# A Type Class for Bidirectionalization

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

Kazutaka Matsuda  Meng Wang

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 $\gamma$ is abstracted to a type $\alpha$, and the 'observations' made by a transformation on values of type $\gamma$ are recorded by a monad $\mu$". 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.

## 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][5] 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 us 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. 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

---

双方向化のための型クラス、あるいはビュー更新問題への軽量なアプローチ

松田 一孝, 東京大学, The University of Tokyo.

Meng Wang, チャルマーズ工科大学, Chalmers University of Technology.

The international-conference version of this paper will appear in PPDP 2013

```
<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>
```

**Fig. 1 An XML Source**

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

Using terminologies advocated by the *lens* framework [6] 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. The correctness of the pair of functions is governed by the following *definitional properties* [3]:

**Consistency** $\quad get\ s' = v \quad \text{if} \quad put\ s\ v = s'$

**Acceptability** $\quad put\ s\ (get\ s) = s$

```
<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>
```

**Fig. 2 Query Q1**

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

**Fig. 3 Result of Applying Q1 to the Source in Figure 1**

Here *consistency* (also known as the PutGet law [6]) 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 that 1992 in the above example violates this law), and *acceptability* (also known as the GetPut law [6]) 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* [6]. In additions 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 expect 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 [13] [6] [7] [2]: forward, as ordinary functions from source to view, and backward, from original source and updated view to updated source. A disadvantage of the combinator-based approach is that transformations have to be encoded in a some-

what inconvenient programming style.

Other than creating special purpose bidirectional languages, an alternative is to mechanically transform existing unidirectional programs to obtain a backward counterpart, a technique known as *bidirectionalization* [17]. Different flavors of bidirectionalization have been proposed: syntactic [17], semantic [22], and a combination of the two [24]. 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 [25] arising from parametricity [21] 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 (sometime awkward) 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 [22] [24] because the possibility of creating values of polymorphic types complicates the use of free theorems.

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 uses run-time logging to reinstate the applicability of free theorems amid the creation of polymorphic elements, which in turn provides a solution to the problem of comparison with constant values.

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

Intuitively, a type class *PackM* encapsulates the fact that a concrete datatype $\gamma$ is abstracted to a type $\alpha$, and the "observations" made by a transformation on values of type $\gamma$ are recorded by a monad $\mu$ so that the information can be used to check the validity of updates. For example, in our library the equality comparison found in query Q1 can be expressed by the following code fragment (the complete code can be found in Figure 4 in Section 6)

$$\mathit{liftO2}\ (\texttt{==})\ (\mathit{label}\ p)\ (\mathit{tx}\ \texttt{"Addison-Wesley"})$$

In the above the constant `"Addison-Wesley"` is turned into a polymorphic value through the application of the function *new*, which does not instantiate the type, and the lifting operator *liftO2*, which is a specialized version of *liftO*, enables run-time logging of the equality comparison `==` through a monad. As a result, the backward transformation is able to reject any changes to the publishers in the view to satisfy the consistency law.

The rest of the paper is organized as follows. In Section 2, we firstly review the concept of semantic bidirectionalization [22], 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 [23]. 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 [22]. Consider that we are given a polymorphic function of type $\forall\alpha.\ [\alpha] \to [\alpha]$. Parametricity [21] 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 updates to a view element to be translated into an update to its origin in the source, which forms the basis of a backward transformation. In a sense, we can see semantic bidirectionalization as a functional way to characterize a "pointer-based" update-translation mechanism satisfying the consistency law.

### 2.1 Construction of Backward Transformation

We present a simple implementation of semantic bidirectionalization that captures the core idea found in the original paper [22], which will be expanded in Section 3. We assume basic knowledge of Haskell 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*. Its type tells us 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 behaviour 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 (pointer). 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 show of correspondence.

Now 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 that is assigned 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) =
    case lookup i upd of
        Nothing → Loc x i
        Just y  → Loc y i
```

And the matching of the location-aware view and the updated view that produces the update is defined as the following.

```
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 "Inconsitent 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 duplicates from an update, which is defined as the following.

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

Here, (\\), 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 behaviour 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 counter-part can be realized as the following.

$$\begin{aligned}
&bwd :: Eq\ \gamma \Rightarrow (\forall\alpha.\ [\alpha] \to [\alpha]) \to [\gamma] \to [\gamma] \to [\gamma] \\
&bwd\ h = \lambda s\ v.\ \mathbf{let}\ sx\ \ = zipWith\ Loc\ s\ [1..] \\
&\qquad\qquad\qquad\qquad\ \ vx\ \ = h\ sx \\
&\qquad\qquad\qquad\qquad\ \ upd = matchViewsSimple\ vx\ v \\
&\qquad\qquad\qquad\ \mathbf{in}\ map\ (body \circ update\ upd)\ sx
\end{aligned}$$

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 [22] [8].

