BiFluX: A Bidirectional Functional Update Language for XML

Hugo Pacheco
Cornell University, USA
hpacheco@cs.cornell.edu

Tao Zan
The Graduate University for Advanced Studies, Japan
zantao@nii.ac.jp

Zhenjiang Hu
National Institute of Informatics, Japan
hu@nii.ac.jp

Abstract
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, with a clear and well-behaved bidirectional semantics and a decidable static type system based on regular expression types.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; H.2.3 [Database Management]: Language—Data manipulation languages

Keywords XML, bidirectional programming, update languages

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 [11] 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 [11], including functional programming, software engineering and database. These languages come in different flavors, including many focused on the transformation of tree-structured data with a particular application to XML documents [7][12][13][18][21][22][25], and can be classified into three main paradigms. The first relational paradigm [7][21] prescribes writing a declarative (non-deterministic) consistency relation between two formats, from which a suitable BX is automatically derived. The second bidirectionalization paradigm [12][22][24] asks users to write a transformation in a traditional unidirectional language, that plays the role of a functional consistency relation. The last combinatorial paradigm [14][18][25] 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 of 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 combinational languages, that have the theoretical potential to fully specify the behavior of a BX [13][27], 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 [25].

Intuitively, the goal of a BX is to translate updates on a target model into updates on a source model (and vice-versa) so that the updated models are kept consistent. As SQL stands for relational databases, a few high-level XML update languages [8][15][30] facilitate common modification operations over XML documents. Contrarily to XML transformation languages, XML update languages are well-suited for specifying small in-place changes to in a concise way, leaving all the remaining parts of a document unchanged.

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 model to embed information from a target model, and the system derives a query from source to target that evinces the consistency between both models. Such a bidirectional update allows to express the relationship between source and target models in a simple way —as in the
We define the high-level syntax of BiLUX with bidirectional features to write, directly and at a nice level of bidirectional update, short description of paper our path expressions, and is desugared in BiLUX and tested with many examples including those in this paper. 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 BiLUX, a type-safe, declarative and expressive language for the bidirectional updating of XML documents that is deeply inspired by FLUX [8], a simple and well-designed functional XML update language. We lift unidirectional FLUX updates to bidirectional BiLUX 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 BiLUX programs to ensure that they build well-behaved BXs. We demonstrate the usefulness of BiLUX by illustrating typical examples of BXs written as bidirectional update programs. These help clarifying the stylistic differences to traditional bidirectional programming approaches, and substantiate that bidirectional update languages can combine a declarative language notation with a programming approaches and the stepwise style of combinatorial approaches. Another significant difference to FLUX is that updates on sources carry an additional notion of view, what becomes syntactically evident with a new non-in-place UPDATE FOR VIEW operation that synchronizes a source sequence with a view sequence. Such synchronization can be configured by the programmer via a matching condition that aligns source and view nodes, and a triple of matching/unmatching clauses that describe the actions for individual source-view nodes. When two source and view nodes MATCH, a bidirectional statement is executed to update the source using the view; an unmatched view node (UNMATCHV) creates a new node in the source, either as a default or according to a unidirectional CREATE statement that provides a fresh source node to be normally updated with the existing view node; an unmatched source node (UNMATCHS) is DELETED by default, but we may KEEP it by providing a unidirectional statement describing how to invalidate the given WHERE SOURCE selection criteria. While UPDATE FOR VIEW statements are intrinsically bidirectional, the same BiLUX syntax (e.g., DELETE) may be overloaded and denote either a bidirectional or a unidirectional update depending on the context. The rule is that all BiLUX statements are bidirectional, except inside UNMATCHS or UNMATCHV clauses. An example that puts all these features to use is illustrated later in Figure 3.

2. A Bidirectional Update Language

Before giving concrete examples in Section 3, we start with a brief explanation of the features of BiLUX and its informal semantics to show the big picture of our general framework.

2.1 BiLUX syntax

We define the high-level syntax of BiLUX in Figure 1 as a modest syntactic extension to FLUX; the new features are highlighted in green. Similarly to FLUX [8], BiLUX enjoys a clear semantics and syntactic typechecking by translating programs into a canonical core language. Our examples assume an informal familiarity with commonplace XML technologies like XQuery expressions, XPath paths and XDUce-style regular expression types.

To have a taste of BiLUX, imagine that we want to update the last author of a particular book with title ‘Querying XML’ in a database of books with type

```plaintext
books [book [title [string], author [string] ] ]
```

using a view of type `author [string]`. We can accomplish this by writing an update (with source `$source` and view `$view`)

```plaintext
UPDATE $source/books/book BY {
   REPLACE author(last()) WITH $view
} WHERE SOURCE title = "Querying XML"
```

FLUX-like syntax At first glance, bidirectional BiLUX programs look just like regular FLUX programs. We omit the syntactic definitions of expressions Expr, paths Path, and patterns Pat. Variables Var are written $x, $y, etc. Statements Stmt include conditionals, composition, let-binding, case expressions or updates, which may be guarded by a WHERE clause that defines a set of conditions. Statements can be empty {}, or parenthesized using braces { Stmt}. As in FLUX, in-place updates_upd can be singular, to update single trees, or plural, to update the children of each selected tree. Single insertions (INSERT BEFORE/AFTER) insert a value before or after each node selected by a path, while plural insertions (INSERT AS FIRST/LAST INTO) insert a value at the first or last position of the child-list of each selected node. Singular deletions (DELETE FROM) delete each selected node, while plural deletions (DELETE FROM) delete their content. Single replacements (REPLACE WITH) replace each node selected by a path, while plural replacements (REPLACE IN) replace their content. Single updates UPDATE BY apply a statement to each tree in the result of a path.

