in a list as $\mathit{nub}\ [1,1,2,3,2] = [1,2,3]$.

$$
\begin{array}{l}
\mathit{nubF} :: \mathit{Eq}\ a \Rightarrow [L\ s\ a] \to R\ s\ [L\ s\ a] \\
\mathit{nubF}\ [\,] \qquad\;\; = \mathit{return}\ [\,] \\
\mathit{nubF}\ (x : \mathit{xs}) = \mathbf{do}\ \mathit{xs}' \leftarrow \mathit{deleteF}\ x\ \mathit{xs} \\
\qquad\qquad\qquad\quad\;\; r \leftarrow \mathit{nubF}\ \mathit{xs}' \\
\qquad\qquad\qquad\quad\;\; \mathit{return}\ (x : r) \\
\mathit{deleteF} :: \mathit{Eq}\ a \Rightarrow L\ s\ a \to [L\ s\ a] \to R\ s\ [L\ s\ a] \\
\mathit{deleteF}\ x\ [\,] = \mathit{return}\ [\,] \\
\mathit{deleteF}\ x\ (y : \mathit{ys}) = \mathbf{do}\ b \leftarrow \mathsf{liftO}_2\ (==)\ x\ y \\
\qquad\qquad\qquad\qquad\quad\;\; r \leftarrow \mathit{deleteF}\ x\ \mathit{ys} \\
\qquad\qquad\qquad\qquad\quad\;\; \mathit{return}\ (\mathbf{if}\ b\ \mathbf{then}\ r\ \mathbf{else}\ y : r) \\
\mathit{nubL} :: \mathit{Eq}\ a \Rightarrow \mathit{Lens}\ [a]\ [a] \\
\mathit{nubL} = \mathsf{unliftMT}\ (\mathit{fmap}\ \mathit{lsequence} \circ \mathit{nubF})
\end{array}
$$

The obtained lens *nubL* works as follows.

`Main>` $\mathit{get}\ \mathit{nubL}\ [1,1,2,3,2]$
$[1,2,3]$
`Main>` $\mathit{put}\ \mathit{nubL}\ [1,1,2,3,2]\ [1,2,6]$
$[1,1,2,6,2]$

However, there is a limitation: *nubL* cannot change any duplicated elements.

`Main>` $\mathit{put}\ \mathit{nubL}\ [1,1,2,3,2]\ [1,5,6]$
$***$ Exception: Changed Observation

Unlike the previous example that updates 3, we have two copies of 2 in the source: the first one appears as the third element and the second one appears as the last element. They are compared by $==$, and the first one comes in the view while the second one is dropped. This also imposes a constraint on the source that the third element and the last element must be equal (but not necessarily remain as 2). Thus, we cannot change the 2 in the view because it changes only the first occurrence of 2 while leaving the second occurrence untouched.

Voigtländer (2009a) addresses the problem by treating equal elements in the source as the "same", where a change to one automatically triggers a change to others. In the above example, if we can update both occurrences of 2 simultaneously to 5, no bidirectional laws will be violated.

With a small amount of additional work, we can incorporate this idea while keeping the definition of *nubF*. First, we prepare a datatype in which the "same" elements are merged to one.

$$\mathbf{data}\ \mathit{EList}\ a = \mathit{EList}\ [\mathit{Int}]\ [(\mathit{Int}, a)]$$

Intuitively, *EList* indexes elements: its first parameter is the list of *Int*-indices and the second parameter is an injective mapping from the indices to actual list elements. It is easy to decompose a list to *EList*, and vice versa.

$$\begin{array}{l}
\mathit{decompose} :: \mathit{Eq}\ a \Rightarrow [a] \rightarrow \mathit{EList}\ a \\
\mathit{decompose}\ \mathit{xs} = \mathbf{let}\ \mathit{ys} = \mathit{nub}\ \mathit{xs} \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{EList}\ [\mathit{fromJust}\ (\mathit{findIndex}\ ({=}{=}\ x)\ \mathit{ys}) \mid x \leftarrow \mathit{xs}]\ (\mathit{zip}\ [0\,..]\ \mathit{ys}) \\
\mathit{recompose} :: \mathit{EList}\ a \rightarrow [a] \\
\mathit{recompose}\ (\mathit{EList}\ \mathit{is}\ m) = [\mathit{fromJust}\ (\mathit{lookup}\ i\ m) \mid i \leftarrow \mathit{is}]
\end{array}$$

Here, $\mathit{findIndex} :: \mathit{Eq}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Maybe}\ \mathit{Int}$ defined in `Data.List`, is a function that takes an element x and a list xs, and returns the index of the first occurrence of x in xs if it exists. The function *fromJust* is a function defined by $\mathit{fromJust}\ (\mathit{Just}\ x) = x$. For example, $\mathit{decompose}\ [A,A,B,C,B]$ results in $\mathit{EList}\ [0,0,1,2,1]\ [(0,A),(1,B),(2,C)]$.

From the two functions *decompose* and *recompose*, we can define lens *decomposeL* as follows.

$$\begin{array}{l}
\mathit{decomposeL} :: \mathit{Eq}\ a \Rightarrow \mathit{Lens}\ [a]\ (\mathit{EList}\ a) \\
\mathit{decomposeL} = \mathit{Lens}\ \mathit{decompose}\ (\lambda_\ v \rightarrow \mathit{recompose}'\ v) \\
\quad \mathbf{where}\ \mathit{recompose}'\ v = \mathbf{let}\ s = \mathit{recompose}\ v \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in\ if}\ v\ {=}{=}\ \mathit{decompose}\ s\ \mathbf{then}\ s\ \mathbf{else}\ \bot
\end{array}$$

Function $\mathit{recompose}'$ is a variant of *recompose* that actually checks the invariant on *EList*. That is, for $\mathit{EList}\ \mathit{is}\ m$, m must be injective and defined for all the indices in is. This check is conservative, but works fine for our purpose.

Now, we are ready to generalize *nubL*.

$$\begin{array}{l}
\mathit{nubL}' :: \mathit{Eq}\ a \Rightarrow \mathit{Lens}\ [a]\ [a] \\
\mathit{nubL}' = \mathsf{unliftMT}\ (\lambda \mathit{xs} \rightarrow \mathit{fmap}\ \mathit{lsequence}\ (\mathit{nubF}\ (\mathit{recompose}\ \mathit{xs})))\ \hat{\circ}\ \mathit{decomposeL}
\end{array}$$

Note that $\mathit{recompose}\ \mathit{xs}$ type-checks because $\mathit{recompose} :: \mathit{EList}\ a \rightarrow [a]$ does not require *Eq* for a.

The new lens nubL' accepts more updates than *nubL*

`Main>` $\mathit{put}\ \mathit{nubL}'\ [1,1,2,3,2]\ [4,5,6]$
$[4,4,5,6,5]$

without compromising the bidirectional laws.

`Main>` $\mathit{put}\ \mathit{nubL}'\ [1,1,2,3,2]\ [4,5,5]$
*** Exception: Changed Observation

As a remark, automatically treating all equal elements as the same may not always be the most desirable. Our previous work (Matsuda & Wang 2014) addresses the problem by selective indexing: only the elements that pass an equality check occurring in the execution of *get* are considered the same. It is not obvious how our current framework can be extended to achieve this because now elements can be compared after applying lifted lens functions, which may require us to index elements in intermediate views, unlike the situation in previous work (Matsuda & Wang 2014; Voigtländer 2009a) where only *source* elements are indexed. □

7 An XML Transformation Example

XML transformation is a common application area of bidirectional programming, where data in different XML formats are synchronized through transformations going both ways. In this section, we program such transformations in our framework with the extensions discussed in Section 6.2. Specifically, we implement a slightly simplified version of the query Q5 of Use Case "STRING" in XML Query Use Cases (`http://www.w3.org/TR/xquery-use-cases`). Different from the existing *first-order* languages *specialized* for bidirectional XML transformations (Fegaras 2010; Liu *et al.* 2007; Pacheco *et al.* 2014a), our language is general purpose, and, as will be demonstrated by this exercise, can be seamlessly integrated with an existing functional framework for transforming XML that involves *higher-order* features.

The basic idea follows from our previous work (Matsuda & Wang 2013, 2014). We use the established HaXML framework (Wallace & Runciman 1999) to construct XML transformations using filters—functions of type $a \to [b]$. Adapting it to our context of bidirectional transformations with observations, we will use filters of type $L\ s\ a \to ListT\ (R\ s)\ (L\ s\ b)$, where the monad transformer *ListT* in *Control.Monad.List* is defined by

$$\textbf{newtype}\ ListT\ m\ a = ListT\ \{runListT :: m\ [a]\}$$

with an implementation of the function "*lift*" of type $Monad\ m \Rightarrow m\ a \to ListT\ m\ a$. To avoid name conflicts, we use the following type-specialized version.

$$\begin{array}{l} liftListT :: Monad\ m \Rightarrow m\ a \to ListT\ m\ a \\ liftListT = Control.Monad.Trans.lift \end{array}$$

The type constructor *ListT m* is an instance of *MonadPlus* in *Control.Monad*, which gives us $mplus :: MonadPlus\ m \Rightarrow m\ a \to m\ a \to m\ a$ and $mzero :: MonadPlus\ m \Rightarrow m\ a$. For those who are familiar with monad transformer laws, $R\ s$ is a commutative monad in our case, and thus $ListT\ (R\ s)$ is a monad.

