"Bidirectionalization for Free" for Monomorphic Transformations

Kazutaka Matsuda[a], Meng Wang[b]

[a] *Graduate School of Information Science and Technology, University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo, 113-0033, Japan*
[b] *Computer Science and Engineering, Chalmers University of Technology, 412 96 Göteborg, Sweden*

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, an approach termed semantic bidirectionalization proposed by Voigtländer stands out in terms of user-friendliness. A unidirectional program can be written using arbitrary language constructs, as long as the function it represents 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, as 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

Email addresses: kztk@is.s.u-tokyo.ac.jp (Kazutaka Matsuda), wmeng@chalmers.se (Meng Wang)

transformations into polymorphic ones that are ready to be bidirectionalized. We demonstrate our technique with case studies of typical applications of bidirectional transformation, namely text processing, XML query and graph transformation, which were commonly considered beyond semantic bidirectionalization because of their monomorphic nature.

Keywords: Bidirectional Transformation, Free Theorem, Type Class, Haskell

---

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, 2, 3, 4, 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's consider a (simplified version of an) 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 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

```
<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

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 [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]—this is very much a current problem.

Using terminologies advocated by the *lens* framework [7] 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* [20] (In this paper, we write $e = e'$ with the assumption that neither e nor e' is undefined.):

$$\begin{array}{ll} \textbf{Consistency} & get\, s' = v \quad \text{if} \quad put\, s\, v = s' \\ \textbf{Acceptability} & put\, s\, (get\, s) = s \end{array}$$

```
<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

```
<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

Here *consistency* (also known as the PutGet law [7]) 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 [7]) 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* [7]. In addition to these definitional properties, some desirable laws such as *composability* and *undoability* are also discussed in the literature [1, 4, 5].

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 [6, 7, 9, 11, 13, 14, 17, 21]: forward and backward. A disadvantage of the combinator-based approach is that transformations have to be encoded in a somewhat inconvenient programming style.

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* [12]. Different flavors of bidirectionalization have been proposed: syntactic [12], semantic [15, 19], and a combination of the two [16]. Syntactic bidirection-

alization 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 [22] arising from parametricity [23] 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 such as text processing, XML query and graph transformation, where the transformations 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 [15, 16, 24, 19] based on the standard free theorems [22].

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 to 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, and 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 bidirectionalization [15], 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 constructor classes [25]. In Section 5, we discuss two extensions of our approach, namely a datatype-generic implementation and finer handling of duplicates. In Section 6, we discuss three typical application scenarios of bidirectional transformation,

including the XML example mentioned in this section, and show how they are 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 system and additional examples are available as the `bff-mono` package in Hackage, which also contains more examples. In addition to the optimized implementation that the package provides, it also includes a file `CodesInPaper.hs` containing all the code shown in the paper (with some renaming), for curious readers.

A preliminary version of this paper appeared as [26], under the title "Bidirectionalization for Free with Runtime Recording—Or, a Light-Weight Approach to the View-Update Problem".

2. The Essence of Semantic Bidirectionalization

As a preparation, we firstly introduce the basic idea of semantic bidirectionalization [15]. Consider that we are given a polymorphic function of type $\forall\alpha.\ [\alpha] \to [\alpha]$. Parametricity [23] 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 [15], 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, given 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 locations. 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 [*Loc* `'a'` 1, *Loc* `'a'` 2, *Loc* `'b'` 3]. If we apply the same polymorphic function to it, we get [*Loc* `'a'` 2, *Loc* `'b'` 3], with a clear correspondence.

Suppose that the view `"ab"` is updated to `"cd"`. Matching it to the location-aware view [*Loc* `'a'` 2, *Loc* `'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 :: Update α → Loc α → Loc α
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 α ⇒ [Loc α] → [α] → Update α
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 :: Eq α ⇒ [(Loc α, α)] → Update α
makeUpdSimple = foldr f []
  where
    f (Loc _ i, y) u =
        case lookup i u of
            Nothing              → (i, y) : u
            Just y' | y == y'    → u
                    | otherwise  → 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.

```
minimize :: Eq α ⇒ [Loc α] → Update α → Update α
minimize vx u = u \\ [(i, x) | Loc x i ← 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 behavior of 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 follows.

```
bwd :: (∀α. [α] → [α]) → (∀γ. Eq γ ⇒ [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. The technique generalizes to arbitrary `Traversable` datatypes such as rose trees [15, 8].

2.2. Limitations of the Polymorphism Requirement

Things become more complicated if the forward function is not fully polymorphic. For example, consider the 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 Voigtländer's original paper [15], 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 equality among elements: locations are equal if and only if the elements are equal, and no update is allowed to break this condition. This challenge with non-fully polymorphic forward functions is more comprehensively studied by Wang and Najd [19].

As we will see in the next section, Voigtländer's original technique of creating specialized location-assigning systems to mimic the actual source is not enough to handle functions that are able to construct new elements at runtime. Together with a user-defined comparison operation, an element now can be compared to arbitrary newly-constructed elements. In this case the location must be equal to the element itself to preserve the comparison structure, ruling out any meaningful update. 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: `ExistentialQuantification`, `FlexibleInstances`, `FlexibleContexts`, `MultiParamTypeClasses` and `FunctionalDependencies` in addition to `RankNTypes`.[1]

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

For example, if we apply Links to the source

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

we get the following view.

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

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.

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

However, this function is monomorphic, and hence it is not subject to the existing bidirectionalization technique.

3.1. First Try: 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"`. A technique to prevent this type instantiation is to avoid the

[1] Actually, only rank-2 polymorphism is required for a language with type classes. Note that `Rank2Types` becomes obsolete from GHC 7.8.1.

direct use of constants, and instead construct new labels from them. The following type class *PackTrial* can be used for this purpose.

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

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.

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

Thanks to the functional dependency $\alpha \to \gamma$, Haskell can infer the type γ from the use of $\mathit{eq} :: (\mathit{Eq}\,\gamma, \mathit{PackTrial}\,\gamma\,\alpha) \Rightarrow \alpha \to \alpha \to \mathit{Bool}$.

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 $\forall\alpha.\,\mathit{PackTrial}\ \mathit{String}\,\alpha \Rightarrow \mathit{Tree}\,\alpha \to \mathit{Tree}\,\alpha$ 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 not possible to assign a suitable location to n to model $\mathit{linkss}_{\mathit{poly}}$'s behavior with arbitrary *new*ly created values. And consequently, we lose the ability to guard against invalid updates that alter n to a value no longer equal to $(\mathit{new}\ \texttt{"a"})$, leading to the violation of the consistency law. In this case, no update can be safely accepted, reducing semantic bidirectionalization to a useless state.

3.2. Tracking Observations Using a 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 $view_{links}$ affects the control, but updating `"text"` to other strings (including `"a"`) 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.

Specifically, 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*.

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

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. 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.

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

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 Links can be defined as follows.

$$
\begin{array}{l}
\mathit{links} :: \forall\alpha.\forall\mu.\ \mathit{PackM}\ \mathit{String}\,\alpha\,\mu \Rightarrow \mathit{Tree}\,\alpha \to \mu\,(\mathit{Tree}\,\alpha) \\
\mathit{links}\,t = \mathbf{do}\ \mathit{as} \leftarrow \mathit{linkss}\,t \\
\qquad\qquad\qquad \mathit{return}\ \texttt{\$}\ \mathit{Node}\,(\mathit{new}\ \texttt{"results"})\,\mathit{as} \\
\mathit{linkss} :: \forall\alpha.\forall\mu.\ \mathit{PackM}\ \mathit{String}\,\alpha\,\mu \Rightarrow \mathit{Tree}\,\alpha \to \mu\,[\mathit{Tree}\,\alpha] \\
\mathit{linkss}\,(\mathit{Node}\,n\,\mathit{ts}) = \\
\quad \mathbf{do}\ b \leftarrow \mathit{liftO2}\ (\texttt{==})\,n\,(\mathit{new}\ \texttt{"a"}) \\
\qquad \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\,[\mathit{Node}\,n\,\mathit{ts}] \\
\qquad \mathbf{else}\qquad \mathit{concatMapM}\ \mathit{linkss}\ \mathit{ts} \\
\quad \mathbf{where}\ \mathit{concatMapM}\,h\,x = \mathbf{do}\ \mathit{ys} \leftarrow \mathit{mapM}\,h\,x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\,(\mathit{concat}\,\mathit{ys})
\end{array}
$$

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.[2]

$$\mathbf{newtype}\ I\,\alpha = I\ \{\mathit{runI} :: \alpha\}$$

We omit the instance declaration of $\mathit{Monad}\,I$ because it is standard. Similarly, we prepare the following identity functor N.

$$\mathbf{newtype}\ N\,\alpha = N\ \{\mathit{runN} :: \alpha\}$$

Accordingly, we prepare the following instances of *Pack* and *PackM*.

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Pack}\,\gamma\,(N\,\gamma)\ \mathbf{where} \\
\quad \mathit{new} = N \\
\mathbf{instance}\ \mathit{PackM}\,\gamma\,(N\,\gamma)\,I\ \mathbf{where} \\
\quad \mathit{liftO}\,p\,x = I\,(p\ \texttt{\$}\ \mathit{map}\,\mathit{runN}\,x)
\end{array}
$$

In the above, $N\,\gamma$ is used instead of γ to satisfy the functional dependency required by *PackM*, together with another instance of *PackM* which will be introduced later. One might notice that we can use I instead of N. This is

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

true. We use the different type N to distinguish abstraction of data (N) and abstraction of computation (I).

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

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

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

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

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

3.4. Backward Execution

In this section, 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.

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

Unlike what we saw in Section 2, the location part of the type is optional (represented by the *Maybe* type): a label newly constructed 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 this is done in Section 3.4.3). Applying *links* to $\mathit{srcx}_{\mathit{links}}$, a location-aware version of $\mathit{src}_{\mathit{links}}$ defined as

$$\begin{aligned} \mathit{srcx}_{\mathit{links}} = \mathit{Node}\ (\texttt{"root"}@1)\ [& \\ & \mathit{Node}\ (\texttt{"a"}@2)\ [\mathit{Node}\ (\texttt{"text"}@3)\ []], \\ & \mathit{Node}\ (\texttt{"p"}@4)\ [\mathit{Node}\ (\texttt{"a"}@5)\ [\mathit{Node}\ (\texttt{"text2"}@6)\ []]]] \end{aligned}$$

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

$$\begin{aligned} \mathit{viewx}_{\mathit{links}} = \mathit{Node}\ (\texttt{"results"}@\#)\ [& \\ & \mathit{Node}\ (\texttt{"a"}@2)\ [\mathit{Node}\ (\texttt{"text"}@3)\ []], \\ & \mathit{Node}\ (\texttt{"a"}@5)\ [\mathit{Node}\ (\texttt{"text2"}@6)\ []]] \end{aligned}$$

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 5.1.

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.

$$\textbf{instance } \mathit{Pack}\,\gamma\,(\mathit{Loc}\,\gamma) \textbf{ where}$$
$$\mathit{new}\,x = \mathit{Loc}\,x\,\mathit{Nothing}$$

Just like pointers, only one value can be assigned to a particular location. This 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 leaves room for different implementations of *assignLocs*. When *Nothing* (#) is assigned to a source element, it simply means that the source element is not updatable through a view.[3] Moreover, we can customize the treatment of duplicated elements in the source. In the example in Section 3.4.1, we have chosen the following assignment with running integers.

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

[3]In an extreme case, we could use $\mathit{assignLocs} = \mathit{fmap}\,\mathit{new}$, which would be permitted by the above specification. However, this definition would be useless in practice because it disallows any updates.

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

$$\texttt{"root"}@1, \texttt{"a"}@2, \texttt{"text"}@3, \texttt{"p"}@4, \texttt{"a"}@2, \texttt{"text2"}@5$$

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 consider the former strategy, and a datatype-generic implementation of it can be found in Section 5.1. Nevertheless, we will discuss the implication of the different choices in Section 5.2.

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 3.4.1, 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. Note that using a universally-quantified constructor is a Haskell idiom to represent an existentially-quantified datatype.

The consistency of a history entry can be easily checked.

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

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

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

A "Writer" monad W is responsible for gathering histories[4]

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

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

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

3.4.4. Updates

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

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

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

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

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

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 5.1 where we present a datatype-generic version of it. We require *matchViews* to satisfy the following conditions.

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

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.

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

Example 3 (*links*)**.** We can instantiate *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.

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

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

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

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

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

is rejected for the reason discussed in Section 3.4.1. □

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 the consistency and acceptability properties of 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.

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

Throughout the section, we fix the function h.

Following the original work [15], we make use of free theorems [22, 25, 27, 28] 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 [23], which states that, for a closed 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 $\forall\alpha.[\alpha] \to [\alpha]$ satisfies $\mathit{map}\,g \circ f = f \circ \mathit{map}\,g$ 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 between σ_1 and σ_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, which 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 map-

ping 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 $[\![\forall\alpha.\forall\beta.\alpha \to \beta]\!]$. For a base type B, we also write B for $[\![B]\!]$.

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

$$[\![[\tau]]\!]_\rho = [[\![\tau]\!]_\rho] \qquad [\![\mathit{Tree}\,\tau]\!]_\rho = \mathit{Tree}\,[\![\tau]\!]_\rho$$

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

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

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

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

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 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 $\mathit{map}\,g \circ f = f \circ \mathit{map}\,g$.

Voigtländer [25] extends parametricity to a type system with type constructors. A key notion in his result is *relational action*.

Definition 2 (Relational Action [25])**.** For type constructors κ_1 and κ_2, $\mathcal{F}$ is called a *relational action* between κ_1 and κ_2, denoted by $\mathcal{F} : \kappa_1 \leftrightarrow \kappa_2$, if $\mathcal{F}$ maps any relation $\mathcal{R} : \tau_1 \leftrightarrow \tau_2$ for every 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 this relational interpretation [28, 27]. Here, κ, κ_1 and κ_2 are type constructors of kind $* \to *$, and thus the quantified $\mathcal{F}$ is a relational action.

Voigtländer [25] 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_κ of f can be seen as $f'\,\mathit{return}_\kappa\,(\gg\!=)_\kappa$. Just 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
- $(\mathit{liftO}_{\sigma_1,\tau_1,\kappa_1,\beta_1}, \mathit{liftO}_{\sigma_2,\tau_2,\kappa_2,\beta_2}) \in ([\mathcal{L}] \to \mathcal{S}) \to [\mathcal{U}] \to \mathcal{F}\,\mathcal{S}$ 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)**.** *Let γ be a type and $\mathcal{L}$ be the relation $\{(e,e) \mid e :: \gamma\}$. 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 (τ_1,κ_1) and (τ_2,κ_2) be pairs of types and type constructors such that (γ,τ_1,κ_1) and (γ,τ_2,κ_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 $(h_{\tau_1,\kappa_1}, h_{\tau_2,\kappa_2}) \in \mathit{Tree}\,\mathcal{U} \to \mathcal{F}\,(\mathit{Tree}\,\mathcal{U})$.* □

4.2. Preservation of Location 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 $W\,(vx, \_) = h\,(\mathit{assignLocs}\; s)$ for a tree s. Then, vx is location consistent.

Proof Sketch. Let s be a source and E be the set of labels in $\mathit{assignLocs}\; s$. Recall that $\mathit{assignLocs}\; s$ is assumed to be location-consistent. 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.

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

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

4.3. Proof of Acceptability

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

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

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 $assignLocs\, s$. Suppose $W\,(vx, hist) = h\, sx$ and $I\, v = h\,(fmap\, N\, s)$. Then, we have $fmap\, body\, vx = fmap\, runN\, v$ and $checkHist\, id\, hist = True$.

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

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

We can show that $(\mathcal{L}, \mathcal{U}, \mathcal{F})$ is a $PackM$-action where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$ (see Appendix A.2). Then, we have $(h, h) \in Tree\,\mathcal{U} \to \mathcal{F}\,(Tree\,\mathcal{U})$ by Theorem 1; that is, for any x and y with $fmap\, runN\, x = fmap\, body\, y$, we obtain $fmap\, runN\, v = fmap\, body\, vx$ and $checkHist\, id\, hist = True$ where $I\, v = h\, x$ and $W\,(vx, hist) = h\, y$. Taking $x = fmap\, N\, s$ 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.

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

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 $I\, v' = h\,(fmap\, N\, s')$ and $W\,(vx, hist) = h\, sx$. Suppose we have $checkHist\,(update\, upd)\, hist = True$. Then

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

holds.

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

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

We can show that $(\mathcal{L}, \mathcal{U}, \mathcal{F})$ is a *PackM*-action where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$ (see Appendix A.3). Then, we can prove the lemma straightforwardly from Theorem 1. □

Remark. It is assumed in the proofs that the functions passed to *liftO* are total. But as a matter of fact, we break the rule ourselves in defining *liftO2* as $\mathit{liftO2}\;p\,x\,y = \mathit{liftO}\,(\lambda[x, y].p\,x\,y)\,[x, y]$. This discrepancy arises as a result of the use of a list parameter to model different arities of a series of functions. The definition of *liftO* is meant to cover a family of functions of types $\mathit{liftO}_n :: (\mathit{PackM}\,\gamma\,\alpha\,\mu, \mathit{Eq}\,\beta) \Rightarrow (\gamma^n \to \beta) \to (\alpha^n \to \mu\,\beta)$ of arity n. If we consider this more refined type, which the instances of *PackM* respect as a matter of fact, the use of *liftO* only concerns total functions. Our proofs can be easily lifted to the refined type. Thus, the partial function fed to *liftO* in the definition of *liftO2* does not pose any problem on correctness.

5. Extensions

In this section, we extend the core idea presented in Section 3 and address some practical concerns.

5.1. Going Generic

So far, we have demonstrated our idea for a specific type, namely *Tree*. It is not difficult to generalize the solution to be datatype-generic.

Actually, this generalization has already been done in the previous work [15, 8]; 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.

$$\mathit{traverse} :: (\mathit{Traversable}\,\kappa, \mathit{Applicative}\,\theta) \Rightarrow (\alpha \to \theta\,\beta) \to \kappa\,\alpha \to \theta\,(\kappa\,\beta)$$

We use *traverse* to define two functions: one collects data from structures (*contents*), and the other decorates structures with given data (*fill*) respectively. [5]

$$
\begin{aligned}
&\mathit{contents} :: \mathit{Traversable}\,\kappa \Rightarrow \kappa\,\alpha \to [\alpha] \\
&\mathit{contents} = \mathit{getConst} \circ \mathit{traverse}\,(\lambda x.\mathit{Const}\,[x]) \\
&\mathit{fill} :: \mathit{Traversable}\,\kappa \Rightarrow \kappa\,\beta \to [\alpha] \to \kappa\,\alpha \\
&\mathit{fill}\,t\,l = \mathit{evalState}\,(\mathit{traverse}\,\mathit{next}\,t)\,l \\
&\quad \textbf{where}\ \mathit{next}\ \_ = \textbf{do}\ (a : x) \leftarrow \mathit{Control.Monad.State.get} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Control.Monad.State.put}\,x \\
&\qquad\qquad\qquad\qquad\ \ \mathit{return}\,a
\end{aligned}
$$

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, *contents* extracts all the elements of a structure as a list, and *fill* replaces the elements of a structure by those of a given list.

In a polymorphic setting where the separation between structure κ and data α is made clear in the type, instances of *Traversable* are straightforward to define, in fact they are systematically derivable [30]. In our case, where transformations are monomorphic, one has to decide where the line is drawn. As a rule of thumb, values that are intended for updating shall be made into atomic data, which is separated from the rest of the structure in the type. For example, in the Links example, we abstract the trees of strings into the polymorphic type *Tree*. As a result, the string labels are subject to updating, but not the tree structure.

The systematically derived instances of *Traversable* satisfy certain laws [29]. We have

$$
\begin{array}{ll}
\textbf{FillContents} & \mathit{fill}\,(\mathit{fmap}\,f\,t)\,(\mathit{contents}\,t) = t \\
\textbf{ContentsFill} & \mathit{contents}\,(\mathit{fill}\,t\,xs) = xs \quad \text{if} \quad \mathit{length}\,xs = \mathit{length}\,(\mathit{contents}\,t)
\end{array}
$$

for any f and t (see Appendix A.4 for a proof), which is needed to establish the correctness of the generic algorithm. Note that every *Traversable* instance is also an instance of *Functor*.

[5] In GHC, the function *contents* is called *toList*, which is defined in `Data.Foldable`. (Every *Traversable* instance is also an instance of *Foldable*). We use the name *contents* to emphasize its role of extracting contents from a structure, following [29].

We then redefine *assignLocs* in this new setting.

