BiFluX: A Bidirectional Functional Update Language for XML

Tao Zan Hugo Pacheco Hsiang-Shang Ko Zhenjiang Hu

Different XML formats are widely used for data exchange and processing, being often necessary to mutually convert between them. Standard XML transformation languages, like XSLT or XQuery, are unsatisfactory for this purpose since they require writing a separate transformation for each direction. Existing *bidirectional transformation* languages mean to cover this gap, by allowing programmers to write a single program that denotes both transformations. However, they often 1) induce a more cumbersome programming style than their traditionally unidirectional relatives, to establish the link between source and target formats, and 2) offer limited configurability, by making implicit assumptions about *how* modifications to both formats should be translated that may not be easy to predict.

This paper proposes a bidirectional XML update language called BiFluX (BIdirectional FunctionaL Updates for XML), inspired by the Flux XML update language. Our language 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. We introduce a core BiFluX language, and translate it into a formally verified bidirectional update language BiGUL to guarantee a BiFluX program is well-behaved.

1 Introduction

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 [6] 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. A variety of bidirectional languages have emerged over the last 10 years to support bidirectional applications in the most diverse computer science disciplines [6], including functional programming, software engineering, and databases. These languages come in different flavors, including many focused on the transformation of tree-structured data with a particular application to XML documents [14] [3] [16] [7] [20] [9] [13], and can be classified into three main paradigms. The first *relational* paradigm [14] [3] prescribes writing a declarative (non-deterministic) consistency relation between two formats, from which a suitable BX is automatically derived. The second *bidirectionalization* paradigm [16] [7] [18] asks users to write a transformation in a traditional unidirectional language,

This article is an extended version of the conference paper entitled "BiFluX: A Bidirectional Functional Update Language for XML" presented at PPDP 2014.

BiFluX: XML に対する双方向的関数型更新言語

ザン涛, 胡振江, 総合研究大学院大学複合科学研究科, SOKENDAI, Japan.

フゴ パチェコ, ミーニョ大学, University of Minho, Portugal.

柯向上, 胡振江, 国立情報学研究所, National Institute of Informatics, Japan.

コンピュータソフトウェア, Vol.0, No.0 (0), pp.0–0.

[研究論文] 2015 年 11 月 30 日受付.

which plays the role of a functional consistency relation. The last *combinatorial* paradigm [20] [9] [13] encompasses the design of a domain-specific bidirectional language in which each combinator denotes a well-behaved BX, allowing users to write correct-by-construction programs by composition.

As most interesting examples of BXs 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 BX— 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. Even for combinatorial languages, which have the theoretical potential to fully specify the behavior of a BX [8] [22], their lower-level programming style requires significant effort and expertise from users to write intricate BXs via the composition of simple, concrete steps; they also scale badly for large formats, since one must explicitly describe how a BX transforms whole documents, including unrelated parts.

In this paper, we propose a novel *bidirectional programming by update* paradigm, in which the programmer writes an update program that describes how to update a source document to embed information from a target document, and the system derives a query from source to target that expresses the consistency between both documents. Such a *bidirectional update* describes the relationship between source and target documents 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* [8], 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. This paper demonstrates that a *bidirectional update language*, featuring a hybrid programming style, can render bidirectional programming more user friendly.

From a linguistic perspective, the main contribution of this paper is conceptual: we propose the idea of extending an update language with bidirectional features to write, directly and at a nice level of abstraction, a view update translation strategy which bundles all the necessary pieces to build a BX. Concretely, we design BiFluX, a declarative and expressive language for the bidirectional updating of XML documents that is deeply inspired by Flux [4], a simple and well-designed functional XML update language. We lift unidirectional Flux updates to bidirectional BiFluX updates by imbuing them with an additional notion of view. Reading updates as BXs will motivate a few language extensions to original Flux, and require a suitable bidirectional semantics and extra static conditions on BiFluX programs to ensure that they build well-behaved BXs.

