Special issue: Advanced Programming Techniques for Construction of Robust, General and Evolutionary Programs

Research Paper

An expressive bidirectional transformation language for XQuery view update

Dongxi LIU[1], Zhenjiang HU[2] and Masato TAKEICHI[3]
[1] *CSIRO ICT Centre*
[2] *National Institute of Informatics*
[3] *Department 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.

Received January 29, 2012; Revised September 14, 2012; Accepted December 12, 2012.
1) *dongxi.liu@csiro.au*, 2) *hu@nii.ac.jp*, 3) *takeichi@niad.ac.jp*
DOI: 10.2201/NiiPi.2013.10.6

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 $\overline{S_1, S_2}$ for clarity. We have $\overline{(), S} = S$ and $\overline{S, ()} = S$.

```
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.

```
<book>
  <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 ... </p>
    <section>
      <title>Audience</title><p>Text ... </p>
    </section>
  </section>
  <section id="syntax" difficulty="medium">
    <title>A Syntax For Data</title><p>Text ... </p>
  </section>
</book>
```

1) The Source Data

```
<toc>
  <section id="intro">
    <title>Introduction</title>
    <section><title>Audience</title></section>
  </section>
  <section id="syntax">
    <title>A Syntax For Data</title>
  </section>
</toc>
```

2) The View

Fig. 2 The XML source data.

$$
\begin{array}{lcl}
S, T & ::= & () \mid V \\
V & ::= & v, V \\
v & ::= & str^{(u,o)} \mid <tag^{(w,o)}>[S] \\
u & ::= & \texttt{ori}(i) \mid \texttt{mod} \mid \texttt{ins} \mid \texttt{del} \\
w & ::= & \texttt{ori} \mid \texttt{ins} \mid \texttt{del} \\
o & ::= & \texttt{s} \mid \texttt{c} \\
i & ::= & \star \mid \uparrow \mid \downarrow \mid \bullet
\end{array}
$$

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 `ins` (`del`) are used for inserted (deleted) strings or elements. A string can be modified and the annotation `mod` is for modified strings. Strings or elements in their original states are annotated by `ori`(i) or `ori`, respectively. The modification to a string with `ori`(i) is instructed by the modification indicator i. The indicator i can be $\star$, $\uparrow$, $\downarrow$ or $\bullet$, 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 `ori`$(\bullet)$ or `ins` for strings, or `ori` or `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 `del` is propagated from values on views back to their

$$
\begin{aligned}
X &::= \texttt{xid} \mid \texttt{xconst}\ T \mid \texttt{xvar}\ Var \mid \texttt{xchild} \mid \texttt{xsetcnt}\ X \\
&\quad \mid X_1; X_2 \mid X_1 \| X_2 \mid \texttt{xmap}\ X \mid \texttt{xif}\ P\ X_1\ X_2 \\
&\quad \mid \texttt{xlet}\ Var\ X \mid \texttt{xfunapp}\ \mathit{fname}\ [X_1, ..., X_n] \\
P &::= \texttt{xeq}\ X_1\ \ X_2 \mid \texttt{xgt}\ X_1\ \ X_2 \mid \texttt{xlt}\ X_1\ \ X_2 \mid \texttt{xwithtag}\ \mathit{str} \mid \texttt{xiselement} \\
G &::= \cdot \mid G, \texttt{fun}\ \mathit{fname}(Var_1, ..., Var_n) = X
\end{aligned}
$$

Fig. 4 Syntax.

origins (called provenance in [18]) in the source data. These values can then be removed by an independent procedure like such as the database trigger, which may take into account some application-specific constraints on the source data to determine what values should be really removed. For example, suppose a `title` element in the source data in Section 2 is annotated with `del`. Then it is reasonable to remove the whole `section` element containing this title if the schema of the source data asks a section must include a title.

3.2 Syntax of the language

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 `xvar` 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 $\mathcal{E}$, which maps variables to pairs of values. If in the environment $\mathcal{E}$ 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 $\overline{C_1, Var \mapsto S, C_2}$.

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

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

- The forward semantics: $[\![X]\!]_C(S) = T$, meaning that applying X to the source S generates the view T under the environment C.
- The backward semantics: $[\![X]\!]_{\mathcal{E}}(S, T') = (S', \mathcal{E}')$, meaning that under the environment $\mathcal{E}$, applying X to the updated target data T' and the original source data S generates the updated source data S' and a new environment $\mathcal{E}'$.

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`.

