ISSN 1884-0760

GRACE TECHNICAL REPORTS

The Under-Appreciated Put: Implementing Delta-Alignment in BiGUL – Functional Pearl –

Jorge Mendes Hsiang-Shang Ko Zhenjiang HU

GRACE-TR 2016–03 April 2016

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

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, and 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.

1. Introduction

Bidirectional transformations are hot! They originated from the *view updating* mechanism in the database community [1, 6, 10], and have been recently 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 intentional update policies of put, in order to deal with various update policies of put in different contexts, significant extensions to the language for writing get are necessary. As a matter of fact, from the original get-based bidirectional language *lenses* [9], we have seen many such extensions, e.g., the *matching*

lenses to deal with alignment policies [2], the *delta lenses* to deal with modification-sensitive update policies [7, 12], and the *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. After a brief review of BiGUL [16], a putback-based bidirectional language, 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

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 $s \to v \to s$. 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

The simplest alignment strategy is the positional one. The following types for source ($Source$) and view ($View$) are used for the running example.

$$\begin{aligned}&\mathbf{type}\ Source = (Int, (Char, Int))\\ &\mathbf{type}\ View\ \ = (Int, Char)\end{aligned}$$

The first Int component of the pair should match the first Int component of the view, and the $Char$ component of the source should match the $Char$ component of the view. This relation between source and view can be expressed with the following BiGUL program:

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

Positional alignment for lists with elements of the above source and view types is pretty straightforward. No moves are 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:

- both source and view are empty, and we just $Skip$;
- all elements of the view were processed, so we adapt the source by removing the extra elements $\lambda\_\ \_ \to [\,]$;
- both source and view have elements, then we update with the head of both source and view, and then recurse $u \times myMapL\ c\ u$;
- the source does not have enough elements and create new ones $\lambda\_\ ((k, v_1) : \_) \to [(k, (v_1, 0))]$.

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

$$\begin{aligned}&myMapL :: BiGUL\ [Source]\ [View]\\ &myMapL = Case\\ &\quad [\$(normalSV\ [p\mid [\,]\mid]\ [p\mid [\,]\mid])\end{aligned}$$

```
    ⟹ $(rearrV [| λ[] → () |]) Skip
  , $(adaptiveV [p | [] |])
    ⟹ λ_ _ → []
  , $(normalSV [p | (_ : _) |] [p | (_ : _) |])
    ⟹ $(rearrV [| λ(v : vs) → (v, vs) |]) $
      $ (rearrS [| λ(s : ss) → (s, ss) |]) $
        myBX × myMapL
  , $(adaptiveV [p | (_ : _) |])
    ⟹ λ_ ((k, v1) : _) → [(k, (v1, 0))]
  ]
```

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.

Running the *get* function with this BiGUL program, we obtain the following result:

```
> get myMapL [(0,('a',0)),(1,('b',1)),(2,('c',2))]
[(0,'a'),(1,'b'),(2,'c')]
```

We can perform the changes that we want to this view, e.g., modify the characters to upper case, and put that view back into the original source[1]:

```
> put myMapL [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'A'),(1,'B'),(2,'C')]
[(0,('A',0)),(1,('B',1)),(2,('C',2))]
```

Moreover, we can see the limitations of positional update when removing an element `(1,'b')`:

```
> put myMapL [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'a'),(2,'c')]
[(0,('a',0)),(2,('c',1))]
```

or adding a new one `(3,'d')` before the end:

```
> put myMapL [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'a'),(1,'b'),(3,'d'),(2,'c')]
[(0,('a',0)),(1,('b',1)),(3,('d',2)),(2,('c',0))]
```

The *myMapL* 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, and with a BiGUL program to be run on the elements:

```
mapL :: (v → s) → BiGUL s v → BiGUL [s] [v]
```

4. Key-Based Alignment

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.

The idea to implement this strategy is to separate the program in two parts:

- alignment of the elements;
- the actual update.

[1] The symbol ↩ denotes line continuation.