This is an extended version of our original paper [23]. We have made two improvements on the original BiFluX: *1*) A new *adaption* mechanism, which is useful when the source is incompatible with the view, and needs to be adapted to a compatible one while still keeping necessary information of the original source. This mechanism plays an important role in Zhu et al.'s work [26]. *2*) BiFluX was implemented by a putback-based combinatorial language called putlenses [22]. However, the translation was hard to understand and the BX properties were not formally verified. We compile BiFluX into our newly developed bidirectional core update language BiGUL [15], which has been fully formally verified in the dependently typed language Agda [19].

The rest of the paper is organized as follows. We explain the novel features of BiFluX with a representative running example in Section 2. High-level BiFluX programs are first normalized (desugared) into the core BiFluX language, presented in Section 3, which is a streamlined version of the high-level BiFluX syntax and is easier to compile. Section 4 introduces the underlying engine BiGUL, and Section 5 discusses the compilation of core BiFluX to BiGUL. Section 6 compares our approach with related work on bidirectional and XML programming, and Section 7 concludes with a sum-

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

Fig. 1: A simple address book DTD.

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

Fig. 2: An NII address book DTD.

mary of the main ideas and directions for future work.

2 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.

2.1 Our running example

Consider a typical address book whose format is represented by the DTD from Figure 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 2. We have a view that simply keeps 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:

```
<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 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

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

```
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"
```

Fig. 3: BiFluX update for the institutional address book example.

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.

2.2 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 3. The syntax for BiFluX's main constructs is defined in Figure 4, which is based on Flux [4], 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 (`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.

2.2.1 Procedure

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

`PROCEDURE` $P($ *Var* `AS` τ, *Var* `AS` $\tau) =$ *Stmt*

The first argument is the source and the second one is the view. τ is a regular expression type (whose details will be presented in Section 5.1).

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.

2.2.2 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

```
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

Conds   ::= Expr [; Conds]
          | Var := Expr [; Conds]
Cases   ::= Pat → Stmt
          | Pat → ADAPT SOURCE BY Stmt
          | Cases '|' Cases
VStmt   ::= { VStmt } | VUpd
          | VUpd '|' VUpd
VUpd    ::= MATCH → Stmt
          | UNMATCHS → Stmt
          | UNMATCHV → Stmt
Match   ::= MATCHING BY Path
          | MATCHING SOURCE BY Path
                     VIEW BY Path
PatPath ::= [Pat IN] Path
```

Fig. 4: Concrete syntax of BiFluX updates.

`$source/person` extracts all the three people under `$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".

2.2.3 Pattern

BiFluX supports pattern matching, which is a very useful feature of XML transformation languages like XDuce [12] or CDuce [2], allowing matching tree patterns against the input data to transform it into an output of different shape. Typical XML update languages like XQuery! [10] or Flux [4] 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 in BiFluX can be used to guide the update based on the structure of the data.

Our pattern language follows that of XDuce [12]:

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

A pattern can be a variable pattern restricted by a regular expression type τ, a type pattern τ, 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 [25]. 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 [11], but these simple patterns suffice for our practical needs.

For example, the pattern used in our running example:

```
person[$sname AS s:name, $semail AS s:email,
       $affil AS s:affiliation]
```

decomposes a `person` element into three parts: `$sname`, `$semail` and `$affil`.

2.2.4 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 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.

2.2.5 Flux operations

Some update operations are inherited from Flux, which can update single XML trees or update the children of the selected tree. For example, the replacement statement

```
REPLACE $semail WITH $vemail
```

replaces an `email` element pointed by `$semail` is replaced by another `email` element `$vemail`, while the statement

```
REPLACE IN $semail WITH 'zantao@nii.ac.jp'
```

updates the string wrapped inside the `email` element pointed to by `$semail` with the string `'zantao007@nii.ac.jp'`. For other operations like insertion (`INSERT BEFORE/AFTER`) and deletion (`DELETE`, `DELETE FROM`), the reader is referred to Flux [4].

2.2.6 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 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.

2.3 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 functions may be partial. For the running example in Figure 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 2.1 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