### 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 [22], the problem is solved by a more sophisticated indexing scheme catered for each observer function. In the case of *nub*, where equality is used, we need to make sure that the indices fully reflect the equality among elements: indices 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 indexing system to mimic the actual source is not enough to handle functions that construct elements of the polymorphic type. 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

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

$$\begin{aligned}
&\mathbf{data}\ Tree\ \alpha = Node\ \alpha\ [Tree\ \alpha] \\
&\qquad \mathbf{deriving}\ (Eq, Functor)
\end{aligned}$$

To run the codes in this section, the following language options must be turned on: `DeriveFunctor`, `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

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

we get the following view.

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

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.

$$\begin{aligned}
&links_{mono} :: Tree\ String \to Tree\ String \\
&links_{mono}\ t = Node\ \texttt{"results"}\ (linkss_{mono}\ t) \\
&linkss_{mono} :: Tree\ String \to [Tree\ String] \\
&linkss_{mono}\ (Node\ n\ ts) = \\
&\quad \mathbf{if}\ n\ \texttt{==}\ \texttt{"a"}\ \mathbf{then} \\
&\qquad Node\ \texttt{"a"}\ ts \\
&\quad \mathbf{else} \\
&\qquad concatMap\ linkss_{mono}\ ts
\end{aligned}$$

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

### 3.1 Making Monomorphic Queries Polymorphic

The reasons for $links_{mono}$ 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 a polymorphic type, and function *eq* compares polymorphic labels for equality. With them, we can implement our query as a polymorphic function.

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

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 polymorphic values and compare them with arbitrary labels, the free theorems of the type $\forall\alpha.\ \mathit{PackTrial}\ \gamma\ \alpha \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha$ are no longer strong enough to support 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. In such a case, accepting any update risks violating the consistency law, 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 overconservative: for a given query such as Links, not all the updates can affect its control. For example, updating `"a"` to any other strings in $\mathit{view}_{links}$ 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 construction of new labels into a different class *Pack*.

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

In this new design, we no longer deal with specific observer functions such as *eq*; instead function *liftO* lifts any observer function $\mathit{Eq}\ \beta \Rightarrow [\gamma] \to \beta$ on a concrete datatype $\gamma$ to a monadic one $\mathit{Eq}\ \beta \Rightarrow [\alpha] \to \mu\ \beta$ on an abstracted datatype $\alpha$. 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 two specific instances of *liftO* that operates on unary and binary observer functions respectively.

$$\begin{aligned}\mathit{liftO1}\ p\ x &= \mathit{liftO}\ (\lambda[x].p\ x)\ [x]\\ \mathit{liftO2}\ p\ x\ y &= \mathit{liftO}\ (\lambda[x,y].p\ x\ y)\ [x,y]\end{aligned}$$

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 $\alpha$ which is suitable for semantic bidirectionalization. And importantly, the type is polymorphic also in $\mu$ so that the monad cannot be manipulated directly in the definitions of the forward functions, which guarantees the in-

tegrity of the observation results recorded in the monad.

### 3.3 Extended Semantic Bidirectionalization (An Overview)

We are now ready to bidirectionalize Links. In our new setting, the query can be defined as the following.

```
links :: PackM String α μ ⇒ Tree α → μ (Tree α)
links t = do as ← linkss t
             return $ Node (new "results") as

linkss :: PackM String α μ ⇒ Tree α → μ [Tree α]
linkss (Node n ts) =
  do b ← liftO2 (==) n (new "a")
     if b then
       return $ Node (new "a") 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 $links_{poly}$.

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

```
data Loc α = Loc {body :: α, location :: 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 *Loc x* (*Just i*) and $x@\#$ for *Loc x Nothing*.

Let's assume that a monadic infrastructure is prepared (we will see how it is done very soon in Section 3.5). Applying *links* to a location-aware version $srcx_{links}$ of $src_{links}$ 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 $viewx_{links}$ of $view_{links}$

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

together with the following observation history that is recoded in the monad.

| Observation | Arg-1 | Arg-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 that are 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, "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, "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 | Arg-1 | Arg-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 the rest of this section, we present a realization of the above discussed idea.

### 3.4 Forward Execution

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]

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

$$\begin{aligned}&\textbf{newtype } I\ \alpha = I\ \{runI :: \alpha\}\\&\textbf{instance } Monad\ I \textbf{ where}\\&\quad return = I\\&\quad I\ x \gg\!= f = I\ (f\ x)\end{aligned}$$

And accordingly we prepare the following instances of *Pack* and *PackM*.