7.1 A Datatype for XML

To start with, we define a datatype to represent XML elements. Following our previous work (Matsuda & Wang 2013, 2014), we use a simple rose-tree representation as follows.

$$\begin{array}{l} \textbf{data}\ Tree\ a = Node\ a\ [Tree\ a]\ \textbf{deriving}\ (Eq, Functor, Foldable, Traversable) \\ \textbf{data}\ Label = E\ String \mid T\ String\ \textbf{deriving}\ Eq \end{array}$$

Here, *E* and *T* stand for "element name" and "text" respectively. We shall omit other features of XML that cannot be expressed in this datatype, notably attributes, IDs and IDREFs, schemas, and namespaces.

For example, an XML fragment

```
<content><par>Today, Gorilla Corporation announced that ...</par>
         <par>As a result of this acquisition, ...</par></content>
```

is represented as follows.

```
Node (E "content") [
  Node (E "par") [
    Node (T "Today, Gorilla Corporation announced that ...") [],
  Node (E "par") [
    Node (T "As a result of this acquisition, ...") []]]
```

The following function *label* is sometimes useful to write examples.

$label :: Tree\ a \to a$
$label\ (Node\ lab\ \_) = lab$

Then, we define a type of (bidirectional) filters as follows.

$\textbf{type}\ BFilter\ s\ a = Tree\ (L\ s\ a) \to ListT\ (R\ s)\ (Tree\ (L\ s\ a))$

7.2 Basic Filters

As in our previous work (Matsuda & Wang 2013, 2014), we introduce several basic filters. The simplest filter *keep* keeps its input.

$keep :: BFilter\ s\ a$
$keep\ x = return\ x$

Filter *children* extracts the children of a node.

$children :: BFilter\ s\ a$
$children\ (Node\ \_\ ts) = ListT\ \$\ return\ ts$

Filter *ofLabel lab* returns the input if its root has the label *lab*, and fails otherwise.

$ofLabel :: L\ s\ Label \to BFilter\ s\ Label$
$ofLabel\ lab\ t = \textbf{do}\ guardM\ \$\ liftListT\ \$\ \mathsf{liftO}_2\ (==)\ (label\ t)\ lab$
$\qquad\qquad\qquad\quad return\ t$

Here, *guardM* is a variant of *guard* from `Control.Monad` which takes a monadic argument instead (function *guard* fails if its argument is *False*, and does nothing otherwise.)

$guardM :: MonadPlus\ m \Rightarrow m\ Bool \to m\ ()$
$guardM\ x = x \gg\!= guard$

Filters are composable by combinators.

$(/\!>) :: BFilter\ s\ a \to BFilter\ s\ a \to BFilter\ s\ a$
$f\ /\!>\ g = f \ggg\!> children \ggg\!> g$

Here, $(\ggg\!>)$ is the Kleisli composition operator in `Control.Monad` defined by $(f \ggg\!> g)\ x = f\ x \gg\!= g$. The operator $(/\!>)$ is useful for implementing the XPath axis "/". For example, the filter $\mathit{keep} \mathbin{/\!>} \mathit{ofLabel}\ (\mathit{new}\ (E\ \texttt{"content"}))$ extracts `content` elements from the children of its input, and the filter $\mathit{keep} \mathbin{/\!>} \mathit{keep} \mathbin{/\!>} \mathit{keep}$ extracts the grandchildren of its input.

Another useful combinator is *deep* defined as follows.

$$
\begin{array}{l}
\mathit{deep} :: \mathit{BFilter}\ s\ a \to \mathit{BFilter}\ s\ a \\
\mathit{deep}\ f\ t = \mathit{bfs}\ [t]\ [\,] \\
\quad \textbf{where} \\
\qquad \mathit{bfs}\ [\,]\ [\,] = \mathit{mzero} \\
\qquad \mathit{bfs}\ [\,]\ \mathit{qs} = \mathit{bfs}\ (\mathit{reverse}\ \mathit{qs})\ [\,] \\
\qquad \mathit{bfs}\ (t@(\mathit{Node}\ \mathit{lab}\ \mathit{ts}) : \mathit{rest})\ \mathit{qs} = \textbf{do}\ \mathit{ck} \leftarrow \mathit{gather}\ (f\ t) \\
\qquad\qquad\qquad\qquad \textbf{case}\ \mathit{ck}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad [\,] \to \mathit{bfs}\ \mathit{rest}\ (\mathit{reverse}\ \mathit{ts} +\!\!+ \mathit{qs}) \\
\qquad\qquad\qquad\qquad\quad \_ \to \mathit{return}\ t\ \text{`}\mathit{mplus}\text{`}\ \mathit{bfs}\ \mathit{rest}\ \mathit{qs}
\end{array}
$$

The expression $\mathit{deep}\ f\ t$ applies filter f to each subtree of t in the breadth-first manner, and combines by *mplus* the subtrees for which f succeeds. An auxiliary function *gather* gathers results: for example $\mathit{children}\ y$ produces one child at a time, and $\mathit{gather}\ (\mathit{children}\ y)$ gathers the children in a list.

$$
\begin{array}{l}
\mathit{gather} :: \mathit{Monad}\ m \Rightarrow \mathit{ListT}\ m\ a \to \mathit{ListT}\ m\ [a] \\
\mathit{gather}\ (\mathit{ListT}\ x) = \mathit{ListT}\ \$\ x \gg\!= (\lambda a \to \mathit{return}\ [a])
\end{array}
$$

Combinator *deep* is useful for implementing the XPath axis "`//`". For example, the filter $\mathit{deep}\ (\mathit{ofLabel}\ (\mathit{new}\ \$\ E\ \texttt{"news\_item"}))$ returns all the `news_item` elements within the input tree.

Sometimes, we want to extract the nth element of a query result. This is done by using the filter $/!\,n$ defined as follows.

$$
\begin{array}{l}
(/!) :: \mathit{BFilter}\ s\ a \to \mathit{Int} \to \mathit{BFilter}\ s\ a \\
f\ /!\ n = \lambda xs \to \textbf{do}\ \mathit{rs} \leftarrow \mathit{gather}\ (f\ xs) \\
\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{rs}\ !!\ n)
\end{array}
$$

7.3 Query Example

Now, we are ready to write a bidirectional query of Q5 (of Use Case "STRING" in XML Query Use Cases). The query extracts summaries of the news items (specifically, titles, dates and the first paragraphs) that contains "`Gorilla Corporation`" in their "`content`"s; for example, for the input shown in Figure 1, it returns the XML shown in Figure 2.[5] Assuming that the input is in a file named "`string.xml`", this query is written in XQuery as shown in Figure 3.

Figure 4 shows the bidirectional version of Q5 implemented in our framework. Here, *catDateL* is a lens whose *get* takes a triple (t,d,p) and returns a concatenated string with

[5] Both XMLs are a simplified version of the sample input and output for Q5 of Use Case "STRING" in XML Query Use Cases (`http://www.w3.org/TR/xquery-use-cases`). In the original, `par` may contain a sequence of text and elements rather than merely text. This simplification does not affect the original Q5 in XQuery, but does simplify our version written in Haskell.

```
<news>
  <news_item>
    <title>Gorilla Corporation acquires Example.com</title>
    <content>
      <par>Today, Gorilla Corporation announced that ...</par>
      <par>As a result of this acquisition, ...</par>
    </content>
    <date>2000-01-20</date>
  </news_item>
  <news_item>
    <title>Foobar Corporation releases its new line of Foo products</title>
    <content>
      <par>Foobar Corporation releases ...</par>
      <par>The President of Foobar Corporation announced that ...</par>
    </content>
    <date>2000-01-20</date>
  </news_item>
  <news_item>
    <title>Foobar Corporation is suing Gorilla Corporation</title>
    <content>
      <par>In surprising developments today, ...</par>
      <par>The tension between Foobar and Gorilla Corporations ...</par>
    </content>
    <date>2000-01-20</date>
  </news_item>
</news>
```

Fig. 1. Input XML

```
<item_summary>Gorilla Corporation acquires Example.com. 2000-01-20.
Today, Gorilla Corporation announced that, ...</item_summary>
<item_summary>Foobar Corporation is suing Gorilla Corporation. 2000-01-20.
In surprising developments today, ...</item_summary>
```

Fig. 2. Output XML

```
for $item in doc("string.xml")//news_item
where contains(string($item/content), "Gorilla Corporation")
return
    <item_summary>
        { concat($item/title,". ") }
        { concat($item/date,". ") }
        { string(($item//par)[1]) }
    </item_summary>
```

Fig. 3. Query Q5 of Use Case "String" in XML Query Use Cases

`". "`, and whose *put* takes a string in the format "`\. \d\d\d\d-\d\d-\d\d\. `" (the Perl-compatible regular-expression format), and decompose it to a triple. The code looks complicated, but this complication mainly comes from writing XML queries in a functional programming language, instead of bidirectional programming. It is worth mentioning that *q5* cannot be written in our previous framework (Matsuda & Wang 2013, 2014) as we are