```
assignLocs :: Traversable κ ⇒ κ γ → κ (Loc γ)
assignLocs t = fill t (assignLocsList $ contents t)
    where assignLocsList l = zipWith (λx i. Loc x (Just i)) l [1..]
```

Using **FillContents** and **ContentsFill**, we can prove that the conditions we posed on *assignLocs* in Section 3.4 hold (see Appendix A.5 for a proof).

We also redefine *matchViews*, which performs element-wise comparisons after a shape-equality check.

```
matchViews :: (Traversable κ, Eq (κ ()), Eq γ) ⇒ κ (Loc γ) → κ γ → Update γ
matchViews vx v = if fmap ignore vx == fmap ignore v then
                      let lx = contents vx
                          l  = contents v
                      in minimize lx $ makeUpd $ zip lx l
                  else
                      error "Shape Mismatch"
```

Here, *ignore* is a function defined by *ignore* _ = (), 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 :: Eq γ ⇒ [(Loc γ, γ)] → Update γ
makeUpd = foldr f []
  where
    f (Loc x Nothing, y) u | x == y      = u
                           | otherwise = error "Update of Constant"
    f (Loc _ (Just i), y) u =
        case lookup i u of
            Nothing            → (i, y) : u
            Just y' | y == y'  → u
                    | otherwise → error "Inconsistent Update"
```

Again, the conditions we pose on *matchViews* hold (see Appendix A.6).

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

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

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

It is worth noting that there exist GHC language options `DeriveFunctors` and `DeriveTraversables` that allow instances of *Functor* and *Traversable* to be automatically derived, which comes in handy for bidirectionalizing transformations on user-defined datatypes.

5.2. Finer Control of Duplication

It is usually expected that in any reasonable system, duplicates shall be updated consistently to the same value. In theory, there is little controversy about this statement; yet in practice, given two equal values it is less obvious whether they are actual duplicates, or merely being incidentally equal.

5.2.1. Interpretation of Equality

In our system, updates are controlled by source locations: only values with locations can be updated, and those that share the same locations are updated simultaneously. In Section 3.4.2, we have taken a conservative approach in location assignment 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 clear that our choice is not the only meaningful one. In the original work on semantic bidirectionalization [15], when the system is extended to non-fully polymorphic forward functions such as those of type $\forall\alpha.\, Eq\,\alpha \Rightarrow [\alpha] \to [\alpha]$, they use a strategy of considering elements as duplicates, as long 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 presence 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. For example, the version of *assignLocs* assigns locations for `"abba"` as [`'a'`@1, `'b'`@2, `'b'`@2, `'a'`@1]. Consider function *nub* in `Data.List` that removes duplicated elements from a list, where *nub* `"abba"` results in `"ab"`. Let *nubB* be the corresponding backward transformation of *nub* obtained in the above-mentioned manner. Then *nubB* `"abba"` `"cb"` results in `"cbbc"`, as the two `'a'`s are considered duplicates, and the changing of one changes the other.

In our system, this behavior of *nubB* can easily be mimicked by giving an appropriate definition of *assignLocs* that assigns the same locations to equal values. In general, we can express variations of interpretations of duplication by changing the definition of *assignLocs*; for example, for a source `"aaa"`, we can express the situation that the first two `'a'`s are actual duplicates but the third one is accidentally equal to the first two by assigning the locations as [`'a'`@1, `'a'`@1, `'a'`@2]. However, allowing user-defined *assignLocs* unnecessarily exposes implementation details that we wish to hide.

5.2.2. Dynamic Management of Duplication

Our solution is to use an interface function that builds duplication information dynamically. Inspired by the "merge" combinator in [7], we add the following method *eqSync* to $PackM\,\gamma\,\alpha\,\mu$ to turn equal elements into duplicates.

$$eqSync :: Eq\,\gamma \Rightarrow \alpha \to \alpha \to \mu\,Bool$$

Intuitively, *eqSync* is essentially the same as *liftO2* (`==`), but with the additional behavior of recording the two arguments as duplicates when they are equal. As a result, a forward execution creates a witness of equivalence classes in terms of locations (for example, Union-Find tree), and an equivalence relation ($==_e$) can be induced from such a witness e. Then, our backward transformation propagates updates to one location to all those that are equivalent to it.