$$
\begin{aligned}
&\mathtt{mg}((),()) &&= () \\
&\mathtt{mg}(str^{(u,o)}, str^{(u,o)}) &&= str^{(u,o)} \\
&\mathtt{mg}(str^{(\mathtt{ori}(\star),\mathtt{s})}, str'^{(u,\mathtt{s})}) &&= str'^{(u,\mathtt{s})} \\
&\mathtt{mg}(str^{(\mathtt{ori}(\uparrow),\mathtt{s})}, str'^{(\mathtt{mod},\mathtt{s})}) &&= str'^{(\mathtt{mod},\mathtt{s})}, \text{if } str' > str \\
&\mathtt{mg}(str^{(\mathtt{ori}(\downarrow),\mathtt{s})}, str'^{(\mathtt{mod},\mathtt{s})}) &&= str'^{(\mathtt{mod},\mathtt{s})}, \text{if } str' < str \\
&\mathtt{mg}(str^{(\mathtt{ori}(\uparrow),\mathtt{s})}, str^{(\mathtt{ori}(\downarrow),\mathtt{s})}) &&= str^{(\mathtt{ori}(\bullet),\mathtt{s})} \\
&\mathtt{mg}(str^{(\mathtt{ori}(\bullet),\mathtt{s})}, str^{(\mathtt{ori}(i),\mathtt{s})}) &&= str^{(\mathtt{ori}(\bullet),\mathtt{s})} \\
&\mathtt{mg}(str^{(\mathtt{del},\mathtt{s})}, str^{(\mathtt{ori}(i),\mathtt{s})}) &&= str^{(\mathtt{del},\mathtt{s})} \\
&\mathtt{mg}(\texttt{<}tag^{(w,o)}\texttt{>}[S_1], \texttt{<}tag^{(w,o)}\texttt{>}[S_2]) &&= \texttt{<}tag^{(w,o)}\texttt{>}[\mathtt{mg}(S_1,S_2)] \\
&\mathtt{mg}(\texttt{<}tag^{(\mathtt{ori},\mathtt{s})}\texttt{>}[S_1], \texttt{<}tag^{(\mathtt{del},\mathtt{s})}\texttt{>}[S_2]) &&= \texttt{<}tag^{(\mathtt{del},\mathtt{s})}\texttt{>}[\mathtt{mg}(S_1,S_2)] \\
&\mathtt{mg}(\overline{v_1,S_1}, \overline{v_2,S_2}) &&= \mathtt{mg}(v_1,v_2), \mathtt{mg}(S_1,S_2) \\
&\mathtt{mg}(S_1,S_2) &&= \mathtt{mg}(S_2,S_1), \text{if one case above applies to } \mathtt{mg}(S_2,S_1) \\
&\mathtt{mg}(S_1,S_2) &&= \mathtt{fail}, \text{if no other case applies}
\end{aligned}
$$

Fig. 5 The `mg` operator.

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 [5] except for the evaluation environments.

$$
\begin{aligned}
[\![\mathtt{xid}]\!]_{C}(S) &= S \\
[\![\mathtt{xid}]\!]_{\mathcal{E}}(S,T) &= (T,\mathcal{E})
\end{aligned}
$$

Constant transformation: This transformation returns its argument T_c for any source data in the forward execution. T_c must be (), $str^{(\mathrm{ori}(\bullet),\mathrm{c})}$ or $<tag^{(\mathrm{ori},\mathrm{c})}>[()]$. Together with other constructs, more complex constant views can be generated. In the backward direction, since the target data T_c 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.

$$
\begin{aligned}
[\![\mathtt{xconst}\ T_c]\!]_{C}(S) &= \begin{cases} T_c, & \text{if } T_c \in \{(), str^{(\mathrm{ori}(\bullet),\mathrm{c})}, <tag^{(\mathrm{ori},\mathrm{c})}>[()]\} \\ \mathtt{fail}, & \text{otherwise} \end{cases} \\
[\![\mathtt{xconst}\ T_c]\!]_{\mathcal{E}}(S,T) &= \begin{cases} (S,\mathcal{E}), & \text{if } T_c = T \\ \mathtt{fail}, & \text{otherwise} \end{cases}
\end{aligned}
$$

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 $\mathcal{E}$ is updated. In the new environment $\mathcal{E}'$, the `mg` operator, defined in Figure 5, is used to merge the updates within S_2 and T'. When merging two replicas of modified strings, the `mg` operator allows only one of them is modified or both of them are modified into the same value. In addition, the change to the modified string must be consist 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.

$$
\begin{aligned}
[\![\mathtt{xvar}\ Var]\!]_{C}(S) &= \begin{cases} T, & \text{if } C = \overline{C_1, Var \mapsto T, C_2} \text{ and } Var \notin Dom(C_2) \\ \mathtt{fail}, & \text{otherwise} \end{cases} \\
[\![\mathtt{xvar}\ Var]\!]_{\mathcal{E}}(S,T') &= \begin{cases} (S,\mathcal{E}'), & \text{if } \mathcal{E} = \overline{\mathcal{E}_1, Var \mapsto (S_1,S_2), \mathcal{E}_2} \text{ and } Var \notin Dom(\mathcal{E}_2) \\ \mathtt{fail}, & \text{otherwise} \end{cases}
\end{aligned}
$$

where $\mathcal{E}' = \mathcal{E}_1, Var \mapsto (S_1, \mathtt{mg}(S_2,T')), \mathcal{E}_2$

For view updating of XQuery, it is possible that one source value has several replicas, which may contain different updates. The merging operator `mg` returns a new value that combines all updates within two replicas if there are no conflicting updates. For example, merging elements <title$^{(\texttt{ori},s)}$>[xquerie$^{(\texttt{ori}(\star),\texttt{s})}$] and <title$^{(\texttt{ori},\texttt{s})}$>[XQuery$^{(\texttt{mod},\texttt{s})}$] will generate the element <title$^{(\texttt{ori},\texttt{s})}$>[XQuery$^{(\texttt{mod},\texttt{s})}$], while merging <price$^{(\texttt{ori},\texttt{s})}$>[30$^{(\texttt{mod},\texttt{s})}$] and <price$^{(\texttt{ori},\texttt{s})}$>[25$^{(\texttt{mod},\texttt{s})}$] 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.

$$
\begin{aligned}
[\![\texttt{xchild}]\!]_{\mathcal{C}}(S) &= \begin{cases} S', & \text{if } S = \textit{<tag}^{(w,o)}\textit{>}[S'] \\ \texttt{fail}, & \text{otherwise} \end{cases} \\
[\![\texttt{xchild}]\!]_{\mathcal{E}}(S,T) &= \begin{cases} (\textit{<tag}^{(w,o)}\textit{>}[T],\mathcal{E}), & \text{if } S = \textit{<tag}^{(w,o)}\textit{>}[S'] \\ \texttt{fail}, & \text{otherwise} \end{cases}
\end{aligned}
$$

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.

