# A Putback-Based Approach to Bidirectional Programming

Tao Zan

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences
SOKENDAI (The Graduate University for Advanced Studies)

# A Putback-Based Approach to Bidirectional Programming

by

**Tao Zan**

**Dissertation**

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

***Doctor of Philosophy***

SOKENDAI (The Graduate University for Advanced Studies)

July 2016

# Abstract

In Kevin Kelly's new book *The Inevitable*, he said that the most important thing you need to focus is change. Even every change is small and unintentionally to be omitted, but it happens anytime and anywhere. If you do not keep your software up to date, the inconsistency will finally leads to a huge gap which may cause a big disaster. How to correctly propagate the changes and keep them consistency is a critical issue. Bidirectional transformation is a promising approach for maintaining consistency between two related information (we call them source and view), and many bidirectional programming languages are designed that let programmer merely write one program which denotes either a forward transformation (*get*) from source to view or a backward transformation (*put*back) that updates the source with the view.

Since usually source and view are not in a one-to-one correspondence, the backward transformation (change propagation from view back to source) is inherently ambiguous. Nevertheless, existing bidirectional transformation languages focus mainly on enforcing consistency and provide developers only limited control over the backward transformation, solving the latent ambiguity via default strategies whose behavior is unclear to developers. We propose a new *programming by update* paradigm in which developers write putback programs merely as updates that succinctly describe how a view can be used to update a source, such that the bidirectional behavior is fully determined. We have implemented BiFluX, which is a bidirectional update language for XML structured data that supports programmer in specifying flexible update strategies. Based on the core language of BiFluX, we built a clean core language BiGUL for putback-based bidirectional programming and also implemented a library Brul for relational database. Our putback-based languages have been used in refactoring, parsing and printing, and self-adaptive systems.

# Acknowledgements

When I was a high school student, my dream was to get in one of the top universities in China and find a decent job. After entering the University of Science and Technology of China, my alumni who are top researchers around the world gave me a huge impression about their pure passion of pursing the truth and discovering unknown knowledge. I want to be one of them, then I went to Tokyo where I stayed fives years for pursing my own truth.

I am especially grateful to Professor Zhenjiang Hu for his delicate supervision of my research. Before I joined the Programming Research Lab, I have no experience of doing research. Professor Zhenjiang Hu patiently guided me to survey the related works and finally find the key research problem. He always gives me wise suggestions and strong support, he taught me how to write a good research paper hand by hand, he cared about me even more than myself, he is a role model for me not only in research but also how to behave.

I must thank Professor Hiroyuki Kato and Professor Soichiro Hidaka for being my co-supervisors. Professor Hiroyuki Kato gives me a lot of useful advices and always asks important questions during my presentation to ensure I am really clear about the technical detail. Professor Soichiro Hidaka carefully introduces the GRoundTram system from which I started my research life. He always check my slides seriously to make sure I do not have any misunderstanding and gives me a lot of useful suggestions.

I must also thank our Postdoc researcher Hsiang-Shang Ko. He is really an extremely excellent researcher. Even he is not one of my supervisors, but he really did a lot of work in helping me sort out my research, giving advices, and guiding me.

I would like to thank the members of my dissertation committee, Professor Shin Nakajima, Professor Nobukazu Yoshioka, Professor Keisuke Nakano, and Professor Kanae Tsushima for giving insight comments.

I would like to thank all other members in our lab, Yu Liu, Chong Li, Lionel Montrieux, Atsushi Koike, Le Duc Tung, Zirun Zhu, and Yongzhe Zhang for having a great research life together. And to all the intern students that helped me a lot through the journey.

Not less important, I would like to thank my parents, my wife for supporting me all the time, without your support I could not go so far.

Tao Zan National Institute of Informatics & Tokyo, 2016

# Contents

# Chapter 1

# Introduction

## 1.1 Research Objective

The rapid progress of technology lets us surf the Internet anywhere with any devices conveniently, and nowadays people usually possess more than one smart device such as a laptop, a tablet, and even several mobile phones. Normally we prefer each device to be used in a particular situation that fit our needs. For example, we may lookup research papers through Google Scholar Search at the lab using desktop computer instead of using mobile since the screen size of desktop is big enough and desktop operation systems are more convenient for other research activities (e.g., implement algorithms, write papers, etc.); we may read Blog posts through Safari using iPhone during the transportation other than using a laptop since mobile devices are small and lightweight; we may watch YouTube movies at home through tablet or iPad since it provides touch facilities and is easy to be brought everywhere (e.g., during eating breakfast, before sleeping in the bed, etc.) and also the screen size is big enough compared with mobile devices.

As Kevin Kelly says in his new book *The Inevitable* [45] that the new technology greatly facilitates our daily lives but also introduces new problems. Typically, what we discuss here is the synchronization of information between different devices. Intuitively, the first solution comes to our mind would be

cloud synchronization services such as Dropbox [2], Google Drive [3], and Microsoft OneDrive [6]. I agree that they work well for synchronizing documents and pictures at the file level, but what I intend to say is more fine-grained synchronization services that work with structured data. For example, synchronization of bookmarks between different browsers. Since different web browsers have different bookmark formats, it is impossible to directly copy from one to another.

Concretely, the following is a demo bookmark file that used in Netscape [71] which is an HTML file with the bookmarks stored inside the <dl> tag. A bookmark is represented inside a <dt> tag including an url and the title. A folder that stores a set of bookmarks is wrapped by a <dd> tag and the name of this folder is wrapped by a <h3> tag.

```
<html>
<head>My Bookmarks</head>
<body>
    <h1>my bookmarks</h1>
    <dl>
        <dt><a href="foo.com">Foo's</a></dt>
        <dd>
            <h3>my folder</h3>
            <dl>
                <dt><a href="stefanzan.com">stefanzan</a></dt>
            </dl>
        </dd>
        <dt><a href="bar.edu">Bar's</a></dt>
    </dl>
</body>
</html>
```

While the following is another demo bookmark file that in the XBEL [7] format, which directly uses the <folder> tag for bookmark folders and <bookmark> tag for bookmarks.

```
<xbel>
```

```
    <title>NII bookmarks</title>
    <folder>
        <title>National Institute of Informatics</title>
        <bookmark href="http://www.nii.ac.jp/en">
            <title>English</title>
        </bookmark>
        <bookmark href="http://www.nii.ac.jp">
            <title>Japanese</title>
        </bookmark>
    </folder>
    <folder>
        <title>my folder</title>
        <bookmark href="stefanzan.com">
            <title>stefanzan</title>
        </bookmark>
    </folder>
    <bookmark href="bar.edu">
        <title>Bars</title>
    </bookmark>
</xbel>
```

In order to synchronize between these two bookmark files, a naive solution would be manually implementing two unidirectional transformations that one from Netscape to Xbel and another one from Xbel to Netscape, and also proving they are work correct together. This ad hoc solution is time inefficient and error-prone, since we have to write two transformations while these two are somehow related and any changes of the format requires the refactoring of both transformations carefully in order not to break the consistency.

Lenses [28] originated from the view updating problem in the database research area elegantly solves this problem. A lens consists of a pair of two transformations: a forward transformation *get* that extracts information from source to construct a view and a backward transformation *put* that propagates the updates on the view back to the source, which can be used to ensure the consistency between two related information (a source and a view). Each lens

has to satisfy the following two properties:

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

to guarantee the correctness which is called well-behavedness. Lenses are also compositional which is convenient for composing small lenses together to achieve large bidirectional programs. Programmers use lenses to simply write program like writing a unidirectional forward transformation, while it can be interpreted as a backward transformation as well. Since each lens is well-behaved, the program written use lenses are correct by construction.

Even the lenses framework provides a new direction to do synchronization, but this bidirectional programming style is based on an impractical assumption [26]:

> For a forward transformation *get*, it is sufficient to derive a suitable *put* that can be combined to form a well-behaved bidirectional transformation.

This is because in general the forward transformation *get* is not injective, and thus there could be many possible backward transformation *put*s together with this *get* to form well-behaved lenses. The *put* function can not be uniquely determined by the *get*, we call it the *nondeterminism* of *put*. Lenses solve the ambiguity by providing one default *put* semantics, but this default one may not be acceptable in practice.

For example, suppose we have a source `s` which is basically a list of integers: `[1, 2, 3, 4, 5]`, the *get* function we define (in Haskell [52]) is to extract all the odd numbers.

```
get s = filter isOdd s
```

`filter` is the Haskell built-in function, and `isOdd` is a function that checks a given number is odd or not. The result of `get s` is `[1, 3, 5]`. If we change the view to `[1,3]`, what the source would be ?

In fact, there are more than one *put*s for this *get* function, we can choose either to delete the number 5 in the source or change from 5 to arbitrary even

number. Using the traditional lenses, it is hard to encode arbitrary correct update strategies into the *put* function as it only provides a default one.

Consider another example, source is still a list of integers: `[1, 2, 3, 4, 5]`, and the get function is the summation of the source list.

```
get s = sum s
```

`sum` is a built-in function in Haskell that computes the summation of a list. The result of `get s` is `15`. If we update the view to `20`, there are many possible ways to modify the source list. For example, we can insert a new element `5` before the first or after the last of the list, update the first or the last element in the list by increasing `5`, increase each element in the source by `1`, and so on.

## 1.2 Contributions

In this dissertation, our contributions can be summarized as follows: 1. We proposed a new *programming by update* paradigm that let programmers write bidirectional transformations as simple as merely describing flexible updates on the source. Since the update program is in fact the put direction of a bidirectional transformation, generally we call it putback-based approach. Due to the powerfulness of put, the corresponding *get* will be uniquely determined. 2. We designed and implemented a typical bidirectional programming language BiFluX following this paradigm for XML structured data, and we lifted the core XML update language of BiFluX to a generic update language that is served as the general core for our putback-based bidirectional programming. 3. We implemented a bidirectional update library for relational database to solve the view updating problem.

We claim that to deal with the *nondeterminism* of *put*, bidirectional programming languages should be designed in a better way. We managed to achieve that by providing a *programming by update* paradigm to bidirectional programming that lets programmers write bidirectional programs as updates. Our language design is based on the theory (2.4.1) which states that a well-behaved *put* can uniquely determine the *get* function, in sharp contrast with the truth

that *get* cannot always uniquely determine the *put*. Since a well-behaved *put* can uniquely determine the *get*, if we carefully design the bidirectional programming language from the perspective of *put*, we will not suffer from choosing a proper *get* for a given *put* anymore since it is unique.

We have designed and implemented BiFluX [73, 74, 65, 75] which is short for *Bidirectional Functional lightweight update language for XML*. BiFluX borrows the language syntax from FluX [15] which is a functional update language for XML, and several new syntaxes are added such as an important `UPDATE FOR VIEW` syntax that means updating a source XML with a view XML, and pattern matching for decomposing source/view into small components. BiFluX allows programmers to write bidirectional programs by only concentrating on describing how to decompose source/view and specify the update policies he/she wants to update the specific part of the source using the corresponding view component, and a unique forward transformation from source to view can be derived free from the written update program. We provide programmers with full control over the puts which is the key contribution of our putback-based design. This is a big step of moving forward for bidirectional programming in order for it to be useful in real word applications, as the behavior of a bidirectional program can be fully controlled by programmer.

Based on the core language of BiFluX, we implemented a core language BiGUL [46] for generic data. The goal of designing BiGUL is to build a clean and solid foundation for higher-level *putback*-based languages. BiGUL is concise that only consists of a set of essential and orthogonal bidirectional statements, such as source/view rearrangement, case analyses, production for handling larger source and view, composition for creating complex programs. BiGUL is completely formally verified in the dependently typed programming language AGDA to guarantee that any program written in BiGUL is well-behaved.

Brul [72] is short for *Bidirectional relational update library*. View updating has been intensively studied in relational database community, Keller [43, 44] systematically proposed a series of update translation algorithms for different query (selection, projection, and join) and different update operation on the view (replacement, deletion, and insertion). He proposed a dialog based approach that lets user to choose the update translation policy at the view definition time

which is kind of combination of the *get* and *put* function, but still lacks of a good language to support that. We designed the library BRUL which provides basic *put* combinators that can 1) let programmers directly describe the *put* behavior, and 2) give programmers full control of the bidirectional behavior of their programs, since the *put* behavior uniquely determines the *get* behavior. BRUL is implemented on top of the generic bidirectional update language BIGUL. The well-behavedness of BRUL can thus be easily proved. BRUL is more powerful than the relational lenses that all the operations of relational lenses can be encoded in BRUL .

Our putback-based languages are used in many applications such as parser and pretty-printer [77], refactoring [16], attribute-based access-control [55], and self-adaptive systems [20, 76].

## 1.3 Overview of the Thesis

Chapter 2 gives an introduction of bidirectional transformations, shows the limitations of the current approaches, gives the theory based of our putback-based approach, and finally gives a survey of related work. Chapter 3 first explains the relation between put and update, then introduces our core XML update language and the generic bidirectional core update language. Chapter 4 introduces the bidirectional functional update language BIFLUX which is built for XML data. Chapter 5 introduces BRUL which is a *putback*-based library for relational database. Chapter 6 concludes the dissertation.

# Chapter 2

# Bidirectional Transformation

Bidirectional transformations (BX) can be used to keep different information consistent in a concise way: when either one is changed, a predefined bidirectional transformation program can propagate the changes from one to another to make them synchronized.

In this chapter, we will first explain the birth of bidirectional transformation and three basic approaches for synchronizing data, i.e. manually write two functions, bidirectionalization of an existing unidirectional language, and design domain specific bidirectional programming languages such as lenses typically. Since our work is based on the lenses solution, we introduce the lenses in detail, and show the limitations of current approaches. Finally, we give a survey of works on bidirectional transformations.

## 2.1 The Birth of Bidirectional Transformation

Bidirectional transformations are a mechanism for maintaining consistency between two or more related informations. The idea of bidirectional transformation comes from the view updating problem [8] in relational database. In relational database, a source table can be really huge that contains millions to even billions of records, so it is nearly impossible to directly manipulate data on this huge table. A proper solution is to create a smaller view from this big

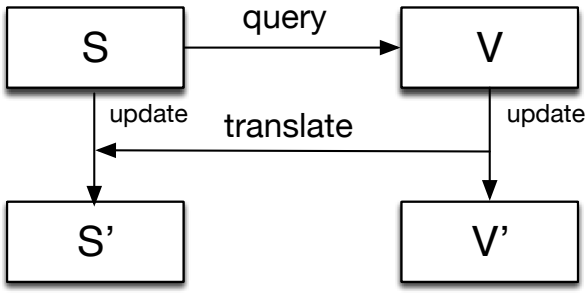


**Figure 2.1** View Updating

source, then it can be manipulated by database programmers easily. When the view is updated, the changes on the view need to be reflected back to the original source table as shown in Figure 2.1 since the view and the original source table become inconsistent. Researchers in relational database area proposed many approaches [8, 23, 31, 43, 47] to correctly propagate the changes on the modified view back to the original source.

In recent years, there is an emerging interest in bidirectional transformations from different research areas with different approaches inspired by different problems, but all of them have a common core idea that is to maintain the consistency between different information, which inherently related to the bidirectionality. Czarnecki and Hu et al. [22, 40] give a wide-ranging survey of bidirectional transformations from a variety of disciplines in computer science including programming languages, databases, and model-driven software engineering.

In model-driven software engineering, every entity in the world can be described using a model. For the same entity, there can be many different models described from different aspects. Models can be composed, some models are sub-models of other big models. These models are related and share some information in common. On the other hand, in order to easily create new models, model transformation languages are designed to describe

how to generate a new model from an existing model (source), e.g. ATL [41], and QVT-R [59]. Suppose there are modifications on the source model, we can run the model transformation program to generate a new model to keep the information consistent with the source model; but if the modifications are happened on the target model, we need a way to reflect the updates back to the source model. In model-driven software engineering, bidirectional model transformation are utilized to handle this issue. Stevens [69] gives a detailed survey of bidirectional model transformation and its challenges.

As the technologies evolves rapidly, smart devices become more and more powerful and convenient. People normally possess more than one device, and information are scattered among multiple devices, typically for example bookmarks among different browsers among different devices. How to synchronize information between different devices is a critical issue. For example, Suppose I bookmarked a useful website in my desktop computer using chrome and want to look into the details later on the mobile using safari during the transportation to go back home, I have to send an email to myself with the link information and check it out on the mobile email application which is really inconvenient. We need a way to synchronize the bookmarks among different browsers in different devices, and lenses [28] have been proposed that contains a set of useful combinators to write bidirectional programs to synchronize the data. Note that this is different from the file synchronization services like Dropbox [2] which only handle data at the file level, what we want to process is deeper into the data. For example, you cannot use Dropbox to synchronize bookmarks between safari and chrome. Synchronization is also needed in many other different areas such as database, model-driven software development. Each of them uses different formalization of bidirectional transformations to synchronize between their specific formats.

Due to different problems in different research areas, several different approaches are proposed to this problem. In summary, there are three approaches: *1*) Manually write two functions to synchronize between two information; *2*) Bidirectionlization of an existing unidirectional programming language by giving the language a specific update semantics; *3*) Design domain specific languages that each constructor has both a forward semantics that is used to

construct a view and a backward semantics that describes how to update the source.

The most straight-forward approach is to manually write two functions: one describes the transformation from source to view and the other one vice versa. While there are some problems for this approach: these two functions need to be carefully designed to guarantee they work well together. when the program is small, it is easy to check the behavior of these two programs is correct, but when they become complex and large, it is hard to guarantee those two functions together are will not break the consistency relation; what is more, when the specification changes, these two programs both are need to be revised to keep them up-to-date.

Another approach is bidirectionalization of an existing programming language which requires carefully giving a proper backward semantics for all the language constructs. If the target language is complex (for example, supports regular expressions, and graph data structures), it will be very hard to give a correct backward semantics. Liu et.al. [48] gives a backward interpretation of XQuery [30], and Hidaka et.al. [33] bidirectionalize UnQL [14] (a graph query language) to handle bidirectional transformations on graph structured data.

The third approach is to design domain specific bidirectional programming languages. When handling domain specific data, usually it is better to have a specialized language that provides more suitable syntax for programming, and the clean syntax also makes the semantics easy to define and prove. For synchronization between source and view data, lenses [28] as a bidirectional transformation framework that formalizes the bidirectional transformation as a lens that can be interpreted bidirectionally, and the bidirectional semantics are well-designed to satisfy certain properties called *well-behavedness* (Theory 2.2.1). Since our approach is kind of refined lenses, we will explain lenses in detail in the next section.

## 2.2 Lenses

Lenses are originated from the view updating problem in relational database with the purpose of synchronizing between multiple devices [5]. In this section, we will introduce the definition of lenses and the properties they need to be satisfied.

### 2.2.1 Definition

Before going to the definition, we first introduce a little about the convention. The function application is denoted by juxtaposition (i.e. the Haskell style). Normally, if a function $f$ that accepts an argument $a$, we will write $f(a)$, but it is written as

$$f\ a$$

in Haskell. If there are multiple arguments, normally we will consider them as the arguments of the function (e.g. $f(a, b)$). We will use the curried style instead, which means for multiple-argument functions, it will take one argument each time and the result will be another function which accepts the rest arguments. For example, the function $f(a, b)$ will be written as

$$f\ a\ b$$

, which means the function $f$ is applied to argument $a$, yield another function that accepts argument $b$, and finally computes the result.

A *lens* ($l$ for short) as shown in Figure 2.2 is a basic combinator that can be interpreted as a pair of functions: a *get* function that extracts part of information from the source to construct a view, and a *put* function that accepts an updated view and the original source, to produce an update source.

Formally, given a source type $S$ and a view type $V$, the type signature for *get* and *put* function can be written in Haskell as :

$$\begin{aligned} get &:: S \to V \\ put &:: S \to V \to S \end{aligned}$$

Note we will use the dot notation to identify the *get* and *put* function for a lens

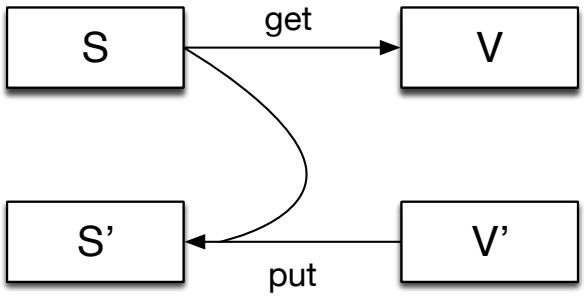


**Figure 2.2** Lenses framework

$l$, i.e. $l.get$ and $l.put$, and the type signature for a lens is $Len\ S\ V$ if the source type is $S$ and view type is $V$.

**Example 2.2.1.** We define a lens $l_1$ that contains a *get* function which extracts the second element from a source pair:

$$l_1.get\ \ (a, b) = b$$

, and a *put* function that updates the source by replacing the second element with the view:

$$l_1.put\ \ (a, b)\ \ b' = (a, b')$$

Suppose we have a source $s$ that is (1, "a"), then the get direction of the lens $l_1$ would extract the second element:

$$l_1.get\ \ s = \text{“}a''$$

If we update the view from "a" to "b", then the put direction of the lens $l_1$ will update the second element of the source:

$$l_1.put\ \ (1, \text{“}a'')\ \ \text{“}b'' = (1, \text{“}b'')$$

The advantage of lenses is that each lens is carefully designed to have both a straight-forward *get* semantics that behaves like a unidirectional program and one default backward *put* semantics that propagates the updates on the view back to the source. What is more, there is another combinator called *comp* that

can be used to compose small lenses into larger ones. By using composition, we can use lenses to write complex bx programs to solve large problems. The definition of *composition* is as follows:

$$comp :: Lens\ S\ V_1 \rightarrow Lens\ V_1\ V \rightarrow Lens\ S\ V$$

, which means if we have a lens $l_1$ can be used to synchronize between $S$ and $V_1$, and another lens $l_2$ that can be used to synchronize between $V_1$ and $V$, then we can create a new lens $comp\ l_1\ l_2$ that can be used to synchronize between $S$ and $V$.

### 2.2.2 Properties

Not arbitrary pair of *get* and *put* functions can form a correct pair as a lens to synchronize the source and the view, and thus certain properties need to be satisfied.

**Definition 2.2.1.** *(GetPut law) For a given source s, if a lens l satisfies the following property: l.put s (l.get s) = s, then the lens l is source preservable.*

If a lens $l$ is source preservable, then no changes on view will not impose any change on the source.

**Definition 2.2.2.** *(PutGet law) For a given source s and an updated view v, if a lens l satisfies the following property: l.get (l.put s v) = v, then the lens l is update preservable.*

If a lens $l$ is update preservable, then all the updates on the view will be reflected back to the source, and thus it can be reconstructed from the updated source exactly as the original view.

**Theorem 2.2.1.** *If a lens is both source preservable and update preservable, then it is well-behaved.*

A well-behaved lens guarantees the *put* will reflect no more than updates, and all the updates are reflected to the source.

The lens defined in example 2.2.1 is well-behaved, we can easily prove the two properties:

*Proof.*

$$
\begin{aligned}
& l_1.put\ \ (a,b)\ \ (l_1.get\ \ (a,b)) && (GetPut)\\
= \quad & \{- \textit{ Definition of } l_1.get\ -\}\\
& l_1.put\ \ (a,b)\ \ b\\
= \quad & \{- \textit{ Definition of } l_1.put\ -\}\\
& (a,b)
\end{aligned}
$$

$$
\begin{aligned}
& l_1.get\ \ (l_1.put\ \ (a,b)\ \ b') && (PutGet)\\
= \quad & \{- \textit{ Definition of } l_1.put\ -\}\\
& l_1.get\ \ (a,b')\\
= \quad & \{- \textit{ Definition of } l_1.get\ -\}\\
& b'
\end{aligned}
$$

□

We can easily define another lens that it is not well-behaved. The following is another example.

**Example 2.2.2.** Given a source which is a pair of integers, a view which is the second element of the pair, and we define the lens $l_2$ as follows:

$$
\begin{aligned}
& l_2.get\ \ (a,\ b) = b\\
& l_2.put\ \ (a,\ b)\ \ b' = (a,\ b'+1)
\end{aligned}
$$

The *put* function will update the second element of the source by increasing the view value by 1.

The lens $l_2$ does not satisfy the source preservation property, since even no updates on the view, the second element of the source pair will always be increased by 1.

*Proof.*

$$
\begin{aligned}
& l_2.get\ \ (l_2.put\ \ (a,b)\ \ b')\\
= \quad & \{- \textit{ Definition of } l_2.put\ -\}\\
& l_2.get\ \ (a,b'+1)\\
= \quad & \{- \textit{ Definition of } l_2.get\ -\}\\
& b'+1
\end{aligned}
$$

Since the result is $b' + 1$ which is different from the original view $b'$, so it does not satisfy the *PutGet* property and $l_2$ is not a well-behaved lens. □

As shown in Example 2.2.2, not arbitrary *put* function couped with the *get* function is well-behaved, the lens semantics need to be carefully designed (especially for the *put* function) to satisfy the well-behavedness properties.

Sometimes there may exist more than one possible *put* function together with a *get* function to be well-behaved, in this situation, lenses normally should be designed with one default *put* behavior in the underline semantics.