$q5 :: Tree\ (L\ s\ Label) \to ListT\ (R\ s)\ [Tree\ (L\ s\ Label)]$
$q5\ doc = gather\ \$\ \mathbf{do}$
$\quad item \leftarrow deep\ (ofLabel\ (new\ \$\ E\ \texttt{"news\_item"}))\ doc$
$\quad cont \leftarrow (keep\ /\!>\ ofLabel\ (new\ \$\ E\ \texttt{"content"}))\ item$
$\quad guardM\ \$\ liftListT\ \$$
$\quad\quad \mathsf{liftO}\ (\lambda s \to \texttt{"Gorilla Corporation"}\ `isInfixOf`\ strings\ s)\ \$\ lsequence\ cont$
$\quad title \leftarrow (keep\ /\!>\ ofLabel\ (new\ \$\ E\ \texttt{"title"})\ /\!>\ keep)\ item$
$\quad date \leftarrow (keep\ /\!>\ ofLabel\ (new\ \$\ E\ \texttt{"date"})\ /\!>\ keep)\ item$
$\quad \mathbf{let}\ t\ = \mathsf{lift}\ unTextL\ \$\ label\ title$
$\quad \mathbf{let}\ d = \mathsf{lift}\ unTextL\ \$\ label\ date$
$\quad par0 \leftarrow (deep\ (ofLabel\ (new\ \$\ E\ \texttt{"par"}))\ /!\ 0\ /\!>\ keep)\ item$
$\quad \mathbf{let}\ p = \mathsf{lift}\ unTextL\ \$\ label\ par0$
$\quad return\ \$\ Node\ (new\ \$\ E\ \texttt{"item\_summary"})\ [Node\ (\mathsf{lift}_3\ catDateL\ (t,d,p))\ [\,]]$
$unTextL :: Lens\ Label\ String$
$unTextL = Lens\ (\lambda(T\ t) \to t)\ (\lambda\_\ t \to T\ t)$
$strings :: Tree\ Label \to String$
$strings\ (Node\ (T\ x)\ xs) = x$
$strings\ (Node\ \_\ \quad xs) = concatMap\ strings\ xs$
$q5L :: Lens\ (Tree\ Label)\ [Tree\ Label]$
$q5L = \mathsf{unliftMT}\ (\lambda x \to fmap\ (lsequence \circ fmap\ lsequence)\ \$\ pick\ \$\ q5\ x)$
$pick :: Monad\ m \Rightarrow ListT\ m\ a \to m\ a$
$pick\ (ListT\ x) = x \gg\!= \lambda a \to return\ (head\ a)$

Fig. 4. Query Q5 in Our Framework

reusing lenses (such as *catDateL*) as blackboxes through lifting—a key advantage of our framework.

7.4 Updatability

By applying *get q5L* to the XML data in Figure 1 (encoded in Haskell), we obtain a piece of data that corresponds to the XML in Figure 2. We can update the extracted strings as long as they still contain delimiters matching the regular expression "`\. \d\d\d\d-\d\d-\d\d\. `". This means other updates such as insertions, deletions and changes to element names `item_summary` are (rightfully) prohibited.

As an example, consider changing the text of the second extracted item as follows.

```
Foobar Corporation is suing Gorilla Corporation today. 2015-10-20.
In surprising developments today, YEAH! ...
```

We have appended "`today`" to the title part, changed the date string, and inserted "`YEAH!`". Executing *put q5L* on the new text succeeds and changes the corresponding parts in the original input XML. That is, "`today`" is appended to the title text, "`2015-10-20`" is set as the new date and "`YEAH!`" is inserted in the first paragraph of the content.

8 Correctness

In this section, we prove Theorem 6 (Proofs of Theorems 1 and 4 are similar and thus omitted). Our proof is based on the free theorems (Reynolds 1983; Voigtländer 2009b; Wadler 1989). It is worth noting that we only need to use *unary* parametricity instead of the binary one adopted in previous approaches (Matsuda & Wang 2013, 2014; Voigtländer 2009a).

8.1 Free Theorem

We firstly review the free theorems based on unary parametricity.

Roughly speaking, free theorems are theorems obtained as corollaries of relational parametricity (Bernardy *et al.* 2012; Reynolds 1983; Vytiniotis & Weirich 2010), which states that, for a closed term f of type T, f belongs to a certain relational interpretation of T. A simple example of a free theorem is that a (total) function f of type $\forall a.\, a \to a$ is the identity function, because f preserves any properties on the input.

We start by introducing some notations. We write $\mathcal{R} :: \mathsf{Pred}(A)$ if $\mathcal{R}$ is a unary relation (i.e., a predicate) on A; we identified a predicate on A with the set of A-elements satisfying the predicate. For predicates $\mathcal{R} :: \mathsf{Pred}(A)$ and $\mathcal{R}' :: \mathsf{Pred}(B)$, we write $\mathcal{R} \to \mathcal{R}' :: \mathsf{Pred}(A \to B)$ for the predicate on functions $\{f \mid \forall x \in \mathcal{R}.\, f\, x \in \mathcal{R}'\}$, and $(\mathcal{R}, \mathcal{R}') :: \mathsf{Pred}((A,B))$ for the predicate on pairs $\{(x,y) \mid x \in \mathcal{R}, y \in \mathcal{R}'\}$. For a polymorphic term f of type $\forall a. T$ and a type S, we write f_S for the instantiation of f with S, which has type $T[S/a]$. For simplicity, we sometimes omit the subscript and simply write f for f_S if S is clear from the context or irrelevant.

We introduce a *unary relational interpretation* $[\![\tau]\!]^1_\rho$ of types, where ρ is a mapping from type variables to predicates, as follows.

$$
\begin{array}{lll}
[\![a]\!]^1_\rho & = \rho(a) & \\
[\![B]\!]^1_\rho & = \{e \mid e :: B\} & \text{if } B \text{ is a base type} \\
[\![T_1 \to T_2]\!]^1_\rho & = [\![T_1]\!]^1_\rho \to [\![T_2]\!]^1_\rho & \\
[\![(T_1, T_2)]\!]^1_\rho & = ([\![T_1]\!]^1_\rho, [\![T_2]\!]^1_\rho) & \\
[\![\forall a. T]\!]^1_\rho & = \left\{ u \,\middle|\, \forall \mathcal{R} :: \mathsf{Pred}(S).\, u_S \in [\![T]\!]^1_{\rho[\alpha \to \mathcal{R}]} \right\}. &
\end{array}
$$

Here, $\rho[a \mapsto \mathcal{R}]$ extends ρ with $a \mapsto \mathcal{R}$. If $\rho = \emptyset$, we sometimes write $[\![T]\!]^1$ instead of $[\![T]\!]^1_\emptyset$. We abuse the notation to write $[\![\forall \alpha.\tau]\!]^1$ as $\forall \mathcal{R}.\mathcal{F}$ where $\mathcal{F}$ is the interpretation $[\![\tau]\!]^1_{\{\alpha \mapsto \mathcal{R}\}}$. For example, we write $\forall \mathcal{R}.\forall \mathcal{S}.\, \mathcal{R} \to \mathcal{S}$ for $[\![\forall \alpha.\forall \beta.\alpha \to \beta]\!]$. For a base type B, we also write B for $[\![B]\!]^1$. We identify the lens type *Lens A B* with the pairs of functions $(A \to B, A \to B \to A)$. Accordingly, we write *Lens* $\mathcal{S}\ \mathcal{T}$ to mean $(\mathcal{S} \to \mathcal{T}, \mathcal{S} \to \mathcal{T} \to \mathcal{S})$.

Then, parametricity states that, for a closed term f of a closed type τ, f is in $[\![\tau]\!]^1$. Free theorems are theorems obtained by instantiating parametricity.

Voigtländer (2009b) extends parametricity to a type system with type constructors. A key notion in his result is *relational action*.

Definition 5 ((Unary) Relational Action). For a type constructor κ, $\mathcal{F}$ is called a *relational action* on κ, denoted by $\mathcal{F} :: \mathsf{Pred}(\kappa)$, if $\mathcal{F}$ maps any predicate $\mathcal{R} :: \mathsf{Pred}(\tau)$ for every closed type τ to $\mathcal{F}\mathcal{R} :: \mathsf{Pred}(\kappa\, \tau)$. $\square$

Accordingly, the relational interpretations are extended as:

$$\begin{array}{lll} [\![\kappa]\!]^1_\rho & = & \rho(\kappa) \\ [\![\tau_1\ \tau_2]\!]^1_\rho & = & [\![\tau_1]\!]^1_\rho\ [\![\tau_2]\!]^1_\rho \\ [\![\forall\kappa.\tau]\!]^1 & = & \left\{ u \,\middle|\, \forall\mathcal{F} :: \mathsf{Pred}(\kappa).\, u_\kappa \in [\![\tau]\!]^1_{\rho[\kappa\mapsto\mathcal{F}]} \right\} \end{array}$$

Parametricity holds also with this relational interpretation (Bernardy *et al.* 2012; Vytiniotis & Weirich 2010). Here, κ is a type constructor of kind $* \to *$, and thus the quantified $\mathcal{F}$ is a relational action. The notation of relational action can be extended to type constructors of kinds $* \to * \to *$, $* \to * \to * \to *$ and so on.