```
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"
```

Fig. 5: Another update Strategy in BiFluX.

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 [14], 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 [6].

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

2.4 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 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 actu-

ally 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 5.

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 [8]), 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 [8] 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.

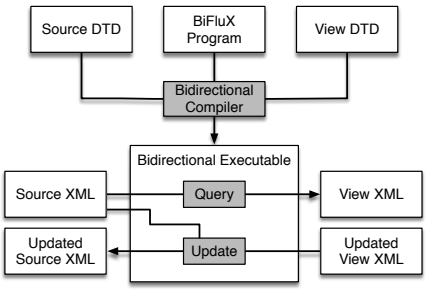


Fig. 6: Architecture of the BiFLuX framework.

2.5 General framework

The general architecture of our bidirectional updating framework is illustrated in Figure 6. 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. 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.

3 Core Language

The high-level language presented in Section 2 has a user-friendly syntax, but its operations are overlapping and complex. Following the design of Flux and as standard for many other languages, we introduce a core update language of canonical operations whose semantics are easier to define, and high-level BiFLuX are normalized into this core language.

To give the reader a taste of the core language, the core program for the address book running ex-

```
$source / child / :: person
[ [ alignkey
  (case self of
    person [$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 ]
```

Fig. 7: Core program of the institutional address example in Figure 3

ample is shown in Figure 7. Roughly speaking, it traverses down a source path $*source* / `child` / :: *person* to get a list of people, then retrieves all the employees by the view path $*view* / `child` / :: *employee*, and finally uses a six-argument `alignkey` operation to update this list of people by the list of employees.

3.1 Bidirectionalizable updates

Unlike conventional XML update languages, our core update language mainly consists of bidirectionalizable updates. Their names suggest what their update semantics are, but in Section 5 they will be interpreted as BXs [15] 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:

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

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, in Figure 7, we have a source path $*source* / `child` / :: *person* followed by a slightly complex bidirectional update b. After evaluation of this source path, the source will be a list of people, and b is run on this list.

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. In Figure 7, after evaluation of the expression on the last line, i.e. the path $*view* / `child` / :: *employee*, the view for the inner `alignkey` will be a list of employees.

Composition $b_1; b_2$ updates a part of the source with b_1, and another part of the source with b_2. To guarantee the UpdateQuery property, the same part of the source cannot be updated twice. This is enforced by requiring that the two statements b_1 and b_2 update different source variables. For example, in Figure 7, we have a composition state-

ment 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 ($\$vname$).

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}. In Figure 7, the three expressions e_f, e_{ms}, and e_{mv} for the alignment operation (`alignkey`) are `case` expressions. Since e_f and e_{ms} are evaluated on the source, the `self` in e_f and e_{ms} refers to the current source element (i.e. a person), while the `self` in e_{mv} refers to the current view element (i.e. an employee).

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. 3. Recover statements r are enriched unidirectional updates of the form:

$$r ::= \texttt{delete} \mid u \mid \texttt{if}\ e\ \texttt{then}\ r\ \texttt{else}\ r' \mid \texttt{case}\ e\ \texttt{of}\ \overrightarrow{pat \to r}$$

