The Under-Appreciated Put: Implementing Delta-Alignment in BiGUL

Jorge Mendes
HASLab, INESC TEC & Universidade do Minho, Portugal
jorgemendes@di.uminho.pt

Hsiang-Shang Ko Zhenjiang Hu
National Institute of Informatics, Japan
{hsiang-shang,hu}@nii.ac.jp

Abstract

There are two approaches to bidirectional programming: One is the get-based method where one writes get, and put is automatically derived; the other is the put-based method where one writes put and get is automatically derived. In this paper, we argue that the put-based method deserves more attention, because a good language for programming put can not only give full control over the behavior of bidirectional transformations, but also enable us to efficiently develop various domain-specific bidirectional languages and use them seamlessly in one framework, which would be non-trivial with the get-based method. We demonstrate how the matching/delta/generic lenses can be implemented in BiGUL, a putback-based bidirectional language embedded in Haskell.

1. Introduction

Bidirectional transformations are hot! They originated from the *view updating* mechanism in the database community [1, 6, 10], and have been attracting a lot of attention from researchers in the communities of programming languages and software engineering [5, 13], since the pioneering work of Foster et al. on a combinatorial language for bidirectional tree transformations [9].

A bidirectional transformation (BX for short) is simply a pair of functions

$$\begin{aligned} &get :: Source \to View \\ &put :: Source \to View \to Source \end{aligned}$$

where the *get* function extracts a view from a source and the *put* function updates the original source with information from the new view. As a simple example, suppose that we wish to synchronize between rectangles and their heights. We can define

$$\begin{aligned} &getHeight\ (height, width) &&= height \\ &putHeight\ (height, width)\ height' &&= (height', width) \end{aligned}$$

where a rectangle is represented by a pair of its height and width.

Certainly not any pair of get and put can form bidirectional transformations for synchronization. get and put should satisfy the *well-behavedness* laws:

$$put\ s\ (get\ s) = s \qquad \textsc{GetPut}$$
$$get\ (put\ s\ v) = v \qquad \textsc{PutGet}$$

The GETPUT law requires that no changing on the view shall be reflected as no changing on the source, while the PUTGET law requires all changes in the view to be completely reflected to the source so that the changed view can be computed again by applying the forward transformation to the changed source. For instance, if we change the above put to

$$putHeight'\ (height, width)\ height' = (height' + 1, width)$$

get and put' will break the laws.

A straightforward approach to developing well-behaved BXs in order to solve various synchronization problems is to write both get and put. The approach has the practical problem that the programmer needs to show that the two transformations satisfy the well-behavedness laws, and a modification to one of the transformations requires a redefinition of the other transformation as well as a new well-behavedness proof. To ease and enable maintainable bidirectional programming, it is preferable to write just a single program that can denote both transformations, which has motivated two different approaches:

- *Get-based method*: allowing users to write get and derive a suitable put [2, 3, 9, 11, 12, 17, 18];
- *Put-based method*: allowing users to write put and derive the unique get if there is one [8, 14, 16, 19, 20].

The get-based method has been intensively studied for over ten years and got much appreciated. It is attractive, because get is easy to write, and if the system knows how to derive a put, there would be no additional burden for users to go from unidirectional to bidirectional. In contrast, the put-based method is new and far from being appreciated. One main criticism is that put is more difficult to write than get.

However, the get-based method hardly describes the full behavior of a bidirectional transformation, so automatically derived put may not match the programmers' intention, which would prevent it from being used in practice. More specifically, for a non-injective get there usually exist many possible put functions that can be combined with it to form a valid BX. For instance, for the same $getHeight$, the following is a valid put too:

$$\begin{aligned} &putHeight''\ (height, weight)\ height' \\ &\quad = (height', weight\ x\ (height' \,/\, height)) \end{aligned}$$

In fact, it is impossible in general to automatically derive the most suitable valid put that can be paired with the get to form a bidirectional transformation [4].

Since get does not contain sufficient information for a system to automatically derive intended update policies of put, in order to deal with various update policies of put for solving different problems, significant extensions to the language for writing get are necessary. One representative problem is *alignment*: Both the source and view are lists, and the get direction is simply a map on lists.

The *put* direction, on the other hand, has a great amount of freedom: The elements of the view list may be inserted or deleted. A view deletion can only be reflected as a source deletion, if we want the get direction to be just a map; for an insertion, however, how do we create a corresponding source element from a usually less informative view element? The view list may be reordered. How do we determine which source element should be matched with each view element? *Matching lenses* [2] were developed to be able to customize such policies. There are other finer-grained considerations: Given an updated view list, how do we decide whether an element is only modified — so the corresponding source element also only needs modification — or newly inserted — so we should delete the corresponding source element and insert a new one? This requires tracking of how the view list is modified, and frameworks for expressing modification-sensitive update policies were developed [7, 12]. We may want to go beyond lists to trees and in general any inductive data structures, and, of course, there is work on *generic lenses* to deal with any updates on inductive data structures [18]. All these extensions, as seen in the related papers, are nontrivial, where one has to rework all the original lens framework by adding new information to get to indirectly control of the behavior of put, and to prove that the extension is sound in the sense that the new get and put are well-behaved.

In this paper, we put up the slogan "One put for All", in the sense that a good language for programming put can not only give full control over the behavior of bidirectional transformations, but also enable us to systematically develop various domain-specific bidirectional languages and use them seamlessly in one framework, which would be nontrivial with the get-based method as seen above. In the rest of this paper, after a brief review of BiGUL [16], a putback-based bidirectional language embedded in Haskell, we demonstrate how it can be used to concisely implement the matching/delta/generic lenses that are guaranteed to be well-behaved.

