An expressive bidirectional transformation language for XQuery view update

Dongxi LIU1, Zhenjiang HU2 and Masato TAKEICHI3
1CSIRO ICT Centre
2National Institute of Informatics
3Department of Mathematical Informatics, University of Tokyo

ABSTRACT

This paper presents an expressive bidirectional XML transformation language and uses it to address the problem of updating XML data through materialized XQuery views. The transformations of this bidirectional language can be executed in two directions: in the forward direction, they generate materialized views from XML source, while in the backward direction, they update the source by reflecting back the updates on views. When XQuery is interpreted with this bidirectional language, it can query XML in its forward execution, and update XML source after its backward execution.

We propose the extended round-tripping property for characterizing the good behavior of bidirectional transformations. This property is more flexible for an expressive bidirectional transformation language. The difficulties of updating view insertions are analyzed with detailed examples, and the type information is novelly used to guide backward transformation when views include insertions. A type system with recursive regular expression types for XML is designed for this bidirectional language. Well-typed programs preserve the source type after backward executions. A prototype of our approach is implemented and tested on a number of XQuery use cases.

KEYWORDS
Bidirectional programming language bidirectional transformation view update XQuery type system

1 Introduction

XQuery [1] is a powerful functional language designed to query XML data. The role of XQuery to XML is just like that of SQL to relational databases. However, XQuery still lacks an important feature that SQL has. This feature is view update [2]–[4], which means that updates on a view can be reflected back to the underlying relational database that makes up this view. In other words, XQuery can generate views from XML source data, but it cannot propagate view updates back into the source data.

This paper presents a translational semantics for XQuery with a bidirectional transformation language. In this bidirectional language, every program can be executed in two directions: in the forward direction, it produces a materialized view from the source data; while in the backward direction, it updates the source data by reflecting back the updates on the view. By this way, every XQuery expression can be executed in two directions, and the backward execution will put the updates on views back into the source data.

There have been a number of bidirectional languages designed recently such as [5]–[12]. These languages show that the method originally proposed in [5] is powerful to design bidirectional languages applicable to many domains.
However, there is no a bidirectional language that is systematically designed and used to interpret a language as expressive as XQuery. In [13], we designed a bidirectional language that is expressive enough to interpret XQuery by defining the bidirectional semantics of variable binding and primitive XML processing combinators. However, the language in [13] can only deal with modifications and deletions on views.

In this paper, we present a more expressive bidirectional language extended from [13] to interpret XQuery. This language is designed to support three kinds of updates to XQuery views: modification, insertion, and deletion. The insertions on views are more tricky to transform backward than modifications and deletions. This is because inserted values do not have counterparts in the original source data. Hence, it is difficult to determine the structure of the updated source data without the information derived from the original source data for where and how to put back inserted values. In this work, we address this problem by exploiting the types of the sources and views to provide guiding information for putting inserted values back in a reasonable way.

The transformations in our bidirectional language exploit types to deal with insertions. Since these transformations are expected to be generated by automatic translation of XQuery expressions, they are always longer and harder to understand than high-level XQuery expressions. It is not desirable to ask programmers to annotate types manually. To increase the usability of our language, we design a type system with recursive regular expression types for this bidirectional language. Given a transformation of this language and the type of source data, the type system can check whether this transformation is well-typed, and if yes, it generates the corresponding view type and annotates this transformation with appropriate type information.

Moreover, our type system annotates accurate type information to the transformation, for example, by considering the choice of path in a conditional transformation. A view insertion may not be put back successfully if the annotated type is not accurate enough. This type system is sound with respect to the forward semantics of the language, that is, a well-typed program does not get stuck in its forward execution and generates the view with the correct view type. The backward executions of well-typed programs may fail even if the updated views have correct view types since views may include conflicting or improper updates, which cannot be detected statically by this type system. For successful backward executions, well-typed transformations preserve the types of their source.

For an expressive bidirectional language, it is hard to define the property for characterizing the good behavior of bidirectional transformations. The existing bidirectional languages [5]–[9], [12] mainly adopt the round-tripping property, which is proposed in [2], [3] for relational view update. The round-tripping property says if the source is updated with respect to an updated view, then executing the same query on the updated source should get a view identical to the updated view. However, the round-tripping property is not suitable for our bidirectional language. Actually, this property is also too limited to be accepted in practice by major database management systems, such as Microsoft SQL Server and Oracle DB. This property is violated if a view includes replicas of values (i.e., the dependency in view [14]) and one replica is updated. Value replicas are very common in views generated by joining two tables in a database or two pieces of XML data.

In this paper, we propose the extended round-tripping property for our bidirectional language. This property does not require the updated view and the view generated from the updated source data be identical, or does not relate the two views in an identical relation. Instead, it relates the two views in an update-keeping relation, which requires all updates in the two views be kept or changed in a reasonable way.

The extended round-tripping property is not as limited as the round-tripping property. However, it is still not easy to design our bidirectional language with the extended round-tripping property satisfied. The problem lies in the semantics of the conditional transformation, which has two transformation branches. The conditional transformation must ensure that the same branch is executed to reflect back updates and query the original and updated source, such that the updated view and the new view from the updated source are generated by the same transformation. The extended round-tripping property makes sense and can be proved inductively only when two views are from the same transformation. This same-branch requirement is hard to satisfy because a conditional transformation in our language may refer to variables in its condition and cannot know whether the variables are updated, leading to a changed condition and hence a changed branch selection, by other transformation executed in parallel. In this paper, we refine the data model by tagging values with modification indicators, which are used by the conditional transformation to indicate allowed modifications to the values in its condition, such that other transformations cannot update such values to unexpected ones.

The main technical contributions in this paper are summarized as follows.

- We design a bidirectional language expressive enough to interpret XQuery and able to support in particular view insertions. For this expressive bidirectional language, we propose the extended round-tripping property
for characterizing view updating semantics. The translation from XQuery Core to the bidirectional language is presented.

- We design a type system with recursive regular expression types for this bidirectional language. The types are novelly used to guide the backward execution of transformations to put updates in a reasonable way. This type system is sound with respect to the forward semantics of this language, and ensures well-typed transformations preserve the types of their source data after backward execution.

- We have implemented our approach and applied it to some XQuery use cases from a W3C draft [15]. There are detailed examples designed to discuss and explain the bidirectional semantics of our language, the type system, the language property and the difficulties of processing view insertions.

The remainder of the paper is organized as follows. Section 2 gives an example to illustrate our motivation. Section 3 defines the bidirectional language without considering insertions on views and Section 4 proves the properties of this language. Section 5 interprets XQuery with the bidirectional language. Section 6 presents the type system. Section 7 discusses the insertion problems and revises the bidirectional semantics of the language for supporting insertions according to the annotated types. Section 8 introduces our implementation. Section 9 and 10 give the related work and conclusions of the paper, respectively.

2 A motivating example

We explain the motivation of this work by the XQuery expression in Figure 1, which is an example from W3C XML Query Use Cases [15]. Suppose the file “book.xml” contains the source data in Figure 2. When executing the query in Figure 1, we get the view in Figure 2, which can be regarded as the table-of-contents of the source data.

On this view, users may expect to do some updates, such as modifying titles or attributes, inserting or deleting sections. For example, we change the subsection title of the first section on the view from “Audience” into “Prospective Readers”, and insert a new section after the second section. Obviously, the updated view and the source data currently contain inconsistent information. With bidirectional interpretation of XQuery, this problem can be easily solved. We just need to execute backward the query in Figure 1 and then the updates on the view will be reflected back to the source file. That is, the subsection title of the first section in the file “book.xml” becomes “Prospective Readers” and the second section is followed by a newly inserted section. This example can be found at [16].

3 The bidirectional language

In this section, we define the bidirectional language for interpreting XQuery. The backward semantics of the language in this section does not consider insertions on views (also called target data). In Section 7, we will discuss the problems encountered when views include insertions and revise the language semantics to support insertions.

3.1 The representation of data

Figure 3 gives the syntax of source data and target data, denoted by $S$ or $T$, which is either an empty sequence () or a sequence $V$ of strings or XML elements. To save space, the end tags of XML elements are omitted and their contents are enclosed by brackets. For example, the element `<author>Tom</author>` is represented as `<author>[Tom]`. This form is borrowed from [17]. Two sequences $S_1$ and $S_2$ can be concatenated as a longer sequence, written as $S_1, S_2$, or sometimes $S_1, S_2$ for clarity. We have $(\cdot), S = S$ and $S, (\cdot) = S$.

```
declare function local:toc($book-or-section)
{
  for $section in $book-or-section/section return
  <section>
    {$section/@id, $section/title, local:toc($section)}
  </section>
};
<toc>
  { for $s in doc("book.xml")/book return local:toc($s)}
</toc>
```

Fig. 1 The motivating XQuery expression.
<title>Data on the Web</title>
<author>Serge</author>
<author>Peter</author>
<author>Dan Suciu</author>
<section id="intro" difficulty="easy">
<title>Introduction</title><p>Text ...
</section>
<section>
<title>Audience</title><p>Text ...
</section>
</section>
<section id="syntax" difficulty="medium">
<title>A Syntax For Data</title>
</section>

1) The Source Data

2) The View

Fig. 2 The XML source data.

\[
S, T := () | V \\
V := v, V \\
v := \text{str}(u, o) | \text{tag}(w, o)[S] \\
u := \text{ori}(i) | \text{mod} | \text{ins} | \text{del} \\
w := \text{ori} | \text{ins} | \text{del} \\
o := s | c \\
i := \star | \downarrow | \downarrow | \star
\]

Fig. 3 Syntax of data.

Strings or elements are annotated with a pair \((u, o)\) or \((w, o)\), in which \(u\) and \(w\) indicate their updating states, and \(o\) denotes their origins. The origin annotation \(o\) is either \(s\) for values originating from source data or \(c\) for values originating from code. For example, if a program outputs a hard-coded string, then this string is said to have an origin from code. The updating annotation \(\text{ins}\) (\(\text{del}\)) are used for inserted (deleted) strings or elements. A string can be modified and the annotation \(\text{mod}\) is for modified strings. Strings or elements in their original states are annotated by \(\text{ori}\) (\(i\)) or \(\text{ori}\), respectively. The modification to a string with \(\text{ori}(i)\) is instructed by the modification indicator \(i\). The indicator \(i\) can be \(\star\), \(\downarrow\), \(\downarrow\) or \(\star\), which means respectively that the annotated string can be modified to any other string, to a bigger string, to a smaller string or to the current string (that is, no modification allowed). Strings are compared in alphabetical order. For values with the origin \(c\), their updating annotations can only be \(\text{ori}\) or \(\text{ins}\) for strings, or \(\text{ori}\) or \(\text{ins}\) for elements since such values come from code and hence cannot be changed. However, they still can be inserted on views together with values from source data. This requirement is reflected in the semantics of the bidirectional language.

In this work, after backward executions, the annotation \(\text{del}\) is propagated from values on views back to their
The syntax of our bidirectional language is defined in Figure 4, where Var and fname represent the variable names and function names, respectively. Each language construct there represents a bidirectional transformation between source data and views. The transformations xid and xconst are for identity and constant transformations, respectively. The transformations xchild and xsetcnt are used to deconstruct or construct XML elements. The conditional transformation is represented by xif. The transformation $X_1; X_2$ is to execute $X_1$ and $X_2$ sequentially with the view of $X_1$ as the source data of $X_2$, while the transformation $X_1 || X_2$ executes $X_1$ and $X_2$ independently with their views combined together as the final view. The transformation xmap applies its component transformation $X$ to each item in its source data, corresponding to the map function in functional programming. The constructs xlet and xfunapp provide the mechanism for variable binding and variable reference, just like the let and variable expressions in conventional functional languages. The function applications are represented by xfunapp and functions are declared in G. Other language constructs, such as those to deal with element attributes or name spaces, are included in the language implementation, but omitted in this paper for brevity.

### 3.3 Evaluation environments

This language has forward and backward semantics, so we need two evaluation environments, one for forward semantics, and the other for backward semantics. The environment for forward semantics is denoted by $C$, which maps variables to values; the environment for backward semantics is denoted by $E$, which maps variables to pairs of values. If in the environment $E$ a variable $Var$ is bound to a pair $(S, S')$, then $S$ is the original value of $Var$, and $S'$ is the updated value of $Var$ during backward executions.

An empty environment is represented by a period $\cdot$. We can build new environments by concatenating environments and variable bindings with the comma operator. For example, the new environment $C_1, Var \mapsto S, C_2$ is the result of concatenating the environment $C_1$, the binding $Var \mapsto S$, and the environment $C_2$. For clarity, this new environment is also written as $C_1, Var \mapsto S, C_2$.

For an environment $E$, the notation $E.1$ denotes an environment which maps every variable in $E$ to the first component of the pair bound to this variable by $E$. Formally, $E.1$ is defined as: 1) if $E = \cdot$, then $E.1 = \cdot$; and 2) if $E = E', Var \mapsto (S, S')$, then $E.1 = E'.1, Var \mapsto S$. Similarly, the notation $E.2$ denotes an environment which maps every variable in $E$ to the second component of the pair mapped by $E$ for this variable. $Dom(E) \ (or \ Dom(C))$ means the domain of $E \ (or \ C)$.

The forward and backward semantics of each language construct is defined in the following forms, respectively.

- **The forward semantics**: $\llbracket X \rrbracket_E (S) = T$, meaning that applying $X$ to the source $S$ generates the view $T$ under the environment $E$.
- **The backward semantics**: $\llbracket X \rrbracket_E (S, T') = (S', E')$, meaning that under the environment $E$, applying $X$ to the updated target data $T'$ and the original source data $S$ generates the updated source data $S'$ and a new environment $E'$.

The above two forms are applicable to successful forward and backward executions of $X$. If its executions get stuck, $X$ returns the special value fail.
3.4 Semantics of the language

We will define the forward and backward semantics for each language construct in Figure 4.