Concretely, we extend the monad for backward evaluation to include the equivalence-class witnesses. (The name *SW* means the composition of the "State" monad and the "Writer" monad.)

$$\textbf{newtype}\ SW\,\alpha\,\beta = SW\ \{runSW :: Equiv \to ((\beta, [Result\,\alpha]), Equiv)\}$$

The datatype is actually an instance of *Monad* as follows.[6]

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\,(\mathit{SW}\,\alpha)\ \textbf{where} \\
\quad \mathit{return}\,x = \mathit{SW}\ \$\ \lambda e.((x,[]),e) \\
\quad \mathit{SW}\,g \gg\!= f = \mathit{SW}\ \$\ \lambda e.\ \textbf{let}\ ((x,h_1),e') = g\,e \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ((y,h_2),e'') = \mathit{runSW}\ (f\,x)\,e' \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ ((y,h_1 \mathbin{+\!\!+} h_2),e'')
\end{array}
$$

Here, *Equiv* is a datatype to express witnesses of equivalence, which we leave abstract in this paper. We assume the following interface of the abstract type. [7]

$$
\begin{array}{l}
\mathit{empty} :: \mathit{Equiv} \\
\mathit{equate} :: \mathit{Int} \to \mathit{Int} \to \mathit{Equiv} \to \mathit{Equiv} \\
\mathit{equal} :: \mathit{Int} \to \mathit{Int} \to \mathit{Equiv} \to \mathit{Bool}
\end{array}
$$

In the above, *empty* is the empty witness, which means that no two locations are considered equivalent unless they are equal. Function *equate* adds two locations as equivalence to a witness, and *equal* decides the equivalence between two locations with respect to a given witness. Often, we write $i \mathbin{==_e} j$ for $\mathit{equal}\,i\,j\,e$.

In the forward execution, *eqSync* is not different from *liftO2* (==):

$$
\begin{array}{l}
\textbf{instance}\ \mathit{PackM}\,\gamma\,(N\,\gamma)\,I\ \textbf{where} \\
\quad \mathit{liftO}\,p\,x = I\,(p\ \$\ \mathit{map}\,\mathit{runN}\,x) \quad \texttt{--}\ \text{kept unchanged} \\
\quad \mathit{eqSync} = \mathit{liftO2}\ (==)
\end{array}
$$

In the backward execution, *eqSync* records the two compared elements as duplicates if they are equal, by adding their locations to the witness. Note

[6] Actually, using a state monad $\mathit{State}\,([\mathit{Result}\,\alpha],\mathit{Equiv})$ instead of *SW* suffices for our purpose, and then we do not need this explicit *Monad*-instance declaration. We used *SW* here so that we can keep the existing code as much as possible. Also, monad transformers can be used to write $\mathit{WriterT}\,[\mathit{Result}\,\alpha]\,(\mathit{State}\,\mathit{Equiv})$ instead of *SW*. We avoid the use of monad transformers here because it complicates our theoretical development unnecessarily.

[7] For efficiency, we should use a version of *equal* of type $\mathit{Int} \to \mathit{Int} \to \mathit{Equiv} \to (\mathit{Bool},\mathit{Equiv})$ if we use a Union-Find tree to implement *Equiv*. However, we put more weight on presentation simplicity here.

that equal elements don't necessarily have equal locations.

```
instance PackM γ (Loc γ) (SW (Loc γ)) where
  liftO p x = SW $ λe.((p' x, [Result p' x (p' x)]), e)
      where p' = p ∘ map body
  eqSync x y | body x == body y, Just i ← location x, Just j ← location y =
                  SW $ λe. let p [x, y] = body x == body y
                           in ((True, [Result p [x, y] True]), equate i j e)
             | otherwise = liftO2 (==) x y
```

Note that only elements with locations can be synchronized in the above definition. This is not a restriction in real terms, since elements without locations cannot be updated anyway, and enforced by the observation-history check, any other elements that are compared equal with elements without locations are not updatable either.

The definition of *bwd* is adapted to match the new monad, as follows.

```
bwd h = λs v. let sx              = assignLocs s
                  ((vx, hist), equiv) = runSW (h sx) empty
                  upd             = matchViews equiv vx v
              in if checkHist (update equiv upd) hist then
                     fmap (body ∘ update equiv upd) sx
                 else
                     error "Inconsistent History"
```

Additionally, we have to replace the calls to *lookup* in *update* and *matchViews* (more precisely, *makeUpd*) with $lookupBy\ (==_{equiv})$ so that equivalent (not just equal) locations are updated to the same values; this is the reason why *matchViews* and *update* take an extra argument *equiv*.

Let us go back to the example of function *nub* from the beginning of this

subsection, which can be defined for bidirectional execution as follows.

$$
\begin{aligned}
&\mathit{nubM} :: (\mathit{Eq}\,\gamma, \mathit{PackM}\,\gamma\,\alpha\,\mu) \Rightarrow [\alpha] \to \mu\,[\alpha] \\
&\mathit{nubM}\,[] \qquad\quad = \mathit{return}\,[] \\
&\mathit{nubM}\,(x : \mathit{xs}) = \mathbf{do}\ r \leftarrow \mathit{deleteAllM}\,x\,\mathit{xs} \\
&\qquad\qquad\qquad\quad \mathit{xs}' \leftarrow \mathit{nubM}\,r \\
&\qquad\qquad\qquad\quad \mathit{return}\,(x : \mathit{xs}') \\
&\mathit{deleteAllM} :: (\mathit{Eq}\,\gamma, \mathit{PackM}\,\gamma\,\alpha\,\mu) \Rightarrow \alpha \to [\alpha] \to \mu\,[\alpha] \\
&\mathit{deleteAllM}\,x\,[] \qquad\quad = \mathit{return}\,[] \\
&\mathit{deleteAllM}\,x\,(y : \mathit{ys}) = \mathbf{do}\ b \leftarrow \mathit{eqSync}\,x\,y \\
&\qquad\qquad\qquad\qquad\qquad\quad r \leftarrow \mathit{deleteAllM}\,x\,\mathit{ys} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\,(\mathbf{if}\ b\ \mathbf{then}\ r\ \mathbf{else}\ y : r)
\end{aligned}
$$

We use a helper function *deleteAllM* to remove all the elements from xs that are equal to x, and at the same time, through the use of *eqSync*, record that x and the deleted elements are duplicates. As a result, the corresponding backward transformation, *bwd nubM*, behaves in the way described in the beginning of this subsection.

5.2.3. Correctness

We now look at how the adding of *eqSync* affects the correctness discussion in Section 4. One notion that has to be changed is location consistency, where the equality operator = used to compare locations needs to be replaced with $==_e$; we call this version of the definition *location consistency with respect to e*. Consequently, the proof of Lemma 1 has to be adapted to take into account the fact that the witness of equivalence, which location consistency now depends on, changes at runtime. In contrast, Lemmas 2 and 3 can be easily extended to admit *eqSync* because they can only refer to the final value of the witness when the execution finishes (*i.e.*, *equiv* in *bwd*). Note that every $==_{\mathit{equiv}}$ can be emulated by giving an appropriate *assignLocs*.

In the following, we prove the *eqSync*-supported version of Lemma 1. Concretely, we prove the following lemma.

Lemma 4 (Location-Consistency of *vx* with *eqSync*)**.** Suppose that we have $((\mathit{vx}, \_), \mathit{equiv}) = \mathit{runSW}\,(h\,(\mathit{assignLocs}\,s))\,\mathit{empty}$ for a tree s. Then, *vx* is location consistent with respect to *equiv*.

Proof Sketch. The basic idea of the proof follows that of Lemma 1. Let s be a source and E be the set of elements in *assignLocs s*. We say E is location-consistent with respect to e if, for any two elements $x@i$ in $y@j$ (with $i \neq \#$

and $j \neq \#$), $i \texttt{==}_e j$ implies $x = y$. Then, it follows that vx is location consistent with respect to e if all the elements $x@i$ in vx (with $i \neq \#$) are also in E and E is location consistent with respect to e. We write $L_E(e)$ if E is location consistent with respect to e.

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

$$\mathcal{U} \quad = \{(x@i, x@i) \mid i \neq \# \Rightarrow x@i \in E\}$$

$$\mathcal{F}\,\mathcal{R} = \left\{ (SW\ f_1, SW\ f_2) \;\middle|\; \begin{array}{r} \forall e_1, e_2. L_E(e_1) \wedge L_E(e_2) \Rightarrow \\ (x_1, x_2) \in \mathcal{R} \wedge L_E(e_1') \wedge L_E(e_2') \\ \text{where } ((x_i, \_), e_i') = f_i\, e_i \ (i = 1, 2) \end{array} \right\}$$

Then, we can prove that $(\mathcal{L},\mathcal{U},\mathcal{F})$ is a *PackM*-action and the rest of the proof is similar to Lemma 1's. □

6. Putting the System to Use

In this section, we look at how our system can be used in some of the most common applications of bidirectional transformation, namely text processing, XML query and graph transformation. We demonstrate with examples, showing how the examples can be programmed in our system, and how updates can be checked and executed.

6.1. Text Processing

Consider the scenario of processing text represented as strings. The concrete task is to extract words from the text along with the number of occurrences of each word. For example, for text `"a b a a c"`, the expected result is $[(\texttt{"a"}, 3), (\texttt{"b"}, 1), (\texttt{"c"}, 1)]$.

6.1.1. A Datatype for the Source

We start by representing the source data (in this case a string) in a structured format so that the distinction between structure and data is made clear. In this case, we are interested in updating individual words, which suggests the encoding of the text as a list of elements with the element type instantiated as *String*.

Note that the polymorphic nature of the list data type is important here: it provides a clear separation between the structural container and the payload data. This separation is needed for our bidirectionalization technique to apply, as payload data are subject to updating, whereas structures are not.

As a result, a monomorphic data type declaration like the following will not work

$$\textbf{data}\ \mathit{ListS} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{String}\ \mathit{ListS}$$

because one can no longer draw a line between data and structure.

The function that converts the raw text string into the list representation is

$$\mathit{words} :: \mathit{String} \to [\mathit{String}]$$

Here, the type parameter is instantiated.

6.1.2. The Forward Transformation

As a reference implementation, we firstly program the example without considering bidirectionalization.

$$\begin{aligned}
&\mathit{countWords} :: [\mathit{String}] \to [(\mathit{String}, \mathit{Int})]\\
&\mathit{countWords}\ [] \qquad\quad = []\\
&\mathit{countWords}\ (w : ws) = \textbf{let}\ (c, ws') = \mathit{deleteAndCount}\ w\ ws\\
&\qquad\qquad\qquad\qquad\ \ \textbf{in}\ (w, c+1) : \mathit{countWords}\ ws'\\
&\mathit{deleteAndCount} :: \mathit{String} \to [\mathit{String}] \to (\mathit{Int}, [\mathit{String}])\\
&\mathit{deleteAndCount}\ x\ [] \qquad\quad = (0, [])\\
&\mathit{deleteAndCount}\ x\ (w : ws) = \textbf{let}\ (c, ws') = \mathit{deleteAndCount}\ x\ ws\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{in if}\ x == w\ \textbf{then}\ (c+1, ws')\ \textbf{else}\ (c, w : ws')
\end{aligned}$$

We now turn the above functions into ones that have the right types involving PackM, which are suitable for bidirectional execution. This is a rather straightforward rewrite: we convert the definitions into a *Monad*ic style and lift the comparison operator `==`, as follows.

$$\begin{aligned}
&\mathit{countWordsM} :: \mathit{PackM}\ \mathit{String}\ \alpha\ \mu \Rightarrow [\alpha] \to \mu\ [(\alpha, \mathit{Int})]\\
&\mathit{countWordsM}\ [] \qquad\quad = \mathit{return}\ []\\
&\mathit{countWordsM}\ (w : ws) = \textbf{do}\ (c, ws') \leftarrow \mathit{deleteAndCountM}\ w\ ws\\
&\qquad\qquad\qquad\qquad\qquad\ \ r \leftarrow \mathit{countWordsM}\ ws'\\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathit{return}\ \$\ (w, c+1) : r\\
&\mathit{deleteAndCountM} :: \mathit{PackM}\ \mathit{String}\ \alpha\ \mu \Rightarrow \alpha \to [\alpha] \to \mu\ (\mathit{Int}, [\alpha])\\
&\mathit{deleteAndCountM}\ x\ [] \qquad\quad = \mathit{return}\ (0, [])\\
&\mathit{deleteAndCountM}\ x\ (w : ws) =\\
&\quad \textbf{do}\ (c, ws') \leftarrow \mathit{deleteAndCountM}\ x\ ws\\
&\qquad b \leftarrow \mathit{eqSync}\ x\ w\\
&\qquad \mathit{return}\ (\textbf{if}\ b\ \textbf{then}\ (c+1, ws')\ \textbf{else}\ (c, w : ws'))
\end{aligned}$$

```
["Old","MacDonald","had","a","farm","E-I-E-I-O",
 "And","on","the","farm","he","had","a","cow","E-I-E-I-O",
 "With","a","moo","moo","here","and","a","moo","moo","there",
 "Here","a","moo","there","a","moo","everywhere","a","moo","moo",
 "Old","MacDonald","had","a","farm","E-I-E-I-O"]
```

Figure 4: A List of Words

Note that we have chosen to use *eqSync* as the lifted version of `==` to capture the intention of considering equal elements duplicates, and having changes to one propagate to all, as discussed in Section 5.2.

