Bidirectionalization for Free with Runtime Recording

Or, a Light-Weight Approach to the View-Update Problem

Kazutaka Matsuda
University of Tokyo
kztk@is.s.u-tokyo.ac.jp

Meng Wang
Chalmers University of Technology
wmeng@chalmers.se

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. Over the years, a lot of effort has been made to offer better language support for programming such transformations. In particular, a technique known as *bidirectionalization* is able to analyze and transform unidirectional programs written in general purpose languages, and "bidirectionalize" them.

Among others, a technique termed as semantic bidirectionalization proposed by Voigtländer stands out in term of user-friendliness. The unidirectional program can be written using arbitrary language constructs, as long as the function is polymorphic and the language constructs respect parametricity. The free theorems that follow from the polymorphic type of the program allow a kind of forensic examination of the transformation, determining its effect without examining its implementation. This is convenient, in the sense that the programmer is not restricted to using a particular syntax; but it does require the transformation to be polymorphic.

In this paper, we lift this polymorphism requirement to improve the applicability of semantic bidirectionalization. Concretely, we provide a type class $PackM\ \gamma\ \alpha\ \mu$, which intuitively reads "a concrete datatype γ is abstracted to a type α, and the 'observations' made by a transformation on values of type γ are recorded by a monad μ". With $PackM$, we turn monomorphic transformations into polymorphic ones, that are ready to be bidirectionalized. We demonstrate our technique with a case study of standard XML queries, which were considered beyond semantic bidirectionalization because of their monomorphic nature.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Bidirectional Transformation, Free Theorem, Type Class, Haskell, XML Transformation

1. INTRODUCTION

Bidirectionality is a fundamental aspect of computing: transforming data from one format to another, and requiring a transformation in the opposite direction that is in some sense an inverse. The most well-known instance is the *view-update problem* [1, 6, 7] 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.

Let's consider a (simplified version of) XML example taken from `http://www.w3.org/TR/xquery-use-cases/`: a source (in Figure 1) can be transformed by query Q1 (in Figure 2) to produce a view (Figure 3). Here the query Q1 is the "forward transformation", and a corresponding "backward" transformation maps an updated view back to the source. For example, one may change the title "TCP/IP Illustrated" to "TCP/IP Illustrated (second edition)" in the view and expect the source to be updated accordingly. Things are more interesting with the year of a publication: this attribute's value is observed by the query in producing the view, so whatever changes to it shall not alter the existing observations, to ensure that the view change can be reflected by a source change. For example, we can change the year for the first book to 2000, but not to any value that is less than 1992. The backward transformation is required to correctly register the former valid change, but to reject the latter invalid one.

By dint of hard effort, one can construct separately the forward transformation from source to view together with the corresponding backward transformation. 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 bidirectional. 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

```
<bib>
 <book year="1994">
   <title>TCP/IP Illustrated</title>
   <author>Stevens W.</author>
   <publisher>Addison-Wesley</publisher>
   <price>65.95</price>
 </book>
 <book year="1992">
   <title>Advanced Programming in the Unix environment</title>
   <author>Stevens W.</author>
   <publisher>Addison-Wesley</publisher>
   <price>65.95</price>
 </book>
 <book year="2000">
   <title>Data on the Web</title>
   <author>Abiteboul Serge</author>
   <author>Buneman Peter</author>
   <author>Suciu Dan</author>
   <publisher>Morgan Kaufmann Publishers</publisher>
   <price>39.95</price>
 </book>
</bib>
```

Figure 1: An XML Source

```
<bib>
{
  for $b in doc("http://example.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley"
        and $b/@year > 1991
  return
  <book year="{ $b/@year }">{ $b/title }</book>
}
</bib>
```

Figure 2: Query Q1

in linguistic (mostly functional) approaches to streamlining bidirectional transformations—this is very much a current problem.

Using terminologies advocated by the *lens* framework [8] that traces back to database research: the forward function is commonly known as *get* having type $S \to V$, and the backward one as *put* having type $S \to V \to S$. The idea is that *put*, in addition to an updated view, takes the original source as an input, so that *get* does not have to be bijective to have a backward semantics. As a result, *put* is often partial even for total and surjective *get*. The correctness of the pair of functions is governed by the following *definitional properties* [5]:

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

(In this paper, when we write $e = e'$, we assume that neither e nor e' is undefined.) Here *consistency* (also known as the PutGet law [8]) roughly corresponds to right-invertibility, basically ensuring that all updates on a view are captured by the updated source (the change of year to values less than 1992 in the above example violates this law), and *acceptability* (also known as the GetPut law [8]) roughly corresponds to left-invertibility, prohibiting changes to the source if no update has been made on the view. Bidirectional transformations satisfying the above two laws are sometimes called *well-behaved* [8]. In addition to these definitional properties, some desirable laws such as *composability* and *undoability* are also discussed in the literature [1, 11].

The paradigm of bidirectional programming is about constructing *get* in a bidirectional language and expecting a corresponding *put* to be created automatically. Very often, such languages are defined as a collection of combinators which can be read in two ways [4,8,9,13]: forward and backward. A disadvantage of the combinator-based approach is that transformations have to be encoded in a somewhat inconvenient programming style.

```
<bib>
 <book year="1994">
   <title>TCP/IP Illustrated</title>
 </book>
 <book year="1992">
   <title>Advanced Programming in the Unix environment</title>
 </book>
</bib>
```

Figure 3: Result of Applying Q1 to the Source in Figure 1

Other than constructing special purpose bidirectional languages, an alternative is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as *bidirectionalization* [14]. Different flavors of bidirectionalization have been proposed: syntactic [14], semantic [19], and a combination of the two [21]. Syntactic bidirectionalization inspects a *get* definition written in a somehow restricted syntactic representation and synthesizes a definition for the backward version. Semantic, or extensional, bidirectionalization on the other hand treats polymorphic *get* as an opaque semantic object, applying the function independently to a collection of unique identifiers, and the free theorem [23] arising from parametricity [18] states that whatever happens to those identifiers happens in the same way to any other inputs—this information is sufficient to construct the backward transformation. (We will give more details of the technique in Section 2.) This is convenient, in the sense that the programmer is not confined to a certain syntactic representation; but it does require that the transformation is polymorphic.

This polymorphism requirement has prevented the use of semantic bidirectionalization in many applications, for example XML transformations where queries are predominantly monomorphic. Consider query Q1 we have seen earlier on (Figure 2). The attribute value of *year* and content of *publisher* are compared to constant values, which instantiates their types to monomorphic ones, and the creation of new element *book* is also beyond the reach of the existing techniques of semantic bidirectionalization [19, 21] based on the standard free theorems [23].

In this paper, we propose a novel bidirectionalization approach that circumvents the polymorphism restriction, and allows us to program and bidirectionalize monomorphic transformations in a convenient manner. At the heart of the technique is a type class *PackM* that provides a solution the problems of creation of constants and comparison with constant values. More concretely, the constant values are "created" into equivalent, and yet abstract in type, values, which do not instantiate the type variables when used in comparison. And, through methods of *PackM*, such comparisons are recorded during runtime, which are checked before the backward execution, so that free theorems for correct bidirectionalization can be established.

The rest of the paper is organized as follows. In Section 2, we firstly review the concept of semantic bidirection-

alization [19], and in Section 3, we describe our proposal and the handling of monomorphic transformations. In Section 4, we prove the correctness (consistency and acceptability) of our approach, based on the free theorems concerning type classes [20]. In Section 5, we discuss a datatype-generic implementation of our approach. In Section 6, we revisit the XML example mentioned in this section and show how it is handled by our technique. In Section 7, we review additional issues of bidirectionalization. In Section 8, we discuss related work, and conclude in Section 9.

A prototype implementation of our framework is available from `https://bitbucket.org/kztk/cheap-b18n`, which also contains more examples.

2. THE ESSENCE OF SEMANTIC BIDIRECTIONALIZATION

As a preparation, we firstly introduce the basic idea of semantic bidirectionalization [19]. Consider that we are given a polymorphic function of type $\forall\alpha.\,[\alpha]\to[\alpha]$. Parametricity [18] asserts that the function can only drop or reorganize its input list elements, without inspecting them or constructing new ones. In other words, an element in a view must come directly from an element in the source, and this correspondence enables an update to a view element to be translated into an update to its origin in the source, which forms the basis of a backward transformation.

2.1 Construction of Backward Transformation

We present a simple implementation of semantic bidirectionalization that captures the core idea found in the original paper [19], which will be expanded in Section 3. We assume basic knowledge of Haskell with some GHC extensions and its standard libraries, and may use functions from `Prelude` without explanation. We use rank-2 polymorphism; thus the language option `Rank2Types` is required to run the code in this section.

Consider an arbitrary polymorphic function of type $\forall\alpha.\,[\alpha]\to[\alpha]$, for example *tail*. We know from the type that the function can only reorganize or drop its input list elements, and an element in a view must have a unique corresponding element in the source as its origin. However, it is not possible to conclude the behavior of the function by observing the source-view pair. For example, giving a source `"aab"` and its view `"ab"`, it is not clear which `"a"`-element in the source corresponds to the `"a"`-element in the view.