**Identity transformation:** This transformation keeps the (updated) source data and the (updated) view identical in both directions. It is just the identity lens in $\mathcal{S}$ except for the evaluation environments.

\[
\begin{align*}
\llbracket \text{id} \rrbracket_{\mathcal{E}}(S) &= S \\
\llbracket \text{id} \rrbracket_{\mathcal{E}}(S, T) &= (T, E)
\end{align*}
\]

**Constant transformation:** This transformation returns its argument $T$, for any source data in the forward execution. $T$, must be $\langle \cdot, \text{str}_{\text{ori}}^{\langle \cdot, \cdot \rangle} \rangle$ or $\langle \text{tag}_{\text{ori}}^{\langle \cdot, \cdot \rangle} \rangle$. Together with other constructs, more complex constant views can be generated. In the backward direction, since the target data $T$, is not allowed to change, this transformation just returns the original source data and evaluation environment. The special value fail may be generated by this transformation or other transformations defined later. One occurrence of fail will cause the whole transformation being executed to terminate immediately with the value fail returned.

\[
\begin{align*}
\llbracket \text{xconst} \rrbracket_{\mathcal{E}}(S) &= I_f, \text{if } T_c \in \langle \cdot, \text{str}_{\text{ori}}^{\langle \cdot, \cdot \rangle} \rangle, \langle \text{tag}_{\text{ori}}^{\langle \cdot, \cdot \rangle} \rangle \rangle \rangle \\
\llbracket \text{xconst} \rrbracket_{\mathcal{E}}(S, T) &= (T, T) \text{, otherwise}
\end{align*}
\]

**Variable reference:** The forward execution of xvar hides the source data $S$ and returns the value of the variable $Var$ as the view. In its backward execution, the source data is not changed, and instead the value of the variable $Var$ in $E$ is updated. In the new environment $E'$, the xvar operator, defined in Figure 5, is used to merge the updates within $S_1$ and $T'$. When merging two replicas of modified strings, the xvar operator allows only one of them to be modified or both of them are modified into the same value. In addition, the change to the modified string must be consistent with the modification indicator tagged on the unmodified string. Otherwise, there is a modification conflict. Note that $S_1$ is the original value of $Var$, which is needed at such cases as defining the backward semantics of sequential composition and merging values that include insertions.

\[
\begin{align*}
\llbracket \text{xvar} \rrbracket_{\mathcal{E}}(S) &= I_f, \text{if } C = C_1, Var \mapsto T, C_2 \text{ and } Var \notin \text{Dom}(C_2) \\
\llbracket \text{xvar} \rrbracket_{\mathcal{E}}(S') &= (S, E'), \text{if } E = E_1, Var \mapsto (S_1, S_2), C_2 \text{ and } Var \notin \text{Dom}(E_2) \\
\llbracket \text{xvar} \rrbracket_{\mathcal{E}}(S, T') &= \text{fail}, \text{otherwise}
\end{align*}
\]

where $E' = E_1, Var \mapsto (S_1, \text{mg}(S_2, T'))$. $E_2$
For view updating of XQuery, it is possible that one source value has several replicas, which may contain different updates. The merging operator \( \text{mg} \) returns a new value that combines all updates within two replicas if there are no conflicting updates. For example, merging elements \(<\text{title}(\text{ori.1},\text{s})>\text{xquery}(\text{ori.1,s})\text{}</\text{title}>\) and \(<\text{title}(\text{ori.1},\text{s})>\text{XQuery}(\text{mod,s})\text{}</\text{title}>\) will generate the element \(<\text{title}(\text{ori.1},\text{s})>\text{XQuery}(\text{mod,s})\text{}</\text{title}>\), while merging \(<\text{price}(\text{ori.1},\text{s})>[30\text{mod,s}]\text{}</\text{price}>\) and \(<\text{price}(\text{ori.1},\text{s})>[25\text{mod,s}]\text{}</\text{price}>\) will cause a conflict and thus terminate the current transformation by generating fail.

**Element deconstruction:** This transformation deconstructs an element and returns its contents in the forward execution. If the source data is not an element, it will fail. In the backward execution, it replaces the contents of the source element with the updated contents.

\[
\text{xchild}_c(S) = \begin{cases} S', & \text{if } S = <\text{tag}(w,o)>[S'] \\ \text{fail}, & \text{otherwise} \end{cases}
\]

\[
\text{xchild}_l(S,T) = \begin{cases} <\text{tag}(w,o)>[T], & \text{if } S = <\text{tag}(w,o)>[S'] \\ \text{fail}, & \text{otherwise} \end{cases}
\]

**Element construction:** The source data of this transformation is also required to be an element. In its forward execution, the contents of the source element are transformed by the component transformation \( X \), and then the generated result will be used as the new contents of the source element. This procedure is reversed in the backward execution. The backward execution of \( X \) generates the updated contents for the source element.

\[
\text{xsetcnt}_c(X)(S) = \begin{cases} <\text{tag}(w,o)>(X(S'))), & \text{if } S = <\text{tag}(w,o)>[S'] \\ \text{fail}, & \text{otherwise} \end{cases}
\]

\[
\text{xsetcnt}_l(X)(S,T) = \begin{cases} <\text{tag}(w,o)>[S'], & \text{if } S = <\text{tag}(w,o)>[S'] \\ \text{fail}, & \text{otherwise} \end{cases}
\]

**Sequential composition:** This transformation takes two component transformations \( X_1 \) and \( X_2 \) and applies them one by one. This definition is the same as that in [5] except that the definition here takes into account the evaluation environments. The backward execution of \( X_2 \) needs to invoke the forward execution of \( X_1 \) under the environment \( E.1 \) to generate the intermediate source data. For this purpose, the variables is bound to their original values by using the operator \( E.1 \).

\[
\begin{align*}
[X_1;X_2]c(S) &= \llbracket X_2 \rrbracket_c(\llbracket X_1 \rrbracket_c(S)) \\
[X_1;X_2]l(S,T) &= \llbracket X_1 \rrbracket_l(S,T'), \text{where } \theta(T', E') = \llbracket X_2 \rrbracket_l(\llbracket X_1 \rrbracket_e(S,T))
\end{align*}
\]

**Parallel composition:** In the forward execution, this transformation applies its component transformations \( X_1 \) and \( X_2 \) independently to the empty sequence \( () \), and concatenates their views as the final view. Though their sources are both \( () \), \( X_1 \) and \( X_2 \) can get nonempty values to transform by referring to variables in the environment. In the backward execution, the updated view is separated into two subviews for \( X_1 \) and \( X_2 \), respectively, according to the lengths of the corresponding original subviews. The lengths of updated subviews and the corresponding ones are the same since insertion is not considered in this section. The operator \( \text{len} \) returns the length of a sequence. With the updated subviews, the backward executions of \( X_2 \) and \( X_1 \) are performed in sequence, with the environment \( E' \) generated by \( X_2 \) fed into \( X_1 \) for producing the final updated environment \( E'. \) Alternatively, we can choose to execute backward \( X_1 \) before \( X_2 \) for this transformation. The source \( S \) is not changed by the backward execution.

\[
\begin{align*}
\llbracket X_1 \rrbracket_c(S) &= \llbracket X_1 \rrbracket_c((()) \llbracket X_2 \rrbracket_c(())) \\
\llbracket X_1 \rrbracket_l(S,T) &= \theta(S,E') \\
\end{align*}
\]

where

\[
\begin{align*}
T &= T_1, T_2, \text{len}(T_i) = \text{len}(\llbracket X_i \rrbracket_{e.1}(())) (i \in \{1, 2\}) \\
(();E') &= \llbracket X_2 \rrbracket_{e.1}((()) \llbracket X_1 \rrbracket_{e.0}(())) \\
((();E') &= \llbracket X_1 \rrbracket_{e.0}(() \llbracket X_1 \rrbracket_{e.1}(()))
\end{align*}
\]

**Mapping transformation:** The \( xmap \) transformation applies its argument transformation \( X \) to each string or element in the source. If the source is \( () \), then the view is also \( () \). In the backward execution, we separate the updated view \( T \) into a list of subsequences by using the split operator, each of which is the updated view for a string or an element
in the original source data. This is done by the split operator defined in Figure 6, which inputs the sequence \( T \) and an integer list \([l_1, \ldots, l_n]\), and divides \( T \) into a list of \( n \) subsequences \( T_i \) (1 \( \leq i \leq n \)), where \( \text{len}(T_i) = l_i \). For example,\( \text{split}(T_1, v_2, v_3, (2, 0, 1)) \) generates three subsequences: \([v_1, v_2, v_3] \), \((v_1, v_3)\). An empty list is represented by [], and the concatenation of an item \( x \) with a list \( ls \) is represented by \( x:ls \). The updated source and environment are generated by the \text{iter} operator, defined in Figure 6, which iterates the backward execution of \( X \) on each source item and its updated view. In Section 7, a new split operator will be defined to separate the updated view with inserted values.

\[
\begin{align*}
\text{split}(\emptyset, []) &= [] \\
\text{split}(T, [], l_0) &= (\text{split}(T, l_0)) \\
\text{split}(T, l_0) &= T, \text{split}(T_2, l_2), \text{if} \ l_0 > 0, T = T_1, T_2, \text{and len}(T_1) = l_0 \\
\text{iter}(X, [], (S', E')) &= (S', E') \\
\text{iter}(X, T, S, v, S', E') &= \text{iter}(X, T, S, S', v', E'), \text{where } [X]_E(v, T) = (v', E')
\end{align*}
\]

**Fig. 6** Two Operators: split and iter.

\[
\begin{align*}
\text{clean}(S, S) &= S \\
\text{clean}(\text{str}('x'\.\cdot\cdot\cdot), S, \text{str}('x'\.\cdot\cdot\cdot)) &= \text{str}('x'\.\cdot\cdot\cdot), \text{if } i \in \{1, \cdot, \cdot\} \\
\text{clean}(\text{str}('x'\.\cdot\cdot\cdot), \text{str}('x'\.\cdot\cdot\cdot)) &= \text{str}('x'\.\cdot\cdot\cdot) \\
\text{clean}(\text{str}('w'\.\cdot\cdot\cdot), \text{str}('u'\.\cdot\cdot\cdot)) &= \text{str}('u'\.\cdot\cdot\cdot), \text{if } u \in \{\text{ori}(\cdot), \text{mod}, \text{del}\} \\
\text{clean}(\text{<tag}(w'\.\cdot\cdot\cdot)>S_1, \text{<tag}(w'\.\cdot\cdot\cdot)>S_2) &= \text{<tag}(w'\.\cdot\cdot\cdot)>[\text{clean}(S_1, S_2)] \\
\text{clean}(v_1, S_1, v_2, S_2) &= \text{clean}(v_1, v_2), \text{clean}(S_1, S_2) \\
\text{clean}(S_1, S_2) &= \text{fail}, \text{if no other case applies}
\end{align*}
\]

**Fig. 7** The clean operator.

**Conditional transformation**: This transformation executes \( X_1 \) if the predicate \( P \) holds, otherwise it executes \( X_2 \). In the backward direction, \( X_1 \) or \( X_2 \) is executed backward to generate the updated source data \( S' \) and the updated evaluation environment \( E' \), which are then used as arguments to execute backward \( P \) before finishing up this conditional transformation. The backward execution of \( P \) is involved to make sure the existing and future updates to \( S' \) and \( E' \) do not affect the validity of \( P \). That is, if \( P \) is true (or false) under the environment \( E \) and the source data \( S \), then \( P \) should still have the same value under \( E' \) and \( S' \) even if \( E' \) and \( S' \) may be updated further by other transformations. This condition is needed when proving the well-behavedness of bidirectional transformations. The predicate \( P \) is defined in the next subsection and more illustrations are given there by examples.

\[
\begin{align*}
\text{\small xif } P \ X_1, X_2 [S] &= \begin{cases} \text{\small clean}(S, X_1), & \text{if } P \text{\small clean}(S) = \text{true} \\
\text{\small clean}(S, X_2), & \text{if } P \text{\small clean}(S) = \text{false}
\end{cases} \\
\text{\small xif } P \ X_1, X_2 [S], T &= \begin{cases} \text{\small clean}(S', X_1), & \text{if } P \text{\small clean}(S) = \text{true and } [X_1]_E(S, T) = (S', E') \\
\text{\small clean}(S', X_2), & \text{if } P \text{\small clean}(S) = \text{false and } [X_2]_E(S, T) = (S', E')
\end{cases}
\end{align*}
\]

In the lens language [5], the conditional lenses \( \text{cond} \) and its two special instances \( \text{ccnd} \) and \( \text{acnd} \) are defined. The lens \( \text{cond} \) transforms backward an updated view with the component lens that generates the original view, while the lenses \( \text{acnd} \) and \( \text{ccnd} \) allow an updated view to be reflected back by the component lens that is not the one generating the original view. For the lenses \( \text{acnd} \) and \( \text{ccnd} \), if the component lens used to process an updated view is not the one generating the original view, the original source data is completely discarded or needs to be fixed.
up for the backward execution. We argue that this is not acceptable for XQuery view update since the source data generally contains more information than views, and thus discarding or fixing-up the source data will lose or destroy the data in the source but not in views. On the other hand, it is not reasonable either by using the original source data directly. This is because two branches of \( xif \) are independent, so a pair of source and target data related by one branch does not imply they can be related sensibly by another branch.

**Variable binding:** This construct provides the primitive variable binding mechanism for this bidirectional language. It will be used to define other constructs that need bound variables, such as function calls, and the \( \text{let} \) and for expressions in XQuery.

\[
\begin{align*}
\text{xlet } \text{Var X} \&\text{t}(S) &= \llbracket \text{X} \rrbracket_{\text{C}}((()) \text{t}), \text{where } \text{C'} = \text{C}, \text{Var } \mapsto S' \\
\text{xlet } \text{Var X} \&\text{t}(S, T) &= \begin{cases} 
\llbracket \text{X} \rrbracket_{\text{C'}}((()) \text{t}), \text{if } \text{Sinlet } \in \text{Dom}(\text{E}) \\
\text{clean}(S, S', \text{E'}), \text{otherwise}
\end{cases}
\end{align*}
\]

where \( ((()), \text{E'}, \text{Var } \mapsto (S, S')) = \llbracket \text{X} \rrbracket_{\text{C'}}((()), \text{T}) \text{ and } \text{E''} = \text{E'}, \text{Var } \mapsto (S, S), \text{Sinlet } \mapsto (1, 1), \text{otherwise} \)

The forward semantics of this construct is defined similarly as that of the let construct in conventional functional programming languages. That is, the source \( S \) is bound to variable \( \text{Var} \) before executing the argument transformation \( X \). In this language, the source can be replicated only after it is bound to a variable.

The backward semantics is defined in two cases, depending on whether a special variable \( \text{Sinlet} \) is bound or not in the execution environment \( \text{E} \). A binding \( \text{Sinlet } \mapsto (1, 1) \) in \( \text{E} \) is used to indicate the current \( \text{xlet} \) is enclosed by another \( \text{xlet} \) (the value (1,1) in the binding does not matter). If the current \( \text{xlet} \) is an enclosed one, its backward execution executes backward the transformation \( X \) under the environment \( \text{E'}, \text{Var } \mapsto (S, S') \), where the variable \( \text{Var} \) is bound to a pair of the original source \( S \), since at this point the value of this variable has not been updated. The backward execution of \( X \) produces the environment \( \text{E''}, \text{Var } \mapsto (S, S'') \), which contains the updated source \( S'' \) and environment \( \text{E''} \).