8.2 Proof of Theorem 6

First, we state a free theorem for functions of the type mentioned in Definition 4.

Lemma 3 (A Free Theorem). Let $f :: \tau$ be a function in which L and R are abstract, and τ' be a type $\tau[\ell/L, r/R]$. For any $\mathcal{F} :: \mathsf{Pred}(L)$ and $\mathcal{M} :: \mathsf{Pred}(R)$ satisfying the following conditions:

- $\mathsf{lift} \in \forall\mathcal{T}.\forall\mathcal{U}.\mathit{Lens}\ \mathcal{T}\ \mathcal{U} \to (\forall\mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U})$.
- $\mathsf{unlift} \in \forall\mathcal{T}.\forall\mathcal{U}.(\forall\mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U}) \to \mathit{Lens}\ \mathcal{T}\ \mathcal{U}$.
- $\mathsf{unit} \in \forall\mathcal{T}.\mathit{Lens}\ \mathcal{T}\ ()$.
- $(\circledast) \in \forall\mathcal{T}.\forall\mathcal{U}.\forall\mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U} \to \mathcal{F}\ \mathcal{S}\ (\mathcal{T},\mathcal{U})$.
- $\mathsf{unlift}_2 \in \forall\mathcal{T}.\forall\mathcal{U}.\forall\mathcal{R}.(\forall\mathcal{S}.(\mathcal{F}\ \mathcal{S}\ \mathcal{T}, \mathcal{F}\ \mathcal{S}\ \mathcal{U}) \to \mathcal{F}\ \mathcal{S}\ \mathcal{R}) \to \mathit{Lens}\ (\mathcal{T},\mathcal{U})\ \mathcal{R}$.
- $\mathsf{observe} \in \forall\mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{M}\ \mathcal{S}\ \mathcal{T}$ for any unary relation $\mathcal{T}$ on a type with decidable semantic equality $(==)$.
- $\mathsf{unliftM} \in \forall\mathcal{T}.\forall\mathcal{U}.(\forall\mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{M}\ \mathcal{S}\ (\mathcal{F}\ \mathcal{S}\ \mathcal{U})) \to \mathit{Lens}\ \mathcal{T}\ \mathcal{U}$.
- $\mathsf{unliftM}_2 \in \forall\mathcal{T}.\forall\mathcal{U}.\forall\mathcal{R}.(\forall\mathcal{S}.(\mathcal{F}\ \mathcal{S}\ \mathcal{T}, \mathcal{F}\ \mathcal{S}\ \mathcal{U}) \to \mathcal{M}\ \mathcal{S}\ (\mathcal{F}\ \mathcal{S}\ \mathcal{R})) \to \mathit{Lens}\ (\mathcal{T},\mathcal{U})\ \mathcal{R}$.

we have $f \in [\![\tau']\!]_{\{\ell\mapsto\mathcal{F}, r\mapsto\mathcal{M}\}}$. □

Thanks to the abstract nature of L and R in f, we can use Lemma 3. Concretely, we use the following $\mathcal{F}$ and $\mathcal{M}$.

$$\mathcal{F}\ \mathcal{S}\ \mathcal{R} = \{\ell \mid \ell \in L\ \mathcal{S}\ \mathcal{R}, \ell \text{ is locally well-behaved}\}$$

$$\mathcal{M}\mathcal{S}\mathcal{R} = \left\{ R\ m \,\middle|\, \begin{array}{l} \forall s \in \mathcal{S}.\ \text{let}\ (x,p)\ \text{be}\ m\ s, \\ \qquad x \in \mathcal{R} \land \\ \qquad p\ s = \mathit{True} \land \\ \qquad \forall s' \in \mathcal{S}.\ p\ s' = \mathit{True} \Rightarrow m\ s = m\ s' \end{array} \right\}$$

It is worth noting that $\mathcal{F}\ \mathcal{S}\ \mathcal{R} \subseteq L\ \mathcal{S}\ \mathcal{R}$ and $\mathcal{M}\ \mathcal{S}\ \mathcal{R} \subseteq R\ \mathcal{S}\ \mathcal{R}$.

Assume that all the conditions required by Lemma 3 are fulfilled. Then, for a function f of type $\forall s.\,(L\ s\ A, L\ s\ B) \to R\ s\ (L\ s\ C)$ in which L and R are abstract, we have $f \in (\mathcal{F}\ \mathcal{S}\ A, \mathcal{F}\ \mathcal{S}\ B) \to \mathcal{M}\ \mathcal{S}\ (\mathcal{F}\ \mathcal{S}\ C)$ for any predicate $\mathcal{S}$. Since $\mathit{fst}'_{\mathrm{L}}$ belongs to $\mathcal{F}\ (\mathit{Tag}\ A, \mathit{Tag}\ B)\ A$ and $\mathit{snd}'_{\mathrm{L}}$ belongs to $\mathcal{F}\ (\mathit{Tag}\ A, \mathit{Tag}\ B)\ B$, we have that ℓ in the definition of $\mathit{mkLens}\ f\ s$ called by $\mathsf{unliftM}_2\, f$ is locally well-behaved for all s, and thus ℓ' is well-behaved by Lemma 2. Since $p\ (\mathit{get}\ \mathit{tag2}_{\mathrm{L}}\ s) == \mathit{True}$ holds from the definition of $\mathcal{M}$, $\mathit{mkLens}\ f\ s$ satisfies acceptability. Since $\mathit{put}\ (\mathit{mkLens}\ f\ s)$ is less defined than $\mathit{put}\ \ell'$, $\mathit{mkLens}\ f\ s$ satisfies consistency. This means that $\mathit{mkLens}\ f\ s$ is well-behaved for any s. We are left to show $\mathsf{unliftM}_2\ f$ is well-behaved. Although the acceptability of $\mathsf{unliftM}_2\, f$ comes almost directly from the acceptability of *mkLens f s*, more effort is needed to show the consistency of $\mathsf{unliftM}_2\, f$. Notice that this is the main difference between Theorem 6 and Theorem 4 *after* application of the free theorem.

Here, the last line of $\mathcal{M}$ plays an important role. Assume that $put\ (mkLens\, f\ s)\ s\ v$ succeeds in s'. We have $p\ (get\ tag2_{\mathrm{L}}\ s) = p\ (get\ tag2_{\mathrm{L}}\ s') = True$ for p in the definition of *mkLens f s*. Then, by the definition of $\mathcal{M}$, we have that $(\mathbf{let}\ R\ m = f\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}})\ \mathbf{in}\ (get\ tag2_{\mathrm{L}}\ s))$ is equal to $(\mathbf{let}\ R\ m = f\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}})\ \mathbf{in}\ (get\ tag2_{\mathrm{L}}\ s'))$ by $f\ (fst'_{\mathrm{L}}, snd'_{\mathrm{L}}) \in \mathcal{M}\ \mathcal{S}\ (\mathcal{F}\ \mathcal{S}\ C)$. The rest of computation of *mkLens f s* does not depend on s, and thus $mkLens\, f\ s = mkLens\, f\ s'$ holds. Therefore, we have

$$
\begin{array}{ll}
 & get\ (\mathsf{unliftM}_2\, f)\ (put\ (\mathsf{unliftM}_2\, f)\ s\ v) \\
= & \{\ put\ (mkLens\, f\ s)\ s\ v \text{ succeeds in } s'\ \} \\
 & get\ (mkLens\, f\ s')\ (put\ (mkLens\, f\ s)\ s\ v) \\
= & \{\ \text{the above discussion}\ \} \\
 & get\ (mkLens\, f\ s)\ (put\ (mkLens\, f\ s)\ s\ v) \\
= & \{\ \text{the consistency of } mkLens\, f\ s\ \} \\
 & v
\end{array}
$$

which proves the consistency of $\mathsf{unliftM}_2\, f$.

Now, we go back to show that the conditions in Lemma 3 are actually fulfilled for $\mathcal{F}$ and $\mathcal{M}$.

For the cases of lift and $(\circledast)$, we just use Lemma 2. Here, we have used the assumption that lift is applied only to well-behaved lenses.

For the case of unit, the proof is obvious.

For the cases of unlift, unlift_2, $\mathsf{unliftM}$ and $\mathsf{unliftM}_2$, the proofs are straightforward because $\mathcal{F}\ \mathcal{S}\ \mathcal{R}$ is a subset of $Lens\ \mathcal{S}\ \mathcal{R}$.

For the case of $\mathsf{observe}$, the proof is still straightforward. The last two lines of $\mathcal{M}$ are obtained from the fact that $(==)$ is semantic equality. □

Note that, to prove correctness also for the datatype-generic unlifting functions like $\mathsf{unliftT}$ and $\mathsf{unliftMT}$, we need to keep an additional invariant that a lens ℓ in $\mathcal{F}\ \mathcal{S}\ \mathcal{R}$ must be shape-preserving if S of $\mathcal{S} :: \mathsf{Pred}(S)$ has a shape (recall that *get* and *put* are defined separately also for these datatype-generic functions, and thus similar discussions to $\mathsf{unliftM}$ are required for them). The above proof still works for this case.