We are almost done here. A small remaining technicality is that the view type $[(\alpha, Int)]$ needs to be an instance of *Traversable* for our generic implementation to work. The standard workaround is to wrap it up into a *newtype*.

$$\textbf{newtype}\ CountList\ \alpha = CountList\ \{runCountList :: [(\alpha, Int)]\}$$

Then, we can simply derive a *Traversable* instance for it using GHC.

Accordingly, the forward and backward transformations of *countWords* perform wrapping and unwrapping of the constructor.

$$\begin{aligned}
&countWordsF :: [String] \to [(String, Int)]\\
&countWordsF\ ws =\\
&\quad runCountList\ \$\ fwd\ (liftM\ CountList \circ countWordsM)\ ws\\
&countWordsB :: [String] \to [(String, Int)] \to [String]\\
&countWordsB\ ws\ cs =\\
&\quad bwd\ (liftM\ CountList \circ countWordsM)\ ws\ (CountList\ cs)
\end{aligned}$$

Here, $liftM :: Monad\ \mu \Rightarrow (\alpha \to \beta) \to (\mu\,\alpha \to \mu\,\beta)$ from `Control.Monad` lifts a function to a monad.

6.1.3. Permitted Updates

We now can try to apply our transformations. Let us consider the list of words shown in Figure 4. Applying *countWordsF* to the list produces the list in Figure 5. The intended updates are to change the spelling of individual words and have all the occurrences updated uniformly in the source. For example, we may update `"cow"` to `"lamb"` and `"moo"` to `"bar"`. But such updates are not supposed to alter the observation made during the forward execution, for example updating `"Here"` to `"here"` will be rejected.

[("Old", 2),	("MacDonald", 2),	("had", 3),	("a", 8),	("farm", 3),
("E-I-E-I-O", 3),	("And", 1),	("on", 1),	("that", 1),	("he", 1),
("cow", 1),	("With", 1),	("moo", 8),	("here", 1),	("and", 1),
("there", 2),	("Here", 1),	("everywhere", 1)]		

Figure 5: Result of Applying *countWordsF* to the List in Figure 4

Structure changes are also ruled out as expected: it is made clear in the definition of *CountList* that the list structure together with the integer counts is considered to be the structure here.

6.2. XML Query

Next, 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.

6.2.1. A Datatype for XML

As always, we firstly need to decide on a datatype to describe the source data. Here, we use the rose-tree datatype defined in Section 3, and instantiate the element type to the following labels.

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

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

$$\begin{aligned} &Node\ (E\ \texttt{"book"})\ [Node\ (A\ \texttt{"year"})\ [Node\ (T\ \texttt{"1994"})\ []], \\ &\qquad Node\ (E\ \texttt{"title"})\ [Node\ (T\ \texttt{"Text"})\ []]] \end{aligned}$$

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

$$label\ (Node\ l\ \_) = l$$

6.2.2. Programming the Forward Transformation

We have seen in the previous subsection the rather straightforward adaptation of an ordinary program into one ready for bidirectional execution. Here, we do not try to repeat the process; instead we directly implement Q1 as a function of type $\forall\alpha.\forall\mu.PackM\ L\,\alpha\,\mu \Rightarrow Tree\,\alpha \to \mu\,(Tree\,\alpha)$. A standard way to write XML transformations in a functional language is to use "filters" [31]. In [31], the filters (if we simplify and customize them to our rose trees) are of type $Tree\,L \to [Tree\,L]$, which will be made polymorphic in our setting as

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

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

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

which has an implementation for the method $lift :: \mu\,\alpha \to ListT\,\mu\,\alpha$ 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. Note that μ of $ListT\,\mu$ must be commutative, and otherwise, $ListT\,\mu$ is not necessarily a monad. In the following discussion, μ of $ListT\,\mu$ will be instantiated by the monads I and $W\,(Loc\,\gamma)$ (or, $SW\,(Loc\,\gamma)$), where they all are commutative because we do not care the order of observation histories (and that of equivalence witnesses).

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

$$\begin{aligned} &keep :: PackM\ L\,\alpha\,\mu \Rightarrow Tree\,\alpha \to ListT\,\mu\,(Tree\,\alpha)\\ &keep = return \end{aligned}$$

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

$$\begin{aligned} &children :: PackM\ L\,\alpha\,\mu \Rightarrow Tree\,\alpha \to ListT\,\mu\,(Tree\,\alpha)\\ &children\,(Node\ \_\ ts) = ListT\,(return\ ts) \end{aligned}$$

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

$$\begin{aligned} &ofLabel :: PackM\ L\,\alpha\,\mu \Rightarrow \alpha \to Tree\,\alpha \to ListT\,\mu\,(Tree\,\alpha)\\ &ofLabel\,l\,t = \mathbf{do}\ guardM\ \$\ lift\ \$\ liftO2\ (==)\ (label\,t)\,l\\ &\qquad\qquad\quad return\,t \end{aligned}$$

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 *False* and does nothing otherwise.

$$\begin{aligned}&\mathit{guardM} :: \mathit{MonadPlus}\,\kappa \Rightarrow \kappa\,\mathit{Bool} \to \kappa\,()\\&\mathit{guardM}\;x = x \gg\!= (\lambda b.\mathbf{if}\;b\;\mathbf{then}\;\mathit{return}\,()\;\mathbf{else}\;\mathit{mzero})\end{aligned}$$

Filters are composable [31]. For example, with the following operator `/>`

$$\begin{aligned}(\texttt{/>}) :: \mathit{PackM}\;L\,\alpha\,\mu \Rightarrow (\mathit{Tree}\,\alpha &\to \mathit{ListT}\,\mu\,(\mathit{Tree}\,\alpha))\\ &\to (\mathit{Tree}\,\alpha \to \mathit{ListT}\,\mu\,(\mathit{Tree}\,\alpha))\\ &\to (\mathit{Tree}\,\alpha \to \mathit{ListT}\,\mu\,(\mathit{Tree}\,\alpha))\\ f\;\texttt{/>}\;g &= f \gg\!\Rightarrow \mathit{children} \gg\!\Rightarrow g\end{aligned}$$

where $(\gg\!\Rightarrow)$ is Kleisli-composition operator in `Control.Monad` defined by $(f \gg\!\Rightarrow g)\,x = f\,x \gg\!= g$, 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.

Now we are ready to implement Q1 in Figure 6. The code is mostly declarative though complicated; the complication largely comes from the encoding of XML queries in a functional language, not from the bidirectionalization effort. 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.

6.2.3. 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 document 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.

```
q1 :: PackM L α μ ⇒ Tree α → μ (Tree α)
q1 t = pick $ do bs ← gather $ (keep /> (tag "book" >=> h)) t
                 return $ Node (new $ E "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 (T l1) (T 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 6: Query Q1 in Our Framework

- 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.

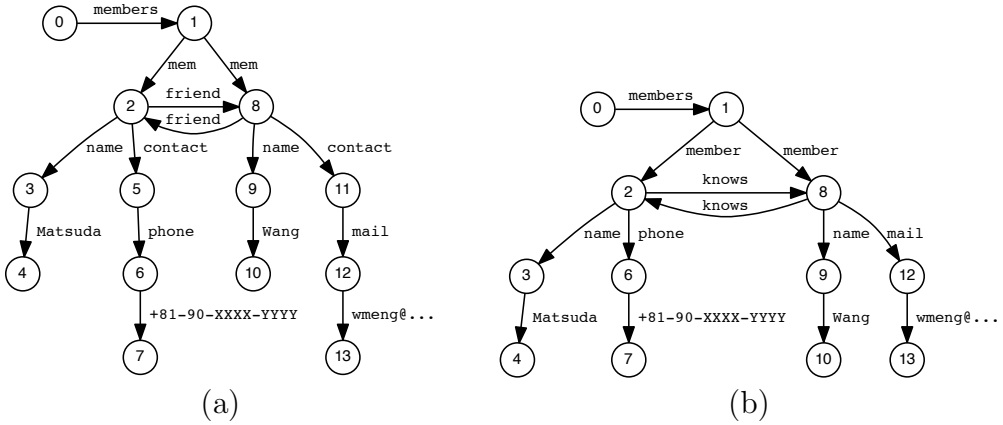

Figure 7: A source and its corresponding view graphs.

6.3. Graph Transformation

The last example we look at is graph transformation, which is at heart of many model-based software engineering applications.

We take an example from [32], which is a friend list in an edge-labeled graph format (see Figure 7-(a)). And, suppose that we want to apply transformation that renames `mem` to `member` and `friend` to `knows`, and flattens `contact` (see the resulting graph in Figure 7-(b)).

6.3.1. A Datatype for Graph

There are many ways to represent and manipulate graphs in functional languages [33, 34, 35, 36, 37]. But since our focus in the paper is bidirectionalization, we choose the simplest representation, namely a list of edges, as below.

$$\textbf{newtype}\ \mathit{Graph}\,\alpha = \mathit{Graph}\,[(\mathit{Int}, \alpha, \mathit{Int})]$$

Here, we assume that nodes are represented by integers. The representation makes clear that the edge labels are the intended update targets. As an example, the graph in Figure 7-(a) can be written as $\mathit{Graph}\,[(0, \texttt{"members"}, 1), (1, \texttt{"mem"}, 2), (1, \texttt{"mem"}, 8), (2, \texttt{"name"}, 3), (2, \texttt{"contact"}, 5), (2, \texttt{"friend"}, 8), (3, \texttt{"Matsuda"}, 4), (5, \texttt{"phone"}, 6), (6, \texttt{"+81-90-XXXX-YYYY"}, 7), (8, \texttt{"friend"}, 2), (8, \texttt{"name"}, 9), (9, \texttt{"Wang"}, 10), (8, \texttt{"contact"}, 11), (11, \texttt{"mail"}, 12), (12, \texttt{"wmeng@..."}, 13)]$. We assume that a graph need not be *simple*; *i.e.*, there can be multiple edges that have the same source, destination and label. Also, we consider the edges are unordered.

6.3.2. Programming the Forward Transformation

Graph transformation in general is an area of research by itself [38, 33, 39, 40, 34, 35, 36, 37]. But for our simple example, we can program it rather easily as follows, with the understanding that a different implementation will also work as long as it has the right type.

```
m2m :: PackM String α μ ⇒ Graph α → μ (Graph α)
m2m = rename "mem" "member" >=>
      rename "friend" "knows" >=>
      contract "contact"
```

Here, *rename* is a renaming function defined by using *mapM* as below.

```
rename :: PackM String α μ ⇒ String → String → Graph α → μ (Graph α)
rename x y (Graph es) =
  do r ← mapM (λ(s, e, d).
                 do b ← liftO2 (==) e (new x)
                    return (if b then (s, new y, d) else (s, e, d))) es
     return $ Graph r
```

The definition of *contract*, which contracts edges of a given label, is a bit more complicated. The need of handling cycles and sharings in graphs plays a part here. Since nodes connected by the contracted edge will be unified, and such contracted edges may be cyclic or may share nodes, we have to compute the reflexive symmetric transitive closure to know which nodes will be unified by the contraction. This problem is unique to graphs as simple hoisting of nodes is sufficient for trees.