$$
\begin{aligned}
[\![\texttt{xsetcnt}\ X]\!]_{\mathcal{C}}(S) &= \begin{cases} \textit{<tag}^{(w,o)}\textit{>}[[\![X]\!]_{\mathcal{C}}(S')], & \text{if } S = \textit{<tag}^{(w,o)}\textit{>}[S'] \\ \texttt{fail}, & \text{otherwise} \end{cases} \\
[\![\texttt{xsetcnt}\ X]\!]_{\mathcal{E}}(S,T) &= \begin{cases} (\textit{<tag}^{(w',o)}\textit{>}[S''],\mathcal{E}'), & \text{if } S = \textit{<tag}^{(w,o)}\textit{>}[S'], T = \textit{<tag}^{(w',o)}\textit{>}[T'] \text{ and} \\ & \quad (S'',\mathcal{E}') = [\![X]\!]_{\mathcal{E}}(S',T') \\ \texttt{fail}, & \text{otherwise} \end{cases}
\end{aligned}
$$

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 $\mathcal{E}.1$ to generate the intermediate source data. For this purpose, the variables is bound to their original values by using the operator $\mathcal{E}.1$.

$$
\begin{aligned}
[\![X_1;X_2]\!]_{\mathcal{C}}(S) &= [\![X_2]\!]_{\mathcal{C}}([\![X_1]\!]_{\mathcal{C}}(S)) \\
[\![X_1;X_2]\!]_{\mathcal{E}}(S,T) &= [\![X_1]\!]_{\mathcal{E}'}(S,T'), \text{where } (T',\mathcal{E}') = [\![X_2]\!]_{\mathcal{E}}([\![X_1]\!]_{\mathcal{E}.1}(S),T)
\end{aligned}
$$

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 `len` returns the length of a sequence. With the updated subviews, the backward executions of X_2 and X_1 is performed in sequence, with the environment $\mathcal{E}''$ generated by X_2 fed into X_1 for producing the final updated environment $\mathcal{E}'$. 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.

$$
\begin{aligned}
[\![X_1 || X_2]\!]_{\mathcal{C}}(S) &= [\![X_1]\!]_{\mathcal{C}}(()), [\![X_2]\!]_{\mathcal{C}}(()) \\
[\![X_1 || X_2]\!]_{\mathcal{E}}(S,T) &= (S,\mathcal{E}')
\end{aligned}
$$

where

$$
\begin{aligned}
& T = \overline{T_1, T_2}, \texttt{len}(T_i) = \texttt{len}([\![X_i]\!]_{\mathcal{E}.1}(())) \ (i \in \{1,2\}) \\
& ((),\mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}((),T_2) \\
& ((),\mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}((),T_1)
\end{aligned}
$$

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

$$\begin{aligned}
&\texttt{split}((), []) &&= []\\
&\texttt{split}(T, l{:}ls) &&= ()\texttt{:split}(T, ls), \text{if } l = 0\\
&\texttt{split}(T, l{:}ls) &&= T_1\texttt{:split}(T_2, ls), \text{if } l > 0, T = \overline{T_1, T_2}, \text{and } \texttt{len}(T_1) = l\\
&\texttt{iter}(X, [], (), S', \mathcal{E}) &&= (S', \mathcal{E})\\
&\texttt{iter}(X, T{:}ls, \overline{v, S}, S', \mathcal{E}) &&= \texttt{iter}(X, ls, S, \overline{S', v'}, \mathcal{E}'), \text{where } [\![X]\!]_{\mathcal{E}}(v, T) = (v', \mathcal{E}')
\end{aligned}$$

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

$$\begin{aligned}
&\texttt{clean}(S, S) &&= S\\
&\texttt{clean}(str^{(\texttt{ori}(\star),\texttt{s})}, str^{(\texttt{ori}(i),\texttt{s})}) &&= str^{(\texttt{ori}(\star),\texttt{s})}, \text{if } i \in \{\uparrow, \downarrow, \bullet\}\\
&\texttt{clean}(str^{(\texttt{ori}(\uparrow),\texttt{s})}, str^{(\texttt{ori}(\bullet),\texttt{s})}) &&= str^{(\texttt{ori}(\uparrow),\texttt{s})}\\
&\texttt{clean}(str^{(\texttt{ori}(\downarrow),\texttt{s})}, str^{(\texttt{ori}(\bullet),\texttt{s})}) &&= str^{(\texttt{ori}(\downarrow),\texttt{s})}\\
&\texttt{clean}(str^{(u,o)}, str'^{(u',o)}) &&= str'^{(u',o)}, \text{if } u \in \{\texttt{ori}(\bullet), \texttt{mod}, \texttt{del}\}\\
&\texttt{clean}(\texttt{<}tag^{(w,o)}\texttt{>}[S_1], \texttt{<}tag'^{(w',o)}\texttt{>}[S_2]) &&= \texttt{<}tag'^{(w',o')}\texttt{>}[\texttt{clean}(S_1, S_2)]\\
&\texttt{clean}(\overline{v_1, S_1}, \overline{v_2, S_2}) &&= \texttt{clean}(v_1, v_2), \texttt{clean}(S_1, S_2)\\
&\texttt{clean}(S_1, S_2) &&= \texttt{fail}, \text{if no other case applies}
\end{aligned}$$

Fig. 7 The `clean` operator.