Source and view matching The main difference in BiLUX is that updates on sources carry an additional notion of view, what becomes syntactically evident with a new non-in-place UPDATE FOR VIEW operation that synchronizes a source sequence with a view sequence. Such synchronization can be configured by the programmer via a matching condition that aligns source and view nodes, and a triple of matching/unmatching clauses that describe the actions for individual source-view nodes. When two source and view nodes MATCH, a bidirectional statement is executed to update the source using the view; an unmatched view node (UNMATCHV) creates a new node in the source, either as a default or according to a unidirectional CREATE statement that provides a fresh source node to be normally updated with the existing view node; an unmatched source node (UNMATCHS) is DELETED by default, but we may KEEP it by providing a unidirectional statement describing how to invalidate the given WHERE SOURCE selection criteria. While UPDATE FOR VIEW statements are intrinsically bidirectional, the same BiLUX syntax (e.g., DELETE) may be overloaded and denote either a bidirectional or a unidirectional update depending on the context. This is the rule that all BiLUX statements are bidirectional, except inside UNMATCHS or UNMATCHV clauses. An example that puts all these features to use is illustrated later in Figure 3.

Pattern matching Another significant difference to FLUX is the support for pattern matching. This is a very useful feature of XML transformation languages like XDUce [12] or CDUce [13], that allow matching tree patterns against the input data to transform it into an output of different shape. Typical XML update languages like XQuery [15] or FLUX [8] 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. In BiLUX, pattern matching can be used to guide the update based on the structure of the data (via LET and CASE statements), and is mostly useful for non-in-place updates that match source and view formats of different shapes.

Source/view/normal expressions Our language considers three kinds of source, view or normal variables. Expressions in IF, LET, CASE or WHERE clauses support additional tags to disambiguate if they refer to only the SOURCE, to only the VIEW, or to the global environment of an update. These tags can be ignored at first glance and will in fact be omitted in our examples, as they are inferable from context information for each update. Update procedures are omitted but can be easily added to the language.
3. Examples

This section illustrates how writing a bidirectional update feels like to a programmer, through a series of BX examples using BiFLuX:

Example 3.1 demonstrates how the new bidirectional notation, in combination with ordinary unidirectional FLUX updates, can be used to intuitively describe a bidirectional update;

Example 3.2 illustrates nested bidirectional updates and flexible non-local matching behavior;

Example 3.3 showcases pattern matching and recursive procedures.

3.1 Institutional address book example

Consider a typical address book represented by the DTD from Figure 4. An address book contains a list of persons, each possessing a name, a list of emails and an optional telephone number. Let us start with the following XML address book with three persons:

```
<addrbook>
  <person><name>Hugo Pacheco</name>
  <email>hpacheco@nii.ac.jp</email>
  <email>hpacheco@gmail.com</email>
  </person>
  <person><name>John Doe</name>
  <email>doe@domain.com</email>
  <email>john.doe@gmail.com</email>
  <person><name>Zhenjiang Hu</name>
  <email>zh@nii.ac.jp</email>
  <email>zh@nii.ac.jp</email>
  </person>
</addrbook>
```

On the other hand, the NII’s administrative services may keep only a view with the name and institutional address of employees (people with an email at "nii.ac.jp"), as shown in the DTD from Figure 5.
Tao (in alphabetical order) as a new NII employee, fix Zhenjiang's short description of paper

The names $elem$ and $v:elem$ are BiFlux type variables that refer to the types of source and view elements declared in the respective DTDs.

3.2 Social address book example

In response to the Web 2.0 movement, consider that our address book now supports groups of people according to their social relationships. Consider our address book with people now classified into groups:

```xml
<addrbook><group><name>friends</name></group>
  <person><name>Hugo Pacheco</name><email>hpacheco@nii.ac.jp</email></person>
  <person><name>John Doe</name><email>doe@domain.com</email></person>
  <person><name>Tao Zan</name><email>zantao@nii.ac.jp</email></person>
  <person><name>Zhenjiang Hu</name><email>hu@nii.ac.jp</email></person>
</addrbook>
```

The update function is more interesting. For instance, if we add Tao (in alphabetical order) as a new NII employee, fix Zhenjiang's email and delete Hugo, we get an updated source where John is left unchanged, Tao is created with a default telephone number (as his name does not match any name in the original source), Zhenjiang's NII email is updated and the NII email of Hugo is deleted:

```xml
<addrbook><person><name>Hugo Pacheco</name><email>hpacheco@nii.ac.jp</email></person>
  <person><name>John Doe</name><email>doe@domain.com</email></person>
  <person><name>Tao Zan</name><email>zantao@nii.ac.jp</email></person>
  <person><name>Zhenjiang Hu</name><email>hu@nii.ac.jp</email></person>
</addrbook>
```

We can easily write a bidirectional update in BiFlux to synchronize these two formats, as illustrated in Figure 3. The root procedure niibook takes as arguments the root source and view variables. It focuses on a sequence of source persons by traversing down the path $source/addrbook/person$, selecting only those that have NII emails, and focuses on a sequence of view employees by traversing up the (injective) path $view/niibook/employee$. Elements in the two sequences are matched by their names. For matching person-employee pairs, the person’s first NII email is updated with the employee’s email. If a new unmatched employee exists in the view, a new person with a default telephone is created in the source (inheriting its view name and email). If an old unmatched person exists in the source, all its NII emails are deleted.