A way to distinguish potentially equal elements is to identify them with their unique location. We use the following datatype *Loc* to represent location-aware data.

$$\textbf{data}\ \mathit{Loc}\,\alpha = \mathit{Loc}\ \{\mathit{body}::\alpha, \mathit{location}::\mathit{Int}\}$$

For example, the source `"aab"` may have a location-aware version as $[\mathit{Loc}\ \texttt{'a'}\ 1, \mathit{Loc}\ \texttt{'a'}\ 2, \mathit{Loc}\ \texttt{'b'}\ 3]$. If we apply the same polymorphic function to it, we get $[\mathit{Loc}\ \texttt{'a'}\ 2, \mathit{Loc}\ \texttt{'b'}\ 3]$, with a clear correspondence.

Suppose that the view `"ab"` is updated to `"cd"`. Matching it to the location-aware view $[\mathit{Loc}\ \texttt{'a'}\ 2, \mathit{Loc}\ \texttt{'b'}\ 3]$, we can know that location-2 and location-3 are updated to `'c'` and `'d'` respectively. We represent such an update as a list of pairs of location and the new value assigned to the location.

$$\textbf{type}\ \mathit{Update}\,\alpha = [(\mathit{Int},\alpha)]$$

For the above case, we obtain an update $[(2,\texttt{'c'}),(3,\texttt{'d'})]$. Applying the update to the location-aware source and then extracting the *body*s, gives us an updated source `"acd"`.

The application of the update can be easily implemented in Haskell, as the following function *update*.

```
update upd (Loc x i) =
    maybe (Loc x i) (λy.Loc y i) (lookup i upd)
```

And the matching of the location-aware view and the updated view to produce the update is defined as follows.

```
matchViewsSimple :: Eq γ ⇒ [Loc γ] → [γ] → Update γ
matchViewsSimple vx v =
    if length vx == length v then
        minimize vx $ makeUpdSimple $ zip vx v
    else
        error "Shape Mismatch"
```

Here, *makeUpdSimple* is an auxiliary function defined as

```
makeUpdSimple = foldr f []
  where
    f (Loc x i, y) u =
      case lookup i u of
        Nothing → (i, y) : u
        Just y' →
          if y == y' then u
          else            error "Inconsistent Update"
```

A number of checks are performed by *matchViewsSimple* to ensure that the updates to the view do not cause inconsistency: the updates to the view shall not change the list length, and if a source element appears more than once in the view, the multiple occurrences of the same element need to be updated consistently (that's why the *Eq* context is needed). Note that for simplicity we assume that the user-defined equality (`==`) actually implements semantic equality ($=$) for elements. For example, consider a forward function $f\,[x] = [x,x]$ of type $\forall\alpha.[\alpha]\to[\alpha]$. Suppose an initial source `"a"`, and thus a view `"aa"`. Then, updating the view to `"ab"` will be rejected by *matchViewsSimple*, whereas updating to `"bb"` will be accepted. Note that we also use a function *minimize* to remove the redundant parts from an update, which is defined as follows.

$$\mathit{minimize}\ vx\ u = u \setminus\setminus [(i,x) \mid \mathit{Loc}\,x\,i \leftarrow vx]$$

Here, $(\setminus\setminus)$, imported from `Data.List`, computes the difference of two lists. It is worth remarking that the application of *minimize* is optional, as identical updates will not change the behavior the backward transformation. But having this minimality property of updates simplifies the proofs for correctness that will be discussed in Section 4.

With the ground prepared, the higher-order function that takes a forward function and produces a backward counterpart can be realized as the following.

```
bwd :: Eq γ ⇒ (∀α. [α] → [α]) → [γ] → [γ] → [γ]
bwd h = λs v. let sx  = zipWith Loc s [1..]
                  vx  = h sx
                  upd = matchViewsSimple vx v
              in map (body ∘ update upd) sx
```

This version of *bwd* is specific to list-to-list transformations, which is conveniently used to illustrate the basic idea of semantic bidirectionalization. It is shown that the technique generalizes to arbitrary `Traversable` datatypes such as rose trees [10, 19].

2.2 Extensions and Limitations

Things become more complicated if the forward function is not fully polymorphic. For example, consider function $nub :: \forall\alpha.\ Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$ that removes duplicates from a list based on a given equality comparison operator. Now similar to the case of query Q1 we have seen before, the forward transformation is able to observe equality among elements, and the free theorems on fully polymorphic functions are no longer applicable. In the original paper [19], the problem is solved by a more sophisticated location-assigning scheme tailored to each observer function. In the case of *nub*, where equality is used, we need to make sure that the locations fully reflect the equality among elements: locations are equal if and only if the elements are equal, and no update is allowed to break this condition.

As we will see in the next section, this technique of creating specialized location-assigning system to mimic the actual source is not enough to handle functions that is able to construct new elements at runtime, because for a user-defined comparison operator it is no longer possible to generate locations that model the now potentially infinite set of element values. Nevertheless, semantic bidirectionalization remains particularly attractive because it offers the possibility of programming forward transformations in a general-purpose language that is expressive enough for practical applications.

3. OUR BIDIRECTIONALIZATION WITH RUNTIME RECORDING

In this section, we present our improved semantic bidirectionalization framework in Haskell. For illustration, we use a toy example based on rose trees:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Node}\ \alpha\ [\mathit{Tree}\ \alpha]$$

We assume that *Tree* is an instance of *Functor* with an appropriate implementation of *fmap*. To run the code in this section, we need the following language options: `FlexibleInstances`, `MultiParamTypeClasses` and `FunctionalDependencies` in addition to `Rank2Types`.

Example 1 (Links)**.** Query "Links": Collect all subtrees with `"a"` as root label, and arrange them under a new root labeled `"results"`. □

For example, if we apply Links to the source

$$\begin{aligned}\mathit{src}_{\mathit{links}} = \mathit{Node}\ \texttt{"root"}\ [&\mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"text"}\ []],\\ &\mathit{Node}\ \texttt{"p"}\ [\mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"text2"}\ []]]]\end{aligned}$$

we get the following view.

$$\begin{aligned}\mathit{view}_{\mathit{links}} = \mathit{Node}\ \texttt{"results"}\ [&\mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"text"}\ []],\\ &\mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"text2"}\ []]]\end{aligned}$$

Although being very simple, query Links is representative in the sense that its execution involves comparing source labels with constants, and the construction of new labels. A direct implementation of the query is as follows.

$$\begin{aligned}&\mathit{links}_{\mathit{mono}} :: \mathit{Tree\ String} \to \mathit{Tree\ String}\\ &\mathit{links}_{\mathit{mono}}\ t = \mathit{Node}\ \texttt{"results"}\ (\mathit{linkss}_{\mathit{mono}}\ t)\\ &\mathit{linkss}_{\mathit{mono}} :: \mathit{Tree\ String} \to [\mathit{Tree\ String}]\\ &\mathit{linkss}_{\mathit{mono}}\ (\mathit{Node}\ n\ ts) =\\ &\quad \textbf{if}\ n == \texttt{"a"}\ \textbf{then}\ \mathit{Node}\ n\ ts\\ &\quad \textbf{else}\ \ \mathit{concatMap}\ \mathit{linkss}_{\mathit{mono}}\ ts\end{aligned}$$

However, this function is monomorphic, which is not subject to the existing bidirectionalization technique.

3.1 First Try: Making Monomorphic Queries Polymorphic?

The reasons for $\mathit{links}_{\mathit{mono}}$ to be monomorphic are the equality comparison of tree labels with the constant `"a"` and the construction of the constant `"results"`. So a technique to prevent this type instantiation is to avoid the direct use of constants, and instead construct new labels from them. The following type class *PackTrial* can be used for this purpose.

$$\begin{aligned}&\textbf{class}\ \mathit{Eq}\ \gamma \Rightarrow \mathit{PackTrial}\ \gamma\ \alpha \mid \alpha \to \gamma\ \textbf{where}\\ &\quad \mathit{new} :: \gamma \to \alpha\\ &\quad \mathit{eq} :: \alpha \to \alpha \to \mathit{Bool}\end{aligned}$$

Function *new* abstracts a constant to an abstract type, and function *eq* compares abstract labels for equality. With them, we can implement Links as a polymorphic function.

$$\begin{aligned}&\mathit{links}_{\mathit{poly}} :: \forall\alpha.\ \mathit{PackTrial\ String}\ \alpha \Rightarrow \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha\\ &\mathit{links}_{\mathit{poly}}\ t = \mathit{Node}\ (\mathit{new}\ \texttt{"results"})\ (\mathit{linkss}_{\mathit{poly}}\ t)\\ &\mathit{linkss}_{\mathit{poly}} :: \forall\alpha.\ \mathit{PackTrial\ String}\ \alpha \Rightarrow \mathit{Tree}\ \alpha \to [\mathit{Tree}\ \alpha]\\ &\mathit{linkss}_{\mathit{poly}}\ (\mathit{Node}\ n\ ts) =\\ &\quad \textbf{if}\ \mathit{eq}\ n\ (\mathit{new}\ \texttt{"a"})\ \textbf{then}\ \mathit{Node}\ n\ ts\\ &\quad \textbf{else}\ \ \mathit{concatMap}\ \mathit{linkss}_{\mathit{poly}}\ ts\end{aligned}$$

The functional dependency $\alpha \to \gamma$ in the class definition has helped us to avoid a type annotation for the expression $\mathit{eq}\ n\ (\mathit{new}\ \texttt{"a"})$.