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 $[l_1, ..., l_n]$, and divides T into a list of n subsequences T_i $(1 \leq i \leq n)$, where $\texttt{len}(T_i) = l_i$. For example, $\texttt{split}(\overline{v_1, v_2, v_3}, [2, 0, 1])$ generates three subsequences: $[\overline{v_1, v_2}, (), v_3]$. 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 `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.

$$\begin{aligned}
&[\![\texttt{xmap } X]\!]_C(()) &&= ()\\
&[\![\texttt{xmap } X]\!]_C(\overline{v_1, ..., v_n}) &&= [\![X]\!]_C(v_1), ..., [\![X]\!]_C(v_n)\\
&[\![\texttt{xmap } X]\!]_{\mathcal{E}}((), ()) &&= ((), \mathcal{E})\\
&[\![\texttt{xmap } X]\!]_{\mathcal{E}}(\overline{v_1, ..., v_n}, T) &&= \texttt{iter}(X, ST, S, (), \mathcal{E})\\
&\quad \text{where } ST = \texttt{split}(T, [\texttt{len}([\![X]\!]_{\mathcal{E}.1}(v_1)), ..., \texttt{len}([\![X]\!]_{\mathcal{E}.1}(v_n))])
\end{aligned}$$

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 $\mathcal{E}'$, 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 $\mathcal{E}'$ do not affect the validity of P. That is, if P is `true` (or `false`) under the environment $\mathcal{E}$ and the source data S, then P should still have the same value under $\mathcal{E}'$ and S' even if $\mathcal{E}'$ 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.

$$\begin{aligned}
[\![\texttt{xif } P\ X_1\ X_2]\!]_C(S) &= \begin{cases} [\![X_1]\!]_C(S), & \text{if } [\![P]\!]_C(S) = \texttt{true}\\ [\![X_2]\!]_C(S), & \text{if } [\![P]\!]_C(S) = \texttt{false}\end{cases}\\
[\![\texttt{xif } P\ X_1\ X_2]\!]_{\mathcal{E}}(S, T) &= \begin{cases} [\![P]\!]_{\mathcal{E}'}(S, S'), & \text{if } [\![P]\!]_{\mathcal{E}.1}(S) = \texttt{true} \text{ and } [\![X_1]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')\\ [\![P]\!]_{\mathcal{E}'}(S, S'), & \text{if } [\![P]\!]_{\mathcal{E}.1}(S) = \texttt{false} \text{ and } [\![X_2]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')\end{cases}
\end{aligned}$$

In the lens language [5], the conditional lens `cond` and its two special instances `ccond` and `acond` are defined. The lens `ccond` transforms backward an updated view with the component lens that generates the original view, while the lenses `acond` and `cond` allow an updated view to be reflected back by the component lens that is not the one generating the original view. For the lenses `acond` and `ccond`, 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 `let` and `for` expressions in XQuery.

$$[\![\texttt{xlet}\ Var\ X]\!]_{C}(S) \quad = \quad [\![X]\!]_{C'}(()), \text{ where } C' = \overline{C, Var \mapsto S}$$

$$[\![\texttt{xlet}\ Var\ X]\!]_{\mathcal{E}}(S,T) \quad = \quad \begin{cases} (S', \mathcal{E}'), \text{ if } \texttt{\$inlet} \in Dom(\mathcal{E}) \\ (\text{clean}(S,S'), \mathcal{E}'), \text{ otherwise} \end{cases}$$

$$\text{where } ((), \overline{\mathcal{E}', Var \mapsto (S,S')}) = [\![X]\!]_{\mathcal{E}''}((),T) \text{ and } \mathcal{E}'' = \overline{\mathcal{E}, Var \mapsto (S,S)}, \quad \text{if } \texttt{\$inlet} \in Dom(\mathcal{E})$$

$$((), \overline{\mathcal{E}', Var \mapsto (S,S'), \texttt{\$inlet} \mapsto (1,1)}) = [\![X]\!]_{\mathcal{E}''}((),T) \text{ and}$$

$$\mathcal{E}'' = \overline{\mathcal{E}, Var \mapsto (S,S), \texttt{\$inlet} \mapsto (1,1)}, \quad \text{otherwise}$$

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 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 `$inlet` is bound or not in the execution environment $\mathcal{E}$. A binding `$inlet` $\mapsto (1,1)$ in $\mathcal{E}$ is used to indicate the current `xlet` is enclosed by another `xlet` (the value (1,1) in the binding does not matter). If the current `xlet` is an enclosed one, its backward execution executes backward the transformation X under the environment $\overline{\mathcal{E}, Var \mapsto (S,S)}$, where the variable 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 $\overline{\mathcal{E}', Var \mapsto (S,S')}$, which contains the updated source S' and environment $\mathcal{E}'$.

If the current `xlet` is not enclosed by any other `xlet`, the backward execution of X is under an environment with the binding `$inlet` $\mapsto (1,1)$ pushed into $\mathcal{E}$ after the binding of Var ($Var \mapsto (S,S)$). In this case, the updated source S' needs to be cleaned by using the `clean` operator, which is defined in Figure 7. The purpose of `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 `xeq`, `xlt`, and `xgt` may change the modification indicator i 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 `clean`(S,S') changes a string in S' into the corresponding one in the original source S in the following cases: a string $str^{(\texttt{ori}(i),\text{s})}$ ($i \in \{\uparrow,\downarrow,\bullet\}$) in S' corresponds to a string $str^{(\texttt{ori}(\star),\text{s})}$ in S, or a string $str^{(\texttt{ori}(\bullet),\text{s})}$ in S' corresponds to a string $str^{(\texttt{ori}(i),\text{s})}$ ($i \in \{\uparrow,\downarrow\}$) in S. In these cases, the modification indicators are changed by predicates `xeq`, `xlt` and `xgt`.