2. Preparation: Putback-Based BX

In this paper, we use BiGUL version 0.9, which is available on Hackage. Intuitively, think of a BiGUL program of type $BiGUL\ s\ v$ as describing how to manipulate a state consisting of a source component of type s and a view component of type v; the goal is to copy all information in the view to proper places in the source. In the simplest case, the view has type $()$ and contains no information, and we can use $Skip :: BiGUL\ s\ ()$ to leave the source unchanged; another simple case is when the view has the same type as the source, and we can use $Replace :: BiGUL\ s\ s$ to replace the entire source with the view. BiGUL programs compose — for example, when both the source and the view are pairs, we can use

$$Prod :: BiGUL\ s\ v \to BiGUL\ s'\ v' \to BiGUL\ (s, s')\ (v, v')$$

to compose two BiGUL programs on the left and right components respectively; we will typeset the infix application of $Prod$ as '$\times$'. Of course, in most cases the source and view are in more complex forms, and we should somehow transform and decompose them into simpler forms before we can use $Skip$, $Replace$, or $Prod$; this is usually done using two "rearrangement" operations on the source and view respectively: We can use the source rearranging operation

$$\$\ (rearrS\ [|\ f\ |]) :: BiGUL\ s'\ v \to BiGUL\ s\ v$$

where f is a "simple" λ-expression of type $s \to s'$ for extracting from the source of type s a (usually smaller) source of type s' before performing further updates on the extracted source, or dually the view rearranging operation

$$\$\ (rearrV\ [|\ g\ |]) :: BiGUL\ s\ v' \to BiGUL\ s\ v$$

where the "simple" λ-expression g should have type $v \to v'$, and is used to transform the view from type v to type v' before performing further updates.

Most expressiveness of BiGUL comes from its $Case$ operation for performing case analysis:

$$Case :: [(s \to v \to Bool, Branch\ s\ v)] \to BiGUL\ s\ v$$

$Case$ takes a list of pairs whose first component is a boolean predicate on both the source and the view, and whose second component is a "branch", whose type is defined by

$$\begin{aligned}\mathbf{data}\ Branch\ s\ v = {} & Normal \quad (BiGUL\ s\ v) \\ \mid {} & Adaptive\ (s \to v \to s)\end{aligned}$$

A branch can be a "normal" branch, in which case it is a BiGUL program of type $BiGUL\ s\ v$, or an "adaptive" branch, in which case it is a Haskell function of type $s \to v \to s$. Roughly speaking, the semantics of $Case$ is largely as people would expect: executing the first branch whose associated predicate evaluates to true on the current state, and performing further updates when this branch is normal. More interestingly, when the chosen branch is adaptive, the source will be replaced by the result of evaluating the associated function on the current state, and the whole $Case$ will be executed again.

We introduce some extra notations for writing branches more easily. The two basic ones are for constructing normal and adaptive branches in general:

$$\begin{aligned}&\$\ (normal \quad [|\ p\ |]) \Longrightarrow b = (p, Normal \quad b)\\ &\$\ (adaptive\ [|\ p\ |]) \Longrightarrow f = (p, Adaptive\ f)\end{aligned}$$

Here the boolean predicate p takes both a source and a view. Often this predicate is a conjunction of two unary predicates on the source and view respectively, so we introduce another set of notations:

$$\begin{aligned}&\$\ (normalSV \quad [|\ pS\ |]\ [|\ pV\ |]) \Longrightarrow b\\ &\quad = ((\lambda s\ v \to pS\ s \wedge pV\ v), Normal \quad b)\\ &\$\ (adaptiveSV\ [|\ pS\ |]\ [|\ pV\ |]) \Longrightarrow f\\ &\quad = ((\lambda s\ v \to pS\ s \wedge pV\ v), Adaptive\ f)\end{aligned}$$

The unary predicates (pS and pV) can usually be conveniently expressed as patterns; $normalSV$ and $adaptiveSV$ can also accept patterns, which should be enclosed in pattern quotation brackets like $[p|\ pat\ |]$. There are also other variants of $normal$ and $adaptive$ that are suffixed with only S or V, meaning that they accept only one unary predicate on either the source or the view, respectively.

3. Positional Alignment

As a warm-up, let us try to describe in BiGUL the simplest alignment strategy, which matches elements at the same positions in the source and view lists.

Suppose that a research institution has a listing of authors with their publication count and another listing with its researchers, where authors and researchers are respectively represented by the following types:

$$\begin{aligned}&\mathbf{type}\ Author \quad\ = (ID, (Name, Publications))\\ &\mathbf{type}\ Researcher = (ID,\ \ Name)\end{aligned}$$

where $ID :: Int$ is the identification number (id for short), $Name :: String$ is the name of the author/researcher, and $Publications :: Int$ is the number of publications of the author. The relation between an author and a researcher is straightforward: a researcher is an author without the publication count and thus ids and names of an author must match with the ones of the respective researcher. This can be expressed using the BiGUL program:

$$
\begin{aligned}
&arBX :: BiGUL\ Author\ Researcher \\
&arBX = Replace \times \$(rearrV\ [|\ \lambda c \to (c, ())\ |]) \\
&\qquad\qquad (Replace \times Skip)
\end{aligned}
$$

First, we want to replace the source id (author) with the view id (researcher) and thus we use *Replace* on the left-hand side of a *Prod*, whose right-hand side will deal with the name part. Since there is a mismatch of shape between the source and the view in that part — the source is a pair whilst the view is a single element — we rearrange the view (researcher name) into a pair whose second component is an unit. After the rearrangement, we can use a *Prod* with a *Replace* in the left-hand side to replace the name using a *Replace* and with a *Skip* on the right-hand side ignoring the value of the view and keeping the value of the source (publication count) unchanged.

The listings are represented by lists of the above types. So we have a list of authors and a list of researchers. With positional alignment, the relation between the two lists is simple: each element of the author list matches the element of researcher list at the same position. When performing any operation on the view (list of researchers), reordering of the elements (i.e., an element moved to another position) is not taken into account, and elements are added or deleted at the end of the source. Just as with any other programming practice, the BiGUL program must take into account the several possibilities of source and view values in the update process. For that, we specify a function ($arMapL$) to map positionally a list of researchers with a list of authors:

- both source and view are empty, and we just *Skip*;
- all elements of the view were processed, so we adapt the source (explained below) by removing the extra elements: $\lambda\_\ \_ \to [\,]$;
- both source and view have elements, so we update with the head of both source and view, and then recurse on the tail of the list: $arBX \times arMapL$;
- the source does not have enough elements and we create a new one, setting to 0 the number of publications of the new author: $\lambda\_\ ((k, v_1) : \_) \to [(k, (v_1, 0))]$.