Problem solved? Not really. Due to the uses of *new* together with *eq*, which are able to construct new abstract values and compare them with arbitrary labels, the free theorems of the type $\forall\alpha.\mathit{PackTrial\ String}\ \alpha \Rightarrow \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha$ are no longer strong enough to support the original bidirectionalization. Concretely, since the forward transformation is able to construct new labels and use them in observer functions such as equality comparisons, it is no longer possible, without inspecting the actual implementation of the forward function, to predict what changes may affect the observations, and therefore need to be rejected. For example in $\mathit{linkss}_{\mathit{poly}}$ above, due to the comparison $\mathit{eq}\ n\ (\mathit{new}\ \texttt{"a"})$ it is no longer possible to assign a suitable location to n to model its behavior with arbitrary *new*ly created values. As a result, we can no longer guard against any invalid update that alters n to a value no longer equal $(\mathit{new}\ \texttt{"a"})$, leading to the violation of the consistency law. In such a case, no update can be safely accepted, which reduces semantic bidirectionalization to a completely useless state.

3.2 Tracking Observations Using Monad

As we have seen, with the existing bidirectionalization technique, it is necessary to reject all updates when label construction is used, because of the fear that the update may affect the control (or, computation path) of the forward function. On the other hand, this requirement is certainly over-conservative: for a given query such as Links, not all the updates can affect its control. For example, updating `"a"` to any other strings in $\mathit{view}_{\mathit{links}}$ affects the control, but updating `"text"` to other strings does not.

Our idea is to use a monad to keep track of what observations are performed in the execution of a forward transformation. Then, we can employ a more targeted update-checking strategy by rejecting only those that do affect the observations.

Specially, we extend type class *PackTrial* to *PackM* by including a monad parameter. We also separate the con-

struction of new labels into a different class *Pack*.

$$\begin{array}{l}\textbf{class}\ (\mathit{Pack}\ \gamma\ \alpha, \mathit{Monad}\ \mu) \Rightarrow \mathit{PackM}\ \gamma\ \alpha\ \mu\ \textbf{where}\\ \quad \mathit{liftO} :: \mathit{Eq}\ \beta \Rightarrow ([\gamma] \to \beta) \to ([\alpha] \to \mu\ \beta)\\ \textbf{class}\ \mathit{Pack}\ \gamma\ \alpha \mid \alpha \to \gamma\ \textbf{where}\\ \quad \mathit{new} :: \gamma \to \alpha\end{array}$$

In this new design, we no longer deal with specific observer functions such as *eq*; instead function *liftO* lifts any observer function $[\gamma] \to \beta$ on a concrete datatype γ to a monadic one $[\alpha] \to \mu\ \beta$ on an abstract datatype α where β is an instance of *Eq*. The context $\mathit{Eq}\ \beta$ is needed because we will compare the observation results to check the validity of updates. For convenience, we also introduce a specific instance of *liftO* that operates on binary observer functions respectively.

$$\mathit{liftO2}\ p\ x\ y = \mathit{liftO}\ (\lambda[x,y].p\ x\ y)\ [x,y]$$

As a result, forward functions in our setting have the following type.

$$\forall\alpha.\forall\mu.\ \mathit{PackM}\ \gamma\ \alpha\ \mu \Rightarrow \mathit{Tree}\ \alpha \to \mu\ (\mathit{Tree}\ \alpha)$$

The type is polymorphic in α which is suitable for semantic bidirectionalization. And importantly, the type is polymorphic also in μ so that the monad cannot be manipulated directly in the definitions of the forward functions, which guarantees the integrity of the observation results recorded in the monad.

3.3 Forward Execution

In our new setting, the query can be defined as follows.

```
links :: ∀α. PackM String α μ ⇒ Tree α → μ (Tree α)
links t = do as ← linkss t
             return $ Node (new "results") as
linkss :: ∀α. PackM String α μ ⇒ Tree α → μ [Tree α]
linkss (Node n ts) =
    do b ← liftO2 (==) n (new "a")
       if b then return $ Node n ts
       else      concatMapM linkss ts
  where concatMapM h x = do y ← mapM h x
                            return (concat y)
```

As we can see, the above is a straightforward adaptation of the definition of $\mathit{links}_{\mathit{poly}}$.

To execute *links*, we need to instantiate the monad and provide instances of its type class context. For forward execution, which does not require the recording of observations, the identity monad I is used.[1]

$$\textbf{newtype}\ I\ \alpha = I\ \{\mathit{runI} :: \alpha\}$$

We omit the instance declaration of $\mathit{Monad}\ I$ because it is standard. Accordingly, we prepare the following instances of *Pack* and *PackM*.

$$\begin{array}{l}\textbf{instance}\ \mathit{Pack}\ \gamma\ (I\ \gamma)\ \textbf{where}\\ \quad \mathit{new} = I\\ \textbf{instance}\ \mathit{PackM}\ \gamma\ (I\ \gamma)\ I\ \textbf{where}\\ \quad \mathit{liftO}\ p\ x = I\ (p\ \$\ \mathit{map}\ \mathit{runI}\ x)\end{array}$$

In the above, $I\ \gamma$ is used instead of γ to satisfy the functional dependency required by *Pack*, together with another instance $\mathit{Pack}\ \gamma\ (\mathit{Loc}\ \gamma)$ which will be introduced later.

[1]We do not use *Identity* in Haskell for brevity of the proofs.

Then, we can construct a function *fwd* for forward execution as below.

$$\begin{array}{l}\mathit{fwd} :: (\forall\alpha.\forall\mu.\ \mathit{PackM}\ \gamma\ \alpha\ \mu \Rightarrow \mathit{Tree}\ \alpha \to \mu\ (\mathit{Tree}\ \alpha))\\ \qquad \to \mathit{Tree}\ \gamma \to \mathit{Tree}\ \gamma\\ \mathit{fwd}\ h = \lambda s.\textbf{let}\ I\ v = h\ (\mathit{fmap}\ I\ s)\ \textbf{in}\ \mathit{fmap}\ \mathit{runI}\ v\end{array}$$

Example 2 (*linksF*)**.** We can instantiate *links* for forward execution as follows.

$$\begin{array}{l}\mathit{linksF} :: \mathit{Tree\ String} \to \mathit{Tree\ String}\\ \mathit{linksF} = \mathit{fwd}\ \mathit{links}\end{array}$$

We can apply *linksF* directly to sources as in $\mathit{linksF}\ \mathit{src}_{\mathit{links}} = \mathit{view}_{\mathit{links}}$. □

3.4 Backward Execution

Then, we discuss the construction of backward transformations.

3.4.1 An Overview

Similar to before we attach locations to polymorphic labels, with the following type.

$$\textbf{data}\ \mathit{Loc}\ \alpha = \mathit{Loc}\ \{\mathit{body} :: \alpha, \mathit{location} :: \mathit{Maybe\ Int}\}$$

Unlike what we saw in Section 2, the location part of the type is optional (represented by the *Maybe* type): a newly constructed label by *new* does not have a corresponding source label, and is therefore not updatable. For brevity, we write $x@i$ for $\mathit{Loc}\ x\ (\mathit{Just}\ i)$ and $x@\#$ for $\mathit{Loc}\ x\ \mathit{Nothing}$.

Let's assume that a monadic infrastructure is prepared (we will see how it is done in Section 3.4.3). Applying *links* to a location-aware version $\mathit{srcx}_{\mathit{links}}$ of $\mathit{src}_{\mathit{links}}$ defined as

```
srcx_links =
  Node ("root"@1) [
    Node ("a"@2) [Node ("text"@3) []],
    Node ("p"@4) [Node ("a"@5) [Node ("text2"@6) []]]]
```

gives us a location-aware version $\mathit{viewx}_{\mathit{links}}$ of $\mathit{view}_{\mathit{links}}$

```
viewx_links = Node ("results"@#) [
                Node ("a"@2) [Node ("text"@3) []],
                Node ("a"@5) [Node ("text2"@6) []]]
```

together with the following observation history recorded in the monad.

Observation	Argument-1	Argument-2	Result
==	"root"@1	"a"@#	*False*
==	"a"@2	"a"@#	*True*
==	"p"@4	"a"@#	*False*
==	"a"@5	"a"@#	*True*

Entries in the above table represent the observations made during the execution of *links*, which contribute to the control of the computation path. No update is allowed to alter the results. For example, consider an update $[(3, \texttt{"changed"})]$, which changes the label "text" in the view to "changed". Since the label affected does not appear in the history, the update does not change the table, and thus can be accepted. In contrast, an update $[(2, \texttt{"b"})]$ involves location 2 that appears in the history. We then need to check whether the change, from "a" to "b", alters the observation result.