Function call: Suppose the function *fname* is defined as

$$\texttt{fun}\ fname(Var_1, ..., Var_n) = X$$

Then, the semantics of applying the function *fname* to n arguments $X_1, ..., X_n$ is defined below with the previous constructs.

$$\texttt{xfunapp}\ fname\ [X_1, ..., X_n] \quad = \quad \texttt{xconst}\ (); X'_1$$

where

$$X'_1 = X_1; \texttt{xlet}\ Var_1\ X'_2$$
$$X'_2 = X_2; \texttt{xlet}\ Var_2\ X'_3$$
$$...$$
$$X'_n = X_n; \texttt{xlet}\ Var_n\ X$$

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

$$\begin{aligned}
\mathtt{guard}(str^{(\mathtt{ori}(\star),\mathtt{s})}, i) &= str^{(\mathtt{ori}(i),\mathtt{s})}, \text{if } i \in \{\uparrow,\downarrow,\bullet\} \\
\mathtt{guard}(str^{(\mathtt{ori}(\uparrow),\mathtt{s})}, i) &= str^{(\mathtt{ori}(\bullet),\mathtt{s})}, \text{if } i \in \{\downarrow,\bullet\} \\
\mathtt{guard}(str^{(\mathtt{ori}(\downarrow),\mathtt{s})}, i) &= str^{(\mathtt{ori}(\bullet),\mathtt{s})}, \text{if } i \in \{\uparrow,\bullet\} \\
\mathtt{guard}(str^{(u,o)}, i) &= str^{(u,o)}, \text{otherwise}
\end{aligned}$$

Fig. 8 The `guard` operator.

empty sequence () due to the definition of `xlet`. That is, the function body cannot directly use and update the source data of `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 `true` or `false`, which is used only by `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 `xif`, instead of the target data `true` or `false`.

Comparison: In the forward direction, the predicate `xeq` returns `true` if the views of X_1 and X_2 are the same string (probably with different annotations), or `false` otherwise. The comparison result is preserved by the backward semantics of this predicate. That is, if this predicate returns `true` (or `false`) with the original source and environment, then it still returns `true` (or `false`) with the updated ones. For this purpose, the backward semantics of predicate `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 $\mathtt{guard}(str^{(u,o)}, i)$ incorporates the modification indicator i into the string view $str^{(u,o)}$ if it is modifiable (i.e., if $u \in \{\mathtt{ori}(\star), \mathtt{ori}(\uparrow), \mathtt{ori}(\downarrow)\}$ and $o = \mathtt{s}$). For example, if i is $\uparrow$ and the string view is $str^{(\mathtt{ori}(\star),\mathtt{s})}$, then the result of applying `guard` is an updated string view $str^{(\mathtt{ori}(\uparrow),\mathtt{s})}$. 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 `xlt` and `xgt`, their forward semantics is defined similarly, and their backward semantics are the same as that of `xeq`, hence not given.

$$[\![\mathtt{xeq}\ X_1\ X_2]\!]_C(S) = \begin{cases} \mathtt{true}, & \text{if } [\![X_1]\!]_C(()) = str^{(u,o)} \text{ and } [\![X_2]\!]_C(()) = str^{(u',o')} \\ \mathtt{false}, & \text{if } [\![X_1]\!]_C(()) = str^{(u,o)}, [\![X_2]\!]_C(()) = str'^{(u',o')} \text{ and } str \neq str' \\ \text{fail}, & \text{otherwise} \end{cases}$$

$$[\![\mathtt{xeq}\ X_1\ X_2]\!]_{\mathcal{E}}(S, S') = \begin{cases} (S', \mathcal{E}'), & \text{if } [\![X_1]\!]_{\mathcal{E}.1}(()) = str^{(u,o)}, [\![X_2]\!]_{\mathcal{E}.1}(()) = str^{(u',o')}, \\ & \quad ((), \mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}((), \mathtt{guard}(str^{(u',o')}, \bullet)), \text{ and} \\ & \quad ((), \mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}((), \mathtt{guard}(str^{(u,o)}, \bullet)) \\ (S', \mathcal{E}'), & \text{if } [\![X_1]\!]_{\mathcal{E}.1}(()) = str^{(u,o)}, [\![X_2]\!]_{\mathcal{E}.1}(()) = str'^{(u',o')}, str > str', \\ & \quad ((), \mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}((), \mathtt{guard}(str'^{(u',o')}, \downarrow)), \text{ and} \\ & \quad ((), \mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}((), \mathtt{guard}(str^{(u,o)}, \uparrow)), \text{ and} \\ (S', \mathcal{E}'), & \text{if } [\![X_1]\!]_{\mathcal{E}.1}(()) = str^{(u,o)}, [\![X_2]\!]_{\mathcal{E}.1}(()) = str'^{(u',o')}, str < str', \\ & \quad ((), \mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}((), \mathtt{guard}(str'^{(u',o')}, \uparrow)), \text{ and} \\ & \quad ((), \mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}((), \mathtt{guard}(str^{(u,o)}, \downarrow)), \text{ and} \end{cases}$$

$$[\![\mathtt{xlt}\ X_1\ X_2]\!]_C(S) = \begin{cases} \mathtt{true}, & \text{if } [\![X_1]\!]_C(()) = str^{(u,o)}, [\![X_2]\!]_C(()) = str'^{(u',o')} \text{ and } str < str' \\ \mathtt{false}, & \text{if } [\![X_1]\!]_C(()) = str^{(u,o)}, [\![X_2]\!]_C(()) = str'^{(u',o')} \text{ and } str \not< str' \\ \text{fail}, & \text{otherwise} \end{cases}$$