**Example 2.2.3.** Suppose the source $s$ is a list of pairs [(1, "Tokyo"), (2, "Kyoto"), (3, "Tokyo")], and view is constructed by extracting the first element of the pair whose second element is Tokyo. Let us call the lens $l_3$, the *get* function of the lens $l_3$ is defined as:

$$l_3.get \ = map\ fst \ \cdot \ filter\ (s \rightarrow snd\ s == "Tokyo")$$

The *get* function first uses a *filter* to get those pairs that the second element is Tokyo, then calls a *map* to extract the first element of each pair. The result of $l_3.get\ s$ is [1, 3]. Suppose we delete the first element in the view, resulting in [3]. How would the source be updated ? One default and most commonly chosen implementation of *put* is to delete the first element in the source, i.e. (1, "Tokyo"). While in fact, there are many possible *put* functions that can be defined. For example, instead of deleting this source pair, we can update (1, "Tokyo") to (1, "Kyoto"). Since the second element of the pair is changed to "Kyoto", it will not appear in the view since it not satisfy the filter condition.

The nondeterminism of the *put* also reveals a design restriction of the current lenses, that user cannot flexibility choose or specify the *put* behavior she/he wants, and this will be intensively studied by our research. Since the lenses are designed to let programmer think and write in the direction of *get*, we call it *get*-based lenses.

## 2.3 Limitations of Get-Based Lenses

The Lenses framework relieves the burden of writing bidirectional transformations by providing a set of bidirectional combinators that each one can be interpreted as either a *get* function that extracts a view from a source, or a *put* that propagates the updates on the view back to the original source. Programmers only need to write the program in a unidirectional way (*get*) plus having the pre-designed update semantics in mind (*put*). These bidirectional combinators can be composed together through *composition* (;), which is also a bidirectional combinator to create large bidirectional programs.

Even through the design of Lenses has these advantages, there is an impractical assumption: for a *get* function, it is sufficient to derive a *suitable put* function that these two functions forms a lens and it is well-behaved. Normally a *get* function may not be injective, and thus there may exist more than one possible *put* functions that can be combined with the *get* function to form a well-behaved lens. It is impractical to fix a "suitable" *put* function at the design time for a lens combinator, since the requirements change overtime. The "suitable" choice currently may not be the suitable one the next time. So it is hard to justify what "suitable" is in general.

**Example 2.3.1.** We define a *sum* lens $l_4$ which contains a *get* function that computes the summation of the pair:

$$l_4.get\ \ (a,\, b) = a + b$$

, and a *put* function that updates the second element of the source:

$$l_4.put\ \ (a,\, b)\ \ b' = (a,\, b' - a)$$

Suppose we have a source $s$ that is (1, 2), then the get direction of the lens $l_4$ would be 3:

$$l_4.get\ \ s = 3$$

If we have an updated view 5, then the put direction of the lens $l_4$ will update the second element of the source:

$$l_4.put\ \ (1,\, 2)\ \ 5 = (1,\, 4)$$

It is straight-forward to prove that lens $l_4$ is well-behaved. The *put* function of lens $l_4$ will always update the second element of the pair, we can also define another *put* function that updates the first element of the pair:

$$l_4.put\ \ (a,\ b)\ \ b' = (b' - b,\ b)$$

So, the result of $l_4.put\ (1, 2)\ 5$ will be $(3, 2)$. Or, we can even write more involved *put* programs:

$$l_4.put\ (a,\ b)\ b' = if\ a + b == b'\ then\ (a,\ b)\ else\ (\lfloor b'/2 \rfloor,\ b' - \lfloor b'/2 \rfloor)$$

There could be infinite possibilities to give a definition of the *put* function for a given *get*. It is not practical to only encoding one "suitable" *put* function in the underline semantics for a lens. As Stevens [69] says:

> The developer needs full control of what the transformation does. [...] We claim that determinism is necessary in order to ensure, first, that developers will find tool behavior predictable, and second, that organizations will not be unacceptably "locked in" to the tool they first use.

We need to provide a way, an interface, or a well-designed language to give programmer the flexibility to define the *put* semantics. So we start our research on *putback*-based bidirectional transformation, and proposed *putback*-based bidirectional programming languages to give programmer the choice of defining the *put*.

## 2.4 Putback-Based Bidirectional Transformation

We now introduce mathematical concepts, and then the definition and properties of *putback*-based bidirectional transformation. This Section is written according to the technical report [26] with different notations.

### 2.4.1 Mathematical Propositions

In this subsection, we give a brief introduction of basic mathematical concepts which will be used later for characterizing properties of *put* function.

**Definition 2.4.1.** *(Injectivity). A function f :: S → V is injective if and only if there exists a function g :: V → S, so that for all s. g (f s) = s.*

Intuitively, for each value $s$ in $S$, there is a unique result value $f(s)$ in $V$.

**Definition 2.4.2.** *(Surjectivity). A function f :: S → V is surjective if and only if there exists a function g :: V → S, so that for all s. f (g v) = v.*

Intuitively, for all value $v$ in the result type $V$, there exists an argument value $s$ in the source $S$ that can compute to.

**Definition 2.4.3.** *(Idempotence). A function f :: S → S is idempotent if for all s, f (f s) = f s.*

**Proposition 2.4.1.** *(Injectivity of put). The PutGet law implies that "put s" is injective for all sources s.*

Since the PutGet law is in the form of:

$$get\ (put\ s\ v) = v$$

Let the $get :: S \to V$, and $put :: S \to V \to S$ be the two functions, $s$ is the source of type $S$, $v$ is the view of type $V$, then $put\ s$ is a high-order function of type $V \to S$. Let $put'$ be $put\ s$, we can rewrite the PutGet law as follows:

$$get\ (put'\ v) = v$$

So it is easy to find out that $put'$ i.e. $put\ s$ is injective for all source s.

Since we write functions in curried style, now we provide another function *uncurry* that converts it from a curried style to the normal style:

$$uncurry\ f\ (x, y) = f\ x\ y$$

So the uncurried function $uncurry\ f$ accepts a tuple of arguments, and thus the type signature of *uncurry put* is:

$$uncurry\ put :: (S, V) \to S$$

**Proposition 2.4.2.** *(Surjectivity of uncurry put). The GetPut law implies that uncurry put is surjective.*

*Proof.* let $f = \lambda s \rightarrow (s, get\, s)$.

$$\begin{aligned} & uncurry\ put\ (f\ s) \\ = & \quad \{\ Definition\ of\ f\} \\ & uncurry\ put\ (s, get\ s) \\ = & \quad \{\ Definition\ of\ uncurry\} \\ & put\ s\ (get\ s) \\ = & \quad \{\ GetPut\ law\} \\ & s \end{aligned}$$

□

**Proposition 2.4.3.** *(PutTwice). If a lens is well-behaved, which means it satisfies the GetPut and PutGet laws, then executing two puts with the same view v, the result must be the same as only once. [27]*

$$put\ (put\ s\ v)\ v = put\ s\ v$$

*Proof.*

$$\begin{aligned} & put\ (put\ s\ v)\ v \\ = & \quad \{\ PutGet\ law\} \\ & put\ (put\ s\ v)\ (get\ (put\ s\ v)) \\ = & \quad \{\ GetPut\ law\} \\ & put\ s\ v \end{aligned}$$

□

If we flip the order of the arguments of *put*, i.e. let $put' = flit\ put$, then the type signature of $put'$ is:

$$put' :: V \rightarrow S \rightarrow S$$

The PutTwice property can be written as:

$$put'\ v\ (put'\ v\ s) = put'\ v\ s$$

Then we can easily conclude that $put'\ v$ is idempotent for all $v$.

### 2.4.2 Well-behavedness from Putback

**Definition 2.4.4.** *(Well-behaved put). Assume a put function that satisfies the following propositions:*

1. *((flip put) v) is idempotent for all views v.*
2. *"put s" is injective for all sources s.*
3. *"uncurry put" is surjective on the source type.*

*The put is well-behaved.*

**Proposition 2.4.4.** *(Uniqueness of view). Given a well-behaved put function, there is exactly one view view v such that put s v = s for every source s.*

(a) existence of view v: Suppose for all source s, each source s', such that s = put s' v.

$$
\begin{aligned}
& (put\ s\ v) \\
= \quad & \{\ s\ =\ put\ s'\ v\} \\
& put\ (put\ s'\ v)\ v \\
= \quad & \{\ PutTwice, Proposition\ 2.4.3\} \\
& put\ s'\ v \\
= \quad & \{\ s\ =\ put\ s'\ v\} \\
& s
\end{aligned}
$$

(b) uniqueness of view v: since "put s" is injective for all source s.

**Theorem 2.4.1.** *(Uniqueness of get for well-behaved put). Given a well-behaved put function, there exists only one get function that the get and put functions form a well-behaved lens.*

*Proof.* (a) existence of get: according to Proposition 2.4.4, define get s = v, such

that s = put s v.

$$
\begin{aligned}
& put\ s\ (get\ s) \\
= \quad & \{\ definition\ of\ get\ \} \\
& put\ s\ v, such\ that\ s\ =\ put\ s\ v \\
= \quad & \{\ s\ =\ put\ s\ v\} \\
& s
\end{aligned}
$$

$$
\begin{aligned}
& get(put\ s\ v) \\
= \quad & \{\ linv\ is\ left\ inverse\ of\ (put\ (put\ s\ v)\ for\ any\ view\ \} \\
& linv\ (put\ (put\ s\ v)\ ((get(put\ s\ v)))) \\
= \quad & \{\ GetPut\ \} \\
& linv\ (put\ s\ v) \\
= \quad & \{\ PutTwice\ \} \\
& linv\ (put\ (put\ s\ v)\ v) \\
= \quad & \{\ linv\ is\ left\ inverse\ of\ (put\ (put\ s\ v)\ for\ any\ view\ \} \\
& v
\end{aligned}
$$

(b) uniqueness of get: suppose there exits another get' that with the put is well-behaved.

$$
\begin{aligned}
& get'\ s \\
= \quad & \{\ Proposition\ 2.4.4\ \} \\
& get'\ (put\ s\ v),\ such\ that\ s\ =\ put\ s\ v \\
= \quad & \{\ PutGet\ \} \\
& v,\ such\ that\ s\ =\ put\ s\ v \\
= \quad & \{\ definition\ of\ get\} \\
& get\ s
\end{aligned}
$$

□

## 2.5 Survey

This section gives a survey of varieties of works related to bidirectional transformations started with view-updating, then bidirectional programming languages for different domains.

### 2.5.1 View Updating

In relational databases, view is created for the purpose of manipulating large source database in a relative small scale instead of directly operating on the big source table which is usually impossible and dangerous. Though it has many advantages, how to correctly translate the updates on the view to the update operations on the source as shown in the Figure 2.1 systematically is a great issue. Many research has been intensively done [8, 23, 31, 43, 44, 47] to tackle this problem.

Keller [43] studied the translation of different update operations (deletion, insertion, and replacement) on the view to proper update operations on the source based on different query operations (selection, join, and projection). He proposed a series of algorithms to handle each case. For example, when the query is a selection and the update operation on the view is deletion, then there are two specific algorithms for this case that we can choose either to delete the corresponding record on the source ([43] Chapter 5.1.4 Algorithm S-D-1) or replace the value of the non-key attribute in the record which also mentioned in the selection condition with a value that does not satisfy the selection condition ([43] Chapter 5.1.4 Algorithm S-D-2). For queries that involve composition of several small queries, he also proposed algorithms to analyze the updates and update the source table. It comes out that for one query operation and one specific update operation, there could be many possible update policies and it is impossible for programmer to remember all the algorithms and decide which one to use, and thus he proposed to use a dialog [44] to interact with programmer to let him/her choose a proper view update policy at the view definition time.

### 2.5.2 Bidirectional Programming Languages

The lenses framework [28] as shown in Figure 2.2 means that each lens can be interpreted as a *get* function that extracts information from source to construct a view, or a *put* function that putback the updated view to the original source to get an updated source.

Foster et al. [28] are the first one to propose the linguistic approach to the view-update problem, he designed a bidirectional tree transformation language Focal, formalized the lenses framework, and introduced the well-behavedness properties that the lenses should satisfy. He provided many basic combinators that work on trees (such as *fork*, *split*, *map*, and recursion) and composition. Each combinator has both the forward *get* and backward *put* semantics, and proved to be well-behaved under the type systems. The bidirectional tree language has been used to develop a synchronization system [5] to synchronize data between multiple devices typically bookmarks.

Bohannon et al. [11] developed a new bidirectional transformation language called Boomerang [1] for string data, especially with order. The basic string lenses contains a set of operations based on regular transducers such as union, concatenation, and Kleene-star to handle with a type system based on regular expressions. There are also some restrictions such as concatenation: two lenses shall be unambiguously concatenable, in order to be able to construct a unique way of splitting the result string into two strings in the *put* function of concatenation. They enriched the basic string lenses with dictionary lenses that asks programmers to specify a key for each string chunk. They keys are associated with the corresponding chunks which will be used in the *put* direction for finding the correct chunks to restore the string data globally when the orders are changed on the view.

Relational lenses [12] has been proposed to handle the relational data by Bohannon et al.. They redefine the relational algebra primitives as lenses. The forward semantics of selection and join are the same as in the relational algebra. A drop lens that drops one column of the data which can be used to resemble the projection operation. The backward semantics of each lens are carefully designed in order to be well-behaved. For example, the forward of selection extract the records that satisfy the where condition like a filter, and the backward semantics will reflect deletion on the view back to the source as deletion in order to make the delete records will not appear in the view to satisfy the PutGet law. Several restrictions have been imposed to guarantee well-behavedness, such as they only handle functional dependencies that can be represented in a tree form, and the join key must be one the the table's key.

Many other lenses languages are developed from the same group such as quotient lenses [29], matching lenses [9], symmetric lenses [34], and delta lenses [35]. Quotient lenses refines the well-behavedness by generalizing the equivalence relation. Matching lenses extends the key-based matching in Boomerang to allow arbitrary alignment strategies. Hofmann et al. [34] proposed a symmetric formalization of lenses that generalized from traditional asymmetric lenses. Since the composition of lenses will have many intermediate results and duplicated computation, Hugo et.al. [60, 61] has done several works to optimize the calculation.

Matsuda et al. [53] proposed an approach for bidirectionalizing transformation based on automatic derivation of view complement functions, that automatically derives backward transformation under a constant complement.

Diskin et al. [24] discussed the limitations of state-based approaches and forms an abstract delta lenses framework by two phases: it firstly computes the delta between the updated view (source) and the original view (source), and then transforms the view (source) deltas into source (view) deltas.

Hu et al. [39] designed a programmable editor for developing structured XML documents based on bidirectional transformations, and the editor is built based on X that is a lens language with several combinators. Liu et.al. [49] bidirectionalize XQuery by resembling the XQuery core into a lens language. Nakano et al. [56] built Vu-X system that can be used to describe a bidirectional transformation between XML document and HTML web pages, which allows user to edit directly on the HTML source and reflects the modifications to the XML document. Hidaka et al. [33] works on the bidirectionalization of graph query language UnQL [14] by giving the backward semantics of the core language of UnQL named UnCAL. They developed an IDE called GRoundTram [4] for easily analyzing bidirectional graph transformations. Since the subgraphs may have orders, they have also proposed a new graph transformation language lambdaFG for ordered graphs [32] and try to give backward semantics for this ordered version. Sasano et al. [67] tried to bidirectionalize a subset of ATL [41] transformations by translating the ATL programs into the UnCAL programs and then executed based on GRoundTram [4].

Kawanaka et al. proposed biXid [42] which is a bidirectional transformation language for XML. A biXid program describes a consistency relation between two different XML documents, and one can be generated from another through this biXid program. Cunha et al. [21] use bidirectional transformations to maintain the consistency between spreadsheet models and their instances, and thus the evolution of either side can be propagated to the other to keep them consistent. Query/View/Transformation Relations (QVT-R) [59] is a standard language defined by Object Management Group (OMG) to specify bidirectional model transformations, which allows to define a consistency relation between two models. Macedo et al. [51] implemented a large subset of this specifications using Alloy, and the *put* semantics follows the principle of least changes that computes an updated source that at a minimal distance from the original source. Cicchetti et al. [17] proposed JTL, which is a declarative bidirectional model transformation language that let user write model transformation in a QVT-like syntax, and the program is translated into a logic predicates and then using logic solvers to find all possible models. The problem of this approach is that the number of result models can be huge while the difference between models are very small. It would be better to have more specific constraints to narrow the solution space.

# Chapter 3

# Core Language

In Chapter 2, we have introduced bidirectional transformations, the existing approaches for bidirectional programming and the basic theory for putback-based bidirectional programming. The basic theory for putback-based programming says that there exists a unique *get* for a well-behaved *put*, which means if we can define a well-behaved *put*, a unique *get* can be derived from this *put* for free. In contrast with the traditional lenses, the lens combinators are designed in the *get* direction (programs are written as *get*), and thus there may exist many possible *put*s. This opens a new direction for designing bidirectional programming languages.

Putback-based bidirectional programming is a promising approach for bidirectional transformations, but it still lacks of well-designed user-friendly bidirectional programming languages. In this Chapter, we will firstly show the relationship between a put function and an update, then introduce our putback-based core bidirectional programming language for XML that follows a paradigm of *programming put by update*. Base on this core language, we design and implement a bidirectional update language BiFluX (Chapter 4) for XML which has a nice surface language for developer to flexibly describe puts.

## 3.1 From Put to Update

The *get*-based bidirectional programming languages have the advantage that it is straightforward since it only needs programmer to write a function that extracts values from a source to construct a view as writing a normal program, while if we design languages that let developer to write the backward transformation *put*, we can sense that it is not so easy merely from the type signature. The type signature of the *put* function for a bidirectional transformation in Haskell is as follows:

$$put :: S \to V \to S$$

which accepts a source of type $S$ and a view of type $V$, and outputs an updated source of the same source type $S$.

The goal of designing a good putback-based bidirectional programming language not only needs to guarantee the well-behavedness, but also includes releasing the burden of developers to write the put function.

If we flip the definition of *put*:

$$put :: V \to S \to S$$

Informally, it can be read as the *put* function updates source using view (If we look at the function *put v* which is of type $S \to S$). This parameterization on views motivates a bidirectional update language (in contrast to bidirectional transformation languages) in which programmers write bidirectional updates that modify an original source to embed some view information. Typepreserving updates are simpler to write than typechanging transformations in that one only needs to specify how the view information changes a small part of the source, leaving the remaining data fixed.

We design putback-based bidirectional programming language from the perspective of update, that has user-friendly update syntaxes to let developer merely write how to use a view to update a source. Even though we only flipped the arguments of the put function, it brings us a totally new style of designing bidirectional programming languages. In the following, when we use the term *bidirectional update language*, which also means *putback-based* bidirectional language.

We propose a novel *bidirectional programming by update* paradigm, in which the programmer writes an update program that describes how to update a source to embed information from a view, and the system derives a query from source to view that expresses the consistency between both documents. Such a *bidirectional update* describes the relationship between source and view in a simple way —as in the relational paradigm— by saying *which* related source parts are to be updated, but combined with additional actions that supply the missing pieces to eliminate the ambiguity in *how* target modifications are reflected —as in the combinatorial paradigm. For a wide class of BXs usually known as *lenses* [28], which have a data flow from *source* to *view*, this paradigm opens a new axis in the BX design space that enjoys a unique trade-off between the declarative style of relational approaches and the stepwise style of combinatorial approaches.

## 3.2 Core XML Update Language

Nowadays, various XML formats are widely used for data exchange and processing. Since data evolves naturally over time and is often replicated among different applications, it becomes frequently necessary to mutually convert between such formats. However, traditional XML transformation languages, like the XSLT and XQuery standards of the World Wide Web Consortium (W3C), are unsatisfactory for this purpose as they require writing a separate transformation for each direction.

Bidirectional transformation (BX) languages [22] mean to cover this gap, by allowing users to write a single program that can be executed both forwards and backwards, so that consistency between two formats can be maintained for free. As most interesting examples of bidirectional transformations are not bijective, there may be multiple ways to synchronize two documents into a consistent state, introducing ambiguity. Despite this fact, bidirectional languages are typically designed to satisfy fundamental consistency principles, and support only a fixed set of synchronization strategies (out of a myriad possible) to translate a (non-deterministic) bidirectional specification —the syntactic description of

a bidirectional transformation— into an executable BX procedure. This latent ambiguity often leads to unpredictable behavior, as users have limited power to configure and understand what a BX does from its specification.

Since XML is widely used among real world applications for data exchange and there are limitations for the get-based bidirectional transformations, we designed a bidirectional XML update language BiFluX which will be introduced in next chapter. Since BiFluX is normalized into a core bidirectional XML update language we designed, we will introduce the core language firstly. The core XML update language contains several components: patterns (Section 3.2.1) that are used to do pattern matching on XML data, paths and expressions (Section 3.2.2) that are essential for traversing XML documents and creating XML-structured data, the most important one is the bidirectional updates (Section 3.2.3) that have a set of update operators of which each one has bidirectional semantics, and unidirectional updates (Section 3.2.4) which are used inside of some bidirectional update operations.

### 3.2.1 Patterns

Pattern matching is a very useful feature of XML transformation languages like XDuce [38] or CDuce [10], allowing matching tree patterns against the input data to transform it into an output of different shape. Typical XML update languages like XQuery! [30] or Flux [15] do not support pattern matching, since it is not essential and may be more difficult to optimize, and they use solely paths to navigate to the portions of the input documents that are to be updated in-place. On the other hand, pattern matching can be used to guide the update based on the structure of the data.

Our pattern language follows that of XDuce [38]:

$pat ::= x \texttt{ as } \tau \mid \tau \mid c \mid () \mid n[pat] \mid pat, pat'$

A pattern can be a variable pattern restricted by a regular expression type $\tau$, a type pattern $\tau$, a constant pattern $c$ representing constant values, an empty pattern $()$, an element pattern, or a sequence pattern. We require every variable to be annotated with a type; this simplifies our design, but will also increase the

number of (often unnecessary) annotations in our update programs. We see it as an orthogonal problem that can be mitigated using existing tree-based type inference algorithms [70]. To reduce complexity, we impose a simple but strong syntactic *linearity* restriction on patterns (no alternative choice, no Kleene star) to ensure that matching a value against a pattern binds each variable exactly once. (Note that the restriction is imposed on patterns rather than on types, so we can still annotate patterns with alternation and sequence types.) Less severe linearity restrictions are actually known [36], but these simple patterns suffice for our practical needs.

For example, the following pattern:

$$person\ [\$sname\ \texttt{as}\ s : name, \$semail\ \texttt{as}\ s : email, \$affiliation\ \texttt{as}\ s : location]$$

is used to decompose a source `person` element into three parts: `$sname`, `$semail` and `$affiliation`. The first element of person is a name and it is matched with the `$sname` in the pattern. `$semail` and `$affiliation` are similar.

### 3.2.2 Expressions and Paths

Updates instrumentally use XQuery [66] expressions, XPath [18] paths and XDuce [37] patterns to manipulate XML data. Different expressions are used for different purpose: general expressions are arbitrary which are evaluated into a value, view expressions are a subset of them that need to be invertible, and source paths are again a subset of general paths because they are used to narrow the focus.

#### General Expressions and Paths

We write expressions $e$ in a minimal XQuery-like language, which is a variant of the $\mu$XQ core language proposed in [19]:

$$\begin{aligned} e ::= {} & () \mid e, e' \mid n[e] \mid p \mid \texttt{let}\ pat = e\ \texttt{in}\ e' \\ \mid {} & e = e' \mid \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \\ \mid {} & \texttt{for}\ x\ \texttt{in}\ e\ \texttt{return}\ e' \\ \mid {} & \texttt{case}\ e\ \texttt{of}\ \overrightarrow{pat \rightarrow e'} \end{aligned}$$

An empty expression $()$ means nothing; sequence expression $e, e'$ contains two expressions which are separated by a comma; path expression $p$ will be introduced later; let-binding binds a pattern *pat* to an expression $e$ and the binded variables are used in the expression $e'$; boolean expression $e = e'$ checks whether two expressions are equal; conditional expression, for-loop, and case expression are also supported in the core language. Note that there are no case expressions in $\mu$XQ as they can be emulated by conditional expressions; we extend our expression language with case expressions to make the core language easier to be used.

We differentiate paths $p$ in a core path language that represents a minimal dialect of XPath [18]:

$$
\begin{aligned}
p &::= \texttt{self} \mid \texttt{child} \mid ::nt \mid \texttt{where}\ e \mid p\ /\ p' \\
&\quad \mid x \mid w \mid \texttt{true} \mid \texttt{false} \\
nt &::= n \mid \texttt{text}() \mid \texttt{node}()
\end{aligned}
$$

Path `self` points to the current XML element itself, and `child` points to children of the current XML document. *nt* is short for name test that tests whether the current XML document node satisfies the given name ($n$), is a text node (`text()`) or a normal node (`node()`). Where condition ( `where` $e$) is used to extracts those that satisfy the expression $e$. Complex path $p\ /\ p'$ traverses deeper into the XML document that it firstly evaluates the path $p$ to a sequence of values, and then evaluates the path $p'$ over the sequence. Simple path can be a variable $x$, a constant $w$, a boolean value `true` or `false`.

