# BRUL: A Putback-Based Bidirectional Transformation Library for Updatable Views

Tao Zan[1] Li Liu[2] Hsiang-Shang Ko[3] Zhenjiang Hu[1,3]

[1] SOKENDAI (The Graduate University for Advanced Studies), Japan
{zantao, hu}@nii.ac.jp
[2] Shanghai Jiao Tong University, China
lee.liu@sjtu.edu.cn
[3] National Institute of Informatics, Japan
hsiang-shang@nii.ac.jp

## Abstract

In work on relational databases, the view-update problem is about how to translate update operations on the view table to corresponding update operations on the source table properly. It is a problem that the translation policies are not unique in many situations. Relational lenses try to solve this problem by providing a list of combinators that let the user write *get* functions (queries) with specified updated policies for *put* functions (updates); however this can only provide limited control of update policies which still may not satisfy the user's real needs. In this paper, we implement a library BRUL that provides putback-based basic combinators for the user to write the *put* function with flexible update policies easily; from the *put* function, a unique *get* function can be derived automatically. BRUL is implemented in terms of BIGUL, a core bidirectional programming language which has been formalized in AGDA and implemented as a Haskell library.

## 1 Introduction

Bidirectional transformations, which originated from the *view-update* problem, provide a novel mechanism for maintaining consistency between two pieces of data. A bidirectional transformation consists of a pair of functions: a *get* function that extracts part of information from a source to construct a view, and a *put* function that accepts an updated view and the original source, resulting in an updated source that is consistent with the updated view. The pair of functions needs to satisfy the *well-behavedness* laws:

$$\begin{array}{lcll} put\ s\ (get\ s) & = & s & (GetPut) \\ get\ (put\ s\ v) & = & v & (PutGet) \end{array}$$

The *GetPut* law says that *put*ting an unmodified view *get s* directly back into the source $s$ should produce the same unmodified $s$, while the *PutGet* law says that *get*ting from an updated source computed by *put*ting a view $v$ should retrieve the same $v$.



In: A. Anjorin, J. Gibbons (eds.): Proceedings of the Fifth International Workshop on Bidirectional Transformations (Bx 2016), Eindhoven, The Netherlands, April 8, 2016, published at http://ceur-ws.org

| name | email | location |
|---|---|---|
| John | john@john.com | Tokyo |
| Mary | mary@mary.com | NewYork |
| Stan | stan@stan.com | Tokyo |

(a) Source table

| name | email | location |
|---|---|---|
| John | john@john.com | Tokyo |
| Stan | stan@stan.com | Tokyo |

(b) View table

Figure 1: Updated view and source table

| name | email | location |
|---|---|---|
| Stan | stan@mary.com | Tokyo |
| Jeff | jeff@jeff.com | Tokyo |

(a) Updated view table

| name | email | location |
|---|---|---|
| Mary | mary@mary.com | NewYork |
| Stan | stan@mary.com | Tokyo |
| Jeff | jeff@jeff.com | Tokyo |

(b) Updated source table

Figure 2: Updated view and source table

*Lenses* [6] were proposed to facilitate construction of bidirectional transformations. A lens is a well-behaved pair of *get* and *put* functions, and one can provide combinators to compose smaller lenses into larger ones. While lenses can be executed either as *get* or *put*, they are usually designed so that the programmers can construct lenses as if they are writing unidirectional *get* programs. For relational databases, Bohannon et al. designed *relational lenses* [3], whose *get* functions are database queries and *put* functions reflect updates on query results back to databases. Specifically, they designed three basic lenses, called *selection*, *drop* and *join*. The *get* semantics for *selection* and *join* are the same as the ones in SQL, while the *get* semantics of *drop* removes one column from the source. Their *put* semantics are carefully formalized in order to satisfy the *well-behavedness* properties. The lenses can be composed in order to create larger programs. Relational lenses have a SQL-like syntax, which let the programmers simply write a bidirectional program as a SQL program. This *get*-based design (meaning that the lens programs look like *get* functions) reduces the burden of writing bidirectional programs, but the *put* behavior is not controlled by the programmers, and may not satisfy their real needs.

To explain the problem, let us look at an example: suppose that a company keeps a table of employee information shown in Figure 1a, in which each record gives a person's name, email address, and current location. We can write a program with relational lenses as follows :

```
select from s where location = "Tokyo" as v
```

It looks exactly the same as writing a SQL query that selects only those people located in Tokyo. Running the *get* direction of this program on the source table in Figure 1a yields the view table shown in Figure 1b. Now we do some modifications on the view table: We delete the person John because he moved to another city Kyoto, insert a new person Jeff, and update Stan's email address. After that, if we run the *put* direction of this program, the source table will be updated by deleting John, inserting Jeff, and updating Stan's email address with the one in the view, while Mary is left unchanged. The resulting table is shown in Figure 2b.

This backward behavior is acceptable in some cases, but it is not what we want here. What we want is updating John's location instead of deleting him, since he just moved to Kyoto and still belongs to the company. Relational lenses cannot describe this behavior as this is more about controlling update policies in the *put* direction. The *put* behavior provided by relational lenses is well-behaved, but the programmers are not given the freedom to customize that behavior.