$$[\![\mathtt{xgt}\ X_1\ X_2]\!]_C(S) = \begin{cases} \mathtt{true}, & \text{if } [\![X_1]\!]_C(()) = str^{(u,o)}, [\![X_2]\!]_C(()) = str'^{(u',o')} \text{ and } str > str' \\ \mathtt{false}, & \text{if } [\![X_1]\!]_C(()) = str^{(u,o)}, [\![X_2]\!]_C(()) = str'^{(u',o')} \text{ and } str \not> str' \\ \text{fail}, & \text{otherwise} \end{cases}$$

As an example, suppose we have the source data $\mathtt{10}^{(\mathtt{ori}(\star),\mathtt{s})}$, and the transformation `xlet` $\$d\ X_1 || X_2$, where

```
fun toc($book-or-section) = XBody
where
  XBody = xvar $book-or-section;    //gets the book or section
          xchild;    //gets the contents of the book or section
          xmap (xif (xwithtag section) XSec (xconst ()))  //processes each section
                                          //with XSec and hides non-section elements
  XSec = xlet $section (
           <section^(ori,c)>[()]   //builds a section element
           xsetcnt (XTitle||XSubTitles) //sets title and subsection titles
         )
  XTitle = xvar $section; //gets a section element
           xchild;   //gets its content
           xmap (xif (xwithtag title) xid (xconst ())) //keeps only its title
  XSubTitles = xfunapp toc [xvar $section] //builds toc of subsections
```

Fig. 9 The XQuery example in bidirectional language.

$X_1 = \texttt{xvar\$d;xid}$, $X_2 = \texttt{xvar\$d;xif}\ P\ \texttt{xid xconst}$ (), and $P = \texttt{xeq xconst}\ 10^{(\texttt{ori}(\bullet),\texttt{c})}\ \texttt{xvar \$d}$. After the forward transformation, the view is $\overline{\texttt{10}^{(\texttt{ori}(\star),\texttt{s})}, \texttt{10}^{(\texttt{ori}(\star),\texttt{s})}}$. For the backward transformation, the `xvar` in `xeq` needs to change the source into $\texttt{10}^{(\texttt{ori}(\bullet),\texttt{s})}$ for preserving the equivalence result of comparison. If we do not change the view, then the backward execution of `xlet` will clean this modification made by `xeq` and generate the original source. If we change the view into $\overline{\texttt{15}^{(\texttt{mod},\texttt{s})}, \texttt{10}^{(\texttt{ori}(\star),\texttt{s})}}$, $\overline{\texttt{10}^{(\texttt{ori}(\star),\texttt{s})}, \texttt{15}^{(\texttt{mod},\texttt{s})}}$, or $\overline{\texttt{15}^{(\texttt{mod},\texttt{s})}, \texttt{15}^{(\texttt{mod},\texttt{s})}}$, then the mg operator used in the backward execution of `xvar` will detect this modification conflict with the modification indicator in $\texttt{10}^{(\texttt{ori}(\bullet),\texttt{s})}$.

Still for the same transformation, if the source data is $\texttt{12}^{(\texttt{ori}(\star),\texttt{s})}$, then its forward execution generates the view $\texttt{12}^{(\texttt{ori}(\star),\texttt{s})}$, with the `xeq` predicate returning false. To preserve the comparison result, the `xvar` in `xeq` needs to change the source into $\texttt{12}^{(\texttt{ori}(\uparrow),\texttt{s})}$. Thus, the updated view $\texttt{15}^{(\texttt{mod},\texttt{s})}$ 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' and the updated environment $\mathcal{E}'$ directly since the tags of elements are not allowed to change and nothing is needed to guard.

$$\begin{array}{lcl}
[\![\texttt{xwithtag}\ str]\!]_{\mathcal{C}}(S) & = & \begin{cases} \texttt{true}, & \text{if } S = \textit{<tag}^{(w,o)}\textit{>}[S_1] \text{ and } tag = str \\ \texttt{false}, & \text{else if } S = \textit{<tag}^{(w,o)}\textit{>}[S_1] \text{ and } tag \neq str \\ \text{fail}, & \text{otherwise} \end{cases} \\
[\![\texttt{xwithtag}\ str]\!]_{\mathcal{E}}(S, S') & = & (S', \mathcal{E})
\end{array}$$

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' and the updated environment $\mathcal{E}'$ directly in its backward execution.

$$\begin{array}{lcl}
[\![\texttt{xiselement}]\!]_{\mathcal{C}}(S) & = & \begin{cases} \texttt{true}, & \text{if } S = \textit{<tag}^{(w,o)}\textit{>}[S_1] \\ \texttt{false}, & \text{else if } S = str^{(u,o)} \\ \text{fail}, & \text{otherwise} \end{cases} \\
[\![\texttt{xiselement}]\!]_{\mathcal{E}}(S, S') & = & (S', \mathcal{E})
\end{array}$$

3.6 Programming examples

A program of this bidirectional language is given in Figure 9. This program implements the recursive `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 `XSec` is used to construct the section element in the view with the help of `XTitle` and `XSubTitles`, which correspond to the expression $section/title and the recursive function call in the example query, respectively.