These possibilities are packed into a *Case* statement which selects the correct action for each situation:

$$
\begin{aligned}
&arMapL :: BiGUL\ [Author]\ [Researcher] \\
&arMapL = Case \\
&\quad [\$(normalSV\ [p \mid [\,]\ |]\ [p \mid [\,]\ |]) \\
&\qquad \Longrightarrow \$(rearrV\ [|\ \lambda[\,] \to ()\ |])\ Skip \\
&\quad ,\$(adaptiveV\ [p \mid [\,]\ |]) \\
&\qquad \Longrightarrow \lambda\_\ \_ \to [\,] \\
&\quad ,\$(normalSV\ [p \mid (\_ : \_)\ |]\ [p \mid (\_ : \_)\ |]) \\
&\qquad \Longrightarrow \$(rearrV\ [|\ \lambda(v : vs) \to (v, vs)\ |])\ \$ \\
&\qquad\quad \$\ (rearrS\ [|\ \lambda(s : ss) \to (s, ss)\ |])\ \$ \\
&\qquad\qquad arBX \times arMapL \\
&\quad ,\$(adaptiveV\ [p \mid (\_ : \_)\ |]) \\
&\qquad \Longrightarrow \lambda\_\ ((k, v_1) : \_) \to [(k, (v_1, 0))] \\
&\quad]
\end{aligned}
$$

When both source and view are empty, or both have elements, a BiGUL program can be applied: When both are empty, the empty list is produced; when both have elements, the head of the source is updated with the head of the view, and then recursion is performed.

In the other two cases, adaptation of the source is required. The first one is when the view is empty, and the source is modified to be the empty list. After this adaption, the *Case* statement looks for a normal branch to apply, entering in the one where both source and view are empty. The second case is when the view still has elements, but the source is empty. In this case, a new source element is created from the source element at the head of the list. Then, the *Case* statement looks for a normal branch, entering in the one where both source and view have elements, updating the heads and recursing.

To demonstrate the BX functions, let

$$source = [(0, (\texttt{"A."}, 3)), (1, (\texttt{"B."}, 5)), (2, (\texttt{"C."}, 8))]$$

Running the *get* function on *source* with the $arMapL$ BiGUL program, we obtain the following result:

```
> get arMapL source
[(0,"A."),(1,"B."),(2,"C.")]
```

Now, if we want to expand the last name of the authors/researchers, we just modify the result above and then put it back into the original source:

```
> put arMapL source [(0,"Ana"),(1,"Bob"),(2,"Carl")]
[(0,("Ana",3)),(1,("Bob",5)),(2,("Carl",8))]
```

We can see that the original authors are updated according the changes made to the view. However, we can see the limitations of positional update when removing an element, e.g., `(1,"B.")`:

```
> put arMapL source [(0,"A."),(2,"C.")]
[(0,("A.",3)),(2,("C.",5))]
```

or adding a new one, e.g., `(3,"D.")` before the end[1]:

```
> put arMapL source ↩
    [(0,"A."),(1,"B."),(3,"D."),(2,"C.")]
[(0,("A.",3)),(1,("B.",5)),(3,("D.",8)),(2,("C.",0))]
```

Notice the number of publications. For the removal example, it is as `(2,"C.")` was removed and `(1,"B.")` was modified to `(2,"C.")`. For the addition example it is as `(2,"C.")` was added at the end of the view, and `(2,"C.")` from the original view was modified to `(3,"D.")`.

The $arMapL$ program can be generalized to work on lists with arbitrary values. For that, it must be parametrized with a *create* function, to produce a source element from a view one as in the fourth branch of the $arMapL$ *Case*, and with a BiGUL program to be run on the elements instead of $arBX$:

$$mapL :: (v \to s) \to BiGUL\ s\ v \to BiGUL\ [s]\ [v]$$

4. Key-Based Alignment

The positional alignment strategy is not the best regarding our example shown in the previous section. Looking at the types, we can define a better strategy using the ids to match the authors and researchers as it is done naturally in other contexts.

More complex alignment strategies can be implemented using BiGUL. One example is a key-based one, where elements of the source and the view are paired based on a key component from each of the elements. Thus, we can use this strategy to implement a better alignment.

One could implement a key-based alignment strategy using a structure similar to the positional alignment. However, a simpler approach is available. The idea to implement this strategy is to separate the program in two parts:

- alignment of the elements;
- the actual update, using a positional mapping which is sufficient when the elements are aligned.

To align the elements, we must first be able to extract a key from source and view elements. For our running example, we use the first component of the source, and the same for the view. Thus, we can use the *fst* function to extract the key from either elements. In order to help with the implementation, we define a function to check if the source and the view are aligned:

[1] The symbol ↩ denotes line continuation.

$$
\begin{aligned}
&\mathit{isAligned}\ s\ v = \mathit{length}\ s \equiv \mathit{length}\ v \\
&\qquad\qquad \wedge\ \mathit{and}\ (\mathit{zipWith}\ \mathit{kmatch}\ s\ v) \\
&\quad \textbf{where}\ \mathit{kmatch}\ \mathit{se}\ \mathit{ve} = \mathit{fst}\ \mathit{se} \equiv \mathit{fst}\ \mathit{ve}
\end{aligned}
$$

We consider that source and view are aligned if both have the same number of elements, and that the keys match element-wise.

In the case that the two lists are not aligned, we define a function that adapts a source such that when applied they become aligned. This is performed by traversing the view and fetching the first corresponding element in the original source. If such element is not present, we create it. At the end, source elements not present in view are discarded. The adaptation of the source can be implemented as:

$$
\begin{aligned}
&\mathit{arKeyMatchAdapt}\ s\ v = \mathit{map}\ \mathit{getSourceElement}\ v \\
&\quad \textbf{where}\ \mathit{getSourceElement}\ \mathit{ve} = \\
&\qquad\qquad \textbf{case}\ \mathit{filter}\ ((\equiv \mathit{fst}\ \mathit{ve}) \circ \mathit{fst})\ s\ \textbf{of} \\
&\qquad\qquad\quad [\,] \to \mathit{create}\ \mathit{ve} \\
&\qquad\qquad\quad (\mathit{se} : \_) \to \mathit{se} \\
&\qquad\qquad \mathit{create}\ (k, v_1) = (k, (v_1, 0))
\end{aligned}
$$