9 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. In addition, we also discuss the partiality of derived backward transformations.

9.1 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 (Matsuda *et al.* 2007). Different flavors of bidirectionalization have been proposed: syntactic (Matsuda *et al.* 2007), semantic (Matsuda & Wang 2013, 2014; Voigtländer 2009a; Wang & Najd 2014), and a combination of the two (Voigtländer *et al.* 2010, 2013). 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 state 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 (Voigtländer 2009a) can be straightforwardly implemented in our framework as follows.

$$\begin{array}{l} \mathit{bff} :: (\forall a. [a] \to [a]) \to (\mathit{Eq}\ a \Rightarrow \mathit{Lens}\ [a]\ [a]) \\ \mathit{bff}\ f = \mathsf{unlift}_{\text{list}}\ (\mathit{lsequence}_{\text{list}} \circ f) \end{array}$$

Replacing $\mathsf{unlift}_{\text{list}}$ and $\mathit{lsequence}_{\text{list}}$ with $\mathsf{unliftT}$ and *lsequence*, we also obtain the datatype generic version (Voigtländer 2009a).

With the addition of observe and the monadic unlifting functions, we are also able to cover extensions of semantic bidirectionalization (Matsuda & Wang 2013, 2014) in a simpler and more fundamental way. For example, liftO_2 (and other n-ary observations-lifting functions) has to be a primitive function previously (Matsuda & Wang 2013, 2014), but can now be derived from observe, lift and $(\circledast)$ in our framework.

Our work's unique ability to combine 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, the XML transformation in Section 7 (Q5 of Use Case "STRING" in XML Query Use Cases), which involves concatenation of strings in the transformation, can be handled by our technique, but not previously with bidirectionalization (Matsuda & Wang 2013, 2014; Voigtländer 2009a; Wang & Najd 2014). We believe that with the results in this paper, all queries in XML Query Use Case can now be bidirectionalized. In a sense we are a step forward to the best of both worlds: gaining convenience in programming without losing expressiveness.

The handling of observation in this paper follows the idea of our previous work (Matsuda & Wang 2013, 2014) to record only the observations that actually happened at run-time, not those that may. The latter approach used in (Voigtländer 2009a; Wang & Najd 2014) has the advantage of not requiring a monad, but at the same time is not applicable to monomorphic transformations, as the set of the possible observation results is generally infinite due to lifted lens functions.

9.2 Functional Representation of Bidirectional Transformations

There exists another functional representation of lenses known as the van Laarhoven representation (O'Connor 2011; van Laarhoven 2009). This representation, adopted by the

Haskell library `lens`, encodes bidirectional transformations of type *Lens A B* as functions of the following type.

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

Intuitively, we can read $A \to f\, A$ as updates on A and a lens in this representation maps updates on B (view) to updates on A (source), resulting in a "put-back based" style of programming (Ko *et al.* 2016; Pacheco *et al.* 2014b). The van Laarhoven representation also has its root in the Yoneda Lemma (Jaskelioff & O'Connor 2015; Milewski 2013); unlike ours which applies the Yoneda Lemma to *Lens* $(-)$ V, they apply the Yoneda Lemma to a functor $(V, V \to (-))$. Note that the lens type *Lens S V* is isomorphic to the type $S \to (V, V \to S)$.

Compared to our approach, the van Laarhoven representation is rather inconvenient for applicative-style programming. It cannot be used to derive a *put* when a *get* is already given, as in bidirectionalization (Matsuda & Wang 2013, 2014; Matsuda *et al.* 2007; Voigtländer 2009a; Voigtländer *et al.* 2010, 2013; Wang & Najd 2014) and the classical view update problem (Bancilhon & Spyratos 1981; Dayal & Bernstein 1982; Fegaras 2010; Hegner 1990), especially in a higher-order setting. In the van Laarhoven representation, a bidirectional transformation $\ell :: \mathit{Lens}\, A\, B$, which has $\mathit{get}\, \ell :: A \to B$, is represented as a function from some B structure to some A structure. This difference in direction poses a significant challenge for higher-order programming, because structures of abstractions and applications are not preserved by inverting the direction of $\to$. In contrast, our construction of *put* from *get* is straightforward; replacing base type operations with the lifted bidirectional versions suffices as shown in the $\mathit{unlines}_{\mathrm{L}}$ and $\mathit{eval}_{\mathrm{L}}$ examples (monadification is only needed when supporting observations). Moreover, the van Laarhoven representation does not extend well to data structures: n-ary functions in the representation do not correspond to n-ary lenses. As a result, the van Laarhoven representation itself is not useful to write bidirectional programs such as $\mathit{unlines}_{\mathrm{L}}$ and $\mathit{eval}_{\mathrm{L}}$. Actually as far as we are aware, higher-order programming with the van Laarhoven representation has not been achieved before.

By using the Yoneda embedding, we obtained the *covariant* monoidal functor *Lens S* $(-)$ that maps lenses of type *Lens A B* to functions $\mathit{Lens}\, S\, A \to \mathit{Lens}\, S\, B$, where S is a *Poset* instance (Section 3.4). This is not the only way to use the Yoneda embedding. It is worth mentioning that, by using the Yoneda embedding, we can also obtain a *contravariant* monoidal functor *Lens* $(-)$ V that maps lenses *Lens A B* to functions $\mathit{Lens}\, B\, V \to \mathit{Lens}\, A\, V$, where V is a monoid satisfying certain conditions. A similar idea can be found in Rajkumar *et al.* (2013), where they use contravariant functors over the category of lenses as an abstraction for bidirectional web forms, or formlenses.

9.3 Partiality of Backward Transformation

Unlike the original lens framework (Foster *et al.* 2007) and their extensions (Bohannon *et al.* 2008; Foster *et al.* 2008) that guarantee the totality of backward transformations, our derived backward transformations are generally partial, similar to the case in bidirectionalization (Matsuda & Wang 2013, 2014; Matsuda *et al.* 2007; Voigtländer 2009a; Voigtländer *et al.* 2010, 2013; Wang & Najd 2014). Being total has the clear advantage that the backward transformations never fail, but at the same time, the totality requirement poses strong restrictions on recursive definitions. For example, even for simple fold-like *get* functions, totality, i.e., termination, of the corresponding *put* functions is already non-trivial to guarantee, as such *put*s are usually implemented by "unfold" (Wang *et al.* 2010). As a result, the Boomerang framework of lenses (Bohannon *et al.* 2008) only supports map-like functions, leaving out other recursion patterns.

In our approach, instead of guaranteeing totality at the expense of expressiveness, we aim to reflect the partiality through types. For example, the type of $unlines_{\mathrm{F}}$ in Example 3, $\forall s.\,[L\;s\;String] \to L\;s\;String$, tells that the shape of a list cannot be changed, while each list element is updatable. But this indication is not perfect. For updatable data as permitted by its type, there may still be failures coming from three sources:

- Non-linear use of updatable variables (by $\curlyvee$).
- Lifting of non-total lenses (by ℓ of lift ℓ).
- Changed observation (by p of $\mathsf{unliftM}/\mathsf{unliftM}_2$).

The first two cases are rather predictable, even though identifying the first case would require some form of linearity analysis. For the last case, since the *R* monad in Section 6.2 essentially records the performed observations, there is the possibility to include diagnostic information when failure happens for improved understandability. Notice that recursion itself does not affect updatability in our framework: if a recursion does not terminate, it just means that no lens is constructed, rather than one with a partial *put*.

9.4 Closedness of Lifted Combinators

In Section 6.1, we looked at the lifting of lens combinators (in contrast to lifting of lenses) and mentioned that there is a closedness restriction on the argument of liftC, which in some cases severely restricts the programming style.

A recent work by the authors aims to address this problem in a standalone bidirectional language named HOBiT (Matsuda & Wang 2018). In HOBiT, lens combinators can be lifted to language constructs with binders, which have no closedness restriction. To achieve this, it uses an explicit variable environment for unlifting (which roughly speaking is the counterpart of the *s* in *L s a* in this paper). The explicit nature of the evironment opens it to complex manipulations, which are required for removing the closedness restriction. But it also means that an embedded implementation is no longer straightforward.

10 Conclusion

We have proposed a novel framework of applicative bidirectional programming, which features the strengths of lenses (Bohannon *et al.* 2008; Foster *et al.* 2007, 2008) and semantic bidirectionalization (Matsuda & Wang 2013, 2014; Voigtländer 2009a; Wang & Najd 2014). 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 to handle shape updates. It is important to relax the restriction that only closed expressions can be unlifted to enable more practical programming. A possible solution to this problem would be to abstract certain kind of containers in addition to base-type values, which is likely to lead to a more fine-grained treatment of lens combinators and shape updates.