A forward execution of invoking the `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

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

Index	Tran	Src	Env	Tar
f1	XBody	()	C_0	<section$^{(\mathtt{ori,c})}$>[S_4]
f1.1	xvar $book-or-section	()	C_0	S
f1.2	xchild	S	C_0	S_1, S_2, S_3
f1.3	xmap (xif (xwithtag section) XSec (xconst ()))	$S_1, S_2,$ S_3	C_0	(),(), <section$^{(\mathtt{ori,c})}$>[S_4]
f1.3.1	xif (xwithtag section) XSec (xconst ()))	S_1	C_0	()
f1.3.1.1	$[\![$xwithtag section$]\!]_{C_0}(S_1)$ = false			
f1.3.1.2	xconst ()	S_1	C_0	()
f1.3.2	xif (xwithtag section) XSec (xconst ()))	S_2	C_0	()
f1.3.2.1	$[\![$xwithtag section$]\!]_{C_0}(S_2)$ = false			
f1.3.2.2	xconst ()	S_2	C_0	()
f1.3.3	xif (xwithtag section) XSec (xconst ()))	S_3	C_0	<section$^{(\mathtt{ori,c})}$>[S_4]
f1.3.3.1	$[\![$xwithtag section$]\!]_{C_0}(S_3)$ = true			
f1.3.3.2	XSec	S_3	C_0	<section$^{(\mathtt{ori,c})}$>[S_4]
f1.3.3.2.1	xconst <section$^{(\mathtt{ori,c})}$>[()]	()	C_1	<section$^{(\mathtt{ori,c})}$>[()]
f1.3.3.2.2	xsetcnt (XTitle||XSubTitles)	<section$^{(\mathtt{ori,c})}$> [()]	C_1	<section$^{(\mathtt{ori,c})}$>[S_4]
f1.3.3.2.2.1	XTitle||XSubTitles	()	C_1	S_4, ()
f1.3.3.2.2.1.1	XTitle	()	C_1	S_4
f1.3.3.2.2.1.1.1	xvar $section	()	C_1	S_3
f1.3.3.2.2.1.1.2	xchild	S_3	C_1	S_4, S_5
f1.3.3.2.2.1.1.3	xmap (xif (xwithtag title) xid (xconst ()))	S_4, S_5	C_1	S_4,()
f1.3.3.2.2.1.1.3.1	xif (xwithtag title) xid (xconst ())	S_4	C_1	S_4
f1.3.3.2.2.1.1.3.1.1	$[\![$xwithtag title$]\!]_{C_1}(S_4)$ = true			
f1.3.3.2.2.1.1.3.1.2	xid	S_4	C_1	S_4
f1.3.3.2.2.1.1.3.2	xif (xwithtag title) xid (xconst ())	S_5	C_1	()
f1.3.3.2.2.1.1.3.2.1	$[\![$xwithtag title$]\!]_{C_1}(S_5)$ = false			
f1.3.3.2.2.1.1.3.2.2	xconst ()	S_5	C_1	()

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

Index	Tran	Src	Env	Tar
f1.3.3.2.2.1.2	XSubTitles	()	C_1	()
f1.3.3.2.2.1.2.1	xvar $section	()	C_1	S_3
f1.3.3.2.2.1.2.2	xlet $book-or-section XBody	S_3	C_1	()
f1.3.3.2.2.1.2.2.1	xvar $book-or-section	()	C_2	S_3
f1.3.3.2.2.1.2.2.2	xchild	S_3	C_2	S_4, S_5
f1.3.3.2.2.1.2.2.3	xmap (xif (xwithtag section) XSec (xconst ()))	S_4, S_5	C_2	(),()
f1.3.3.2.2.1.2.2.3.1	xif (xwithtag section) XSec (xconst ()))	S_4	C_2	()
f1.3.3.2.2.1.2.2.3.1.1	$[\![$xwithtag title$]\!]_{C_2}(S_4)$ = false			
f1.3.3.2.2.1.2.2.3.1.2	xconst ()	S_4	C_2	()
f1.3.3.2.2.1.2.2.3.2	xif (xwithtag section) XSec (xconst ()))	S_5	C_2	()
f1.3.3.2.2.1.2.2.3.2.1	$[\![$xwithtag title$]\!]_{C_2}(S_5)$ = false			
f1.3.3.2.2.1.2.2.3.2.2	xconst ()	S_5	C_2	()

invocation starts with the source () and the environment C_0 and returns the view <section$^{(\mathtt{ori,c})}$>[S_4]. Suppose the view is changed into <section$^{(\mathtt{ori,c})}$>[S'_4]. Then, using this updated view and the source () as the input of the backward invocation under the environment $\mathcal{E}_0$, we will get the updated environment $\mathcal{E}'_0$, in which the updated data S' contains

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

Index	Tran	Src	UTar	Env	USrc	UEnv
b1	XBody	()	$<\text{section}^{(\texttt{ori,c})}>[S'_4]$	$\mathcal{E}_0$	()	$\mathcal{E}'_0$
b1.3	xvar $book-or-section	()	S'	$\mathcal{E}_0$	()	$\mathcal{E}'_0$
b1.2	xchild	S	S_1, S_2, S'_3	$\mathcal{E}_0$	S'	$\mathcal{E}_0$
b1.1	xmap (xif (xwithtag section) XSec (xconst ()))	S_1, S_2, S_3	(),(), $<\text{section}^{(\texttt{ori,c})}>[S'_4]$	$\mathcal{E}_0$	S_1, S_2, S'_3	$\mathcal{E}_0$
b1.1.1	xif (xwithtag section) XSec (xconst ()))	S_1	()	$\mathcal{E}_0$	S_1	$\mathcal{E}_0$
b1.1.1.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_0.1}(S_1) = \text{false}$					
b1.1.1.2	xconst ()	S_1	()	$\mathcal{E}_0$	S_1	$\mathcal{E}_0$