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:

```
isAligned s v = length s ≡ length v
                  ∧ and (zipWith kmatch s v)
  where kmatch se ve = fst se ≡ fst ve
```

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 then they are 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:

```
keyMatchAdapt s v = map getSourceElement v
  where getSourceElement ve =
          case filter ((≡ fst ve) ∘ fst) s of
            [] → create ve
            (se : _) → se
        create (k, v1) = (k, (v1, 0))
```

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 a the following BiGUL program:

```
myKeyMatch :: BiGUL [Source] [View]
myKeyMatch = Case
  [$(normal [| isAligned |]) ⟹ myMapL
  , $(adaptive [| λ_ _ → True |]) ⟹ keyMatchAdapt]
```

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

```
> get myKeyMatch ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))]
[(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 myKeyMatch ↩
    [(0,('a',0)),(1,('b',1)), (2,('c',2))] ↩
    [(0,'A'),(1,'B'),(2,'C')]
[(0,('A',0)),(1,('B',1)),(2,('C',2))]
```

However, when removing elements `(1,'b')` or adding new ones `(3,'d')`, key-based alignment is more precise than positional:

```
> put myKeyMatch ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'a'),(2,'c'),(3,'d')]
[(0,('a',0)),(2,('c',2)),(3,('d',0))]
```

Nonetheless, key-based alignment also has its limitations, e.g., when modifying the key of an element (`(1,'b')` to `(3,'b')`):

```
> put myKeyMatch ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'a'),(3,'b'),(2,'c')]
[(0,('a',0)),(3,('b',0)),(2,('c',2))]
```

As with the positional update, this program can be generalized for key-based alignment on lists with arbitrary contents. For that, the *keyMatch* 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 and BiGUL update program as with *mapL*:

$$keyMatch :: Eq\ k \Rightarrow (s \to k) \to (b \to k) \to (v \to s) \to BiGUL\ s\ v \to BiGUL\ [s]\ [v]$$

5. Delta-Based List Alignment

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

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

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

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

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:

$$\delta \equiv getId\ artifact$$

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

$$getIdL :: [a] \to Delta$$
$$getIdL = map\ (\lambda l \to (l, l)) \circ locs$$

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. Implementing this in the running example:

```
myAlignL' :: BiGUL ([Source], Delta) [View]
myAlignL' = Case
  [$(normal [| λ(s, d) v → d ≡ getIdL v
                         ∧ d ≡ getIdL s |])
    ⟹ $(rearrS [| λ(s, _) → s |]) myMapL
  ,$(adaptiveS [| const True |])
    ⟹ λ(s, d) v → let s' = myAdaptDeltaL s v d
                  in (s', getIdL v)]
```

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:

```
myAdaptDeltaL :: [Source] → [View] → Delta
                 → [Source]
myAdaptDeltaL s v d =
  map idOrCreate (elems $ locs v)
  where idOrCreate i = let js = rngOf i d
                       in if js ≢ ∅
                          then s !! findMin js
                          else let (k, v1) = v !! i
                               in (k, (v1, 0))
```

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:

```
myAlignL :: Delta → BiGUL [Source] [View]
myAlignL d = emb g p
  where g s    = get myAlignL' (s, getIdL s)
        p s v = fst $ put myAlignL' (s, d) v
```

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:

```
put myAlignL' (s, getIdL v) (get myAlignL' (s, getIdL s))
 ≡ { getIdL v ≡ getIdL s }
put myAlignL' (s, getIdL s) (get myAlignL' (s, getIdL s))
 ≡ { GetPut for myAlignL' }
(s, getIdL s)
```

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 $(s', \delta') = put\ myAlignL'\ (s, \delta)\ v$, thus (s', δ') is consistent with v and $\delta' \equiv getIdL\ s'$. Applying the *get* function g:

```
get myAlignL' (s', getIdl s')
 ≡ { δ' = getIdL s' }
get myAlignL' (s', δ')
 ≡ { let binding }
get myAlignL' (put myAlignL' (s, δ))
 ≡ { PutGet for myAlignL' }
v
```

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

```
emb :: Eq v ⇒ (s → v) → (s → v → s) → BiGUL s v
emb g p = Case
  [$(normal [| λx y → g x ≡ y |]) $
     $ (rearrV [| λx → ((), x) |]) $
       Dep Skip (λx () → g x)
  ,$(adaptive [| \_ _ → True |]) p]
```

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.