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.

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 source adaptation mechanism introduced in Section 2. 2. 6 is handled by `caseS` at this level, which can have adaptive branches. Case statements are used in Figure 7 for decomposing the source and view by pattern matching.

There are still some operations that do not appear in Figure 7. The operation `iter` b embeds the same view into each element of a source sequence. A procedure call $P(p_s, e_v)$ updates the subsource 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.

3. 2 Expressions and paths

In BiFluX, updates instrumentally use XQuery expressions, XPath paths and XDuce 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.

3. 2. 1 General expressions and paths

We write expressions e in a minimal XQuery-like language, which is a variant of the μXQ core language proposed in [5]:

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

Note that there are no case expressions in μXQ as they can be emulated by conditional expressions; we extend our expression language with case expressions to simplify the translation from the BiFluX surface language. We differentiate paths p in a core path language that represents a minimal dialect of XPath:

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

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

3. 2. 2 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. 1 is as follows:

$e ::= () \mid e, e' \mid n[e] \mid p \mid w \mid \texttt{true} \mid \texttt{false}$

$p ::= x \mid \texttt{self} \mid \texttt{child} \mid ::nt \mid p \ / \ p'$

3. 2. 3 Source paths

A source path p, as used in the statement $p[b]$, narrows the source focus to only part of the current source. Like in FLUX, 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. 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:

$p ::= x \mid \texttt{self} \mid \texttt{child} \mid :: nt \mid p \ / \ p'$

3. 3 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 [4]:

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

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 given an empty sequence as focus; 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. 4 BiFluX to Core Update Normalization

The translation from the high-level BIFLUX language to the core language is usually referred to as *normalization* in languages like XQuery and FLUX. Since most translation rules are straightforward (which can be seen by comparing Figures 3 and 7), we will only explain the special `UPDATE FOR VIEW` statements as shown in Figure 8. The *splitVStmt* function parses a *VStmt* into a matching statement and two optional unmatched-view and unmatched-source statements. The matching statement is translated using $[\![-]\!]^b_{MStmt}(pat_s, pat_v, p_s, p_v)$ that accepts source/view patterns and paths. The source/view patterns (pat_s, and pat_v) are used to do a pattern match on the current source and view, to decompose them into smaller subparts, and the source/view paths (p_s, and p_v) are used to extract the source and view keys for matching; the paths are also used to generate a replace statement that updates the source key with the view key, guaranteeing that the updated source still matches with the view. The translated core expression first gives a case analysis on the current source using *pat* and view using pat', then executes the core statement normalized form surface language. 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; 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.

The translation denotes a partial function from high-level BIFLUX to core BIFLUX. For example, `INSERT` 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.

4 BiGUL