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

In the above, $I\ \gamma$ is used instead of $\gamma$ to satisfy the functional dependency required by *Pack*.

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

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

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

$$\begin{aligned}&linksF :: Tree\ String \to Tree\ String\\&linksF = fwd\ links\end{aligned}$$

We can apply *linksF* directly to sources as in $linksF\ src_{links} = view_{links}$. □

### 3. 5 Backward Execution

The construction of backward transformations is a bit more complicated than that of forward ones. As explained in the previous section, updates are reflected in 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 their specifications, especially for the ones that may have multiple different correct implementations.

#### 3. 5. 1 Locations

As mentioned in Section 3. 3, 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.

$$\begin{aligned}&\textbf{instance } Pack\ \gamma\ (Loc\ \gamma) \textbf{ where}\\&\quad new\ x = Loc\ x\ Nothing\end{aligned}$$

Just like pointers, the mapping between values and locations has to be injective: only one value can be assigned to a particular location.

**Definition 1** (Location Consistency)**.** Let $\gamma$ to be a label type. A tree $t :: Tree\ (Loc\ \gamma)$ 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.

$$assignIDs :: Tree\ \gamma \to Tree\ (Loc\ \gamma)$$

And it must satisfy the following conditions.

**Condition** (*assignIDs*)**.** *assignIDs* must satisfy: For all $s$

- $assignIDs\ s$ is location consistent, and
- $fmap\ body\ (assignIDs\ s) = s$. □

This specification is rather loose and leave room for different implementations of *assignIDs*. 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. 5. 2 Observation History

The observation history is represented as the following datatype.

$$\begin{aligned}&\textbf{data } Result\ \alpha =\\&\quad \forall\beta.Eq\ \beta \Rightarrow Result\ ([\alpha] \to \beta)\ [\alpha]\ \beta\end{aligned}$$

Roughly speaking, a value $Result\ p\ [x_1, \ldots, x_n]\ r$ corresponds to a line in the observation table shown in Section 3. 3, as below.

| Observation | Arg-1 | ... | Arg-$n$ | 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.

$$\begin{aligned}&\mathit{check} :: \mathit{Result}\ \alpha \to \mathit{Bool}\\&\mathit{check}\ (\mathit{Result}\ p\ \mathit{xs}\ r) = p\ \mathit{xs} == r\end{aligned}$$

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

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

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

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

which allows us to define the following instance of $\mathit{PackM}$ for backward executions.

$$\begin{aligned}&\textbf{instance}\ \mathit{PackM}\ \gamma\ (\mathit{Loc}\ \gamma)\ (W\ (\mathit{Loc}\ \gamma))\ \textbf{where}\\&\quad \mathit{liftO}\ p\ x = W\ (p'\ x, \mathit{Result}\ p'\ x\ (p'\ x))\\&\qquad \textbf{where}\ p' = p \circ \mathit{map}\ \mathit{body}\end{aligned}$$

### 3. 5. 3 Updates

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

$$\begin{aligned}&\mathit{update} :: \mathit{Update}\ \gamma \to (\mathit{Loc}\ \gamma) \to (\mathit{Loc}\ \gamma)\\&\mathit{update}\ \mathit{upd}\ (\mathit{Loc}\ x\ \mathit{Nothing}) = \mathit{Loc}\ x\ \mathit{Nothing}\\&\mathit{update}\ \mathit{upd}\ (\mathit{Loc}\ x\ (\mathit{Just}\ i)) =\\&\quad \textbf{case}\ \mathit{lookup}\ i\ \mathit{upd}\ \textbf{of}\\&\qquad \mathit{Nothing} \to \mathit{Loc}\ x\ (\mathit{Just}\ i)\\&\qquad \mathit{Just}\ y \quad \to \mathit{Loc}\ y\ (\mathit{Just}\ i)\end{aligned}$$

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

**Condition** ($\mathit{update}$)**.** $\mathit{update}$ satisfies the following conditions. For any $x$,

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

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

$$\begin{aligned}&\mathit{matchViews} ::\\&\quad \mathit{Eq}\ \gamma \Rightarrow \mathit{Tree}\ (\mathit{Loc}\ \gamma) \to \mathit{Tree}\ \gamma \to \mathit{Update}\ \gamma\end{aligned}$$

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

**Condition** ($\mathit{matchViews}$)**.** $\mathit{matchViews}$ must satisfy:

- **Correctness.** for any $v'$ and location-consistent $vx$, let $\mathit{upd} = \mathit{matchViews}\ vx\ v'$, $\mathit{fmap}\ (\mathit{body} \circ \mathit{update}\ \mathit{upd})\ vx = v'$ holds.
- **Minimality.** for any $v$ and location-consistent $vx$ such that $\mathit{fmap}\ \mathit{body}\ vx = v$, $\mathit{matchViews}\ vx\ v = []$ holds. □

### 3. 5. 4 Putting Everything Together

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

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

**Example 3** ($\mathit{links}$)**.** We can instantiate $\mathit{links}$ for backward execution as follows.

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

Suppose that $\mathit{view}_{\mathit{links}}$ is updated to the following tree.