For example, the following is a case expression that checks whether the current XML document is a person and his/her affiliation is NII.

```
case self of
  person [$sname as s : name, $semail as s : email, $affiliation as s : location]
    → $affiliation / child / :: text() = "NII"
```

It uses `self` to locate on the current XML document, then does pattern matching on this document to extract the sub-element (e.g. $\$affiliation$), and finally uses a boolean expression to check whether the affiliation is NII. Since the affiliation is an element node:

```
<affiliation>NII</affiliation>
```

To extract text string, we need first to get the children of the element node which is a text node, and then extract the text from the text node by :: $text()$.

To simplify the formal treatment, we consider nodetests $::nt$ that apply to atomic values and where clauses `where` $e$ that filter values satisfying an expression $e$. As syntactic sugar, we write $p :: nt \triangleq p$ / $::nt$, $p[e] \triangleq p$ / `where` $e$, and $p$ / $n \triangleq p$ / `child` $:: n$.

### View Expressions and Paths

Restrictions have to be placed on expressions used in the statement $[b]e$, since such expressions should express invertible computations to allow the statement to be bidirectionalized. The allowed subset of the expressions and paths defined in Section 3.2.2 is as follows:

$e ::= () \mid e, e' \mid n[e] \mid p \mid w \mid$ `true` $\mid$ `false`
$p ::= x \mid$ `self` $\mid$ `child` $\mid ::nt \mid p$ / $p'$

### Source Paths

A source path $p$, as used in the statement $p[b]$, narrows the source focus to only part of the current source. Not all paths can be used to change the source focus: We do not allow constant string paths ($w$) and boolean paths (*true* and *false*), as they are meaningless for focus narrowing. Note that it is intrinsically different from the unidirectional update, as the source type does not change and the source data is not updated in-place, but modified to embed the view data. Also, only the `self` and `child` axes are supported; this ensures that only descendants of the source focus can be selected as the new source focus and that a selection contains no overlapping elements. To sum up, the valid source paths are as follows:

$p ::= x \mid$ `self` $\mid$ `child` $\mid :: nt \mid p$ / $p'$

### 3.2.3 Bidirectionalizable Updates

Unlike conventional XML update languages, our core update language mainly consists of bidirectionalizable updates. Their names suggest what their update semantics are, and they will be interpreted as bidirectional transformations that update a source document given a view document or query a source document to compute its view fragment. The grammar of core *bidirectionalizable updates* $b$ is as follows:

$$
\begin{aligned}
b ::= {} & \texttt{skip} \mid \texttt{fail} \mid \texttt{replace} \mid p[b] \mid [b]e_v \mid b; b' \\
\mid {} & \texttt{alignpos}\ e_f\ b\ c\ r \\
\mid {} & \texttt{alignkey}\ e_f\ e_{ms}\ e_{mv}\ b\ c\ r \\
\mid {} & \texttt{caseS}\ p\ \texttt{of}\ \overrightarrow{pat \to b \mid u} \\
\mid {} & \texttt{caseV}\ e_v\ \texttt{of}\ \overrightarrow{pat \to b} \\
\mid {} & \texttt{ifS}\ e\ \texttt{then}\ b\ \texttt{else}\ b' \\
\mid {} & \texttt{ifV}\ e\ \texttt{then}\ b\ \texttt{else}\ b' \\
\mid {} & \texttt{iter}\ b \mid \texttt{view}\ x := e_v\ \texttt{in}\ b \mid P(p_s, e_v)
\end{aligned}
$$

We will informally describe their update semantics below.

The operation `skip` keeps the source unchanged provided that the view is empty, the `fail` operation aborts an update, and the `replace` operation replaces the source with the view.

The operation $p[b]$ traverses the source along a source path $p$, and runs a further update $b$ on the sub-source. For example, if a source variable $\$source$ points to an XML document that consists of a list of people.

```
<person>
  <name>Adam Smith</name>
  <email>as@as.com</email>
</person>
<person>
  <name>James Bird</name>
  <email>jb@jb.com</email>
</person>
  <name>Michael Johnson</name>
```

```
  <email>mj@mj.com</email>
</person>
```

Then a source path $*source* / `child` / :: *person* extracts all the people under the $*source*. If we want to update all the people by replacing them with the view (suppose the view is a person), we can write a simple program $*source* / `child` / :: *person*[`replace`]. It will first evaluates the source path, and the source will be a list of people, and each person will be replaced by the given view.

Dually, the operation $[b]e_v$ changes the current view by evaluating an expression $e_v$ on it and uses the result as the new view for the update $b$. This is often used when the view contains some extra structural information and it can not be directly used to update the source. Then a view expression is used to extract the data to be used in the bidirectional update $b$. For example, we have a view that is an XML element with a tag named employee.

```
<employee>
  <person>
    <name>David Edison</name>
    <email>de@mj.com</email>
  </person>
</employee>
```

This view shall not be directly used to update the source person using the program $*source* / `child` / :: *person*[`replace`], since each source person shall be updated with a view that has the same shape. In this situation, we can use a view expression $*view* / `child` / :: *employee* to extract the person element under the employee, and the whole program is written as [$*source* / `child` / :: *person*[`replace`]]($*view* / `child` / :: *employee*).

Composition $b_1; b_2$ updates a part of the source with $b_1$, and another part of the source with $b_2$. For example, we have a composition statement as follows:

$*semail* [[`replace`] $*vemail*];
$*sname* [[`replace`] $*vname*]

Each one is in the form of $p[[b]e_v]$. The first one replaces source email ($*semail*) with view email ($*vemail*), and the second one replace source name ($*sname*)

with view name ($*vname*). To guarantee the PutGet property, the same part of the source cannot be updated twice with two different view variables. This is enforced by requiring that the two statements $b_1$ and $b_2$ update different source variables. It is possible that one view variable depends on another view variable, even points to the same value. We provide view dependency description syntax to express the dependency between view components which will be introduced later.

The two special alignment statements (`alignpos` and `alignkey`) update a source sequence using a view sequence. They receive a source filtering expression $e_f$ which evaluates to a boolean, and match source elements satisfying $e_f$ with view elements by position (`alignpos`) or by key (`alignkey`). In the latter case, keys are computed from the source and view respectively by evaluating two expressions $e_{ms}$ and $e_{mv}$. Expression $e_s$ and $e_v$ specify the matching condition between source and view element (i.e. the evaluated result of both are equal).

After aligning the source sequence with the view sequence, there are three cases to consider: For matched source–view element pairs, we use a bidirectionalizable statement $b$ to synchronize them; for an unmatched view, we use a *create statement* $c$ to create a suitable source to match with the view; for an unmatched source, we use a *recover statement* $r$ to either delete or transform it. Create statements $c$ are simply unidirectional updates which will be introduced in Section 3.2.4. Recover statements $r$ are enriched unidirectional updates of the form:

$$
\begin{aligned}
r ::= {} & \texttt{delete} \mid u \mid \texttt{if}\ e\ \texttt{then}\ r\ \texttt{else}\ r' \\
\mid {} & \texttt{case}\ e\ \texttt{of}\ \overrightarrow{pat \rightarrow r}
\end{aligned}
$$

We can use `delete` for deleting an unmatched source, or unidirectional updates to modify an unmatched source so that the $e_s$ filtering expression evaluates to false.

For example, the source is a list of people, and view is a list of employees. Suppose each employee is decomposed into name ($*vname*) and email ($*vemail*) through pattern matching. If we want to update the source people that work at NII with the corresponding view employees that have the same name, we can match the source people list with the view employee list by their name using

the alignment (`alignkey`) operation.

```
alignkey
  $person / affiliation / :: text() = "NII"
  $person / name / :: text()
  $vname / :: text()
  $person / email [replace [$vemail]];$person / name [[replace] $vname]
  insert person [name [""], email [""], affiliation ["NII"]]
  delete
```

The first argument is a boolean expression that is used to extract those at NII, the second and third arguments extract the name text from source person and view name respectively which will be used to match the source person and view employee. If a source person and view employee are matched, then the email and name of the source person will be replaced by the corresponding view email and name; if there is an employee that has no matching source person, a new person will be inserted into source with the affiliation marked as NII (the name and email will be updated with the employee); if there is a person that has no matching view employee, this person will be deleted from the source.

The core language also provides two kinds of conditionals (`ifS` and `ifV`) and case statements (`caseS` and `caseV`), whose expressions or paths are evaluated on the source and view respectively. The case statement on source also supports source adaptation which means if the source satisfies certain pattern *pat*, we can choose to update this source using a unidirectional update $u$ (which will be introduced later) instead of directly giving a bidirectional update $b$. This is really useful when the source is in a shape that cannot be directly updated by view, we first restructure the source through a unidirectional update, then the restructured source will again be matched with any of the case pattern and be updated.

The operation `iter` $b$ embeds the same view into each element of a source sequence through the bidirectional update $b$. A procedure call $P(p_s, e_v)$ updates the sub-source at the end of the source path $p_s$ using the result of evaluating the view expression $e_v$ as the view. Procedures may be recursive. The statement

`view` $x := e_v$ `in` $b$ states that the value for a view variable $x$ can be computed from the rest of the view using the view expression $e_v$, and then runs $b$ using the remaining view. This means that part of the view binded to the variable $x$ depends on the other parts of the view. Since the part binded to the variable $x$ can be computed by the rest parts, the variable $x$ will not be used in the bidirectional update $b$.

### 3.2.4 Unidirectional Updates

Unidirectional updates are used in the create statement of the alignment operations (`alignpos` and `alignkey`). Our core *unidirectional updates* are adapted from the core FLUX update language [15]:

$$\begin{aligned} u ::= & \texttt{skip} \mid u; u' \mid \texttt{insert}\ e \mid \texttt{delete} \\ & \mid \texttt{if}\ e\ \texttt{then}\ u\ \texttt{else}\ u' \mid \texttt{case}\ e\ \texttt{of}\ \overrightarrow{pat \to u} \\ & \mid p[u] \mid \texttt{left}[u] \mid \texttt{right}[u] \mid \texttt{children}[u] \end{aligned}$$

These include standard operations such as the no-op `skip`, sequential composition, conditionals, and case expressions. The basic operations are `insert` $e$, which inserts a value $e$ to the current location; and `delete`, which replaces any value with the empty sequence. We can also apply an update in a specific *direction* (that traverses down a path $p$, moves to the `left` or `right` of a value, or focuses on the `children` of a labeled node).

## 3.3 Generic Core Language

The core language presented in Section 3.2 still retains XML-specific details and freely uses pattern matching and variables to manage data-flow, making it difficult to directly give a formal bidirectional semantics to the language. To achieve bidirectionality more reliably, we designed a generic bidirectional update language BIGUL, which is an invariant of BIGUL [46]. The data representation is XML-free and the data-flow management is done in a point-free style. The advantage of using BIGUL is that it is completely formally verified in the dependently typed language AGDA [58] to guarantee that any program

$$
\begin{aligned}
\mathit{bigul} \;::=\; & \texttt{Replace} \mid \texttt{Fail} \mid \texttt{Skip} \mid \texttt{Update}\ \mathit{upat} \mid \texttt{Iter}\ \mathit{bigul} \\
\mid\; & \texttt{RearrS}\ \mathit{sRearr}\ \mathit{bigul} \\
\mid\; & \texttt{RearrV}\ \mathit{vRearr}\ \mathit{bigul} \\
\mid\; & \texttt{CaseS}\ \overrightarrow{\mathit{caseSBranch}} \\
\mid\; & \texttt{CaseV}\ \overrightarrow{\mathit{caseVBranch}} \\
\mid\; & \texttt{Align}\ \mathit{filter}\ \mathit{match}\ \mathit{bigul}\ \mathit{create}\ \mathit{conceal} \\
\mid\; & \mathit{Compose}\ \mathit{bigul}\ \mathit{bigul} \\
\mathit{upat} \;::=\; & \texttt{UVar}\ \mathit{bigul} \mid \texttt{UIn}\ \mathit{upat} \mid \texttt{UProd}\ \mathit{upat}\ \mathit{upat} \\
\mid\; & \texttt{UConst}\ a \mid \texttt{ULeft}\ \mathit{upat} \mid \texttt{URight}\ \mathit{upat} \\
\mathit{caseSBranch} \;::=\; & (\mathit{predicate}, \mathit{branch}) \\
\mathit{branch} \;::=\; & \texttt{Normal}\ \mathit{bigul} \\
\mid\; & \texttt{Adaptive}\ \mathit{adaptSource} \\
\mathit{caseVBranch} \;::=\; & (\mathit{predicate}, \mathit{bigul})
\end{aligned}
$$

**Figure 3.1** Syntax of BiGUL.

written in BiGUL satisfies the BX properties. The semantics of BiGUL is concretely defined as monadic programs, with all dynamic checks for guaranteeing well-behavedness explicitly appearing in the programs.

### 3.3.1 Syntax and Semantics

The syntax of BiGUL is shown in Figure 3.1, which originates from, and thus many operations —e.g., `Replace`, `Fail`, and `Skip`— resemble those presented in Section 3.2.3. 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 `()`.

```
bigul1 :: BiGUL Int ()
bigul1 = Skip
```

For the BiGUL program `bigul1`, given a source `1` and view `2` in the interpretation of *put*, it will fail since the view type must be `()`; if the view is `()`, then the result of program `put bigul1 1 ()` will be `1`; no matter what the source integer value it is, the get direction of `bigul1` will always be `()`.

`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.

```
bigul2 :: BiGUL String String
bigul2 = Replace
```

For example, suppose source `s` is `"Tokyo"` and view `v` is `"NewYork"`, the result of `put bigul2 s v` will be `"NewYork"` which means the source string is replaced by the view string. If we execute `get bigul2 "NewYork"`, the result will be `"NewYork"` as the view.

### Source and View Rearrangement

`RearrS` updates a source $s$ using view $v$ by first computing a (usually) smaller source $s'$ 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.

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 *bigul4* to update the source with the view:

```
bigul4 :: BiGUL (a,b) b
bigul4 = 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 also provide a more readable surface syntax (implemented with Template Haskell [68]):

```
bigul4' :: BiGUL (a,b) b
bigul4' = 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"`.

### Update

One important new operation is `Update`, which is used to decompose a source into parts by pattern matching, and then update each part by a separate BiGUL program. The patterns used are called update patterns (*upat*): `UVar` updates the current source with a *bigul* statement, `UIn` updates the children of the current source, `UProd` decomposes the source into a product of two parts and updates each part separately, `UConst` matches the current source with a given value, and `ULeft` and `URight` handles the situation in which source is a choice. The view for the `Update` operation should have the same structure as the update pattern; to rearrange the view into that structure, a new operation `RearrV` is introduced, which computes a new view by a simple invertible function (like what $[e]b$ does). Sometimes the source also needs to be rearranged for updating, so a dual operation `RearrS` is introduced.

Here is a small example about `Update` and `RearrV`. Suppose that the source is a piece of personal information which have name, email, and affiliation as its children, and a view that is a pair of name and email. If we want to update the source's name and email with the information from view, we can write a BiGUL program like this:

```
RearrV
  (λ(vname, vemail) → (vname, (vemail, ())))
  (Update (UIn (UProd (UVar Replace)
                      (UProd (UVar Replace)
                             (UVar Skip)))))
```

We first rearrange the view pair $(vname, vemail)$ into a triple, adding an empty view element at the end in order to match with the update pattern (also matching a triple), then update the source by using `UIn` in order to update its children, which is a product of elements, and finally decompose the product by two `UProd` patterns. After the source is decomposed into a triple, we use the updates `Replace`, `Replace`, and `Skip` associated with the `UVar` patterns to replace the name and email and leave the affiliation unchanged.

### Case Analysis

`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. Concretely, a Haskell function that returns a boolean value is used to check the given source and view satisfy the specific branch, and two different kinds of branches are provided: either a normal BiGUL operation followed by a boolean condition on source (The boolean condition is used to reducing the checking complexity in the interpretation of get) or an adaptive branch that changes the source to a new one (We will explain more about the adaptation).

Let us use an example to see how it works.

```
bigul5 :: BiGUL (Bool, Int) Bool
bigul5 = Case
  [(\(b, i) v -> b == v,
     Normal (RearrV [|\v -> (v, ())|]
                    (Prod Replace Skip)) (\_ -> True)),
   (\_ _ -> True,
     Normal (RearrV [|\v -> (v, 0)|]
                    (Prod Replace Replace))) (\_ -> True))
```

```
  ]
```

The program `bigul5` is implemented use the `Case` with two normal branches. Source is a pair of a boolean value and an integer, and view is a boolean value. The first branch checks whether the boolean value `b` is the same as the boolean value of view `v`, and if they are equal, then view is rearranged to a pair with the second one is `()`, and finally the first element of the source is replaced by the view `v` and the second element is skipped; the second branch handles the other cases, which updates the first element of the source pair with view and reset the value of the second element by 0.

Note that both branches have the "acceleration" conditions that return true which in fact does not accelerate the interpretation of get since it will always return true. This is because during the interpretation of get, it will skip this branch if the "acceleration" condition on source returns false. Otherwise, we need to first compute a view from the branch and then check the computed view together with the source matches the case condition which may require lots of recursive computation.

#### Source Adaptation

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 [Int] Int
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 `v` to the $i_{th}$ location in the source list, and `get embedAt i s` to retrieve the value from source at the $i_{th}$ location. 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.

### Source-view Alignment

The `Align` operation in BiGULx unifies the two core operations `alignpos` and `alignkey` into one, based on the observation that `alignpos` can be regarded as a special case of `alignkey` that uses position as the key. The boolean *filter* function corresponds to $e_f$, while the boolean *match* function specifies when a source and view element are matched. The remaining three arguments deal with the three cases arising from source–view alignment: the *bigul* program deals with matched pairs, the *create* function creates a new source from an unmatched view element, and the *conceal* function deletes or modifies an unmatched source element.

`CaseS` and `CaseV` are two kinds of case analysis on either source or view.

They differ from their counterparts in the core in two aspects: BiGULx's `CaseS` and `CaseV` always match the whole source or view with the branches, and the source or view is fed into a boolean function (*predicate*) to decide whether it matches a branch. BiGULx does not include conditionals like `ifS` and `ifV` in the core, since they are subsumed by `CaseS` and `CaseV`.

### Composition

Composition is an important combinator, which can be used to concatenate BiGUL programs sequentially.

For example, suppose we have a source list `s`:

```
[(1, "Tokyo"), (2, "Kyoto"), (3, "Osaka")]
```

We want to update the first element of the list from Tokyo to NewYork. Since we have already written two BiGUL programs: `bigul4` and `embedAt`, we can use `Compose` to compose these two programs together to achieve that.

Even though the `embedAt` program is specialized to integers, we only need to modify a little to satisfy our needs.

```
embedAt' :: Int -> BiGUL [(Int, String)] String
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 (\_ _ -> [(i+1, "Dft")])),
   (\_ _ -> i > 0,
     Normal (RearrS [|\(x:xs) -> (x, xs)|]
                    (Prod Skip (embedAt (i-1)))))
  ]
```

Compared with `embedAt`, the `embedAt'` only changes the type signature and the case branch when the index i is greater and equal to zero while the list is empty,

we adaptive the source to `[(i+1, "Dft")]`.

Finally, the following BiGUL program update the $i_{th}$ element of the source list with the view value.

```
bigul7 :: Int -> [(Int, String)] String
bigul7 i = Compose bigul4 (embedAt i)
```

The result of `put (bigul7 0) s "NewYork"` will be:

```
[(1, "NewYork"), (2, "Kyoto"), (3, "Osaka")]
```

If we update the $4_{th}$ element of the source using `put (bigul7 4) s "NewYork"`, the result will be:

```
[(1, "Tokyo"), (2, "Kyoto"), (3, "Osaka"), (4, "Dft"), (5, "NewYork")]
```

Since length of the the original source list is less than 4, it will add new default value to the end of the list and update the one with the correct index with the view.

### 3.3.2 Well-behavedness

The well-behavedness of BiGUL is fully verified in Agda [57]. For use in practical applications, BiGUL is ported to Haskell as an embedded domain-specific language (shown in Figure **??**). A BiGUL program with type `BiGUL s v` describes a bidirectional program between source of type `s` and view of type `v`. 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

$$\begin{aligned} \texttt{get b s} = \texttt{Right v} &\Rightarrow \texttt{put b s v} = \texttt{Right s} \quad (\mathit{GetPut}) \\ \texttt{put b s v} = \texttt{Right s}' &\Rightarrow \texttt{get b s}' = \texttt{Right v} \quad (\mathit{PutGet}) \end{aligned}$$

These are the well-behavedness properties verified in Agda.

# Chapter 4

# A Bidirectional Functional Update Language for XML

In Chapter 3, we designed a bidirectional core update language for XML-structured data that includes expressions, paths, bidirectional updates and unidirectional updates. Since this core retains many XML specific features like paths and expressions, the processing of XML details and bidirectional semantics are mixed together which makes it hard to guarantee the correctness of the bidirectional semantics. In order to target this problem, we lift the core to a generic one by removing the XML specific features that the bidirectional semantics can be defined and proved easily.

Even the core language supports XML specific operations such as paths, expressions, etc., but it is lower-level and inconvenient for programmer to use. In this chapter, we introduce the surface language BiFluX (BIdirectional FunctionaL Updates for XML) that has a user-friendly syntax for programmer to write bidirectional update programs. BiFluX is inspired by the Flux (a functional XML update language), which adopts a novel *bidirectional programming by update* paradigm, where a program succinctly and precisely describes *how* to update a source document with a target document in an intuitive way, such that there is a unique "inverse" source query for each update program. BiFluX extends Flux with bidirectional actions that describe the connection between source and target formats.

## 4.1 Syntax, Informal Semantics, and General Framework

This section explains the syntax and informal semantics of BiFluX with a typical example, and then shows the big picture of our general framework.

### 4.1.1 Our Running Example

Consider a typical address book whose format is represented by the DTD from Figure 4.1. An address book contains a list of people, each possessing a name, an email address, and the person's affiliation. Let us start with the following XML address book with three people:

```
<addrbook>
  <person>
    <name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email>
    <affiliation>NII</affiliation>
  </person>
  <person>
    <name>Josh Ko</name>
    <email>joshko@ox.ac.uk</email>
    <affiliation>Oxford</affiliation>
  </person>
  <person>
    <name>Zhenjiang Hu</name>
    <email>zh@nii.ac.jp</email>
    <affiliation>NII</affiliation>
  </person>
</addrbook>
```

On the other hand, the NII's administrative services may keep only a view with the name and email address of employees (people who are affiliated to NII), as shown in the DTD from Figure 4.2. We have a view that simply keeps

```
<!DOCTYPE addrbook [
<!ELEMENT addrbook(person*)>
<!ELEMENT person(name,email,affiliation)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT email(#PCDATA)>
<!ELEMENT affiliation(#PCDATA)>]>
```

**Figure 4.1** A simple address book DTD.

```
<!DOCTYPE niibook [
<!ELEMENT niibook (employee*)>
<!ELEMENT employee (name,email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>]>
```

**Figure 4.2** An NII address book DTD.

the email of each person working at NII:

```
<niibook>
  <employee>
    <name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email>
  </employee>
  <employee>
    <name>Zhenjiang Hu</name>
    <email>zh@nii.ac.jp</email>
  </employee>
</niibook>
```

We can perform update operations on this view XML. For instance, we can add Tao (in alphabetical order) as a new NII employee, fix Zhenjiang's email, and delete Hugo:

```
PROCEDURE niibook($source AS s:addrbook, $view AS v:niibook) =
UPDATE person[$sname AS s:name,
              $semail AS s:email,
              $affil AS s:affiliation] IN $source/person BY
{ MATCH -> REPLACE $semail WITH $vemail
| UNMATCHV -> CREATE VALUE
       <person>
         <name/>
         <email/>
         <affiliation>NII</affiliation>
       </person>
| UNMATCHS -> DELETE .
} FOR VIEW employee[$vname AS v:name,
                    $vemail AS v:email] IN $view/employee
MATCHING SOURCE BY $sname VIEW BY $vname
WHERE $affil/text() = "NII"
```

**Figure 4.3** BiFluX update for the institutional address book example.

```
<niibook>
  <employee>
    <name>Tao Zan</name>
    <email>zantao@nii.ac.jp</email>
  </employee>
  <employee>
    <name>Zhenjiang Hu</name>
    <email>zhenjhu@nii.ac.jp</email>
  </employee>
</niibook>
```

Now the updated view and source XMLs become inconsistent, and we need to update the source XML to restore consistency. We can write a simple program in BiFluX to describe how the update should proceed, which is shown in

Figure 4.3[1]. The program has a single procedure *niibook*, which consists of a single `UPDATE FOR VIEW` statement specifying how to update the source using the view. It focuses on a sequence of people in the source by traversing down the path $*source*/*person*, selecting only those that work at NII through the `WHERE` expression, and also focuses on a sequence of employees in the view by traversing down the path $*view*/*employee*. Elements in the two sequences are matched by their names, as specified by the `MATCHING` condition. Both the source and view elements are decomposed by pattern matching. A matching person-employee pair is processed according to the `MATCH` clause, updating the person's email with the employee's email. If an unmatched employee exists in the view, according to the `UNMATCHV` clause, a new person with NII as the affiliation is created in the source. We do not need to fill in the name with `$vname` and email with `$vemail`, as the underlying semantics will pass the newly created source person into the `MATCH` clause, and thus both the name and email will be updated with the corresponding view elements. (Although the `MATCH` clause does not include a statement `REPLACE $sname WITH $vname`, this statement in fact will be derived from the `MATCHING` condition and implicitly inserted.) If an unmatched person exists in the source, the person will be deleted, according to the `UNMATCHS` clause. This `UPDATE FOR VIEW` syntax is specifically designed for specifying flexible alignment strategies in update programs, and can be regarded as a novel feature of BiFluX.