The core language presented in Section 3 still retains XML-specific details from the high-level BIFLUX 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 [15], in which the data representation is XML-free and the data-flow management is done in a point-free style. One of the design

$$
\begin{aligned}
&[\![\texttt{UPDATE}\ pat\ \texttt{IN}\ p\ \texttt{BY}\ vs\ \texttt{FOR VIEW}\ pat'\ \texttt{IN}\ p'\ \texttt{MATCHING SOURCE BY}\ p_s\ \texttt{VIEW BY}\ p_v]\!]^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) \\
&\quad = p[[b]p'] \\
&\qquad \textbf{where}\ ((s_{SV}, ms_V, ms_S), p_s{}', p_v{}') = (splitVStmt(vs), \texttt{case self of}\ pat \to p_s, \texttt{case self of}\ pat' \to p_v) \\
&b = \texttt{alignkey}\ (\texttt{case self of}\ pat \to e_S)\ p'_s\ p'_v\ [\![s_{SV}]\!]^b_{MStmt}(pat, pat', p_s, p_v)\ [\![ms_V]\!]^c_{MStmt}(pat')\ [\![ms_S]\!]^r_{MStmt}(pat) \\
&\quad splitVStmt : VStmt \to (Maybe\ Stmt, Maybe\ Stmt, Maybe\ Stmt) \\
&\quad splitVStmt\ (\texttt{MATCH} \to s) = (Just\ s, Nothing, Nothing) \\
&\quad splitVStmt\ (\texttt{UNMATCHV} \to s) = (Nothing, Just\ s, Nothing) \\
&\quad splitVStmt\ (\texttt{UNMATCHS} \to s) = (Nothing, Nothing, Just\ s) \\
&\quad splitVStmt\ (\texttt{MATCH} \to s\ \texttt{'|'}\ vs) = (Just\ s, ms_V, ms_S) \\
&\qquad \textbf{where}\ splitVStmt\ (vs) = (Nothing, ms_V, ms_S) \\
&\quad splitVStmt\ (\texttt{UNMATCHV} \to s\ \texttt{'|'}\ vs) = (ms_{SV}, Just\ s, ms_S) \\
&\qquad \textbf{where}\ splitVStmt\ (vs) = (ms_{SV}, Nothing, ms_S) \\
&\quad splitVStmt\ (\texttt{UNMATCHS} \to s\ \texttt{'|'}\ vs) = (ms_{SV}, ms_V, Just\ s) \\
&\qquad \textbf{where}\ splitVStmt\ (vs) = (ms_{SV}, ms_V, Nothing)
\end{aligned}
$$

Fig. 8: BiFluX `UPDATE FOR VIEW` statement normalization.

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

Fig. 9: Syntax of BiGUL.

goals of BiGUL is to serve as the underlying engine for BiFluX —we will compile the core language into BiGUL in Section 5. The advantage of using BiGUL is that it is completely formally verified in the dependently typed language Agda [19] to guarantee that any program written in BiGUL satisfies the BX properties. We have ported it into Haskell as an embedded domain-specific language, and the compilation rules in Section 5 can freely use Haskell's language features to produce BiGUL programs.

The syntax of BiGUL is shown in Figure 9, which originates from BiFluX's core language, and thus many operations —e.g., `Replace`, `Fail`, and `Skip`— resemble those presented in Section 3. 1. In this section, we will mainly focus on the operations that are different from the core language.

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.

The `Align` operation in BiGUL 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: BiGUL'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. BiGUL does not include conditionals like `ifS` and `ifV` in the core, since they are subsumed by `CaseS` and `CaseV`.

5 Core Compilation

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 7 is compiled into the BiGUL program in Figure 10. The compilation (Section 5. 2) basically consists of five parts: translating the core bidirectionalizable updates to BiGUL operations (Sections 5. 2. 3, 5. 2. 4, and 5. 2. 5), source paths into BiGUL update patterns (Section 5. 2. 1), view expressions into BiGUL's view rearrangement operation (Section 5. 2. 2), general expressions into Haskell expressions (Section 5. 3), 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 in this paper 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 2— and capture that 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 7) 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 5. 1.

5. 1 XML values and regular expression types

As several other XML processing languages [12] [4] [5], we consider a type system of regular expression types with structural subtyping[†2]:

†2 We use ‖ for syntax alternatives in the type grammar to prevent confusion.