If the current \( \text{xlet} \) is not enclosed by any other \( \text{xlet} \), the backward execution of \( X \) is under an environment with the binding \( \text{Sinlet } \mapsto (1, 1) \) pushed into \( \text{E} \) after the binding of \( \text{Var } \mapsto (S, S') \). In this case, the updated source \( S' \) needs to be cleaned by using the \( \text{clean} \) operator, which is defined in Figure 7. The purpose of \( \text{clean} \) is to remove changes to modification indicators made by backward executions of some predicates. As to be seen later, the backward executions of the predicates \( \text{xeq}, \text{xlt}, \text{and xgt} \) may change the modification indicator \( j \) in a string to express their requirements on how other transformations executed in parallel should modify the string. The changes to modification indicators should be cleaned if the strings are not modified by users, so that the language can satisfy the stability property described later.

Briefly, the operation \( \text{clean}(S, S') \) changes a string in \( S' \) into the corresponding one in the original source \( S \) in the following cases: a string \( st_{\text{ori}}(i, s) \) \((i \in \{\uparrow, \downarrow, \bullet\})\) in \( S' \) corresponds to a string \( st_{\text{ori}}'(i, s) \) in \( S \), or a string \( st_{\text{ori}}'(i, s) \) in \( S' \) corresponds to a string \( st_{\text{ori}}(i, s) \) \((i \in \{\uparrow, \downarrow\})\) in \( S \). In these cases, the modification indicators are changed by predicates \( \text{xeq}, \text{xlt} \) and \( \text{xgt} \).

**Function call:** Suppose the function \( \text{fname} \) is defined as

\[
\text{fun } \text{fname}(\text{Var}_1, \ldots, \text{Var}_n) = X
\]

Then, the semantics of applying the function \( \text{fname} \) to \( n \) arguments \( X_1, \ldots, X_n \) is defined below with the previous constructs.

\[
\text{xfunapp } \text{fname} ([X_1, \ldots, X_n]) = \text{xconst }(); X'_1
\]

where

\[
\begin{align*}
X'_1 &= X_1; \text{xlet } \text{Var}_1 X'_2 \\
X'_2 &= X_2; \text{xlet } \text{Var}_2 X'_3 \\
& \vdots \\
X'_n &= X_n; \text{xlet } \text{Var}_n X
\end{align*}
\]

In this definition, all component transformations are first evaluated, and then their results are bound to the corresponding variables. And then, the function body \( X \) is executed. The source data for the function body is always the
empty sequence () due to the definition of \texttt{xlet}. That is, the function body cannot directly use and update the source data of \texttt{xfunapp}. Hence, any data to be processed by the function body should be passed as the arguments of the function call.

### 3.5 Predicates

Each predicate is also defined with both forward and backward semantics. The target data produced by predicates is either \texttt{true} or \texttt{false}, which is used only by \texttt{xif} and cannot appear in views. Like transformations, predicates also take two arguments for their backward executions, but the second argument is the updated source data generated by the backward executions of branch transformations of \texttt{xif}, instead of the target data \texttt{true} or \texttt{false}.

\textbf{Comparison:} In the forward direction, the predicate \texttt{xeq} returns \texttt{true} if the views of $X_1$ and $X_2$ are the same string (probably with different annotations), or \texttt{false} otherwise. The comparison result is preserved by the backward semantics of this predicate. That is, if this predicate returns \texttt{true} (or \texttt{false}) with the original source and environment, then it still returns \texttt{true} (or \texttt{false}) with the updated ones. For this purpose, the backward semantics of predicate \texttt{xeq} expresses its modification requirements to the string views of $X_1$ and $X_2$ through the guard operator, defined in Figure 8. The operation $\text{guard}(\text{str}^{(o,i)}, i)$ incorporates the modification indicator $i$ into the string view $\text{str}^{(o,i)}$ if it is modifiable (i.e., if $u \in \{\text{ori}(\star), \text{ori}(\top), \text{ori}(\bot)\}$ and $o = s$). For example, if $i$ is $\top$ and the string view is $\text{str}^{(\text{ori}(\star),s)}$, then the result of applying the guard is an updated string view $\text{str}^{(\text{ori}(\top),s)}$. That is, the string can only be modified to a bigger one by other transformations that may execute in parallel. Preserving the comparison result is critical for the language to satisfy the round-tripping property described later. This is because if the comparison result changes due to updates on view, the forward transformation applied to the original source and updated source are actually different, and hence the updated view and the view from the updated source cannot be sensibly related. For predicates \texttt{xlt} and \texttt{xgt}, their forward semantics is defined similarly, and their backward semantics are the same as that of \texttt{xeq}, hence not given.

\[
[x_{\text{eq}}X_1X_2]_{c}(S) = \begin{cases} 
\text{true}, & \text{if } [X_1]_{c}(()) = \text{str}^{(x_{\text{eq}})} \text{ and } [X_2]_{c}(()) = \text{str}^{(x_{\text{eq}})} \\
\text{false}, & \text{if } [X_1]_{c}(()) = \text{str}^{(x_{\text{eq}})} \text{ and } [X_2]_{c}(()) = \text{str}^{(x_{\text{eq}})} \text{ and } \text{str} \neq \text{str}' \\
\text{fail}, & \text{otherwise} \\
\end{cases}
\]

\[
[x_{\text{eq}}X_1X_2]_{c}(S,S') = \begin{cases} 
(S', E'), & \text{if } [X_1]_{c}(()) = \text{str}^{(x_{\text{eq}})}, [X_2]_{c}(()) = \text{str}^{(x_{\text{eq}})}, \\
& (0, E') = [X_2]_{c}(()), \text{guard}(\text{str}^{(x_{\text{eq}})}, \top)), \text{ and} \\
& (0, E') = [X_1]_{c}(()), \text{guard}(\text{str}^{(x_{\text{eq}})}, \bot)) \\
\end{cases}
\]

\[
[x_{\text{lt}}X_1X_2]_{c}(S) = \begin{cases} 
\text{true}, & \text{if } [X_1]_{c}(()) = \text{str}^{(x_{\text{lt}})} \text{ and } [X_2]_{c}(()) = \text{str}^{(x_{\text{lt}})} \text{ and } \text{str} < \text{str}' \\
\text{false}, & \text{if } [X_1]_{c}(()) = \text{str}^{(x_{\text{lt}})} \text{ and } [X_2]_{c}(()) = \text{str}^{(x_{\text{lt}})} \text{ and } \text{str} \neq \text{str}' \\
\text{fail}, & \text{otherwise} \\
\end{cases}
\]

As an example, suppose we have the source data $10^{(\text{ori}(\star),s)}$, and the transformation \texttt{xlet Sd $X_1||X_2$, where
backward execution is given in Table 3 and Table 4. The data and environment used in these executions are given.

If we change the view into $S^\prime$ and vice versa, this predicate returns the updated source data.

This predicate holds if the source data is an element. Since an element is not allowed to change into $S^\prime$, the backward execution will detect this modification conflict with the modification indicator in $10^{\text{mod}15}$. Still for the same transformation, if the source data is $12^{\text{mod}15}$, then its forward execution generates the view $12^{\text{mod}15}$ with the $\text{xeq}$ predicate returning false. To preserve the comparison result, the $\text{xvar}$ in $\text{xeq}$ needs to change the source into $12^{\text{mod}15}$. Thus, the updated view $15^{\text{mod}15}$ can be reflected back successfully since 15 is greater than 12, consistent with the modification indicator.

**Element selection:** This predicate holds if the source data is an element with the specified tag. In the backward direction, this predicate returns the updated source data $S^\prime$ and the updated environment $E^\prime$ directly since the tags of elements are not allowed to change and nothing is needed to guard.

$$\text{xwithtag}_{\mathcal{C}}[\text{str}](S) = \begin{cases} 
\text{true}, & \text{if } S = \text{tag}(w_1) > [S_1] \text{ and } \text{tag} = \text{str} \\
\text{false}, & \text{else if } S = \text{tag}(w_1) > [S_1] \text{ and } \text{tag} \neq \text{str} \\
\text{fail}, & \text{otherwise}
\end{cases}$$

$$\text{xwithtag}_{\mathcal{D}}[\text{str}](S, S^\prime) = (S^\prime, E)$$

**Content filter:** This predicate holds if the source data is an element. Since an element is not allowed to change into a text, and vice versa, this predicate returns the updated source data $S^\prime$ and the updated environment $E^\prime$ directly in its backward execution.

$$\text{xiselement}_{\mathcal{C}}(S) = \begin{cases} 
\text{true}, & \text{if } S = <\text{tag}(w_1)>[S_1] \\
\text{false}, & \text{else if } S = \text{str}(w_1) \\
\text{fail}, & \text{otherwise}
\end{cases}$$

$$\text{xiselement}_{\mathcal{D}}(S, S^\prime) = (S^\prime, E)$$

### 3.6 Programming examples

A program of this bidirectional language is given in Figure 9. This program implements the recursive $\text{toc}$ function in Figure 1. The program is divided into several pieces just for the convenience of reading and explanation, with the help of an informal keyword where. The function body first gets the contents of the input element. Its contents consist of the author, title, section and other elements. Next, only section elements are chosen, and for each section element, the code $\text{XSec}$ is used to construct the section element in the view with the help of $\text{XTitle}$ and $\text{XSubTitles}$, which correspond to the expression $\text{Section}/\text{title}$ and the recursive function call in the example query, respectively.

A forward execution of invoking the $\text{toc}$ function is illustrated in Table 1 and Table 2, and a corresponding backward execution is given in Table 3 and Table 4. The data and environments used in these executions are given in Figure 10, where the data simplifies the XML data in Figure 2 to make the execution traces shorter. The forward
invocation starts with the source \( \cdot \) and the environment \( E_0 \) and returns the view \( \langle \text{section}^{\text{ori}, e} > [S_4] \). Suppose the view is changed into \( \langle \text{section}^{\text{ori}, e} > [S_4'] \). Then, using this updated view and the source \( \cdot \) as the input of the backward invocation under the environment \( E_0 \), we will get the updated environment \( E_0' \), in which the updated data \( S' \) contains

### Table 1  An example of forward execution (Part 1).

<table>
<thead>
<tr>
<th>Index</th>
<th>Tran</th>
<th>Src</th>
<th>Env</th>
<th>Tar</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>XBody</td>
<td>1</td>
<td>( C_0 )</td>
<td>( \langle \text{section}^{\text{ori}, e} &gt; [S_4] )</td>
</tr>
<tr>
<td>f1.1</td>
<td>xvar Shook-or-section</td>
<td>1</td>
<td>( C_0 )</td>
<td>( S )</td>
</tr>
<tr>
<td>f1.2</td>
<td>xchild</td>
<td>1</td>
<td>( C_0 )</td>
<td>( S_1, S_2 )</td>
</tr>
<tr>
<td>f1.3</td>
<td>xmap xif (xwriting section) XSec (xconst ()))</td>
<td>( S_1, S_2 )</td>
<td>( C_0 )</td>
<td>( \langle \text{section}^{\text{ori}, e} &gt; [S_4'] )</td>
</tr>
<tr>
<td>f1.3.1</td>
<td>xif (xwriting section) XSec (xconst ()))</td>
<td>( S_1 )</td>
<td>( C_0 )</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 2  An example of forward execution (Part 2).

<table>
<thead>
<tr>
<th>Index</th>
<th>Tran</th>
<th>Src</th>
<th>Env</th>
<th>Tar</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3.3.2.2.1.2</td>
<td>XSubTitles</td>
<td>1</td>
<td>( C_1 )</td>
<td>( S_3 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.2.1</td>
<td>xvar Shook-or-section XBody</td>
<td>1</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.2.2</td>
<td>xlet XBody</td>
<td>1</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.2.3</td>
<td>xmap xif (xwriting section) xid (xconst ()))</td>
<td>( S_4 )</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.2.3.1</td>
<td>xif (xwriting section) XSec (xconst ()))</td>
<td>( S_4 )</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.3.1.1</td>
<td>xif (xwriting title) xid (xconst ()))</td>
<td>( S_4 )</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.3.1.2</td>
<td>xvar xid</td>
<td>1</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
<tr>
<td>f3.3.2.2.1.3.1.3</td>
<td>xif (xwriting title) xid (xconst ()))</td>
<td>( S_4 )</td>
<td>( C_1 )</td>
<td>( S_4 )</td>
</tr>
</tbody>
</table>

Table 1  An example of forward execution (Part 1).
Table 3 An example of backward execution (Part 1).

<table>
<thead>
<tr>
<th>Index</th>
<th>Tran</th>
<th>Sec</th>
<th>UEnv</th>
<th>USec</th>
<th>UEval</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1.1.1</td>
<td>XBody</td>
<td>()</td>
<td>&lt;section xwithtag section ()&gt;</td>
<td>E6</td>
<td>E6</td>
</tr>
<tr>
<td>b1.1.2</td>
<td>xvar Shook-or-section</td>
<td>()</td>
<td>S'</td>
<td>E6</td>
<td>E6</td>
</tr>
<tr>
<td>b1.1.3</td>
<td>xchild</td>
<td>E6</td>
<td>S6, S7,S8</td>
<td>E6</td>
<td>S6, S7, S8</td>
</tr>
<tr>
<td>b1.1.4</td>
<td>xmap (xwithtag section) XSec (xconst ())</td>
<td>S6, S7, S8</td>
<td>&lt;section xwithtag section ()&gt;</td>
<td>E6</td>
<td>S6, S7, S8</td>
</tr>
</tbody>
</table>

Table 4 An example of backward execution (Part 2).

<table>
<thead>
<tr>
<th>Index</th>
<th>Tran</th>
<th>Sec</th>
<th>UEnv</th>
<th>USec</th>
<th>UEval</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1.1.5</td>
<td>XSubTitles</td>
<td>()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b1.1.6</td>
<td>xvar Shook-or-section</td>
<td>()</td>
<td>S'</td>
<td>E6</td>
<td>E6</td>
</tr>
<tr>
<td>b1.1.7</td>
<td>xlet Shook-or-section XBody</td>
<td>()</td>
<td>S7</td>
<td>E6</td>
<td>S7</td>
</tr>
<tr>
<td>b1.1.8</td>
<td>xchild</td>
<td>S7</td>
<td>S8</td>
<td>E6</td>
<td>S8</td>
</tr>
<tr>
<td>b1.2.1</td>
<td>xmap (xwithtag section) XSec (xconst ())</td>
<td>S7, S8</td>
<td>&lt;section xwithtag section ()&gt;</td>
<td>E6</td>
<td>S7, S8</td>
</tr>
</tbody>
</table>

The modified section title. The source () is not updated.

The executions illustrated in the aforementioned tables are organized by using indexes, which indicate the execution sequence or dependency of transformations. For example, the transformation f1 depends on the transformations
the transformation $b_1$ needs the forward transformations $f_1$ and $f_1$ to generate its source data. For example, the line $f_1.3.3$ in Table 1 says under the context $c_0$ the conditional transformation generates the target $<\text{section}(\text{ori}.)>$ from $S_3$, and if the details of this conditional transformation is wanted, then we can check the lines with the indexes $f_1.3.3.1$ and $f_1.3.3.2$.