In order to give the programmers more control over the *put* behavior, we provide a new library Brul (bidirectional relational update library) that follows our previous work [18, 12] on *putback*-based bidirectional programming. Instead of letting the programmers write the forward *get* function and implicitly deriving a backward *put* which might not satisfy the programmers' real needs, we carefully define the library in a way that allows the programmers to specify the update policies flexibly as a *put* function, from which the necessarily unique forward transformation is derived. Our contributions can be summarized as follows:

- We design a library Brul that provides basic *put* combinators that can 1) let the programmers directly describe the *put* behavior, and 2) give the programmers full control of the bidirectional behavior of their programs, since the *put* behavior uniquely determines the *get* behavior [5].
- Brul is implemented on top of BiGUL [12], which is a fully formalized putback-based language. The

```
data BiGUL :: * -> * -> * where
  Fail :: String -> BiGUL s v
  Skip :: BiGUL s ()
  Replace :: BiGUL s s
  Prod :: BiGUL s v -> BiGUL s' v' -> BiGUL (s, s') (v, v')
  RearrS :: Lambda s s' -> BiGUL s' v -> BiGUL s v
  RearrV :: Lambda v v' -> BiGUL s v' -> BiGUL s v
  Case :: [(s -> v -> Bool, CaseBranch s v)] -> BiGUL s v
  Compose :: BiGUL u v -> BiGUL s u -> BiGUL s v

data CaseBranch s v = Normal (BiGUL s v) (s -> Bool)
                    | Adaptive (s -> v -> s)
```

Figure 3: BiGUL language

well-behavedness of Brul can thus be easily proved. The source code of the implementation is available at the Brul website [1].

- We demonstrate Brul's expressiveness by encoding in Brul all the operations of relational lenses and giving examples of more flexible *put*s that cannot be described with relational lenses.

The rest of the paper is organized as follows: Section 2 briefly introduces the BiGUL language, Section 3 shows the design of the Brul library, Section 4 explains the implementation of Brul in detail, Section 5 gives two typical examples which are described in the relational lenses paper, Section 6 reviews the related work in both relational database and bidirectional transformation, and Section 7 gives a conclusion of our work.

## 2 Core language BiGUL

Brul is a library built on the BiGUL language. In order to let the reader understand how Brul is implemented, we will introduce the basic operators of BiGUL and also present a series of simple examples in this section.

BiGUL is a putback-based bidirectional core language that provides a set of basic combinators for describing the *put* functions of bidirectional transformations. The well-behavedness of BiGUL is fully verified in Agda [13]; for use in practical applications, BiGUL is ported to Haskell as an embedded domain-specific language (shown in Figure 3), which we will use in this paper. A BiGUL program can be evaluated as either a *put* or a *get*. That is, there are two interpreters for BiGUL programs:

```
put :: BiGUL s v -> s -> v -> Either ErrorInfo s
get :: BiGUL s v -> s -> Either ErrorInfo v
```

The bidirectional transformations described by BiGUL programs are potentially partial, and hence the results of these two interpreters are wrapped in the `Either` monad — a value returned by *put* or *get* is either `Left errMsg` for some error message `errMsg` (of type `ErrorInfo`) or `Right result` for some successfully computed `result`. The well-behavedness laws are accordingly revised to

$$\texttt{get b s} = \texttt{Right v} \;\Rightarrow\; \texttt{put b s v} = \texttt{Right s} \quad (GetPut)$$
$$\texttt{put b s v} = \texttt{Right s}' \;\Rightarrow\; \texttt{get b s}' = \texttt{Right v} \quad (PutGet)$$

These are the well-behavedness properties verified in Agda.

Let us briefly explain the combinators in Figure 3. The three primitive operations are `Fail`, `Skip`, and `Replace`: `Fail` always fails to compute (i.e., returns a `Left`-value) when interpreted as either *put* or *get*. The *put* behavior of `Skip` returns the unchanged original source, while its *get* interpretation returns an empty view — note that the view type specified for `Skip` is `()`. `Replace` works on sources and views of the same type; its *put* interpretation replaces the whole source with the view, while its *get* interpretation returns the whole source. The *put* behavior of the `Prod` combinator uses two BiGUL sub-programs to update a source pair by a view pair in parallel, while its *get* interpretation extracts a view pair from a source pair again by using the two sub-programs. `RearrS` updates a source $s$ using view $v$ by first computing a (usually) smaller source $s'$

[1] http://www.prg.nii.ac.jp/project/brul

with a simple $\lambda$-expression (given as the first argument of `RearrS`), then using a BiGUL sub-program to update the source $s'$ with $v$, and finally putting $s'$ back into $s$ by "inverting" the $\lambda$-expression; its *get* semantics again computes a smaller source from which the resulting view is extracted. `RearrV` is dual to `RearrS`, acting on the view instead of the source. `Case` does case analysis on both the source and the view, and performs either a normal BiGUL operation to update the source using the view or an adaptive operation to change the source to a new one. Since it is quite complicated, we will use an example to explain this in detail below. `Compose` concatenates BiGUL programs sequentially.

In the following, we will use BiGUL to write several programs to give the reader a better understanding of how BiGUL works. Suppose that we have a source `(1, "Tokyo")` and a view `"Tokyo"` which is the second component of the source. If we modify the view into `"NewYork"`, then the source and view become inconsistent. We can write a simple BiGUL program *bigul1* to update the source with the view:

```
bigul1 :: BiGUL (a,b) b
bigul1 = RearrV (Lambda RVar (EProd (EConst ()) (EDir (DRight DVar)))) (Prod Skip Replace)
```

Note that the first argument of `RearrV` is a deeply (syntactically) represented $\lambda$-expression `\v -> ((), v)`. We do not actually need to present the definition of `Lambda`, since in the rest of this paper we will use a more readable surface syntax (implemented with Template Haskell [19]):

```
bigul1' :: BiGUL (a,b) b
bigul1' = RearrV [|\v -> ((), v)|] (Prod Skip Replace)
```

*bigul1′* is expanded into *bigul1* at compile-time. It first rearranges a view $v$ to $((), v)$ (as specified by the Template Haskell code `[|\v -> ((), v)|]`). Then, the first element of the source pair is skipped, and the second element of the source pair is replaced by $v$ which is the second element of the rearranged view pair. If we execute the *put* direction of the program with the above source pair and modified view, the source will be updated to `(1, "NewYork")`, and if we execute the *get* direction of the program on this updated source, it will output `"NewYork"`.