When the source and the view are aligned, a simple positional update, as defined in the previous section, can be used. Thus, putting it all together, we obtain the following BiGUL program:

$$
\begin{aligned}
&\mathit{arKeyMatch} :: \mathit{BiGUL}\ [\mathit{Author}]\ [\mathit{Researcher}] \\
&\mathit{arKeyMatch} = \mathit{Case} \\
&\quad [\$(\mathit{normal}\ [|\ \mathit{isAligned}\ |]) \Longrightarrow \mathit{arMapL} \\
&\quad ,\$(\mathit{adaptive}\ [|\ \lambda\_\ \_ \to \mathit{True}\ |]) \Longrightarrow \mathit{arKeyMatchAdapt}]
\end{aligned}
$$

The result of running the *get* function with *arKeyMatch* is the same as with *arMapL* since they only differ in the alignment strategy:

```
> source
[(0,("A.",3)),(1,("B.",5)),(2,("C.",8))]
> get arKeyMatch source
[(0,"A."),(1,"B."),(2,"C.")]
```

Running the *put* function also has the same result when the elements are the same and the order did not change:

```
> put arKeyMatch source ↩
    [(0,"Ana"),(1,"Bob"),(2,"Carl")]
[(0,("Ana",3)),(1,("Bob",5)),(2,("Carl",8))]
```

However, when removing elements, e.g., `(1,"B.")` or adding new ones, e.g., `(3,"D.")`, key-based alignment is more precise than positional:

```
> put arKeyMatch source [(0,"A."),(2,"C.")]
[(0,("A.",3)),(2,("C.",8))]
> put arKeyMatch source ↩
    [(0,"A."),(1,"B."),(3,"D."),(2,"C.")]
[(0,("A.",3)),(1,("B.",5)),(3,("D.",0)), ↩
  (2,("C.",8))]
```

Nonetheless, key-based alignment also has its limitations, e.g., when modifying the key of an element (`(1,"B.")` to `(4,"B.")`):

```
> put arKeyMatch source [(0,"A."),(4,"B."),(2,"C.")]
[(0,("A.",3)),(4,("B.",0)),(2,("C.",8))]
```

As with the positional update, this program can be generalized for key-based alignment on lists with arbitrary contents. For that, the *arKeyMatch* function must be parametrized with a function to get a key component from the source, another function to get the key component from the view, and the create function and BiGUL update program as with *mapL*:

$$
\begin{aligned}
&\mathit{keyMatch} :: \mathit{Eq}\ k \Rightarrow (s \to k) \to (b \to k) \\
&\qquad\qquad \to (v \to s) \to \mathit{BiGUL}\ s\ v \to \mathit{BiGUL}\ [s]\ [v]
\end{aligned}
$$

5. Delta-Based List Alignment

The pattern used for the implementation of the key-based alignment strategy — the separation of the alignment of source and view in a step and then the element-wise update in another one — is actually very powerful and allows to implement much more complex alignment strategies.

For instance, going back to the running example, if one wants to change the id of a researcher alongside with other operations, that would not be possible with either the positional nor the key-based alignment strategies. More concretely, putting back `[(0,"A."),(1,"C.")]`, where `(1,"B.")` was removed and `(2,"C.")` was changed to `(1,"C.")`, into `[(0,("A.",3)),(1,("B.",5),(2,("C.",8))]` would not yield the expected `[(0,("A.",3)),(1,("C.",8))]`. In this case, we are missing information about the operations performed which would lead to a correct update of the list of authors.

Alignment can be made more precise using information about how the view is modified. If we extract the relation of elements in the original view to the elements in the modified view, then the alignment performed when updating the source can be completely correct.

The relation of elements in the original view with the ones in the modified view can be defined by a mapping from the location of the element in the original artifact to the location of the element in the modified artifact. The location can be defined as an integer index within the container

$$
\textbf{type}\ \mathit{Loc} = \mathit{Int}
$$

and the mapping, i.e., the delta, can be defined as a set of pairs of these locations

$$
\textbf{type}\ \mathit{Delta} = \mathit{Set}\ (\mathit{Loc}, \mathit{Loc})
$$

Furthermore, we need a method to determine from a delta if some artifact has undergone any positional change (movement within the container, addition, or removal), which can be accomplished by checking if all elements are in the delta and that each location in the delta is related to the same location:

$$
\delta \equiv \mathit{getId}\ \mathit{artifact}
$$

The *getId* function creates an identity delta based on the locations of the artifact:

$$
\begin{aligned}
&\mathit{getIdL} :: [a] \to \mathit{Delta} \\
&\mathit{getIdL} = \mathit{map}\ (\lambda l \to (l, l)) \circ \mathit{locs}
\end{aligned}
$$

5.1 Delta Alignment for Lists

In order to implement such kind of alignment in BiGUL, the delta can be inserted into the source, since we can manipulate it using adaptation in *Case* branches.

The implementation of delta-based alignment is similar to the key-based one:

1. modification of the source aligning to the view using a delta;
2. a positional update.

However, the delta in the source introduces a bit more complexity to deal with the additional information. We no longer use the keys of the elements to check if the lists are aligned, but we verify that purely based on the delta: If the identity delta of the source and the identity delta of the view are both equal to the given delta, then both source and view elements are aligned:

$$
\mathit{isDeltaAlignedL}\ (s, d)\ v = d \equiv \mathit{getIdL}\ v \wedge d \equiv \mathit{getIdL}\ s
$$

When the source and the view are not aligned, we adapt the source similarly to the key-based alignment. However, in this case we also have the delta present in the source, which should be the same as

both the source and the view after the adaptation so that we can perform the positional mapping. Implementing this in the running example:

$$
\begin{aligned}
&arAlignL' :: BiGUL\ ([Author], Delta)\ [Researcher]\\
&arAlignL' = Case\\
&\quad [\$(normal\ [|\ isDeltaAlignedL\ |])\\
&\qquad \Longrightarrow \$(rearrS\ [|\ \lambda(s, \_) \to s\ |])\ arMapL\\
&\quad ,\$(adaptiveS\ [|\ const\ True\ |])\\
&\qquad \Longrightarrow \lambda(s, d)\ v \to \mathbf{let}\ s' = arAdaptDeltaL\ s\ v\ d\\
&\qquad\qquad\qquad\ \mathbf{in}\ (s', getIdL\ v)]
\end{aligned}
$$