### 4.1.2 Syntax and Informal Semantics of BiFluX

In this section we will explain the syntax and informal semantics of BiFluX in terms of the running example in Figure 4.3. The syntax for BiFluX's main constructs is defined in Figure 4.4 and Figure 4.5, which is based on Flux [15], a high-level, purely functional language for writing XML updates. Statements *Stmt* include updates, composition, conditionals, let-binding, case expressions, and procedure calls. Update statements *Upd* include insertion, deletion, replacement, update under a path (`UPDATE BY`), update of a source using a view

[1]The names `s:elem` and `v:elem` are BiFluX type variables that refer to the types of source and view `elements` declared in the respective DTDs.

```
Stmt  ::=  Upd [WHERE Conds] | Stmt ; Stmt
       |   { Stmt } | { }
       |   IF Expr THEN Stmt ELSE Stmt
       |   LET Pat = Expr IN Stmt
       |   CASE Expr OF { Cases }
       |   P(Path, Expr)
Upd    ::=  INSERT (BEFORE | AFTER) PatPath
              VALUE Expr
       |   INSERT AS (FIRST | LAST) INTO PatPath
              VALUE Expr
       |   DELETE [FROM] PatPath
       |   REPLACE [IN] PatPath WITH Expr
       |   UPDATE PatPath BY Stmt
       |   UPDATE PatPath BY VStmt
              FOR VIEW PatPath [Match]
       |   CREATE VALUE Expr
```

**Figure 4.4** Concrete syntax of BiFluX updates (Part I).

(`UPDATE FOR VIEW`), and source creation. They may be guarded by a `WHERE` clause that defines a set of conditions constraining when the updates are executed.

In general, a BiFluX update is executed for a particular source and view as follows: by evaluating a source path or performing pattern matching on the current source, we obtain a *source focus selection*, which is recursively updated using a *view focus selection* computed by evaluating a view path or performing pattern matching on the current view, until all the view information is embedded into the source. View and source focus selections denote the parts of the source and view that can be respectively updated and used by the update.

Below we will go through each of the constructs.

$$
\begin{array}{lcl}
\mathit{Conds} & ::= & \mathit{Expr}\ [;\ \mathit{Conds}] \\
& | & \mathit{Var} := \mathit{Expr}\ [;\ \mathit{Conds}] \\
\mathit{Cases} & ::= & \mathit{Pat} \rightarrow \mathit{Stmt} \\
& | & \mathit{Pat} \rightarrow \texttt{ADAPT SOURCE BY}\ \mathit{Stmt} \\
& | & \mathit{Cases}\ '|'\ \mathit{Cases} \\
\mathit{VStmt} & ::= & \{\ \mathit{VStmt}\ \} \mid \mathit{VUpd} \\
& | & \mathit{VUpd}\ '|'\ \mathit{VUpd} \\
\mathit{VUpd} & ::= & \texttt{MATCH} \rightarrow \mathit{Stmt} \\
& | & \texttt{UNMATCHS} \rightarrow \mathit{Stmt} \\
& | & \texttt{UNMATCHV} \rightarrow \mathit{Stmt} \\
\mathit{Match} & ::= & \texttt{MATCHING BY}\ \mathit{Path} \\
& | & \texttt{MATCHING SOURCE BY}\ \mathit{Path} \\
& & \qquad \texttt{VIEW BY}\ \mathit{Path} \\
\mathit{PatPath} & ::= & [\mathit{Pat}\ \texttt{IN}]\ \mathit{Path}
\end{array}
$$

**Figure 4.5** Concrete syntax of BiFluX updates (Part II).

### Procedure

In BiFluX, large bidirectional update programs are constructed by using a list of small procedures. A procedure is defined in the following syntax:

$$\texttt{PROCEDURE}\ P(\mathit{Var}\ \texttt{AS}\ \tau,\ \mathit{Var}\ \texttt{AS}\ \tau) = \mathit{Stmt}$$

The first argument is the source and the second one is the view. $\tau$ is a regular expression type (whose details will be presented in Section 4.3.2).

In the running example, we declare a procedure named `niibook` with source argument `$source` and view argument `$view`, whose types are `s:addrbook` and `v:niibook` respectively. XML element names appearing in types (in this case `addrbook` and `niibook`) are prefixed with either `s:` or `v:` to specify that they are from the source or view DTD, since there may be elements with the same name but different definitions in the source and view DTDs.

### Path

In BiFluX, we use a subset of XPaths to traverse XML data. We omit the detailed definition of XPaths in the paper for brevity, only giving explanations in terms of a couple of paths used in the running example instead. For one, the path `$source/person` extracts all the three people under `$source`, which points to an `addrbook`, and produces:

```
<person>
  <name>Hugo Pacheco</name>
  <email>hpacheco@nii.ac.jp</email>
  <affiliation>NII</affiliation>
</person>
<person>
  <name>Josh Ko</name>
  <email>joshko@ox.ac.uk</email>
  <affiliation>Oxford</affiliation>
</person>
<person>
  <name>Zhenjiang Hu</name>
  <email>zh@nii.ac.jp</email>
  <affiliation>NII</affiliation>
</person>
```

Function `text()` extracts text information. For example, `$affil/text()` gets the text in the `affiliation` element pointed to by `$affil`: If `$affil` points to `<affiliation>NII</affiliation>`, then the result will be "NII".

### Source and view matching

The main difference between FluX and BiFluX is that updates on sources can use view information. Such an update is performed by a new `UPDATE FOR VIEW` operation, which synchronizes elements in the results of evaluating a source path and a view path. In the running example, we update a list of people denoted by `$source/person` with a list of employees denoted by `$view/employee`.

The *Pat* `IN` *Path* notation is used to decompose the elements in the two lists, so that we can specify further updates on the sub-elements.

One of the key design issues for the bidirectional-programming-by-update paradigm is to invent a nice syntax for describing flexible *alignment* strategies, and our solution is the `UPDATE FOR VIEW` operation. The operation comes along with a matching condition that means the synchronization can be configured by the programmer via the matching condition that aligns source and view elements, and a triple of matching/unmatching clauses (*VUpd*) that describe the actions for individual source-view elements. When two source and view elements `MATCH`, a bidirectional statement is executed to update the source using the view; during compilation, a `REPLACE` statement derived from the `MATCHING` condition is implicitly inserted into the `MATCH` clause to guarantee that the source and the view still match after the update. An unmatched view element (`UNMATCHV`) creates a temporary element in the source according to a unidirectional `CREATE` statement, and the temporary source element will be updated using the view element via the `MATCH` clause. An unmatched source element (`UNMATCHS`) is `DELETE`d by default, but we may keep it by providing a unidirectional statement describing how to invalidate the given `WHERE SOURCE` selection criteria. The rule is that all BiFluX statements are bidirectional, except inside `UNMATCHS` or `UNMATCHV` clauses.

Let us use the running example illustrated in Figure 4.3 for explanation. It matches a list of source person elements that satisfies the where condition (`WHERE $affil/text() = "NII"`) with a list of view employees by source person's name ($*sname*) and view person's name ($*vname*). For the matched source person and view employee, update its email by the view employee's email; for the view employee that there is no corresponding matching source element person, create a new source person with default affiliation set to NII; for the source person that there is no corresponding matching view element employee, delete this person.

### Flux Operations

Some update operations are inherited from Flux, which can be *singular*, to update single trees, or *plural*, to update the children of the selected tree. Singular replacements (`REPLACE WITH`) replace each node selected by a path, while plural replacements (`REPLACE IN`) replace their content. Singular insertions (`INSERT BEFORE/AFTER`) insert a value before or after each node selected by a path, while plural insertions (`INSERT AS FIRST/LAST INTO`) insert a value at the first or last position of the child-list of each selected node. Singular deletions (`DELETE`) delete each selected node, while plural deletions (`DELETE FROM`) delete their content.

Below we use several examples to show how these updates work. Suppose that we have an XML database of type `addrbook` pointed to by variable `$s`, which is initially an empty list. We insert a `person` into the database by using the following plural insertion:

```
INSERT AS FIRST INTO $s VALUE
  <person>
    <name>Tao Zan</name>
    <email>zantao@nii.ac.jp</email>
    <affiliation>NII</affiliation>
  <person>
```

(Since the database is empty, either `INSERT AS FIRST` or `INSERT AS LAST` works.) Now the database contains one entry for Tao. If we want to change Tao's affiliation, we can use the following update:

```
REPLACE $s/person/affiliation/text() WITH "Osaka"
```

This update can also be written using a *plural* replacement (`REPLACE IN`):

```
REPLACE IN $s/person/affiliation WITH "Osaka"
```

Finally, we can clear the database by `DELETE FROM $s` or even delete the database itself by `DELETE $s`.

### Source Adaptation

Different from the first version of BiFluX, case statements have been extended to include a source adaptation mechanism. With the first version of BiFluX, if a source is not compatible with the given view, we can only throw away the source and create a new one from the view. Sometimes, however, we do want some information in the old source to be preserved in the new one. The source adaptation mechanism is added for this purpose: when incompatibility arises, the old source can be transformed to a new one compatible with the view, while keeping part of the original source information.

To illustrate how source adaptation works, consider the following scenario: The source is a record about either a book or a magazine that contains the title, authors, price, and publication year, e.g.,

```
<book>
  <title>Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year><price>30.00</price>
</book>
```

and there is a variable `$s` pointing to the above record. The view can be either a book or a magazine with only the title and price, e.g.,

```
<magazine>
  <title>Everyday Italian</title>
  <price>15.00</price>
</magazine>
```

Suppose that we have decomposed the above view with the following pattern `magazine[$vtitle AS v:title, $vprice as v:price]`. The following BiFluX program with *source adaptation* can transform the source into a `magazine` and then update it with the view values:

```
CASE $s of
  magazine[$title AS s:title, s:author+,
           s:year, $price AS s:price]
```

```
  -> REPLACE $title WITH $vtitle;
     REPLACE $price WITH $vprice
book[s:title, $ars AS s:author+,
     $y AS s:year, s:price]
  -> ADAPT SOURCE BY CREATE VALUE
      <magazine><title/>{$ars}
        {$y}<price/></magazine>
```

Since the source is a book, it matches the second, adaptive branch and is transformed to a magazine with the author and year information preserved in the new source. After encountering an adaptive branch and executing the associated transformation, the case statement will be run again on the new source, which, in this case, is a magazine and matches the first normal branch. The replacement statements are then executed, producing the following updated source:

```
<magazine>
  <title>Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year><price>15.00</price>
</magazine>
```

The programmer should adapt the source with the intention of making it match with a normal branch — to avoid falling into adaptive branches repeatedly and resulting in non-termination, the underlying engine BiGUL will check that the adapted source matches a normal branch; subsequently, the adapted source will be updated by the bidirectional update statement in that branch and an updated source will be generated.

### 4.1.3 Bidirectional Execution

Although the emphasis is on writing updates, BiFluX programs have a bidirectional interpretation. They can be read as 1) an *update function* $U(s, v') = s'$ that updates a source $s$ into a new source $s'$ which contains a given view $v'$, or 2) a *query function* $Q(s) = v$ that computes a view $v$ from a given source $s$; these

```
PROCEDURE niibook($source AS s:addrbook, $view AS v:niibook) =
UPDATE person[$sname AS s:name,
              $semail AS s:email,
              $affil AS s:affiliation] IN $source/person BY
{ MATCH -> REPLACE $semail WITH $vemail
| UNMATCHV -> CREATE VALUE
       <person>
         <name/>
         <email/>
         <affiliation>NII</affiliation>
       </person>
| UNMATCHS -> REPLACE IN $affil WITH "NCI"
} FOR VIEW employee[$vname AS v:name,
                    $vemail AS v:email] IN $view/employee
MATCHING SOURCE BY $sname VIEW BY $vname
WHERE $affil/text() = "NII"
```

**Figure 4.6** Another update Strategy in BiFluX.

functions may be partial. For the running example in Figure 4.3, (assuming that people are uniquely identified by their names) the query function is semantically equivalent to the XQuery expression:

```
<niibook>
{
  for $person in $s/person
  where $person/affiliation/text() = "NII"
  return <employee>
           {$person/name}
           {$person/email}
         </employee>
}
</niibook>
```

For example, a typical use case is to run the BiFluX program as a query on the source in Section 4.1.1 and get the first view in that section, which is then modified to the second view. To produce a new, consistent source, the BiFluX program is run as an update on the original source and the modified view. In the new source, Josh is left unchanged, Tao is created with the default affiliation NII (as his name does not match any name in the original source), Zhenjiang's email is updated, and the Hugo is deleted:

```
<addrbook>
  <person>
    <name>Josh Ko</name>
    <email>joshko@ox.ac.uk</email>
    <affiliation>Oxford</affiliation>
  </person>
  <person>
    <name>Tao Zan</name>
    <email>zantao@nii.ac.jp</email>
    <affiliation>NII</affiliation>
 </person>
  <person>
    <name>Zhenjiang Hu</name>
    <email>zhenjhu@nii.ac.jp</email>
    <affiliation>NII</affiliation>
  </person>
</addrbook>
```

Note that it is possible to preserve information in the original source (in this case Josh's information) since in a update program we can choose to update only part of the source and keep everything else. This is not always supported by "bidirectional" XML transformation languages: biXid [42], for example, supports transformation of one XML format into the other and vice versa, creating a new XML document from scratch every time; consequently, when biXid converts a more informative format $F_1$ into a less informative format $F_2$, the information exclusive to $F_1$ will be lost and cannot be recovered when converting a $F_2$-formatted document back to $F_1$.

Our language is carefully designed to ensure that the inferred relationship between sources and views is deterministic, so that capturing it by a query function is appropriate. In other words, there exists a unique query function for each update program written in our language. Moreover, its bidirectional semantics satisfies two basic synchronization properties: that an update $U$ consistently embeds view information to the source:

$$U(s, v') = s' \;\Rightarrow\; Q(s') = v' \qquad \textsc{UpdateQuery}$$

and that it does not update already consistent sources:

$$Q(s) = v \;\Rightarrow\; U(s, v) = s \qquad \textsc{QueryUpdate}$$

These two properties are commonly known as the well-behavedness laws of lenses in the bidirectional programming community [22].

The UpdateQuery property indicates that view information must be *fully embedded* into the source and cannot be arbitrarily discarded. This calls for careful language design that helps the programmer to manage view information and check that the view is indeed fully embedded. In BiFluX, full embedding is checked during compilation to guarantee that the view can be reconstructed from the source. For example, if we write an empty statement ({}) in the `MATCH` clause of the running example instead of `REPLACE $semail WITH $vemail`, the program will fail to compile, as it will be discovered that the view variable `$vemail` is not used and hence not embedded into the source.

Sometimes a part of the view contains only redundant information in the sense that it can be computed from other parts, and hence does not need to be embedded. This situation can be explicitly described with a `WHERE` clause. For example, suppose that in the view we include for each name some extra indexing information that can be derived from the name, and this indexing information is not present in the source. At some point in the BiFluX program for synchronizing this kind of source and view, we might have two view variables `$vname` and `$index`, denoting a name and an associating indexing information. We can embed `$vname` into the name part of the source, but cannot do so for `$index`, since `$index` does not have a corresponding part in the source. In this case, we indirectly embed `$index` into the source by specifying the dependency between `$index` and `$vname` as follows: `WHERE $index := index[$vname]`. Af-

ter that, `$index` is considered embedded, and we only need to embed `$vname` into the source.

### 4.1.4 Other Update Strategies

In this section, we show that BiFLuX is flexible enough for describing other update strategies that may better reflect the user's intention.

For the running example we have explained, even though the BiFLuX program in Figure 4.3 gives a reasonable update strategy for many situations, this strategy is not the only one possible; for example, deleting a person from the view may actually mean that the person just moves to another institute instead of disappearing from the source database. We can easily describe this alternative update strategy by modifying the `UNMATCHS` case, as shown in Figure 4.6.

Running this second update moves people like Hugo to a new institute, in this case "NCI", producing the updated source:

```
<addrbook>
  <person>
    <name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email>
    <affiliation>NCI</affiliation>
  </person>
  <person>
    <name>Josh Ko</name>
    <email>joshko@ox.ac.uk</email>
    <affiliation>Oxford</affiliation>
  </person>
  <person>
    <name>Tao Zan</name>
    <email>zantao@nii.ac.jp</email>
    <affiliation>NII</affiliation>
 </person>
```

```
    <person>
      <name>Zhenjiang Hu</name>
      <email>zhenjhu@nii.ac.jp</email>
      <affiliation>NII</affiliation>
    </person>
  </addrbook>
```

For simplicity, we omit updating his email address accordingly in the BiFLuX program. In general, we can describe even more complicated strategies like conditionally delete or modify the person's information in the UNMATCHS clause.

This behavior cannot usually be described using the typical BX languages (e.g. lenses [28]), which are designed from the perspective of *get*, as they only provide one default update strategy for the *put* direction, usually reflecting deletion on the view to deletion on the source, and the user has no way of specifying a different update strategy for the *put* direction.

The main difference between BiFluX and Foster's lenses [28] is that the emphasis is now on writing a *put* transformation instead of a *get* transformation. This will allow a much more flexible and intuitive control over backward synchronization strategies, by making several put design choices explicit in the design of a bidirectional update.

### 4.1.5 General Framework

The general architecture of our bidirectional updating framework is illustrated in Figure 4.7. A BiFluX program is evaluated in two stages. First, it is statically compiled against a source and a view schema (represented as DTDs), producing a bidirectional executable. The generated executable can then be evaluated bidirectionally for particular XML documents conforming to the DTDs: in forward mode as a query $Q$, or in backward mode as an update $U$.

The compilation of BiFluX has two stages: The high-level BiFluX language is first normalized into a clean core language with syntax simplification, and then the core language is compiled into an XML-oblivious language BiGUL.

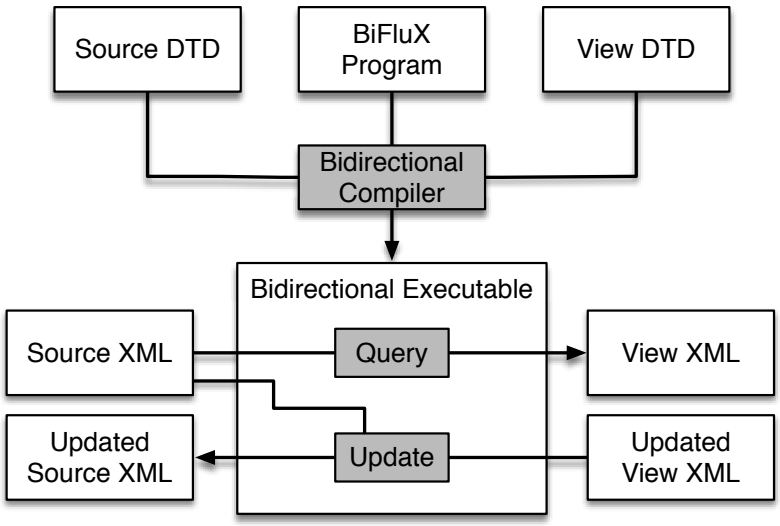


**Figure 4.7** Architecture of the BiFluX framework.

The second stage is the key part that includes handling XML values and regular expression types, checking necessary bidirectional transformation constraints and bidirectionalizing the core language by dealing with paths, composition etc. We will explain the core language, introduce BiGUL, and describe the compilation rules from core to BiGUL in the following sections.

## 4.2 BiFluX to Core Update Normalization

In this section, we formalize the translation from the high-level BiFluX language to the core language presented in Section 3.2, highlighting the significant gap between them. This process is usually referred to as *normalization* in languages like XQuery [30] and Flux [15]. We define two main normalization functions that interpret statements as bidirectional $[\![ - ]\!]^b_{Stmt}$ and unidirectional updates $[\![ - ]\!]^u_{Stmt}$.

```
$source / child / :: person
[ [ alignkey
  (case self of
    person [$sname as s : name, $semail as s : email, $affiliation as s : location]
      → $affiliation / child / :: text() = "NII")
  (case self of
    person [$sname as s : name, $semail as s : email, $affiliation as s : affiliation]
      → $sname)
  (case self of
    employee [$vname as v : name, $vemail as v : email] → $vname)
  (caseS self of
    person [$sname as s : name, $semail as s : email, $affiliation as s : affiliation]
      → caseV self of
        employee [$vname as v : name, $vemail as v : email] →
          $semail [[replace] $vemail];
          $sname [[replace] $vname]
    )
  (insert person [name [""], email [""], affiliation ["NII"]])
  delete ]
$view / child / :: employee ]
```

**Figure 4.8** Core program of the institutional address example in Figure 4.3

### 4.2.1 Bidirectional Update Normalization

The translation rules for BiFluX surface language to the core XML update language are shown in Figure 4.9 and Figure 4.10. Simple bidirectional updates are translated by a function $[\![ - ]\!]^b_{Upd}(e_s, e_v, \vec{x} = \vec{e})$, where the extra parameters group the `WHERE` clauses of the update into a source selection expression $e_s$, a view selection expression $e_v$ and a sequence of view bindings $\vec{x} = \vec{e}$; these triples are parsed from a set of conditions, according to their source/view tags, by a function $[\![ - ]\!]_{Conds}$.