The unique query for this example simply keeps the first institutional email of each person working at the NII:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE niibook [ <!ELEMENT niibook (employee*)> <!ELEMENT employee (name,email)> ]>

DECLARE $source AS s:addrbook, VIEW $view AS v:niibook =
PROCEDURE niibook($source AS s:addrbook, VIEW $view AS v:niibook) =
  UPDATE $source/addrbook/person BY {
    MATCH -> REPLACE email[ends-with(text(),'nii.ac.jp')][1] WITH $email' 
    UNMATCHV -> CREATE VALUE <person> $name AS v:name, $email' AS v:email IN $view/niibook/employee: 
    MATCHING BY name WHERE email[ends-with(text(),'nii.ac.jp')] 
  } FOR VIEW employee:$name AS v:name, $email' AS v:email IN $view/niibook/employee:
```

Figure 3: BiFlux update for the institutional address book example.

This time, a social media application may only be interested in the groups and names of people, according to a view schema:

```xml
socialbook[group[name[string]. person[name[string]]]]
```

A bidirectional update that synchronizes address and social books is written in Figure 4. It starts by matching groups, proceeding recursively for persons within groups (with default behavior for unmatched groups). Inside, for source-view persons matching on their name, no update is necessary. For each unmatched view person, we attempt to retrieve its address book information from any other group in the original source, or otherwise create a default person.

The corresponding query function produces a view with the same structure but showing only names of groups and persons. For the update function, imagine that we modify the view by reordering the two groups, changing Hugo’s group and creating a new group for family members:

```xml
<addrbook><group><name>friends</name></group>
  <person><name>Hugo Pacheco</name><email>hpacheco@nii.ac.jp</email></person>
  <person><name>John Doe</name><email>doe@domain.com</email></person>
</addrbook>
```

The correspondingly updated source is as follows:

```xml
<addrbook><group><name>coworkers</name></group>
  <person><name>Hugo Pacheco</name><email>hpacheco@nii.ac.jp</email></person>
  <person><name>John Doe</name><email>doe@domain.com</email></person>
</addrbook>
```

Since Hugo changed from group coworkers to friends, he is considered an unmatched view person under his new group. Our bidirectional update avoids his original address details to be lost, by looking them up in all groups, instead of only in Hugo’s original group (what would be the default behavior). An analogous example motivates an extension to the alignment-aware language of [1].

3.3 Bookmark example

For a different bidirectional updating example, consider the conversion between two popular browser bookmark formats studied in [2]. Netscape stores its bookmarks in an HTML format (Figure 7), while...
PROCEDURE socialbook(SOURCE $source AS s:addrbook, VIEW $view v:socialbook) =
  UPDATE $source/addressbook/group BY { MATCH -> UNMATCHV -> LET $old = $source/addressbook/group/person IN
  LIST $oldperson = $old/name/text() = $person/name/text() || [1] IN
  IF $oldperson THEN CREATE VALUE $oldperson ELSE ()
} FOR VIEW $person' IN $persons' MATCHING BY name
FOR VIEW group[$name AS v:name, $persons AS v:person*] IN $view/socialbook/group MATCHING BY name
PROCEDURE top(SOURCE $html AS s:html, VIEW $xbel v:xbel) =
  UPDATE html/[(head(String), body($) WITH s:h1, dl($) AS (s:dt|s:dd)*)] IN $html BY
  { REPLACE IN $h1 WITH $t ; contents($nc,$xc) }
FOR VIEW xbel/title[$t AS String], $xc AS (v:bookmark|v:folder)*] IN $xbel
PROCEDURE contents(SOURCE $nc AS (s:dt|s:dd)*, VIEW $xc AS (v:bookmark|v:folder)*] = UPDATE $nc BY { CASE $v OF {
  bookmark[href[$url AS String], title[$title AS String]] -> REPLACE . WITH <dt><a><href>{$url}</href>{$title}</a></dt>
  for folder[$title AS String], $xc AS (v:bookmark|v:folder)*] -> REPLACE IN $h3 WITH $title ; contents($dl*/,$fxc) 
} } FOR VIEW $v IN $xc

Figure 6: BiFLUX update for the social address book example.
Figure 7: A Netscape bookmark format DTD.

Figure 8: BiFLUX update for the bookmark example.

Figure 9: A XBEL bookmark format DTD.

We use ## for syntax alternatives in the type grammar to prevent confusion.

4.1 XML values and regular expression types

As several other XML processing languages [6,9,17], we consider a type system of regular expression types with structural subtyping.

Atomic types $\alpha := \text{bool} \mid \text{string} \mid n[\tau]$

Sequence types $\tau := \alpha \mid (\tau \mid \tau' \mid \tau^* \mid X)$

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

Type definitions $\tau_D := \alpha \mid (\tau_D \mid \tau_D \cdot \tau_D \cdot \tau_D^*)$

Type signatures $E := ::= E \cdot \tau \mid X = \tau_D$

Type definitions $\tau_D$ are sequences with no top-level variables (to avoid non-label-guarded recursion [9]). 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 [9,17], 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 [22].

\[ t ::= \text{true} \mid \text{false} \mid w \mid n[v] \]

\[ v ::= t \mid () \mid L	ext{v} \mid R	ext{v} \mid (v, v) \mid [v_0, \ldots, v_n] \]
Atomic values \( t \in \text{Tree} \) can be \text{true}, \text{false} \in \text{Bool}, strings \( w \in \Sigma^* \) (for some alphabet \( \Sigma \)), or singleton trees \( n[v] \) with a node label \( n \). Forest values \( v \in \text{Val} \) include the empty sequence (), left-\( L \), or right-\( R \) tagged choices, binary sequences \( (v, v') \) and lists of arbitrary length \( [v_0, \ldots, v_n] \). The semantics of a type \( \tau \) denotes a set of values \( [\tau] \) that is defined as the minimal solution (formally the least fixed point) of the following set of equations:

\[
\begin{align*}
\text{bool} & \triangleq \{ \text{true}, \text{false} \}, \\
\text{string} & \triangleq \Sigma^*, \\
[\tau, \tau'] & \triangleq \{ (v, v') \mid v \in [\tau], v' \in [\tau'] \} \cup \emptyset, \\
\tau & \triangleq \{ (L v) \mid v \in [\tau] \} \cup \{ (R v) \mid v \in [\tau] \}, \\
[\tau] & \triangleq \{ (v_0, \ldots, v_n) \mid v_0, \ldots, v_n \in [\tau], n \geq 0 \}.
\end{align*}
\]

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

- Flat atomic values: \( ft ::= \text{true} | \text{false} | w | n[ft] \)
- Flat forest values: \( fv ::= () | ft, fv \)

and introduce a function \( \text{flat}: \text{Val} \to \text{FVal} \) that ignores markup.

We denote the usual flat semantic of a type \( \tau \) as \( [\tau]_{\text{flat}} \).

4.2 Core expression, path and pattern language

In BiFlUX, updates instrumentally use XQuery expressions, XPath paths and XDuce patterns to manipulate XML data. This subsection succinctly describes their syntax in our core language, but is not essential for our design and may be skipped on a first reading.

We write expressions \( e \) in a minimal XQuery-like language that is a variant of the \( \mu \text{X} \) core language proposed in [9]:

\[
e ::= () | e', n[e] | \text{let } pat = e \text{ in } e' | p | e \equiv e' | \text{if } e \text{ then } e' \text{ else } e'' | \text{for } x \text{ in } e \text{ return } e''
\]

Despite expressions can be used in updates rather indiscriminately, in BiFlUX only a particular subset of the expression language is suitable for denoting foci. Therefore, we differentiate paths \( p \) in a core path language that represents a minimal dialect of XPath:

\[
p ::= \text{self} | \text{child} | \text{child} \cdot \text{child} | \text{child} | \text{child} \cdot \text{child} | \text{child} | \text{child} \cdot \text{child} | \text{child}
\]

\[nt ::= n | \text{true} | \text{false} | F(e)
\]

To simplify the formal treatment, we consider nodetests \( ::= nt \) that apply to atomic values and where clauses \( ::= e \) that filter values satisfying an expression \( e \). We write the syntactic sugar \( p \equiv nt \triangleq p / nt \) and \( p[e] \triangleq p / e \) where \( e \). XPath-like traversals can be defined as: \( ::= \text{self} ::= \text{node}() \), \( p / n \triangleq p \cdot n ::= \text{child} ::= a \) and \( p / n = p / \text{node}() / \text{child} ::= a \).

In contrast to \( \mu \text{X} \), our core expression language supports pattern expressions in let bindings: \( \text{let } pat = e \text{ in } e' \) matches a pattern \( pat \) against the result of an expression \( e \) and then executes an expression \( e' \) that may refer to the variables bound by \( pat \). We consider a language of XDuce-style patterns [17]:

\[
pat ::= x \text{ as } \tau \mid () | n[pat] | pat'
\]

Note that 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. Less severe linearity restrictions are actually known [16], but these simple patterns suffice for our practical needs. Also worth noting is that we require every variable to be annotated with a type. This simplifies our design, but will in turn increase the number of (often unnecessary) annotations in our bidirectional update programs. We see it as an orthogonal problem that can be mitigated using existing tree-based type inference algorithms [12].

4.3 Tripartite environments

A type environment \( \Gamma \) consists of a set of bindings \( x : \tau \) of variables to types. An environment is a function \( \gamma: \text{Var} \to \text{Val} \) from variables to values. As usual, we assume variable names in an environment to be distinct. We write \( \Gamma(x) \) and \( \gamma(x) \) for the type and value of a variable; \( \Gamma[x : \tau] \) and \( \gamma(x) := v \) add a new variable to an environment. We say that \( \gamma \) has type \( \Gamma \), written \( \gamma: \Gamma \), if \( \gamma(x) : \Gamma(x) \) for all \( x \in \text{dom}(\Gamma) \).