4 Well-behaved bidirectional transformations

In this section, we will give two properties that well-behaved bidirectional transformations should have. The transformations defined in the previous section and its revised semantics for supporting view insertions are designed with such two properties considered. A rigorous proof will be completed as our future work.

4.1 Stability property

This property says if the view is not updated, then after backward transformation the source is not changed either. This property is called GETPUT property in [5], [6], and acceptable condition in [2]. The stability property of our bidirectional transformation is described in Theorem 1, where $X$ does not have free variables (i.e., all variables are bound by $xlet$) and the evaluation environment is $\cdot$.

**Theorem 1** If $[X](S) = T$, then $[X](S, T) = (S, \cdot)$. 

![Fig. 10 Data and environments for the XQuery example.](image-url)
4.2 Extended round-tripping property

The second property is called extended round-tripping property. Before introducing this property, we first discuss why the existing round-tripping property is not suitable for the view updating problem of XQuery. This property is initially proposed to characterize view updating schemes for relational databases [2], and then taken by a series of bidirectional transformation languages such as [5], [6]. Actually, this property is not suitable for relational view updating, either. In Appendix Appendix A., we show that this property is not adopted in the major database management systems such as Microsoft SQL Server 2003, Oracle DB 10g and MySQL 5.0. The round-tripping property is called PUTGET property in [5], [6], and consistent condition in [2].

4.2.1 Limitations of round-tripping property

Suppose X is a bidirectional transformation. Then, in our setting, the round-tripping property is represented as: if \( \lbrack X \rbrack(S.T) = (S'.E') \) and \( \lbrack X \rbrack(S.T') = T' \), then \( T = T' \). However, this property is too limited for our bidirectional language, which is expressive to interpret XQuery. The following example explains the limitation. Suppose we have the following source data.

\[
\begin{align*}
\text{bib}^{(\text{ori}, s)} & \left[ \text{book}^{(\text{ori}, s)} \right] \left[ \text{title}^{(\text{ori}, s)} \right] \left[ \text{Database}^{(\text{ori}, s)} \right], <\text{price}^{(\text{ori}, s)}>[2]^{(\text{ori}, s)}], \\
\text{book}^{(\text{ori}, s)} & \left[ \text{title}^{(\text{ori}, s)} \right] \left[ \text{Network}^{(\text{ori}, s)} \right], <\text{price}^{(\text{ori}, s)}>[10]^{(\text{ori}, s)}]]
\end{align*}
\]

And then, we want a view that consists of all books and a table-of-contents (toc) containing their titles. The transformation \text{BibView} and the generated view are given below.

\[
\text{BibView} = \text{xchild} \times \text{let} \text{bbooks} (xconst <\text{bibview}^{(\text{ori}, c)}>[()])); \\
\times \text{setcnt} (\text{MkToc} \times \text{xvar} \text{bbooks}); \text{mmap} (\text{xchild}; \text{GetTitle})
\]

\[
\text{MkToc} = \text{xconst} <\text{toc}^{(\text{ori}, c)}>[(); \times \text{setcnt}(\text{xvar} \text{bbooks}; \text{xmap} (\text{xchild}; \text{GetTitle}))
\]

\[
\text{GetTitle} = \text{xmap} (\text{xiif} (\text{with} \text{tag} \text{title})\text{xid} xconst ())
\]

\[
\begin{align*}
\text{bibview}^{(\text{ori}, c)} & \left[ \text{toc}^{(\text{ori}, c)} \right] \left[ \text{title}^{(\text{ori}, c)} \right] \left[ \text{Database}^{(\text{ori}, c)} \right], <\text{title}^{(\text{ori}, c)}>[\text{Network}^{(\text{ori}, c)}]]], \\
\text{book}^{(\text{ori}, c)} & \left[ \text{title}^{(\text{ori}, c)} \right] \left[ \text{Database}^{(\text{ori}, c)} \right], <\text{price}^{(\text{ori}, c)}>[2]^{(\text{ori}, c)}], \\
\text{book}^{(\text{ori}, c)} & \left[ \text{title}^{(\text{ori}, c)} \right] \left[ \text{Network}^{(\text{ori}, c)} \right], <\text{price}^{(\text{ori}, c)}>[10]^{(\text{ori}, c)}]]
\end{align*}
\]

In the \text{bibview} element above, each title element appears twice. Suppose we change <\text{title}^{(\text{ori}, c)}>[\text{Database}^{(\text{ori}, c)}] in the toc element into <\text{title}^{(\text{ori}, c)}>[\text{Network}^{(\text{mod}, c)}]. Then, after updating the source data and running forward the transformation \text{BibView} again, we get the following new updated view.

\[
\begin{align*}
\text{bibview}^{(\text{ori}, c)} & \left[ \text{toc}^{(\text{ori}, c)} \right] \left[ \text{title}^{(\text{ori}, c)} \right] \left[ \text{Database}^{(\text{mod}, c)} \right], <\text{title}^{(\text{ori}, c)}>[\text{Network}^{(\text{ori}, c)}]]], \\
\text{book}^{(\text{ori}, c)} & \left[ \text{title}^{(\text{ori}, c)} \right] \left[ \text{Database}^{(\text{mod}, c)} \right], <\text{price}^{(\text{ori}, c)}>[2]^{(\text{ori}, c)}], \\
\text{book}^{(\text{ori}, c)} & \left[ \text{title}^{(\text{ori}, c)} \right] \left[ \text{Network}^{(\text{mod}, c)} \right], <\text{price}^{(\text{ori}, c)}>[10]^{(\text{ori}, c)}]]
\end{align*}
\]

Hence, the old updated view and the new view from the updated source are not identical and the round-tripping property fails to accommodate the above transformation of our bidirectional language. In the new view, the change of title in the book element is called an update side-effect. The round-tripping property does not take update side-effects into account.

More generally speaking, the above example shows a case where the round-tripping property fails when a view includes data replicas. Such kind of views are very common when a query performs join operations on relational tables or XML data. For example, a value on the view is replicated if the record including it is joined with several records in another table. Any change to this value can lead to a failure of the round-tripping property.

4.2.2 Extended round-tripping property

Our bidirectional transformation language satisfies the extended round-tripping property, which is shown by the following theorem. The transformation \( X \) in the theorem does not contain free variables.

**Theorem 2** If \( \lbrack X \rbrack(S,T) = (S',E') \) and \( \lbrack X \rbrack(S') = T' \), then \( T \subseteq T' \).
Translation of XQuery core into bidirectional language

For the example in the previous section, the updated view and the new view from the updated source satisfies the update-keeping relation, though they are not identical. The update-keeping relation is reflexive, and hence if a view updating scheme satisfies the round-tripping property, then it also satisfies the extended round-tripping property.

5 Translation of XQuery core into bidirectional language

That is, we do not require the updated view $T$ and the new view $T'$ from the updated source to be identical. Instead, we relate $T$ to $T'$ with the update-keeping relation $\sqsubseteq$, which is defined in Figure 11. Informally, this property requires all updates made to $T$ are still kept in $T'$, and moreover $T'$ may include more updates than $T$ due to update side-effects. The update-keeping relation characterizes what update side-effects can appear on $T'$. For example, a string annotated by del cannot be changed into mod due to update-side effects. Though update side-effects on views have been recognized in work [14], it does not give a way to describe what update side-effects are allowed on the new views generated from the updated source. The rationale behind some cases of the update-keeping relation is explained below.

\begin{itemize}
  \item $str^{\text{ori}(i)} \sqsubseteq str^{\text{mod}(i)}$. A replica of string $str^{\text{ori}(i)}$ is changed to $str^{\text{mod}(i)}$, so the string becomes $str^{\text{mod}(i)}$ in the view generated from the updated source due to update side-effects.
  \item $str^{\text{ori}(i)} \sqsubseteq str^{\text{mod}(i)}$, if $str' > str$. A replica of string $str^{\text{ori}(i)}$ is changed to $str^{\text{mod}(i)}$, which is consistent with the modification indicator in this string, so the string becomes $str^{\text{mod}(i)}$ in the new view generated from the updated source.
  \item $str^{\text{ori}(i)} \sqsubseteq str^{\text{del}(i)}$, $i \in \{↑, ↓, →, •\}$. The string $str^{\text{ori}(i)}$ becomes $str^{\text{del}(i)}$ if it has a replica on the view tagged as deleted.
  \item $\langle \text{tag}^{\text{ori}(i)} \rangle[S] \sqsubset \langle \text{tag}^{\text{del}(i)} \rangle[S']$, if $S \sqsupseteq S'$. The $\langle \text{tag}^{\text{ori}(i)} \rangle[S]$ has a replica on the view tagged as deleted, so it becomes $\langle \text{tag}^{\text{del}(i)} \rangle[S']$ after the transformation is applied to the updated source.
\end{itemize}

For the example in the previous section, the updated view and the new view from the updated source satisfy the update-keeping relation, though they are not identical. The update-keeping relation is reflexive, and hence if a view updating scheme satisfies the round-tripping property, then it also satisfies the extended round-tripping property.

5 Translation of XQuery core into bidirectional language

The expressions of XQuery can be normalized to the equivalent expressions in XQuery Core, for instance, by the Galax XQuery engine [19]. The syntax of XQuery core is more compact. Hence, like the work [20], we implement bidirectional XQuery based on the XQuery Core syntax.

5.1 Introduction of XQuery core

The syntax of XQuery Core interpreted in this paper is given in Figure 12. The XQuery Core expressions include the literal string value $\text{String}$, the empty sequence $()$, the sequence construction expression, the variable $\$Var$, the for and let expressions, the conditional expression, the expression for XPath steps, the element construction expression and the function call. The formal semantics of XQuery Core is defined at [21].

In the aforementioned syntax, the constructs child, descendant and self are three forward XPath axes. Evaluating XPath axes needs to know their context nodes. As in [21], the special variable $\$$fs: dot is used to represent context nodes of XPath axes. The XQuery Core here does not include reverse axes of XPath, such as the parent axis. This axis returns the parent of the current context node. Actually, it is difficult to implement reverse axes using the technique in the previous section since from the source element we have no information about its parent node or
An expressive bidirectional transformation language for XQuery view update

Fig. 12 Syntax of the XQuery core.

its ancestor node. This limitation might be addressed with the technique in [22] by rewriting path expressions with reverse axes into equivalent reverse-axis-free ones.

XQuery includes a lot of built-in functions, such as \texttt{fn:sum} and \texttt{fn:data}. In order to support full XQuery, we need to define the bidirectional versions of these functions in the underlying bidirectional language. The tricky thing is that these implementations must satisfy the well-behaved conditions for bidirectional transformations. For example, the backward semantics of \texttt{fn:sum} must ensure that the sum of all items in its argument is not changed for the updated source and updated evaluation environment, otherwise the extended round-tripping property will be violated. Implementing these built-in functions is our future work.

5.2 The translation

Figure 13 gives the rules for translating XQuery Core into the bidirectional language. The operation $\llbracket expr \rrbracket_I$ returns the translation result for the XQuery Core expression $expr$. Note that $I$ is just a part of the operation name, not a parameter of the translation. With such an interpretation, XQuery Core expressions can be executed in two directions: generating the view in the forward direction and putting view updates back in the backward direction. The translation is not difficult due to the expressiveness of the underlying bidirectional language. Some rules are illustrated below.

In the rule of \texttt{for} expression, the subexpression $Expr_1$ is first translated, and then composed with an \texttt{xmap}, which takes the transformation $\llbracket Var [Expr_2] \rrbracket_I$ as its argument. That is, the variable $Var$ is bound to each value in the sequence returned by $\llbracket Expr_1 \rrbracket_I$, and then used within the scope $\llbracket Expr_2 \rrbracket_I$.

In the XQuery Core, the expression \texttt{Axis NodeTest} means that \texttt{Axis} first produces a list of nodes from its context node, and then from this list \texttt{NodeTest} selects the nodes by its conditions. In the translation of this expression, we need to explicitly get the context node of an axis by referring to the value of the reserved variable $\$fs:dot$, and then compose the translation results of \texttt{Axis} and \texttt{NodeTest}.

The \texttt{child} axis of XPath is primitively defined by \texttt{xchild} in the bidirectional language, while the \texttt{descendant} axis is not. Instead, this axis is implemented by the function \texttt{xdes} below, which returns all descendant nodes of the input node $\$node$.

\begin{verbatim}
fun xdes($node) = xvar $node; xif xisnode ($xchild;$id||xmap DeepNodes) (xconst ())
  where DeepNodes = xlet $cnode (xfunapp xdes [xvar $cnode])
\end{verbatim}

If the input node is a text node, then it does not have any descendant, so the function \texttt{xdes} returns (); if the input node is an element node, then the result includes its content nodes and their descendants.

The functions in XQuery are translated into functions in the bidirectional language. For example, the following
The translation satisfies the property in Theorem 3, which says that the translation preserves the semantics of XQuery Core. The values in the underlying bidirectional language are annotated with updating and origin annotations. To be consistent with XQuery values, these annotations should be erased. The erase operator for this purpose is defined below.

\[
\text{erase}(S) = \begin{cases} 
  (), \text{ if } S = () \\
  \text{str}, \text{ if } S = \text{str}^{(w, o)} \\
  <\text{tag}>[\text{erase}(S)], \text{ if } S = <\text{tag}^{(w,o)}>[S] \\
  \text{erase}(v), \text{erase}(V), \text{ if } S = v, V 
\end{cases}
\]

The \text{erase} operator is extended to an environment \( C \) by the following definition.

\[
\text{erase}(C) = \begin{cases} 
  ., \text{ if } C = . \\
  \text{erase}(C'), \text{Var} \mapsto \text{erase}(S), \text{ if } C = C', \text{Var} \mapsto S
\end{cases}
\]
Theorem 3 Let $C$ be an environment that maps variables to values. If an XQuery Core expression $Expr$ has the value $Val$ under $\text{erase}(C)$, then the forward transformation $T = \text{erase}(T) = Val$.

The proof is done by induction on the structure of the XQuery Core expression $Expr$.

6 The type system

In this section, we will design a type system for the bidirectional transformation language and prove that this type system is sound with respect to the forward semantics of this language. On the other hand, we also prove that the types of updated source data are preserved after backward executions of well-typed programs. The type system annotates well-typed programs with types. In the next section, we will see annotated types provide guiding information for the language to process updated views with insertions in its backward semantics.