```
contract :: PackM String α μ ⇒ String → Graph α → μ (Graph α)
contract x (Graph es) =
  do let nodes = nub $ concatMap (λ(s, _, d).[s, d]) es
     conts ← concatMapM (λ(s, e, d).
                           do b ← liftO2 (==) e (new x)
                              return (if b then [(s, d)] else [])) es
     let rstCls = mkSymTransClosure (conts ++ [(z, z) | z ← nodes])
     let repr z = minimum [y | (z', y) ∈ rstCls, z' == z]
     es' ← concatMapM (λ(s, e, d).
                 do b ← liftO2 (==) e (new x)
                    return (if b then [] else [(repr s, e, repr d)])) es
     return $ Graph es'
```

In the definition, we reuse the function *concatMapM* from Section 3.3. Intuitively function *contract* firstly computes the set of the nodes that will be unified by the removal of the x-labeled edges (*repr* is the computed unifier), and then actually removes the x-labeled edges with the node unification by *repr*. The function *mkSymTransClosure* computes the symmetric transitive closure of a given relation represented by a list of pairs. We omit its definition for brevity, as it is a pure function which is not affected by the bidirectionalization. Also, note that the execution of *contract* may create duplicated edges with the same starting/ending nodes and labels. This duplication may be removed by using a *nub* like operation, as discussed in Section 5.2. We omit this step for simplicity.

As a remark, the behavior of *contract* itself is controversial. For example, in a certain model of graphs in which we identify graphs with infinite trees, the *contract* operation may convert a finite graph to an infinite one and does not terminate in such a case without careful treatment [41, 42]. Our framework is applicable even to such model of graphs, as long as the parametricity and the totality assumption, including the termination of a graph transformation, are kept (Section 4).

6.3.3. Permitted Updates

Since *m2m* has the right type, we bidirectionalize it directly with *fwd* and *bwd*. As mentioned previously, the only permitted updates are to the edge labels.

Consider the view in Figure 7-(b). Then, we have that

- Labels `member` and `knows` are not updatable because they are introduced by *m2m*.
- For other labels, updates that change a label to any values other than `mem`, `friend` and `contact` are permitted.

7. Discussion on 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 can either be variants of the definitional properties, or additional optional ones.

7.1. Desirable Laws

Two of the desirable but optional laws are undoability and composability [12, 1, 4, 5], which are variants of PutPut [7].[8]

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 [43, 4]: for example, composability allows us to schedule updates on the view separately from those on the source [43]. 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 structure of the source and the observation history serve as the constant complement in our framework.

The desirable laws are sometimes considered too restrictive [44, 7] because a framework that supports structure 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 [16, 24], where semantic bidirectionalization is combined with another bidirectionalization technique [12] aiming at structure 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 [43, 5, 45].

There is another law discussed by Hegner [4, 5] which, with the four laws, ensures that information not present in the view cannot be accessed.

Uniformity $put\, s\, v = s' \Rightarrow (\forall s''.get\, s'' = get\, s \Rightarrow \exists s'''.\ put\, s''\, v = s''')$

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

$$f\, [x, y] = liftO2\ (\texttt{==})\, x\, y \gg return\, [x]$$

[8] If *put* is total, composability coincides with PutPut [7] and implies undoability.

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

7.2. Weaker Consistency Laws for Duplication

The handling of duplication as seen in Section 5.2, represented by function *nub*, focuses on the scenario that duplicates of source elements are removed by the forward transformation, which leaves no duplicates in the view. Things become more complicated when this assumption does not hold because discrepancies arise when some but not all duplicated elements are updated.

It is potentially useful to allow such updates which leads to temporary discrepancies among duplicates. For example, one may want to change one element of a duplication group, and propagate the update to all [11, 13]. We have seen this behavior with source duplication, but also for views here. In such a situation, the consistency law is deliberately breaking, as the updated source is supposed to produce the "corrected" view in which the temporary discrepancies is resolved. In response, weaker versions of the consistency law have been proposed [11, 13, 10].

Weak-PutGet [10]	$put\ s\ v = s' \Rightarrow put\ s\ (get\ s') = s'$
PutGetPut [11, 13]	$put\ s\ v = s' \Rightarrow put\ s'\ (get\ s') = s'$
GetPutGet [11, 13]	$put\ s\ (get\ s) = s' \Rightarrow get\ s' = get\ s$

Note that **PutGetPut** is a corollary of the acceptability law; and it is only meaningful when the acceptability law is weakened too, such as **GetPutGet** above. The above laws are not closed under function composition (see Appendix B), and thus do not work well with combinator-based techniques for constructing bidirectional transformations. Intuitively, this is because even if we know that (x_1, x_2) and (x_2, x_3) are duplicates, it is difficult to know by local reasoning that (x_1, x_3) are also duplicates.

Our technique based on bidirectionalization does not suffer the same difficulty because duplication handling is done globally by *assignLocs* (and *eqSync*). The only change required is to *matchViews*. A new version of *matchViews* relaxes the requirement that updated duplicates in the view must all be changed to the same value. Instead it tries to identify changes and propagate the changes to the unchanged members of duplication groups. This is done by replacing the line in the definition of *matchViews*

$$minimize\ lx\ \$\ makeUpd\ \$\ zip\ lx\ l$$

with

$$makeUpd\ \$\ filter\ hasChanged\ \$\ zip\ lx\ l$$

Here, *hasChanged* is defined as follows.

$$hasChanged\,(Loc\,x\,\_, y) = not\,(x == y)$$

The excluding of unchanged elements in constructing the update avoids some conflicts. For example, consider a function $dup\,[x] = return\,[x, x]$ and its derived backward function $dupB = bwd\ dup$. Executing both $dupB\,[0]\,[1, 0]$ and $dupB\,[0]\,[0, 1]$ result in $[1]$ (instead of an error previously), because the changes from 0 to 1 is propagated to the unchanged duplicates. On the other hand, executing $dupB\,[0]\,[1, 2]$ still fails because more than one elements of a duplication group are updated to different values. It is not difficult to see that this version of our system satisfies the weakened laws above.

8. Related Work

Another way of bidirectionalizing programs is through syntactically transforming forward-function definitions [12], termed as syntactic bidirectionalization in [15], in contrast to the semantic approach that does not inspect the program definitions. The syntactic approach has the advantage of deriving more efficient and effective (in the sense of allowing certain shape updates) backward transformations. For example, the technique in [12] can derive a backward *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, 7, 11, 13, 14, 17], and the programmer builds a bidirectional transformation by combining smaller ones with special combinators. Some combinator languages can be implemented as libraries [11, 13], which is rather light-weight, while some languages [6, 9, 7, 17] 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 [6, 14, 17], 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 one does have to program in an unusual style, 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, 12, 3]. The lens framework [7] provides a combinator *ccond* that performs conditional branching in the forward execution, and in the backward execution, the recorded history prohibits updates that may cause the execution to take a different branch. This treatment of branching is more explicit in [12], where branching information is recorded in a complement, which is kept constant to guarantee the bidirectional laws.

Fegaras [3] extends Voigtländer [15]'s original technique for updating XML views over relational databases. In addition to locations, his framework also records the "join"ing structure of a query at runtime. The recorded information is used to reflect changes on one copy of duplicated view elements caused by joins to establish the consistency law. For example, a change on such a copy can be handled by changing data on key columns used in the joins and then inserting new tuples. His framework also supports insertions and deletions on a view, leveraging on the fact that queries are written in a specific query language, and the sources are relational data.

Both ours and the original work on semantic bidirectionalization [15] focus on in-place updates. Reflecting in-place updates is a non-trivial problem given complex forward transformations [10]. This limitation of updates can be relaxed to some extent by combining semantic and syntactic bidirectionalization [16, 24]. 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. It is obvious that some traversals in our implementation may be fused and more involved data structures can be used instead of lists. Perhaps more importantly, we expect semantic bidirectionalization in general, to fit the requirement for incremental computation [18] of updates because it essentially maps update operations on the view to those on the source.

9. Conclusion

In this paper, we have extended semantic bidirectionalization to handle monomorphic transformations, by programming them in a polymorphic way through a type class *PackM*. Specifically, we have replaced 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 have proved that the transformations produced by our bidirectionalization system satisfy the bidirectional properties, *i.e.*, the acceptability and consistency laws. The practicality of our system has been demonstrated by three case studies: text processing, XML query, and graph transformation.