Fresh environments are denoted by the empty set \( \emptyset \).

Since our language is bidirectional, we will consider three kinds of variables (and environments): source variables, that are accessible from the current source; view variables, that are accessible from the current view; and normal variables, that are accessible from the global environment of the update and independent from the source or the view. We will talk about source/view/normal paths or expressions, that may only refer to source variables, view variables or any variable, respectively; non-source expressions may refer to view and normal variables simultaneously.

To describe source or view environments, we introduce record types \( \nu \) that denote sequences of variable-annotated types:

\[
\nu ::= x : \tau, \nu | ()
\]

As an abuse of notation, we see a record type \( \nu \) as an ordinary type (by forgetting variable names) or as a type environment; we cast a conforming value into an environment \( \gamma: \nu \).

4.4 Core update language

Unlike conventional XML update languages, our core update language comprises two kinds of update statements: unidirectional FLUX updates, interpreted as arrows [20] that modify a document in-place, changing its schema; and bidirectional BiFlUX updates, interpreted as Bxs [27] that update a source document given a view document or query a source document to compute its view fragment, for fixed source and view schemas. Our core unidirectional updates \( u \) are adapted from the core FLUX update language [8]:

\[
u ::= \text{skip} | u; u' | \text{insert } e | \text{delete}
\]

These include standard operations such as the no-op \( \text{skip} \), sequential composition, conditionals or case expressions. The basic operations are \( \text{insert } e \), that inserts a value given an empty sequence as focus; and \( \text{delete} \), that 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).

Core bidirectional updates \( b \) are denoted by the grammar:

\[
b ::= \text{skip} | \text{fail} | b_1; b_2 | \text{view } x ::= e \text{ in } b
\]

Here, \( p \) is the name of a BiFlUX procedure. A procedure is defined as a declaration \( P(\vec{x} : \vec{\tau}) : \nu \leftarrow \nu_0 \triangleq \vec{s} \), meaning that \( P \) takes a vector of parameters \( \vec{x} \) of types \( \vec{\tau} \) and builds a BX between a source of type \( \nu_0 \) and a view of type \( \nu \). Accordingly, a procedure call \( P(\vec{p}, \vec{e}, \vec{c}) \) takes three kinds of arguments: source paths \( \vec{p} \); non-source expressions \( \vec{e} \); and normal expressions \( \vec{c} \). We collect procedure declarations into a set \( \Delta \), that we will assume to be fixed. Procedures may also be recursive.

The bidirectional operation \text{skip} keeps the source unchanged for an empty view; for a non-empty view, we must \text{fail} to update the source (as \text{UPDATEQUERY} precludes that all view information must
be used to update the source). Composition $b_1 \cdot b_2$ updates the source with $b_1$, and then runs $b_2$ over the updated source. The statement $\text{view } u := e \text{ in } b$ models a view dependency, by stating that a view variable $x$ can be computed using the view expression $e$ (written $\text{Be} := e$ in the high-level BiFlUX language), and then runs $b$ using the remaining view. Bidirectional statements may also change the current source or view, by focusing down a source path and updating the resultant source $(p[b])$, or by evaluating a non-source expression “backwards” from the view and updating the source with the resultant view $(b)e$. The basic replace operation embeds the view into the source, while $\text{iter } b$ embeds the same view into each tree in a source forest. As before, we consider three kinds of bidirectional conditionals and case expressions, whose expressions or paths are respectively source, view/non-source or normal variables.

The two special alignment statements update a source sequence using a view sequence. They receive a filtering source expression $e_s$, and match source elements satisfying $e_s$ with view elements by position (alignpos) or by keys (alignkey), defined by two source ($p_s$) and view paths ($p_v$). Then, a bidirectional statement $b$ processes matched source-view elements, a create statement $c$ instructs how to create a suitable source to match with an unmatched view, and a recover statement $r$ denotes how to process an unmatched source. Create statements $c$ are simply optional unidirectional updates $mu$.

A recover statement $r$ is a unidirectional update of the form:

$$r ::= \text{if } e \text{ then } r' \text{ else } r'' \mid \text{ delete } \text{ keep } u \mid \text{ case } e \text{ of } \text{ pat } \rightarrow r$$

It supports conditionals and case expressions like regular updates, and two primitive operations: delete, to delete an unmatched source; and keep, to keep an unmatched source modified with $u$ so that it does not satisfy the above $e_s$ filtering expression.

### 4.5 Core bidirectional lens language

To interpret our core language bidirectionally, we translate core bidirectional updates into a core language of “putback”-style lenses over generalized tree structures [27], supporting regular expression types. For the context of this paper, a lens is a BX $l : \tau_s \leftrightarrow \tau_v$ between a source type $\tau_s$ and a view type $\tau_v$, under an environment of type $\Gamma$, defined using the combinators from Figure 10. The concrete syntax for lenses is not essential in our design, and can be skipped unless for understanding the bidirectional update semantics discussed in Section 5. The intuition for each combinator (read as a transformation from view to source) should be understandable from its type signature, and more details regarding their concrete bidirectional semantics can be found in [27].

Each lens in the above language comprises two partial functions $U : \Gamma \rightarrow \text{Maybe } \tau_s \rightarrow \tau_v \rightarrow \tau_s$ and $Q : \tau_s \rightarrow \tau_v \rightarrow \tau_v$, satisfying laws similar to $\text{UPDATEQUERY}$ and $\text{QUERYUPDATE}$. Since these functions are partial, updating or querying may fail at runtime. This is sometimes inevitable, for instance, whenever a view value does not satisfy a view condition or a view dependency written in a WHERE VIEW clause. Remark that the update function $U$ receives an additional environment of type $\Gamma$ and an optional source of type $\text{Maybe } \tau_s$ (as in Haskell, but with short-hand notation $\text{Nothing } =$ . and Just $v = v$ for values), to account for cases (like UNMATCH clauses) when a new source must be reconstructed from the view without updating an existing source [52]. For a lens polymorphic over its environment type, we often write just $l : \tau_s \leftrightarrow \tau_v$.

---

**4.6 XML subtyping and ambiguity**

The notion of subtyping plays a crucial role in XML approaches with regular expression types. A type $\tau_1$ is said to be a subtype of $\tau_2$, written $\tau_1 <: \tau_2$, if the flat values belonging to $\tau_1$ are also values of $\tau_2$, i.e., $[\tau_1]_{\text{flat}} \subseteq [\tau_2]_{\text{flat}}$. Under a flat value representation, a value of a type naturally belongs simultaneously to all its supertypes. This motivates a notion of structural equality between types: two types $\tau_1$ and $\tau_2$ are equivalent ($\tau_1 \equiv \tau_2$), if they satisfy the same set of flat values, if both $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_1$, e.g., $\tau := \tau \equiv \tau$. It also induces an equivalence relation $\sim$ that ignores structure and relates values parsing the same data using different markup, formally, $\sim : \text{lift}(v) \equiv \text{flat}(v'), \text{ e.g., } \text{L}v \sim Rv$.