6.1 Syntax of types

The syntax of types is given in Figure 14, which defines the recursive regular expression types for XML. They are similar to the regular expression types in [23] except the recursive type, the boxed type $\mu.a.\tau$, and the origin annotation $o$, the string type and element type. The origin annotation in the type is still either s or c. The boxed type will be introduced later when we discuss the typing rule for $\text{xmap}$. The notation $S \in \tau$, defined in Figure 15, means $S$ has the type $\tau$. A boxed type contains boxed values, which will be used in the next section for describing the property of the revised split operator. The boxed type will be ignored wherever possible since it appears only in the annotations of $\text{xmap}$ as shown later.

The recursive type $\mu.a.\tau$ is necessary to describe recursive XML data, such as a book that has recursive subsections. The recursive type $\mu.a.\tau$ is regarded as equivalent to its unfolded form $\tau[\mu.a.\tau/a]$, which means all occurrences of the free type variable $a$ in $\tau$ are replaced with $\mu.a.\tau$. For brevity, recursive types and type variables will not be considered in some operators defined over type cases. That is, when a recursive type encountered, we just use its unfolded form. We require that all recursive types have different bound type variables. This can be achieved by alpha-conversion.

Note that $\tau^*$ is not necessary in the syntax, since it can be defined as $\mu.a.()\tau,a$.

6.2 Typing rules

The typing rules for the bidirectional transformation language are defined in Figure 16. The judgment has the form $\Gamma \vdash X : \tau \Rightarrow \tau' \Rightarrow X'$, meaning that under the typing context $\Gamma$, if the source data has the type $\tau$, then the transformation $X$ will generate a view with the type $\tau'$ after forward executions, and on the other hand, if the updated view has the type $\tau'$, then $X$ will generate an updated source data with the type $\tau$ after backward executions. The typing context $\Gamma$ maps variables to types, or maps function names together with the types of their arguments to types. The transformation $X'$ is the result of annotating $X$ with types. The semantics of $X'$ will be described in the next section.

In the typing rule for $\text{xconst } S$, the operation $\text{mkty}(S)$ makes a type, say $\tau'$, from $S$: if $S = ()$, then $\tau' = ()$; if $S = \text{xstr}(),$ then $\tau' = \text{string}$; if $S = \text{<tag>>(xstr[c])}()$, then $\tau' = \text{<tag>}(c)$; $\tau_1, \tau_2, ..., \tau_n$, and $\tau$, hence, the type annotation on $X$ needs to take into account all possible input types, which are then represented as $\tau_1[...]\tau_n$ to produce $X''$.

There are seven constructs annotated with types: $\text{xvar}$, $\text{xchild}$, the parallel composition transformation $\llbracket X \rrbracket$, $\text{xmap}$, $\text{xif}$, and $\text{xfunapp}$. The first six constructs need type information to process updated views with insertions. The last construct $\text{xfunapp}$ uses annotated information to dynamically annotate the function body to be called. A function

\[
\tau := a | () | \text{string} | \text{<tag>}[\tau] | \tau \ast | \tau, \tau | \tau | \mu.a.\tau | [\tau]
\]

Fig. 14 Syntax of recursive regular expression types.

\[
\begin{array}{c}
\begin{array}{c}
\Gamma \vdash S_1 \in \tau_1 \\
\Gamma \vdash S_2 \in \tau_2 \\
\end{array}
\end{array}
\]

Fig. 15 Semantics of types.
boxed view types of its two argument transformations.

different points with arguments of different types, so its body needs to be annotated according to argument types at each function calling point.

xmap.

\[
\Gamma \vdash \text{xid} : \tau \rightarrow \tau \Rightarrow \text{xid}
\]

\[
\Gamma \vdash \text{xvar} : \tau \rightarrow \tau' \Rightarrow \text{xvar} \Rightarrow \text{Var}
\]

\[
\tau = <\text{tag}_{a_1}^\tau [\tau_1] \ldots <\text{tag}_{a_n}^\tau [\tau_n]>
\]

\[
\Gamma \vdash \text{xchild} : \tau \rightarrow \tau_1 \ldots \tau_n \Rightarrow \text{xchild} \Rightarrow \tau'' \Rightarrow \text{X''}
\]

\[
\Gamma \vdash \text{xconst} : \tau \rightarrow \tau'' \Rightarrow \text{X''}
\]

\[
\Gamma \vdash \text{xsetcunt} : \tau \rightarrow \text{xsetcunt} \Rightarrow \text{X''}
\]

\[
\Gamma \vdash \text{xfunapp} : \tau \rightarrow \tau' \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xif} : \tau \rightarrow \text{xif} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xlet} : \tau \rightarrow \text{xlet} \Rightarrow \text{Var} \times \text{X'}
\]

\[
\Gamma \vdash \text{xbox} : \tau \rightarrow \text{xbox} \Rightarrow \text{xbox} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xmap} : \tau \rightarrow \text{xmap} \Rightarrow \text{xmap} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xbox} : \tau \rightarrow \text{xbox} \Rightarrow \text{xbox} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xmap} : \tau \rightarrow \text{xmap} \Rightarrow \text{xmap} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xfunapp} : \text{frame} \times \tau \rightarrow \text{xfunapp} \Rightarrow \text{xfunapp} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xfunapp} : \text{frame} \times \tau \rightarrow \text{xfunapp} \Rightarrow \text{xfunapp} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xfunapp} : \text{frame} \times \tau \rightarrow \text{xfunapp} \Rightarrow \text{xfunapp} \Rightarrow \text{X'}
\]

\[
\Gamma \vdash \text{xfunapp} : \text{frame} \times \tau \rightarrow \text{xfunapp} \Rightarrow \text{xfunapp} \Rightarrow \text{X'}
\]

Fig. 16 Typing rules.
or element type, that is, when singlety(τ) = ∅, which implies the source data of xmap is (). The singlety operator is defined in Figure 19. At this case, the view type of xmap is trivially (), and the component transformation X is not checked and annotated since it does not have a chance to execute. That is, by using this rule, we do not need to care whether X is type-correct or not. The second typing rule applies when singlety(τ) ≠ ∅. At this case, the component transformation X will be applied to each single value in the source data. In the following, we explain for the second case how the view type of xmap is generated and how the component transformation X is annotated.

If the source type τ satisfies singlety(τ) ≠ ∅, the rules in Figure 18 are used to generate the view type of xmap. These rules derive the judgment of the form Γ ⊢ a xmap X : τ → τ', where the type τ' is obtained by traversing the structure of τ, and replacing each string or element type in τ with a boxed type obtained by type-checking X with this string or element type as the source type under Γ. The environment Θ consists of a sequence of substitutions, each having the form μa.τ/a. These substitutions can be applied to a type τ', written as τ'Θ, to replace all free variables a in τ' with μa. If Θ does not include any substitution, it is represented as -.

The relation between τ and τ' is reflected by the Lemma 4. The type τ' is then used to annotate xmap for supporting backward transformations. The view type of xmap is obtained by removing the boxes in τ' with the operation rmbox(τ'). That is, the boxed types only appear as annotations, and do not used as types for source data or views.

The rmbox operator is defined in Figure 19.

**Lemma 4** Given a transformation X, a context Γ and a sequence of substitutions Θ, if Γ; Θ ⊢ a X : τ ↔ τ', then τ has the identical structure as τ' at the box level modulo typing of X under Γ, written as τ ≜ X; Γ; Θ τ'. That is,

- if τ = a, then τ' = a;
- if τ = (), then τ' = ();
semantics of \textit{xmap} \textit{xchild} insertions. For example, suppose we have the transformation of \textit{X}

The boxed type can help update the source data in a more reasonable way for updated views with inserted values. Boxed types will be used in Section 7 when revising the backward semantics of \textit{xmap}. An example below explains the benefits of using boxed types.

Boxed types help identify which subsequence in the view of \textit{xmap} should be transformed by one backward execution of \textit{X} when the view includes inserted values. Boxed types will be used in Section 7 when revising the backward semantics of \textit{xmap}. An example below explains the benefits of using boxed types.

The component transformation of \textit{X} of \textit{xmap} should be annotated by taking into account every possible top-level \textit{string} or \textit{element} type in the source type of \textit{xmap}. These types are collected in a set by the \textit{singlety} operator and then represented as a choice type by using the \textit{mkchoicy} operator below to check \textit{X}. Thus, \textit{X} is annotated with the type information from all possible \textit{string} and \textit{element} types in \tau.

\begin{equation}
\text{mkchoicy}(\text{Set}) = \begin{cases} 
\tau, & \text{if } \text{Set} = \{\tau\} \\
\tau\text{mkchoicy}(\text{Set'})', & \text{if } \text{Set} = \{\tau\} \cup \text{Set'}
\end{cases}
\end{equation}

The boxed type can help update the source data in a more reasonable way for updated views with insertions. For example, suppose we have the transformation \textit{xmap} \textit{xchild}. If the source type is \textit{<bag>}{\textit{<apple>}{\textit{<tag>}[\{\tau\}]}}; then the view type annotation of \textit{xmap} is \textit{<apple>}{\textit{<tag>}[\{\tau\}]}; if the source type is \textit{<bag>}{\textit{<apple>}{\textit{<tag>}[\{\tau\}]}}. The different position of box can tell us whether the \textit{apple} elements in a view come from the same \textit{bag} element or from different \textit{bag} elements. If we insert a new \textit{apple} element on the view already containing some \textit{apple} elements, then in the first case, the new \textit{apple} element should be used together with other existing \textit{apple} elements by \textit{xchild} to update the source data, resulting in a new \textit{bag} element containing all \textit{apple} elements; while in the second case, the inserted \textit{apple} element should be processed independently by \textit{xchild}, and a new \textit{bag} element for this new \textit{apple} element is generated in the updated source data. In both cases, the updated source data has the valid type due to the information from the boxed type.

The typing rule of \textit{xiif} checks its two branches with its source type \tau refined by \textit{T}(\tau, P) and \textit{F}(\tau, P), respectively, which are defined in Figure 20. These operators generate more precise source types for each branch. Here are some brief explanation of these operators: the operator \textit{T}(\tau, \textit{xwithtag} str) selects in \tau \textit{element} types with the tag \textit{str}, while \textit{F}(\tau, \textit{xwithtag} str) returns the \textit{element} types with the tag other than \textit{str}; the operator \textit{T}(\tau, \textit{xiselement}) selects the \textit{element} types in \tau, while the operator \textit{F}(\tau, \textit{xiselement}) selects the \textit{string} types; both operators \textit{T}(\tau, \textit{xeq}) and \textit{F}(\tau, \textit{xeq}) return the source type without further selection since the source type \tau for \textit{xeq} can only a \textit{string} type or a choice of strings. The set of types returned by \textit{T}(\tau, P) and \textit{F}(\tau, P) are combined into a choice type by using \textit{mkchoicy}. An empty set returned by \textit{T}(\tau, P) means there is no source data of type \tau that makes \textit{P} true. At this
case, the true branch will not be type-checked, since it is never be executed. It is also similar for the false branch when $F(\tau, P)$ returns an empty set. It is a typing error when both sets $T(\tau, P)$ and $F(\tau, P)$ are empty based on the semantics of the predicates. Hence, there is no the typing rule of $\text{xif}$ where both $T(\tau, P)$ and $F(\tau, P)$ are empty.

The following example shows the usefulness of this typing feature. In this example, suppose we have the code $\text{xif}(\text{xwithtag} \text{book}) \ x\text{child} (\text{xconst}())$ and the source type $\langle \text{book} \rangle [\text{string}] [\text{string}]$. If the source type of $\text{xif}$ is directly applied to check its branches, the true branch will cause a type error since $\text{xchild}$ can only accept elements as its source. Actually, if the true branch is chosen at runtime, we know the $\text{xwithtag}$ predicate must hold and the source data of this branch must be an element. Hence, we can use the element type $\langle \text{book} \rangle [\text{string}]$ to check the true branch $\text{xchild}$.

There are two typing rules for function calls. If a function together with the types of its arguments is not in the domain of $F$, then the first rule is used, otherwise the second is taken. In the first rule, the function body $X$ is checked under the typing context, where the variable $\text{Var}$ is mapped to the type $\tau$, and the function name $\text{funname}$ together with these argument types is mapped to a fresh type variable $a$. The view type $\tau'$ of the function body $X$ probably contains the free type variable $a$ because of recursive function calls. Therefore the view type of $\text{xfunapp}$ in the first typing rule is a recursive type $\mu a. \tau'$. In the second rule, the function body will not be checked since its resulting type is already available. Note that the type-annotated function body in the first rule is not used in the typing result. This does not mean that we do not need type annotations in the function body. Instead, this is because we want to avoid the trouble of managing different versions of the same function with different type annotations. Our solution is to annotate function calls with the types of their arguments, and then use these types to dynamically type-check and annotate function bodies when meeting with function calls at runtime. The dynamic type checking does not fail since it just replays a static checking to produce annotations for the function body. Suppose an annotated function $\text{xfunapp}[\text{funname} \langle X_1, \ldots, X_n \rangle]$ is to be executed backward and the function $\text{funname}$ is defined as

$$\text{fun name}(\text{Var}_1, \ldots, \text{Var}_n) = X$$

Then, the body $X$ can be annotated by reasoning about the following judgment.

$$[\text{Var}_1 \mapsto \tau_1, \ldots, \text{Var}_n \mapsto \tau_n, \text{funname}(\tau_1, \ldots, \tau_n) \mapsto a] \vdash X : () \Rightarrow \tau' \Rightarrow X'$$

where $a$ is a fresh type variable. The annotations in $X'$ may include free occurrences of variable $a$, which need to be resolved before executing $X'$. The operation $X'[\mu a. \tau/a]$ defined in Figure 21 replaces all free type variables $a$ in $X'$ by its definition $\mu a. \tau$.

### 6.3 Type soundness

We will prove the type system in this section is sound with respect to the forward semantics of the bidirectional language. The forward semantics of the bidirectional language depends on the evaluation environment $C$. In the fol-
Following definition, we define the well-typed evaluation environment \( C \) with respect to a typing context \( \Gamma \), represented by \( C \in \Gamma \), which is needed to define the type-soundness property.

**Definition 1**  Given the evaluation environment \( C \) and the typing context \( \Gamma \), we say \( C \in \Gamma \) if for all \( \text{Var} \in \text{Dom}(C) \) we have \( \text{Var} \in \text{Dom}(\Gamma) \) and \( C(\text{Var}) \in \Gamma(\text{Var}) \).

The soundness property of this type system is stated by Theorem 5. As usual, well-typed programs do not get stuck in their forward executions, and generate views that have the view types derived by the type system.

**Theorem 5**  Given a transformation \( X \), a typing context \( \Gamma \), an evaluation environment \( C \) and a source value \( S \), if \( \Gamma \vdash X : \tau \iff \tau' \Rightarrow X' \), \( C \in \Gamma \) and \( S \in \tau \), then \( \text{Eval}^X_C(S) = T \) and \( T \in \tau' \).