```
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)))))
```

Fig. 10: Compiled BiGUL program of the address example.

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 $\alpha \in$ *Atom* are primitive booleans, strings or labeled sequences $n[\tau]$. *Sequence types* $\tau \in$ *Type* are defined using regular expressions, including empty sequence $()$, alternative choice $\tau \mid \tau'$, sequential composition τ, τ', iteration τ^* 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:

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 τ_D are sequences with no top-level variables (to avoid non-label-guarded recursion [5]). 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 [12] [5], 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. In this paper, 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 [17]:

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

$$\frac{}{\Gamma \vdash_{sp} x \Rightarrow \lambda upat.genupat(\Gamma, x, upat)} \qquad \frac{}{\{\tau\} \vdash_{sp} \texttt{self} \Rightarrow id} \qquad \frac{\tau : n\ [\tau_1]}{\{\tau\} \vdash_{sp} \texttt{child} \Rightarrow \texttt{UIn}}$$

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

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

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

$$\frac{\{\tau_1\} \vdash^{\texttt{iter}}_{sp} :: nt \Rightarrow f_1 \quad \{\tau_2\} \vdash^{\texttt{iter}}_{sp} :: nt \Rightarrow f_2}{\{(\tau_1, \tau_2)\} \vdash^{\texttt{iter}}_{sp} :: nt \Rightarrow f_1 \times f_2} \qquad \frac{\{\alpha\} \vdash_{sp} p \Rightarrow f}{\{\alpha\} \vdash^{\texttt{iter}}_{sp} p \Rightarrow f} \qquad \frac{\{E(X)\} \vdash^{\texttt{iter}}_{sp} p \Rightarrow f}{\{X\} \vdash^{\texttt{iter}}_{sp} p \Rightarrow f}$$

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

Fig. 11: Compilation of source path.

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 $t \in Tree$ can be $\texttt{true}, \texttt{false} \in Bool$, strings $w \in \Sigma^*$ (for some alphabet Σ), or singleton trees $n[v]$ with a node label n. *Forest values* $v \in Val$ 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 τ denotes a set of values $[\![\tau]\!]$ that is defined as the minimal solution (formally the least fixed point [12]) 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 τ_1 is said to be a *subtype* of τ_2, written $\tau_1 <: \tau_2$, if the flat values belonging to τ_1 are also values of τ_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 τ_1 into a supertype τ_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 τ_2. This problem has been considered in [17], 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* [9], which is a bit like a lens from τ_2 to τ_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.

5.2 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.

5. 2. 1 Source Paths

The $p[b]$ operation in Section 3. 2. 3 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 11. Γ 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 Γ 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 Γ. As Γ 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 Γ follows the alphabetical order of the variables in Γ 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[`replace`] in which the path is a variable x. Then the generated function is $\lambda upat.$(`UProd` *upat* (`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 5. 1, so we define rules that enumerate all the cases. When the result type is τ^* and the path p_2 is a $::nt$, an *id* function is returned if τ 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 7, there is a source path $\$source \,/$ `child` $/ :: person$, which is compiled into $id \circ$ `UIn` $\circ$ id, i.e. `UIn`.

5. 2. 2 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

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

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

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

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

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

Fig. 12: Compilation of view expression.

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 (Γ_p) that maps each path to a fresh Haskell variable name, which will be used for view expression compilation. Figure 12 gives the compilation rules for constructing a Haskell expression (*hexp*) from a view expression under the environment Γ_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 Γ_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 (λ(*Niibook hEmployeelst*) $\rightarrow$ *hEmployeelst*), as shown in the third line of Figure 10.

A related operation is `view` $x := e$ `in` b, which is compiled into a combination of two operations in BiGUL, as shown in Figure 13: 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.

5. 2. 3 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 13, 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. The compiled BiGUL fragment can be found in Figure 10.

5. 2. 4 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 14), 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

Fig. 13: Compilation of bidirectional updates.

boolean expression e, and both of them are translated into case statements in BiGUL (`CaseS` and `CaseV` respectively).

5. 2. 5 Source-view alignment