Observation	Argument-1	Argument-2	Result
==	"root"@1	"a"@#	*False*
==	"b"@2	"a"@#	*True*
==	"p"@4	"a"@#	*False*
==	"a"@5	"a"@#	*True*

In this case, the comparison `"b" == "a"` returns *False*, which is different from the result in the history. As a result, the observation table becomes inconsistent, and the update needs to be rejected. This consistency check of the history is key for the application of free theorems, and therefore the correctness of our proposal, which will be discussed formally in Section 4.

In summary, updates are reflected in the following steps.

- Firstly, an observation history is constructed by applying the forward function, instantiated with an appropriate monad, to the location-aware source.
- Then, given an updated view, an update is constructed and checked against the observation history obtained in the previous step.
- Finally, if the update passes the check, it is applied to the source.

In the following, we explain these steps in detail. For generality, we introduce helper functions primarily with specifications, and defer concrete implementations to Section 5.

3.4.2 Locations

As mentioned in Section 3.4.1, locations for labels that appear in a view are optional. In particular, the labels that are newly constructed by function *new* do not have obvious origins in the source, and therefore won't have locations. This is reflected in the instance declaration of *Pack* for backward execution.

$$\begin{array}{c}\textbf{instance}\ \mathit{Pack}\ \gamma\ (\mathit{Loc}\ \gamma)\ \textbf{where}\\ \mathit{new}\ x = \mathit{Loc}\ x\ \mathit{Nothing}\end{array}$$

Just like pointers, only one value can be assigned to a particular location. The property is formally defined as follows.

Definition 1 (Location Consistency)**.** Let γ be a label type. A tree $t :: \mathit{Tree}\ (\mathit{Loc}\ \gamma)$ is *location consistent* if, for any labels $x@i$ and $y@j$ in t such that $i \neq \#$ and $j \neq \#$,

$$i = j \Rightarrow x = y$$

holds. □

We use the following function to assign locations to all source labels.

$$\mathit{assignLocs} :: \mathit{Tree}\ \gamma \to \mathit{Tree}\ (\mathit{Loc}\ \gamma)$$

And it must satisfy the following conditions.

Condition (*assignLocs*)**.** *assignLocs* must satisfy: For all s

- $\mathit{assignLocs}\ s$ is location consistent, and
- $\mathit{fmap}\ \mathit{body}\ (\mathit{assignLocs}\ s) = s$. □

This specification is rather loose and leave room for different implementations of *assignLocs*. In Example 2, we have chosen the following assignment with running integers.

`"root"`@1, `"a"`@2, `"text"`@3, `"p"`@4, `"a"`@5, `"text2"`@6

Another implementation is to assign the same locations to "identical" labels as in the following.

`"root"`@1, `"a"`@2, `"text"`@3, `"p"`@4, `"a"`@2, `"text2"`@5

This assignment still guarantees location consistency as `"a"` = `"a"`. The difference is that now the two `"a"`s are considered duplicates, instead of separate labels with the same value.

In this paper, we mainly discuss the former strategy, and a datatype-generic implementation of it can be found in Section 5. Nevertheless, we will discuss the implication of the different choices in Section 7.

3.4.3 Observation History

The observation history is represented as a list of the following datatype.

$$\textbf{data}\ \mathit{Result}\ \alpha = \forall\beta.\mathit{Eq}\ \beta \Rightarrow \mathit{Result}\ ([\alpha] \to \beta)\ [\alpha]\ \beta$$

Roughly speaking, a value $\mathit{Result}\ p\ [x_1, \ldots, x_n]\ r$ corresponds to a line in the observation table shown in Section 3.3, as below.

Observation	Argument-1	...	Argument-n	Result
p	x_1	...	x_n	r

The actual type of the observation outcome is existentially quantified, so that results of different observers can be more easily kept together.

The consistency of a history entry can be easily checked.

$$\begin{array}{l}\mathit{check} :: \mathit{Result}\ \alpha \to \mathit{Bool}\\ \mathit{check}\ (\mathit{Result}\ p\ \mathit{xs}\ r) = p\ \mathit{xs}\ \texttt{==}\ r\end{array}$$

And we can check whether a history remains consistent after an update.

$$\begin{array}{l}\mathit{checkHist} :: (\alpha \to \alpha) \to [\mathit{Result}\ \alpha] \to \mathit{Bool}\\ \mathit{checkHist}\ u\ h = \mathit{all}\ (\mathit{check} \circ u')\ h\\ \quad \textbf{where}\ u'\ (\mathit{Result}\ p\ \mathit{xs}\ r) = \mathit{Result}\ p\ (\mathit{map}\ u\ \mathit{xs})\ r\end{array}$$

A "Writer" monad W is responsible for gathering histories[2]

$$\begin{array}{l}\textbf{newtype}\ W\ \eta\ \beta = W\ (\beta, [\mathit{Result}\ \eta])\\ \textbf{instance}\ \mathit{Monad}\ (W\ \alpha)\ \textbf{where}\\ \quad \mathit{return}\ x = W\ (x, [])\\ \quad W\ (x, h_1) \gg\!= f = \textbf{let}\ W\ (y, h_2) = f\ x\ \textbf{in}\ W\ (y, h_1\ \texttt{++}\ h_2)\end{array}$$

which allows us to define the following instance of *PackM* for backward executions.

$$\begin{array}{c}\textbf{instance}\ \mathit{PackM}\ \gamma\ (\mathit{Loc}\ \gamma)\ (W\ (\mathit{Loc}\ \gamma))\ \textbf{where}\\ \mathit{liftO}\ p\ x = W\ (p'\ x, [\mathit{Result}\ p'\ x\ (p'\ x)])\\ \quad \textbf{where}\ p' = p \circ \mathit{map}\ \ \mathit{body}\end{array}$$

3.4.4 Updates

The application of updates remains straightforward. The only change from Section 2 is that we now deal with optional locations.

$$\begin{array}{l}\mathit{update} :: \mathit{Update}\ \gamma \to (\mathit{Loc}\ \gamma) \to (\mathit{Loc}\ \gamma)\\ \mathit{update}\ \mathit{upd}\ (\mathit{Loc}\ x\ \mathit{Nothing}) = \mathit{Loc}\ x\ \mathit{Nothing}\\ \mathit{update}\ \mathit{upd}\ (\mathit{Loc}\ x\ (\mathit{Just}\ i)) =\\ \quad \mathit{maybe}\ (\mathit{Loc}\ x\ (\mathit{Just}\ i))\ (\lambda y.\mathit{Loc}\ y\ (\mathit{Just}\ i))\ (\mathit{lookup}\ i\ \mathit{upd})\end{array}$$

The above definition satisfies the following conditions, which will be used in the proofs in Section 4.

Condition (*update*)**.** *update* satisfies the following conditions. For any x,

- $\mathit{update}\ []\ x = x$, and
- $\mathit{update}\ \mathit{upd}\ (\mathit{new}\ x) = \mathit{new}\ x$. □

[2]We do not use *Writer* in Haskell instead of W for brevity of the proofs.

Updates are extracted by comparing a location-aware view and an updated view with a function of the following type.

$$matchViews :: Eq\,\gamma \Rightarrow Tree\,(Loc\,\gamma) \to Tree\,\gamma \to Update\,\gamma$$

The definition of *matchViews* is similar to *matchViewsSimple* in Section 2, except that *matchViews* needs to recognize labels without locations and reject any changes to them. We postpone the definition of *matchViews* to Section 5 where we present a datatype-generic version of it. We require *matchViews* to satisfy the following conditions.

Condition (*matchViews*)**.** *matchViews* must satisfy:

- **Correctness.** for any v' and location-consistent vx, if $matchViews\ vx\ v'$ succeeds and results in upd, then $fmap\ (body \circ update\ upd)\ vx = v'$ holds.
- **Minimality.** for any v and location-consistent vx such that $fmap\ body\ vx = v$, $matchViews\ vx\ v = []$ holds. □

3.4.5 Putting Everything Together

With all the ground prepared, we are now ready to set up the backward execution.

$$\begin{aligned}
&bwd :: Eq\,\gamma \Rightarrow (\forall\alpha.\forall\mu.PackM\ \gamma\ \alpha\ \mu \Rightarrow Tree\ \alpha \to \mu\ (Tree\ \alpha)) \\
&\qquad \to Tree\,\gamma \to Tree\,\gamma \to Tree\,\gamma \\
&bwd\ h = \lambda s\ v.\ \mathbf{let}\ sx = assignLocs\ s \\
&\qquad\qquad W\ (vx, hist) = h\ sx \\
&\qquad\qquad upd = matchViews\ vx\ v \\
&\qquad\quad \mathbf{in\ if}\ checkHist\ (update\ upd)\ hist\ \mathbf{then} \\
&\qquad\qquad\quad fmap\ (body \circ update\ upd)\ sx \\
&\qquad\quad \mathbf{else} \\
&\qquad\qquad\quad error\ \texttt{"Inconsistent History"}
\end{aligned}$$

Example 3 (*links*)**.** We can instantiate *links* for backward execution as follows.

$$\begin{aligned}
&linksB :: Tree\ String \to Tree\ String \to Tree\ String \\
&linksB = bwd\ links
\end{aligned}$$

Suppose that $view_{links}$ is updated to the following tree.

$$\begin{aligned}
view' = Node\ \texttt{"results"}\ [&Node\ \texttt{"a"}\ [Node\ \texttt{"changed"}\ []], \\
&Node\ \texttt{"a"}\ [Node\ \texttt{"text2"}\ []]]
\end{aligned}$$

Then, $linksB\ src_{links}\ view'$ results in the following updated source.

$$\begin{aligned}
Node\ \texttt{"root"}\ [&Node\ \texttt{"a"}\ [Node\ \texttt{"changed"}\ []], \\
&Node\ \texttt{"p"}\ [Node\ \texttt{"a"}\ [Node\ \texttt{"text2"}\ []]]]
\end{aligned}$$

On the other hand, an update to the view $view''$

$$\begin{aligned}
view'' = Node\ \texttt{"results"}\ [&Node\ \texttt{"b"}\ [Node\ \texttt{"text"}\ []], \\
&Node\ \texttt{"a"}\ [Node\ \texttt{"text2"}\ []]]
\end{aligned}$$

is rejected for the reason discussed in Section 3.3. □

4. CORRECTNESS

In this section, we prove the correctness of our approach. That is, we prove that a forward transformation and the derived backward transformation satisfy consistency and acceptability mentioned in Section 1.