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:

$$\delta_1, \delta_2, \delta_3 :: \mathit{Delta}$$
$$\delta_1 = \mathit{fromList}\ [(0,0),(1,1),(2,2)]$$
$$\delta_2 = \mathit{fromList}\ [(0,0),(1,2),(2,1)]$$
$$\delta_3 = \mathit{fromList}\ [(0,0),(1,1)]$$

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

```
> get (myAlignL δ1) ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))]
[(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 (myAlignL δ1) ↩
  [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
  [(0,'A'),(1,'B'),(2,'C')]
[(0,('A',0)),(1,('B',1)),(2,('C',2))]
```

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

```
> put (myAlignL δ2) ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'A'),(1,'B'),(2,'C')]
[(0,('A',0)),(1,('B',2)),(2,('C',1))]
```

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 (myAlignL δ3) ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'A'),(1,'B'),(2,'C')]
[(0,('A',0)),(1,('B',1)),(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:

$$\mathit{adaptDeltaL} :: (v \to s) \to [s] \to [v] \to \mathit{Delta} \to [s]$$
$$\mathit{alignL}' :: \mathit{BiGUL}\ s\ v \to (v \to s) \to \mathit{BiGUL}\ ([s], \mathit{Delta})\ [v]$$
$$\mathit{alignL} :: \mathit{Eq}\ v \Rightarrow \mathit{BiGUL}\ s\ v \to (v \to s) \to \mathit{Delta} \to \mathit{BiGUL}\ [s]\ [v]$$

6. Delta-Based Tree Alignment

Another container where delta alignment can be implemented is a tree. Many kinds of trees exist, but we use binary tree with labels in the nodes:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Nil} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\ \textbf{deriving}\ (\mathit{Show}, \mathit{Functor})$$

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

$$\begin{aligned}
&\mathit{locsT} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Loc}\\
&\mathit{locsT} = \mathit{fst} \circ \mathit{aux}\ 0\\
&\quad \textbf{where}\ \mathit{aux}\ i_0\ \mathit{Nil} = (\mathit{Nil}, i_0)\\
&\qquad \mathit{aux}\ i_0\ (\mathit{Node}\ _\ l_0\ r_0) =\\
&\qquad\quad \textbf{let}\ (l, i_1) = \mathit{aux}\ i_0\ l_0\\
&\qquad\qquad (r, i\ ) = \mathit{aux}\ (i_1 + 1)\ r_0\\
&\qquad\quad \textbf{in}\ (\mathit{Node}\ i_1\ l\ r, i)\\
&\mathit{flattenT} :: \mathit{Tree}\ a \to [a]\\
&\mathit{flattenT}\ \mathit{Nil} = []\\
&\mathit{flattenT}\ (\mathit{Node}\ a\ l\ r) = \mathit{flattenT}\ l +\!\!+ [a] +\!\!+ \mathit{flattenT}\ r
\end{aligned}$$

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

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

$$\begin{aligned}
&\mathit{myAdaptDeltaT} :: \mathit{Tree}\ \mathit{Source} \to \mathit{Tree}\ \mathit{View} \to \mathit{Delta}\\
&\qquad \to \mathit{Tree}\ \mathit{Source}\\
&\mathit{myAdaptDeltaT}\ s\ v\ d = \mathit{fmap}\ \mathit{idOrCreate}\ (\mathit{locsT}\ v)\\
&\quad \textbf{where}\ \mathit{idOrCreate}\ i =\\
&\qquad \textbf{let}\ \mathit{js} = \mathit{rngOf}\ i\ d\\
&\qquad \textbf{in}\ \ \textbf{if}\ \mathit{js} \not\equiv \emptyset\\
&\qquad\quad \textbf{then}\ \mathit{flattenT}\ s\ !!\ \mathit{findMin}\ \mathit{js}\\
&\qquad\quad \textbf{else let}\ (k, v_1) = \mathit{flattenT}\ v\ !!\ i\\
&\qquad\qquad \textbf{in}\ (k, (v_1, 0))
\end{aligned}$$

where we take advantage of the *fmap* function, deriving from the fact that *Tree* is a functor.