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

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

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

$$\begin{array}{l} \mathit{Node}\ \texttt{"root"}\ [ \\ \quad \mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"changed"}\ []], \\ \quad \mathit{Node}\ \texttt{"p"}\ [\mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"text2"}\ []]]] \end{array}$$

On the other hand, an update to the view $\mathit{view}''$

$$\begin{array}{l} \mathit{view}'' = \mathit{Node}\ \texttt{"results"}\ [ \\ \qquad\qquad \mathit{Node}\ \texttt{"b"}\ [\mathit{Node}\ \texttt{"text"}\ []], \\ \qquad\qquad \mathit{Node}\ \texttt{"a"}\ [\mathit{Node}\ \texttt{"text2"}\ []]] \end{array}$$

is rejected for the reason discussed in Section 3. 3. □

## 4 Correctness of Our Bidirectionalization

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

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

**Acceptability** $\quad \mathit{bwd}\ h\ s\ (\mathit{fwd}\ h\ s) = s$

**Consistency** $\quad \mathit{fwd}\ h\ s' = v \quad \text{if} \quad \mathit{bwd}\ h\ s\ v = s'$

Following the original work [22], we make use of free theorems [25] [23] in the proofs, and aim at being "morally correct" [4]; that is, our reasoning is correct in the absence of $\bot$. In another words, we 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 [25] [23] are theorems obtained for free as corollaries of relational parametricity [21], which states that, for a term $f$ of type $\tau$, $(f, f)$ belongs to a certain relational interpretation of $\tau$. A simple example of a free theorem is that $f$ of type $\forall\alpha.[a] \to [a]$ satisfies $\mathit{map}\ g \circ f = f \circ \mathit{map}\ g$ for any function $g$ of type $\sigma \to \tau$. This theorem is key to the correctness of the original work [22].

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 a base type $B$, we abuse the notation to write $B$ for the relation $\{(f, f) \mid f :: B\}$. 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 $\sigma$, we write $f_\sigma$ for the instantiation of $f$ to $\sigma$ that has type $\tau[\sigma/\alpha]$. For simplicity, we sometimes omit the subscript and simply write $f$ for $f_\sigma$ if $\sigma$ is clear from the context or irrelevant.

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

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

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

$$\begin{array}{ll} [\![[\tau]]\!]_\rho & = [[\![\tau]\!]_\rho] \\ [\![\mathit{Tree}\ \tau]\!]_\rho & = \mathit{Tree}\ [\![\tau]\!]_\rho \end{array}$$

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

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

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

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

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 $\mathit{Tree}\ \mathcal{R}$ 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 $\tau$, $(f, f)$ 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 $\mathcal{R} : \sigma_1 \leftrightarrow \sigma_2$, $(f_{\sigma_1}, f_{\sigma_2}) \in [\mathcal{R}] \to [\mathcal{R}]$, if we take $\mathcal{R} = \{(x, g\ x) \mid x :: \sigma_1\}$ for some $g :: \sigma_1 \to \sigma_2$, we obtain $\mathit{map}\ g \circ f = f \circ \mathit{map}\ g$.

Voigtländer [23] extends parametricity to a type

system with type constructors. A key notation in his result is *relational action*.

**Definition 2** (Relational Action [23])**.** For type constructor $\kappa_1$ and $\kappa_2$, $\mathcal{F}$ is called a *relational action* between $\kappa_1$ and $\kappa_2$, 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 $\tau_1$ and $\tau_2$ to $\mathcal{F}\ \mathcal{R} : \kappa_1\ \tau_1 \leftrightarrow \kappa_2\ \tau_2$. □

Accordingly, the relational interpretation is extended as:

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

Parametricity holds also on the relational interpretation. Here, $\kappa$, $\kappa_1$ and $\kappa_2$ are type constructors of kind $* \to *$, and thus the quantified $\mathcal{F}$ is a relational action.