We rewrite the laws in our setting. Let h be a function of type $\forall\alpha.\forall\mu.\ PackM\ \gamma\ \alpha\ \mu \Rightarrow Tree\ \alpha \to \mu\ (Tree\ \alpha)$. We prove that the following laws hold for any sources s and s', and view v.

$$\begin{aligned}
&\textbf{Acceptability} \quad bwd\ h\ s\ (fwd\ h\ s) = s \\
&\textbf{Consistency} \quad fwd\ h\ s' = v \quad \text{if} \quad bwd\ h\ s\ v = s'
\end{aligned}$$

Throughout the section, we fix the function h.

Following the original work [19], we make use of free theorems [2,20,22,23] in the proofs. We assume that a polymorphic function h that we bidirectionalize is total, and sources and views do not contain any undefined values. We also implicitly use the fact that our backward transformation can be made total through the explicit handling of exceptions (*e.g.*, by *Maybe*). Thus, we can interpret types as sets and functions as set-theoretic functions. This totality assumption is reasonable in the context of bidirectional transformation.

4.1 Free Theorem

Roughly speaking, free theorems are theorems obtained for free as corollaries of relational parametricity [18], which states that, for a term f of type τ, (f, f) belongs to a certain relational interpretation of τ. A simple example of a free theorem is that f of type $\forall\alpha.[a] \to [a]$ satisfies $map\ g \circ f = f \circ map\ g$ for any function g of type $\sigma \to \tau$.

We start by introducing some notations. We write $\mathcal{R} :: \sigma_1 \leftrightarrow \sigma_2$ if $\mathcal{R}$ is a relation on $\sigma_1 \times \sigma_2$. For relations $\mathcal{R} :: \sigma_1 \leftrightarrow \sigma_2$ and $\mathcal{R}' :: \tau_1 \leftrightarrow \tau_2$, we write $\mathcal{R} \to \mathcal{R}' :: (\sigma_1 \to \tau_1) \leftrightarrow (\sigma_2 \to \tau_2)$ for the relation $\{(f, g) \mid \forall(x, y) \in \mathcal{R}.(f\,x, g\,y) \in \mathcal{R}'\}$. For a polymorphic term f of type $\forall\alpha.\tau$ and a type σ, we write f_σ for the instantiation of f to σ that has type $\tau[\sigma/\alpha]$. For simplicity, we sometimes omit the subscript and simply write f for f_σ if σ is clear from the context or irrelevant.

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

$$\begin{aligned}
[\![\alpha]\!]_\rho &= \rho(\alpha) \\
[\![B]\!]_\rho &= \{(e, e) \mid e :: B\} \quad \text{if } B \text{ is a base type} \\
[\![\tau_1 \to \tau_2]\!]_\rho &= [\![\tau_1]\!]_\rho \to [\![\tau_2]\!]_\rho \\
[\![\forall\alpha.\tau]\!]_\rho &= \left\{(u, v) \,\middle|\, \forall\mathcal{R} :: \sigma_1 \leftrightarrow \sigma_2.\ (u_{\sigma_1}, v_{\sigma_2}) \in [\![\tau]\!]_{\rho[\alpha \mapsto \mathcal{R}]}\right\}
\end{aligned}$$

Here, $\rho[\alpha \mapsto \mathcal{R}]$ is an extension of ρ with $\alpha \mapsto \mathcal{R}$. If $\rho = \emptyset$, we sometimes write $[\![\tau]\!]$ instead of $[\![\tau]\!]_\emptyset$. We abuse the notation to write $[\![\forall\alpha.\tau]\!]$ as $\forall\mathcal{R}.\mathcal{F}$ where $\mathcal{F}$ is the interpretation $[\![\tau]\!]_{\{\alpha \mapsto \mathcal{R}\}}$. For example, we write $\forall\mathcal{R}.\forall\mathcal{S}.\ \mathcal{R} \to \mathcal{S}$ for $[\![\forall\alpha.\forall\beta.\alpha \to \beta]\!]$.

The relational interpretation can be extended to the list type $[\,\cdot\,]$ and the rose-tree type *Tree*, as follows.

$$[\![[\tau]]\!]_\rho = [[\![\tau]\!]_\rho] \qquad [\![Tree\ \tau]\!]_\rho = Tree\ [\![\tau]\!]_\rho$$

Here, we write $[\mathcal{S}]$ for the smallest relation satisfying

$$\begin{aligned}
&([], []) \in [\mathcal{S}], \qquad \text{and} \\
&(a_1 : x_2, a_1 : x_2) \in [\mathcal{S}] \Leftrightarrow (a_1, a_2) \in \mathcal{S} \wedge (x_1, x_2) \in [\mathcal{S}],
\end{aligned}$$

and write $Tree\ \mathcal{R}$ for the smallest relation satisfying

$$\begin{aligned}
&(Node\ x_1\ ts_1, Node\ x_2\ ts_2) \in Tree\ \mathcal{R} \\
&\qquad \Leftrightarrow (x_1, x_2) \in \mathcal{R}, (ts_1, ts_2) \in [Tree\ \mathcal{R}].
\end{aligned}$$

Intuitively, $[\mathcal{R}]$ relates two lists with the same length of which each pair of the elements in a same position are related by $\mathcal{R}$, and similarly $Tree\ \mathcal{R}$ relates two trees with the same shape of which each pair of labels in the same position are related by $\mathcal{R}$.

Then, parametricity states that, for a term f of a closed type τ, (f, f) is in $[\![\tau]\!]$.

Free theorems are theorems obtained by instantiating parametricity. For example, for $f :: \forall\alpha.[\alpha] \to [\alpha]$, we must have $(f, f) \in \forall\mathcal{R}.[\mathcal{R}] \to [\mathcal{R}]$. Thus, for any $\mathcal{R} : \sigma_1 \leftrightarrow \sigma_2$, $(f_{\sigma_1}, f_{\sigma_2}) \in [\mathcal{R}] \to [\mathcal{R}]$ holds. That is, if we take

$\mathcal{R} = \{(x, g\,x) \mid x :: \sigma_1\}$ for any $g :: \sigma_1 \to \sigma_2$, we obtain $map\ g \circ f = f \circ map\ g$.

Voigtländer [20] extends parametricity to a type system with type constructors. A key notion in his result is *relational action*.

Definition 2 (Relational Action [20])**.** For type constructor 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 closed type τ_1 and τ_2 to $\mathcal{F}\ \mathcal{R} : \kappa_1\ \tau_1 \leftrightarrow \kappa_2\ \tau_2$. □

Accordingly, the relational interpretation is extended as:

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

Parametricity holds also on the relational interpretation [2, 22]. Here, κ, κ_1 and κ_2 are type constructors of kind $* \to *$, and thus the quantified $\mathcal{F}$ is a relational action.

Voigtländer [20] handles a function with type-class constraints as a higher-order function that takes the methods of the type classes as inputs. For example, a function $f :: Monad\ \mu \Rightarrow \tau$ can be seen as a function $f' :: (\forall\alpha.\ \alpha \to \mu\,\alpha) \to (\forall\alpha.\forall\beta.\ \mu\,\alpha \to (\alpha \to \mu\,\beta) \to \mu\,\beta) \to \tau$ with $f' = \lambda return.\lambda(\gg\!=).f$, and an instance f_κ of f can be seen as $f'\ return_\kappa\ (\gg\!=)_\kappa$. As he packed the conditions posed by the above interpretation of *Monad* by *Monad*-action, we introduce a similar notion of *PackM*-action for *PackM*.

Definition 3 (*PackM*-action)**.** For relations $\mathcal{L} :: \sigma_1 \leftrightarrow \sigma_2$ and $\mathcal{U} :: \tau_1 \leftrightarrow \tau_2$ and a relational action $\mathcal{F} :: \kappa_1 \leftrightarrow \kappa_2$, a triple $(\mathcal{L},\mathcal{U},\mathcal{F})$ is called a *PackM-action* if all the following conditions hold.