The backward semantics of the bidirectional language depends on the evaluation environment \( \mathcal{E} \). The following definition defines the well-typed evaluation environment \( \mathcal{E} \) with respect to a typing context \( \Gamma \).

**Definition 2**  For the evaluation environment \( \mathcal{E} \) and the typing context \( \Gamma \), we say \( \mathcal{E} \in \Gamma \), if for all \( \text{Var} \in \text{Dom}(\mathcal{E}) \) we have \( \text{Var} \in \text{Dom}(\Gamma) \), \( \mathcal{E}(\text{Var}) = (S, S') \) and both \( S \in \Gamma(\text{Var}) \) and \( S' \in \Gamma(\text{Var}) \).

The type system presented has the property of backward type preservation, which is given in Theorem 6. By this property, the type of source data is respected after it is updated by backward executions of well-typed programs. The type system cannot guarantee that well-typed programs do not fail because conflicting and improper updates cannot be checked statically by this type system.

**Theorem 6**  Let \( \mathcal{E} \in \Gamma \) and \( S \in \tau \). If \( \Gamma \vdash X : \tau \iff \tau' \Rightarrow X' \), \( T \in \tau' \), and \( \text{Eval}^X_C(S, T) = (S', \mathcal{E}') \) or \( \text{Eval}^X_C(S, T) = (S', \mathcal{E}'') \), then \( S' \in \tau \) and \( \mathcal{E}' \in \Gamma \).

6.4 An example of type derivation

Figure 22 lists the types and typing contexts used by a type derivation example illustrated in Table 5 and Table 6. This example demonstrates how to type-check a call to the function \( \text{toc} \) in Figure 9. In this example, a judgment \( \Gamma_0 \vdash \text{XBody}() : () \iff \text{VSecTypes} \Rightarrow \text{XBody}' \) is derived. The annotated function body \( \text{XBody}' \) is given in Figure 23, where another annotated function body \( \text{XBody}'' \) is the result of type-checking \( \text{XSubTitles} \) in \( \text{XBody}' \). The derivation generating \( \text{XBody}' \) is indicated by the index \( t_1 \) in Table 5, while the derivation for \( \text{XBody}'' \) is indicated by the index \( t_1 \). These two annotated function bodies will be referenced by the examples in Section 7.

The indexes in the aforementioned tables serve as the same purposes as those in the examples in Section 3.6. That is, the indexes are used to represent the sequence and dependency relations between derivations. For example, the derivation of \( t_1 \) depends on the sequential derivations of \( t_1.1, t_1.2 \) and \( t_1.3 \). Typing an \( \text{xmap} \) transformation needs a
either the same as or similar to the existing ones for generating view types. For example, the xmap
transformation to generate its view type and also a derivation to annotate its component transformation. In the aforementioned tables, the derivations for annotating the component transformations of xmap are omitted because they are either the same as or similar to the existing ones for generating view types. For example, the xmap transformation of index 1.1.3 needs to annotate its component transformation xif (xwithtag “section”) XSec (xconst ()) under Γ₀ with the source type TitleTy|AuthorTy|SecTy (i.e., the result of singlety(Tlty, AuthorTy, SecTy)). With the help of T operator for xwithtag, the true branch XSec needs to be checked under Γ₀ with the source type SecTy. This check is already done by the derivation 1.1.3.2. With the F operator, the false branch xconst () is checked with the source type TitleTy|AuthorTy, which is trivial and similar to the derivations 1.1.3.1 and 1.1.3.2.

Together with the examples in Section 3.6, for the forward transformation, we have C₀ ∈ Γ₀, () ∈ () and hence <section>Ξ₃[0] ∈ VSecTy*: for the backward transformation, we have E₀ ∈ Γ₀, () ∈ (), <section>Ξ₂[0] ∈ VSecTy*, and then () ∈ () and E₀ ∈ Γ₀.

7 Revised bidirectional semantics for insertions

In this section, we will explain how to transform back the updated views that include insertions. For this purpose, we need to revise the forward or backward semantics of some transformations. In particular, from the revised backward semantics, we will see the types annotated on transformations provide guiding information for backward executions.

All examples in this section use the following source data. It includes a list of books and each book contains a title and a sequence of authors. This source data is called BookList.

```xml
<book><title>[a<orys>, s], <author>[b<orys>], [c<orys>]]
<book><title>[a<orys>, s], <author>[b<orys>], [c<orys>]]
```

7.1 Missing source data

The bidirectional transformations defined in Section 3 need the original source data to guide their backward executions. For example, the transformation xchild needs the original source element to determine what tag the updated source element could have in its backward semantics. However, if the updated views include insertions, it is

```
BookTy = <book>|[TitleTy, AuthorTy, SecTy]|
TitleTy = <title>|[string]*|
AuthorTy = <author>|[string]*|
SecTy = μ<sec>|[section*>][[TitleTy, SCntTy]]|
SCntTy = ParaTy

<table>
<thead>
<tr>
<th>Γ₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>toc(BookTy) $b</td>
</tr>
<tr>
<td>$book-or-section BookTy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Γ₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>toc(SecTy) $s</td>
</tr>
<tr>
<td>$book-or-section SecTy</td>
</tr>
<tr>
<td>$section SecTy</td>
</tr>
<tr>
<td>toc(BookTy) $b</td>
</tr>
<tr>
<td>$book-or-section BookTy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Γ₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>toc(SecTy) $s</td>
</tr>
<tr>
<td>$book-or-section SecTy</td>
</tr>
<tr>
<td>$section SecTy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Γ₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>toc(BookTy) $b</td>
</tr>
<tr>
<td>$book-or-section BookTy</td>
</tr>
</tbody>
</table>
```

Fig. 22 The source type and contexts for typing toc.
possible that some inserted values do not correspond to any original source data. Thus, the backward transformation of these inserted values will cause problems according to the existing bidirectional semantics. The following example explains how the source data is missing for inserted values on views.

Suppose we have the transformation \( \text{xmap} \) \( \text{bidir} \), where type annotations on \( \text{xmap} \) are ignored for brevity (also in some examples later), and apply it to the source BookList. The generated view is identical to the source. The following is an updated view with a new inserted book.

\[
\text{book(ori,s), <author(ori,s)>[<title(ori,s)>], \text{title(ori,s)>[<author(ori,s)>]<title(ori,s)>], \text{author(ori,s)>[<title(ori,s)>], \text{title(ori,s)>[<author(ori,s)>]<title(ori,s)>], \text{author(ori,s)>[<title(ori,s)>], \text{title(ori,s)>[<author(ori,s)>]<title(ori,s)>],}}
\]

Now we transform backward the updated view, that is, transform backward each book in the view and its corresponding book in BookList by \( \text{xid} \). For the first book and the second book in the updated view, they have the first book and the second book in BookList as their sources, respectively. However, for the third book in the updated view, it does not correspond to any book in BookList.

The missing source data is denoted by \( \Omega \) as in [5]. When \( \text{xmap} \) \( X \) is executed backward, if its view includes inserted values on views, it takes \( \Omega \) as its source, then \( \Omega \) may also be the source of the involved forward executions. For this case, we have the convention that a forward transformation maps \( \Omega \) to \( \Omega \). Since \( \Omega \) represents a nonexistent
value, its length is regarded as 0 and its type can be any type, that is \( \Omega \in \tau \) for any \( \tau \).

7.2 Revised backward semantics

The difficulty of revising backward semantics is caused by the missing source data. Without information provided by the source data some backward executions do not know how to proceed. At this case, we will use the types annotated on transformations to guide the backward executions.

7.2.1 Constant transformation

If the source data is missing, the constant transformation will fail since we cannot construct the updated source data even if we have the source-data type of \( xconst \). In addition, the updated view may be an inserted value, as shown by the following revised backward semantics of \( xconst \).

\[
\| xconst \|_{T_c}(S, T) = \begin{cases} (S, E), & \text{if } S \neq \Omega \text{ and } T = T_c \\ (S, E), & \text{if } S \neq \Omega, T = <\text{tag}^{\text{ins},c}>([]) \text{ or } T = s^{\text{ins},c} \\ \text{fail}, & \text{otherwise} \end{cases}
\]

Here is an example that explains when \( xconst \) could have an inserted element as its updated view. The code for this example is given below.

\[
xmap (xlet $b (<\text{pack}^{\text{ori},c}>([])): xsetcnt(xvar $b)))
\]
Recall the definition of its backward semantics, \( xchild \) the last element in the updated BookList based on the forward semantics of ori BookList code to the updated \( xchild \) take () as its source data and < this view is supposed to come from a source element with the type updated source data. 

7.2.2 Element deconstruction

considers view insertions. \( xchild \) the updated element. This problem is solved by the annotated type on ins Applying the above code to the source BookList It should mention that if the source data of \( xconst \), \( c \) 

\[
\llbracket xchild \rrbracket_d(S, T) =
\begin{cases}
<T, E>, & \text{if } S = <tag^{(w,o)}>[T_i], T \in T_i \text{ and } S \in T_i \\
<T, E>, & \text{if } S = \Omega, T = <tag^{(i,n)}>[T_i], T \in T_i \\
\text{fail}, & \text{otherwise}
\end{cases}
\]

Fig. 23 The annotated function bodies of \( toc \) 

Applying the above code to the source BookList, we get a view that includes two pack elements, each of which contains inside a book from BookList. Now if we insert into the view a new pack element with the annotation (ins, c) containing a new book element as the last element, then the backward execution of the xconst above will take () as its source data and xconst <pack[ins,c]>[()] as its updated view. The inserted book element will appear as the last element in the updated BookList after the backward execution of xmap. Note that after applying the above code to the updated BookList the updating status ins on the last pack element in the view will be changed into ori based on the forward semantics of xconst. This will be reflected in the new update-keeping relation, which considers view insertions.

7.2.2 Element deconstruction

If the transformation xchild does not have its source, it does not know what tag the updated source should have. Recall the definition of its backward semantics, xchild needs the tag of the original element to determine the tag of the updated element. This problem is solved by the annotated type on xchild, which is the type of the source data. The source-data type of xchild must have the form \(<tag^{(w,o)}>[T_1],...<tag^{(i,n)}>[T_n]\). If the view \( T \) has the type \( T_1 \), then this view is supposed to come from a source element with the type \(<tag^{(i,n)}>[T_1]\), so we use \( tag^{(i,n)}_j \) as the tag of the updated source data.
has the same type as the content of the original source. This check is necessary for the revised backward semantics of xchild to satisfy backward type preservation property, and the reason is given below.

Suppose the source-data type of xchild is \(<tag>\tau_1</tag>\)\([\tau_2]_1[\tau_2]\). Then, its view type for this source-data type is \(\tau_1[\tau_2]\) according to the typing rule of xchild. Let the source data \(S\) have the type \(<tag>\tau_1</tag>\), so the original view of xchild has the type \(\tau_1\). However, after updated with insertions, even if the updated view \(T\) has the type \(\tau_1[\tau_2]\), \(T\) not necessarily has the type \(\tau_1\). Hence, the updated source data may not have the type \(<tag>\tau_1</tag>\), or even not the type \(<tag>\tau_1</tag>\)\([\tau_2]\).

7.2.3 Variable reference

The xvar \(Var\) transformation returns the original source data in its backward semantics, so if the source data is missing, its backward executions will fail. In the revised backward semantics of xvar, a new \(mg\) operator, defined in Figure 24, is used to merge \(T\) and the current updated value of \(Var\) in \(E\). This new operator is directed by the type annotated on xvar, which characterizes the structure of the expected merging result. This new \(mg\) operator is able to merge two values that may not have identical structures due to insertions.

\[
\llbracket xvar^* \rrbracket_{E}(S, T') = \begin{cases} 
(S, E'), & \text{if } S \neq \emptyset, E = \emptyset, Var \mapsto (S_1, S_2), E_2, Var \notin Dom(E_2) \\
\emptyset, & \text{otherwise}
\end{cases}
\]

In the revised backward semantics of xvar, the variable \(Var\) in \(E\) may be bound to a pair of \(\emptyset\)s by the most recent enclosed xlet. Then after merging, the binding for \(Var\) may change from \(Var \mapsto (\emptyset, \emptyset)\) into \(Var \mapsto (\emptyset, T')\).

The following are two examples explaining the new \(mg\) operator. In the definition of new \(mg\), the operation \(allins(S)\) returns true if all top-level strings or elements in \(S\) are annotated with \(ins\), and the operation \(orilen(S)\) returns the number of top-level strings or elements in \(S\) that are not annotated with \(ins\). In the first example, we suppose \(S_1\) and \(S_2\) shown below will be merged, where \(year\) is an optional element.

\[
\begin{align*}
S_1 & : \text{<title>}{\text{[str1]}}\text{,[author>}{\text{[str2]}}\text{, <author>}{\text{[str3]}}\text{, <year>}{\text{[str4]}}\text{, <year>}{\text{[str5]}}\text{]}
S_2 & : \text{<title>}{\text{[str6]}}\text{,[author>}{\text{[str7]}}\text{, <author>}{\text{[str8]}}\text{, <author>}{\text{[str9]}}\text{, <year>}{\text{[str10]}}\text{]}
\end{align*}
\]

Let the expected merging result have the type \(\tau = \text{Title,Authors,Year}()\), where \(title\), \(author\), and \(year\) represent the types for the title, author and year elements, respectively. Then, the operation \(mg(\emptyset, \emptyset, \emptyset)\) could return the following data that merges all insertions and modifications in \(S_1\) and \(S_2\):

\[
\begin{align*}
\text{<title>}{\text{[str11]}}\text{,[author>}{\text{[str12]}}\text{, <author>}{\text{[str13]}}\text{, <author>}{\text{[str14]}}\text{, <year>}{\text{[str15]}}\text{, <year>}{\text{[str16]}}\text{]}
\end{align*}
\]

The second example explains a scenario where the view is a join of a book title and all its authors. In this scenario, when a new author and the book title is inserted, the inserted title needs to merge with the originally existing title. This merge in this example is processed by the ninth and tenth rules in Figure 24. Suppose the original source contains an element \(\text{Bob}^{\text{ori}(s)}\) for an author and the environment \(C\) maps a variable \(\text{Stitle}\) for a book title to Database\(^{\text{ori}(s)}\). The type of title is assumed to be string*, and the source takes type string*. For the transformation \(\text{map} \ (\text{xlet $a$ xvar $S$ xvar $S$ title})\), we get the view \(\text{Bob}^{\text{ori}(s)}\) Database\(^{\text{ori}(s)}\). That is, the transformation joins the book title Database\(^{\text{ori}(s)}\) to each author in the source. If the view is updated to \(\text{Bob}^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), then the backward execution of xvar needs to merge Database\(^{\text{ori}(s)}\) and Database\(^{\text{ori}(s)}\) by using the ninth rule of the \(mg\) operator. If the view is updated to \(\text{Bob}^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), Database\(^{\text{ori}(s)}\), the title is changed to \(\text{WebDB}^{\text{mod}(s)}\) according to the tenth rule of the \(mg\) operator. That is, when an inserted string has a corresponding one from the source, it is not regarded as inserted after merging, so does when an inserted string has a corresponding one from code (shown by the eleventh rule of the \(mg\) operator).