The `Case` operator is heavily used in the implementation of Brul. The most distinctive feature of `Case` is source adaptation, which is useful when the source is not in a suitable shape to be updated with the view, and needs to be modified through an adaptation function to a new source to make it updatable. After adapting to a new source, the whole `Case` statement is run again on the new source and the view. To ensure termination, this second run should not match with any adaptive cases again.

Let us use another example to show how to use the `Case` operator, in particular the source adaptation mechanism. Suppose that the source $s$ is a list and we want to put a value $v$ into the $i$-th place in the source list. We can write a bidirectional program in BiGUL as follows:

```
embedAt :: Int -> BiGUL [a] a
embedAt i = Case
  [(\s _ -> i == 0 && not (null s), Normal (RearrS [|\(x:xs) -> (x, xs)|]
                          (RearrV [|\v -> (v, ())|] (Prod Replace Skip)))),
   (\s _ -> i >= 0 && null s, Adaptive (\_ _ -> [-1])),
   (\_ _ -> i > 0, Normal (RearrS [|\(x:xs) -> (x, xs)|] (Prod Skip (embedAt (i-1)))))
  ]
```

We can call *put* (*embedAt i*) $s$ $v$ to achieve our goal of embedding, and *get* (*embedAt i*) $s$ to retrieve the value from source. The implementation of *embedAt* is easy to understand: when $i$ is greater than zero and source is not null (the last branch), it leaves the first element of source unchanged and calls the *embedAt* function recursively with $i - 1$; when $i$ is greater than zero and source is null (the middle branch), the source is adapted to a non-empty list by changing it to a one-element list with a default value $-1$; when $i$ is zero and the source is not null (the first branch), then the first element of the source will be replaced with the view value, and the rest of the list will be skipped.

## 3 Brul library

Unlike relational lenses [3] where a fixed putback semantics (update policy) is preset to the forward query with three relational operators (select, projection, join), we propose a new library Brul, where two *putback*-based combinators (operators) are designed to specify update policies, from which forward queries can be automatically

derived. Two distinguished features of Brul are (1) it is powerful to be used to specify various update policies (put), and (2) update policies written in Brul are guaranteed to be well-behaved as a consequence of the formalization of BiGUL.

In the following, we will explain the Brul library functions in detail. The basic library functions are: *1*) `align`, which is used to update a source list with a view list by aligning part of source elements filtered by a predicate with view elements according to a matching criteria between source element and view element; *2*) `unjoin`, which is used to decompose a join view to update two sources. The first argument defines the update policy for deletion on the two sources when there is deletion on the view, and the remaining three arguments specify the common attributes for the two sources and the view. Note that lists are used to represent tables in this paper.

### 3.1 Align

The function `align`:

```
align :: (s -> Bool) -> (s -> v -> Bool) ->
        -> BiGUL s v {* Case 1: matched *}
        -> (v -> s) {* Case 2: missing source element *}
        -> (s -> Maybe s) {* Case 3: missing view element *}
        -> Brul [s] [v]
```

describes an update on a source which is a list of records with type `s` using a view which a list of records with type `v`. It starts by using the first argument (a source filter function (`s -> Bool`)) to extract the satisfied source records, and then uses the second argument (a matching function (`s -> v -> Bool`)) to match these source elements with the view elements. The matching result has three cases, and each case uses different update operation: when source and view elements are matched, the third argument (a BiGUL program (`BiGUL s v`)) is used to update the source element by the view element; when a view element has no corresponding matching source element, the fourth argument (a user-defined function (`v -> s`)) is used to create a source element from this view element; when a source element has no corresponding matching view element, the fifth argument (a user-defined *conceal* function (`s -> Maybe s`)) is used to conceal the element (from the view) by either deleting this source element or modifying it so that it does not satisfy the filter condition. Note that the type of `Brul s v` is similar with `BiGUL s v` except that it can call a function to correct the source of type $s$ if it is inconsistent (say, not satisfying necessary functional dependencies if the source is a table).

To see an example, consider the updating strategy given by relational lenses in the introduction. Using `align`, we can specify it as

```
uEmployee :: Brul [Record] [Record]
uEmployee = align (\r -> (r!!2) == RString "Tokyo") (\s v -> (s!!0 == v!!0))
                Replace
                id
                (\_ -> Nothing)
```

Tables are represented as a list of records of type `Record`, which is defined in Section 4. The Brul program `uEmployee` — whose source and view types are specified as `[Record]` — matches by name the source records whose location is Tokyo with the view records. If they are matched, the source record is replaced by the view record; for unmatched view record, create a new source record using `id` function since source and view records are the same; for unmatched source record, it will be deleted in the source.