- $(\sigma_i, \tau_i, \kappa_i)$ is an instance of *PackM* for $i = 1, 2$,
- $(return_{\kappa_1}, return_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\,\mathcal{R}$,
- $((\gg\!=)_{\kappa_1}, (\gg\!=)_{\kappa_2}) \in \forall\mathcal{R}.\forall\mathcal{S}.\mathcal{F}\,\mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\,\mathcal{S}) \to \mathcal{F}\,\mathcal{S})$,
- $(new_{\sigma_1,\tau_1}, new_{\sigma_2,\tau_2}) \in \mathcal{L} \to \mathcal{U}$, and
- $(liftO_{\sigma_1,\tau_1,\mu_1,\beta_1}, liftO_{\sigma_2,\tau_2,\mu_2,\beta_2}) \in ([\mathcal{L}] \to \mathcal{S}) \to [\mathcal{U}] \to \mathcal{F}\ \mathcal{S}$ for all $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ satisfying $((==)_{\beta_1}, (==)_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to Bool$. □

Intuitively, a *PackM*-action is a property that is "preserved" under *PackM*-methods.

Now, we are ready to state a free theorem for a function h of the type $\forall\alpha.\forall\mu.\ PackM\ \gamma\,\alpha\,\mu \Rightarrow Tree\ \alpha \to \mu\,(Tree\ \alpha)$.

Theorem 1 (A Free Theorem)**.** *Suppose h be a function of type $\forall\alpha.\forall\mu.\ PackM\ \gamma\,\alpha\,\mu \Rightarrow Tree\ \alpha \to \mu\,(Tree\ \alpha)$. Let γ be a type and $\mathcal{L}$ be a relation $\{(e, e) \mid e :: \gamma\}$. Let (τ_1, κ_1) and (τ_2, κ_2) be pairs of types and type constructors such that $(\gamma, \tau_1, \kappa_1)$ and $(\gamma, \tau_2, \kappa_2)$ are instances of PackM. Then, for every PackM-action $(\mathcal{L},\mathcal{U} :: \tau_1 \leftrightarrow \tau_2, \mathcal{F} :: \kappa_1 \leftrightarrow \kappa_2)$, we have $(h_{\tau_1,\kappa_1}, h_{\tau_2,\kappa_2}) \in Tree\ \mathcal{U} \to \mathcal{F}\ (Tree\ \mathcal{U})$.* □

4.2 Preservation of Local Consistency

First of all, we prove that a tree vx constructed in *bwd* is location consistent. This is used to apply the properties on *matchViews*.

Lemma 1 (Location-Consistency of vx)**.** Suppose $W\ (vx, \_) = h\ (assignLocs\ s)$ for a tree s. Then, vx is location consistent.

Proof Sketch. Let s be a source and E be the set of labels in *assignLocs* s. Then, it follows that vx is location-consistent if all the labels $e = x@i$ in vx with $i \neq \#$ are also in E.

Let $\mathcal{U}$ and $\mathcal{F}$ be a relation and a relational action.

$$\begin{aligned}
\mathcal{U} &= \{(x@i, x@i) \mid x@i \in E, i \neq \#\} \\
\mathcal{F}\,\mathcal{R} &= \{(W\ (x, _), W\ (y, _)) \mid (x, y) \in \mathcal{R}\}
\end{aligned}$$

Then, we can prove that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action, and $(h, h) \in Tree\,\mathcal{U} \to \mathcal{F}\,(Tree\,\mathcal{U})$ from Theorem 1 where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$. Since $(assignLocs\ s, assignLocs\ s) \in Tree\,\mathcal{U}$, we have $(vx, vx) \in Tree\,\mathcal{U}$. Thus, vx is location consistent. □

4.3 Proof of Acceptability

The overall structure of our (calculational-style) proof of acceptability is as follows.

$$\begin{aligned}
& bwd\ h\ s\ (fwd\ h\ s) \\
=\ & \{\ \text{Unfolding } bwd\ \} \\
& \textbf{if } checkHist\ (update\ upd)\ hist \textbf{ then} \\
& \qquad fmap\ (body \circ update\ upd)\ sx \textbf{ else } \ldots \\
=\ & \{\ (*) \text{ — see below}\ \} \\
& \textbf{if } True \textbf{ then } fmap\ (body \circ id)\ sx \textbf{ else } \ldots \\
=\ & \{\ \text{Reduction}\ \} \\
& fmap\ body\ sx \\
=\ & \{\ \text{Property of } assignLocs\ \} \\
& s
\end{aligned}$$

At (*), we use two properties: one is $upd = []$, and the other is $checkHist\ (update\ [])\ hist = True$. To show them, it suffices to use the following lemma together with the properties on *matchViews* (with Lemma 1) and *update*.

Lemma 2. Let sx be $assignLocs\ s$. Suppose $W\ (vx, hist) = h\ sx$ and $I\ v = h\ (fmap\ I\ s)$. Then, we have $fmap\ body\ vx = fmap\ runI\ v$ and $checkHist\ id\ hist = True$.

Proof Sketch. Let $\mathcal{U} :: I\,\gamma \leftrightarrow (Loc\,\gamma)$ and $\mathcal{F} :: I \leftrightarrow W\ (Loc\,\gamma)$ be a relation and a relational action defined by:

$$\begin{aligned}
\mathcal{U} &= \{(x, y) \mid runI\ x = body\ y\} \\
\mathcal{F}\,\mathcal{R} &= \{(I\,x, W\ (y, w)) \mid (x, y) \in \mathcal{R} \wedge checkHist\ id\ w\}
\end{aligned}$$

We can show that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$. Then, we have $(h, h) \in Tree\,\mathcal{U} \to \mathcal{F}\,(Tree\,\mathcal{U})$ from Theorem 1; that is, for any x and y with $fmap\ runI\ x = fmap\ body\ y$, we obtain $fmap\ runI\ v = fmap\ body\ vx$ and $checkHist\ id\ hist = True$ where $I\ v = h\ x$ and $W\ (vx, hist) = h\ y$. Taking $x = fmap\ I\ s$ and $y = assignLocs\ s$, we obtain the lemma. □

4.4 Proof of Consistency

The proof is a bit more complicated but has a similar structure. The overall structure of our proof is as follows.

$$\begin{aligned}
& fwd\ h\ (bwd\ h\ s\ v) \quad (\text{assuming } bwd\ h\ s\ v \text{ succeeds}) \\
=\ & \{\ \text{Unfolding } bwd, \text{ and } bwd \text{ succeeded}\ \} \\
& fwd\ h\ (fmap\ (body \circ update\ upd)\ sx) \\
=\ & \{\ (*) \text{ — see below}\ \} \\
& fmap\ (body \circ (update\ upd))\ vx \\
=\ & \{\ \text{Property of } matchViews \text{ and Lemma 1}\} \\
& v
\end{aligned}$$

At (*), we used the following lemma.

Lemma 3. Let sx be $assignLocs\ s$, and s' be $fmap\ (body \circ update\ upd)\ sx$. Let v', vx and $hist$ be those obtained from $I\ v' = h\ (fmap\ I\ s')$ and $W\ (vx, hist) = h\ sx$. Suppose we have $checkHist\ (update\ upd) = True$. Then we have

$$fmap\ runI\ v' = fmap\ (body \circ update\ upd)\ vx$$

Proof Sketch. Let $\mathcal{U} :: I\ \gamma \leftrightarrow Loc\ \gamma$ and $\mathcal{F} :: I \leftrightarrow W\ (Loc\ \gamma)$ be a relation and a relational action defined by:

$$\begin{aligned}&\mathcal{U} \quad = \{(x, y) \mid runI\ x = body\ (update\ upd\ y)\}\\ &\mathcal{F}\ \mathcal{R} =\\ &\quad \{(I\ x, W\ (y, w)) \mid checkHist\ (update\ upd)\ w \Rightarrow (x, y) \in \mathcal{R}\}\end{aligned}$$

We can show that $(\mathcal{L}, \mathcal{U}, \mathcal{F})$ is a $PackM$-action where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$. Then, we can prove the lemma straightforwardly from Theorem 1. □

5. GOING GENERIC

So far, we have demonstrated our idea for a specific type, namely *Tree*. In this section, we extend the solution to be datatype-generic.

Actually, this generalization has already been done in the previous work [10, 19], and we borrow the ideas and adapt them for our new setup. More concretely, we use the datatype-generic function *traverse* from `Data.Traversable` to define *assignLocs* and *matchViews*, and change the type declarations of *fwd* and *bwd* accordingly.

$$\begin{aligned}&traverse :: (Traversable\ \kappa, Applicative\ \theta) \Rightarrow\\ &\qquad (\alpha \to \theta\ \beta) \to \kappa\ \alpha \to \theta\ (\kappa\ \beta)\end{aligned}$$

We also use the following helper functions.

```
toList :: (Traversable κ) ⇒ κ α → [α]
toList = getConst ∘ traverse (λx.Const [x])

fill :: Traversable κ ⇒ κ β → [α] → κ α
fill t l = evalState (traverse f t) l
  where f x = do (a : x) ← Control.Monad.State.get
                 Control.Monad.State.put x
                 return a
```

Here, *Const* and *getConst* are from `Control.Applicative`. We qualified the state monad operations *get* and *put* with *Control.Monad.State* to distinguish them from the *get* and *put* as bidirectional transformations. Intuitively, *toList* extracts all the elements of a container as a list, and *fill* replaces the elements of a container by those of a given list.