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.

$$\begin{aligned}
&\mathit{myMapT} :: \mathit{BiGUL}\ (\mathit{Tree}\ \mathit{Source})\ (\mathit{Tree}\ \mathit{View})\\
&\mathit{myMapT} = \mathit{Case}\\
&\quad [\$(\mathit{normalSV}\ [p \mid \mathit{Nil} \mid]\ [p \mid \mathit{Nil} \mid])\\
&\qquad \Longrightarrow \$(\mathit{rearrV}\ [\mid \lambda \mathit{Nil} \to () \mid])\ \mathit{Skip}\\
&\quad ,\$(\mathit{adaptiveV}\ [p \mid \mathit{Nil} \mid])\\
&\qquad \Longrightarrow \lambda_\ _ \to \mathit{Nil}\\
&\quad ,\$(\mathit{normalSV}\ [p \mid \mathit{Node}\ _\ _\ _ \mid]\ [p \mid \mathit{Node}\ _\ _\ _ \mid])\\
&\qquad \Longrightarrow\ \$\ (\mathit{rearrV}\ [\mid \lambda(\mathit{Node}\ v\ \mathit{vl}\ \mathit{vr})\\
&\qquad\qquad\qquad \to (v, (\mathit{vl}, \mathit{vr})) \mid])\ \$\\
&\qquad\quad \$\ (\mathit{rearrS}\ \ [\mid \lambda(\mathit{Node}\ s\ \mathit{sl}\ \mathit{sr})\\
&\qquad\qquad\qquad \to (s, (\mathit{sl}, \mathit{sr})) \mid])\ \$\\
&\qquad\qquad \mathit{myBX} \times (\mathit{myMapT} \times \mathit{myMapT})\\
&\quad ,\$(\mathit{adaptiveV}\ [p \mid (\mathit{Node}\ _\ _\ _) \mid])\\
&\qquad \Longrightarrow \lambda_\ (\mathit{Node}\ (k, v_1)\ _\ _) \to \mathit{Node}\ (k, (v_1, 0))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nil}\ \mathit{Nil}\\
&\quad]
\end{aligned}$$

Having the adaptation and the positional update, we can now define a delta-based alignment for trees in a similar way as with lists:

$$\begin{aligned}
&\mathit{myAlignT}' :: \mathit{BiGUL}\ (\mathit{Tree}\ \mathit{Source}, \mathit{Delta})\ (\mathit{Tree}\ \mathit{View})\\
&\mathit{myAlignT}' = \mathit{Case}\\
&\quad [\$(\mathit{normal}\ [\mid \lambda(s, d)\ v \to d \equiv \mathit{getIdT}\ v\\
&\qquad\qquad\qquad\qquad \wedge\ d \equiv \mathit{getIdT}\ s \mid])\\
&\qquad \Longrightarrow \$(\mathit{rearrS}\ [\mid \lambda(s, _) \to s \mid])\ \mathit{myMapT}\\
&\quad ,\$(\mathit{adaptiveS}\ [\mid \mathit{const}\ \mathit{True} \mid])\\
&\qquad \Longrightarrow \lambda(s, d)\ v \to \textbf{let}\ s' = \mathit{myAdaptDeltaT}\ s\ v\ d\\
&\qquad\qquad\qquad\qquad \textbf{in}\ (s', \mathit{getIdT}\ v)]
\end{aligned}$$

and corresponding wrapper:

$$\begin{aligned}
&\mathit{myAlignT} :: \mathit{Delta} \to \mathit{BiGUL}\ (\mathit{Tree}\ \mathit{Source})\ (\mathit{Tree}\ \mathit{View})\\
&\mathit{myAlignT}\ d = \mathit{emb}\ g\ p
\end{aligned}$$

$$\begin{aligned}&\textbf{where } g\ s \quad = get\ myAlignT'\ (s, getIdT\ s)\\ &\qquad p\ s\ v = fst\ \$\ put\ myAlignT'\ (s, d)\ v\end{aligned}$$

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 (myAlignT δ1) (Node (1,('b',1)) ↩
      (Node (0,('a',0)) Nil Nil) ↩
      (Node (2,('c',2)) 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 (myAlignT δ1) ↩
    (Node (1,('b',1)) ↩
      (Node (0,('a',0)) Nil Nil) ↩
      (Node (2,('c',2)) Nil Nil)) ↩
    (Node (1,'B') ↩
      (Node (0,'A') Nil Nil) ↩
      (Node (2,'C') Nil Nil))
Node (1,('B',1))  (Node (0,('A',0)) Nil Nil) ↩
                  (Node (2,('C',2)) 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

Delta-based alignment can also be implemented for other containers. The implementations for the list and tree cases are generalizable to 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 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 a 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 elementwise. 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:

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

Starting with the adaptation, we can make use of the functions resulting from the fact that we can see a container as a shape and data. Therefore, we recover 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.