The update policy that cannot be dealt with by relational lenses in the introduction can now be easily obtained by changing the definition of the last function for `uEmployee`:

```
uEmployee' :: Brul [Record] [Record]
uEmployee' = align (\r -> (r!!2) == RString "Tokyo") (\s v -> (s!!0 == v!!0))
                Replace
                id
                (\[name, email, location] -> [name, email, RString "Kyoto"])
```

In the last line, an unmatched source record is modified by changing the location to Kyoto.

Returning to the example in the introduction, if we wish to adopt the update strategy of deleting John in the source, we may use `uEmployee`; if we wish to adopt the strategy of moving John to Kyoto, we may use

| A | B |
|---|---|
| a1 | b1 |
| a2 | b2 |
| a3 | b3 |
| a5 | b5 |

(a) Source table I

| B | C |
|---|---|
| b1 | c1 |
| b3 | c3 |
| b4 | c4 |
| b5 | c5 |

(b) Source table II

| A | B | C |
|---|---|---|
| a1 | b1 | c1 |
| a5 | b5 | c5 |

(c) Updated view table

Figure 4: Join example I

| A | B |
|---|---|
| a1 | b1 |
| a2 | b1 |

(a) Source table I

| B | C |
|---|---|
| b1 | c1 |
| b1 | c2 |

(b) Source table II

| A | B | C |
|---|---|---|
| a1 | b1 | c1 |
| a1 | b1 | c2 |
| a2 | b1 | c1 |
| a2 | b1 | c2 |

(c) View table

Figure 5: Join example II

`uEmployee'`. It is worth noting that these two BRUL programs have the same *get* behavior as one written in relational lenses.

### 3.2 Unjoin

The type signature of function `align` is:

```
unjoin :: DeleteFlag -> (s1 -> a) -> (s2 -> a) -> (v -> a) -> Brul ([s1],[s2]) [v]
```

The library function `unjoin delStrategy extracts1 extracts2 extractv` specifies how to update two source tables using a view table, the join result of the two source tables, when the view table is modified, based on a deletion policy `delStrategy` and three functions `extracts1, extracts2, extractv` for extracting the common parts from these three tables. The deletion policy is used to describe the deletion strategy on the source tables when some records are deleted on the view table. We offer three kinds of deletion policies when deletions on both source tables are allowed: deleting records only on the left source table (`DeleteLeft`), deleting records only on the right source table (`DeleteRight`), and deleting records on both tables (`DeleteBoth`). The remaining three arguments are functions with the same output type, since the result is the common part of the three tables.

As an example, consider the two source tables shown in Figure 4a and Figure 4b, each with four records. Consider the view table in Figure 4c, the join of these two source tables by attribute B, being updated by deleting record (a3, b3, c3). We can write a simple program in BRUL to update the source tables as follows:

```
uss1 = unjoin DeleteLeft fs1 fs2 fv
  where fs1 = \[a,b] -> [b]
        fs2 = \[b,c]-> [b]
        fv  = \[a,b,c] -> [b]
```

Here, we use the deletion policy `DeleteLeft`. Function `fs1` extracts the value of attribute B from a record in source table I, and `fs2` and `fv` extract the same attribute value from source table I and view table respectively. With `uss1`, deletion of record (a3, b3, c3) from the view will lead to deletion of record (a3, b3) from source table I, while source table II remains unchanged. If we choose `DeleteBoth` as the update policy as in the following `uss2`:

```
uss2 = unjoin DeleteBoth fs1 fs2 fv
```

the same deletion from the view will lead to deletion of record (a3, b3) from source table I and that of (b3, c3) from source table II.

Notice that `unjoin` is more general and powerful than the join lenses (i.e. *join_dl*, *join_dr* etc.) in relational lenses, where the join lenses have the restriction that the join attribute should be the key of the right table. We do not have this restriction, allowing arbitrary join attribute. This, however, introduces challenges in implementation, which will be given in Section 4.3. To see this, let us consider the tables in Figure 5. Each

of the two source tables has two records, and joining these two table by attribute B (which is not the key of source table II) yields a view table with four records as shown in Figure 5c. It is important to see that arbitrary updates on view are not necessarily valid if B is not the key of source table II. For example, if we delete (a1, b1, c1) from the view table, there is no correct way of updating the two source tables. If we would delete (a1, b1) from source table I, then joining the updated source table I with source table II would remove another record (a1, b1, c2) from the view table; if we would delete (b1, c1) from source table II, record (a2, b1, c1) would be removed from the view. But if we give a valid update on the view, say deleting (a1, b1, c1) and (a1, b1, c2) from the view table, there is one possible update strategy that is to delete record (a1, b1) from source table I. The good news is that programs written in BRUL (both `uss1` and `uss2`) can correctly identify the invalid cases to report dynamic errors while allowing valid view updates.

# 4 BRUL implementation over BIGUL

In this section, we show how BRUL is implemented using BIGUL to enjoy the well-behavedness of BIGUL. The *align* function is implemented basically in `Case`; The *unjoin* function is composed by three sub BIGUL programs. Some details are tedious and complex, so we just illustrate them simply.

## 4.1 Relational database representation

Before explaining the implementation of two libary functions on a relational database (i.e. table), we briefly explain how tables are represented in our context. A table is a list of records (`[Record]`), and each record is a list of attribute elements of type `RType`:

```
type Record = [RType]
data RType = RInt Int
           | RString String
           | RFloat Float
           | RDouble Double
```

For example, the source table in Figure 1a is represented as follows.

```
[[RString "John", RString "john@john.com", RString "Tokyo"],
 [RString "Mary", RString "mary@mary.com", RString "NewYork"],
 [RString "Stan", RString "stan@stan.com", RString "Tokyo"]]
```

A table may have functional dependencies. In the above example, the second attribute (email) may depend on the first (name). We use `FDMap` to store such functional dependencies of a table:

```
type FDMap = Map Int [Int]
```

which maps from one attribute to a list of attributes that depend on it. Here each attribute is represented by the index in the record list. More than one attribute may depend on the same attribute, and one attribute may depend on more than one attribute. For simplicity, in our example, we do not consider the case where one attribute depends on more than one attribute.