An alternative *Case* statement is used to check which of these two steps are to be performed. This is done based on the changes performed on the view: if no changes were performed, the delta maps each element's position to the same position, i.e., the identity delta. However, the delta being the same as $getIdL\ v$ does not mean that no changes were performed to the view, e.g., some values were deleted, thus not present in the view nor in the delta relation. To deal with this situation, we ensure that the delta is also equal to the identity delta of the source, i.e., both source and view contain the same positions and the update can be safely performed. Otherwise, a transformation is performed on the source to rearrange the elements based on the delta, create missing view elements, and delete no longer existent view elements:

$$
\begin{aligned}
&arAdaptDeltaL :: [Author] \to [Researcher] \to Delta\\
&\qquad\qquad \to [Author]\\
&arAdaptDeltaL\ s\ v\ d =\\
&\quad map\ idOrCreate\ (elems\ \$\ locs\ v)\\
&\quad \mathbf{where}\ idOrCreate\ i = \mathbf{let}\ js = rngOf\ i\ d\\
&\qquad\qquad\qquad\qquad\ \mathbf{in\ if}\ js \not\equiv \emptyset\\
&\qquad\qquad\qquad\qquad\quad \mathbf{then}\ s\ !!\ findMin\ js\\
&\qquad\qquad\qquad\qquad\quad \mathbf{else\ let}\ (k, v_1) = v\ !!\ i\\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ (k, (v_1, 0))
\end{aligned}
$$

However, having the delta paired with the source might be inconvenient. To deal with such situation, a wrapper is made that takes care of dealing with the delta:

$$
\begin{aligned}
&arAlignL :: Delta \to BiGUL\ [Author]\ [Researcher]\\
&arAlignL\ d = emb\ g\ p\\
&\quad \mathbf{where}\ g\ s\quad = get\ arAlignL'\ (s, getIdL\ s)\\
&\qquad\quad p\ s\ v = fst\ \$\ put\ arAlignL'\ (s, d)\ v
\end{aligned}
$$

This wrapper implements directly the *get* and *put* functions (respectively g and p), and embeds them into a BiGUL program, since this pair of *get*/*put* functions is well-behaved:
GETPUT – this law states that if no changes to the view are performed, then putting it back into the source does not alter the source. Since no changes are performed, the delta is the identity delta of the view, i.e., $\delta = getIdL\ v$ where $v = g\ s$. Furthermore, v is consistent with $(s, getIdL\ s)$, so we know that $getIdL\ s = getIdL\ v$. Applying *fst* to both sides of the following equation gives us GETPUT:

$$
\begin{aligned}
&put\ arAlignL'\ (s, getIdL\ v)\ (get\ arAlignL'\ (s, getIdL\ s))\\
&\equiv \{\, getIdL\ v \equiv getIdL\ s \,\}\\
&put\ arAlignL'\ (s, getIdL\ s)\ (get\ arAlignL'\ (s, getIdL\ s))\\
&\equiv \{\, \text{GETPUT}\ for\ arAlignL' \,\}\\
&(s, getIdL\ s)
\end{aligned}
$$

PUTGET – this law states that the view after updating a source is the same as the one used for the update. As the result of the *put* function, let $(s', \delta') = put\ arAlignL'\ (s, \delta)\ v$, thus (s', δ') is consistent with v and $\delta' \equiv getIdL\ s'$. Applying the *get* function g:

$$
\begin{aligned}
&get\ arAlignL'\ (s', getIdl\ s')\\
&\equiv \{\, \delta' = getIdL\ s' \,\}\\
&get\ arAlignL'\ (s', \delta')\\
&\equiv \{\, \mathbf{let}\ binding \,\}\\
&get\ arAlignL'\ (put\ arAlignL'\ (s, \delta))\\
&\equiv \{\, \text{PUTGET}\ for\ arAlignL' \,\}\\
&v
\end{aligned}
$$

As an aside, the embedding of *get* and *put* functions can be defined as a BiGUL program:

$$
\begin{aligned}
&emb :: Eq\ v \Rightarrow (s \to v) \to (s \to v \to s) \to BiGUL\ s\ v\\
&emb\ g\ p = Case\\
&\quad [\$(normal\ [|\ \lambda x\ y \to g\ x \equiv y\ |])\ \$\\
&\qquad \$\ (rearrV\ [|\ \lambda x \to ((), x)\ |])\ \$\\
&\qquad\quad Dep\ Skip\ (\lambda x\ () \to g\ x)\\
&\quad ,\$(adaptive\ [|\ \backslash\_\_ \to True\ |])\ p]
\end{aligned}
$$

Here what the normal branch does is, roughly speaking, leaving the source x as it is while ignoring the view, since we know that the view is necessarily $g\ x$. In order for an embedding to be well-behaved, running the *put* function should produce a source that when running *get* should return the view given to the former, as stated by the GETPUT law and enforced by the case structure. Furthermore, the view should be completely defined by the source. It is interesting to note that the two-branch structure of *emb* is comparable with that of $arAlignL'$: The normal branch of $arAlignL'$ deals with the case where the source and view are roughly consistent, i.e., aligned, but the elements are not yet completely synchronized pairwise; otherwise, when the source and view are too inconsistent, i.e., not aligned, the adaptive branch comes in and restores enough consistency such that the normal branch can take over. The normal branch of *emb*, on the other hand, applies when the source and view are fully consistent (as specified by g), and its adaptive branch restores full consistency (by p) when encountering inconsistent pairs of source and view. In short, *emb* is an extreme instance of the two-branch structure.