$$\begin{aligned}&align' :: (Shapely\ t, Positional\ t)\\ &\qquad \Rightarrow BiGUL\ s\ v \to (v \to s)\\ &\qquad \to BiGUL\ (t\ s, Delta)\ (t\ v)\\ &align'\ b\ c = Case\\ &\quad [\$(normal\ [|\ \lambda(s, d)\ v \to d \equiv getId\ v\\ &\qquad\qquad\qquad \wedge\ d \equiv getId\ s\ |])\\ &\qquad \Longrightarrow \$(rearrS\ [|\ \lambda(s, \_) \to s\ |])\ (positionalMap\ c\ b)\\ &\quad , \$(adaptiveS\ [|\ const\ True\ |])\\ &\qquad \Longrightarrow \lambda(s, d)\ v \to \textbf{let } s' = adaptDelta\ c\ s\ v\ d\\ &\qquad\qquad\qquad \textbf{in } (s', getId\ v)]\end{aligned}$$

$$\begin{aligned}&align :: (Shapely\ t, Positional\ t, Eq\ (t\ v))\\ &\qquad \Rightarrow BiGUL\ s\ v \to (v \to s) \to Delta\\ &\qquad \to BiGUL\ (t\ s)\ (t\ v)\\ &align\ b\ c\ d = emb\ g\ p\\ &\quad \textbf{where } g\ s \quad = get\ (align'\ b\ c)\ (s, getId\ s)\\ &\qquad\quad p\ s\ v = fst\ \$\ put\ (align'\ b\ c)\ (s, d)\ v\end{aligned}$$

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:

$$\begin{aligned}&keyAlign :: (Shapely\ s, Positional\ s, Eq\ (s\ b), Eq\ b, Eq\ k)\\ &\quad \Rightarrow BiGUL\ a\ b \to (b \to a) \to (a \to k) \to (b \to k)\\ &\quad \to BiGUL\ (s\ a)\ (s\ b)\end{aligned}$$

$$
\begin{aligned}
&keyAlign\ b\ c\ sk\ vk = emb\ g\ p \\
&\quad \textbf{where}\ g\ s\ \ \ = get\ (align'\ b\ c)\ (s, getId\ s) \\
&\qquad\quad p\ s\ v = fst\ \$\ put\ (align'\ b\ c) \\
&\qquad\qquad\qquad\qquad (s, keyDelta\ sk\ vk\ s\ v)\ v
\end{aligned}
$$

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:

$$
\begin{aligned}
&keyDelta :: (Shapely\ s, Eq\ k) \\
&\quad \Rightarrow (a \to k) \to (b \to k) \to s\ a \to s\ b \to Delta \\
&keyDelta\ sk\ vk\ ss\ vs = [(sp, vp) \mid (s, sp) \leftarrow sps \\
&\qquad\qquad\qquad\qquad\qquad\qquad , (v, vp) \leftarrow vps \\
&\qquad\qquad\qquad\qquad\qquad\qquad , sk\ s \equiv vk\ v] \\
&\quad \textbf{where}\ sps = zip\ (data\_\ ss)\ (elems\ \$\ locs\ ss) \\
&\qquad\quad\ \ vps = zip\ (data\_\ vs)\ (elems\ \$\ locs\ vs)
\end{aligned}
$$

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 myBX myCreate fst fst) ↩
    [(0,('a',0)),(1,('b',1)),(2,('c',2))] ↩
    [(0,'A'),(1,'B'),(2,'C')]
[(0,('A',0)),(1,('B',1)),(2,('C',2))]
```

and for trees:

```
> put (keyAlign myBX myCreate fst fst) ↩
    (Node (1,('b',1)) ↩
      (Node (0,('a',0)) Nil Nil) ↩
      (Node (2,('c',2)) Nil Nil)) ↩
    (Node (1,'B') ↩
      (Node (0,'A') Nil Nil) ↩
      (Node (2,'C') Nil Nil))
Node (1,('B',1))  (Node (0,('A',0)) Nil Nil) ↩
                  (Node (2,('C',2)) Nil Nil)
```

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

8. Conclusion

We hope to send the following two messages through this paper. One is that putback-based programming is not that difficult in BiGUL, a simple but powerful put-based bidirectional language. The other is that a *single* well-designed putback-based bidirectional programming language can serve as basis for developing many useful domain-specific bidirectional languages/libraries.

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`.