For special `UPDATE FOR VIEW` statements, the *splitVStmt* function parses

$$[\![\ u\ \texttt{WHERE}\ cs\ ]\!]^b_{Stmt} = [\![\ u\ ]\!]^b_{Upd}([\![\ cs\ ]\!]_{Conds}) \qquad [\![\ s; s'\ ]\!]^b_{Stmt} = [\![\ s\ ]\!]^b_{Stmt}; [\![\ s'\ ]\!]^b_{Stmt}$$

$$[\![\ pat\ \texttt{IN}\ p\ ]\!]^{\texttt{iter}}_{PatPath}(e_S, b) =$$
$$(p\ /\ \texttt{where}\ (\texttt{let}\ pat = \texttt{self}\ \texttt{in}\ e_S))[\texttt{iter}\ (\texttt{caseS}\ \texttt{self}\ \texttt{of}\ pat \to b)]$$

$$[\![\ pat\ \texttt{IN}\ p\ ]\!]^{S}_{PatPath}(e_S, b) =$$
$$(p\ /\ \texttt{where}\ (\texttt{let}\ pat = \texttt{self}\ \texttt{in}\ e_S))[\texttt{caseS}\ \texttt{self}\ \texttt{of}\ pat \to b]$$

$$[\![\ \texttt{DELETE}\ patp\ ]\!]^b_{Upd}(e_S, \texttt{true}, \cdot) = [\![\ patp\ ]\!]^S_{PatPath}(e_S, [\texttt{replace}]())$$

$$[\![\ \texttt{DELETE FROM}\ patp\ ]\!]^b_{Upd}(e_S, \texttt{true}, \cdot) = [\![\ patp\ ]\!]^S_{PatPath}(e_S, \texttt{child}[[\texttt{replace}]()])$$

$$[\![\ \texttt{REPLACE}\ patp\ \texttt{IN}\ p\ \texttt{WITH}\ e'\ ]\!]^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) =$$
$$[\![\ patp\ ]\!]^{\texttt{iter}}_{PatPath}(e_S, \texttt{view}\ \vec{x} := \vec{e}\ \texttt{in}\ \texttt{ifV}\ e_V\ \texttt{then}\ [\texttt{replace}]e'\ \texttt{else}\ \texttt{fail})$$

$$[\![\ \texttt{REPLACE IN}\ patp\ \texttt{WITH}\ e'\ ]\!]^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) =$$
$$[\![\ patp\ ]\!]^{\texttt{iter}}_{PatPath}(e_S, \texttt{view}\ \vec{x} := \vec{e}\ \texttt{in}\ \texttt{ifV}\ e_V\ \texttt{then}\ \texttt{child}[[\texttt{replace}]e']\ \texttt{else}\ \texttt{fail})$$

$$[\![\ \texttt{UPDATE}\ patp\ \texttt{BY}\ s\ ]\!]^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) =$$
$$[\![\ patp\ ]\!]^{\texttt{iter}}_{PatPath}(e_S, \texttt{view}\ \vec{x} := \vec{e}\ \texttt{in}\ \texttt{ifV}\ e_V\ \texttt{then}\ [\![\ s\ ]\!]^b_{Stmt}\ \texttt{else}\ \texttt{fail})$$

$$[\![\ \texttt{UPDATE}\ pat\ \texttt{IN}\ p\ \texttt{BY}\ vs\ \texttt{FOR VIEW}\ pat'\ \texttt{IN}\ p'$$
$$\texttt{MATCHING SOURCE BY}\ p_s\ \texttt{VIEW BY}\ p_v\ ]\!]^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) = p[[b]p']\ \textbf{where}$$
$$((s_{SV}, ms_V, ms_S), p_s{}', p_v{}') =$$
$$(splitVStmt(vs), \texttt{case}\ \texttt{self}\ \texttt{of}\ pat \to p_s, \texttt{case}\ \texttt{self}\ \texttt{of}\ pat' \to p_v)$$
$$b = \texttt{alignkey}\ (\texttt{case}\ \texttt{self}\ \texttt{of}\ pat \to e_S)\ p'_s\ p'_v$$
$$[\![\ s_{SV}\ ]\!]^b_{Stmt}\ (pat, pat', p_s, p_v)\ [\![\ ms_V\ ]\!]^c_{MStmt}(pat')\ [\![\ ms_S\ ]\!]^r_{MStmt}(pat)$$

$$splitVStmt : VStmt \to (Maybe\ Stmt, Maybe\ Stmt, Maybe\ Stmt)$$
$$splitVStmt\ (\texttt{MATCH} \to s) = (Just\ s, Nothing, Nothing)$$
$$splitVStmt\ (\texttt{UNMATCHV} \to s) = (Nothing, Just\ s, Nothing)$$
$$splitVStmt\ (\texttt{UNMATCHS} \to s) = (Nothing, Nothing, Just\ s)$$
$$splitVStmt\ (\texttt{MATCH} \to s\ \texttt{'|'}\ vs) = (Just\ s, ms_V, ms_S)$$
$$\textbf{where}\ splitVStmt\ (vs) = (Nothing, ms_V, ms_S)$$
$$splitVStmt\ (\texttt{UNMATCHV} \to s\ \texttt{'|'}\ vs) = (ms_{SV}, Just\ s, ms_S)$$
$$\textbf{where}\ splitVStmt\ (vs) = (ms_{SV}, Nothing, ms_S)$$
$$splitVStmt\ (\texttt{UNMATCHS} \to s\ \texttt{'|'}\ vs) = (ms_{SV}, ms_V, Just\ s)$$
$$\textbf{where}\ splitVStmt\ (vs) = (ms_{SV}, ms_V, Nothing)$$

**Figure 4.9** BiFluX bidirectional statement normalization. (Part I)

$$[\!|\ \{\}\ |\!]^b_{Stmt} = \texttt{skip}$$
$$[\!|\ u\ |\!]^b_{Stmt} = [\!|\ u\ |\!]^b_{Upd}(\texttt{true},\texttt{true},\cdot)$$
$$[\!|\ \texttt{LET SOURCE}\ pat = p\ \texttt{IN}\ s\ |\!]^b_{Stmt} = \texttt{caseS}\ p\ \texttt{of}\ p \rightarrow\ [\!|\ s\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{LET VIEW}\ pat = e\ \texttt{IN}\ s\ |\!]^b_{Stmt} = \texttt{caseV}\ p\ \texttt{of}\ p \rightarrow\ [\!|\ s\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{LET}\ pat = e\ \texttt{IN}\ s\ |\!]^b_{Stmt} = \texttt{case}\ p\ \texttt{of}\ p \rightarrow\ [\!|\ s\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{SOURCE}\ e_1; cs\ |\!]_{Conds} =$$
$$\quad (e_1 \wedge e_S, e_V, \vec{x} = \vec{e})\ \ \textbf{where}\ \ [\!|\ cs\ |\!]_{Conds} = (e_S, e_V, \vec{x} = \vec{e})$$
$$[\!|\ \texttt{VIEW}\ e_1; cs\ |\!]_{Conds} =$$
$$\quad (e_S, e_1 \wedge e_V, \vec{x} = \vec{e})\ \ \textbf{where}\ \ [\!|\ cs\ |\!]_{Conds} = (e_S, e_V, \vec{x} = \vec{e})$$
$$[\!|\ \texttt{VIEW}\ x_0 := e_0; cs\ |\!]_{Conds} =$$
$$\quad (e_S, e_V, x_0, \vec{x} = e_0, \vec{e})\ \ \textbf{where}\ \ [\!|\ cs\ |\!]_{Conds} = (e_S, e_V, \vec{x} = \vec{e})$$
$$[\!|\ \texttt{SOURCE}\ e\ |\!]_{Conds} = (e, \texttt{true}, \cdot)$$
$$[\!|\ \texttt{VIEW}\ e\ |\!]_{Conds} = (\texttt{true}, e, \cdot)$$
$$[\!|\ \texttt{VIEW}\ x := e\ |\!]_{Conds} = (\texttt{true}, \texttt{true}, x = e)$$
$$[\!|\ \texttt{CASE SOURCE}\ p\ \texttt{OF}\ \{pat_1 \rightarrow s_1 \mid \ldots \mid pat_n \rightarrow s_n\}\ |\!]^b_{Stmt} =$$
$$\quad \texttt{caseS}\ p\ \texttt{of}\ \vec{pat} \rightarrow [\!|\ \vec{s}\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{CASE VIEW}\ e\ \texttt{OF}\ \{pat_1 \rightarrow s_1 \mid \ldots \mid pat_n \rightarrow s_n\}\ |\!]^b_{Stmt} =$$
$$\quad \texttt{caseV}\ e\ \texttt{of}\ \vec{pat} \rightarrow [\!|\ \vec{s}\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{CASE}\ e\ \texttt{OF}\ \{pat_1 \rightarrow s_1 \mid \ldots \mid pat_n \rightarrow s_n\}\ |\!]^b_{Stmt} = \texttt{case}\ e\ \texttt{of}\ \vec{pat} \rightarrow [\!|\ \vec{s}\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{IF SOURCE}\ e\ \texttt{THEN}\ s\ \texttt{ELSE}\ s'\ |\!]^b_{Stmt} = \texttt{ifS}\ e\ \texttt{then}\ [\!|\ s\ |\!]^b_{Stmt}\ \texttt{else}\ [\!|\ s'\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{IF VIEW}\ e\ \texttt{THEN}\ s\ \texttt{ELSE}\ s'\ |\!]^b_{Stmt} = \texttt{ifV}\ e\ \texttt{then}\ [\!|\ s\ |\!]^b_{Stmt}\ \texttt{else}\ [\!|\ s'\ |\!]^b_{Stmt}$$
$$[\!|\ \texttt{IF}\ e\ \texttt{THEN}\ s\ \texttt{ELSE}\ s'\ |\!]^b_{Stmt} = \texttt{if}\ e\ \texttt{then}\ [\!|\ s\ |\!]^b_{Stmt}\ \texttt{else}\ [\!|\ s'\ |\!]^b_{Stmt}$$

**Figure 4.10** BiFluX bidirectional update normalization. (Part II)

a *VStmt* into a matching statement and two optional unmatched-view and unmatched-source statements. Optional unmatched-view statements are translated using a function $[\![-]\!]^{c}_{MStmt}(mpat)$ that takes an extra optional view pattern and returns a core create update; if no `UNMATCHV` clause is defined, the $U$ function of the underlying lens will be evaluated without an original source. Optional unmatched-source statements are translated using a function $[\![-]\!]^{r}_{MStmt}(mpat)$ that takes an extra optional source pattern and returns a core recover update; if no `UNMATCHS` clause is defined, all unmatched source elements are deleted by default.

### 4.2.2 Unidirectional Update Normalization

The syntax of ordinary unidirectional updates that we use in BiFluX differs slightly from that of Flux [15]. For example, there is no `iter` $u$ update that iterates over a sequence by applying the same update $u$, and instead we support arbitrary paths as directions. In our design, iteration occurs automatically at the `child` axis (for updates of the form `child`$[u]$). The two different `let` $x = e$ `in` $u$ and `snapshot` $x$ `in` $u$ updates in Flux, that respectively bind a variable to the result of evaluating an expression under the current environment and bind a variable to the current value of the focus, are both subsumed by our `case` $e$ `of` $\overrightarrow{pat \rightarrow u}$ update, that allows the expression $e$ to depend on the current value of the focus. Simple unidirectional updates are translated by a function $[\![-]\!]^{u}_{Upd}(e)$, where $e$ is the conjunction of all the `WHERE` clauses of the update. The translation rules for normalizing high-level unidirectional updates are shown in Figure 4.11. The normalization for particular create and recover unidirectional updates are given in Figure 4.12.

The translation denotes a partial function from high-level BiFluX to core BiFluX. For example, `INSERT` operation as a unidirectional update is not supported for bidirectional updates, `UPDATE FOR VIEW` is not supported for unidirectional updates, and `CREATE` is only supported under `UNMATCHV` or `UNMATCHS` clauses, respectively. We assume that paths and expressions are expressed in terms of our core languages; this is standard practice as normalization of XQuery expressions or XPath paths can be done independently. To simplify the presentation, we also

$$[\![\, e \,]\!]^u_{Conds} = e$$
$$[\![\, p \,]\!]^u_{PatPath}(u) = p[u]$$
$$[\![\, e; cs \,]\!]^u_{Conds} = e \wedge [\![\, cs \,]\!]^u_{Conds}$$
$$[\![\, pat \texttt{ IN } p \,]\!]^u_{PatPath}(u) = p[\texttt{case self of } pat \rightarrow u]$$
$$[\![\, u \texttt{ WHERE } cs \,]\!]^u_{Stmt} = [\![\, u \,]\!]^u_{Upd}([\![\, cs \,]\!]^u_{Conds})$$
$$[\![\, u \,]\!]^u_{Stmt} = [\![\, u \,]\!]^u_{Upd}(\texttt{true})$$
$$[\![\, \{\} \,]\!]^u_{Stmt} = \texttt{skip}$$
$$[\![\, s; s' \,]\!]^u_{Stmt} = [\![\, s \,]\!]^u_{Stmt}; [\![\, s' \,]\!]^u_{Stmt}$$
$$[\![\, \texttt{IF } e \texttt{ THEN } s \texttt{ ELSE } s' \,]\!]^u_{Stmt} = \texttt{if } e \texttt{ then } [\![\, s \,]\!]^u_{Stmt} \texttt{ else } [\![\, s' \,]\!]^u_{Stmt}$$
$$[\![\, \texttt{LET } pat = e \texttt{ IN } s \,]\!]^u_{Stmt} = \texttt{case } e \texttt{ of } pat \rightarrow [\![\, s \,]\!]^u_{Stmt}$$
$$[\![\, \texttt{CASE } e \texttt{ OF } \{pat_1 \rightarrow s_1 \mid \ldots \mid pat_n \rightarrow s_n\} \,]\!]^u_{Stmt} = \texttt{case } e \texttt{ of } \vec{pat} \rightarrow [\![\, \vec{s} \,]\!]^u_{Stmt}$$
$$[\![\, \texttt{UPDATE } patp \texttt{ BY } s \,]\!]^u_{Upd}(e) = [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[[\![\, s \,]\!]^u_{Stmt}])$$
$$[\![\, \texttt{DELETE } patp \,]\!]^u_{Upd}(e) = [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{delete}])$$
$$[\![\, \texttt{DELETE } patp \,]\!]^u_{Upd}(e) = [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{children}[\texttt{delete}]])$$
$$[\![\, \texttt{REPLACE } patp \texttt{ WITH } e' \,]\!]^u_{Upd}(e) = [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{delete; insert } e'])$$
$$[\![\, \texttt{REPLACE IN } patp \texttt{ WITH } e' \,]\!]^u_{Upd}(e) =$$
$$\quad [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{children}[\texttt{delete; insert } e']])$$
$$[\![\, \texttt{INSERT BEFORE } patp \texttt{ VALUE } e' \,]\!]^u_{Upd}(e) =$$
$$\quad [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{left}[\texttt{insert } e']])$$
$$[\![\, \texttt{INSERT AFTER } patp \texttt{ VALUE } e' \,]\!]^u_{Upd}(e) =$$
$$\quad [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{right}[\texttt{insert } e']])$$
$$[\![\, \texttt{INSERT AS FIRST INTO } patp \texttt{ VALUE } e' \,]\!]^u_{Upd}(e) =$$
$$\quad [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{children}[\texttt{left}[\texttt{insert } e']]])$$
$$[\![\, \texttt{INSERT AS LAST INTO } patp \texttt{ VALUE } e' \,]\!]^u_{Upd}(e) =$$
$$\quad [\![\, patp \,]\!]^u_{PatPath}((\texttt{where } e)[\texttt{children}[\texttt{right}[\texttt{insert } e']]])$$

**Figure 4.11** Unidirectional update normalization.

$$[\![ \{\,\} ]\!]^r_{MStmt}(mpat) = \texttt{delete}$$
$$[\![ \cdot ]\!]^r_{MStmt}(mpat) = \texttt{delete}$$
$$[\![ s ]\!]^r_{MStmt}(\cdot) = [\![ s ]\!]^r_{Stmt}$$
$$[\![ s ]\!]^r_{MStmt}(pat) = \texttt{case self of } pat \rightarrow [\![ s ]\!]^r_{Stmt}$$
$$[\![ \texttt{IF } e \texttt{ THEN } s \texttt{ ELSE } s' ]\!]^r_{Stmt} = \texttt{if } e \texttt{ then } [\![ s ]\!]^r_{Stmt} \texttt{ else } [\![ s' ]\!]^r_{Stmt}$$
$$[\![ \texttt{LET } pat = e \texttt{ IN } s ]\!]^r_{Stmt} = \texttt{case } e \texttt{ of } pat \rightarrow\ [\![ s ]\!]^r_{Stmt}$$
$$[\![ \texttt{CASE } e \texttt{ OF } \{pat_1 \rightarrow r_1 \mid \ldots \mid pat_n \rightarrow r_n\} ]\!]^r_{Stmt} = \texttt{case } e \texttt{ of } \vec{pat} \rightarrow\ [\![ \vec{r} ]\!]^r_{Cases}$$
$$[\![ \texttt{DELETE self} ]\!]^r_{Stmt} = \texttt{delete}$$
$$[\![ \{\,\} ]\!]^c_{MStmt}(mpat) = \cdot$$
$$[\![ \cdot ]\!]^c_{MStmt}(mpat) = \cdot$$
$$[\![ s ]\!]^c_{MStmt}(\cdot) = \texttt{delete;}\, [\![ s ]\!]^c_{Stmt}$$
$$[\![ s ]\!]^c_{MStmt}(pat) = \texttt{case } pat \texttt{ of self} \rightarrow \texttt{delete;}\, [\![ s ]\!]^c_{Stmt}$$
$$[\![ \texttt{IF } e \texttt{ THEN } s \texttt{ ELSE } s' ]\!]^c_{Stmt} = \texttt{if } e \texttt{ then } [\![ s ]\!]^c_{Stmt} \texttt{ else } [\![ s' ]\!]^c_{Stmt}$$
$$[\![ \texttt{LET } pat = e \texttt{ IN } s ]\!]^c_{Stmt} = \texttt{case } e \texttt{ of } pat \rightarrow\ [\![ s ]\!]^c_{Stmt}$$
$$[\![ \texttt{CASE } e \texttt{ OF } \{pat_1 \rightarrow r_1 \mid \ldots \mid pat_n \rightarrow r_n\} ]\!]^c_{Stmt} = \texttt{case } e \texttt{ of } \vec{pat} \rightarrow\ [\![ \vec{r} ]\!]^c_{Cases}$$
$$[\![ \texttt{CREATE } e; s ]\!]^c_{Stmt} = \texttt{insert } e;\, [\![ s ]\!]^u_{Stmt}$$

**Figure 4.12** Create and recover unidirectional update normalization.

assume explicit `SOURCE` and `VIEW` tags, though our implementation is elaborated to implicitly distinguish between source/view/normal expressions, using the additional environment information available at the time of typechecking the core language.

## 4.3 Core Compilation

In last Section, we have explained the core language and the normalization process from the BiFluX surface language to the core language. In this section, we will give a detailed explanation of the compilation from the core language to a bidirectional core update language BiGUL.

### 4.3.1 Overview of Compilaton

The core language is compiled into BiGUL, which is the most complex part of this work since details about XML and bidirectionality are dealt with here. For the address book running example, the normalized core program in Figure 4.8 is compiled into the BiGUL program in Figure 4.13. The compilation (Section 4.3.3) basically consists of five parts: translating the core bidirectionalizable updates to BiGUL operations (Sections 4.3.3, 4.3.3, and 4.3.3), source paths into BiGUL update patterns (Section 4.3.3), view expressions into BiGUL's view rearrangement operation (Section 4.3.3), general expressions into Haskell expressions (Section 4.3.4), and unidirectional updates into Haskell functions. The more interesting part is, naturally, the translation of the bidirectionalizable updates, and we will devote this section to this part. The translation of the unidirectional updates are straightforward and in fact tedious, so we omit them for brevity.

We should emphasize that we intend the compilation rules to serve as the (preliminary) *definition* of BiFluX semantics. That is, instead of defining a semantics for the surface language and then proving that the compilation rules preserve the semantics, we will rely on the intuitive understanding of what BiFluX programs should do —as presented in Section 4.1— and capture that

```
Update (UIn (UVar
  (RearrV
    (λ(Niibook hEmployeelst) → hEmployeelst)
    (Align
      (λhPerson →
        case hPerson of {Person (sName, sEmail, sAffil) → out sAffil ≡ "NII"})
      (λhPerson hEmployee →
        case hPerson of {Person (sName, sEmail, Affil) → sName}
        ≡ case hEmployee of {Employee (vName, vEmail) → vName})
      (Update (UVar
        (CaseS [
          ((λhs → case hs of {Person (sName, (sEmail, sAffil)) → True; _ → False}),
          Normal (RearrS
            (λPerson (sName, (sEmail, sAffil)) → (sName, (sEmail, sAffil)))
            (RearrV (λhv → (hv, ()))
              (RearrV (λ(hv, hvv) → (hv, hvv))
                (CaseV [
                  ((λ(hv, _) → case hv of
                    {Employee (vName, vEmail) → True; _ → False}),
                  RearrV (λ(Employee (vName, vEmail), ()) → (vName, hEmail))
                    (RearrS (λ(sName, (sEmail, sAffil)) → ((sEmail, sName), sAffil))
                      (RearrV (λ(vName, vEmail) → ((vEmail, vName), ()))
                        (Update (UProd
                          (UProd
                            (UVar (Update (UVar
                              (RearrV (λvEmail → vEmail) Replace))))
                            (UVar (Update (UVar
                              (RearrV (λvName → vName) Replace))))
                          (UVar Skip)))))))]))))]))))
      (λ(Employee (vName, vEmail)) → Person ("", ("", "NII")))
      (λ_ → Nothing)))))
```

**Figure 4.13** Compiled BiGUL program of the address example.

$\boxed{\Gamma \vdash_{sp} p \Rightarrow f}$

$$\frac{}{\Gamma \vdash_{sp} x \Rightarrow \lambda upat.genupat(\Gamma, x, upat)} \qquad \frac{}{\{\tau\} \vdash_{sp} \texttt{self} \Rightarrow id}$$

$$\frac{\tau : n\ [\tau_1]}{\{\tau\} \vdash_{sp} \texttt{child} \Rightarrow \texttt{UIn}} \qquad \frac{\tau <: nt}{\{\tau\} \vdash_{sp} :: nt \Rightarrow id}$$

$$\frac{\tau \not<: nt}{\{\tau\} \vdash_{sp} :: nt \Rightarrow const\ (\texttt{UVar Skip})} \qquad \frac{\Gamma \vdash_{sp} p_1 \Rightarrow f_1 \quad \Gamma \vdash_{sp}^{\texttt{iter}} p_2 \Rightarrow f_2}{\Gamma \vdash_{sp} p_1\ /\ p_2 \Rightarrow f_1 \circ f_2}$$

$\boxed{\Gamma \vdash_{sp}^{\texttt{iter}} p \Rightarrow f}$

$$\frac{}{\{()\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow const\ (\texttt{UConst}\ ())} \qquad \frac{\{\tau\} \vdash_{sp} :: nt \Rightarrow f}{\{\tau\} \vdash_{sp}^{\texttt{iter}} :: nt \Rightarrow f}$$

$$\frac{\{\tau\} \vdash_{sp} :: nt \Rightarrow f}{\{\tau^*\} \vdash_{sp}^{\texttt{iter}} :: nt \Rightarrow f} \qquad \frac{\{\tau_1\} \vdash_{sp}^{\texttt{iter}} :: nt \Rightarrow f_1 \quad \{\tau_2\} \vdash_{sp}^{\texttt{iter}} :: nt \Rightarrow f_2}{\{(\tau_1, \tau_2)\} \vdash_{sp}^{\texttt{iter}} :: nt \Rightarrow f_1 \times f_2}$$

$$\frac{\{\alpha\} \vdash_{sp} p \Rightarrow f}{\{\alpha\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow f} \qquad \frac{\{E(X)\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow f}{\{X\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow f}$$

$$\frac{\{\tau_1\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow f_1 \quad \{\tau_2\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow f_2}{\begin{array}{l}\{\tau_1 \mid \tau_2\} \vdash_{sp}^{\texttt{iter}} p \Rightarrow \\ \quad \lambda upat.\texttt{UVar}\ (\texttt{CaseS}\ [(isLeft, \texttt{Normal}\ (\texttt{Update}\ (f_1\ upat))), \\ \qquad\qquad\qquad (isRight, \texttt{Normal}\ (\texttt{Update}\ (f_2\ upat)))])\end{array}}$$

**Figure 4.14** Compilation of source path.

intuition with the compilation rules. Admittedly, it is hard to make a semantics defined by compilation as clear as one hopes for, but such a semantics is usually sufficient for an experimental language. Our main purpose of designing BiFluX is to experiment with the paradigm of bidirectional programming by update, and we expect to make further changes and extensions (some of which will be mentioned in Section 4.5) to the language. We plan to give a better formalization, in particular specifying a semantics for the surface language, after the language is more mature and stabilized.

What we have refrained from saying explicitly up until now is that all of the high-level BiFluX language, the core language, and BiGUL are typed. We have omitted the typing rules for the languages from the paper since they are in general straightforward. The compiler, however, sometimes needs to use type information in a core program to generate appropriate BiGUL code, and hence the compilation rules need to refer to the types. Therefore, before presenting the compilation rules, we first give an account of the type system used by the core language in Section 4.3.2.

### 4.3.2 XML Values and Regular Expression Types

As several other XML processing languages [38, 15, 19], we consider a type system of regular expression types with structural subtyping[2]:

Atomic types
$\alpha ::= \texttt{bool} \parallel \texttt{string} \parallel n[\tau]$
Sequence types
$\tau ::= \alpha \parallel () \parallel \tau \mid \tau' \parallel \tau, \tau' \parallel \tau^* \parallel X$

*Atomic types* $\alpha \in \mathit{Atom}$ are primitive booleans, strings or labeled sequences $n[\tau]$. *Sequence types* $\tau \in \mathit{Type}$ are defined using regular expressions, including empty sequence $()$, alternative choice $\tau \mid \tau'$, sequential composition $\tau, \tau'$, iteration $\tau^*$ or type variables $X$; choice and composition are right-nested. We define the usual $\tau^+ = \tau, \tau^*$ and $\tau^? = \tau \mid ()$. Types can also be recursively defined:

[2]We use $\parallel$ for syntax alternatives in the type grammar to prevent confusion.

Type definitions
$\tau_D ::= \alpha \parallel () \parallel \tau_D \mid \tau'_D \parallel \tau_D, \tau'_D \parallel {\tau_D}^*$
Type signatures
$E \ ::= \cdot \parallel E, \texttt{type}\ X = \tau_D$

*Type definitions* $\tau_D$ are sequences with no top-level variables (to avoid non-label-guarded recursion [19]). A *type signature* $E$ is a set of named type definitions of the form $X = \tau_D$, and is well-formed if no two types have the same name and all type variables in definitions are declared in $E$. We write $E(X)$ for the type bound to $X$ in $E$. Hereafter, we will assume the signature $E$ to be fixed.