## 4.2 Align

The *align* function is implemented using the `Case` operator in BIGUL. As discussed in Section 2, `Case` provides a flexible way to do case analysis on both the source and the view, and perform either a normal BIGUL operation to update the source using the view or an adaptive operation to change the source to a new one. Six cases should be considered for implementing

$$align\ p\ m\ b\ c\ d$$

using a `Case`, where $p$ is a predicate for the filter function, $m$ is a matching function, $b$ is a BIGUL program to do updating when the source and the view are matched, $c$ is a source create function, $d$ is a *conceal* function. We use $fd$ to represent the function for updating the source record according to the functional dependency, which can be automatically generated from the functional dependencies on the database. In the following, we give an intuitive explanation for each case analysis in this `Case`; the detailed implementation is available in the BRUL website.

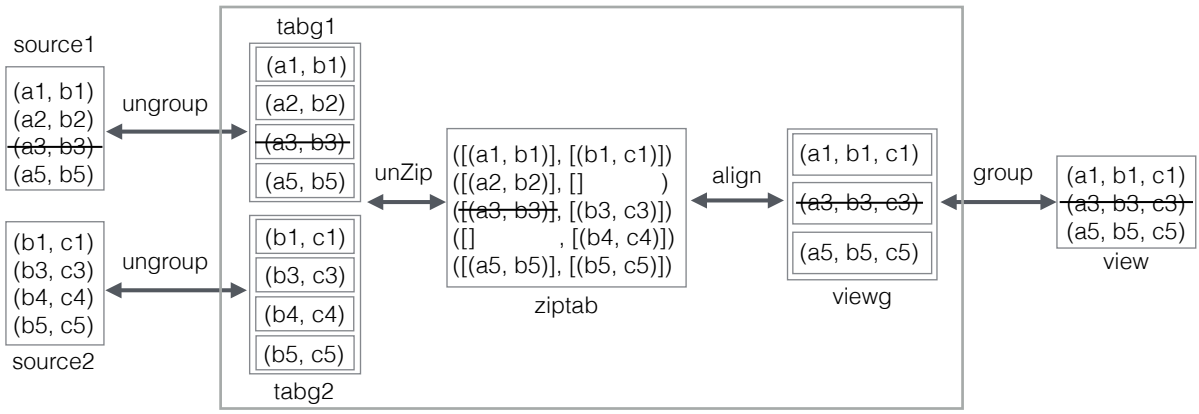


Figure 6: Implementation sketch for `unjoin`

1. When the view is empty and the source meets the requirement of functional dependences and there is no record in the source satisfying $p$, we keep the source and make no change. The condition for this case can be written in Haskell as `\ss vv -> null vv && null (filter p ss) && map fd ss == ss`.

2. When the view is empty but the source does not satisfy the condition in the first case, we perform an adaption, where we update each record of the source by function *fd* and conceal those updated records that satisfy the filter condition $p$. Note that according to the semantics of `Case`, the adapted source will be fed into this `Case` again to match other cases.

3. When the first record of the source does not satisfy the filter condition $p$ and also does not meet the requirement of functional dependencies, the source is adapted by updating the first record using function *fd*. We will check the adapted record shall still not satisfy the filter condition $p$.

4. When the first record of the source does not satisfy the filter condition $p$ but follows the functional dependency, we keep the first record and update the rest part of the source with current view by recursively calling the *align* function.

5. When the first record of the source satisfies $p$ and matches with the first element of the view, we use the given function $b$ to update the matched source record, and recursively update the rest part of the source with the rest of the view. The condition is:

   ```
   \ss vs -> not (null (filter p ss)) && p (head ss)
         && not (null vs) && m (head ss) (head vs)
   ```

6. Otherwise, we do adaption on the source. In this case, both the source satisfying $p$ and the view should not be empty. The first record of the view is used to find a corresponding element in the source according to the matching function $m$. If we find one, we move this record to the head of the source; otherwise, we use the create function $v$ to create a new head record of the source based on the view record.

### 4.3 Unjoin

The second important library function is *unjoin* for describing how to update a pair of tables using its join view with three strategies. It is implementation by sequential composition of four bidirectional transformation:

```
unjoin :: DeleteFlag -> (s1 -> a) -> (s2 -> a) -> (v -> a) -> Brul ([s1],[s2]) [v]
unjoin flag fs1 fs2 fv = productUnGroup fs1 fs2 ;
                         unZip fs1 fs2 ;
                         alignWithsubUnjoin flag fs1 fv;
                         group fv
```

where `f ; g` denotes composition of two BiGUL transformations `f` and `g` (i.e., `(Compose g f)`).

The `unjoin` is mainly composed by four bidirectional transformations: `group fv`, `alignWithsubUnjoin flag`, `unZip fs1 fs2` and `productUnGroup fs1 fs2`. As demonstrated in Figure 6, the basic idea to update the two sources (`source1` and `source2`) using the view (`view`), is to use grouped view (`viewg`) to update the zipped table (`ziptab`) of two grouped sources (`tabg1` and `tabg2`).