Acknowledgments

We would like to thank Shin-ya Katsumata, Makoto Hamana, Kazuyuki Asada, and Patrik Jansson for their helpful comments on categorical discussions in this paper. Especially, Shin-ya Katsumata and Makoto Hamana pointed out the relationship from a preliminary version of our method to the Yoneda lemma. We would like to thank Oleg Kiselyov for his informative comments on higher-order abstract syntax. We also would like to thank 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 University of Kent, UK.

References

Bancilhon, François, & Spyratos, Nicolas. (1981). Update semantics of relational views. *Acm trans. database syst.*, **6**(4), 557–575.

Barbosa, Davi M. J., Cretin, Julien, Foster, Nate, Greenberg, Michael, & Pierce, Benjamin C. (2010). Matching lenses: alignment and view update. *Pages 193–204 of:* Hudak, Paul, & Weirich, Stephanie (eds), *ICFP*. ACM.

Bernardy, Jean-Philippe, Jansson, Patrik, & Paterson, Ross. (2012). Proofs for free - parametricity for dependent types. *J. funct. program.*, **22**(2), 107–152.

Bird, Richard S., Gibbons, Jeremy, Mehner, Stefan, Voigtländer, Janis, & Schrijvers, Tom. (2013). Understanding idiomatic traversals backwards and forwards. *Pages 25–36 of:* chieh Shan, Chung (ed), *Haskell*. ACM.

Bohannon, Aaron, Foster, J. Nathan, Pierce, Benjamin C., Pilkiewicz, Alexandre, & Schmitt, Alan. (2008). Boomerang: resourceful lenses for string data. *Pages 407–419 of:* Necula, George C., & Wadler, Philip (eds), *POPL*. ACM.

Church, Alonzo. (1940). A formulation of the simple theory of types. *J. symb. log.*, **5**(2), 56–68.

Dayal, Umeshwar, & Bernstein, Philip A. (1982). On the correct translation of update operations on relational views. *Acm trans. database syst.*, **7**(3), 381–416.

Ellis, Tom. 2012 (Sep). *Category and lenses*. Blog post: `http://web.jaguarpaw.co.uk/~tom/blog/posts/2012-09-30-category-and-lenses.html`.

Fegaras, Leonidas. (2010). Propagating updates through XML views using lineage tracing. *Pages 309–320 of:* Li, Feifei, Moro, Mirella M., Ghandeharizadeh, Shahram, Haritsa, Jayant R., Weikum, Gerhard, Carey, Michael J., Casati, Fabio, Chang, Edward Y., Manolescu, Ioana, Mehrotra, Sharad, Dayal, Umeshwar, & Tsotras, Vassilis J. (eds), *ICDE*. IEEE.

Foster, J. Nathan, Greenwald, Michael B., Moore, Jonathan T., Pierce, Benjamin C., & Schmitt, Alan. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *Acm trans. program. lang. syst.*, **29**(3).

Foster, J. Nathan, Pilkiewicz, Alexandre, & Pierce, Benjamin C. (2008). Quotient lenses. *Pages 383–396 of:* Hook, James, & Thiemann, Peter (eds), *ICFP*. ACM.

Foster, Nate, Matsuda, Kazutaka, & Voigtländer, Janis. (2010). Three complementary approaches to bidirectional programming. *Pages 1–46 of:* Gibbons, Jeremy (ed), *SSGIP*. Lecture Notes in Computer Science, vol. 7470. Springer.

Hayashi, Yasushi, Liu, Dongxi, Emoto, Kento, Matsuda, Kazutaka, Hu, Zhenjiang, & Takeichi, Masato. (2007). A web service architecture for bidirectional XML updating. *Pages 721–732 of:* Dong, Guozhu, Lin, Xuemin, Wang, Wei, Yang, Yun, & Yu, Jeffrey Xu (eds), *APWeb/WAIM*. Lecture Notes in Computer Science, vol. 4505. Springer.

Hegner, Stephen J. (1990). Foundations of canonical update support for closed database views. *Pages 422–436 of:* Abiteboul, Serge, & Kanellakis, Paris C. (eds), *ICDT*. Lecture Notes in Computer Science, vol. 470. Springer.

Hidaka, Soichiro, Hu, Zhenjiang, Inaba, Kazuhiro, Kato, Hiroyuki, Matsuda, Kazutaka, & Nakano, Keisuke. (2010). Bidirectionalizing graph transformations. *Pages 205–216 of:* Hudak, Paul, & Weirich, Stephanie (eds), *ICFP*. ACM.

Hofmann, Martin, Pierce, Benjamin C., & Wagner, Daniel. (2011). Symmetric lenses. *Pages 371–384 of:* Ball, Thomas, & Sagiv, Mooly (eds), *POPL*. ACM.

Hu, Zhenjiang, Mu, Shin-Cheng, & Takeichi, Masato. (2004). A programmable editor for developing structured documents based on bidirectional transformations. *Pages 178–189 of:* Heintze, Nevin, & Sestoft, Peter (eds), *PEPM*. ACM.

Huet, Gérard P., & Lang, Bernard. (1978). Proving and applying program transformations expressed with second-order patterns. *Acta inf.*, **11**, 31–55.

Jaskelioff, Mauro, & O'Connor, Russell. (2015). A representation theorem for second-order functionals. *Journal of functional programming*, **25**(e13), 1–36.

Ko, Hsiang-Shang, Zan, Tao, & Hu, Zhenjiang. (2016). Bigul: a formally verified core language for putback-based bidirectional programming. *Pages 61–72 of:* Erwig, Martin, & Rompf, Tiark (eds), *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM.

Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electr. notes theor. comput. sci.*, **229**(5), 97–117.

Liu, Dongxi, Hu, Zhenjiang, & Takeichi, Masato. (2007). Bidirectional interpretation of XQuery. *Pages 21–30 of:* Ramalingam, G., & Visser, Eelco (eds), *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. ACM.

Mac Lane, Saunders. (1998). *Categories for the working mathematician*. Second edition edn. Graduate Texts in Matheematics, vol. 5. Springer.

Matsuda, Kazutaka, & Wang, Meng. (2013). Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem. *Pages 297–308 of:* Peña, Ricardo, & Schrijvers, Tom (eds), *PPDP*. ACM.

Matsuda, Kazutaka, & Wang, Meng. (2014). "Bidirectionalization for free" for monomorphic transformations. *Science of computer programming*, **111**(1), 79–109. DOI: `10.1016/j.scico.2014.07.008`.

Matsuda, Kazutaka, & Wang, Meng. (2015). Applicative bidirectional programming with lenses. *Pages 62–74 of:* Fisher, Kathleen, & Reppy, John H. (eds), *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM.

Matsuda, Kazutaka, & Wang, Meng. (2018). HOBiT: Programming lenses without using lens combinators. *ESOP*. Accepted.

Matsuda, Kazutaka, Hu, Zhenjiang, Nakano, Keisuke, Hamana, Makoto, & Takeichi, Masato. (2007). Bidirectionalization transformation based on automatic derivation of view complement functions. *Pages 47–58 of:* Hinze, Ralf, & Ramsey, Norman (eds), *ICFP*. ACM.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *J. funct. program.*, **18**(1), 1–13.

Milewski, Bartosz. 2013 (Oct). *Lenses, Stores, and Yoneda*. blog post: `http://bartoszmilewski.com/2013/10/08/lenses-stores-and-yoneda/`.

Miller, Dale, & Nadathur, Gopalan. (1987). A logic programming approach to manipulating formulas and programs. *Pages 379–388 of: Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987*. IEEE-CS.

Mu, Shin-Cheng, Hu, Zhenjiang, & Takeichi, Masato. (2004). An algebraic approach to bi-directional updating. *Pages 2–20 of:* Chin, Wei-Ngan (ed), *APLAS*. Lecture Notes in Computer Science, vol. 3302. Springer.

O'Connor, Russell. (2011). Functor is to lens as applicative is to biplate: Introducing multiplate. *Corr*, **abs/1103.2841**. Accepted in WGP'11, but not included in its proceedings.

Pacheco, Hugo, Zan, Tao, & Hu, Zhenjiang. (2014a). Biflux: A bidirectional functional update language for XML. *Pages 147–158 of:* Chitil, Olaf, King, Andy, & Danvy, Olivier (eds), *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*. ACM.

Pacheco, Hugo, Hu, Zhenjiang, & Fischer, Sebastian. (2014b). Monadic combinators for "putback" style bidirectional programming. *Pages 39–50 of:* Chin, Wei-Ngan, & Hage, Jurriaan (eds), *PEPM*. ACM.

Paterson, Ross. (2001). A new notation for arrows. *Pages 229–240 of:* Pierce, Benjamin C. (ed), *ICFP*. ACM.

Paterson, Ross. (2012). Constructing applicative functors. *Pages 300–323 of:* Gibbons, Jeremy, & Nogueira, Pablo (eds), *MPC*. Lecture Notes in Computer Science, vol. 7342. Springer.

Pfenning, Frank, & Elliott, Conal. (1988). Higher-order abstract syntax. *Pages 199–208 of:* Wexelblat, Richard L. (ed), *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. ACM.

Rajkumar, Raghu, Lindley, Sam, Foster, Nate, & Cheney, James. (2013). *Lenses for web data*. In Preliminary Proceedings of Second International Workshop on Bidirectional Transformations (BX 2013).