A type $\tau$ is said to be unambiguous if the equivalence relation for structured values of that type ($\sim_\tau$) is the equality relation ($\equiv_\tau$), intuitively meaning that there is only one way to parse a flat value of type $\tau$ into a structured value of type $\tau$. Unambiguous regular expression types have a direct correspondence to algebraic data types [29], and standard automata algorithms exist for deciding unambiguity of regular expression types in polynomial time [10][31]. Since we retain a structured representation of values, upcasting a value $v_1$ of type $\tau_1$ into a supertype $\tau_2$ requires more than a proof of subtyping: we must also change $v_1$ into a value $v_2$ that contains the same flat information as $v_1$ but conforms to the structure of $\tau_2$. This problem has been considered in [23], 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, $\exists \gamma : c \Rightarrow \tau_1$, is called a canonizer [14], which is a bit like a lens from $\tau_2$ to $\tau_1$ that comprises a total upcast function $\text{ucast}: \tau_1 \rightarrow \tau_2$, and a partial downcast function $\text{decast}: \tau_2 \rightarrow \tau_1$. In our sense, canonizers satisfy two properties stating that they only handle structure: $\text{Up}\sim \text{ucast } v_1 \sim v_1 \downarrow \text{Down}\sim \text{decast } v_2 = v_1 \Rightarrow v_1 \sim v_2$.

For an unambiguous type $\tau_2$, we can lift a canonizer $c : \tau_1 \leftrightarrow \tau_2$ into a well-behaved lens lift $c : \tau_1 \Rightarrow \tau_2$. Similarly, for an unambiguous type $\tau_1$, the inverse $c' : \tau_2 \leftrightarrow \tau_1$—even though not a canonizer itself due to partiality—can be lifted into a well-behaved lens lift $c' \downarrow : \tau_2 \Rightarrow \tau_1$.

---

**5. Type System and Staged Semantics**

In traditional XML-based languages with regular expression types [8][9][17], the loose separation between types and values lends itself to a Curry-style interpretation: terms in the language are given semantics regardless of typing, and typing rules assess the well-formedness of terms. In BiFlUX, however, types come prior to semantics, as an update program describes a BX between two known source and view types on which its meaning may naturally depend. This leads to a Church-style interpretation, where semantics is given to type derivations instead of independent terms. Also, as we have seen before, updates in our core language are given semantics in two stages: they are first interpreted as lens expressions (typed between given types), and lenses have themselves a bidirectional semantics.

---

**5.1 Interpreting expressions, paths and patterns**

The type system and semantics for expressions and paths is similar to that of µXQ [9] and FLUX [8], adapted to our context. We merge typing and semantics into a single judgment $\Gamma ; \mu x := e : \tau \Rightarrow \lambda \gamma. v$, meaning “in type environment $\Gamma$ with optional root variable $\mu x$, expression $e$ has type $\tau$, and given environment $\gamma$ it evaluates to value $v$”. Type soundness is guaranteed by construction, such that the argument environment $\gamma$ is of type $\Gamma$ and the produced $v$ is of type $\tau$. The concrete rules can be found in our technical report [28]. We also define a judgment $\Gamma ; \tau := e : \tau' \Rightarrow \lambda \gamma. v. v'$, meaning “in type environment $\Gamma$ with root type $\tau$, expression $e$ has type $\tau'$, and...
Figure 10: Language of point-free lenses for translating core bidirectional updates.

given environment $\gamma : \Gamma$ and root value $v : \tau$ it evaluates to value $v' : \tau'$. The root type and underlying value are added to the (type and value) environments under a fresh root variable:

$$\Gamma[x : \tau; x \vdash e : \tau'] = \lambda \nu. \gamma[x := v]. v' \notin \text{dom}(\Gamma)$$

In BiFlux, we follow a simple approach to pattern matching. We first define pattern type inference as a judgment $\vdash \text{pat} : \tau \Rightarrow \Pi$, that reads “pattern pat has type $\tau$ and yields a lens environment $\Pi$”. This is straightforward since we require all pattern variables to be annotated with a type; many others, including [16][32], have studied more advanced XML-based pattern inference techniques. A lens environment $\Pi$, is a set of bindings $x := (l, \tau')$ of variables to lenses from a source type $\tau$, such that $l : \tau \Rightarrow \tau'$. As an abuse of notation, we see $\Pi_r$ as a type environment; we cast a source value $v$ of type $\tau$ into an environment of source variables $\gamma_{\nu, \Pi_r} = \{x := Q_l(v) | x := (l, \tau) \in \Pi_r\}$. The second step for matching a pattern against an input value is to simply check for subtyping between the input type and the inferred pattern type [16], as the witness functions of our subtyping algorithm already give us a way to convert values in both directions. For ambiguous patterns that may match an input value in multiple ways, such functions are responsible for resolving the ambiguity. Different matching policies can be modelled by adapting the subtyping proof system, as noted in [23], despite not being essential for our approach.

5.2 Interpreting unidirectional updates

The typechecking and operational semantics of core unidirectional updates mimic those of the core FLUX language, and can be found in the companion technical report [28]. The judgment $\Gamma \vdash \{\tau\} \cup \{\tau'\} \Rightarrow \lambda \gamma. \nu. v'. v$ means that “in type environment $\Gamma$, a unidirectional update maps values of type $\tau$ to values of type $\tau'$, and given environment $\gamma$ and input value $v$ (of type $\tau$) it produces the updated value $v'$ (of type $\tau'$).

5.3 Interpreting bidirectional updates

Contrarily to unidirectional updates, that modify values from an input to an output type, bidirectional updates are evaluated against two given source and view types and return a lens between those types. The judgment $\Gamma; \Pi_r \vdash \{\tau\} \cup \{\nu\} \Rightarrow l$ indicates that “in type environment $\Gamma$ and lens environment $\Pi_r$, a statement $b$ produces a lens $l$ between source type $\tau$ and view type $\nu$ under type environment $\Gamma$”. The type environment $\Gamma$ denotes normal variables (that are in scope of the update function of the generated lens); the lens environment $\Pi_r$ denotes source variables from the source $\tau$; and the view type $\nu$ defines a view environment (also referred to as $\nu$) that denotes view variables from the view $\nu$. The dichotomy between our representations of source and view environments is justified by the need to keep a special account of view information: declared view variables must be used at least once in the update, while source variables might not be used at all to update the source. Most rules for typechecking and evaluating bidirectional updates as lenses are shown in Figure 11 (The listifyS and listifyV functions and the judgment $\vdash \tau_1 \subseteq \tau_2 \Rightarrow l$ will be introduced in Section 5.4).