Rather than giving formal definitions for the above four bidirectional transformations in BiGUL, we explain their intuitive putback behavior through a concrete example. Recall the `uss1` defined with `unjoin` in Section 3.2, where we have given the updated results for the sources and the view in Figure 4. Figure 6 shows how the update on the view is propagated to the two sources.

- As the first step from the view to the source, `group fv` updates the grouped view (`viewg`) with the updated view (`view`), resulting in deleting (a3,b3,c3) from `viewg`.
- In the second step, `alignWithsubJoin flag fs1 fv` aligns `ziptab` with `viewg`. ([(a1, b1)], [(b1, c1)]) will be updated with [(a1, b1, c1)]; ([(a5, b5)], [(b5, c5)]) will be updated with [(a5, b5, c5)]; (a3,b3) will be deleted from ([(a3, b3)], [(b3, c3)]); ([(a2, b2)], [ ]) and ([ ], [(b4, c4)]) will remain unchanged.
- In the third step, for each pair of `ziptab`, `unZip fs1 fs2` putbacks its left/right part list (non-empty) to `tabg1`/`tabg2`, so `tabg1` and `tabg2` will be [[(a1, b1)], [(a2, b2)], [(a5, b5)]] and [[(b1, c1)], [(b3, c3)], [(b4, c4)], [(b5, c5)]].
- In the last step, `productUnGroup fs1 fs2` ungroups `tabg1` and `tabg2` to get the new sources [(a1, b1), (a2, b2), (a5, b5)] and [(b1, c1), (b3, c3), (b4, c4), (b5, c5)].

To put it a bit more concretely, the important bidirectional transformations are implemented as follows. The `group` bidirectional transformation

```
group :: (s -> a) -> BiGUL [[s]] [s]
```

is bijective and can be easily implemented in BiGUL. Its dual `ungroup` can be defined using `group`. With `ungroup`, we can define `productUnGroup`, a product of two `ungroup` BiGUL programs that apply to different functions.

```
productUnGroup :: (s1 -> a) -> (s2 -> a) -> BiGUL ([s1],[s2]) ([[s1]],[[s2]])
productUnGroup f1 f2 = Prod (ungroup f1) (ungroup f2)
```

We omit the implementation of `unZip`. Bidirectional transformation `alignWithsubUnjoin flag fs1 fv` is implemented using `align` and another new bidirectional transformation `subUnjoin`. The filter condition for `align` is: `not (null (fst s)) && not (null (snd s))`, which means both parts of the source pair are non-empty. The match condition is : `fs1 (head (fst s)) == fv (head v)`. Since both source and view elements are not null, the `head` function will always succeed. As we mentioned before, the results of alignment are three cases. For matched source and view elements, we use `subUnjoin` to synchronize them. For the unmatched view element, we create a proper source element and use `subUnjoin` to synchronize them. The most interesting one is the unmatched source case where the source pair will be updated by deleting either the first one, the second one or both according to the given flag. The `subUnjoin` is a kind of restricted `unjoin`. It requires that all shared field values of two source lists must be the same and both source lists must be non-empty. Since `subUnjoin` is only used in the matched and unmatched view cases of `alignWithsubUnjoin`, the requirement is satisfied automatically.

## 5 Example

In this section, we use one typical example proposed in the relational lenses paper [3] to illustrate how to use our putback-based bidirectional transformation library Brul to write more expressive and powerful programs.

Suppose the source table shown in Figure 7 stores five music track records, and each record contains its Track name, release Date, Rating, Album, and the Quantity of this Album. There are also functional dependencies from Track to Date (denoted as $Track \rightarrow Date$), Track to Rating (denoted as $Track \rightarrow Rating$), and Album to Quantity (denoted as $Album \rightarrow Quantity$). For example, the first two records have the same Date and Rating value since they have the same Track name, and the first and the third record have the same Quantity value since they have the same Album name even their Track names are different.

| Track | Date | Rating | Album | Quantity |
|---|---|---|---|---|
| Lullaby | 1989 | 3 | Galore | 2 |
| Lullaby | 1989 | 3 | Show | 3 |
| Lovesong | 1989 | 5 | Galore | 2 |
| Lovesong | 1989 | 5 | Paris | 4 |
| Trust | 1992 | 4 | Wish | 5 |

Figure 7: Source table

| Track | Rating | Album | Quantity |
|---|---|---|---|
| Lullaby | 3 | Show | 3 |
| Lovesong | 5 | Paris | 4 |
| Trust | 4 | Wish | 5 |

(a) View table

| Track | Rating | Album | Quantity |
|---|---|---|---|
| Lullaby | 4 | Show | 3 |
| Lovesong | 5 | Disintegration | 7 |

(b) Updated view table

Figure 8: View tables

The view table in Figure 8a consists of three records whose Quantity value is greater than 2 and whose Data is dropped. Suppose the view table is updated as shown in Figure 8b: the Rating of Track named Lullaby is raised to 4, the Quantity of Paris is updated to 7, and the record with Track name Trust is deleted.

In the following, we will show how to implement a putback-based bidirectional transformation program to synchronize the source table with the updated view table using our library functions based on BiGUL. Here, we use lists to simulate sets and represent tables and records for simplicity.

The function `u1` shown in Figure 9a accepts an argument of type `RType` which is used as a default value to fill in the Date attribute when there is a new record inserted into the view and no records in the source have the same Track value. The program is implemented using the library function `align`, which aligns the part of the source list satisfying certain filter condition (line 3) with the view list by a matching condition (line 4). The filter function determines whether the Quantity is greater than 2. The Haskell expression `r!!4` retrieves the fifth value of a list. The matching condition states that a source record and a view record are matched if they share the same Track name and Album name.