Voigtländer [23] 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 $f :: \mathit{Monad}\,\mu \Rightarrow \tau$ 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 $f' = \lambda \mathit{return}.\lambda(\gg\!=).f$, and an instance $f_\kappa$ of $f$ can be seen as $f'\ \mathit{return}_\kappa\ (\gg\!=)_\kappa$. 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 $i = 1, 2$,
- $(\mathit{return}_{\kappa_1}, \mathit{return}_{\kappa_2}) \in \forall\mathcal{R}.\ \mathcal{R} \to \mathcal{F}\,\mathcal{R}$,
- $((\gg\!=)_{\kappa_1}, (\gg\!=)_{\kappa_2}) \in \forall\mathcal{R}.\forall\mathcal{S}.\ \mathcal{F}\,\mathcal{R} \to ((\mathcal{R} \to \mathcal{F}\,\mathcal{S}) \to \mathcal{F}\,\mathcal{S})$,
- $(\mathit{new}_{\sigma_1,\tau_1}, \mathit{new}_{\sigma_2,\tau_2}) \in \mathcal{L} \to \mathcal{U}$, and
- $(\mathit{liftO}_{\sigma_1,\tau_1,\mu_1,\beta_1}, \mathit{liftO}_{\sigma_2,\tau_2,\mu_2,\beta_2}) \in ([\mathcal{L}] \to \mathcal{S}) \to [\mathcal{U}] \to \mathcal{F}\ \mathcal{S}$ for all $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ satisfying $((==)_{\beta_1}, (==)_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$. □

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

**Theorem 1** (A Free Theorem on $h$)**.** *Suppose $h$ be a function of type $\forall\alpha.\forall\mu.\,\mathit{PackM}\ \gamma\,\alpha\,\mu \Rightarrow \mathit{Tree}\,\alpha \to \mu\,(\mathit{Tree}\,\alpha)$. Let $\gamma$ be a type and $\mathcal{L}$ be a relation $\{(x,x) \mid x :: \gamma\}$. Let $(\tau_1,\kappa_1)$ and $(\tau_2,\kappa_2)$ are 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 $(\mathcal{L} :: \gamma \leftrightarrow \gamma, \mathcal{T} :: \tau_1 \leftrightarrow \tau_2, \mathcal{R} :: \kappa_1 \leftrightarrow \kappa_2)$, we have $(h_{\tau_1,\kappa_1}, h_{\tau_2,\kappa_2}) \in \mathit{Tree}\ \mathcal{U} \to \mathcal{F}\ (\mathit{Tree}\ \mathcal{U})$.* □

### 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 $W\ (vx, \_) = h\ (\mathit{assignIDs}\ s)$ for a tree $s$. Then, vx is location consistent.*

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

The statement is proved by the free theorem on $h$, taking $\mathcal{U}$ and $\mathcal{F}$ as below.

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

Note that $\mathit{Tree}\,\mathcal{U}$ is a diagonal relation, and $v$ such that $(v,v) \in \mathit{Tree}\,\mathcal{U}$ is location-consistent. Then, assuming that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action, we have $(h,h) \in \mathit{Tree}\,\mathcal{U} \to \mathcal{F}\,(\mathit{Tree}\,\mathcal{U})$. Since $(\mathit{assignIDs}\ s, \mathit{assignIDs}\ s) \in \mathit{Tree}\,\mathcal{U}$, we have $(vx, vx) \in \mathit{Tree}\,\mathcal{U}$. Thus, *vx* is location consistent.

Then, the remaining obligation is to check that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action as we have assumed above. We omit the straightforward proof here. □

### 4.3 Proof of Acceptability

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

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

At (*), we use two properties: one is $\mathit{upd} = []$, and the other is $\mathit{checkHist}\ (\mathit{update}\ [])\ \mathit{hist} = \mathit{True}$.

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 assignIDs s. Suppose we have* $W\,(vx, hist) = h\,sx$ *and* $I\,v = h\,(fmap\,I\,s)$. *Then, we have fmap body vx = fmap runI v and checkHist id hist = True.* □

*Proof.* 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:

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

Assume that the triple $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action. Then, we can obtain $(h,h) \in Tree\,\mathcal{U} \to \mathcal{F}\,(Tree\,\mathcal{U})$ from the free theorem on $h$; that is, for any $x$ and $y$ satisfying $fmap\,runI\,x = fmap\,body\,y$, we have $fmap\,runI\,v = fmap\,body\,vx$ and $checkHist\,id\,hist = True$ where $I\,v = h\,x$ and $W\,(vx, hist) = h\,y$. Taking $x = fmap\,I\,s$ and $y = assignIDs\,s$, we obtain the lemma.

Then, the remaining obligation is to prove that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action. We omit the straightforward proof. □

### 4.4 Proof of Consistency

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

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

At (*), we used the following lemma.

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

*Proof.* 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:

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

Then, assuming that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action, we can prove the lemma straightforwardly from the free theorem on $h$.