We redefine *assignLocs* as a function of type

$$assignLocs :: Traversable\ \kappa \Rightarrow \kappa\ \gamma \to \kappa\ (Loc\ \gamma)$$

with definition

```
assignLocs t = fill t (assignLocsList $ toList t)
  where assignLocsList l =
          zipWith (λx i. Loc x (Just i)) l [1..]
```

For "lawful" instances of *Traversable* [3], including those that are systematically derived [15], the conditions we posed on *assignLocs* in Section 3.4 hold.

We also redefine *matchViews*.

$$\begin{aligned}&matchViews :: (Traversable\ \kappa, Eq\ (\kappa\ ())) \Rightarrow\\ &\qquad \kappa\ (Loc\ \gamma) \to \kappa\ \gamma \to Update\ \gamma\end{aligned}$$

What *matchViews* does is to perform element-wise comparison after shape-equality check.

```
matchViews vx v =
  if fmap ignore vx == fmap ignore v then
    let lx = toList vx
        l  = toList v
    in minimize lx $ makeUpd $ zip lx l
  else
    error "Shape Mismatch"
```

Here, *ignore* is a function defined by $ignore\ x = ()$, which ignores its input but leaves a place-holder. Function *makeUpd*, which is defined below, constructs an update from two views, making sure that elements without locations are not changed, and location consistency is not violated.

```
makeUpd = foldr f []
  where
    f (Loc x Nothing, y) u =
      if x == y then u
      else           error "Update of Constant"
    f (Loc x (Just i), y) u =
      case lookup i u of
        Nothing → (i, y) : u
        Just y' →
          if y == y' then u
          else          "Inconsistent Update"
```

Again, the conditions we pose on *matchViews* hold by lawful *Traversable* instances [3].

Accordingly, the types of *fwd* and *bwd* are updated.

$$\begin{aligned}&fwd\ :: (Traversable\ \kappa_1, Traversable\ \kappa_2, Eq\ (\kappa_2\ ()), Eq\ \gamma) \Rightarrow\\ &\quad (\forall \alpha\ \mu. PackM\ \gamma\ \alpha\ \mu \Rightarrow \kappa_1\ \alpha \to \mu\ (\kappa_2\ \alpha))\\ &\quad \to \kappa_1\ \gamma \to \kappa_2\ \gamma\\ &bwd\ :: (Traversable\ \kappa_1, Traversable\ \kappa_2, Eq\ (\kappa_2\ ()), Eq\ \gamma) \Rightarrow\\ &\quad (\forall \alpha\ \mu. PackM\ \gamma\ \alpha\ \mu \Rightarrow \kappa_1\ \alpha \to \mu\ (\kappa_2\ \alpha))\\ &\quad \to \kappa_1\ \gamma \to \kappa_2\ \gamma \to \kappa_1\ \gamma\end{aligned}$$

No change is required in the definitions of *fwd* and *bwd*.

Note that there exist GHC extensions `-XDeriveFunctors` and `-XDeriveTraversables` that allow instances of *Functor* and *Traversable* to be automatically derived, which comes handy for bidirectionalizing transformations on user-defined datatypes.

6. XML QUERY EXAMPLE

In this section, we revisit our motivating example of XML querying, and show how we can bidirectionalize the query Q1 shown in Figure 2. Recall that, if we apply Q1 to the XML source in Figure 1 we get the XML view in Figure 3.

6.1 A Datatype for XML

Firstly, we define a datatype to describe XML documents, using the rose-tree datatype defined in Section 3 with the following label type.

$$\textbf{data}\ L = \mathsf{A}\ String \mid \mathsf{E}\ String \mid \mathsf{T}\ String\ \textbf{deriving}\ Eq$$

Here, A, E and T stand for "attribute", "element" (in terms of XML) and "text" (attribute values and character data). We omit other features of XML that are not expressed by this datatype, such as namespaces.

For example, an XML fragment

```
<book year="1994"><title>Text</title></book>
```

is represented as

```
Node (E "book") [Node (A "year") [Node (T "1994")],
                 Node (E "title") [Node (T "Text")]]
```

The following function *label* is handy when we write programs that manipulate the rose trees.

```
label (Node l _) = l
```

6.2 Programming the Forward Transformation

Then, we implement Q1 as a function of type $\forall\alpha.\forall\mu.$ *PackM L* α μ $\Rightarrow$ *Tree* α $\rightarrow$ μ (*Tree* α) that is suitable for bidirectionalization. A standard way to write XML transformations in a functional language is to use "filters" [24]. In [24], the filters (if we simplify and customize them to our rose trees) are of type *Tree L* $\rightarrow$ [*Tree L*], which will be made polymorphic in our setting as

```
PackM L α μ ⇒ Tree α → ListT μ (Tree α).
```

where monad transformer *ListT* in `Control.Monad.List` is defined by:

```
newtype ListT μ α = ListT {runListT :: μ [α]}
```

which has an implementation for the method *lift* :: $\mu\,\alpha \rightarrow$ *ListT* $\mu\,\alpha$ in `Control.Monad.Trans`. In addition, we will make use of the fact that *ListT* μ is an instance of *MonadPlus* in `Control.Monad`. Specifically, we use function *mzero* :: *MonadPlus* $\kappa \Rightarrow \kappa\,\alpha$ to represent computation failure.

A simple example of a filter is *keep* that keeps its input.

```
keep :: PackM L α μ ⇒ Tree α → ListT μ (Tree α)
keep = return
```

Another simple example is *children* that returns the children of a node, defined as follows.

```
children :: PackM L α μ ⇒ Tree α → ListT μ (Tree α)
children (Node _ ts) = ListT (return ts)
```

Also, a useful example is *ofLabel l t* that returns *t* as is if the root of *t* has label *l*, and fails otherwise.

```
ofLabel :: PackM L α μ ⇒ α → Tree α → ListT μ (Tree α)
ofLabel l t = do guardM $ lift $ liftO2 (==) (label t) l
                 return t
```

Here, *guardM* is a function that is similar to *guard* in `Control.Monad` except that *guardM* takes a monadic argument. It fails if its argument is *True* and does nothing otherwise.

```
guardM :: MonadPlus κ ⇒ κ Bool → κ ()
guardM x = x >>= (λb.if b then return () else mzero)
```

Filters are composable [24]. For example, with following operator `/>`

```
(/>) :: PackM L α μ ⇒ (Tree α → ListT μ (Tree α))
                    → (Tree α → ListT μ (Tree α))
                    → (Tree α → ListT μ (Tree α))
f /> g = f >=> children >=> g
```

where (>=>) is Kleisli-composition operator in `Control.Monad` defined by $(f \gg\!=\!\!> g)\,x = f\,x \gg\!= g$, for example, we can make a filter *keep* `/>` *ofLabel* (*new* `$ E "book"`) that extracts `book` elements from the children of its input, and a filter *keep* `/>` *keep* `/>` *keep* that extracts the grandchildren of its input.

```
q1 :: PackM L α μ ⇒ Tree α → μ (Tree α)
q1 t = pick $ do bs ← gather $ (keep /> (tag "book" >=> h)) x
                 return $ Node (el "bib") bs
  where
    h b =
      do y ← (keep /> attr "year" /> keep) b
         t ← (keep /> tag "title") b
         p ← (keep /> tag "publisher" /> keep) b
         guardM $ lift $ liftO2 gtInt (label y) (new $ T "1991")
         guardM $ lift $
              liftO2 (==) (label p) (new $ T "Addison-Wesley")
         return $ Node (new $ E "book")
                       [Node (new $ A "year") [y], t]
    gtInt l1 l2 = (read l1 :: Int) > (read l2 :: Int)
tag s = ofLabel (new $ E s)
attr s = ofLabel (new $ A s)
gather :: Monad μ ⇒ ListT μ α → ListT μ [α]
gather (ListT m) = ListT $ do {x ← m; return [x]}
pick :: Monad μ ⇒ ListT μ α → μ α
pick (ListT x) = do {a ← x; return $ head a}
```

Figure 4: Query Q1 in Our Framework

Now we are ready to implement Q1 in Figure 4. The code is mostly declarative. An auxiliary function *gather* gathers results: for example *children y* produces one child at a time, and *gather* (*children y*) collects the children in a list.

6.3 Permitted Updates

Given that *q1* has the right type, we can easily bidirectionalize it with *fwd* and *bwd*. It is not difficult to see that *fwd q*1 implements Q1, although there is a subtle difference that Q1 reads an input XML documents from a certain URL while *q1* takes the input as a parameter.

Let us discuss what kind of updates will be permitted by *bwd q1*. Consider the view in Figure 3. There are the following kinds of in-place updates:

- Changing `bib`-tags to other tags.
- Changing `book`-tags to other tags.
- Changing `year` to other attributes.
- Changing the values of `year`-attributes.
- Changing `title`-tags to other tags.
- Changing title-texts under `title`s.

The first three updates should be rejected because these elements and attributes are those introduced by the query *q1* instead of coming from the original source. As expected, *bwd q1* rejects the three updates; more precisely, an error `"Update of Constant"` is raised by *matchViews*.