To run the delta alignment, we thus need to provide a delta to the BiGUL program. With the running example, we can use the following deltas:

$$
\begin{aligned}
&\delta_1, \delta_2, \delta_3 :: Delta\\
&\delta_1 = fromList\ [(0,0), (1,1), (2,2)]\\
&\delta_2 = fromList\ [(0,0), (1,2), (2,1)]\\
&\delta_3 = fromList\ [(0,0), (1,1)]
\end{aligned}
$$

For the *get* direction, the delta is ignored, and the result is the same as for the previous kinds of alignment:

```
> source
[(0,("A.",3)),(1,("B.",5)),(2,("C.",8))]
> get (arAlignL δ1) source
[(0,"A."),(1,"B."),(2,"C.")]
```

However, in the put direction, results may vary depending on the given delta, e.g., no changes are performed (using δ_1):

```
> put (arAlignL δ1) source ↩
    [(0,"A."),(1,"B."),(2,"C.")]
[(0,("A.",3)),(1,("B.",5)),(2,("C.",8))]
```

versus a swap between the last two elements (using δ_2):

```
> put (arAlignL δ2) source ↩
    [(0,"A."),(1,"B."),(2,"C.")]
[(0,("A.",3)),(1,("B.",8)),(2,("C.",5))]
```

Note that the elements were not swapped in the view, but the delta δ_2 indicates that the elements were swapped. This is equivalent to swapping those elements and modifying the values to the ones at the same position in the original view. A similar situation occurs when the view is not modified, but one element is not in the delta:

```
> put (arAlignL δ3) source ↩
    [(0,"A."),(1,"B."),(2,"C.")]
[(0,("A.",3)),(1,("B.",5)),(2,("C.",0))]
```

In this case, it is equivalent to remove the last element and inserting it again.

The delta alignment implementation can be generalized for arbitrary list contents, resulting in the following equivalent functions with additional parameters for the create function and BiGUL update program to apply to the elements:

```
adaptDeltaL :: (v → s) → [s] → [v] → Delta → [s]
alignL' :: BiGUL s v → (v → s)
        → BiGUL ([s], Delta) [v]
alignL :: Eq v ⇒ BiGUL s v → (v → s) → Delta
       → BiGUL [s] [v]
```

6. Delta-Based Tree Alignment

Lists are not the only structures capable of storing information, nor the only ones that are used in bidirectional transformation applications. However, other containers bring new challenges and we need to take them into account, e.g., the shape of the container. One such container where delta alignment can be implemented is the tree. Many kinds of trees exist, but we use a binary tree with labels in the nodes:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
  deriving (Show, Functor)
```

Tree elements can also be indexed by locations. The position of tree elements can be established linearly in an in-order fashion:

```
locsT :: Tree a → Tree Loc
locsT = fst ∘ aux 0
  where aux i0 Nil = (Nil, i0)
        aux i0 (Node _ l0 r0) =
          let (l, i1) = aux i0 l0
              (r, i ) = aux (i1 + 1) r0
          in (Node i1 l r, i)
flattenT :: Tree a → [a]
flattenT Nil = []
flattenT (Node a l r) = flattenT l ++ [a] ++ flattenT r
```

Thus the *Delta* type used for lists can also be used for trees, and the identity delta can be obtained with the function $getIdT :: Tree\ a \rightarrow Delta$.

The approach to implement the delta-based alignment for trees is similar to the approach used in the other implementations:

1. modification of the source aligning to the view using a delta;
2. a positional update.

The adaptation function for tree can be

```
arAdaptDeltaT :: Tree Author → Tree Researcher
              → Delta → Tree Author
arAdaptDeltaT s v d = fmap idOrCreate (locsT v)
  where idOrCreate i =
          let js = rngOf i d
          in if js ≢ ∅
             then flattenT s !! findMin js
             else let (k, v1) = flattenT v !! i
                  in (k, (v1, 0))
```

where we take advantage of the *fmap* function, deriving from the fact that *Tree* is a functor. This adaptation does two important tasks: it molds the source tree in order to match the shape of the view one; and, it aligns the elements in order to perform a positional update.

The implementation of the positional tree update is similar to the one for lists, since both have only two data constructors. However, trees have double recursion which must be taken into account.

```
arMapT :: BiGUL (Tree Author) (Tree Researcher)
arMapT = Case
  [$(normalSV [p | Nil |] [p | Nil |])
     ⟹ $(rearrV [| λNil → () |]) Skip
  ,$(adaptiveV [p | Nil |])
     ⟹ λ_ _ → Nil
  ,$(normalSV [p | Node _ _ _ |] [p | Node _ _ _ |])
     ⟹ $ (rearrV [| λ(Node v vl vr)
                    → (v, (vl, vr)) |]) $
          $ (rearrS [| λ(Node s sl sr)
                       → (s, (sl, sr)) |]) $
            arBX × (arMapT × arMapT)
  ,$(adaptiveV [p | (Node _ _ _) |])
     ⟹ λ_ (Node (k, v1) _ _) → Node (k, (v1, 0))
                                    Nil Nil
  ]
```

Before implementing the delta-based alignment for trees, we must revisit the function to check if the source and the view are aligned. For the list case, the shape is isomorphic to naturals and thus we have the shape of a list equal to its length. This aspect is taken into account when verifying the identity delta of the list with the given delta. However, for trees this is different and we thus need to include another condition:

```
isDeltaAlignedT (s, d) v = d ≡ getIdT v
                         ∧ d ≡ getIdT s
                         ∧ locsT s ≡ locsT v
```

Comparing the positions of the source tree and the view one we ensure that both have the same shape.

At this point, we can define a delta-based alignment for trees in a similar way as with lists:

```
arAlignT'
  :: BiGUL (Tree Author, Delta) (Tree Researcher)
arAlignT' = Case
  [$(normal [| isDeltaAlignedT |])
     ⟹ $(rearrS [| λ(s, _) → s |]) arMapT
  ,$(adaptiveS [| const True |])
     ⟹ λ(s, d) v → let s' = arAdaptDeltaT s v d
                   in (s', getIdT v)]
```

and corresponding wrapper:

```
arAlignT :: Delta
         → BiGUL (Tree Author) (Tree Researcher)
arAlignT d = emb g p
  where g s    = get arAlignT' (s, getIdT s)
        p s v = fst $ put arAlignT' (s, d) v
```

The application of *get* and *put* to trees is similar to the application of them to lists. The *get* functions takes the source tree and produces a view tree where its elements are the view of their correspondence in the source:

```
> get (arAlignT δ1) (Node (1,("B.",5)) ↩
      (Node (0,("A.",3)) Nil Nil) ↩
      (Node (2,("C.",8)) Nil Nil))
Node (1,"B.")  (Node (0,"A.") Nil Nil) ↩
               (Node (2,"C.") Nil Nil)
```

The delta specification in the *put* transformation is the same as with lists:

```
> put (arAlignT δ1) ↩
    (Node (1,("B.",5)) ↩
      (Node (0,("A.",3)) Nil Nil) ↩
```