As described toward the end of Section 4, the core alignment operations `alignpos` and `alignkey` correspond closely to BiGUL's `Align` operation.

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

$$\frac{\vdash \tau \Rightarrow (\_, \emptyset) \quad \vdash x \texttt{ as } \tau \Rightarrow (hx, \{(x, hx)\})}{\vdash () \Rightarrow ((), \emptyset) \quad \vdash n[pat] \Rightarrow (hn\ hpat, \Gamma_{pat})} \quad \frac{\vdash pat_1 \Rightarrow (hpat_1, \Gamma_{pat_1}) \quad \vdash pat_2 \Rightarrow (hpat_2, \Gamma_{pat_2})}{\vdash pat_1, pat_2 \Rightarrow ((hpat_1, hpat_2), \Gamma_{pat_1} \cup \Gamma_{pat_2})}$$

Fig. 14: Pattern Compilation.

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`.

5.3 Compilation of expressions, paths and patterns

Finally, Figure 15 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 Γ_{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 Γ_{var} (if any) and then adding (x, hx). Like source paths in Section 5.2.1, 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.

6 Related Work

6.1 XML update languages

Several XML update languages have been proposed, including (among many others) XQuery! [10], Flux [4] and the standard W3C XQuery Update Facility [24]. 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.

6.2 XML view updating

In [7], 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 [16] supports updatable views of XML data by giving a bidirectional semantics to the XQuery Core language. The semantic bidirectionalization technique of [18] interprets various XQuery use cases as BXs by encoding them as polymorphic Haskell functions. The Multifocal language [20] 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.

6.3 Bidirectional XML languages

Many bidirectional programming languages support tree-structured or XML data formats. Two popular bidirectional XML languages are XSugar [3] and biXid [14], 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 [8] [21] [22] [13], and follow a combinatorial style that puts special emphasis on building complex lenses by composition of smaller combinators. Depend-

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

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

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

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

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

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

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

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

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

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

$$\frac{x \in dom(\Gamma_{var})}{\Gamma_{var} \vdash_p x \Rightarrow const\ \Gamma_{var}(x)} \qquad \frac{}{\Gamma_{var} \vdash_p \texttt{self} \Rightarrow id} \qquad \frac{}{\Gamma_{var} \vdash_p \texttt{child} \Rightarrow out} \qquad \frac{\alpha <: nt}{\Gamma_{var} \vdash_p :: nt \Rightarrow id}$$

$$\frac{\alpha \not<: nt}{\Gamma_{var} \vdash_p :: nt \Rightarrow const\ ()} \qquad \frac{e : ()}{\Gamma_{var} \vdash_p \texttt{where}\ e \Rightarrow const\ ()} \qquad \frac{\Gamma_{var} \vdash \texttt{if}\ e\ \texttt{then self else}\ () \Rightarrow f}{\Gamma_{var} \vdash_p \texttt{where}\ e \Rightarrow f}$$

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

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

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

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

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

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

Fig. 15: Compilation of Expression and Path.

ing on the choice of combinators, lens languages can become very powerful at specifying application-specific behavior [22] [1] [21]. 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.

7 Conclusion

In this paper, we propose a novel *bidirectional programming by update* paradigm that comes to light from the idea of extending a traditional update language with bidirectional features. Under the new paradigm, programmers write bidirectional updates that specify how to update a source document by embedding view information. To demonstrate the potential of this paradigm, we designed BiFluX, a high-level bidirectional XML update language. 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. For example, during alignment, the programmer might wish to specify the exact source positions into which unmatched views are inserted, but currently there is no way to do that with BiFluX; this requires us to go back to the basic BX theory and then come up with a new syntax for such specifications. 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.

Acknowledgements

We would like to thank the anonymous reviewers for carefully reviewing the draft and giving so many useful comments, and our lab members for intensive discussion.

References

[1] Barbosa, D. M. J., Cretin, J., Foster, J. N., Greenberg, M., and Pierce, B. C.: Matching lenses: alignment and view update, International Conference on Functional Programming, ACM, 2010, pp. 193–204.

[2] Benzaken, V., Castagna, G., and Frisch, A.: CDuce: an XML-centric general-purpose language, International Conference on Functional Programming, ACM, 2003, pp. 51–63.

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