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of:* Mason, R.E.A. (ed), *Information Processing*. Elsevier Science Publishers B.V. (North-Holland).

van Laarhoven, Twan. 2009 (Jul). *CPS based functional references*. blog post: `http://www.twanvl.nl/blog/haskell/cps-functional-references`.

Voigtländer, Janis. (2009a). Bidirectionalization for free! (pearl). *Pages 165–176 of:* Shao, Zhong, & Pierce, Benjamin C. (eds), *POPL*. ACM.

Voigtländer, Janis. (2009b). Free theorems involving type constructor classes: functional pearl. *Pages 173–184 of:* Hutton, Graham, & Tolmach, Andrew P. (eds), *ICFP*. ACM.

Voigtländer, Janis, Hu, Zhenjiang, Matsuda, Kazutaka, & Wang, Meng. (2010). Combining syntactic and semantic bidirectionalization. *Pages 181–192 of:* Hudak, Paul, & Weirich, Stephanie (eds), *ICFP*. ACM.

Voigtländer, Janis, Hu, Zhenjiang, Matsuda, Kazutaka, & Wang, Meng. (2013). Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *J. funct. program.*, **23**(5), 515–551.

Vytiniotis, Dimitrios, & Weirich, Stephanie. (2010). Parametricity, type equality, and higher-order polymorphism. *J. funct. program.*, **20**(2), 175–210.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: FPCA*.

Wallace, Malcolm, & Runciman, Colin. (1999). Haskell and XML: Generic combinators or type-based translation? *Pages 148–159 of:* Rémy, Didier, & Lee, Peter (eds), *ICFP*. ACM.

Wang, Meng, & Najd, Shayan. (2014). Semantic bidirectionalization revisited. *Pages 51–62 of:* Chin, Wei-Ngan, & Hage, Jurriaan (eds), *PEPM*. ACM.

Wang, Meng, Gibbons, Jeremy, Matsuda, Kazutaka, & Hu, Zhenjiang. (2010). Gradual refinement: Blending pattern matching with data abstraction. *Pages 397–425 of:* Bolduc, Claude, Desharnais, Jules, & Ktari, Béchir (eds), *MPC*. Lecture Notes in Computer Science, vol. 6120. Springer.

Wang, Meng, Gibbons, Jeremy, & Wu, Nicolas. (2011). Incremental updates for efficient bidirectional transformations. *Pages 392–403 of:* Chakravarty, Manuel M. T., Hu, Zhenjiang, & Danvy, Olivier (eds), *ICFP*. ACM.

Wang, Meng, Gibbons, Jeremy, Matsuda, Kazutaka, & Hu, Zhenjiang. (2013). Refactoring pattern matching. *Sci. comput. program.*, **78**(11), 2216–2242.

Xiong, Yingfei, Liu, Dongxi, Hu, Zhenjiang, Zhao, Haiyan, Takeichi, Masato, & Mei, Hong. (2007). Towards automatic model synchronization from model transformations. *Pages 164–173 of:* Stirewalt, R. E. Kurt, Egyed, Alexander, & Fischer, Bernd (eds), *ASE*. ACM.

Yu, Yijun, Lin, Yu, Hu, Zhenjiang, Hidaka, Soichiro, Kato, Hiroyuki, & Montrieux, Lionel. (2012). Maintaining invariant traceability through bidirectional transformations. *Pages 540–550 of:* Glinz, Martin, Murphy, Gail C., & Pezzè, Mauro (eds), *ICSE*. IEEE.

A Proof of Lemma 1

The proof is based on free theorems (the standard binary version) (Reynolds 1983; Voigtländer 2009b; Wadler 1989).

The difficulty of the proof lies in the treatment of unlift. Usually, a proof based on free theorems is done by encoding relationship between two arguments (e.g., ℓ and id_{L}) of a polymorphic function to a relation, and then by using the fact that such a polymorphic function preserves the relation. Here, in addition, we have to prove that lift and unlift preserve the relation because f can use lift and unlift internally. Our proof obligation for unlift is that two *arbitrary* polymorphic functions g_1 and g_2 that preserve the relation satisfies that $g_1\ id_{\mathrm{L}} = g_2\ id_{\mathrm{L}}$. That is, it might seem that the relation must contain the pair (ℓ, id_{L}) and must be diagonal at the same time, which appears contradictory. Very roughly speaking, this difficulty comes from the fact that we have to encode *two* different goals, $f\ \ell = f\ id_{\mathrm{L}} \mathbin{\hat{\circ}} \ell$ and $g_1\ id_{\mathrm{L}} = g_2\ id_{\mathrm{L}}$ where f, g_1 and g_2 are of the same polymorphic type, to *one* relation. To overcome the problem, we use the polymorphic nature of s and the fact that such a relation can depend on the choice of s, which is the reason why our proof becomes tricky.

A.1 Free Theorems (Binary Version)

We write $\mathcal{R} :: A_1 \leftrightarrow A_2$ if $\mathcal{R}$ is a binary relation between A_1 and A_2. For relations $\mathcal{R} :: A_1 \leftrightarrow A_2$ and $\mathcal{R}' :: B_1 \leftrightarrow B_2$, we abuse the notation to write $\mathcal{R} \to \mathcal{R}' :: (A_1 \to B_1) \leftrightarrow (A_2 \to B_2)$ for the relation $\{(f_1, f_2) \mid \forall (x_1, x_2) \in \mathcal{R}.\, (f_1\, x_1, f_2\, x_2) \in \mathcal{R}'\}$, and $(\mathcal{R}, \mathcal{R}') :: (A_1, B_1) \leftrightarrow (A_2, B_2)$ for $\{((x_1, y_1), (x_2, y_2)) \mid (x_1, x_2) \in \mathcal{R}, (y_1, y_2) \in \mathcal{R}'\}$.

We introduce a (binary) *relational interpretation* $[\![\tau]\!]^2_\rho$ of types, where ρ is a mapping from type variable to binary relations, as follows.

$$
\begin{array}{lll}
[\![a]\!]^2_\rho & = \rho(a) & \\
[\![B]\!]^2_\rho & = \{(e,e) \mid e :: B\} & \text{if } B \text{ is a base type} \\
[\![T_1 \to T_2]\!]^2_\rho & = [\![T_1]\!]^2_\rho \to [\![T_2]\!]^2_\rho & \\
[\![(T_1, T_2)]\!]^2_\rho & = ([\![T_1]\!]^2_\rho, [\![T_2]\!]^2_\rho) & \\
[\![\forall a.\, T]\!]^2_\rho & = \left\{ (u,v) \,\middle|\, \forall \mathcal{R} :: S_1 \leftrightarrow S_2.\, (u_{S_1}, v_{S_2}) \in [\![T]\!]^2_{\rho[a \mapsto \mathcal{R}]} \right\} &
\end{array}
$$

Here, $\rho[a \mapsto \mathcal{R}]$ is an extension of ρ with $a \mapsto \mathcal{R}$. If $\rho = \emptyset$, we sometimes write $[\![T]\!]^2$ instead of $[\![T]\!]^2_\emptyset$. Similarly to the unary case, we write $[\![\forall \alpha.\tau]\!]^2$ as $\forall \mathcal{R}.\mathcal{F}$ where $\mathcal{F}$ is the interpretation $[\![\tau]\!]^2_{\{\alpha \mapsto \mathcal{R}\}}$. For a base type B, we also write B for $[\![B]\!]^2$.

Then, parametricity states that, for a closed term f of a closed type τ, $(f, f) \in [\![\tau]\!]^2$ holds. Free theorems are theorems obtained by instantiating parametricity.

Next, we introduce the binary-version of *relational action* (Voigtländer 2009b).

Definition 6 ((Binary) Relational Action). For type constructors κ_1 and κ_2, $\mathcal{F}$ is called a *relational action* between κ_1 and κ_2, denoted by $\mathcal{F} :: \kappa_1 \leftrightarrow \kappa_2$, if $\mathcal{F}$ maps any relation $\mathcal{R} :: \tau_1 \leftrightarrow \tau_2$ for every pair of closed types τ_1 and τ_2 to $\mathcal{F}\mathcal{R} :: \kappa_1\, \tau_1 \leftrightarrow \kappa_2\, \tau_2$. □

Accordingly, the relational interpretations are extended as:

$$
\begin{array}{ll}
[\![\kappa]\!]^2_\rho & = \rho(\kappa) \\
[\![\tau_1\ \tau_2]\!]^2 & = [\![\tau_1]\!]^2_\rho\ [\![\tau_2]\!]^2_\rho \\
[\![\forall \kappa.\tau]\!]^2 & = \left\{ (u,v) \,\middle|\, \forall \mathcal{F} : \kappa_1 \leftrightarrow \kappa_2.\, (u_{\kappa_1}, v_{\kappa_2}) \in [\![\tau]\!]^2_{\rho[\kappa \mapsto \mathcal{F}]} \right\}
\end{array}
$$

Parametricity holds also for this relational interpretation (Bernardy *et al.* 2012; Vytiniotis & Weirich 2010). Here, κ, κ_1 and κ_2 are type constructors of kind $* \to *$, and thus the quantified $\mathcal{F}$ is a relational action. The notation of relational action can be extended to type constructors of kinds $* \to * \to *, * \to * \to * \to *$ and so on.