Source/view expressions are interpreted under only a lens/view environment: “in lens environment $\Pi_r$ (and source type $\tau$), a source expression $e$ has type $\tau'$, and given a value $v : \tau$ produces a value $v' : \tau''$; “in view environment $\nu$ (and view type $\nu$), a view expression $e$ has type $\tau'$, and given a value $v : \nu$ produces a value $v' : \nu'$”. Source/view expressions in BiFlux are only evaluated in the context of a bidirectional update. Basic combinators The $\textsf{skip}$ combinator returns the lens that does not update the source if the view is the empty sequence, while $\textsf{fail}$ yields the bottom lens that always fails to update the source; $\textsf{replace}$ completely replaces the source with a view that is “smaller”, such that the view type is a subtype of the source type.

Composition Bidirectional composition $b_1 ; b_2$ first splits the view into two parts $v_1$ and $v_2$, evaluating $b_1$ with view $v_1$ followed by $b_2$ with view $v_2$ over the same source. The auxiliary function $\textsf{split}(v, \theta_1, \theta_2) = (v_1, v_2, l_2)$ splits $v$ such that $\textsf{dom}(v_1) = \textsf{dom}(v) \cap \theta_1$ and $\textsf{dom}(v_2) = \textsf{dom}(v) \cap \theta_2$, returning a lens $l_2 : l_1 v_2 \Rightarrow v$. Parallel lens composition (unfork $l_2$) is different from sequential lens composition ($l_1 ; l_2$); for it to be well-behaved, the second update will not affect the result of the first query, i.e., $Q_l(U_2(v, v_2)) \subseteq Q_l(v)$. This property is currently checked dynamically by unfork, that fails to update the source otherwise. We believe that this check could be done at static time by combining recent work on XML query-update independence [2][4][6].

Changing source focus The bidirectional update $p[b]$ changes the source focus by traversing down the source path $p$, and then evaluates $b$ with a fresh lens environment. Note that it is intrinsically different from the unidirectional update $p[u]$, as the source type does not change and the source data is not updated in-place, but modified to embed the view data. Precisely, this requires interpreting the source path as a lens via a judgment $\Gamma_r \vdash S \{\tau\} p \{\tau'\} \Rightarrow l$, that reads “in lens environment $\Pi_r$, path $p$ changes the source focus from type $\tau$ to type $\tau'$, and produces a lens $l : \tau \Rightarrow \tau'”$. Like in FLUX, arbitrary paths can not be used to change the source focus (see [28]). For example, only the $\textsf{self}$ and $\textsf{child}$ axes (and no absolute paths) 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. The $\textsf{iter}$ $b$ operator shifts the source focus to all tree values in a source forest, and runs $b$ for each tree. Its corresponding BX will duplicate the view value for each source tree during updates, and enforce all selected source trees to be the same during queries.
Changing view focus The operation |b| e| changes the view focus according to the non-expression source e, and then evaluates b for the intermediate view. The judgment \( \Gamma \vdash \{ e \} \ e \nu \Rightarrow l \) indicates that “in type environment \( \Gamma \) expression e changes the view focus from type \( \nu \) to type \( \tau \), and produces a lens l : \( \tau \Rightarrow \nu \)”. Note that, to be successfully interpreted as a “backward” lens, the expression e may only add information to the view, since all view information must be used to update the source; this implies that only a restricted subset of our path language that can be statically checked to be injective can be used to change the view focus. The operation view \( x := e \in b \) removes a view variable x from the current view environment without loss of information (with the view expression e evidencing how x can be computed from the other view variables), followed by running b.

Conditionals The conditional operations ifS, ifV and ifS choose between two statements \( b_1 \) or \( b_2 \) according to a boolean expression e, and differ slightly on the bidirectional nature of the underlying lenses (see (28)), given that e is a source, view or normal expression. The same rationale is applied to case expressions (28).

Source-view alignment The alignment operations alignpos and alignkey synchronize source and view forests. They 1) uniformize the source and view types into lists using listifyS and listifyV, 2) align source and view elements using matching algorithms on lists by position or by keys —calculated as paths on the current source/view—and 3) run the statement b for matching source/view pairs, the create statement c for unmatched source elements or the recover statement r for unmatched source elements. Create statements are interpreted unidirectionally using a judgment \( \Gamma \vdash \{ c \} \ c \tau_2 \Rightarrow \lambda \gamma \ v_2 \ v_1 \) saying that “in type environment \( \Gamma \), a create statement c between source type \( \tau_2 \) and view type \( \tau_2 \) returns an optional function (denoted by \( \lambda \gamma \ v_2 \ v_1 \cdot \cdot \cdot )\) that given an environment \( \gamma \) and a view value \( v_2 \) creates a source value \( v_1 \)”. In reverse direction, the judgment \( \Gamma \vdash \{ r \} \ r \tau \Rightarrow \lambda \gamma \ v_1 \ v_1 \) states that “in type environment \( \Gamma \), a recover statement r for a source type \( \tau_1 \) returns a function that given an environment \( \gamma \) and a source value \( v_1 \) returns an optional recovered source value \( v_1 \cdot \cdot \cdot \)”. Note that our alignment operations receive a source predicate e denoting which source values have a correspondence to view values; the underlying lenses enforce that newly created source values must satisfy e, while recovered source values do not originate from the view and must not satisfy e.
As the reader may guess, we could compute via a judgment \( \Pi(x) \rightarrow \tau \) that ~\( \nu \rightarrow \nu \rightarrow l \), and the new view environment is computed from the non-source expressions \( \epsilon \) via a judgment \( \Gamma \rightarrow \nu \rightarrow \tau \rightarrow \nu \rightarrow l \). The semantics of a BiFLUX program is given by converting a set \( \Delta \) of procedure declarations \( P(\xi: \tau): \nu \rightarrow \nu \rightarrow l \rightarrow s \) into a set \( \Delta_{\nu} \) of lenses \( P = l: \nu \rightarrow (\xi_1, \xi_2, \ldots, \xi_n) \nu \rightarrow l \), according to the following derivation:

\[
(\xi_1: \tau_1, \ldots, \xi_n: \tau_n) \rightarrow \nu \rightarrow \nu: \nu \rightarrow \nu \rightarrow l \rightarrow s \rightarrow P = \text{withEnv } f l
\]

4.5 Type normalization

The semantics of our core language relies instrumentally on subtyping to match source and view types. The simplest example is the replace bidirectional update, that requires the view type \( \tau \) to be “smaller” than the source type \( \tau \). (Figure 1), according to a judgment \( \forall \rightarrow \tau \rightarrow \tau \rightarrow l \) that returns as evidence a lens \( \gamma: l \rightarrow \tau \). As the reader might guess, we could compute \( \forall \rightarrow \tau \rightarrow \tau \rightarrow l \) by checking for subtyping between \( \tau \) and \( \tau \). \( \forall \rightarrow \tau \rightarrow \tau \rightarrow l \) and lifting the resulting canonizer into a lens. But, before doing so, we must verify that: the view type \( \tau \) is unambiguous, a requirement to lift \( c \) into a lens \( \gamma \) and the source type \( \tau \) is ambiguous, since the uncast function of the canonizer does not consider the original source and a naive implementation of an update would potentially discard source information projected away by the lens.

As an example, recapitulate the source database of books used in Section 3 and imagine how we could evaluate a simple update:

\[