In traditional XML-centric approaches [38, 19], values are encoded using a uniform representation that does not record the structure that types impose on values. This "flat" representation is economical and simplifies subtyping, but makes it harder to realize that a value belongs to a type and therefore to integrate regular expression features into functional languages with non-structural type equivalence, such as Haskell or ML. We instead consider a structured representation of values (in line with values of algebraic data types) that keep explicit annotations which, in a way, witness how to parse a flat value as an instance of a type [50]:

Atomic values
$t ::= \texttt{true} \mid \texttt{false} \mid w \mid n[v]$
Forest values
$v ::= t \mid () \mid L\ v \mid R\ v \mid (v, v) \mid [v_0, \ldots, v_n]$

*Atomic values* $t \in \mathit{Tree}$ can be $\texttt{true}, \texttt{false} \in \mathit{Bool}$, strings $w \in \Sigma^*$ (for some alphabet $\Sigma$), or singleton trees $n[v]$ with a node label $n$. *Forest values* $v \in \mathit{Val}$ include the empty sequence $()$, left- $L\ v$ or right- $R\ v$ tagged choices, binary sequences $(v, v)$ and lists of arbitrary length $[v_0, \ldots, v_n]$. The semantics of a type $\tau$ denotes a set of values $[\![\tau]\!]$ that is defined as the minimal solution (formally the least fixed point [38]) of the following set of equations:

$[\![()]\!] \triangleq \{()\} \qquad [\![\texttt{string}]\!] \triangleq \Sigma^*$
$[\![\texttt{bool}]\!] \triangleq \{\texttt{true}, \texttt{false}\} \qquad [\![X]\!] \triangleq [\![E\ (X)]\!]$
$[\![n[\tau]]\!] \triangleq \{n[v] \mid v \in [\![\tau]\!]\}$
$[\![\tau, \tau']\!] \triangleq \{(v, v') \mid v \in [\![\tau]\!], v' \in [\![\tau']\!]\}$

$$[\![\tau \mid \tau']\!] \triangleq \{L\, v \mid v \in [\![\tau]\!]\} \cup \{R\, v \mid v \in [\![\tau']\!]\}$$
$$[\![\tau^*]\!] \triangleq \{[v_0, \ldots, v_n] \mid v_0, \ldots, v_n \in [\![\tau]\!], n \geqslant 0\}$$

In our context, values in the type semantics preserve the type structure. We will denote flat values $ft \in FTree$ and $fv \in FVal$ (dropping left/right tags, parenthesis and list brackets) by:

Flat atomic values
$ft ::= \texttt{true} \mid \texttt{false} \mid w \mid n[fv]$
Flat forest values
$fv ::= () \mid ft, fv$

The notion of subtyping plays a crucial role in XML approaches with regular expression types. A type $\tau_1$ is said to be a *subtype* of $\tau_2$, written $\tau_1 <: \tau_2$, if the flat values belonging to $\tau_1$ are also values of $\tau_2$, i.e. $[\![\tau_1]\!]_{flat} \subseteq [\![\tau_2]\!]_{flat}$. Since we retain a structured representation of values, upcasting a value $v_1$ of type $\tau_1$ into a supertype $\tau_2$ requires more than a proof of subtyping: we must also change $v_1$ into a value $v_2$ that contains the same flat information as $v_1$ but conforms to the structure of $\tau_2$. This problem has been considered in [50], that introduces a subtyping algorithm as a proof system with judgments of the form $\vdash \tau_1 <: \tau_2 \Rightarrow c$, that we treat as a "black box". In BX terms, $c : \tau_2 \leftrightarrow\!\!\!\triangleright\!\!\!\rightarrow \tau_1$ is called a *canonizer* [29], which is a bit like a lens from $\tau_2$ to $\tau_1$ that comprises a total upcast function $ucast : \tau_1 \to \tau_2$, and a partial downcast function $dcast : \tau_2 \to \tau_1$. In our sense, canonizers satisfy two properties stating that they only handle structure:

$$ucast\ v_1 \sim v_1 \qquad \textsc{Up}_\sim$$
$$dcast\ v_2 = v_1 \Rightarrow v_1 \sim v_2 \qquad \textsc{Down}_\sim$$

The equivalence relation $\sim$ used above ignores structure and relates values parsing the same data using different markup, e.g., $L\, v \sim R\, v$; formally,

$$v \sim v' \triangleq flat(v) = flat(v')$$

where the function $flat : Val \to FVal$ flattens a structured value.

### 4.3.3 Compilation of Bidirectionalizable Updates

We can now move on to the compilation rules from the core language to BiGUL. The first three basic operations —`replace`, `skip`, and `fail`— are simply compiled into their counterparts in BiGUL. The rest are explained in the following subsections.

#### Source Paths

The $p[b]$ operation in Section 3.2.2 updates part of the source, and in BiGUL this behavior is implemented by the `Update` operation. The major difference between the two operations is that the former uses a source path to point to the sub-source, while the latter uses an update pattern to decompose the source and execute a sub-update on the sub-source. A source path should thus be compiled into a "pattern with a hole", into which we can fill in the sub-update. Since we can use whatever the host language Haskell offers to describe the compilation, we can simply express the semantics of a source path —i.e. a "pattern with a hole"— as a function mapping a BiGUL update to an update pattern. To be able to define the semantics of source paths compositionally, however, we instead compile source paths to functions mapping a pattern to another pattern, and these functions will be easily composable. After a source path is compiled into such a function, we can apply the function to `UVar` *bigul* where *bigul* is the sub-update. The resulting update pattern can then be supplied as the argument to an `Update` operation.

The compilation rules are shown in Figure 4.14. $\Gamma$ is an environment that maps variables to their type, and always include a special variable '.' for recording the type of the current focus which is useful for paths like `self`. When $\Gamma$ contains only the focus, we write $\{\tau\}$ for $\{(.,\tau)\}$ for simplicity.

The compilation of a variable path ($x$) needs a helper function *genupat* to create an update pattern for the current environment $\Gamma$. As $\Gamma$ may have more than one source variables, others except $x$ in fact will not be updated (the special variable '.' will not be considered during this computation), and thus we use `UVar Skip` to skip them and combine all of them by `UProd`. The construction

of *upat* from $\Gamma$ follows the alphabetical order of the variables in $\Gamma$ in order to keep the generation of patterns consistent. For example, suppose that we have an environment $\Gamma = \{(y, \tau_2), (x, \tau_1)\}$ and a bidirectionalizable update $x[\texttt{replace}]$ in which the path is a variable $x$. Then the generated function is $\lambda upat.(\texttt{UProd}\ upat\ (\texttt{UVar Skip}))$. Variable $y$ will be skipped, and $x$ will be updated using `UVar Replace`.

`self` is compiled into the identity function, and `child` is compiled into `UIn` for updating the children of the current focus. For a node-test path $::nt$, if the current source type is a subtype of $nt$, then the current source is returned; otherwise it is skipped. (*const* is a Haskell function that always return the first argument, ignore the second one.)

Given a path $p_1$ / $p_2$, the result type of $p_1$ can be any one of the types introduced in Section 4.3.2, so we define rules that enumerate all the cases. When the result type is $\tau^*$ and the path $p_2$ is a $::nt$, an *id* function is returned if $\tau$ is a subtype of $nt$, or otherwise it is skipped; when the result type is $\{\tau_1 \mid \tau_2\}$, the translated function involves a case analysis on the current source in order to perform different updates.

For Figure 4.8, there is a source path $\$source$ / `child` / $::person$, which is compiled into $id \circ \texttt{UIn} \circ id$, i.e. `UIn`.

### View Expressions

This subsection gives the compilation rules for view expressions described in Section 3.2.2. In the update direction, a view expression is regarded as a function computing a new view from values bound to the view variables; conversely, in the query direction, we compute the values for the view variables by inverting the function. We thus restrict the forms of expressions that can be used for this purpose, requiring them to be invertible.

In detail: A view expression $e$ is compiled into a lambda expression used as the first argument to a rearrangement (`RearrV`) operation in BiGUL. In order to construct this lambda expression, we first compute a set of paths that are used in $e$. A path can be used in multiple locations in the expression, while two

different paths with the same root variable are not allowed. To be able to check the invertibility of $e$, complex paths are not allowed; instead, the programmer should use pattern matching to fully decompose a view into small pieces. Let us give a counter-example: Suppose that $\$vbookstore$ contains a list of books and each book has a list of authors. The path $\$vbookstore/book/author$ retrieves authors of all the book in $\$vbookstore$ as a single list. This path is not invertible since, in the query direction, there is no way to determine how to divide the list of authors into sublists for the books.

Our next job is to compute a Haskell pattern (*hpat*) from the above set of paths, and an environment ($\Gamma_p$) that maps each path to a fresh Haskell variable name, which will be used for view expression compilation. Figure 4.15 gives the compilation rules for constructing a Haskell expression (*hexp*) from a view expression under the environment $\Gamma_p$. The most interesting case is that when the view expression is a path $p$, it suffices to fetch the corresponding Haskell variable name from $\Gamma_p$ —there is no need to analyze $p$. The view expression in our running example is a path $\$view$ `/ child /` $::employee$, and the compiled lambda expression is $(\lambda(Niibook\ hEmployeelst) \rightarrow hEmployeelst)$, as shown in the third line of Figure 4.13.

A related operation is `view` $x := e$ `in` $b$, which is compiled into a combination of two operations in BiGUL, as shown in Figure 4.16: a view rearrangement `RearrV` to separate $x$ from the rest of the view, and a `Dep` operation stating that the value for $x$ can be computed from the other part.

### Composition

The compilation of composition statement $b_1; b_2$ guarantees that $b_1$ and $b_2$ update different parts of the source by splitting and rearranging the source into three parts, one to be updated by $b_1$, another by $b_2$, and the third part to be kept unchanged.

Specifically, the core composition statement $b_1; b_2$ will be compiled into a source rearrangement (`RearrS`), a view rearrangement (`RearrV`), followed by an `Update` operation. In the compilation rule for composition in Figure 4.16

$\boxed{\Gamma_p \vdash_v e \Rightarrow hexp}$

$$\frac{}{\Gamma_p \vdash_v () \Rightarrow ()} \quad \frac{}{\Gamma_p \vdash_v w \Rightarrow w}$$

$$\frac{}{\Gamma_p \vdash_v \texttt{true} \Rightarrow \texttt{True}} \quad \frac{}{\Gamma_p \vdash_v \texttt{false} \Rightarrow \texttt{False}}$$

$$\frac{}{\Gamma_p \vdash_v p \Rightarrow \Gamma_p(p)} \quad \frac{\Gamma_p \vdash_v e \Rightarrow hexp}{\Gamma_p \vdash_v n[e] \Rightarrow hn\ hexp}$$

$$\frac{\Gamma_p \vdash_v e_1 \Rightarrow hexp_1 \quad \Gamma_p \vdash_v e_2 \Rightarrow hexp_2}{\Gamma_p \vdash_v e_1, e_2 \Rightarrow (hexp_1, hexp_2)}$$

**Figure 4.15** Compilation of view expression.

and Figure 4.17, $s$ denotes the set of source variables, while $s_1$ and $s_2$ are the source variables used in $b_1$ and $b_2$ respectively. We use an abstract notation $s \prec ((s_1, s_2), s_3)$ for the lambda expression that rearranges a tuple of values for the variables in $s$ to a triple whose components are values for the variables in $s_1$, $s_2$, and $s \setminus (s_1 \cup s_2)$ respectively. The view is similarly rearranged. Finally, with an `Update`, the three parts of the source are updated using the three parts of the view (the last of which is empty) by `Replace`, `Replace`, and `Skip`.

To illustrate, let us look at the composition used in the running example:

$\$semail$ [[`replace`] $\$vemail$];
$\$sname$ [[`replace`] $\$vname$]

At this point, the source is of the form $(\$sname, \$semail, \$affiliation)$ (which is a simplified representation for expository purpose), and the source rearranging lambda expression we synthesize is $\lambda(\$sname, \$semail, \$affiliation) \rightarrow ((\$semail, \$sname), \$affiliation)$, since the left-hand side statement updates $\$semail$, the right-hand side statement updates $\$sname$, and $\$affiliation$ is untouched. Similarly the view rearrangement is synthesized, followed by the `Update` operation.

$\boxed{b \Rightarrow \mathit{bigul}}$

$$\frac{}{\texttt{skip} \Rightarrow \texttt{Skip}} \qquad \frac{}{\texttt{fail} \Rightarrow \texttt{Fail}}$$

$$\frac{}{\texttt{replace} \Rightarrow \texttt{Replace}} \qquad \frac{b \Rightarrow \mathit{bigul}}{\texttt{iter}\ b \Rightarrow \texttt{Iter}\ \mathit{bigul}}$$

$$\frac{(s_1, v_1) = \mathit{vars}(b_1) \quad (s_2, v_2) = \mathit{vars}(b_2) \quad b_1 \Rightarrow \mathit{bigul}_1 \quad b_2 \Rightarrow \mathit{bigul}_2}{b_1; b_2 \Rightarrow \texttt{RearrS}\ (s \prec ((s_1, s_2), s \setminus (s_1 \cup s_2)))\ (\texttt{RearrV}\ (v \prec ((v_1, v_2), ()))\ (\texttt{Update}\ (\texttt{UProd}\ (\texttt{UProd}\ (\texttt{UVar}\ \mathit{bigul}_1)\ (\texttt{UVar}\ \mathit{bigul}_2))\ (\texttt{UVar}\ \texttt{Skip}))))}$$

$$\frac{x \in \mathit{dom}(v) \quad v \setminus_x \Rightarrow (\Gamma_{\mathit{var}}, \mathit{hpat}) \quad \Gamma_{\mathit{var}} \vdash_v e \Rightarrow \mathit{hexp} \quad b \Rightarrow \mathit{bigul}}{\texttt{view}\ x := e\ \texttt{in}\ b \Rightarrow \texttt{RearrV}\ (v \prec (v \setminus_x, x))\ (\texttt{Dep}\ \lambda \mathit{hpat}.\mathit{hexp}\ \mathit{bigul})}$$

$$\frac{\Gamma_s \vdash_{sp} p \Rightarrow f \quad b \Rightarrow \mathit{bigul}}{p[b] \Rightarrow \texttt{Update}\ (f\ (\texttt{UVar}\ \mathit{bigul}))} \qquad \frac{b \Rightarrow \mathit{bigul} \quad \vdash_v e \Rightarrow f_e}{[b]e \Rightarrow \texttt{RearrV}\ f_e\ \mathit{bigul}}$$

$$\frac{\Gamma_s \vdash s \Rightarrow (\Gamma_{\mathit{var}}, \mathit{hpat}) \quad \Gamma_{\mathit{var}} \vdash e \Rightarrow \mathit{hexp} \quad b_1 \Rightarrow \mathit{bigul}_1 \quad b_2 \Rightarrow \mathit{bigul}_2}{\texttt{ifS}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2 \Rightarrow \texttt{CaseS}\ [(\lambda \mathit{hpat}.\mathit{boolean}_\tau(\mathit{hexp}), \texttt{Normal}\ \mathit{bigul}_1), (\lambda\_.\texttt{True}, \texttt{Normal}\ \mathit{bigul}_2)]}$$

$$\frac{\Gamma_v \vdash v \Rightarrow (\Gamma_{\mathit{var}}, \mathit{hpat}) \quad \Gamma_{\mathit{var}} \vdash e \Rightarrow \mathit{hexp} \quad b_1 \Rightarrow \mathit{bigul}_1 \quad b_2 \Rightarrow \mathit{bigul}_2}{\texttt{ifV}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2 \Rightarrow \texttt{CaseV}\ [(\lambda \mathit{hpat}.\mathit{boolean}_\tau(\mathit{hexp}), \texttt{Normal}\ \mathit{bigul}_1), (\lambda\_.\texttt{True}, \texttt{Normal}\ \mathit{bigul}_2)]}$$

**Figure 4.16** Compilation of bidirectional updates. (Part I)

$$\frac{\{hs\} \vdash e \Rightarrow hexp \quad b \Rightarrow bigul \quad \vdash_{create} c \Rightarrow f_c \quad \vdash_{recover} r \Rightarrow f_r}{\texttt{alignpos}\ e\ b\ c\ r \Rightarrow \texttt{Align}\ (\lambda hs.hexp)\ (\lambda\_\ \_.\texttt{True})\ bigul\ f_c\ f_r}$$

$$\frac{\begin{array}{c}\{hs\} \vdash e_f \Rightarrow hexp \quad \{hs\} \vdash e_{ms} \Rightarrow hexp_{ms} \quad \{hv\} \vdash e_{mv} \Rightarrow hexp_{mv} \\ b \Rightarrow bigul \qquad \vdash_{create} c \Rightarrow f_c \qquad \vdash_{recover} r \Rightarrow f_r\end{array}}{\texttt{alignkey}\ e_f\ e_{ms}\ e_{mv}\ b\ c\ r \Rightarrow \texttt{Align}\ (\lambda hs.hexp)\ (\lambda hs\ hv.(hexp_{ms} \equiv hexp_{mv}))\ bigul\ f_c\ f_r}$$

$$\frac{\Gamma_s \vdash_{sp} p \Rightarrow f \quad \overrightarrow{\Gamma_s \vdash pat \Rightarrow (hpat, \Gamma_{pat}) \quad b \Rightarrow bigul \quad \Gamma_{pat} \vdash_u a \Rightarrow u}}{\begin{array}{l}\texttt{caseS}\ p\ \texttt{of}\ \overrightarrow{pat \to b \mid a} \Rightarrow \texttt{Update}\ ( \\ \quad f\ (\texttt{UVar}\ (\texttt{CaseS}\ [\overline{\lambda hs.\texttt{case}\ hv\ \texttt{of}\ \{hpat_i \to \texttt{True}; \_ \to \texttt{False}\}}, \\ \quad\quad \overrightarrow{((\texttt{Normal}\ (\texttt{RearrS}\ (s \prec vars(pat_i))\ bigul_i)) \mid (\texttt{Adaptive}\ u_i))}]))))\end{array}}$$

$$\frac{\begin{array}{c}\Gamma_{var} \vdash_v e \Rightarrow hexp \quad \Gamma_{var} \vdash hpat_v \\ \overrightarrow{\Gamma_v \vdash pat \Rightarrow (hpat, \Gamma_{pat}) \quad b \Rightarrow bigul} \quad \texttt{let}\ var_e\ bind\ to\ e\end{array}}{\begin{array}{l}\texttt{caseV}\ e\ \texttt{of}\ \overrightarrow{pat \to b} \Rightarrow \\ \quad \texttt{RearrV}\ (v \prec (vars(e), v \setminus_{vars(e)})) \\ \quad\quad (\texttt{RearrV}\ ((vars(e), v \setminus_{vars(e)}) \prec ((var_e, v \setminus_{vars(e)}))) \\ \quad\quad\quad (\texttt{CaseV}\ [\overline{\lambda(hv, \_).\texttt{case}\ hv\ \texttt{of}\ \{hpat_i \to \texttt{True}; \_ \to \texttt{False}\}}, \\ \quad\quad\quad\quad \overrightarrow{\texttt{RearrV}\ ((vars(pat_i), v \setminus_{vars(e)}) \prec vars(pat_i) \cup v \setminus_{vars(e)})\ bigul_i}]\end{array}}$$

$\boxed{\vdash_{create} c \Rightarrow f_c}$ $\qquad$ $\boxed{\vdash_{recover} r \Rightarrow f_r}$

$$\frac{\vdash_u u \Rightarrow f}{\vdash_{create} u \Rightarrow f} \qquad \frac{}{\vdash_{recover} \texttt{delete}\ u \Rightarrow \lambda\_.\texttt{Nothing}}$$

$$\frac{\{hs\} \vdash e \Rightarrow hexp \quad \vdash_{recover} r_1 \Rightarrow f_1 \quad \vdash_{recover} r_2 \Rightarrow f_2}{\vdash_{recover} \texttt{if}\ e\ \texttt{then}\ r_1\ \texttt{else}\ r_2 \Rightarrow \texttt{if}\ hexp\ \texttt{then}\ f_1\ \texttt{else}\ f_2}$$

$$\frac{\{hs\} \vdash e \Rightarrow hexp \quad \overrightarrow{\vdash pat \Rightarrow (hpat, \Gamma_{pat}) \quad \Gamma_{pat} \vdash_{recover} r \Rightarrow f}}{\vdash_{recover} \texttt{case}\ e\ \texttt{of}\ \overrightarrow{pat \to r} \Rightarrow \lambda hs.\texttt{let}\ hs' = hexp\ \texttt{in}\ \textbf{case}\ hs'\ \textbf{of}\ \{\overrightarrow{hpat \to f(\Gamma_{pat})}\}}$$

**Figure 4.17** Compilation of bidirectional updates. (Part II)

The compiled BiGUL fragment can be found in Figure 4.13.

### Cases and Conditionals

The source case statement is essentially compiled into `CaseS` in BiGUL wrapped in an `Update` due to the need to compile the source path. Each source pattern $pat_i$ is compiled into a Haskell pattern (by the rules shown in Figure 4.18), a boolean function, and a source rearrangement. An adaptive operation $a_i$ is compiled into a plain Haskell function which computes a new source from the current one.

The view case statement, on the other hand, is compiled into `CaseV`, along with "three" view rearrangement operations. The first rearrangement operation splits the view into those used in the view expression and the rest; the second one evaluates the expression, while keeping the rest as it is; in each branch, after matching the result of evaluating the expression with the $pat_i$, the third rearrangement merges the values bound to the variables in the $pat_i$ and the rest back into one view.

The conditional operations `ifS` and `ifV` choose between two statements $b_1$ or $b_2$ according to a boolean expression $e$, and both of them are translated into case statements in BiGUL (`CaseS` and `CaseV` respectively).

### Source-view Alignment

As described toward the end of Section 3.3, the core alignment operations `alignpos` and `alignkey` correspond closely to BiGUL's `Align` operation. The compilation is thus straightforward, turning bidirectionalizable updates into BiGUL programs and expressions and unidirectional updates into functions. Notably, the matching-by-position and matching-by-key variants can be expressed by providing suitable matching predicate functions to `Align`.

$\boxed{\vdash pat \Rightarrow (hpat, \Gamma_{pat})}$

$$\frac{}{\vdash \tau \Rightarrow (\_, \emptyset)} \quad \frac{}{\vdash x \texttt{ as } \tau \Rightarrow (hx, \{(x, hx)\})}$$

$$\frac{}{\vdash () \Rightarrow ((), \emptyset)} \quad \frac{\vdash pat \Rightarrow (hpat, \Gamma_{pat})}{\vdash n[pat] \Rightarrow (hn\ hpat, \Gamma_{pat})}$$

$$\frac{\vdash pat_1 \Rightarrow (hpat_1, \Gamma_{pat_1}) \quad \vdash pat_2 \Rightarrow (hpat_2, \Gamma_{pat_2})}{\vdash pat_1, pat_2 \Rightarrow ((hpat_1, hpat_2), \Gamma_{pat_1} \cup \Gamma_{pat_2})}$$

**Figure 4.18** Pattern Compilation.

### 4.3.4 Compilation of Expressions, Paths and Patterns

Finally, Figure 4.19 and Figure 4.20 shows the rules for compiling expressions and paths. The judgement $\Gamma_{var} \vdash e \Rightarrow hexp$ says that the expression $e$ is compiled into a Haskell expression $hexp$ under the environment $\Gamma_{var}$, which maps from BiFluX variable names to distinct, fresh Haskell variable names. In the rule for element expressions $n[e]$, $hn$ is the Haskell datatype constructor name for the given element $n[e]$ computed from an ambient type environment. A path $p$ is compiled into a Haskell function $f$, which is applied to the current focus. The notation $\Gamma_{var} \lhd (x, hx)$ denotes an environment obtained by removing $x$ from $\Gamma_{var}$ (if any) and then adding $(x, hx)$. Like source paths in Section 4.3.3, we also have another set of translation rules $\Gamma_{var} \vdash_{\texttt{for}} x \texttt{ in } \tau \to p \Rightarrow f$ that enumerate all the types, which are used in the compilation of *for* expressions and paths of the form $p_1$ / $p_2$.