When aligning, there are three cases:

- A source record is matched with a view record: we first use a rearrangement function (line 5-9) to rearrange the view from a four-element list (`[t,r,a,q]`) to a five-element list (`[t,_,r,a,q]`) with the second element marked as underscore. This rearrangement function reshapes the view to match the shape of the source. Then, the element in the source is `Replace`d by the corresponding element in the view by an update (`[d|t = Replace|]`) which means the $t$ in the source record will be replaced by the $t$ in the view record.
- A view record that has no matching source record: a new source record is created (line 10) with a default value $d$ filled into the Date.
- A source record that has no matching view record: we simply delete this record by return `Nothing`.

The program can be executed bidirectionally either as a *put* function to update source by view, or as a *get* function that extracts a view from the source. Figure 10 shows the updated source table after executing the *put* direction of the program. The most interesting case is that even though the first record of the source table does not appear in the view, its Rating is updated from 3 to 4 according to the functional dependency since it has the same Track name as the second record.

The BRUL program written in Figure 9a only describes the *put* behavior, that is, how to align source and view records and update the source records, but in fact the *get* direction of this program implicitly also behaves as a SQL query:

```
select Track, Rating, Album, Quantity as v
from s where Quantity > 2
```

Running *get* of the program will get the view shown in Figure 8b. This BRUL program expresses the same behavior as the example shown in the relational lenses paper for both *get* and *put* functions.

```
u1 :: RType -> Brul [Record] [Record]
u1 d = align
  (\r -> (r!!4) > RInt 2)
  (\s v -> (s!!0 == v!!0)&&(s!!3 == v!!2))
  (RearrV
     [p|\(t:r:a:q:[]) -> (t:_:r:a:q:[])|]
     [d|t = Replace; r = Replace;
        a = Replace; q = Replace|])
  (\(t:r:a:q:[]) -> (t:d:r:a:q:[]))
  (\rs -> Nothing)
```

(a) Update program I

```
u2 :: RType -> Brul [Record] [Record]
u2 d = align
  (\r -> (r!!4) > RInt 2)
  (\s v -> (s!!0 == v!!0)&&(s!!3 == v!!2))
  (RearrV
     [p|\(t:r:a:q:[]) -> (t:_:r:a:q:[])|]
     [d|t = Replace; r = Replace;
        a = Replace; q = Replace|])
  (\(t:r:a:q:[]) -> (t:d:r:a:q:[]))
  (\(t:d:r:a:q:[]) -> (t:d:r:a:RInt 2:[]))
```

(b) Update program II

Figure 9: Two program in Brul

| Track | Date | Rating | Album | Quantity |
|---|---|---|---|---|
| Lullaby | 1989 | 4 | Galore | 2 |
| Lullaby | 1989 | 4 | Show | 3 |
| Lovesong | 1989 | 5 | Galore | 2 |
| Lovesong | 1989 | 5 | Disintegration | 7 |

Figure 10: Updated Source I

While the *put* semantics of the selection combinator in relational lenses for unmatched source records is fixed to deletion that can be described using $u1$, we carefully design the interface of `align` that allows users to specify the update policy. Figure 9b gives another update program `u2`. Instead of deleting the unmatched source record (Trust, 1992, 4, Wish, 5), we update it by changing the Quantity to 2. This is also a valid update: (Trust, 1992, 4, Wish, 2) will not appear in the view when performing *get*, since it will fail the filter condition.

Moreover, we can use an environment to update the source as shown in Figure 11. The environment stores a mapping from Album name to Quantity of this Album. The function `uSWithEnv` tries to find the Quantity of the specific Album in the environment. If it succeeds, we update the Quantity of the source record by the value retrieved from the environment otherwise by 0. For example, if we update the source with an `env` that stores a mapping from "Wish" to 1, the Quantity of unmatched source record "Trust" will be updated to 1. Last but not least, `align` checks that those unmatched source records will not satisfy the filter function after updating to guarantee that those records will not appear in the view.

## 6 Related Work

In relational databases, much research has been devoted to correct translation of updates on the view to updates on the source (the *view-update* problem) [1, 4]. To deal with different update strategies, Keller [11] classified different update translation algorithms for different update operations (i.e. replacement, insertion, and deletion) with the view that is computed using either selection, projection, join or a combination of these three operations, and suggested using a dialog to ask the user to choose a proper update translation algorithm where there exists more than one update translation algorithm for a given view and update operations on the view. In this paper, we show that Brul provides a concise way to specify and choose update translations.

Bidirectional transformations [6] originate from the *view-update* problem in relational databases, and many bidirectional languages [6, 2, 8, 9, 15, 16, 10, 7] have been proposed. Among them are a series of bidirectional languages called lenses; they are basically get-based in the sense that they provide a set of combinators that are intended to describe the *get* direction of bidirectional transformations. One problem with this get-based approach is that there often exist many *put*s for a given *get* and there is no way to choose a suitable *put* just by writing a good *get* and the get-based approach only provides one possible *put* as the default that may not satisfy the user's need.