7.2.4 Conditional transformation

If the source data of xif is not missing and its predicate has the value true or false, then the revised backward semantics is the same as that before revision, that is, the branch \(X_1\) or \(X_2\) is chosen to run backward according to the value of the predicate. On the other hand, if the source data of xif is \(\emptyset\), then xif loses the information of how to
advance its backward executions. At this case, the types annotated on xif provide such information. If the updated view belongs to the view type of the branch X₁, then it is supposed to be generated by X₁, so X₁ will be chosen; if the updated view has the view type of X₂, then X₂ is chosen. After backward executions of X₁ or X₂, the updated
source and updated evaluation environment are required to make sure the predicate of xif has the corresponding value. That is, if \( X_1 \) is chosen, then the predicate must be true for the updated source under the updated evaluation environment, and similarly if \( X_2 \) is chosen. This requirement keeps xif transformation onto the same branch in its backward and forward transformations, as discussed in Section 3.4. If the updated view belongs to the view types of both branches, then either branch with the above requirement satisfied can be chosen.

\[
\xi_x^f(P, X_1, X_2) \equiv \begin{cases} \|P\|L(S, S'), & \text{if } S \neq \Omega, \|P\|L_1(S) = \text{true} \text{ and } \|X_1\|L(S, T) = (S', \mathcal{E}') \\ \|P\|L(S, S'), & \text{if } S \neq \Omega, \|P\|L_1(S) = \text{false} \text{ and } \|X_1\|L(S, T) = (S', \mathcal{E}') \\ \|P\|L(S, S'), & \text{if } S = \Omega, T \in \tau, \|X_1\|L(\Omega, T) = (S', \mathcal{E}') \\ \text{fail}, & \text{otherwise} \\ \end{cases}
\]

### 7.2.5 Mapping transformation

The revision of \( \text{xmap} \) mainly focuses on the \( \text{split} \) operator. The operator \( \text{split} \) is used by \( \text{xmap} \) to divide their views into subsequences, and then each subsequence is transformed backward to update the corresponding string or element in the source data or insert a new one. When a view does not include inserted values, it can be divided precisely according to the expected length for each subsequence computed from the original source data. However, if the view includes inserted values, the length information is not enough to determine how to split the view.

The following example shows that a more flexible splitting mechanism is needed for \( \text{xmap} \) if its views include insertions. For example, for the source BookList, the transformation \( \text{xmap xchild} \) produces a view consisting of a sequence of titles and authors of each book. Then, consider the following updated view.

\[
\text{<title}^{\text{orig.s}}[^{[\text{a}]}^{\text{ins.s}}], \text{<author}^{\text{orig.s}}[^{[\text{b}]}^{\text{ins.s}}], \text{<author}^{\text{ins.s}}[^{[\text{c}]}^{\text{ins.s}}], \text{<author}^{\text{ins.s}}[^{[\text{d}]}^{\text{ins.s}}], \text{<title}^{\text{ins.s}}[^{[\text{e}]}^{\text{ins.s}}], \text{<title}^{\text{ins.s}}[^{[\text{f}]}^{\text{ins.s}}]
\]

where three authors and two titles are inserted. In the backward execution of \( \text{xmap xchild} \), this view is first divided into subsequences and then each of them is used as the updated view of \( \text{xchild} \) to generate an updated or a new book element. However, since the view length is changed, the lengths of the original subsequences do not provide enough information for dividing the updated view. The view type annotated on \( \text{xmap} \) will provide extra information for the split operator, which needs to determine how many subsequences will be generated and how long a subsequence should be.

The new \text{split} operator is given in Figure 25. It has three arguments: the updated view to be split, a list of integers for the expected length of each subsequence, and the view type of \( \text{xmap} \). This operator returns a list, each item in which is a pair of a subsequence and a flag \( m \) or \( e \) indicating whether the corresponding source of this subsequence is missing or existing, respectively. Two list can be contacted by using \( ++ \).

The following are two examples explaining the new \text{split} operator. Let \text{Title} and \text{Author} be the types for title and author elements as in the source BookList. The first example has a successful split operation, while the second has a failed one.

The first example uses the above transformation and the updated view denoted as \( T \). In this example, the source type for BookList is specified as \( \text{<book}^{\text{2}}[^{[\text{Title,Author}]}^{*}] \), and hence \( T \) has the type \( \text{[Title,Author]*} \). The first book element in BookList generates two elements in \( T \), and the second generates three, so \( \text{split}(T,[2,3],\text{[Title,Author]*}) \) is used to split the updated view \( T \). The result are four subsequences consisting of the first three elements, the fourth and fifth elements, the next four elements and the last one, respectively. The first and the third subsequences are flagged by \( e \) and the second and the fourth are flagged by \( m \). Thus, after the source BookList is updated, the first book element is inserted with a new author, the second book element is a newly inserted one with the inserted title and author, the third book element has an inserted author and the fourth book element is also a newly inserted one with only the inserted title.

In the second example, suppose we have the source type \( \text{Title,Author}* \). Then, for the code \( \text{xmap (xif (xwithtag title xid xconst ()))} \), the annotated view type will be \( \text{[Title,]}[^{[\text{()}}^{*}] \). Let the source for this example be \( () \). Then, the original view is also \( () \). If we change the view into \( \text{<title}^{\text{ins.s}}[^{[\text{k}]}^{\text{ins.s}}] \), then the splitting...
For a subsequence, if its source data is missing, then in the execution of $xmap$ well-typed source data if being transformed backward successfully. The revised backward semantics of $xmap$ means some source data is hidden from the view, so for the type $()$ the split operator does not allow a subsequence () with the flag $m$. This is to make sure that the inserted data contains enough information to build a well-typed source data if being transformed backward successfully.

The revised backward semantics of $xmap$ is given below, where the new $iter$ operator is defined in Figure 26. For a subsequence, if its source data is missing, then in the $iter$ operator its source is replaced by $\Omega$ at the backward execution of $X$.

$$\text{split}([],[],[]) = []$$
$$\text{split}(T,[l],[\tau]) = [(T,e)], \text{if } T \in \tau \text{, oril}(T) = l \text{ and allins}(T) = false$$
$$\text{split}(T,[l],[\tau]) = [(T,n)], \text{if } T \in \tau \text{ and allins}(T) = true$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$
$$\text{split}(T,[l],\tau) = \text{split}(T,[l],[\tau])$$

Fig. 25 The split operator.

$$\text{iter}(X, [], S, S', E) = (S', E)$$
$$\text{iter}(X, (T, a) : ls, S, S', E) = \text{iter}(X, ls, S, S', E'), \text{where } [X]_E(\Omega, T) = (v', E')$$
$$\text{iter}(X, (T, a) : ls, S, S', E) = \text{iter}(X, ls, S, S', E'), \text{where } [X]_E(v, T) = (v', E')$$

Fig. 26 The iter operator.

The following lemma says after wrapping each subsequence from a splitting operation with a box, their concatenation has the type used by this splitting operation. That is, each boxed subsequence belongs to a boxed type in the type used by this splitting operation. Together with Lemma 4, this lemma will help prove that the revised backward semantics of $xmap$ satisfies the backward type preservation property.

**Lemma 7** Given a transformation $xmap'$ $X$ and a sequence $T$, if $\text{split}(T, ls, \tau) = [(T_1, k_1), ..., (T_n, k_n)]$, where $k_i \in \{m, e\}$, then $[T_1], ..., [T_n] \in \tau$. 
### 7.2.6 Parallel composition

The backward semantics of $X_1\parallel [\tau_1] X_2$ is revised on how to split the updated views into two subsequences. The updated views may include insertions, so the splitting cannot be completed only based on the lengths of two original subviews. As in the revised semantics of `xmap`, the type annotations $\tau_1$ and $\tau_2$ provide extra information for splitting, which are the view types of $X_1$ and $X_2$, respectively. Since this transformation does not update its source $S$, we require $S \neq \Omega$, such that it never returns $\Omega$ as the updated source.

$$\llbracket [\tau_1] X_1, [\tau_2] X_2 \rrbracket_{\mathcal{E}}(S, T) = \begin{cases} (S', E'), & \text{if } S \neq \Omega \\ \text{fail}, & \text{otherwise} \end{cases}$$

where $T = T_1, T_2; T_i \in \tau_i$, orilen($T_i$) = len($\llbracket X_i \rrbracket_{\mathcal{E}}(S))$

$$(S', E') = \llbracket X_1 \rrbracket_{\mathcal{E}}(S, T_2), (S, E) = \llbracket X_2 \rrbracket_{\mathcal{E}}(S, T_1)$$

### 7.2.7 Variable binding and element construction

If the source is $\Omega$, `xlet` will bind its variable to $(\Omega, \Omega)$ in the evaluation environment. However, the second $\Omega$ may not be updated by the component transformation $X$ of `xlet`. The backward semantics of `xlet` is only revised to make sure the updated source $S'$ is not $\Omega$. For example, in the backward transformation `xlet` $\xi xconst()$, the source of `xlet` is not updated by the component transformation `xconst`(). The `clean` operator also needs to be extended a little. That is, `clean(\Omega, S') = S'$.

$$\llbracket \text{`xlet Var} \ X \rrbracket_{\mathcal{E}}(S, T) = \begin{cases} (S', E'), & \text{if } \text{Sinlet} \in \text{Dom}(\mathcal{E}) \text{ and } S' \neq \Omega \\ (\text{clean}(S, S'), E'), & \text{if } \text{Sinlet} \notin \text{Dom}(\mathcal{E}) \text{ and } S' \neq \Omega \\ \text{fail}, & \text{otherwise} \end{cases}$$

where $(((), E'), \text{Var} \mapsto (S, S')) = \llbracket X \rrbracket_{\mathcal{E}}((((), T) \text{ and } E'' = \mathcal{E}. \text{Var} \mapsto (S, S)), \text{if } \text{Sinlet} \in \text{Dom}(\mathcal{E})$

$$((), E', \text{Var} \mapsto (S, S'), \text{Sinlet} \mapsto (1, 1)) = \llbracket X \rrbracket_{\mathcal{E}}((((), T) \text{ and } E'' = \mathcal{E}. \text{Var} \mapsto (S, S), \text{Sinlet} \mapsto (1, 1), \text{if } \text{Sinlet} \notin \text{Dom}(\mathcal{E})$$

If the source is $\Omega$, `xsetcnt` needs to use $\Omega$ as the source for the backward transformation of its component transformation. Since the tag of the updated element is derived from the updated view, `xsetcnt` does not need type annotations as `xchild`.

$$\llbracket \text{`xsetcnt} \ X \rrbracket_{\mathcal{E}}(S, T) = \begin{cases} (<\text{tag}^{(w.o.)}[S'], E'), & \text{if } S = <\text{tag}^{(w.o.)}[S'], T = <\text{tag}^{(w.o.)}[T'] \text{ and } S''', E'' = \llbracket X \rrbracket_{\mathcal{E}}(S', T) \\ (<\text{tag}^{(w.o.)}[S'], E'), & \text{if } S = \Omega, T = <\text{tag}^{(w.o.)}[T'] \text{ and } S''', E'' = \llbracket X \rrbracket_{\mathcal{E}}(\Omega, T') \\ \text{fail}, & \text{otherwise} \end{cases}$$

### 7.3 Revised update-keeping relation

The update-keeping relation in Figure 11 only relates two views of the same length. However, the updated view includes insertions, its length may be different from the new view from the updated source. Figure 27 lists six rules to extend the update-keeping relation. The first and last rules allow two sequences with different lengths to be related. Other rules consider the relation between inserted values and non-inserted values. The $(\text{ins}, s)$ tag on strings might be changed into $(\text{mod}, s)$ or $(\text{ori}(\cdot), s)$, as shown by the second example in Section 7.2.3. Similarly, the $(\text{ins}, s)$ tag on elements is changed into $(\text{ori}, s)$ when they are merged with non-inserted elements.

For the values tagged with $(\text{ins}, c)$, they cannot be reflected back into the source by backward transformations since they originate from code. Thus, a subsequent forward transformation based on the updated source changes their tags back to $(\text{ori}, c)$ or $(\text{ori}(\cdot), c)$. The following example explains the tag change and the unmatched length between an updated view and the new view from the updated source. Suppose the variable `$src` in the environment is initially bound to () and has the type `string[*]`.

```
(xvar $src; xmap xlet $s (<item [ori(c)]>()[i]}; xsetcnt xvar $s))((xvar $src; xmap xid).
```

Then, the above transformation produces the view $(),$ which has the type `<item [ori(c)]>string[*].string[*]`. If we change the view into `<item [ins(c)>][hello[ins.s]]$, then after backward transformation the variable `$src` is updated to
hello(\texttt{ins.s}). Based on the updated environment, executing the transformation forward again generates the new view \texttt{\langle item[rev1,c]\rangle[\langle hello[rev1.s]\rangle, hello[rev1.s]]}. We can see the updated view and the new view have different length, and the updating status on the item tag, which originates from code, changes from \texttt{ins} into \texttt{ori}.

### 7.4 An example for insertion

With the transformation toc annotated with types in Section 6.4, we demonstrate how view insertions are reflected back into source. The example is explained in the Table 7 and Table 8. The data and environments used by the example are shown in Figure 28. The value $T'$ is the updated view, in which $S'_4$ is a section newly inserted and $S'_8$, which is generated from $S'_6$. The following are the results of \texttt{split} operations used by \texttt{xmap} at b1.1, b1.1.3.2.1.1.1, b1.1.3.2.1.1.2.1.1 and b1.1.3.2.1.1.2.1.3.2.1.1.1.
8 Implementation

The approach proposed in this work has been implemented in Java with JDOM. The Galax XQuery engine [19] is used to normalize XQuery expressions to generate XQuery core code. Our implementation supports more XQuery input supported in our implementation. To support XML data input and output, we implemented the constructs and the attribute axis, XML name spaces, and the constructors for constructing and destructing sequences (or lists) are become.

Our implementation also supported the definition of libraries to contain commonly used bidirectional functions. The approach proposed in this work has been implemented in Java with JDOM. The Galax XQuery engine [19] is

Note that the split operator is not deterministic. For example, the third split below may also return

((i, e), (S’), (m), (i, e)). If this splitting result is taken, then the updated source of xmap at b1.3.2.1.1.1.1 will become S4, S’’ and S5, rather than S4, S5, S’’. However, both S4, S5, S’’ and S4, S’’, S5 belong to the source type TitleTy, SCntTy of this xmap.