[4] Cheney, J.: FLUX: functional updates for XML, International Conference on Functional Programming, ACM, 2008, pp. 3–14.

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

[6] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J.: Bidirectional transformations: A cross-discipline perspective, International Conference on Model Transformation, Springer, vol. 5563 of *LNCS*, 2009, pp. 260–283.

[7] Fegaras, L.: Propagating updates through XML views using lineage tracing, International Conference on Data Engineering, IEEE, 2010, pp. 309–320.

[8] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM TOPLAS 29*, 3 (2007), 17.

[9] Foster, J. N., Pilkiewicz, A., and Pierce, B. C.: Quotient lenses,International Conference on Functional Programming, ACM, 2008, pp. 383–396.

[10] Ghelli, G., Ré, C., and Siméon, J.: XQuery!: An XML query language with side effects,International Conference on Extending Database Technology, Springer, vol. 4254 of *LNCS.*, 2006, pp. 178–191.

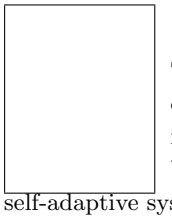
[11] Hosoya, H., and Pierce, B.: Regular Expression Pattern Matching for XML, ,ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2001, pp. 67–80.

[12] Hosoya, H., Vouillon, J., and Pierce, B. C.: Regular expression types for XML, International Conference on Functional Programming, ACM, 2000, pp. 11–22.

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

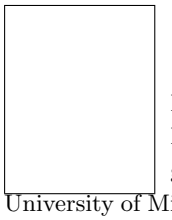
[14] Kawanaka, S., and Hosoya, H.: biXid: a bidirectional transformation language for XML, International Conference on Functional Programming,

ACM, 2006, pp. 201–214.
[15] Ko, H.-S., Zan, T., and Hu, Z.: BiGUL: A formally verified core language for putback-based bidirectional programming, ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, ACM, 2016, pp. 61–72.
[16] Liu, D., Hu, Z., and Takeichi, M.: An expressive bidirectional transformation language for XQuery view update, *Progress in Informatics* (2013), pp. 89–130.
[17] Lu, K. Z. M., and Sulzmann, M.: An implementation of subtyping among regular expression types, Asian Symposium on Programming Languages and Systems, Springer, vol. 3302 of *LNCS*, 2004, pp. 57–73.
[18] Matsuda, K., and Wang, M.: Bidirectionalization for Free with Runtime Recording: Or, a Lightweight Approach to the View-update Problem, International Symposium on Principles and Practice of Declarative Programming, ACM, 2013, pp. 297–308.
[19] Norell, U.: Dependently Typed Programming in Agda, *Advanced Functional Programming*, P. Koopman, R. Plasmeijer, and D. Swierstra, Eds., vol. 5832 of *LNCS*. Springer, 2009, pp. 230–266.
[20] Pacheco, H., and Cunha, A.: Multifocal: A strategic bidirectional transformation language for XML schemas, International Conference on Model Transformation, Springer, vol. 7307 of *LNCS*, 2012, pp. 89–104.
[21] Pacheco, H., Cunha, A., and Hu, Z.: Delta lenses over inductive types, Bidirectional Transformations, vol. 49 of *Electronic Comms. of the EASST*.
[22] Pacheco, H., Hu, Z., and Fischer, S.: Monadic combinators for "putback" style bidirectional programming, ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, ACM, 2014, pp. 39–50.
[23] Pacheco, H., Zan, T., and Hu, Z.: BiFluX: A Bidirectional Functional Update Language for XML, International Symposium on Principles and Practice of Declarative Programming, ACM, 2014, pp. 147–158.
[24] Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., and Siméon, J.: Xquery update facility 1.0, W3C Recommendation, `http://www.w3.org/TR/xquery-update-10/`, March 2011.
[25] Vansummeren, S.: Type inference for unique pattern matching, *ACM TOPLAS 28*, 3 (2006), ACM, pp .389–428.
[26] Zhu, Z., Ko, H.-S., Martins, P., Saraiva, J., and Hu, Z.: BiYacc: Roll your parser and reflective printer into one, *Bidirectional Transformations* (2015), 43.



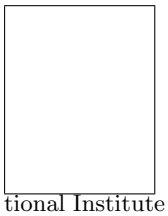
ザン　涛

Tao Zan currently is a Ph.D. student at SOKENDAI. His research interest is programming language, bidirectional transformation, and self-adaptive system.



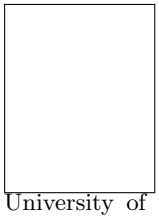
Hugo PACHECO

Hugo Pacheco is a Postdoctoral Research Assistant of INESC TEC, Portugal. He obtained a PhD degree in Computer Science from the University of Minho in 2012. Since then, he has led research at the National Institute of Informatics, Japan and at Cornell University, US. His research interests include programming languages, model transformations and formal methods.



柯　向　上

Hsiang-Shang Ko received his DPhil degree from the University of Oxford in 2014, and now works as a postdoctoral researcher at the National Institute of Informatics, Japan. His research interests include dependently typed programming, datatype-generic programming, and bidirectional programming.



胡　振　江

Zhenjiang Hu received his B.S. and M.S. degrees from Shanghai Jiao Tong University in 1988 and 1991, respectively, and Ph.D. degree from University of Tokyo in 1996. He was a lecturer (1997-2000) and an associate professor (2000-2008) in University of Tokyo, before joining National Institute of Informatics (NII) and the Graduate School for Advanced Studies (SOKENDAI) as a full professor from 2008. His main research interest is in programming languages and software engineering in general, and functional programming, parallel programming, and bidirectional model-driven software development in particular. He is a member of JSSST, IPSJ, ACM and IEEE.