The *putback-based* approach [5] opens a new research direction to resolve this problem for bidirectional transformations, and several *putback-based* bidirectional programming languages have been proposed to handle different kinds of data: putlenses [17] defines a set of bidirectional combinators that support *putback* style bidirectional programming. BiFluX [18] is a bidirectional functional update language for XML data; with BiFluX, the

```
u3 :: Env -> RType -> Brul [Record] [Record]
u3 env d = align
  (\r -> (r!!4) > RInt 2)
  (\s v -> (s!!0 == v!!0)&&(s!!3 == v!!2))
  (Rearrv
     [p|\(t:r:a:q:[]) -> (t:_:r:a:q:[])|]
     [d|t = Replace; r = Replace;
        a = Replace; q = Replace|])
  (\(t:r:a:q:[]) -> (t:d:r:a:q:[]))
  (\rs -> uSWithEnv rs env)
```

```
Type Env = Map RType RType

uSWithEnv :: Record -> Env -> Maybe Record
uSWithEnv r env =
  case Map.lookup (r!!3) env of
    Just q -> Just $ uRecord 4 q r
    Nothing -> Just $ uRecord 4 (RInt 0) r
uRecord :: Int -> RType -> Record -> Record
uRecord 0 v (x:xs) = v:xs
uRecord i v (x:xs) = x : uRecord (i-1) v xs
```

Figure 11: Update program III

| Track | Date | Rating | Album | Quantity |
|---|---|---|---|---|
| Lullaby | 1989 | 4 | Galore | 2 |
| Lullaby | 1989 | 4 | Show | 3 |
| Lovesong | 1989 | 5 | Galore | 2 |
| Lovesong | 1989 | 5 | Disintegration | 7 |
| Trust | 1992 | 4 | Wish | 1 |

Figure 12: Updated Source II

programmer can describe the *putback* function explicitly and the system derives a unique *get* function for free. BiYacc [20] lets the programmer write simple production-like rules which in fact describe how to update the concrete syntax tree using the abstract syntax tree as a "reflective" printer. Finally, BiGUL [12] was designed to serve as the foundation for higher-level putback-based bidirectional programming languages; BiGUL is designed to be concise yet powerful, and its well-behavedness has been fully verified in the dependently typed programming language Agda [14], guaranteeing that any program written in BiGUL is well-behaved.

Brul follows the *putback-based* approach, and it removes the ambiguity of *put* function by providing operations that let the user write update translations explicitly. Our careful design of the Brul library not only offers flexibility, but also guarantees that any *put* program written in Brul has a unique forward *get* semantics that forms a well-behaved bidirectional transformation with this *put*. We also extend *put* with parameters to allow the user to update the source by an environment.

## 7 Conclusion

We implemented a library Brul for writing bidirectional programs on relational databases, offering programmers the ability to specify flexible update policies. The programmers write *put* programs that describe how to use a view table to update a source table; corresponding *get* programs — queries that extract data from a source table to construct a view table — are then automatically derived. Brul is implemented on top of the putback-based bidirectional programming language BiGUL which is formalized in Agda, and hence all programs written with Brul are guaranteed to be well-behaved. We also explore the expressiveness of putback-based bidirectional programming by adding parameters to the *put* function to write more interesting examples, i.e. using a third environment when updating the source table. For future work, we plan to investigate how to write the putback behavior for aggregate functions (average, maximum, etc.), which are also frequently used in relational databases.

## Acknowledgments

We thank Jeremy Gibbons for carefully reading through our draft and giving plenty of suggestions, Zirun Zhu for helping with the intensive revision before the submission deadline, and all the reviewers for giving useful comments. This work was partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (A) No. 25240009.

## 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, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *International Conference on Functional Programming*, ICFP '10, pages 193–204. ACM, 2010.

[3] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: A language for updatable views. In *Principles of Database Systems*, PODS '06, pages 338–347. ACM, 2006.

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

[5] S. Fischer, Z. Hu, and H. Pacheco. "Putback" is the essence of bidirectional programming. Technical Report GRACE-TR 2012-08, GRACE Center, National Institute of Informatics, 2012.

[6] 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*, 29(3):17, 2007.

[7] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *International Conference on Functional Programming*, ICFP '08, pages 383–396. ACM, 2008.

[8] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Principles of Programming Languages*, POPL '11, pages 371–384. ACM, 2011.

[9] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *Principles of Programming Languages*, POPL '12, pages 495–508. ACM, 2012.

[10] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2):89–118, 2008.

[11] A. Keller. Choosing a view update translator by dialog at view definition time. In *International Conference on Very Large Data Bases*, VLDB '86, pages 467–474. Morgan Kaufmann Publishers, 1986.

[12] H.-S. Ko, T. Zan, and Z. Hu. BiGUL: A formally verified core language for putback-based bidirectional programming. In *Partial Evaluation and Program Manipulation*, PEPM'16. ACM, 2016.

[13] U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[14] U. Norell. Dependently Typed Programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009.

[15] H. Pacheco and A. Cunha. Generic point-free lenses. In *Mathematics of Program Construction*, MPC '10, pages 331–352. Springer, 2010.

[16] H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. In *International Workshop on Bidirectional Transformations*, volume 49 of *Electronic Communications of the EASST*, 2012.

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

[18] H. Pacheco, T. Zan, and Z. Hu. BiFluX: A bidirectional functional update language for XML. In *Principles and Practice of Declarative Programming*, PPDP '14, 2014.

[19] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, Haskell '02, pages 1–16. ACM, 2002.

[20] Z. Zhu, H.-S. Ko, P. Martins, J. Saraiva, and Z. Hu. BiYacc: Roll your parser and reflective printer into one. In *International Workshop on Bidirectional Transformations*, pages 43–50. CEUR-WS, 2015.