Also for binary relations $\mathcal{R}$ and $\mathcal{S}$, we write $\mathit{Lens}\ \mathcal{R}\ \mathcal{S}$ for $(\mathcal{R} \to \mathcal{S}, \mathcal{R} \to \mathcal{S} \to \mathcal{R})$. The following lemma holds for *Lens*.

Lemma 4. For binary relations $\mathcal{R}$, $\mathcal{S}$ and $\mathcal{T}$, if $(f_1, f_2) \in \mathit{Lens}\ \mathcal{R}\ \mathcal{S}$ and $(g_1, g_2) \in \mathit{Lens}\ \mathcal{S}\ \mathcal{T}$, then $(g_1 \mathbin{\hat{\circ}} f_1, g_2 \mathbin{\hat{\circ}} f_2) \in \mathit{Lens}\ \mathcal{R}\ \mathcal{T}$. □

A.2 Proof

Let us consider a function f of type $\forall s. \mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B$ in which *Lens* is abstract. This means that we have a function h

$$\begin{aligned}&\forall \ell. (\forall a\, b. \mathit{Lens}\ a\ b \to (\forall s. \ell\ s\ a \to \ell\ s\ b))\\ &\qquad \to (\forall a\, b. (\forall s. \ell\ s\ a \to \ell\ s\ b) \to \mathit{Lens}\ a\ b)\\ &\qquad \to \forall s. \ell\ s\ A \to \ell\ s\ B\end{aligned}$$

such that $f = h\ \mathsf{lift}\ \mathsf{unlift}$.

For functions of the type, we have the following free theorem.

Lemma 5 (A Free Theorem). Let f be a function of type $\forall s. \mathit{Lens}\ s\ A \to \mathit{Lens}\ s\ B$ in which *Lens* is abstract. Suppose that $\mathcal{F} :: \kappa_1 \leftrightarrow \kappa_2$ is a relational action satisfying the following conditions.

- $(\mathsf{lift}, \mathsf{lift}) \in \forall \mathcal{T}. \forall \mathcal{U}. \mathit{Lens}\ \mathcal{T}\ \mathcal{U} \to (\forall \mathcal{S}. \mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U})$.
- $(\mathsf{unlift}, \mathsf{unlift}) \in \forall \mathcal{T}. \forall \mathcal{U}. (\forall \mathcal{S}. \mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U}) \to \mathit{Lens}\ \mathcal{T}\ \mathcal{U}$.

Then, $(f, f) \in \forall \mathcal{S}. \mathcal{F}\ \mathcal{S}\ A \to \mathcal{F}\ \mathcal{S}\ B$. □

Let $\ell :: \mathit{Lens}\ S_1\ S_2$ be a lens. Then, we define $\mathcal{F}$ as follows.

$$\mathcal{F}\ (\mathcal{S} : A_1 \leftrightarrow A_2)\ (\mathcal{R} : B_1 \leftrightarrow B_2) = \begin{cases} \left\{ (x_1, x_2) \;\middle|\; \begin{array}{l} \exists (z_1, z_2) \in \mathit{Lens}\ S_1\ \mathcal{R}. \\ (x_1, x_2) = (z_1, z_2 \mathbin{\hat{\circ}} \ell) \end{array} \right\} & \text{if } \mathcal{S} = \emptyset :: S_1 \leftrightarrow S_2 \\ \mathit{Lens}\ \mathcal{S}\ \mathcal{R} & \text{otherwise} \end{cases}$$

Notice that we do not require that $\mathcal{F}\ \mathcal{S}\ \mathcal{R} \subseteq \mathit{Lens}\ \mathcal{S}\ \mathcal{R}$ when $\mathcal{S} = \emptyset$. Also notice that $(\ell_1, \ell_2) \in \mathit{Lens}\ \emptyset\ \mathcal{R}$ for any l_1 and l_2 with appropriate types. The complication of $\mathcal{F}$'s definition comes from the two different contexts where f can be instantiated: (1) f in the proof of $f\ \mathit{id}_{\mathrm{L}} \mathbin{\hat{\circ}} \ell = f\ \ell$, and (2) f in the proof of $f\ \mathit{id}_{\mathrm{L}} = f\ \mathit{id}_{\mathrm{L}}$. Also notice that any pair of lenses (ℓ_1, ℓ_2) of appropriate types belongs to $\mathit{Lens}\ \emptyset\ R$, because $\emptyset \to R$ contains any pairs of functions of the giving types.

Assume that the required conditions in Lemma 5 are fulfilled. Then, by the lemma, we have $(f, f) \in \mathcal{F}\ \mathcal{S}\ A \to \mathcal{F}\ \mathcal{S}\ B$ for any $\mathcal{S}$. Taking S_1 as A, we have $(\mathit{id}_{\mathrm{L}}, \ell) \in \mathcal{F}\ \emptyset\ A$ because $(\mathit{id}_{\mathrm{L}}, \mathit{id}_{\mathrm{L}}) \in \mathcal{F}\ A\ A$. Since $(f, f) \in \mathcal{F}\ \emptyset\ A \to \mathcal{F}\ \emptyset\ B$, we have $(f\ \mathit{id}_{\mathrm{L}}, f\ \ell) \in \mathcal{F}\ \emptyset\ B$. Thus, there is a pair (z_1, z_2) that is related by the relation $\mathit{Lens}\ A\ B$ satisfying $f\ \mathit{id}_{\mathrm{L}} = z_1$ and $f\ \ell = z_2 \mathbin{\hat{\circ}} \ell$. Since A and B are diagonal relations on A and B respectively, $A \to B$ is also diagonal, and so does $\mathit{Lens}\ A\ B$. Since $\mathit{Lens}\ A\ B$ is diagonal, we have $z_1 = z_2$. Thus, we obtain $f\ \mathit{id}_{\mathrm{L}} \mathbin{\hat{\circ}} \ell = f\ \ell$.

Case: $(\mathsf{lift}, \mathsf{lift}) \in \forall \mathcal{T}.\forall \mathcal{U}.\mathit{Lens}\ \mathcal{T}\ \mathcal{U} \to (\forall \mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U})$**.** Let $\mathcal{T} :: A_1 \leftrightarrow A_2$ and $\mathcal{U} :: B_1 \leftrightarrow B_2$ be relations. Let $x_1 :: \mathit{Lens}\ A_1\ B_1$ and $x_2 :: \mathit{Lens}\ A_2\ B_2$ be lenses. Let $\mathcal{S} :: C_1 \leftrightarrow C_2$ be a relation. Let (z_1, z_2) be lenses such that $(z_1, z_2) \in \mathcal{F}\ \mathcal{S}\ \mathcal{T}$. Then, by Lemma 4, we have $(\mathsf{lift}\ x_1\ z_1, \mathsf{lift}\ x_2\ z_2) = (x_1 \mathbin{\hat{\circ}} z_1, x_2 \mathbin{\hat{\circ}} z_2) \in \mathcal{F}\ \mathcal{S}\ \mathcal{U}$.

Case: $(\mathsf{unlift}, \mathsf{unlift}) \in \forall \mathcal{T}.\forall \mathcal{U}.(\forall \mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U}) \to \mathit{Lens}\ \mathcal{T}\ \mathcal{U}$**.** Let $\mathcal{T} :: A_1 \leftrightarrow A_2$ and $\mathcal{U} :: B_1 \leftrightarrow B_2$ be relations. Let g_1 and g_2 be functions satisfying $(g_1, g_2) :: \forall \mathcal{S}.\mathcal{F}\ \mathcal{S}\ \mathcal{T} \to \mathcal{F}\ \mathcal{S}\ \mathcal{U}$. Take $\mathcal{S} = \mathcal{T}$. Suppose $\mathcal{T} = \emptyset :: S_1 \leftrightarrow S_2$. Then, we trivially have $(g_1\ \mathit{id}_{\mathrm{L}}, g_2\ \mathit{id}_{\mathrm{L}}) \in \mathit{Lens}\ \emptyset\ \mathcal{U}$. Recall that $(\ell_1, \ell_2) \in \mathit{Lens}\ \emptyset\ \mathcal{U}$ for any ℓ_1 and ℓ_2 with appropriate types. Suppose $\mathcal{T} \neq \emptyset :: S_1 \leftrightarrow S_2$. Then, we have $(\mathit{id}_{\mathrm{L}}, \mathit{id}_{\mathrm{L}}) \in \mathcal{F}\ \mathcal{T}\ \mathcal{T}$ and thus $(g_1\ \mathit{id}_{\mathrm{L}}, g_2\ \mathit{id}_{\mathrm{L}}) \in \mathcal{F}\ \mathcal{T}\ \mathcal{U}$. Since we have assumed $\mathcal{T} \neq \emptyset :: S_1 \leftrightarrow S_2$, we have $(g_1\ \mathit{id}_{\mathrm{L}}, g_2\ \mathit{id}_{\mathrm{L}}) \in \mathit{Lens}\ \mathcal{T}\ \mathcal{U}$. $\square$