In the future, we plan to extend our work with more operations on abstracted values. For example string concatenation is used by many XML transformations, but is at the moment not a method in *PackM*. Moreover, we plan to support structural changes on views (*i.e.*, insertion or deletion of subtrees), which is currently prohibited. One possible solution would be to abstract containers such as *Tree* in addition to elements in the making of the polymorphic functions. Alternatively, we could try to combine our semantic bidirectionalization with other bidirectional transformation techniques on "shapes" [24]. On the practical side, it is useful is to support abstraction of multiple element types. For example, we would like to update both *Int* and *String* values in a view of type $[(Int, String)]$. Theoretically, it is straightforward to extend our technique to such cases by preparing n different variants of *fill* and *contents* for containers with n element types. However practically this also means n different *fwd* and *bwd* functions, which would clutter the library interface.

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. We thank Shin-Cheng Mu for the counterexample in Appendix B, which is proposed by him originally for **PutGetPut** and **GetPutGet**. We also thank Kazuyuki Asada, Soichiro Hidaka and Zhenjiang Hu for their comments and discussions with us for the counterexample for **Weak-PutGet**

This work was partially supported by JSPS KAKENHI Grant Number 24700020, the Grand-Challenging Project on the "Linguistic Foundation for Bidirectional Model Transformation" of the National Institute of Informatics, and Swedish Foundation for Strategic Research (SSF), project RAWFP.

References

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

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

[3] L. Fegaras, Propagating updates through XML views using lineage tracing, in: F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, V. J. Tsotras (Eds.), ICDE, IEEE, 2010, pp. 309–320.

[4] S. J. Hegner, Foundations of canonical update support for closed database views, in: S. Abiteboul, P. C. Kanellakis (Eds.), ICDT, volume 470 of *Lecture Notes in Computer Science*, Springer, 1990, pp. 422–436.

[5] S. J. Hegner, An order-based theory of updates for closed database views, Ann. Math. Artif. Intell. 40 (2004) 63–125.

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

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

[8] N. Foster, K. Matsuda, J. Voigtländer, Three complementary approaches to bidirectional programming, in: J. Gibbons (Ed.), SSGIP, volume 7470 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 1–46.

[9] J. N. Foster, A. Pilkiewicz, B. C. Pierce, Quotient lenses, in: J. Hook, P. Thiemann (Eds.), ICFP, ACM, 2008, pp. 383–396.

[10] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, Bidirectionalizing graph transformations, in: [46], 2010, pp. 205–216.

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

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

[13] S.-C. Mu, Z. Hu, M. Takeichi, An algebraic approach to bi-directional updating, in: W.-N. Chin (Ed.), APLAS, volume 3302 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 2–20.

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

[15] J. Voigtländer, Bidirectionalization for free! (pearl), in: Z. Shao, B. C. Pierce (Eds.), POPL, ACM, 2009, pp. 165–176.

[16] J. Voigtländer, Z. Hu, K. Matsuda, M. Wang, Combining syntactic and semantic bidirectionalization, in: [46], 2010, pp. 181–192.

[17] M. Wang, J. Gibbons, K. Matsuda, Z. Hu, Gradual refinement: Blending pattern matching with data abstraction, in: C. Bolduc, J. Desharnais, B. Ktari (Eds.), MPC, volume 6120 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 397–425.

[18] M. Wang, J. Gibbons, N. Wu, Incremental updates for efficient bidirectional transformations, in: M. M. T. Chakravarty, Z. Hu, O. Danvy (Eds.), ICFP, ACM, 2011, pp. 392–403.

[19] M. Wang, S. Najd, Semantic bidirectionalization revisited, in: W.-N. Chin, J. Hage (Eds.), PEPM, ACM, 2014, pp. 51–62.

[20] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. F. Terwilliger, Bidirectional transformations: A cross-discipline perspective, in: R. F. Paige (Ed.), ICMT, volume 5563 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 260–283.

[21] M. Wang, J. Gibbons, K. Matsuda, Z. Hu, Refactoring pattern matching, Sci. Comput. Program. 78 (2013) 2216–2242.

[22] P. Wadler, Theorems for free!, in: FPCA, 1989, pp. 347–359.

[23] J. C. Reynolds, Types, abstraction and parametric polymorphism, in: R. Mason (Ed.), Information Processing, Elsevier Science Publishers B.V. (North-Holland), 1983, pp. 513–523.

[24] J. Voigtländer, Z. Hu, K. Matsuda, M. Wang, Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins, J. Funct. Program. 23 (2013) 515–551.

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

[26] K. Matsuda, M. Wang, Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem, in: [47], 2013, pp. 297–308.

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

[28] J.-P. Bernardy, P. Jansson, R. Paterson, Proofs for free - parametricity for dependent types, J. Funct. Program. 22 (2012) 107–152.

[29] R. S. Bird, J. Gibbons, S. Mehner, J. Voigtländer, T. Schrijvers, Understanding idiomatic traversals backwards and forwards, in: C. chieh Shan (Ed.), Haskell, ACM, 2013, pp. 25–36.

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

[31] M. Wallace, C. Runciman, Haskell and XML: Generic combinators or type-based translation?, in: D. Rémi, P. Lee (Eds.), ICFP, ACM, 1999, pp. 148–159.

[32] K. Inaba, S. Hidaka, Z. Hu, H. Kato, K. Nakano, Graph-transformation verification using monadic second-order logic, in: P. Schneider-Kamp, M. Hanus (Eds.), PPDP, ACM, 2011, pp. 17–28.

[33] B. C. d. S. Oliveira, W. R. Cook, Functional programming with structured graphs, in: P. Thiemann, R. B. Findler (Eds.), ICFP, ACM, 2012, pp. 77–88.

[34] M. Erwig, Graph algorithms = iteration + data structures? the structure of graph algorithms and a corresponding style of programming, in: E. W. Mayr (Ed.), WG, volume 657 of *Lecture Notes in Computer Science*, Springer, 1992, pp. 277–292.

[35] M. Erwig, Inductive graphs and functional graph algorithms, J. Funct. Program. 11 (2001) 467–492.

[36] T. Johnsson, Efficient graph algorithms using lazy monolithic arrays, J. Funct. Program. 8 (1998) 323–333.

[37] D. J. King, J. Launchbury, Structuring depth-first search algorithms in haskell, in: R. K. Cytron, P. Lee (Eds.), POPL, ACM Press, 1995, pp. 344–354.

[38] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener, The Lorel query language for semistructured data, Int. J. on Digital Libraries 1 (1997) 68–88.

[39] P. Buneman, M. F. Fernandez, D. Suciu, UnQL: A query language and algebra for semistructured data based on structural recursion, VLDB J. 9 (2000) 76–110.

[40] M. P. Consens, A. O. Mendelzon, GraphLog: a visual formalism for real life recursion, in: D. J. Rosenkrantz, Y. Sagiv (Eds.), PODS, ACM Press, 1990, pp. 404–416.

[41] K. Asada, S. Hidaka, H. Kato, Z. Hu, K. Nakano, A parameterized graph transformation calculus for finite graphs with monadic branches, in: [47], 2013, pp. 73–84.

[42] S. Hidaka, K. Asada, Z. Hu, H. Kato, K. Nakano, Structural recursion for querying ordered graphs, in: G. Morrisett, T. Uustalu (Eds.), ICFP, ACM, 2013, pp. 305–318.

[43] G. Gottlob, P. Paolini, R. Zicari, Properties and update semantics of consistent views, ACM Trans. Database Syst. 13 (1988) 486–524.

[44] A. M. Keller, Comments on Bancilhon and Spyratos' "update semantics and relational views", ACM Trans. Database Syst. 12 (1987) 521–523.

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

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

[47] R. Peña, T. Schrijvers (Eds.), 15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013, ACM, 2013.

Appendix A. Proofs

Appendix A.1. Supplement to Proof Sketch of Lemma 1

We will prove that, for $\mathcal{U} :: \mathit{Loc}\,\gamma \leftrightarrow \mathit{Loc}\,\gamma$ and $\mathcal{F} :: W\,(\mathit{Loc}\,\gamma) \leftrightarrow W\,(\mathit{Loc}\,\gamma)$ defined by

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

and for the diagonal relation $\mathcal{L} :: \gamma \leftrightarrow \gamma$, $(\mathcal{L}, \mathcal{U}, \mathcal{F})$ is a *PackM*-action.

We write τ for $\mathit{Loc}\,\gamma$ and κ for $W\,(\mathit{Loc}\,\gamma)$.

Case: return. We will prove $(\mathit{return}_\kappa, \mathit{return}_\kappa) \in \forall \mathcal{R}.\, \mathcal{R} \to \mathcal{F}\,\mathcal{R}$.

Let $\mathcal{R}$ be a relation and (x, y) be a pair such that $(x, y) \in \mathcal{R}$. In this case, we have $\mathit{return}_\kappa\, x = W\,(x, [])$ and $\mathit{return}_\kappa\, y = W\,(y, [])$. Thus, by the definition of $\mathcal{F}$, we have $(\mathit{return}_\kappa\, x, \mathit{return}_\kappa\, y) \in \mathcal{F}\,\mathcal{R}$.

Case: $(\gg\!=)$. We will prove $((\gg\!=)_\kappa, (\gg\!=)_\kappa) \in \forall \mathcal{R}. \forall \mathcal{S}.\, \mathcal{F}\,\mathcal{R} \to (\mathcal{R} \to \mathcal{F}\,\mathcal{S}) \to \mathcal{F}\,\mathcal{S}$.

Let $\mathcal{R}$ and $\mathcal{S}$ be relations. Let (m_1, m_2) be a pair such that $(m_1, m_2) \in \mathcal{F}\,\mathcal{R}$, and (f_1, f_2) be a pair such that $(f_1, f_2) \in \mathcal{R} \to \mathcal{F}\,\mathcal{S}$. We will show $(m_1 \gg\!=_\kappa f_1, m_2 \gg\!=_\kappa f_2) \in \mathcal{F}\,\mathcal{S}$. Unfolding the definition of $\gg\!=$, we rewrite $(m_1 \gg\!=_\kappa f_1, m_2 \gg\!=_\kappa f_2)$ as $(W\,(y_1, h_{11} \mathbin{+\!\!+} h_{12}), W\,(y_2, h_{21} \mathbin{+\!\!+} h_{22}))$ where $W\,(x_1, h_{11}) = m_1$, $W\,(y_1, h_{12}) = f_1\, x_1$, $W\,(x_2, h_{21}) = m_2$ and $W\,(y_2, h_{22}) = f_2\, x_2$. By $(m_1, m_2) \in \mathcal{F}\,\mathcal{R}$, we have $(x_1, x_2) \in \mathcal{R}$. Thus, by $(f_1, f_2) \in \mathcal{R} \to \mathcal{F}\,\mathcal{S}$, we have $(W\,(y_1, h_{12}), W\,(y_2, h_{22})) \in \mathcal{F}\,\mathcal{S}$. As a consequence, by the definition of $\mathcal{F}$, we have $(W\,(y_1, h_{11} \mathbin{+\!\!+} h_{12}), W\,(y_2, h_{21} \mathbin{+\!\!+} h_{22})) \in \mathcal{F}\,\mathcal{S}$.

Case: new. We will prove $(new_{\gamma,\tau}, new_{\gamma,\tau}) \in \mathcal{L} \to \mathcal{U}$. Since *new* only introduces a location-aware element of the form $x@\#$, the statement is trivially true by the definition of $\mathcal{U}$.

Case: liftO. We will prove $(liftO_{\gamma,\tau,\kappa,\beta_1}, liftO_{\gamma,\tau,\kappa,\beta_2}) \in ([\mathcal{L}] \to \mathcal{S}) \to [\mathcal{U}] \to \mathcal{F}\,\mathcal{S}$ for all $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ such that $((\texttt{==})_{\beta_1}, (\texttt{==})_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$.

Let $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ be a relation satisfying $((\texttt{==})_{\beta_1}, (\texttt{==})_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$. Let (p_1, p_2) be a pair of functions such that $(p_1, p_2) \in [\mathcal{L}] \to \mathcal{S}$ and (x_1, x_2) be a pair of lists such that $(x_1, x_2) \in [\mathcal{U}]$. Then, we can write $\mathit{liftO}\,p_i\,x_i$ as $W\,(p'_i\,x_i, \_)$ where $p'_i = p_i \circ \mathit{map}\,\mathit{body}$ $(i = 1, 2)$. It is worth noting that $(x_1, x_2) \in [\mathcal{U}]$ implies $(\mathit{map}\,\mathit{body}\,x_1, \mathit{map}\,\mathit{body}\,x_2) \in [\mathcal{L}]$, and thus $(p'_1\,x_1, p'_2\,x_2) \in \mathcal{S}$. Then, we have $(W\,(p'_1\,x_1, \_), W\,(p'_2\,x_2, \_)) \in \mathcal{F}\,\mathcal{S}$. □