Then, the remaining obligation is to prove that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action. We omit the straightforward proof. □

## 5 Going Generic

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

As a matter of fact, this generalization has already been done in the previous work [22] [8], and we borrow the ideas and adapt them for our new setup. More concretely, we use the following datatype-generic functions from `Data.Traversable` and `Data.Foldable` to define *assignIDs* and *matchViews*, and change the type declarations of *fwd* and *bwd* accordingly.

```
traverse :: (Traversable κ, Applicative θ) ⇒
            (α → θ β) → κ α → θ (κ β)
toList :: (Foldable κ) ⇒ κ α → [α]
```

We redefine *assignIDs* as a function of type

```
assignIDs :: Traversable κ ⇒ κ γ → κ (Loc γ)
```

with definition

```
assignIDs t = evalState (traverse f t) 0
  where f x =
          do i ← Control.Monad.State.get
             Control.Monad.State.put (i + 1)
             return $ Loc x (Just i)
```

Here, we qualified the state monad operations *get* and *put* with *Control.Monad.State* to distinguish them from the *get* and *put* as bidirectional transformations. The conditions we mentioned in Section 3.5 hold for typical instances of *Traversable*, especially for those that are systematically obtained [18].

We also redefine *matchViews*.

```
matchViews :: (Foldable κ, Eq (κ ())) ⇒
                  κ (Loc γ) → κ γ → Update γ
```

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
                  error "Inconsistent Update"
```

The conditions we posed on *matchViews* hold for typical instances of *Foldable*, with which *toList* extracts all the elements of a container.

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

```
fwd :: (Traversable κ1, Foldable κ2,
        Eq (κ2 ()), Eq γ) ⇒
        (∀α μ.PackM γ α μ ⇒ κ1 α → μ (κ2 α))
        → κ1 γ → κ2 γ
bwd :: (Traversable κ1, Foldable κ2,
        Eq (κ2 ()), Eq γ) ⇒
        (∀α μ.PackM γ α μ ⇒ κ1 α → μ (κ2 α))
        → κ1 γ → κ2 γ → κ1 γ
```

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

It is worth noting that the GHC extensions `-XDeriveFoldables` and `-XDeriveTraversables` allow *Foldable* and *Traversable* instances 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 shown in Figure 1 we get the XML view shown 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.

```
data L = A String | E String | T String 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 $\forall\alpha.\forall\mu.\ PackM\ L\ \alpha\ \mu \Rightarrow Tree\ \alpha \to \mu\ (Tree\ \alpha)$ that is suitable for bidirectionalization. We will use a few auxiliary "filters" [26] that performs various content processings. In [26], the filters (if we simplify and customize them to our rose trees) are of type

*Tree L* → [*Tree L*], which will be made polymorphic in our setting as

$$PackM\ L\ \alpha\ \mu \Rightarrow Tree\ \alpha \rightarrow ListT\ \mu\ (Tree\ \alpha).$$

where we use a monad transformer *ListT* in `Control.Monad.List`, defined by:

$$\textbf{newtype}\ ListT\ \mu\ \alpha = ListT\ \{runListT :: \mu\ [\alpha]\}$$

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 $ListT\ \mu$ 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 *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)
```

Another example is *ofL l* that returns its argument as is if the node has label *l*, and fails otherwise.

```
ofL :: PackM L α μ ⇒
       α → Tree α → ListT μ (Tree α)
ofL 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 do nothing otherwise.

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

Filters are composable [26]. For example, we can compose *children* and *ofL* to obtain a filter *childrenOfL* that extracts the children of a given label, as follows.

$$childrenOfL\ l\ x = \textbf{do}\ \{c \leftarrow children\ x; ofL\ l\ c\}$$

Or by using the Kleisli-composition operator (>=>) in `Control.Monad`, we can write

$$childrenOfL\ l = children >=> ofL\ l$$

Now we are ready to implement Q1. The code can be found in Figure 4 and is self-explanatory.

### 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 *childrenOfL* (*el* `"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 Discussion

In this section, we discuss design issues that are common to bidirectional frameworks.

### 7.1 The Interpretation of Duplication

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 prac-

```
q1 :: PackM L α μ ⇒ Tree α → μ (Tree α)
q1 t = pick $ do bs ← gather $ childrenOfL (el "book") t >>= h
                 return $ Node (el "bib") bs
  where
    h b =
      do y ← childrenOfL (at "year") b >>= children
         ts ← gather $ childrenOfL (el "title") b
         p ← childrenOfL (el "publisher") b >>= children
         guardM $ lift $ liftO2 gtInt (label y) (tx "1991")
         guardM $ lift $
                    liftO2 (==) (label p) (tx "Addison-Wesley")
         return $ Node (el "book")
                       (Node (at "year") [y] : ts)
    gtInt l1 l2 = (read l1 :: Int) > (read l2 :: Int)

tx s = new (T s)
el s = new (E s)
at s = 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
```

**Fig. 4 Query Q1 in Our Framework**

tice, it is less obvious whether two values are actual duplicates, or merely being incidentally equal.