\text{REPLACE } $source/books/book/title WITH $view using a list \text{string}^* \text{ as the view type. Consistently with the unidirectional semantics for paths } [28], \text{ the evaluation of the source path as a lens traverses down the source tree and keeps only titles, destroying element labels and replacing all authors for the empty sequence, and modifies the source focus to the intermediate type } (\text{title}[,\text{string}],(),())^* \text{. Then replace checks for subtyping between the view type and the intermediate type, producing a mediating canonizer. Since the intermediate type is ambiguous, the upcasting function of the canonizer would translate a view sequence } [\text{title}[,\text{mybook}]] \text{ into an intermediate value } [\text{title}[,\text{mybook}]],()\text{, that would in turn lead to an updated source books }[[\text{title}[,\text{mybook}]],(atom1,())] \text{. This is clearly unsatisfactory as it would discard the entire book database except the first author atom1 of the first book!}

6. BiFLUX to Core Update Normalization

In this section, we formalize the translation from the high-level BiFLUX language to the core language presented in Section 4, highlighting the significant gap between them. This process is usually referred to as normalization in languages like XQuery and FLUX. We define two main normalization functions that interpret statements as bidirectional \([-\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\right

short description of paper

10 2014/08/11
The translation denotes a partial function from high-level BiFluX to core BiFluX. For example, \texttt{INSERT} is not supported for bidirectional updates, \texttt{UPDATE FOR VIEW} is not supported for unidirectional updates, and \texttt{CREATE} or \texttt{KEEP} are only supported under \texttt{UNMATCH} or \texttt{UNMATCHES} clauses, respectively. We assume that paths and expressions are expressed in terms of our core languages; this is standard practice as normalization of XQuery expressions or XPath paths can be done independently. To simplify the presentation, we also assume explicit \texttt{SOURCE} and \texttt{VIEW} tags, though our implementation is elaborated to implicitly distinguish between source/view/normal expressions, using the additional environment information available at the time of typechecking the core language.

### Related Work

#### XML update languages

Several XML update languages have been proposed, including (among many others) XQuery! [13], FLUX [8] and the standard W3C XQuery Update Facility [30]. 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. This poses different (BX-related) challenges on how to deal with non-in-place updates (like \texttt{UPDATE FOR VIEW} statements that may change the cardinality of a sequence instead of updating each element), and therefore how to modify the remaining parts of the source document (e.g., by changing branching decisions) to accommodate the new data so that the updated source fits into the same type.

#### XML view updating

In [12], 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 [22] supports updatable views of XML data by giving a bidirectional semantics to the XQuery Core language. The semantic bidirectionalization technique of [24] interprets various XQuery use cases as BXs by encoding them as polymorphic Haskell functions. The Multifocal language [25] 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 he can not configure. In BiFluX, he writes an update translation strategy directly as an update (over the source) and the system derives the uniquely related query.

#### XML bidirectional languages

Many bidirectional programming languages support tree-structured or XML data formats. Two popular XML bidirectional languages are XSugar [2] and biXid [21], that describe XML-to-ASCII and XML-to-XMl mappings as pairs of intertwined grammars. While XSugar restricts itself to bijective grammars, biXid considers nondeterministic specifications and BXs are inherently ambiguous. Most functional bidirectional programming languages are based on lenses [13][18][26][27], and follow a combinatorial style that puts special emphasis on building complex lenses by composition of smaller combinators. Depending on the choice of combinators, lens languages can become very powerful at specifying application-specific behavior [1][26][27]. 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, that 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 (variables, procedures) and the core lens language that gives it semantics (canonical “point-free” combinators). In [19], we have proposed a simple treeless functional language for writing total \texttt{put} (or update) functions, such that existence of a well-behaved lens can be checked statically, and corresponding total \texttt{get} (or query) functions can be derived automatically. The most significant and innovative difference in BiFluX is again the declarative surface language used to specify BXs as bidirectional update programs, at a notably higher-level of abstraction than native \texttt{put} functions.

Quotient lenses [14] propose loosening lenses modulo equivalence relations, for easing the processing of ad-hoc data formats. To enable compositional reasoning, most interesting quotient lens combinators (concatenation, union, iteration) require the equivalence relations to be decomposable. At first glance, we could lift our core lens language into quotient lenses to allow seamless composition with (subtyping) canonizers. However, since our notion of equivalence is not decomposable for ambiguous types, this would not overcome the need for our type normalization procedures.

### Conclusions

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 to reflect additional view information. We substantiate with examples that this enjoys a better tradeoff 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 bundles all the pieces to build a BX.

To demonstrate the potential of this paradigm, we designed BiFluX, a type-safe high-level bidirectional XML update language. We have fully implemented BiFluX in Haskell and the code, together with additional examples, is available from the project’s website. Our tool translates source and view DTDs into Haskell type declarations using the HaXml package and interprets bidirectional updates as bidirectional lens transformations between the given schemas in a robust manner; the implementation of the core-to-lens translation is strongly-typed, serving as a proof of soundness that helped us catching early programming errors at compile-time; and the bidirectional semantics is completely guided by an underlying lens language, whose correctness has been shown separately in [27]. To support a flexible design, with arbitrary conditionals, case statements and expressions, the statically generated lenses may undergo runtime checks (for particular source and view documents) to ensure that the underlying update and query functions are well-behaved.

As future work, we 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 foci. We also plan to improve the efficiency of our prototype for large XML databases by exploring optimizations to the underlying lens language, including incremental update translation.

To empirically study the practical impact of BiFluX, we are currently undergoing a larger model-based code testing use case.

---

*http://www.prg.nii.ac.jp/projects/BiFluX*

*http://hackage.haskell.org/package/HaXml*
Acknowledgments

This research was partially supported by the NSF under grant SHF-1253165 and by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009.

References