Appendix A.2. Supplement to Proof Sketch of Lemma 2

We will prove that for $\mathcal{U} :: N\,\gamma \leftrightarrow \mathit{Loc}\,\gamma$ and $\mathcal{F} :: I \leftrightarrow W\,(\mathit{Loc}\,\gamma)$ defined by

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

$(\mathcal{L}, \mathcal{U}, \mathcal{F})$ is a *PackM*-action where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$.

We write τ_1 and τ_2 for $N\,\gamma$ and $\mathit{Loc}\,\gamma$, respectively, and κ_1 and κ_2 for I and $W\,(\mathit{Loc}\,\gamma)$.

Case: return. We will prove $(\mathit{return}_{\kappa_1}, \mathit{return}_{\kappa_2}) \in \forall\mathcal{R}.\,\mathcal{R} \to \mathcal{F}\,\mathcal{R}$.

Let $\mathcal{R}$ be a relation and (x_1, x_2) be a pair such that $(x_1, x_2) \in \mathcal{R}$. In this case, we have $\mathit{return}_{\kappa_1}\,x_1 = I\,x_1$ and $\mathit{return}_{\kappa_2}\,x_2 = W\,(x_2, [])$. Since $\mathit{checkHist}\,u\,[] = \mathit{True}$ holds for any u, by the definition of $\mathcal{F}$, we have $(\mathit{return}_{\kappa_1}\,x_1, \mathit{return}_{\kappa_2}\,x_2) \in \mathcal{F}\,\mathcal{R}$.

Case: $(\gg\!=)$. We will prove $((\gg\!=)_{\kappa_1}, (\gg\!=)_{\kappa_2}) \in \forall\mathcal{R}.\forall\mathcal{S}.\,\mathcal{F}\,\mathcal{R} \to (\mathcal{R} \to \mathcal{F}\,\mathcal{S}) \to \mathcal{F}\,\mathcal{S}$.

Let $\mathcal{R}$ and $\mathcal{S}$ be relations. Let (m_1, m_2) be a pair such that $(m_1, m_2) \in \mathcal{F}\,\mathcal{R}$, and (f_1, f_2) be a pair such that $(f_1, f_2) \in \mathcal{R} \to \mathcal{F}\,\mathcal{S}$. We will show $(m_1 \gg\!=_{\kappa_1} f_1, m_2 \gg\!=_{\kappa_2} f_2) \in \mathcal{F}\,\mathcal{S}$.

By unfolding the definition of $(\gg\!=)_{\kappa_1}$, we can rewrite $(m_1 \gg\!=_{\kappa_1} f_1)$ as $I\,y_1$ where $I\,x_1 = m_1$ and $I\,y_1 = f_1\,x_1$. Similarly, by unfolding the definition of $(\gg\!=)_{\kappa_2}$, we can rewrite $(m_2 \gg\!=_{\kappa_2} f_2)$ as $W\,(y_2, h_{21} +\!\!+ h_{22})$ where $W\,(x_2, h_{21}) = m_2$ and $W\,(y_2, h_{22}) = f_2\,x_2$. By $(m_1, m_2) \in \mathcal{F}\,\mathcal{R}$ and the definition of $\mathcal{F}$, we have $(x_1, x_2) \in \mathcal{R}$. Thus, we have $(I\,y_1, W\,(y_2, h_{22})) \in \mathcal{F}\,\mathcal{S}$. By the definition

of $\mathcal{F}$, we have $\mathit{checkHist}\ \mathit{id}\ h_{2i} = \mathit{True}$ for each $i = 1, 2$. Thus, according to the definition of $\mathit{checkHist}$, we can conclude $\mathit{checkHist}\ \mathit{id}\ (h_{21} \mathbin{+\!\!+} h_{22}) = \mathit{True}$. This implies $(I\ y_1, W\ (y_2, h_{21} \mathbin{+\!\!+} h_{22})) \in \mathcal{F}\,\mathcal{S}$.

Case: new. We will prove $(\mathit{new}_{\gamma,\tau_1}, \mathit{new}_{\gamma,\tau_2}) \in \mathcal{L} \to \mathcal{U}$. Let $(x_1, x_2) \in \mathcal{L}$. Since $\mathcal{L}$ is diagonal, we have $x_1 = x_2$. Then, we have $\mathit{new}_{\gamma,\tau_1}\ x_1 = N\ x_1$ and $\mathit{new}_{\gamma,\tau_2}\ x_2 = \mathit{Loc}\ x_2\ \mathit{Nothing}$. Since $x_1 = x_2$, we have $(\mathit{new}_{\gamma,\tau_1}\ x_1, \mathit{new}_{\gamma,\tau_2}\ x_2) \in \mathcal{U}$.

Case: liftO. We will prove $(\mathit{liftO}_{\gamma,\tau_1,\kappa_1,\beta_1}, \mathit{liftO}_{\gamma,\tau_2,\kappa_2,\beta_2}) \in ([\mathcal{L}] \to \mathcal{S}) \to [\mathcal{U}] \to \mathcal{F}\,\mathcal{S}$ for all $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ such that $((\texttt{==})_{\beta_1}, (\texttt{==})_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$.

Let $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ be a relation satisfying $((\texttt{==})_{\beta_1}, (\texttt{==})_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$. Let (p_1, p_2) be a pair of functions such that $(p_1, p_2) \in [\mathcal{L}] \to \mathcal{S}$ and (x_1, x_2) be a pair of lists such that $(x_1, x_2) \in [\mathcal{U}]$. By definition, we have $\mathit{liftO}_{\gamma,\tau_1,\kappa_1,\beta_1}\ p_1\ x_1 = I\ (p_1\ \$\ \mathit{map}\ \mathit{runN}\ x_1)$. Also, we have $\mathit{liftO}_{\gamma,\tau_2,\kappa_2,\beta_2}\ p_2\ x_2 = W\ (p_2'\ x_2, [\mathit{Result}\ p_2'\ x_2\ (p_2'\ x_2)])$ where $p_2' = p_2 \circ \mathit{map}\ \mathit{body}$. Since $(x_1, x_2) \in [\mathcal{U}]$, we have $(\mathit{map}\ \mathit{runN}\ x_1, \mathit{map}\ \mathit{body}\ x_2) \in [\mathcal{L}]$, and thus $(p_1 \$ \mathit{map}\ \mathit{runN}\ x_1, p_2'\ x_2) \in \mathcal{S}$. By the definition of $\mathit{checkHist}$, we have $\mathit{checkHist}\ \mathit{id}\ [\mathit{Result}\ p_2'\ x_2\ (p_2'\ x_2)] = \mathit{True}$. Then, we obtain $(\mathit{liftO}_{\gamma,\tau_1,\kappa_1,\beta_1}\ p_1\ x_1, \mathit{liftO}_{\gamma,\tau_2,\kappa_2,\beta_2}\ p_2\ x_2) \in \mathcal{F}\,\mathcal{S}$. □

Appendix A.3. Supplement to Proof Sketch of Lemma 3

We will prove that, for $\mathcal{U} :: N\,\gamma \leftrightarrow (\mathit{Loc}\,\gamma)$ and $\mathcal{F} :: I \leftrightarrow W\ (\mathit{Loc}\,\gamma)$ defined by

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

$(\mathcal{L}, \mathcal{U}, \mathcal{F})$ is a *PackM*-action where $\mathcal{L} = \{(e, e) \mid e :: \gamma\}$.

We write τ_1 and τ_2 for $N\,\gamma$ and $\mathit{Loc}\,\gamma$, respectively, and κ_1 and κ_2 for I and $W\ (\mathit{Loc}\,\gamma)$.

Case: return. We will prove $(\mathit{return}_{\kappa_1}, \mathit{return}_{\kappa_2}) \in \forall\mathcal{R}.\,\mathcal{R} \to \mathcal{F}\,\mathcal{R}$. We omit the proof for this case because it is similar to the one in Appendix A.2.

Case: $(\gg\!=)$. We will prove $((\gg\!=)_{\kappa_1}, (\gg\!=)_{\kappa_2}) \in \forall\mathcal{R}.\forall\mathcal{S}.\,\mathcal{F}\,\mathcal{R} \to (\mathcal{R} \to \mathcal{F}\,\mathcal{S}) \to \mathcal{F}\,\mathcal{S}$.

Let $\mathcal{R}$ and $\mathcal{S}$ be relations. Let (m_1, m_2) be a pair such that $(m_1, m_2) \in \mathcal{F}\,\mathcal{R}$, and (f_1, f_2) be a pair such that $(f_1, f_2) \in \mathcal{R} \to \mathcal{F}\,\mathcal{S}$. We will show $(m_1 \gg\!=_{\kappa_1} f_1, m_2 \gg\!=_{\kappa_2} f_2) \in \mathcal{F}\,\mathcal{S}$.

By unfolding the definition of $(\gg\!=)_{\kappa_1}$, we can rewrite $(m_1 \gg\!=_{\kappa_1} f_1)$ as $I\, y_1$ where $I\, x_1 = m_1$ and $I\, y_1 = (f_1\, x_1)$. Similarly, by unfolding the definition of $(\gg\!=)_{\kappa_2}$, we can rewrite $(m_2 \gg\!=_{\kappa_2} f_2)$ as $W\, (y_2, h_{21} \mathbin{++} h_{22})$ where $W\, (x_2, h_{21}) = m_2$ and $W\, (y_2, h_{22}) = f_2\, x_2$.

Assume $\mathit{checkHist}\, (\mathit{update}\ \mathit{upd})\, (h_{21} \mathbin{++} h_{22}) = \mathit{True}$. Then, we have $\mathit{checkHist}\, (\mathit{update}\ \mathit{upd})\, h_{2i} = \mathit{True}$ for each $i = 1, 2$ by the definition of $\mathit{checkHist}$. By $\mathit{checkHist}\, (\mathit{update}\ \mathit{upd})\, h_{21} = \mathit{True}$, we have $(x_1, x_2) \in \mathcal{R}$. Since $(f_1, f_2) \in \mathcal{R} \to \mathcal{F}\, \mathcal{S}$ and $\mathit{checkHist}\, (\mathit{update}\ \mathit{upd})\, h_{22} = \mathit{True}$, we have $(y_1, y_2) \in \mathcal{S}$. Thus, we have $(I\, y_1, W\, (y_2, h_{21} \mathbin{++} h_{22})) \in \mathcal{F}\, \mathcal{S}$.

Case: new. We will prove $(\mathit{new}_{\gamma,\tau_1}, \mathit{new}_{\gamma,\tau_2}) \in \mathcal{L} \to \mathcal{U}$. Let $(x_1, x_2) \in \mathcal{L}$. Since $\mathcal{L}$ is diagonal, we have $x_1 = x_2$. Then, we have $\mathit{new}_{\gamma,\tau_1}\, x_1 = N\, x_1$ and $\mathit{new}_{\gamma,\tau_2}\, x_2 = \mathit{Loc}\, x_2\, \mathit{Nothing}$. By the required condition on *update* that $\mathit{update}\ \mathit{upd}\, (\mathit{new}\, x_2) = \mathit{new}\, x_2$, and since $x_1 = x_2$, we have $(\mathit{new}_{\gamma,\tau_1}\, x_1, \mathit{new}_{\gamma,\tau_2}\, x_2) \in \mathcal{U}$.

Case: liftO. We will prove $(\mathit{liftO}_{\gamma,\tau_1,\kappa_1,\beta_1}, \mathit{liftO}_{\gamma,\tau_2,\kappa_2,\beta_2}) \in ([\mathcal{L}] \to \mathcal{S}) \to [\mathcal{U}] \to \mathcal{F}\, \mathcal{S}$ for all $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ such that $((\texttt{==})_{\beta_1}, (\texttt{==})_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$.

Let $\mathcal{S} :: \beta_1 \leftrightarrow \beta_2$ be a relation satisfying $((\texttt{==})_{\beta_1}, (\texttt{==})_{\beta_2}) \in \mathcal{S} \to \mathcal{S} \to \mathit{Bool}$. Let (p_1, p_2) be a pair of functions such that $(p_1, p_2) \in [\mathcal{L}] \to \mathcal{S}$ and (x_1, x_2) be a pair of lists such that $(x_1, x_2) \in [\mathcal{U}]$. By definition, we have $\mathit{liftO}_{\gamma,\tau_1,\kappa_1,\beta_1}\, p_1\, x_1 = I\, (p_1\, \$\, \mathit{map}\ \mathit{runN}\, x_1)$. Also, we have $\mathit{liftO}_{\gamma,\tau_2,\kappa_2,\beta_2}\, p_2\, x_2 = W\, (p_2'\, x_2, [\mathit{Result}\, p_2'\, x_2\, (p_2'\, x_2)])$ where $p_2' = p_2 \circ \mathit{map}\ \mathit{body}$.

Assume $\mathit{checkHist}\, (\mathit{update}\ \mathit{upd})\, [\mathit{Result}\, p_2'\, x_2\, (p_2'\, x_2)] = \mathit{True}$. This implies $p_2'\, (\mathit{map}\, (\mathit{update}\ \mathit{upd})\, x_2) = p_2'\, x_2$. Since $(x_1, x_2) \in [\mathcal{U}]$, we have $(\mathit{map}\ \mathit{runN}\, x_1, \mathit{map}\, (\mathit{body} \circ \mathit{update}\ \mathit{upd})\, x_2) \in [\mathcal{L}]$, and thus $(p_1\, \$\, \mathit{map}\ \mathit{runN}\, x_1, p_2'\, \$\, \mathit{map}\, (\mathit{update}\ \mathit{upd})\, x_2) \in \mathcal{S}$. Thus, we have $(p_1\ \$\ \mathit{map}\ \mathit{runN}\, x_1, p_2'\, x_2) \in \mathcal{S}$. As a consequence, we have $(\mathit{liftO}_{\gamma,\tau_1,\kappa_1,\beta_1}\, p_1\, x_1, \mathit{liftO}_{\gamma,\tau_2,\kappa_2,\beta_2}\, p_2\, x_2) \in \mathcal{F}\, \mathcal{S}$. □

Appendix A.4. Proofs of **ContentsFill** *and* **FillContents**

In [29], it is shown that the following theorem holds for the "lawful" *Traversable* instances.

Definition 4 (Make Function [29])**.** For *Traversable* κ, a *make function* *make* is a function of type $\forall \alpha.\, \alpha \to \cdots \to \alpha \to \kappa\, \alpha$ for which the conditions

$$\begin{aligned} \mathit{fmap}\, f\ \$\ \mathit{make}\, x_1\ \ldots\ x_n &= \mathit{make}\, (f\, x_1)\ \ldots\ (f\, x_n) \\ \mathit{contents}\ \$\ \mathit{make}\, x_1\ \ldots\ x_n &= [x_1, \ldots, x_n] \end{aligned}$$

hold. □

Theorem 2 (Representation Theorem [29])**.** *Let κ be a lawful Traversable instance with its traverse function. For every $t :: \kappa\,\tau$ for any type τ, there are a unique n, a unique n-ary make function make and unique values $v_1, \ldots, v_n$ such that*

$$t = \mathit{make}\, v_1\ \ldots\ v_n$$

and