As mentioned in Section 3.5.1, 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 [22], when the system is extended to non-fully polymorphic forward functions such as

$$\forall\alpha.\ Eq\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$$

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 *assignIDs* 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 $nub' :: Eq\ \gamma \Rightarrow \forall\alpha\ \mu.PackM\ \gamma\ \alpha\ \mu \Rightarrow [\alpha] \rightarrow \mu\ [\alpha]$, 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 | Arg-1 | Arg-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 *assignIDs*

needs to be changed.

### 7.2 Bidirectional Properties

We have discussed in this paper the definitional properties of bidirectional transformations, namely the acceptability and consistency laws. There are other laws that are mentioned in the literature, which are very often considered optional but desirable. Two of them are undoability and composability [17][1][10][11], which are variants of PutPut [6][†3]

**Composability** $put\ s\ v = s' \wedge put\ s'\ v' = s'' \Rightarrow put\ s\ v' = s''$

**Undoability** $put\ s\ v = s' \Rightarrow put\ s'\ (get\ s) = s$

Intuitively composability says that subsequent updates can be combined, and undoability says that an update can be undone through the view. These laws are useful properties especially when updates on the view and source sides are conducted by different systems [9][10]: for example, composability allows us to schedule updates on the view separated from those on the source [9]. Without proofs, we state that our derived bidirectional transformations satisfy both the composability and undoability laws. The justification of this claim comes from the fact that semantic bidirectionalization can be seen as a form of constant-complement bidirectionalization [1][8], in which the four laws, *i.e.*, acceptability, consistency, composability and undoability, hold. Specifically, the shape of the source and the observation history serve as the constant complement in our framework.

The desirable laws are sometimes considered too restrictive [16][6] because a framework that supports shape changes (insertion and deletion of elements) of views is unlikely to be able to satisfy them (for example, consider a forward transformation *map fst* and source $[(\texttt{"A"}, 2)]$, and consider what happens if we change a view from $[\texttt{"A"}]$ to $[]$ and then insert `"A"` again). In [24], where semantic bidirectionalization is combined with another bidirectionalization technique [17] aiming at shape updates, the composability and undoability laws are sacrificed. Further studies on the desirable laws in the presence of insertions and deletions can be found in [9][11][15].

There is another law discussed by Hegner [10][11] that, with the four laws, ensures information not in the view cannot be accessed.

**Uniformity** $put\ s\ v = s' \Rightarrow \begin{pmatrix} \forall s''.get\ s'' = get\ s \\ \quad \Rightarrow \exists s'''.\ put\ s''\ v = s''' \end{pmatrix}$

Intuitively, uniformity says that whether an update on the view is accepted or not does not depend on the source. Our framework does not satisfy this law. For example, for $f$ defined by

$$f\ [x, y] = liftO2\ (==)\ x\ y \gg\!= \lambda\_.[x]$$

whether an update is accepted depends on the equality between $x$ and $y$.

## 8 Related Work

Another way of bidirectionalizing programs is through syntactically transforming forward-function definitions [17], a technique termed as syntactic bidirectionalization in [22], 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 [17] 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 [6][13][19][2][20][27], and the programmer constructs a bidirectional transformation by com-

†3 If *put* is total, both of the composability and undoability coincide with PutPut [6].

bining smaller ones with special combinators. Some combinator languages can be implemented as libraries [13] [19], which is rather light-weight, while some languages [6] [2] [27] 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 [20] [27] [2], 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 run-time recording of the forward execution path for the backward execution is not new [6] [17]. The lens framework [6] 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 [17], where branching information is recorded in a complement, which is kept constant to guarantee the bidirectional laws.

Both ours and the original work on semantic bidirectionalization [22] 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 [24]. We expect that a similar extension is also applicable to our proposal, which will be an interesting direction for future work.

## 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 run-time, 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.

At the moment, our system is general purpose, which is expressive, but lacking in conveniency. 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.

## References


[ 1 ] Bancilhon, F. and Spyratos, N.: Update Semantics of Relational Views, *ACM Trans. Database Syst.*, Vol. 6, No. 4(1981), pp. 557–575.

[ 2 ] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A.: Boomerang: resourceful lenses for string data, *POPL*, Necula, G. C. and Wadler, P.(eds.), ACM, 2008, pp. 407–419.

[ 3 ] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F.: Bidirectional Transformations: A Cross-Discipline Perspective, Berlin, Heidelberg, Springer-Verlag, 2009, pp. 260–283.