```
      (Node (2,("C.",8)) Nil Nil)) ↩
    (Node (1,"B.") ↩
      (Node (0,"A.") Nil Nil) ↩
      (Node (2,"C.") Nil Nil))
Node (1,("B",5))  (Node (0,("A.",3)) Nil Nil) ↩
                  (Node (2,("C.",8)) Nil Nil)
```

We can also change the shape of a tree in the view:

```
> put (arAlignT δ1) ↩
    (Node (1,("B.",5)) ↩
      (Node (0,("A.",3)) Nil Nil) ↩
      (Node (2,("C.",8)) Nil Nil)) ↩
    (Node (0,"A.") Nil ↩
      (Node (1,"B.") Nil ↩
        (Node (2,"C.") Nil Nil)))
Node (0,"A.",3) Nil ↩
  (Node (1,"B.",5) Nil ↩
    (Node (2,"C.",8) Nil Nil))
```

The delta alignment implementation can be generalized for arbitrary tree contents, with the following equivalent functions. Similarly to the list version, the functions are parametrized with a *create* function and a BiGUL program to apply to the specific elements.

$$
\begin{aligned}
&mapT :: (v \to s) \to BiGUL\ s\ v \\
&\qquad \to BiGUL\ (Tree\ s)\ (Tree\ v) \\
&adaptDeltaT :: (v \to s) \to Tree\ s \to Tree\ v \to Delta \\
&\qquad \to Tree\ s \\
&alignT' :: \quad BiGUL\ a\ b \to (b \to a) \\
&\qquad \to \quad BiGUL\ (Tree\ a, Delta)\ (Tree\ b) \\
&alignT :: Eq\ v \Rightarrow BiGUL\ s\ v \to (v \to s) \to Delta \\
&\qquad \to BiGUL\ (Tree\ s)\ (Tree\ v)
\end{aligned}
$$

7. Generic Delta-Based Alignment

We have explained how to do delta-based alignment for lists and then for trees, and these experiences actually make us sufficiently prepared to go generic! In this final installment, we show that delta-based alignment can be generically implemented for any other containers.

7.1 Containers as Shape and Data

Pacheco et al. [18] rely on types with explicit notion of shape and data in their delta-alignment over inductive types, a property provided by polymorphic data types in functional programming. Moreover, they apply a notation from *shapely types* [15] in order to have tools to work with these data types. Employing these concepts, one can abstract from the shapes of both source and view, and just take the data into account for the alignment process.

Thus, a polymorphic type $T\ a$ can be characterized by three functions: $shape :: T\ a \to T\ ()$ to extract the shape; $data\_ :: T\ a \to [a]$ to extract the data; and, $recover :: (T\ (), [a]) \to T\ a$ to rebuild the type value from its shape and data. For a list of researchers `l = [(0,"A."),(1,"B."),(2,"C.")]` we have:

```
> shape l
[(),(),()]
> data_ l
[(0,"A."),(1,"B."),(2,"C.")]
> recover (shape l, data_ l)
[(0,"A."),(1,"B."),(2,"C.")]
```

and for a tree of researchers `t = Node (1,"B.")(Node (0,"A.")Nil Nil)(Node (2,"C.")Nil Nil)`:

```
> shape t
Node () (Node () Nil Nil) (Node () Nil Nil)
> data_ t
[(0,"A."),(1,"B."),(2,"C.")]
> recover (shape t, data_ t)
Node (1,"B.") (Node (0,"A.") Nil Nil) ↩
              (Node (2,"C.") Nil Nil)
```

For flexibility, these functions are defined in a type class

$$
\begin{aligned}
&\textbf{class}\ Shapely\ (t :: * \to *)\ \textbf{where} \\
&\quad shape :: t\ a \to t\ () \\
&\quad data\_ :: t\ a \to [a] \\
&\quad recover :: (t\ (), [a]) \to t\ a
\end{aligned}
$$

On top of these functions, it is possible to define new ones, e.g., $locs :: T\ a \to Set\ Loc$ to get all the locations of the data elements within the container.

7.2 Positional Mapping

The positional update is one of the aspects that is specific to each data type. To solve this issue in a simple manner, we introduce a new type class

$$
\begin{aligned}
&\textbf{class}\ Shapely\ t \Rightarrow Positional\ t\ \textbf{where} \\
&\quad positionalMap :: (v \to s) \to BiGUL\ s\ v \\
&\qquad \to BiGUL\ (t\ s)\ (t\ v)
\end{aligned}
$$

where the *positionalMap* function maps a BiGUL program element-wise. For the list container $positionalMap = mapL$ and for the tree container $positionalMap = mapT$.

7.3 Generic Delta Alignment

A key component in delta alignment is the position of elements. Having access to element positions, we can obtain the identity delta:

$$
\begin{aligned}
&getId :: Shapely\ s \Rightarrow s\ a \to Delta \\
&getId = map\ (\lambda l \to (l, l)) \circ locs
\end{aligned}
$$

Firstly, we need a generalization of the adaptation process. With lists, we take the view and then insert the source elements at the correct positions. For trees, the process is similar: we flatten the source tree extracting its data in order to insert it into the shape of the view. Working with shapely types, this is achieved by recovering a container with the shape of the view, but with the data of the original source or with created data when new elements were added:

$$
\begin{aligned}
&adaptDelta :: Shapely\ s \\
&\qquad \Rightarrow (b \to a) \to s\ a \to s\ b \to Delta \to s\ a \\
&adaptDelta\ c\ s\ v\ d = recover\ (newShape, newData) \\
&\quad \textbf{where}\ newShape = shape\ v \\
&\qquad newData = map\ idOrCreate\ (elems\ \$\ locs\ v) \\
&\qquad idOrCreate\ i = \textbf{let}\ js = rngOf\ i\ d \\
&\qquad\qquad \textbf{in if}\ js \not\equiv \emptyset \\
&\qquad\qquad\quad \textbf{then}\ data\_\ s\ !!\ findMin\ js \\
&\qquad\qquad\quad \textbf{else}\ c\ (data\_\ v\ !!\ i)
\end{aligned}
$$

With this function, any shapely type can be adapted, including lists and trees.

Another aspect to take into account is to check if the source and view containers are aligned. With lists we just compare the given delta with the identity deltas of source and view. With trees, in addition to the conditions used with lists, we also check the extracted locations. Actually, the extracted locations contains the shape of the tree, which is what we need in order to verify that some change was performed to the view in addition to the delta. Thus, we check if the source and view are aligned with:

$$
\begin{aligned}
&isDeltaAligned\ (s, d)\ v = d \equiv getIdT\ v \\
&\qquad\qquad\qquad\qquad \wedge\ d \equiv getIdT\ s \\
&\qquad\qquad\qquad\qquad \wedge\ shape\ s \equiv shape\ v
\end{aligned}
$$

and use it directly in the alignment function where we perform a positional update when both source and view are aligned, or we adapt when they are not aligned:

```
align' :: (Shapely t, Positional t)
       ⇒ BiGUL s v → (v → s)
       → BiGUL (t s, Delta) (t v)
align' b c = Case
  [$(normal [| isDeltaAligned |])
     ⟹ $(rearrS [| λ(s, _) → s |]) (positionalMap c b)
  ,$(adaptiveS [| const True |])
     ⟹ λ(s, d) v → let s' = adaptDelta c s v d
                   in (s', getId v)]

align :: (Shapely t, Positional t, Eq (t v))
      ⇒ BiGUL s v → (v → s) → Delta
      → BiGUL (t s) (t v)
align b c d = emb g p
  where g s    = get (align' b c) (s, getId s)
        p s v = fst $ put (align' b c) (s, d) v
```

7.4 Other Matching Algorithms Built Upon Deltas

With the implementation of delta-based alignment, we can implement other alignment strategies upon the deltas without much work. It is possible to make minor changes to the *align* function to implement other kinds of alignments, e.g., key-based:

```
keyAlign :: (Shapely s, Positional s, Eq (s b), Eq b, Eq k)
   ⇒ BiGUL a b → (b → a) → (a → k) → (b → k)
   → BiGUL (s a) (s b)
keyAlign b c sk vk = emb g p
  where g s    = get (align' b c) (s, getId s)
        p s v = fst $ put (align' b c)
                          (s, keyDelta sk vk s v) v
```

The *keyAlign* function, instead of receiving the delta, receives two functions to get the key component of the view and the source, respectively. Then, using the original source and the modified view, another function is used to infer a delta:

```
keyDelta :: (Shapely s, Eq k)
   ⇒ (a → k) → (b → k) → s a → s b → Delta
keyDelta sk vk ss vs = [(sp, vp) | (s, sp) ← sps
                                 , (v, vp) ← vps
                                 , sk s ≡ vk v]
  where sps = zip (data_ ss) (elems $ locs ss)
        vps = zip (data_ vs) (elems $ locs vs)
```

It is then possible to apply key-based alignment on any structure that has an implementation of delta-based alignment. The same function can be used for, e.g., lists:

```
> put (keyAlign arBX arCreate fst fst) ↩
    [(0,("A.",3)),(1,("B.",5)),(2,("C.",8))] ↩
    [(0,"A."),(1,"B."),(2,"C.")]
[(0,("A.",3)),(1,("B.",5)),(2,("C.",8))]
```

and for trees:

```
> put (keyAlign arBX arCreate fst fst) ↩
    (Node (1,("B.",5)) ↩
      (Node (0,("A.",3)) Nil Nil) ↩
      (Node (2,("C.",8)) Nil Nil)) ↩
    (Node (1,"B.") ↩
      (Node (0,"A.") Nil Nil) ↩
      (Node (2,"C.") Nil Nil))
Node (1,("B.",5)) (Node (0,("A.",3)) Nil Nil) ↩
                  (Node (2,("C.",8)) Nil Nil)
```

where $arCreate\ (k, v_1) = (k, (v_1, 0))$.

8. Conclusion

We hope to send the following two messages through this paper. One is that putback-based bidirectional programming is not that difficult in BiGUL, a simple but powerful putback-based language. The other is that a *single* well-designed putback-based bidirectional programming language can serve as the basis for developing many useful domain-specific bidirectional languages/libraries. We have shown that, with BiGUL alone, we can implement various alignment strategies which previously had to be implemented with separate bidirectional frameworks.

References

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

[2] D. M. J. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *15th ACM SIGPLAN international conference on Functional programming*, 2010.

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

[4] J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Towards a principle of least surprise for bidirectional transformations. In A. Cunha and E. Kindler, editors, *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, STAF 2015, L'Aquila, Italy, July 24, 2015.*, volume 1396 of *CEUR Workshop Proceedings*, pages 66–80. CEUR-WS.org, 2015. URL `http://ceur-ws.org/Vol-1396/p66-cheney.pdf`.

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

[6] U. Dayal and P. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7: 381–416, 1982.

[7] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, pages 304–318, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24484-1. URL `http://dl.acm.org/citation.cfm?id=2050655.2050685`.

[8] S. Fischer, Z. Hu, and H. Pacheco. A clear picture of lens laws - functional pearl. In R. Hinze and J. Voigtl"ander, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, K"onigswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 215–223. Springer, 2015. ISBN 978-3-319-19796-8.

[9] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), May 2007. ISSN 0164-0925. doi: 10.1145/1232420.1232424.

[10] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4): 486–524, 1988.

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

[12] M. Hofmann, B. Pierce, and D. Wagner. Edit lenses. In *POPL '12 Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

[13] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Dagstuhl Seminar on Bidirectional Transformations (BX). *SIGMOD Record*, 40(1):35–39, 2011.

[14] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume

8442 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2014. ISBN 978-3-319-06409-3.

[15] C. Jay. A semantics for shape. *Science of Computer Programming*, 25 (2-3):251–283, 1995. doi: 10.1016/0167-6423(95)00015-1. Selected Papers of ESOP'94, the 5th European Symposium on Programming.

[16] H.-S. Ko, T. Zan, and Z. Hu. Bigul: A formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2016, pages 61–72, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4097-7.

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

[18] H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. In *First International Workshop on Bidirectional Transformations*, 2012.

[19] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *PEPM '14*, pages 39–50. ACM, 2014.

[20] H. Pacheco, T. Zan, and Z. Hu. Biflux: A bidirectional functional update language for XML. In O. Chitil, A. King, and O. Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 147–158. ACM, 2014. ISBN 978-1-4503-2947-7. URL `http://dl.acm.org/citation.cfm?id=2643135`.