$$\mathit{traverse}\, f\, (\mathit{make}\, x_1\ \ldots\ x_n) = \mathit{pure}\ \mathit{make}\ \texttt{<*>}\ f\, x_1\ \texttt{<*>}\ \cdots\ \texttt{<*>}\ f\, x_n.$$

hold. □

Lemma 5. FillContents holds.

Proof. By Representation Theorem, we have $t = \mathit{make}\, x_1\ \ldots\ x_n$ for some $x_1, \ldots, x_n$ and some make function *make*. Then, it suffices to prove

$$\mathit{fill}\, (\mathit{make}\, y_1\ \ldots\ y_n)\, [x_1, \ldots, x_n] = \mathit{make}\, x_1\ \ldots\ x_n$$

for any $y_1, \ldots, y_n$ to show **FillContents**. By unfolding the definition of *fill*, we can rewrite the left-hand side as

$$\begin{aligned}(\text{LHS}) &= \mathit{evalState}\, (\mathit{traverse}\, \mathit{next}\, (\mathit{make}\, y_1\ \ldots\ y_n))\, [x_1, \ldots, x_n] \\ &= \mathit{evalState}\, (\mathit{pure}\ \mathit{make}\ \texttt{<*>}\ \mathit{next}\, y_1\ \texttt{<*>}\ \cdots\ \texttt{<*>}\ \mathit{next}\, y_n)\, [x_1, \ldots, x_n]\end{aligned}$$

where *next* is the inner function used in *fill*. For simplicity, we assume that $\mathit{State}\, \sigma\, \tau$ is just a type synonym to $\sigma \to (\tau, \sigma)$. Then, we can rewrite the definitions of *pure*, (<*>), *evalState* and *next* as follows.

$$\begin{aligned}&\mathit{pure}\, x = \lambda s.(x, s) \\ &h\ \texttt{<*>}\ x = \lambda s.\ \mathbf{let}\ (g, s') = h\, s \\ &\qquad\qquad\qquad\quad (y, s'') = x\, s' \\ &\qquad\qquad\quad \mathbf{in}\ (g\, y, s'') \\ &\mathit{evalState}\, h\, s = \mathit{fst}\, (h\, s) \\ &\mathit{next}\ \_ = \lambda(a : x).(a, x)\end{aligned}$$

By the induction on k, we can show the following property.

$$\begin{aligned}&\mathit{pure}\, h\ \texttt{<*>}\ \mathit{next}\, y_1\ \texttt{<*>}\ \cdots\ \texttt{<*>}\ \mathit{next}\, y_k = \\ &\qquad \lambda(z_1 : z_2 : \cdots : z_k : r).(h\, z_1\, z_2\ \ldots\ z_k, r)\end{aligned}$$

Now we can further rewrite the original left-hand side as follows.

$$\begin{aligned}(\text{LHS}) &= \mathit{evalState}\,(\mathit{pure}\,\mathit{make}\,\texttt{<*>}\,\mathit{next}\,y_1\,\texttt{<*>}\cdots\texttt{<*>}\,\mathit{next}\,y_n)\,[x_1,\ldots,x_n] \\ &= \mathit{evalState}\,(\lambda(z_1:\cdots:z_n:r).(\mathit{make}\,z_1\ \ldots\ z_n,r))\,[x_1,\ldots,x_n] \\ &= \mathit{fst}\,((\lambda(z_1:\cdots:z_n:r).(\mathit{make}\,z_1\ \ldots\ z_n,r))\,[x_1,\ldots,x_n]) \\ &= \mathit{make}\,x_1\ \ldots\ x_n = (\text{RHS})\end{aligned}$$

Then, the proof is done. □

Lemma 6. ContentsFill holds.

Proof. By Representation Theorem, we have $t = \mathit{make}\,x_1\ \ldots\ x_n$ for some $x_1,\ldots,x_n$ and some make function *make*. Then, it suffices to prove

$$\mathit{contents}\ \$\ \mathit{fill}\,(\mathit{make}\,x_1\ \ldots\ x_n)\,[y_1,\ldots,y_n] = [y_1,\ldots,y_n]$$

holds for any $y_1,\ldots,y_n$ to show **ContentsFill**. Similar to the proof of **FillContents**, we have

$$\mathit{fill}\,(\mathit{make}\,x_1\ \ldots\ x_n)\,[y_1,\ldots,y_n] = \mathit{make}\,y_1\ \ldots\ y_n.$$

Then, the above clearly holds. □

Appendix A.5. Correctness of the Generic-Version of assignLocs

To prove that the generic-version of *assignLocs* satisfies the imposed conditions, we firstly extend the notion of the location-consistency to any ("lawful") *Traversable* datatypes.

Definition 5 (Location Consistency, Generic Version)**.** For *Traversable* κ satisfying Representation Theorem, $t :: \kappa\,(\mathit{Loc}\,\gamma)$ is called *location-consistent* if $\mathit{content}\,t$ is. □

Lemma 7. The Generic-Version of *assignLocs* is location-consistent.

Proof. Straightforward from the definition of *assignLocs*, **ContentsFill** and the fact that *assignLocsList* is location-consistent. □

Lemma 8. $\mathit{fmap}\ \mathit{body}\,(\mathit{assignLocs}\,s) = s$

Proof. We prove the statement as follows.

$$
\begin{aligned}
(\text{LHS}) &= \{ \text{ by definition } \} \\
&\quad \mathit{fmap}\ \mathit{body}\ (\mathit{fill}\ s\ (\mathit{assignLocsList}\ \$\ \mathit{contents}\ s)) \\
&= \{ \text{ free theorem on } \mathit{fill} \ \} \\
&\quad \mathit{fill}\ s\ (\mathit{fmap}\ \mathit{body}\ \$\ \mathit{assignLocsList}\ \$\ \mathit{contents}\ s) \\
&= \{ \ \mathit{fmap}\ \mathit{body}\ (\mathit{assignLocsList}\ s) = s \ \} \\
&\quad \mathit{fill}\ s\ (\mathit{contents}\ s) \\
&= \{ \ \textbf{FillContents} \ \} \\
&\quad s
\end{aligned}
$$

□

Appendix A.6. Correctness and Minimality of the Generic-Version of matchViews

We firstly show the following invariant of *makeUpd*.

Lemma 9. If $\mathit{makeUpd}\ z$ succeeds and results in *upd*, then $\mathit{map}\ (\mathit{body} \circ \mathit{update}\ \mathit{upd})\ (\mathit{map}\ \mathit{fst}\ z) = \mathit{map}\ \mathit{snd}\ z$.

Proof. By induction. □

Lemma 10 (Correctness)**.** For any v' and the corresponding location-consistent vx, if *matchViews* succeeds and results in *upd*, then $\mathit{fmap}\ (\mathit{body} \circ \mathit{update}\ \mathit{upd})\ vx = v'$.

Proof. By **FillContents**, we can write $vx = \mathit{fill}\ (\mathit{fmap}\ \mathit{ignore}\ vx)\ (\mathit{contents}\ vx)$ and $v' = \mathit{fill}\ (\mathit{fmap}\ \mathit{ignore}\ v')\ (\mathit{contents}\ v')$. By Lemma 9, we have $\mathit{map}\ (\mathit{body} \circ \mathit{update}\ \mathit{upd})\ (\mathit{contents}\ vx) = \mathit{contents}\ v'$. By the free theorem on *fill*, we have

$$\mathit{fmap}\ h\ (\mathit{fill}\ t\ x) = \mathit{fill}\ t\ (\mathit{map}\ h\ x)$$

for any h, t and x such that $\mathit{length}\ x = \mathit{length}\ (\mathit{contents}\ t)$. The success of *matchViews* ensures that $\mathit{fmap}\ \mathit{ignore}\ vx = \mathit{fmap}\ \mathit{ignore}\ v'$. Thus, taking t above as $t = \mathit{fmap}\ \mathit{ignore}\ vx = \mathit{fmap}\ \mathit{ignore}\ v'$ and h as $h = \mathit{body} \circ \mathit{update}\ \mathit{upd}$, we obtain $\mathit{fmap}\ (\mathit{body} \circ \mathit{update}\ \mathit{upd})\ vx = v'$. □

Lemma 11 (Minimality)**.** For any v and location-consistent vx such that $\mathit{fmap}\ \mathit{body}\ vx = v$, $\mathit{matchViews}\ vx\ v = []$ holds. □

Proof. In this case, it is easy to see that *makeUpd* always succeeds and returns an update of the form $[(i, x) \mid \mathit{Loc}\ x\ (\mathit{Just}\ i) \leftarrow \mathit{contents}\ vx]$. Then, $\mathit{minimize}\ (\mathit{contents}\ vx)$ returns $[]$ for the update. □

Appendix B. Examples showing Weaker Consistency Laws being Non-compositional

We will show that the weaker consistency laws are not closed under composition.

We use the term a lens for a pair of forward and backward transformations [7]. Let us write the forward semantics of a lens l as $[\![l]\!]_{\mathrm{F}}$ and its backward semantics as $[\![l]\!]_{\mathrm{B}}$. Then, the composition combinator $\circ$ of lenses is defined as follows [7].

$$\begin{aligned}
&[\![l_1 \circ l_2]\!]_{\mathrm{F}} = [\![l_1]\!]_{\mathrm{F}} \circ [\![l_2]\!]_{\mathrm{F}} \\
&[\![l_1 \circ l_2]\!]_{\mathrm{B}}\, a\, c' = \mathbf{let}\ b \ = [\![l_2]\!]_{\mathrm{F}}\, a \\
&\qquad\qquad\qquad\qquad\quad\ b' = [\![l_1]\!]_{\mathrm{B}}\, b\, c' \\
&\qquad\qquad\qquad\quad\ \mathbf{in}\ [\![l_2]\!]_{\mathrm{B}}\, a\, b'
\end{aligned}$$

The definition of the composition combinator $\circ$ is standard [6, 7, 9, 11, 13, 14, 17, 21]. It is known that the combinator $\circ$ preserves the acceptability and consistency laws.

We prepare two primitive lenses that represent duplicates in a source and a view, and use them to show that the weaker laws are not closed under $\circ$. The primitive *dup* duplicates its input [11, 13] and *merge* treats equal inputs as duplicates [7].

$$\begin{aligned}
&[\![\mathit{dup}]\!]_{\mathrm{F}}\, x = (x, x) \\
&[\![\mathit{dup}]\!]_{\mathrm{B}}\, x\, (x_1, x_2) \mid x_1 == x_2 = x_1 \\
&\qquad\qquad\qquad\quad \mid x == x_1 \ = x_2 \\
&\qquad\qquad\qquad\quad \mid x == x_2 \ = x_1
\end{aligned}
\qquad\qquad
\begin{aligned}
&[\![\mathit{merge}]\!]_{\mathrm{F}}\, (x, _) = x \\
&[\![\mathit{merge}]\!]_{\mathrm{B}}\, (x, y)\, z \mid x == y \ \ \ = (z, z) \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = (z, y)
\end{aligned}$$

Note that *dup* satisfies the acceptability law, not the consistency law; but it satisfies the weaker consistency laws, **Weak-PutGet** and **PutGetPut**. In contrast, *merge* satisfies both the acceptability and consistency laws.

Let us consider the following compound lens.

$$h = \mathit{merge} \times \mathit{merge} \circ \mathit{dist} \circ \mathit{second}_3\ \mathit{dup}$$

Here, the primitive *dist* and the combinator second_3 manipulate triples. The primitive *dist* is the following bijection.

$$\begin{aligned}
&[\![\mathit{dist}]\!]_{\mathrm{F}}\, (a, (b, c), d) = ((a, b), (c, d)) \\
&[\![\mathit{dist}]\!]_{\mathrm{B}}\, _\, ((a, b), (c, d)) = (a, (b, c), d)
\end{aligned}$$

The lens $second_3\, l$ applies l to the second component of a triple.

$$[\![second_3\, l]\!]_{\mathrm{F}}\, (a, b, c) = (a, [\![l]\!]_{\mathrm{F}}\, b, c)$$
$$[\![second_3\, l]\!]_{\mathrm{B}}\, (\_, b, \_)\, (a, b', c) = (a, [\![l]\!]_{\mathrm{B}}\, b\, b', c)$$

The combinator $\times$ is also a standard bidirectional combinator; $l_1 \times l_2$ applies l_1 and l_2 to each component of a pair [7].

$$[\![l_1 \times l_2]\!]_{\mathrm{F}}\, (a, b) = ([\![l_1]\!]_{\mathrm{F}}\, a, [\![l_2]\!]_{\mathrm{F}}\, b)$$
$$[\![l_1 \times l_2]\!]_{\mathrm{B}}\, (a, b)\, (a', b') = ([\![l_1]\!]_{\mathrm{B}}\, a\, a', [\![l_2]\!]_{\mathrm{B}}\, b\, b')$$

Let us consider a source $(0, 0, 0)$. Applying $[\![h]\!]_{\mathrm{F}}$, we obtain $(0, 0)$. Consider an updated view $(1, 0)$. Applying $[\![h]\!]_{\mathrm{B}}$ to the original source and the updated view, we obtain an updated source $(1, 1, 0)$. Applying $[\![h]\!]_{\mathrm{F}}$ to the updated source, we get $(1, 1)$ (recall that $[\![merge]\!]_{\mathrm{F}}$ returns the first component of a pair). However, applying $[\![h]\!]_{\mathrm{B}}$ to the original source and the new view $(1, 1)$, we obtain $(1, 1, 1)$, which violates **Weak-PutGet**.

This example itself does not violate **PutGetPut** because it satisfies the acceptability law which implies **PutGetPut**. To prepare an example that violates **PutGetPut**, we change the example a bit. Concretely, we replace the backward semantics of *merge* as follows.

$$[\![merge]\!]_{\mathrm{B}}\, \_\, x = (x, x)$$

This version of $[\![merge]\!]_{\mathrm{B}}$ still satisfies the consistency law, and thus all the primitives in h satisfy **PutGetPut**. Let us consider a source $(0, 0, 0)$. Applying $[\![h]\!]_{\mathrm{F}}$, we obtain $(0, 0)$. Consider an updated view $(1, 0)$. Applying $[\![h]\!]_{\mathrm{B}}$ to the original source and the updated view, we obtain an updated source $(1, 1, 0)$. Applying $[\![h]\!]_{\mathrm{F}}$ to the updated source, we get $(1, 1)$. However, applying $[\![h]\!]_{\mathrm{B}}$ to the updated source $(1, 1, 0)$ and the new view $(1, 1)$, we obtain $(1, 1, 1)$, which violates **PutGetPut**.