$\boxed{\Gamma_{var} \vdash e \Rightarrow hexp}$

$$\frac{\Gamma_{var} \vdash e \Rightarrow hexp \quad \vdash \tau <: \tau' \Rightarrow c}{\Gamma_{var} \vdash e \Rightarrow ucast\ c\ hexp}$$

$$\frac{\Gamma_{var} \vdash e \Rightarrow hexp}{\Gamma_{var} \vdash n[e] \Rightarrow hn\ hexp} \qquad \frac{\Gamma_{var} \vdash e_1 \Rightarrow hexp_1 \quad \Gamma_{var} \vdash e_2 \Rightarrow hexp_2}{\Gamma_{var} \vdash e_1, e_2 \Rightarrow (hexp_1, hexp_2)}$$

$$\frac{\begin{array}{c}\Gamma_{var} \vdash e_1 \Rightarrow hexp_1 \\ \Gamma_{var} \cup \{(vars(pat), hvars)\} \vdash pat \Rightarrow hpat \\ \Gamma_{var} \cup \{(vars(pat), hvars)\} \vdash e_2 \Rightarrow hexp_2\end{array}}{\Gamma_{var} \vdash \texttt{let}\ pat = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ hpat = hexp_1\ \texttt{in}\ hexp_2}$$

$$\frac{\Gamma_{var} \vdash e_1 \Rightarrow hexp_1 \quad \Gamma_{var} \vdash e_2 \Rightarrow hexp_2}{\Gamma_{var} \vdash e_1 = e_2 \Rightarrow hexp_1 \equiv hexp_2} \qquad \frac{}{\Gamma_{var} \vdash () \Rightarrow ()}$$

$$\frac{\Gamma_{var} \vdash e \Rightarrow hexp \quad \Gamma_{var} \vdash e_1 \Rightarrow hexp_1 \quad \Gamma_{var} \vdash e_2 \Rightarrow hexp_2}{\Gamma_{var} \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow (\textbf{if}\ hexp\ \textbf{then}\ L\ hexp_1\ \textbf{else}\ R\ hexp_2)}$$

$$\frac{\Gamma_{var} \vdash e \Rightarrow hexp \quad \overrightarrow{\Gamma_{var} \vdash pat \Rightarrow (hpat, \Gamma_{pat}) \quad \Gamma_{pat} \cup \Gamma_{var} \vdash e' \Rightarrow hexp'}}{\texttt{case}\ e\ \texttt{of}\ \overrightarrow{pat \rightarrow e'} \Rightarrow \textbf{case}\ hexp\ \textbf{of}\ \{\overrightarrow{hpat \rightarrow hexp'}\}}$$

$$\frac{\Gamma_{var} \vdash e_1 \Rightarrow hexp \quad e_1 : \tau \quad \Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau \rightarrow e_2 \Rightarrow f}{\Gamma_{var} \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \Rightarrow f\ hexp} \qquad \frac{\Gamma_{var} \vdash_p p \Rightarrow f}{\Gamma_{var} \vdash p \Rightarrow f\ \Gamma_{var}(.)}$$

**Figure 4.19** Compilation of Expression and Path. (Part I)

$\boxed{\Gamma_{var} \vdash_p p \Rightarrow f}$

$$\frac{}{\Gamma_{var} \vdash_p w \Rightarrow const\ w}$$

$$\frac{}{\Gamma_{var} \vdash_p \texttt{true} \Rightarrow const\ \texttt{True}} \quad \frac{}{\Gamma_{var} \vdash_p \texttt{false} \Rightarrow const\ \texttt{False}}$$

$$\frac{x \in dom(\Gamma_{var})}{\Gamma_{var} \vdash_p x \Rightarrow const\ \Gamma_{var}(x)} \quad \frac{}{\Gamma_{var} \vdash_p \texttt{self} \Rightarrow id} \quad \frac{}{\Gamma_{var} \vdash_p \texttt{child} \Rightarrow out}$$

$$\frac{\alpha <: nt}{\Gamma_{var} \vdash_p :: nt \Rightarrow id} \quad \frac{\alpha \not<: nt}{\Gamma_{var} \vdash_p :: nt \Rightarrow const\ ()}$$

$$\frac{e : ()}{\Gamma_{var} \vdash_p \texttt{where}\ e \Rightarrow const\ ()} \quad \frac{\Gamma_{var} \vdash \texttt{if}\ e\ \texttt{then self else}\ () \Rightarrow f}{\Gamma_{var} \vdash_p \texttt{where}\ e \Rightarrow f}$$

$$\frac{\Gamma_{var} \lhd (.,\Gamma_{var}(x)) \vdash_p p \Rightarrow f}{\Gamma_{var} \vdash_p x\ /\ p \Rightarrow f} \quad \frac{\begin{array}{c}\Gamma_{var} \vdash_p p_1 \Rightarrow f_1 \quad p_1 : \tau_1 \\ x \notin dom(\Gamma) \quad \Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_1 \to x\ /\ p_2 \Rightarrow f_2\end{array}}{\Gamma_{var} \vdash_p p_1\ /\ p_2 \Rightarrow f_2\ .\ f_1}$$

$\boxed{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau \to p \Rightarrow f}$

$$\frac{}{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ () \to p \Rightarrow const\ ()} \quad \frac{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau \to p \Rightarrow f}{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau^* \to p \Rightarrow map\ f}$$

$$\frac{\begin{array}{c}\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_1 \to p \Rightarrow f_1 \\ \Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_2 \to p \Rightarrow f_2\end{array}}{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_1, \tau_2 \to p \Rightarrow f_1 \times f_2} \quad \frac{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ E(X) \to p \Rightarrow f}{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ X \to p \Rightarrow f}$$

$$\frac{\Gamma_{var} \vdash p \Rightarrow f}{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \alpha \to p \Rightarrow f}$$

$$\frac{\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_1 \to p \Rightarrow f_1 \quad \Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_2 \to p \Rightarrow f_1}{\begin{array}{l}\Gamma_{var} \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_1 \mid \tau_2 \to p \Rightarrow \\ \quad \lambda hexp.\ \textbf{case}\ hexp\ \textbf{of}\ \{L\ hexp_1 \to L\ (f_1\ hexp_1); R\ hexp_2 \to R\ (f_2\ hexp_2)\}\end{array}}$$

**Figure 4.20** Compilation of Expression and Path. (Part II)

## 4.4 Related Work

### 4.4.1 XML Update Languages

Several XML update languages have been proposed, including (among many others) XQuery! [30], FLUX [15] and the standard W3C XQuery Update Facility [66]. Even though the specification style, expressiveness and semantics of the XML updates that can be written may vary significantly, they all focus on updating XML documents in-place, i.e. updating selected parts of an XML document, keeping the remaining parts of the document unchanged. This means that update programs can be seen as unidirectional transformations that insert, delete or replace elements in a source document and produce an updated document conforming to a new target type. XML updates in BIFLUX are different in that they determine how to update a source document (using some view information) while preserving its source type, and this is enforced by the type system.

### 4.4.2 XML View Updating

In [25], the author studies the problem of updating XML views of relational databases by translating view updates written in the XQuery Update Facility into embedded SQL updates. The work of [49] supports updatable views of XML data by giving a bidirectional semantics to the XQuery Core language. The semantic bidirectionalization technique of [54] interprets various XQuery use cases as BXs by encoding them as polymorphic Haskell functions. The Multifocal language [62] allows writing high-level generic XML views that can be applied to multiple XML schemas, producing a view schema and a lens conforming to the schemas. In the four approaches, the programmer writes a view function and the system derives a suitable view update translation strategy using built-in techniques that cannot be configured. In BIFLUX, the programmer writes an update translation strategy directly as an update (over the source) and the system derives the uniquely related query.

### 4.4.3 Bidirectional XML Languages

Many bidirectional programming languages support tree-structured or XML data formats. Two popular bidirectional XML languages are XSugar [13] and biXid [42], which describe XML-to-ASCII and XML-to-XML mappings as pairs of intertwined grammars. While XSugar restricts itself to bijective grammars, biXid programs describe nondeterministic specifications and are thus inherently ambiguous. Most functional bidirectional programming languages are based on lenses [28, 63, 64, 39], and follow a combinatorial style that puts special emphasis on building complex lenses by composition of smaller combinators. Depending on the choice of combinators, lens languages can become very powerful at specifying application-specific behavior [64, 9, 63]. However, their lower-level nature also induces a more cumbersome programming style that makes it impractical and often unintuitive for users to build non-trivial BXs by piping together several small, surgical steps.

BiFluX features a new programming by update paradigm, which enables the high-level syntax of relational languages such as XSugar and biXid while providing a handful of intuitive update strategies. Remember the huge gap between our high-level BiFluX language (pattern matching, procedures, etc.) and the lens-based BiGUL language that gives it semantics. The most significant innovation in BiFluX is thus the declarative surface language used to specify BXs as bidirectional update programs, at a notably higher-level of abstraction than lens-based functions.

## 4.5 Conclusion

In this Chapter, we introduce the syntax and informal semantics of the BiFluX language, give normalization rules to translate a BiFluX program into a core XML update program, and show the compilation rules that compile the core XML update program into a generic bidirectional core update program. We have shown that programming in BiFluX enjoys a better trade-off between the expressiveness and declarativeness of the written bidirectional programs, by

allowing users to write directly, in a friendly notation and at a nice level of abstraction, a view update translation strategy that gives rise to a well-behaved BX.

As future work, we are still seeking to extend BiFluX so as to further increase the flexibility of BiFluX programming. We also plan to provide more static guarantees to BiFluX by incorporating existing path-query static analyses, implement more powerful pattern type inference algorithms to avoid excessive annotations, and extend the class of bidirectional updates that can be written by integrating user-defined lenses for defining source and view focuses. We also plan to improve the efficiency of our prototype for large XML databases by exploring optimizations to the underlying BiGUL language, including incremental update translation.

## 4.6 Discussion

The project of designing and implementing BiFluX has been lasted for more than three years. Since it is our first attempt to design such a complex but user-friendly language for real world data, we encountered many problems (not only because of the XML details, but also the mixture between XML and bidirectional transformation.) which also pushed us to improve the BiFluX language as well as think how to build a clean and concise core for putback-based programming.

BiFluX evolves overtime from February 2013, and our first design of BiFluX [73] was published at the domestic conference of Japan Society for Software and Science (JSSST). After that, we implemented the language based on putlenses [64] and published at the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP) [65]. While the underlying putlenses is complex and hard to maintain, so we directly implemented the bidirectional semantics of the core XML update language (Section 3.2). During the implementation, we found that the core language is a mixture of bidirectional updates (replace, fail, skip, etc.) and XML related features (expressions, and paths) which makes the bidirectional semantics hard to implement clearly since we need to handle the XML details at the same time. So we decided

to extract a clean and concise core language for putback-based bidirectional programming from this core XML update language that completely has nothing related to XML but it is powerful enough that the bidirectional semantics of the core XML update language can be implemented easily. The core language is called BiGUL [46] which is short for bidirectional generic update language that has a clear bidirectional semantics and it is formally verified in Agda by Josh Ko to guarantee that any program written in BiGUL satisfies the well-behavedness properties. Based on this new putback-based core language, we finally reimplemented our BiFluX by compiling the core XML update language to BiGUL and published as a journal article [75] to the Computer Software of Japan Society for Software Science and Technology (JSSST).

# Chapter 5

# A Putback-Based Library for Updatable Views

In this chapter, to explore the powerfulness of the putback-based bidirectional programming, we implement a library for the user to write the *put* function with flexible update policies easily; what is more, a unique *get* function can be derived automatically from the *put* function.

## 5.1 Motivation of Design

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. The problem is that the update translation policies are not unique in many situations. For example, suppose the view is the result of joining two source tables, and we delete one record on the view, we can choose to delete the related record on either source table or both of them. This indicates that it is not possible to determine a proper update policy for this deletion without the user's choice since there are three update (deletion) policies.

As we introduced before, a lens is a well-behaved pair of *get* and *put* functions. Bohannon et al. designed *relational lenses* [12], whose *get* functions are database queries and *put* functions reflect updates on query results back to

| 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 5.1** 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 5.2** Updated view and source table

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, and the *get* semantics of *drop* removes one column from the source table. 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 (even they provide several specific join such as *join_dl* and *join_dr* that specify to delete on the left table or right table, but it still only has limited flexibility.).

To explain the problem, let us look at an example: suppose that a company keeps a table of employee information shown in Figure 5.1a, 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 5.1a yields the view table shown in Figure 5.1b. 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 5.2b.

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 [65, 46] 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 [26].
- BRUL is implemented on top of BIGUL [46], which is a fully formalized

putback-based language. The 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 Chapter is organized as follows: Section 5.2 shows the design of the BRUL library, Section 5.3 gives two typical examples which are described in the relational lenses paper, Section 5.4 explains the implementation of BRUL in detail, Section 5.5 reviews the related work in both relational database and bidirectional transformation, and Section 5.6 gives a conclusion of our work.

## 5.2 BRUL Library

Unlike relational lenses [12] 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. BRUL is designed and implemented based on BIGUL, for the detailed explanation of BIGUL, please refer to Chapter 3.3. The basic library functions are:

*1*) `align`, which updates a source list by a view list through matching part of the source records that satisfy a filter condition with all the view records according to a matching condition between a source and view record, and update the source records with three different update operations for different matching results;

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

2) `unjoin`, which uses a join view to update two sources. We offer three update policies that can be specified as the first argument: `DeleteLeft`, `DeleteRight`, and `DeleteBoth` to delete on the two sources when there is deletion on the view, and the rest three arguments specify the join attributes for the two sources and the view.

Note that we use list to simulate set, since relational tables are normally represented as sets and BIGULdoes not support set yet.

### 5.2.1 Align

The function `align`:

```
align :: (s -> Bool) -> (s -> v -> Bool) ->
  -> BiGUL s v {* Case 1: matched *}
  -> (v -> s) {* Case 2: missing source *}
  -> (s -> Maybe s) {* Case 3: missing view *}
  -> 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 (`b -> a`)) 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 (`a -> Maybe a`)) 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 a b` is similar with `BiGUL a b` except that it can call a function to correct

the source of type *a* if it is inconsistent (say, not satisfying necessary functional dependencies if the source is a table).

Let us use `align` by writing a `brul` program to solve the problem shown in the introduction section:

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

The function we defined is called `uDB` that updates a source table which is a list of `Records` by a view table which is also a list of `Records`. The `Record` is a List type in Haskell, and thus in fact a table is represented as a List of List. Since we use `Record` to represent database record, if we want to retrieve an attribute value from the record, we use the list index operator (`!!`). For example, let `r` represent one record, we can get the location of the record by `r!!2`. (`RString "Tokyo"` simply means a string is wrapped by the constructor `RString`.) The BRUL program above matches the source records that is located at Tokyo with the view records by their names. If they are matched, the source record is replaced by the view record since they have the same structure; if there is a view record that has no corresponding matching source record, create a new source record by the `id` function; if there is a source record that has no corresponding matching view record, delete this source record by return `Nothing`.

### 5.2.2 Unjoin

The function `ujoin`:

```
unjoin :: DeleteFlag
       -> (Record -> a)
       -> (Record -> a)
```

| 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

**Figure 5.3** Source of Join example I

| A | B | C |
|---|---|---|
| a1 | b1 | c1 |
| a3 | b3 | c3 |
| a5 | b5 | c5 |

**(a)** View table

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

**(b)** Updated view table

**Figure 5.4** View of Join example I

```
-> (Record -> a)
-> Brul ([Record], [Record]) [Record]
```

specifies how to update two source tables using a view table when the view table is modified (the view is the join result of the two source tables), by simply giving a deletion policy and three functions to extract the common parts of these three tables. The deletion policy is used to define how to update the source tables when some records are deleted on the view table. We offer three kinds of deletion policies: delete record on the left source table (`DeleteLeft`), delete on the right source table (`DeleteRight`) or delete on both tables (`DeleteBoth`). The rest three arguments are functions with the same output type as the result is the common part of the three tables.

Suppose we have one source table with two attributes A and B shown in Figure 5.3a, and another source table with two attributes B and C shown in Figure 5.3b. Consider the view table shown in Figure 5.4b which is the join of

the two source tables by attribute their common attribute B, and it is updated by deleting record (a3, b3, c3). We can write a BRUL program using `unjoin` to update the two 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` to delete the records on the left table (source table I in Figure 5.3a). Function `fs1`, `fs2` and `fsv` extract the value of attribute B from a record of each table respectively as B is the join attribute. With `uss1`, deletion of record (a3, b3, c3) from the view will lead to deletion of record (a3, b3) from source table I, while the record (b3, c3) in source table II remains unchanged. We can also choose `DeleteRight` to only delete (b3, c3) on table II or `DeleteBoth` to delete both (a3, b3) on table I and (b3, c3) on table II. 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.

## 5.3 Example

The two proposed library functions are powerful to describe various update policies, in this section we use several typical examples from the relational lenses paper [12] to illustrate that BRUL can be used to describe many different update policies, together with an extra join example that shows that we can handle more flexible data.

| 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 5.5** Source table

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

**Figure 5.6** View table

### 5.3.1 Update Single Source

The source table is a track database shown in Figure 5.5 that stores the track's information: track name, release date, rating, album name and the quantity of this album. Both date and rating depend on track, and quantity depends on album. For example, two tracks named Lullaby have the same date and rating, and the quantities of the same album are also the same. The view table in Figure 5.6 extracts part of the records that the quantity is greater than 2, and each record in the view has no rating. Suppose the view table is updated by changing the rating of Lullaby to 4, the album of Lovesong to Disintegration and its corresponding quantity is also modified, and the Trust track is deleted as shown in Figure 5.7. We will show three different update policies written in BRUL in the following sections.

#### Deletion to deletion

The first update program written in BRUL:

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

**Figure 5.7** Updated view table

```
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)
```

The function `u1` is implemented using the library function `align`, which matches part of the source record list that satisfy the filter condition (line 3) with the view record list by the matching condition (line 4). `RType` is our defined datatype that wraps basic Haskell types, and it will be introduced in the next section. `Record` is a list of values of `RType`. The filter condition says that it only retrieves those whose quantity is greater than 2. If a source and view record have the same track and album, then they are matched. Matching the source records with view records have three cases:

- Matching case: we rearrange the view from a four-elements list (`[t,r,a,q]`) to a five-elements list (`[t,_,r,a,q]`) in order to make the view to be matched with the source structurally (line 5-9), and update the corresponding source element in the source record list by `Replace`.
- Unmatched view record case: we create a new source record (line 10), and fill this record with a default date value.
- Unmatched source record case: we delete this source record by return `Nothing`.

| 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 5.8** Updated Source I

Figure 5.8 shows the updated source table after executing the *put* direction of the program `u1` with the source table and updated view table. Even the first record of the source table does not appear in the view, its rating is also updated according to the functional dependencies since it has the same track name with the second record and the second record is updated. 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. Running *get* of the program will get the same view shown in Figure 5.7.

### Deletion to replacement

Here is another update program written in BRUL that uses a different update policy:

```
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, r, a, q] -> [t, d, r, a, RInt 2])
```

Instead of deleting the unmatched source record (Trust, 1992, 4, Wish, 5) in `u1`,

we update it by changing the quantity to 2. Since the view table only contains the records that the quantity is greater than 2, Modifying it to 2 is a valid update and the updated record (Trust, 1992, 4, Wish, 2) will not appear in the view when performing *get*.

### Deletion to replacement with environment

Sometimes we cannot simply update the unmatched source record with a constant value, and we may refer to other information. For example, suppose there is an environment that stores a mapping from album name to quantity of this album, we can update this unmatched source record by retrieving in the environment. Luckily, BRUL supports to use environment to write more flexible update programs:

```
Type Env = Map RType RType
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)
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
```

The function `uSWithEnv` finds the quantity value from the environment by

| 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 5.9** Updated Source II

the album of a record, and updates the quantity of the record with the found value or with a default quantity value 0. For example, if we have an `env` that stores a mapping from album "Wish" to quantity 1, using `u3` to update the source will change the Quantity of unmatched source record "Trust" to 1 as shown in Figure 5.9. Our library function `align` will check that those unmatched source records will not satisfy the filter function after they are updated to guarantee that those records will not appear in the view.

Even the BRUL program `u1`, `u2`, `u3` only describes the *put* behavior, but in fact the *get* direction of these programs implicitly behaves as the same SQL query:

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

It is in fact the same as the one described in the relational lenses paper, which is a composition of the drop operations and selection:

```
drop Date determined by (Track, unknown)
from s as s1;
select from s1 where Quantity > 2 as v
```

### 5.3.2 Update Multiple Sources

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

| 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 5.10** Join example II

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 5.4.4.

To see this, let us consider the tables in Figure 5.10. 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 5.10c. 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.

## 5.4 Implementation

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.

Since BRUL is designed for relational data that is basically set, while BIGUL is for algebraic data types. We start by introducing database table representation in BIGUL , and then explain how to implement BRUL library functions in BIGUL.

### 5.4.1 Relational Database Representation

Before explaining the implementation of two library functions on a relational database (i.e. table), we briefly give a explanation of 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 5.1a 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.

### 5.4.2 Syntax Sugar of BiGUL in Template Haskell

Even BiGUL is powerful to write many useful applications, while the lower-level syntax makes it hard to write and understand, and thus we utilize Template Haskell [] to make a better syntax sugar and implemented Brul based on that. We will briefly introduce several functions that are used in Brul. Reader does not need to fully understand how this is implemented and the main purpose is for explaining the implementation of library functions in brul.

`rearrAndUpdate` rearranges the view by pattern matching on it (the first argument) to construct a new view (the second argument) in order to match with the structure of the source , and finally update the source with this updated view. For example:

```
rearrAndUpdate [p| vs |] [p| _ :vs |] [d| vs = bigul2 |]
```

means view `vs` is rearranged to `(_:vs)` and then the first element of the source is skipped and the rest of the source is updated using `vs` by `bigul2`.

In order to simplify the programming using `Case`, we have several helper notations: `normalV`, `normalSV`, `adaptiveS`, and `adaptiveSV` for writing different case branches.

Both `normalV` and `normalSV` is used to specify conditions for normal case branches. The following is an example of using `normalV`.

```
$(normalV [p| ([],[]) |]) $
    $(rearr [| \([],[]) -> [] |]) Replace
```

The first argument (`[p| ([],[]) |]`) specifies the condition on view which checks whether the view is a pair of empty lists, and the second part is a BiGULupdate (`$(rearr [| \([],[]) -> [] |]) Replace`) that rearranges the view from a pair of empty lists to an empty list and then replace source with

this empty list. `normalV` only has one condition on the view, while the `normalSV` checks both the current source and view, the following example shows that when the source is an empty list and view is a pair of empty lists, then skip this source. View can be reconstructed from this rearrangement operation in the *get* direction.

```
$(normalSV [p| [] |] [p| ([], []) |]) $
    $(rearr [| \([], []) -> () |]) Skip
```

The `Case` operation in BIGULalso support adaptive branches, both `adaptiveSV` and `adaptiveV` are used to specify conditions for adapting the source in the `Case` operator. The function `adaptiveSV` accepts two arguments: the first one is a condition on the source and the second one is a condition on the view. While `adaptiveV` only species a condition for the view. For example:

```
$(adaptiveSV [p| [[]] |] [p| _ |]) $
   (\[[]] _ -> [[],[]])
```

checks that if the source is a list of empty list, and view can be anything, then update the source from a list of one empty list to a list of two empty lists.

### 5.4.3 Align

The *align* function is implemented using the `Case` operator in BIGUL. As discussed in Section 3.3, `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.

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 as a Haskell function:

   ```
   \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

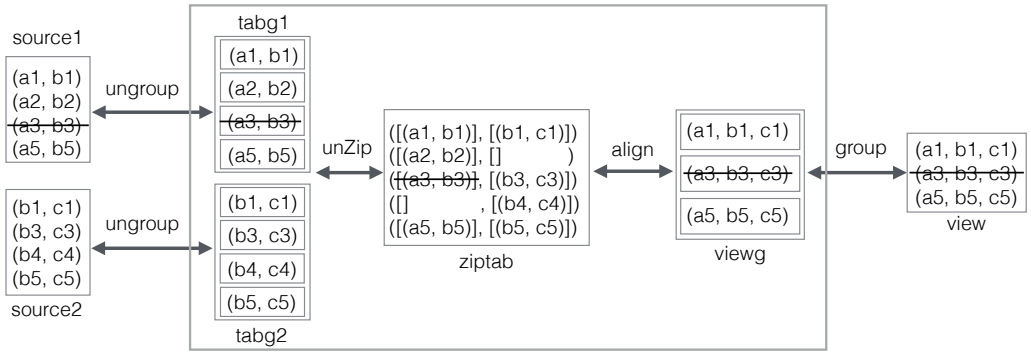


**Figure 5.11** Implementation sketch for unjoin

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.

The implementation of `align` is based on the `Case` constructor in BiGUL, and thus the well-behavedness of `align` is inherited from `BiGUL`. Any program written using `align` is well-behaved.