The fourth update, which is the most interesting case among the six, is conditionally accepted by *bwd q1*; more precisely, we can change the value to any (string representation of) numbers as long as the number is greater than 1991. This behavior is quite natural because if we change the year to one that is no greater than 1991, say 1990, then the book will disappear from the view, which violates the consistency law.

The fifth-update is rejected for a similar reason. Note that the query extracted `title`s by (*keep* `/>` *tag* `"title"`) *b*. Thus, if we allow changing `title`-tags to other tags, the consistency law will be violated.

The last update is unconditionally accepted by *bwd q1* because *q1* does not inspect titles.

7. INTERPRETATION OF DUPLICATION

In this section, we discuss the difference in the interpretation of duplication between the original semantic bidirectionalization [19] and ours.

It is rather natural to expect that in any reasonable system, duplicates shall be updated at the same time to the same values. In theory, there is little controversy about this statement; yet in practice, it is less obvious whether two values are actual duplicates, or merely being incidentally equal.

As mentioned in Section 3.4.2, in this paper we take a conservative approach by considering all source elements as independent data regardless of their values. This decision makes a lot of sense. For example, consider our example Q1 in Figure 2. Suppose that there are books published in the same year in the source, we don't want to have all of them changed just because one is changed.

On the other hand, it is also obvious that our choice is not the only correct one. In the original work on semantic bidirectionalization [19], when the system is extended to non-fully polymorphic forward functions such as $\forall\alpha.\ Eq\,\alpha \Rightarrow [\alpha] \to [\alpha]$, they use a strategy of considering elements as duplicates, as along as they are equal. The idea to handle the above function is to provide a specialized version of *assignLocs* that assigns locations that reflect the uses of *Eq* (*i.e.*, equal elements get the same location so that for every two elements $x@i$ and $y@j$ ($i \neq \#$ and $j \neq \#$) the condition that $x = y$ if and only if $i = j$ holds). Any updates that violate this condition are rejected. As an example, consider function *nub* in `Data.List` that removes duplicated elements from a list. The backward function named as *nubB* has the following behavior.

```
> nub "abba"
ab
> nubB "abba" "cb"
cbbc
```

In the above, the two `'a'`s are considered duplicates, and the changing of one changes the other.

If we use our system on a variant of *nub* of the suitable type $nub' :: Eq\,\gamma \Rightarrow \forall\alpha\ \mu.PackM\,\gamma\,\alpha\,\mu \Rightarrow [\alpha] \to \mu\,[\alpha]$, the result is different due to the different interpretation of duplication. Our system assigns locations as [`'a'`@1, `'b'`@2, `'b'`@3, `'a'`@4], and applying nub' returns the view [`'a'`@1, `'b'`@2] and possibly (depending on the implementation) the following history.

Observation	Argument-1	Argument-2	Result
==	`'a'`@1	`'b'`@2	*False*
==	`'a'`@1	`'a'`@4	*True*
==	`'b'`@2	`'b'`@3	*True*

The same update [(1, `'c'`)] that turns `'a'` into `'c'` is rejected when checking the history.

It is however easy to switch between the above two interpretations in our system: only *assignLocs* needs to be changed.

8. RELATED WORK

Another way of bidirectionalizing programs is through syntactically transforming forward-function definitions [14], a technique termed as syntactic bidirectionalization in [19], in contrast to the semantic approach that does not inspect the program definitions. The advantages of the syntactic approach is that it can derive more efficient and effective (in the sense of allowing certain shape updates) backward transformations. For example, the technique in [14] can derive a backward transformation for function *zip* that accepts arbitrary updates on the view, including insertion and deletion of elements. On the downside, since syntactic bidirectionalization inspects the definition of a program, the resulting backward transformation depends on the syntactic structure of the forward transformation, which is fragile and less predictable. Moreover, the ability of permitting more updates comes partly at the cost of the expressiveness of the forward transformation. It is usually much harder to develop programs that are suitable for syntactic bidirectionalization than that for semantic bidirectionalization.

Instead of trying to bidirectionalize unidirectional programs, one can try to program directly in a "bidirectional" language, in which the resulting programs are bidirectional by construction. Such bidirectional languages are usually combinator based [4, 8, 13, 16, 17, 25], and the programmer builds a bidirectional transformation by combining smaller ones with special combinators. Some combinator languages can be implemented as libraries [13,16], which is rather lightweight, while some languages [4, 8, 25] need richer type systems, which are not available in most general-purpose languages, to be effective. It is usually easy to extend the languages by adding or removing combinators for specific domains [4, 17, 25], and typically users of the bidirectional languages have better control of the behaviors of the programs by not relying on a black-box bidirectionalization system; but the users do have to program in an unusual style and is limited by the expressiveness of the languages.

The use of runtime recording of the forward execution path for the backward execution is not new [7, 8, 14]. The lens framework [8] provides a combinator *ccond* that performs conditional branching in the forward execution, and in the backward direction, the recorded history prohibits updates that may cause the execution to take a different branch. This treatment of branching is more explicit in [14], where branching information is recorded in a complement, which is kept constant to guarantee the bidirectional laws. Fegaras [7] adopts runtime recording to propagate updates through XML views over relational databases, ensuring the translation satisfies the consistency.

Both ours and the original work on semantic bidirectionalization [19] focus on in-place updates, which is a non-trivial problem when the forward transformations are complex [12], It is shown that this limitation of updates can be relaxed to some extent by combining semantic bidirectionalization with other techniques that are good at shape updates [21]. We expect that a similar extension is also applicable to our proposal.

We omit the issue of performance in this paper and opt for simple declarative definitions over more efficient variants. It is obvious that some traversals in our implementation may be fused and more involved data structures such as binary search trees and hash tables can be used instead of lists. Another performance aspect relevant to bidirectional transformation is incremental computation [26], as updates are expected to be relatively small comparing to the source. We expect semantic bidirectionalization, both the original work [19] and our result, to fit the requirement for incremental computation of updates because they essentially map update operations on the view to those on the source.

9. CONCLUSION

In this paper, we extend semantic bidirectionalization to handle monomorphic transformations, by programming them in a polymorphic way through a type class *PackM*. Specifically, we replace monomorphic values in the definition of a transformation with polymorphic elements that are newly constructed from those values. A history of the program execution is recorded at runtime, which can be checked to reinstate the applicability of free theorems in the presence of newly constructed polymorphic elements. We prove that the transformations produced by our bidirectionalization system satisfy the bidirectional properties, *i.e.*, the acceptability and consistency laws. The practicality of our system is demonstrated by a case study of XML transformations.

As future work, we plan to design new or adapt existing domain-specific languages to use our system as a backend, so that bidirectionality can be more easily achieved.

10. ACKNOWLEDGMENTS

We would like to thank Akimasa Morihata, Jean-Philippe Bernardy and Janis Voigtländer for their helpful comments on a preliminary version of the paper. This work was partially supported by JSPS KAKENHI Grant Number 24700020.

11. REFERENCES

[1] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[2] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free – Parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012.

[3] R. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers. Understanding idiomatic traversals backwards and forwards. In *Haskell*, 2013. To appear.

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419. ACM, 2008.

[5] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, *LNCS* 5563, pages 260–283, 2009. Springer.

[6] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[7] L. Fegaras. Propagating updates through XML views using lineage tracing. In *ICDE*, pages 309–320, 2010. IEEE.

[8] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[9] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In J. Hook and P. Thiemann, editors, *ICFP*, pages 383–396. ACM, 2008.

[10] N. Foster, K. Matsuda, and J. Voigtländer. Three complementary approaches to bidirectional programming. In *Generic and Indexed Programming*, *LNCS* 7470, pages 1–46. Springer, 2012.

[11] S. J. Hegner. Foundations of canonical update support for closed database views. In S. Abiteboul and P. C. Kanellakis, editors, *ICDT*, *LNCS* 470, pages 422–436. Springer, 1990.

[12] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP*, pages 205–216. ACM, 2010.

[13] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM*, pages 178–189. ACM, 2004.

[14] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP*, pages 47–58. ACM, 2007.

[15] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[16] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *APLAS*, *LNCS* 3302, pages 2–20. Springer, 2004.

[17] R. Rajkumar, S. Lindley, N. Foster, and J. Cheney. Lenses for web data. In Preliminary Proceedings of Second International Workshop on Bidirectional Transformations (BX 2013), 2013.

[18] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, pages 513–523. Elservier Science Publishers B.V. (North-Holland), 1983.

[19] J. Voigtländer. Bidirectionalization for free! (pearl). In *POPL*, pages 165–176. ACM, 2009.

[20] J. Voigtländer. Free theorems involving type constructor classes: functional pearl. In *ICFP*, pages 173–184. ACM, 2009.

[21] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *ICFP*, pages 181–192. ACM, 2010.

[22] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(2):175–210, 2010.

[23] P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

[24] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP*, pages 148–159. ACM, 1999.

[25] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement: Blending pattern matching with data abstraction. In *MPC*, *LNCS* 6120, pages 397–425. Springer, 2010.

[26] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In *ICFP*, pages 392–403, ACM, 2011.