8 Implementation

The approach proposed in this work has been implemented in Java with JDOM. The Galax XQuery engine [19] is used to normalize XQuery expressions to generate XQuery core code. Our implementation supports more XQuery input supported in our implementation. To support XML data input and output, we implemented the constructs and the attribute axis, XML name spaces, and the constructors for constructing and destructing sequences (or lists) are become.

Our implementation also supported the definition of libraries to contain commonly used bidirectional functions. The approach proposed in this work has been implemented in Java with JDOM. The Galax XQuery engine [19] is
implementation can simulate higher-order functions in functional languages by changing the argument \textit{fname} in \texttt{xfunapp} from a string to a transformation, and therefore a function argument can also be used as a function name. This feature is useful when we use this bidirectional language to interpret HaXML [24], which contains some higher-order XML transformation combinators.

In this implementation, only inserted or deleted elements need to be marked with \texttt{ins} or \texttt{del}, and other flags are derived by the system automatically. For example, by typing the updated view against the view type, we can obtain the origin annotations of strings or elements from their types, and by comparing the updated view with the original view, we can know whether a string is modified or not. This prototype implementation is not used to benchmark the performance of our approach since the implementation itself can be improved and the code generated from XQuery Core has much space to optimize. Our approach does not allow any change to the values generated by \texttt{xconst} or aggregate functions, such as \texttt{sum} and \texttt{count}. We reviewed the first forty-one XQuery use cases in [15]. Only six of

\begin{equation}
S = \langle \text{book} \rangle[S_1, S_2, S_3] \quad S' = \langle \text{book} \rangle[S_1', S_2, S_3']
\end{equation}

\begin{equation}
S_1 = \langle \text{title} \rangle[\text{Data on the Web}] \quad S_1' = \langle \text{title} \rangle[\text{Data on the Web}]
\end{equation}

\begin{equation}
S_2 = \langle \text{author} \rangle[\text{Sergio}] \quad S_2' = \langle \text{author} \rangle[\text{Sergio}]
\end{equation}

\begin{equation}
S_3 = \langle \text{section} \rangle[S_4, S_5] \quad S_3' = \langle \text{section} \rangle[S_4, S_5]
\end{equation}

\begin{equation}
S_4 = \langle \text{title} \rangle[\text{Introduction}] \quad S_4' = \langle \text{title} \rangle[\text{Introduction}]
\end{equation}

\begin{equation}
S_5 = \langle \text{p} \rangle[\text{Text}] \quad S_5' = \langle \text{p} \rangle[\text{Text}]
\end{equation}

\begin{equation}
S_6 = \langle \text{section} \rangle[] \quad S_6' = \langle \text{section} \rangle[]
\end{equation}

\begin{equation}
S_7 = \langle \text{section} \rangle[\langle\text{Introduction}\rangle] \quad S_7' = \langle \text{section} \rangle[\langle\text{Introduction}\rangle]
\end{equation}

\begin{equation}
C_0 \quad C_1 \quad C_2
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{book-or-section} & S & S' \\
\hline
S_1 & S_1' & S_1
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{section} & S_3 & S_3'
\hline
S_3 & S_3' & S_3
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{book-or-section} & S_4 & S_4'
\hline
S_4 & S_4' & S_4
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{section} & S_5 & S_5'
\hline
S_5 & S_5' & S_5
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{book-or-section} & S_6 & S_6'
\hline
S_6 & S_6' & S_6
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{section} & S_7 & S_7'
\hline
S_7 & S_7' & S_7
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{book-or-section} & S_1' & S_1'
\hline
S_1' & S_1' & S_1'
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{section} & S_3' & S_3'
\hline
S_3' & S_3' & S_3'
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{book-or-section} & S_4' & S_4'
\hline
S_4' & S_4' & S_4'
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{section} & S_5' & S_5'
\hline
S_5' & S_5' & S_5'
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{book-or-section} & S_6' & S_6'
\hline
S_6' & S_6' & S_6'
\end{array}
\end{equation}

\begin{equation}
\begin{array}{c|c|c}
\text{section} & S_7' & S_7'
\hline
S_7' & S_7' & S_7'
\end{array}
\end{equation}

\begin{equation}
S_1 = \langle \text{section} \rangle[S_4, S_5, S_6'] \quad S_1' = \langle \text{section} \rangle[S_4, S_5, S_6']
\end{equation}

\begin{equation}
S_2' = \langle \text{section} \rangle[S_4', S_5, S_6'] \quad S_3' = \langle \text{section} \rangle[S_4', S_5, S_6']
\end{equation}

\begin{equation}
S_4' = \langle \text{section} \rangle[S_4', S_5, S_6'] \quad S_5' = \langle \text{section} \rangle[S_4', S_5, S_6']
\end{equation}

\begin{equation}
S_6' = \langle \text{section} \rangle[S_4', S_5, S_6'] \quad S_7' = \langle \text{section} \rangle[S_4', S_5, S_6']
\end{equation}

\begin{equation}
T' = \langle \text{section} \rangle[S_4', S_6']
\end{equation}
them generate views completely consisting of values from xconst or aggregate functions and hence not allowing any update. For other use cases, our approach is found useful by enabling view update of XQuery.

9 Related work

The related work can be described from two aspects. The first is related to the bidirectional language design, and the second is about XML view update.

The bidirectional languages in [5]–[9], [12], [14] cannot be used directly to interpret XQuery for the following reasons. First, they do not have the variable binding mechanism, and consequently the output of a transformation can only be used by its successive transformations or the transformation combinators containing it. However, in XQuery, an output from an expression may be bound to a variable, and then used many times by different subexpressions. Though some languages such as those in [5], [14] have constructs to replicate a piece of source data into the view, these constructs are not suitable to support the interpretation of XQuery variables. This is because 1) the replicated parts do not have names, so it is hard to manage the scope of these replicas if there are many replicas, 2) replicating a value into the intermediate data for possible later reference will produce a value that is not expected by high-level XQuery expressions and hence not expected by (or not type correct for) the bidirectional code translated from the high-level XQuery expressions. Second, these languages do not provide a general setting to interpret functions in XQuery. A function in XQuery can have any number of arguments, each of which may be used as the updatable source data. However, these languages support only functions with one argument as the updatable source data. Third, the constructs in these languages are designed for their particular purposes and are not suitable for processing XML. For example, XPath axes are difficult to interpret in these languages and the condition lenses in [5] require more parameters than the conditional expression in XQuery can provide. In addition, the languages in [5]–[9], [12], [14] do not take expressive types for XML in their type systems. That is, their types cannot be directly used to interpret the XML schema languages used by XQuery such as Document Type Definition (DTD).

The injective language in [25] and the reversible language in [26] can also be executed in bidirectional ways. These languages express only injective functions, so their programs can be inverted. Since XQuery can express non-injective functions, the target languages for interpreting XQuery should be able to support non-injective functions. The work [27] proposes a method that given a function, it can automatically derives the backward function, so bidirectional transformation can be implemented without defining the backward semantics for each language construct. However, the language in [27] is simple in that a bound variable can only be used once and a function call can appear only in data constructors. Due to these restrictions, it is difficult to use this method to interpret XQuery. All these languages do not support recursive regular expression types for XML, either.

The bidirectional languages in [10], [11] are designed for synchronizing software models, not suitable for interpreting XQuery. In addition to the expressive type system for XML, the main difference is that our language allows the backward semantics of variable referencing construct to accumulate updates from its different replicas in views and the conditions in a conditional transformation can be preserved by other transformations which are even outside the scope of this conditional transformation (i.e., not contained within the two branches of the conditional transformation). These two features are necessary for updating XQuery views as explained by the examples in this paper.

The work [28], [29] studies how to update the relational database through XML views, rather than update XML data like our work. This work uses query trees to capture common operations in most XML query languages. However, query trees cannot support recursive functions in XQuery, as shown by our motivating example. The work [30] studies the conditions under which the updates to XML views can be translated into the underlying databases. In our approach, we use dynamic check to determine whether an updated view leads to valid updated source data. For example, the transformations xch1ld and xmap in our bidirectional language perform dynamic type checks to make sure the updated source is well-typed.

The work [31] addresses the problem of updating XML source through XML views. However, only views that remove parts of the document and rename nodes are supported. In [32], the programming language technique is also used to solve the view updating problem. But the view definition language in [32] is not bidirectional, so when defining a view, users have to write the code for putting back possible updates into the source XML data. XML views can also be defined by using XPath. The work [33] deals with the problem of updating XML views defined by XPath, which is not as expressive as XQuery.

XML views may not be materialized. The work [34] addresses the problem of updating XML views by translating updates to virtual views into updates to the source data according to the view definitions. The translation supports only the for-where-return XQuery expressions. That is, XQuery functions as used in our motivating example cannot be permitted as view definitions. The precise condition in [34] does not allow a translation to produce view side-
effects. Thus, the updates to a replicated data may not be translated successfully.

The work [35] describes a tracing mechanism of translating updates to XQuery views into SQL updates to relational databases. Since only atomic data on views can come from the relational databases, the mechanism in [35] only traces the lineage of atomic values. Their lineage tracing mechanism is not suitable for our view updating problem since our backward semantics involves the reconstruction of elements which should appear in the source data, but not in the views. However, the lineage information can be used to optimize the implementation of our bidirectional language by avoiding a complete reconstruction of the updated source data. An update translation scheme is also reported in [36], and since its source data also comes from relational databases, the scheme does not consider the reconstruction of elements, which is not easy as shown in our work particularly for updated views with insertions.

10 Conclusion

In this paper, we have proposed an expressive bidirectional XML transformation language and applied it to address the view updating problem of XQuery. The translation from XQuery to the bidirectional language is developed. The bidirectional language allows modifications, deletions and insertions on views. It is difficult to update view insertions and we proposed a type-based solution to this problem. The types of transformations are used to guide the backward executions, such that inserted values can be put back into the source in a reasonable way. A type system is designed for the bidirectional language to automatically annotate accurate types on the transformation, making the type system and language easy to use.

We proposed originally the extended round-tripping property by introducing the update-keeping relation to relate two updated views. The extended round-tripping property gives us the flexibility to describe the desired view-updating property of the expressive bidirectional language. When designing the language features, we have considered carefully to ensure each language feature does not violate the expected properties and explained the design rationales with many examples. However, due to the complexity of this expressive bidirectional language, we have not completely proved the theorems stated in this paper. It will be our future work to give the complete rigorous proofs.

Although we are motivated by interpreting XQuery, we believe that our work provides a technique to define general-purpose bidirectional functional languages since the bidirectional semantics of functions and constructors for algebraic data types can be defined by following the technique in this paper. It is interesting to analyze bidirectional programs and determine what are valid updates on views, such that valid updates do not lead to failure during backward executions.

Acknowledgment

We are grateful to the anonymous reviewers for their very helpful and detailed comments.

References

An expressive bidirectional transformation language for XQuery view update


Appendix A. Round-tripping property for relational view updates

The restrictions of round-tripping property discussed in Section 4.2.1 also hold on relational view updates. In this section, we use SQL examples to illustrate these restrictions. These examples have been executed on the major database management systems Microsoft SQL Server 2003, Oracle DB 10g and MySQL 5.0. That is, the round-tripping property is not adopted in these major systems.

Suppose there is a database table Persons defined in Table A.1 that contains all staff information for a company. The following SQL statement defines a view MarketMem that contains only the staff information from the market department. This view needs each staff to have both daytime and night phone numbers, which are actually generated from the same phone number in the underlying table.

```sql
create view MarketMem as
    select ID AS ID, Name AS Name, Phone AS DPhone, Phone AS NPhone
    from Persons
    where (Dept = 'Market')
```

By using the SQL statement `select * from MarketMem`, the MarketMem view is materialized as shown in Table A.2. This view is then updated by the following statement, which changes the daytime phone number of Tom into 9805.

```sql
update Members set DTel = 9805 where (Name = 'Tom')
```

After the above update, we materialize the view MarketMem again, which is shown in Table A.3. On this view, both the daytime and night phone numbers are changed into 9805, though the above update changes only the daytime number. That is, the view updated with the above statement is different from the newly materialized view, violating the round-tripping property. However, the extended round-tripping property is still valid for this example if tuples are treated as sequences and values are annotated with ori(•) or mod based on their updating status and the origins.

Table A.1 The persons table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Dept</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom</td>
<td>Market</td>
<td>9801</td>
</tr>
<tr>
<td>2</td>
<td>Peter</td>
<td>Product</td>
<td>9802</td>
</tr>
<tr>
<td>3</td>
<td>Kevin</td>
<td>Market</td>
<td>9803</td>
</tr>
</tbody>
</table>

Table A.2 The view MarketMem based on the persons table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>DTel</th>
<th>NTel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom</td>
<td>9801</td>
<td>9801</td>
</tr>
<tr>
<td>3</td>
<td>Kevin</td>
<td>9803</td>
<td>9803</td>
</tr>
</tbody>
</table>


Table A.3  The view MarketMem based on the updated persons table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>DTel</th>
<th>NTel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom</td>
<td>9805</td>
<td>9805</td>
</tr>
<tr>
<td>3</td>
<td>Kevin</td>
<td>9803</td>
<td>9803</td>
</tr>
</tbody>
</table>

Dongxi Liu
Dongxi Liu is a research scientist in ICT Centre, the Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia. Before joining CSIRO, he was a researcher in University of Tokyo. At that time, he worked on the design and development of bidirectional XML transformation languages. He got his BS and MS from Taiyuan University of Technology in 1996 and 1999, respectively. His PhD degree is from Shanghai Jiao Tong University in 2003. His major research area includes computation over encrypted data (homomorphic encryption, order-preserving secure indexing and encrypted database query), programming languages and formal methods, information security and cloud computing.

Zhenjiang Hu
Zhenjiang Hu is Professor of National Institute of Informatics (NII) in Japan. He received his BS and MS from Shanghai Jiao Tong University in 1988 and 1991 respectively, and PhD degree from University of Tokyo in 1996. He was a lecture (1997-1999) and an associate professor (2000-2007) in University of Tokyo, before joining NII as a full professor in 2008. His main 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 now serving on the steering committees of ACM ICFP, APLAS and FLOPS, and is the academic committee chair of the NII Shonan Meetings.

Masato Takeichi
Masato Takeichi is Professor of National Institution for Academic Degrees and University Evaluation, Japan since April 2011. Before joining the Institution, he was Professor of the University of Tokyo from January 1993 to March 2011. During these years, he established the Graduate School of Information Science and Technology in April 2001, and he was in charge of the Dean of IST from 2004 to 2007. His research concerns Software Science and Mathematical Informatics, and recent papers are related to Functional Programming, Calculational Transformation, and Bidirectional Programming. He is currently engaged in evaluation of higher education and research on academic degrees. He has been a member of the Science Council Japan since 2003, and became Vice-President of the Council at the start of October 2011.