[ 4 ] Danielsson, N. A., Hughes, J., Jansson, P., and Gibbons, J.: Fast and loose reasoning is morally correct, *POPL*, Morrisett, J. G. and Jones, S. L. P.(eds.), ACM, 2006, pp. 206–217.

[ 5 ] Dayal, U. and Bernstein, P. A.: On the Correct Translation of Update Operations on Relational Views, *ACM Trans. Database Syst.*, Vol. 7, No. 3(1982), pp. 381–416.

[ 6 ] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Trans. Program. Lang. Syst.*, Vol. 29, No. 3(2007).

[ 7 ] Foster, J. N., Pilkiewicz, A., and Pierce, B. C.: Quotient lenses, *ICFP*, Hook, J. and Thiemann, P.(eds.), ACM, 2008, pp. 383–396.

[ 8 ] Foster, N., Matsuda, K., and Voigtländer, J.: Three Complementary Approaches to Bidirectional Programming, *Generic and Indexed Programming*, Gibbons, J.(ed.), Lecture Notes in Computer Science, Vol. 7470, Springer Berlin Heidelberg, 2012, pp. 1–46.

[ 9 ] Gottlob, G., Paolini, P., and Zicari, R.: Properties and Update Semantics of Consistent Views, *ACM Trans. Database Syst.*, Vol. 13, No. 4(1988), pp. 486–524.

[10] Hegner, S. J.: Foundations of Canonical Update Support for Closed Database Views, *ICDT*, Abiteboul, S. and Kanellakis, P. C.(eds.), Lecture Notes in Computer Science, Vol. 470, Springer, 1990, pp. 422–436.

[11] Hegner, S. J.: An Order-Based Theory of Updates for Closed Database Views, *Ann. Math. Artif. Intell.*, Vol. 40, No. 1-2(2004), pp. 63–125.

[12] Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., and Nakano, K.: Bidirectionalizing graph transformations, In Hudak and Weirich [14], pp. 205–216.

[13] Hu, Z., Mu, S.-C., and Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations, *PEPM*, Heintze, N. and Sestoft, P.(eds.), ACM, 2004, pp. 178–189.

[14] Hudak, P. and Weirich, S.(eds.): *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, ACM, 2010.

[15] Johnson, M. and Rosebrugh, R. D.: Lens put-put laws: monotonic and mixed, *ECEASST*, Vol. 49(2012).

[16] Keller, A. M.: Comments on Bancilhon and Spyratos' "Update Semantics and Relational Views", *ACM Trans. Database Syst.*, Vol. 12, No. 3(1987), pp. 521–523.

[17] Matsuda, K., Hu, Z., Nakano, K., Hamana, M., and Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions, *ICFP*, Hinze, R. and Ramsey, N.(eds.), ACM, 2007, pp. 47–58.

[18] McBride, C. and Paterson, R.: Applicative programming with effects, *J. Funct. Program.*, Vol. 18, No. 1(2008), pp. 1–13.

[19] Mu, S.-C., Hu, Z., and Takeichi, M.: An Algebraic Approach to Bi-directional Updating, *APLAS*, Chin, W.-N.(ed.), Lecture Notes in Computer Science, Vol. 3302, Springer, 2004, pp. 2–20.

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

[21] Reynolds, J. C.: Types, Abstraction and Parametric Polymorphism, *Information Processing*, Mason, R.(ed.), Elservier Science Publishers B.V. (North-Holland), 1983, pp. 513–523.

[22] Voigtländer, J.: Bidirectionalization for free! (Pearl), *POPL*, Shao, Z. and Pierce, B. C.(eds.), ACM, 2009, pp. 165–176.

[23] Voigtländer, J.: Free theorems involving type constructor classes: functional pearl, *ICFP*, Hutton, G. and Tolmach, A. P.(eds.), ACM, 2009, pp. 173–184.

[24] Voigtländer, J., Hu, Z., Matsuda, K., and Wang, M.: Combining syntactic and semantic bidirectionalization, In Hudak and Weirich [14], pp. 181–192.

[25] Wadler, P.: Theorems for Free!, *FPCA*, 1989, pp. 347–359.

[26] Wallace, M. and Runciman, C.: Haskell and XML: Generic Combinators or Type-Based Translation?, *ICFP*, Rémi, D. and Lee, P.(eds.), ACM, 1999, pp. 148–159.

[27] Wang, M., Gibbons, J., Matsuda, K., and Hu, Z.: Gradual Refinement: Blending Pattern Matching with Data Abstraction, *MPC*, Bolduc, C., Desharnais, J., and Ktari, B.(eds.), Lecture Notes in Computer Science, Vol. 6120, Springer, 2010, pp. 397–425.