### 5.4.4 Unjoin

The function *unjoin* describes how to update a pair of tables using its join view. The implementation contains four bidirectional transformations:

```
unjoin :: DeleteFlag -> (Record -> a) -> (Record -> a)
       -> (Record -> a) -> Brul ([Record], [Record]) [Record]
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)`). As demonstrated in Figure 5.11, the basic idea to update the

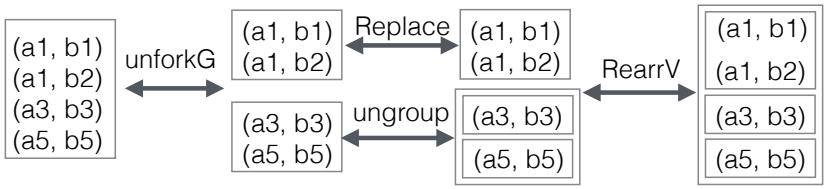


**Figure 5.12** A concrete example for ungroup

two sources (`source1` and `source2`) using the view (`view`), is to use the grouped view (`viewg`) to update the zipped table (`ziptab`) of two grouped sources (`tabg1` and `tabg2`). Let us use Figure 5.11 to illustrate how `uss1` works, in order to understand the implementation of `unjoin` concretely:

- The view is grouped by `group fv` into a list of list (`viewg`) according to attribute B extracted by `fv`. Since (a3, b3, c3) is deleted in the view, it will also be deleted in the `viewg`.
- `alignWithsubJoin flag fs1 fv` aligns `ziptab` with `viewg` by attribute B. Each record in `ziptab` is a pair of lists, and each element in the record has the same value for attribute B. (a3, b3) will be deleted from the pair ([(a3, b3)], [(b3, c3)]), resulting into ([], [(b3, c3)]) since we specify the the flag as `DeleteLeft` in `uss1`.
- `unZip fs1 fs2` generates `tabg1` and `tabg2` from the first and second elements of each pair in `ziptab`.
- `productUnGroup fs1 fs2` ungroups `tabg1` and `tabg2` to get the new sources.

(a3, b3) is deleted in the updated source table `source1`, and another source table `source2` remains unchanged.

Let us explain the implementation details for each operation in the following subsections.

### ungroup

Both `productUnGroup` and `group` are implemented using the transformation `ungroup` :

```
ungroup :: Eq a => (s -> a) -> BiGUL [s] [[s]]
ungroup f =
  Case [ (\_ v -> null v, Normal Replace,
         (\_ _ -> true, Normal
          (unforkG f ;
           RearrV [| \x:xs -> (x, xs) |]
            (Prod Replace (ungroup f))) ]
```

It flattens all sublists of the view list into one as the updated source. Notice that the argument of `ungroup` is a function, which determines the attributes for grouping. We will check whether all the elements in the same sublist in the view have the same value for the common attributes and also the value of two different sublists must be different for those attributes. When view is an empty list, using `Replace` to replace any source list with the empty view; Otherwise, as shown in Figure 5.12, `ungroup` rearranges the view list to a tuple that the first element of the pair is the first element of the view list and the second element is the rest of the view list. The first element of the pair which is a list will be used to update the corresponding elements in the source that is computed by using the *get* of function `unforkG f`; the rest are done by calling `ungroup f` recursively. The transformation `unforkG` is defined as follows:

```
unforkG :: (s -> a) -> BiGUL [s] ([s], [s])
```

In *put* direction, `unforkG` merges two view lists into one as the source and in *get* direction it splits the source into two lists that one list has the same common attribute value while the other list has arbitrary common attribute value which must be different from the value mentioned before.

Based on `ungroup`, we can easily define `productUnGroup`:

```
productUnGroup :: (s1 -> a) -> (s2 -> a)
   -> BiGUL ([s1], [s2]) ([[s1]], [[s2]])
```

```
productUnGroup f1 f2 =
    Prod (ungroup f1) (ungroup f2)
```

which is used to un/group a pair of sources separately.

Notice that ungroup ignores the entire source when putting back the view, and the source and view of ungroup are isomorphic, we can easily define group by swapping the *get* and *put* of ungroup, we get the dual program group by using the emb function:

```
group :: Eq a => (s -> a) -> BiGUL [[s]] [s]
group f = emb (\s -> put (ungroup f) [] s)
              (\_ v -> get (ungroup f) v)
```

The emb function is implemented in BiGUL using Case operator as follows:

```
emb :: Eq v => (s -> v) -> (s -> v -> s) -> BiGUL s v
emb get put = Case
  [ $(normal [| \x y -> get x == y |])$
     $(rearrV [| \x -> ((), x) |])$
       Dep Skip (\x () -> get x)
  , $(adaptive [| \_ _ -> True |]) put ]
```

It accepts a user-defined get and put function, and wraps them as a lens. The well-behavedness of the get and put function is guaranteed by user him/herself. Otherwise, it will always fail.

Since group which is defined based on ungroup is proved to be well-behaved, then we can use them to compose larger program to implement the unjoin function and the unjoin function is also well-behaved guaranteed by the composition of BiGUL.

### unZip

The unZip which is used to implement unjoin is the dual of zip (Note that this zip is different from the built in function zip in Haskell.), we focus on the implementation of zip, and unZip can be implemented easily using zip:

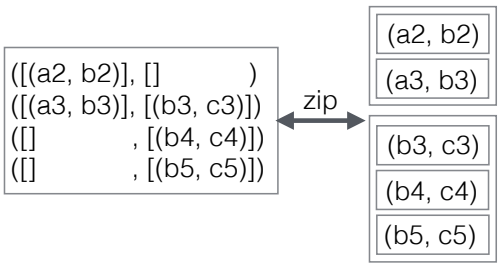


**Figure 5.13** A concrete example for zip

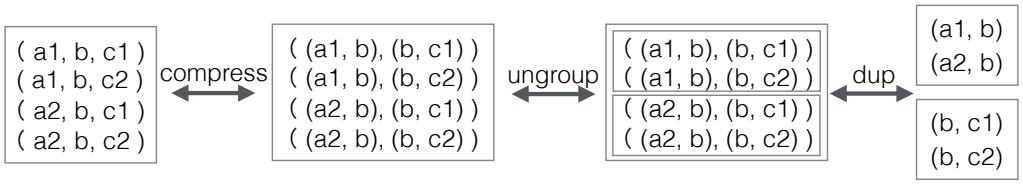


**Figure 5.14** A concrete example for subUnjoin

```
zip :: Ord a => (s1 -> a) -> (s2 -> a) ->
       BiGUL [([s1], [s2])] ([[s1]], [[s2]])
zip f1 f2 =
  Case[(\_ (vls,_) -> null vls, Normal zipLEmpty),
       (\_ (_,vrs) -> null vrs, Normal zipREmpty),
       (\s _ -> null s, Adaptive
                        (\_ _ -> [undefined])),
       (pUnMatchedRight, Normal ...),
       (pUnMatchedLeft, Normal $
          RearrV [| \((vl,vls),vrs)
                 -> ((vl,[]),(vls,vrs)) |] $
             RearrS [| \x:xs -> (x,xs) |]
                (Prod Replace (zip f1 f2))),
       (pMatched, Normal ...)]
  where pUnMatchedLeft (v1, v2) =
        f1 (head (head v1)) < f2 (head (head v2))
```

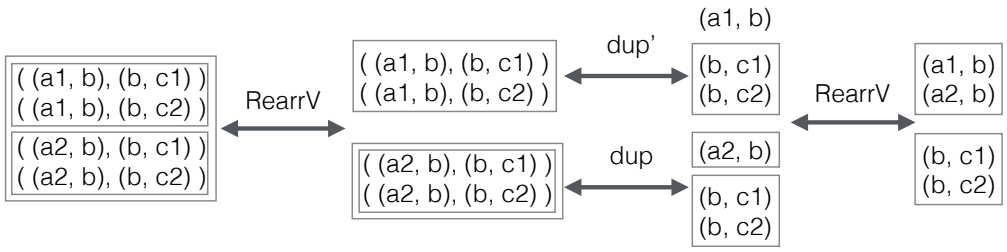


**Figure 5.15** A concrete example for dup

```
pUnMatchedRight ...
pMatched (v1, v2) = ... == ...
```

The `zip` function matches sublists of two lists recursively (the first and second element of the view pair) by their common attributes that are extracted by function `f1` and `f2` (`f1` and `f2` extract common attributes from `s1` and `s2` respectively). If common attributes' value of the two sublists are the same, they will be paired together; otherwise, using an empty list to pair with the smaller one (shown in line 11). `zipLEmpty` create pair for each element in `vls` with the first element as an empty list. `zipREmpty` is similar to `zipLEmpty`. Note that both the first and second elements of the view pair must be in ascending order regarding to their common attributes. We sort the records in the ascending order before fed into `unjoin`, and `ungroup` preserve this order.

Figure 5.13 gives a concrete of how `zip` works. Both `f1` and `f2` are used to extract value of attribute B, and all the records are sorted in ascending order regarding to the subscript of B (e.g. b2 < b3 < b4 < b5). In the first round, it will enter the `pUnMatchedLeft` branch, [(a2, b2)] is picked out and paired with [] since the first sublist for the second element of the view pair is [(b3, c3)]; then [[(a3, b3)]] and [[(b3, c3)] will be paired; after that, the first element of the view pair is empty, it will match the first case of `zip` and use `zipLEmpty` to handle it.

### alignWithsubUnjoin

The `alignWithsubUnjoin` is implemented using `align` which is shown as an alignment between `ziptab` and `viewg` in Figure 5.11. The filter condition for the

`align` function is: `not (null (fst s)) && not (null (snd s))` which means that both parts of the source pair are non-empty, where `s` is each pair in the source list. The source pair in `ziptab` and view element (which is a list) in `viewg` is matched according to their common attributes. As we mentioned before, the results of alignment are three cases: for the matched source pair and view element, we use `subUnjoin` synchronize them; for the unmatched view element, we create a proper source; for the unmatched source pair, it will be updated by either deleting the first one, the second one or both of the pair according to the given flag.

The `subUnJoin` is a special case of `unjoin`, which requires that all values of join attributes must be the same. Since `subUnjoin` is only used in `alignWithsubUnjoin`, this requirement is always satisfied. Figure 5.14 uses a concrete example to show the sketch of `subUnJoin`. Read from the right to the left, it first duplicates the view pair of list (the detail is shown in Figure 5.15), and then flatten the list of list by `ungroup` into a list of pairs, finally compresses each pair into one record by unifying the common attributes.

## 5.5 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) [8, 23]. To deal with different update strategies, Keller [44] 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 [28] originate from the *view-update* problem in relational databases, and many bidirectional languages [28, 9, 35, 60, 39, 29] 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 [26] 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 [64] defines a set of bidirectional combinators that support *putback* style bidirectional programming. BIFLUX [65] is a bidirectional functional update language for XML data; with BIFLUX, the programmer can describe the *putback* function explicitly and the system derives a unique *get* function for free. BIYACC [77] 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 [46] 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 [58], 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.

## 5.6 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.

# Chapter 6

# Conclusion

## 6.1 Summary

In this dissertation, we proposed a new *programming by update* paradigm for *putback*-based bidirectional programming, designed and implemented a bidirectional programming language BiFluX which can be used in many real world applications such as refactoring in Java, and self-adaptive systems, and a bidirectional update library Brul for relational database.

First, we explained the drawback of the current get-based lenses of which the bidirectional programming languages are designed from the perspective of *get*. Programmer only needs to write the *get* transformation, and one "suitable" *put* transformation will be derived from this *get*. While in practice it is impossible to decide which one is "suitable" in general and programmer has no choices of defining the *put* behavior he/she wants. We proposed the new *programming by update* paradigm for *putback*-based bidirectional programming based on the fact that a well-behaved *put* uniquely determines *get*. If we can design a user-friendly *putback* language that lets programmer write the *putback* behavior simply like updates and the language is well-designed to satisfy the well-behavedness, then a unique *get* function can be derived for free.

Then, we introduced our core bidirectional update language for XML structured data which consists of bidirectional updates, XML related expressions

and paths. Since this core language contains many XML related features such as expressions and paths, it is hard to make a clear bidirectional semantics. So we distilled another clean core bidirectional programming language BiGUL from the core of BiFluX language. The initial goal of BiGUL is to serve as the underlying engine for BiFluX. BiGUL is a combinatorial language that consists of a set of *putback* combinators which support composition to compose large bidirectional programs. The language is formalized in the dependently typed programming language Agda to guarantee that any program written in BiGUL is well-behaved. The language is ported into Haskell and served as the core language for the bidirectional programming language BiFluX [75].

BiFluX is our first try of designing *putback*-based bidirectional programming language, which targets XML structured data. XML is widely used in data exchanging on the web, and existing works such as bidirectionalization of XQuery [49] has the same problem of the existing lenses that programmer has no choices of specifying backward behavior. We design the bidirectional programming language BiFluX simply as an update language for programmer to merely write the *putback* of a bidirectional transformation as an update program that uses a view XML to update a source XML. The most significant design of BiFluX is the source-view alignment which aligns a sequence of source elements with a sequence of view elements either by position or by some keys and each aligned source-view pairs are again updated either by a replacement or another subprogram. BiFluX is expressive enough that supports if-then-else condition, case analysis, pattern matching, and recursive definition of a bidirectional update program. BiFluX is firstly normalized into a small core language and then compiled into BiGUL. The well-behavedness of a BiFluX program is guaranteed by the underlying BiGUL.

Brul is a library for relational database which is implemented on top of BiGUL that provides two library functions to simply let programmer write update program by using view table to update source tables. The update program can also be interpreted as a query such as selection, projection and join in the *get* direction. The library function `align` covers the selection and projection, and `unjoin` covers join in relational algebra. Brul also covers the three lens combinators (`selection`, `drop`, and `join`) defined in relational

lenses [12] and in fact the BRUL library is more powerful than relational lenses since BRUL allows programmer to describe flexible *putback* strategies due to the advantage of *putback*-based design of the library functions. Since BRUL is implemented by BIGUL, the well-behavedness of BRUL comes for free.

Our *putback*-based bidirectional programming languages have been used in many real world applications. Cheng et al. [16] use BIFLUX to support reflecting updates on code after refactoring to the original source code. Lionel et al. [55] utilize BIFLUX to implement the BXauthZ which is a policy language to express attribute based rules on XML views. Zhu et al. [77] proposed a declarative bidirectional language BIYACC which supports reflective printing and parsing implemented on top of BIFLUX. Zhao et al. [76] designed a rule-based language *v*Rule for self-adaptation system which is implemented based on BIFLUX. Colson et al. [20] use BIGUL to implement a reusable self-adaptive system which synchronizes the configuration files of different servers such as Apache and Nginx.

## 6.2 Future Work

The research of *putback*-based bidirectional programming just started in three years, even it is promising based on the current results we have gotten, there are still lots of improvements need to be done to make it practical and useful.

### Language

Even the *putback*-based approach has advantages of uniquely determining the forward transformation over the traditional *get*-based approach, programming the backward transformation is still a bit counter-intuitive compared with programming the forward transformation. We already tried to tackle this problem by designing the language more like an update language with bidirectional semantics. We provide basic language constructs that let programmer write programs to express how to update the source using the view which work well when the consistency relation between source and view is not complex. When

the consistency relation between source and view are not straight-forward such as view is the summation of the source list, or view is the reverse of the source list, it can be implemented in our core language BIGUL, but there are two issues: 1. it takes a lot of efforts to implement. 2. it is hard to verify the implementation satisfies the specification. One research direction is to design a more user-friendly general surface *putback*-based bidirectional programming language, which requires a deep understanding of the mechanism and expressiveness of the *putback*. Another research direction is to provide a way to describe the consistency relation between source and view, and check whether the written program will bring the source and view into the specified consistency state or not. Currently, one of our colleges is trying to use Hoare logic to let programmer to define the consistency relation of the written BIGUL program, and check the program will be matched with the specification.

Haskell has many handy packages that can be used directly such as List and Set, if we can provide basic *putback*-based libraries for these basic datatypes, it would be very useful for real world applications.

The *putback*-based design of bidirectional languages also can be employed into other data domains such as JSON which becomes a popular data format for data exchanging, or graphs for bidirectional model transformations which may requires more efforts since graphs may have cycles.

### Optimization

Like *get*-based lenses, composition plays a key role in writing large and complex *putback*-based bidirectional programs which also makes the composition as the bottleneck of the a BX program. Generally, for composition $l_1; l_2$, the backward of $l_2$ requires the execution of forward transformation of $l_1$. There can be a lot of intermediate computations for a long composition sequence. For example the backward transformation of $l_n$ needs all the forward computation from $l_1; l_2; ...; l_{n-1}$. Several optimizations can be done to make the composition more efficient: 1. since the $l_n$ only needs the "original" source computed from $l_{n-1}$, we can compute all the "original" source in advance for all the lenses. 2. for a special category of lenses which is injective, then the backward transformation

do not depend on the original source, we can just chain them together as the forward transformation.

### Applications

Nowadays, people tend to use many social applications to communicate with friends such as Facebook, Twitter, Instagram, Snapchat and so on. The personal information is scattered around, sometimes an unintended action could even lead to a terrible information leak that exposes private personal data to the public. It would be possible to build the personal information as the source, each social service as the view, and then using bidirectional transformations to private limited data access for each service. What is more, by using *putback*-based bidirectional transformation, we can specify the specific update policy for each social services which gives the person full control of his/her personal data. Similarly, our solution can also be used for maintaining hybrid cloud services in order to control the data share between private and public cloud in a company.

Applications are normally isolated, while there are third party applications that the customer want to use to analyze his/her own data to get insights, but it cannot be shared due to security issues and company interest. Bidirectional transformation may be used to allow customer to provide third party applications with limited access to the data to guarantee the security and also give customer self-satisfied services.

# Bibliography


[1] Boomerang. `http://www.seas.upenn.edu/~harmony`. ↰ page 25

[2] Dropbox. `https://www.dropbox.com`. ↰ pages 2 and 11

[3] Google Drive. `https://www.google.com/drive`. ↰ page 2

[4] GRoundTram. `http://www.biglab.org`. ↰ page 26

[5] Harmony: A synchronization framework for heterogeneous tree-structured data. `https://alliance.seas.upenn.edu/~harmony/old`. ↰ pages 13 and 25

[6] Microsoft OneDrive. `https://onedrive.live.com`. ↰ page 2

[7] The XML Bookmark Exchange Language (XBEL). `http://pyxml.sourceforge.net/topics/xbel`. ↰ page 2

[8] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981. ↰ pages 9, 10, 24, and 119

[9] D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 193–204. ACM, 2010. ↰ pages 26, 92, and 119

[10] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 51–63. ACM, 2003. ↰ page 32

[11] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 407–419. ACM, 2008. ↰ page 25

[12] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: A language for updatable views. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 338–347. ACM, 2006. ↰ pages 25, 95, 98, 102, and 125

[13] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4-5):385–406, 2008. ↰ page 92

[14] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *The VLDB Journal*, 9(1):76–110, Mar. 2000. ↰ pages 12 and 26

[15] J. Cheney. FLUX: functional updates for XML. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 3–14. ACM, 2008. ↰ pages 6, 32, 40, 55, 68, 72, 78, and 91

[16] X. Cheng, Y. Chen, Z. Hu, T. Zan, M. Liu, H. Zhong, and J. Zhao. Supporting selective undo for refactoring. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016. ↰ pages 7 and 125

[17] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, chapter JTL: A Bidirectional and Change Propagating Transformation Language, pages 183–202. Springer Berlin Heidelberg, 2011. ↰ page 27

[18] J. Clark and S. DeRose. XML path language (XPath) version 1.0. 1999. ↰ pages 33 and 34

[19] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of XML queries. *Journal of Functional Programming*, 16(4-5):621–661, 2006. ↰ pages 33, 78, and 79

[20] K. Colson, R. Dupuis, L. Montrieux, Z. Hu, S. Uchitel, and P.-Y. Schobbens. Reusable self-adaptation through bidirectional programming. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 4–15, New York, NY, USA, 2016. ACM. ↰ pages 7 and 125

[21] J. Cunha, J. P. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva. Bidirectional transformation of model-driven spreadsheets. pages 105–120, 2012. ↰ page 27

[22] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proceedings of the 2nd International Conference on Model Transformation*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009. ↰ pages 10, 31, and 65

[23] 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. ↰ pages 10, 24, and 119

[24] Z. Diskin. *Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems*, chapter Algebraic Models for Bidirectional Model Synchronization, pages 21–36. Springer Berlin Heidelberg, 2008. ↰ page 26

[25] L. Fegaras. Propagating updates through XML views using lineage tracing. In *Proceedings of the 26th International Conference on Data Engineering*, pages 309 –320. IEEE, 2010. ↰ page 91

[26] 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. ↰ pages 4, 19, 97, and 120

[27] J. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, December 2009. ↰ page 21

[28] 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 TOPLAS*, 29(3):17, 2007. ↰ pages 3, 11, 12, 24, 25, 31, 67, 92, and 119

[29] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 383–396. ACM, 2008. ↰ pages 26, 80, and 119

[30] G. Ghelli, C. Ré, and J. Siméon. XQuery!: An XML query language with side effects. In *Current Trends in Database Technology – EDBT 2006: EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Revised Selected Papers*, volume 4254 of *LNCS*, pages 178–191. Springer, 2006. ↰ pages 12, 32, 68, and 91

[31] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988. ↰ pages 10 and 24

[32] S. Hidaka, K. Asada, Z. Hu, H. Kato, and K. Nakano. Structural Recursion for Querying Ordered Graphs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 305–318, New York, NY, USA, 2013. ACM. ↰ page 26

[33] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010. ↰ pages 12 and 26

[34] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–384. ACM, 2011. ↰ page 26

[35] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 495–508. ACM, 2012. ↰ pages 26 and 119

[36] H. Hosoya and B. Pierce. Regular Expression Pattern Matching for XML. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80. ACM, 2001. ↰ page 33

[37] H. Hosoya and B. C. Pierce. Xduce: A Statically Typed XML Processing Language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003. ↰ page 33

[38] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 11–22. ACM, 2000. ↰ pages 32, 78, and 79

[39] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 178–189, New York, NY, USA, 2004. ACM. ↰ pages 26, 92, and 119

[40] Z. Hu, A. Schurr, P. Stevens, and J. F. Terwilliger. Dagstuhl seminar on bidirectional transformations (bx). *SIGMOD Rec.*, 40(1):35–39, July 2011. ↰ page 10

[41] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: A QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 719–720, New York, NY, USA, 2006. ACM. ↰ pages 11 and 26

[42] S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 201–214. ACM, 2006. ↰ pages 27, 64, and 92

[43] A. Keller. *Updating Relational Databases Through Views*. PhD thesis, Stanford University, 1985. ↰ pages 6, 10, and 24

[44] A. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 467–474. Morgan Kaufmann Publishers, 1986. ↰ pages 6, 24, and 119

[45] K. Kelly. *The Inevitable*. Viking, 2016. ↰ page 1

[46] 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*. ACM, 2016. ↰ pages 6, 40, 94, 97, and 120

[47] J. A. Larson and A. P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145 – 168, 1991. ↰ pages 10 and 24

[48] D. Liu, Z. Hu, and M. Takeichi. Bidirectional Interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 21–30, New York, NY, USA, 2007. ACM. ↰ page 12

[49] D. Liu, Z. Hu, and M. Takeichi. An expressive bidirectional transformation language for XQuery view update. *Progress in Informatics*, 10:89–130, 2013. ↰ pages 26, 91, and 124

[50] K. Z. M. Lu and M. Sulzmann. An Implementation of Subtyping Among Regular Expression Types. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004. ↰ pages 79 and 80

[51] N. Macedo and A. Cunha. *Proceedings of 16th International Conference on Fundamental Approaches to Software Engineering*, chapter Implementing QVT-R Bidirectional Model Transformations Using Alloy, pages 297–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ↰ page 27

[52] S. Marlow (editor). Haskell 2010 language report, 2010. ↰ page 4

[53] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 47–58. ACM, 2007. ↰ page 26

[54] K. Matsuda and M. Wang. Bidirectionalization for Free with Runtime Recording: Or, a Light-weight Approach to the View-update Problem. In

*Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 297–308. ACM, 2013. ↰ page 91

[55] L. Montrieux and Z. Hu. Towards Attribute-Based Authorisation for Bidirectional Programming. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pages 185–196, New York, NY, USA, 2015. ACM. ↰ pages 7 and 125

[56] K. Nakano, Z. Hu, and M. Takeichi. Consistent web site updating based on bidirectional transformation. *International Journal on Software Tools for Technology Transfer*, 11(6):453–468, 2009. ↰ page 26

[57] U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007. ↰ page 48

[58] U. Norell. Dependently Typed Programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2009. ↰ pages 40 and 120

[59] O. M. G. (OMG). MOF 2.0 QVT Final Adopted Specification v1.2, 2015. OMG Adopted Specification formal/2016-02. ↰ pages 11 and 27

[60] H. Pacheco and A. Cunha. Generic point-free lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction*, MPC '10, pages 331–352. Springer, 2010. ↰ pages 26 and 119

[61] H. Pacheco and A. Cunha. Calculating with lenses: optimising bidirectional transformations. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 91–100. ACM, 2011. ↰ page 26

[62] H. Pacheco and A. Cunha. Multifocal: A Strategic Bidirectional Transformation Language for XML Schemas. In *Proceedings of the 5th International Conference on Theory and Practice of Model Transformations*, volume 7307 of *LNCS*, pages 89–104. Springer, 2012. ↰ page 91

[63] H. Pacheco, A. Cunha, and Z. Hu. Delta Lenses over Inductive Types. In *the 1st International Workshop on Bidirectional Transformations*, volume 49 of *Electronic Communications of the EASST*, 2012. ↰ page 92

[64] H. Pacheco, Z. Hu, and S. Fischer. Monadic Combinators for "Putback" Style Bidirectional Programming. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 39–50. ACM, 2014. ↰ pages 92, 93, and 120

[65] H. Pacheco, T. Zan, and Z. Hu. BiFluX: A Bidirectional Functional Update Language for XML. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*, pages 147–158. ACM, 2014. ↰ pages 6, 93, 97, and 120

[66] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. XQuery update facility 1.0. W3C Recommendation. `http://www.w3.org/TR/xquery-update-10/`, March 2011. ↰ pages 33 and 91

[67] I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano. *Proceedings of the 4th International Conference Theory and Practice of Model Transformations*, chapter Toward Bidirectionalization of ATL with GRoundTram, pages 138–151. Springer Berlin Heidelberg, 2011. ↰ page 26

[68] T. Sheard and S. Peyton Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM, 2002. ↰ page 43

[69] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, pages 1–15. Springer, 2007. ↰ pages 11 and 19

[70] S. Vansummeren. Type inference for unique pattern matching. *ACM TOPLAS*, 28(3):389–428, 2006. ↰ page 33

[71] Wikipedia. Netscape (web browser) — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 11-May-2016]. ↰ page 2

[72] T. Zan, L. Lu, H.-S. Ko, and Z. Hu. Brul: A Putback-Based Bidirectional Transformation Library for Updatable Views. CEUR-WS, 2016. ↰ page 6

[73] T. Zan, H. Pacheco, and Z. Hu. BiFluX: A Bidirectional Functional Update Language for XML. *Proceedings of the 30th Conference of Japan Society for Software and Science*, 2013. ↰ pages 6 and 93

[74] T. Zan, H. Pacheco, and Z. Hu. Writing Bidirectional Model Transformations As Intentional Updates. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 488–491. ACM, 2014. ↰ page 6

[75] T. Zan, H. Pacheco, H.-S. Ko, and Z. Hu. BiFluX: A Bidirectional Functional Update Language for XML. Computer Software of Japan Society for Software Science and Technology, 2016. ↰ pages 6, 94, and 124

[76] T. Zhao, T. Zan, H. Zhao, Z. Hu, and Z. Jin. A Novel Approach to Goal-oriented Adaptation with View-based Rules. Technical Report GRACE-TR 2016-01, GRACE Center, National Institute of Informatics, 2016. ↰ pages 7 and 125

[77] Z. Zhu, H.-S. Ko, P. Martins, J. Saraiva, and Z. Hu. BiYacc: Roll Your Parser and Reflective Printer into One. *Bidirectional Transformations*, page 43, 2015. ↰ pages 7, 120, and 125