b1.1.2	xif (xwithtag section) XSec (xconst ()))	S_2	()	$\mathcal{E}_0$	S_2	$\mathcal{E}_0$
b1.1.2.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_0.1}(S_2) = \text{false}$					
b1.1.2.2	xconst ()	S_2	()	$\mathcal{E}_0$	S_2	$\mathcal{E}_0$
b1.1.3	xif (xwithtag section) XSec (xconst ()))	S_3	$<\text{section}^{(\texttt{ori,c})}>[S'_4]$	$\mathcal{E}_0$	S'_3	$\mathcal{E}_0$
b1.1.3.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_0.1}(S_3) = \text{true}$					
b1.1.3.2	XSec	S_3	$<\text{section}^{(\texttt{ori,c})}>[S'_4]$	$\mathcal{E}_0$	S'_3	$\mathcal{E}_0$
b1.1.3.2.2	xconst $<\text{section}^{(\texttt{ori,c})}>[()]$	()	$<\text{section}^{(\texttt{ori,c})}>[()]$	$\mathcal{E}'_1$	()	$\mathcal{E}'_1$
b1.1.3.2.1	xsetcnt (XTitle||XSubTitles)	S_c	$<\text{section}^{(\texttt{ori,c})}>[S'_4]$	$\mathcal{E}_1$	S_c	$\mathcal{E}'_1$
b1.1.3.2.1.1	XTitle||XSubTitles	()	S'_4, ()	$\mathcal{E}_1$	()	$\mathcal{E}'_1$
b1.1.3.2.1.1.1	XTitle	()	S'_4	$\mathcal{E}_1$	()	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.3	xvar $section	()	S'_3	$\mathcal{E}_1$	()	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.2	xchild	S_3	S'_4, S_5	$\mathcal{E}_1$	S'_3	$\mathcal{E}_1$
b1.1.3.2.1.1.1.1	xmap (xif (xwithtag title) xid (xconst ()))	S_4, S_5	S'_4,()	$\mathcal{E}_1$	S'_4, S_5	$\mathcal{E}_1$
b1.1.3.2.1.1.1.1.1	xif (xwithtag title) xid (xconst ())	S_4	S'_4	$\mathcal{E}_1$	S'_4	$\mathcal{E}_1$
b1.1.3.2.1.1.1.1.1.1	$[\![\text{xwithtag title}]\!]_{\mathcal{E}_1.1}(S_4) = \text{true}$					
b1.1.3.2.1.1.1.1.1.2	xid	S_4	S'_4	$\mathcal{E}_1$	S'_4	$\mathcal{E}_1$
b1.1.3.2.1.1.1.1.2	xif (xwithtag title) xid (xconst ())	S_5	()	$\mathcal{E}_1$	S_5	$\mathcal{E}_1$
b1.1.3.2.1.1.1.1.2.1	$[\![\text{xwithtag title}]\!]_{\mathcal{E}_1.1}(S_5) = \text{false}$					
b1.1.3.2.1.1.1.1.2.2	xconst ()	S_5	()	$\mathcal{E}_1$	S_5	$\mathcal{E}_1$

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

Index	Tran	Src	UTar	Env	USrc	UEnv
b1.1.3.2.1.1.2	XSubTitles	()	()	$\mathcal{E}_1$	()	$\mathcal{E}_1$
b1.1.3.2.1.1.2.2	xvar $section	()	S_3	$\mathcal{E}_1$	()	$\mathcal{E}_1$
b1.1.3.2.1.1.2.1	xlet $book-or-section XBody	S_3	()	$\mathcal{E}_1$	S_3	$\mathcal{E}_1$
b1.1.3.2.1.1.2.1.3	xvar $book-or-section	()	S_3	$\mathcal{E}_2$	()	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.2	xchild	S_3	S_4, S_5	$\mathcal{E}_2$	S_3	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1	xmap (xif (xwithtag section) XSec (xconst ()))	S_4, S_5	(),()	$\mathcal{E}_2$	S_4, S_5	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.1	xif (xwithtag section) XSec (xconst ()))	S_4	()	$\mathcal{E}_2$	S_4	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.1.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_2.1}(S_4) = \text{false}$					
b1.1.3.2.1.1.2.1.1.1.2	xconst ()	S_4	()	$\mathcal{E}_2$	S_4	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.2	xif (xwithtag section) XSec (xconst ()))	S_5	()	$\mathcal{E}_2$	S_5	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.2.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_2.1}(S_5) = \text{false}$					
b1.1.3.2.1.1.2.1.1.2.2	xconst ()	S_5	()	$\mathcal{E}_2$	S_5	$\mathcal{E}_2$

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

$S\ = \texttt{<}\text{book}^{(\texttt{ori},\texttt{s})}\texttt{>}[S_1, S_2, S_3]$

$S_1 = \texttt{<}\text{title}^{(\texttt{ori},\texttt{s})}\texttt{>}[\text{Data on the Web}^{(\texttt{ori}(\star),\texttt{s})}]$

$S_2 = \texttt{<}\text{author}^{(\texttt{ori},\texttt{s})}\texttt{>}[\text{Serge}^{(\texttt{ori}(\star),\texttt{s})}]$

$S_3 = \texttt{<}\text{section}^{(\texttt{ori},\texttt{s})}\texttt{>}[S_4, S_5]$

$S_4 = \texttt{<}\text{title}^{(\texttt{ori},\texttt{s})}\texttt{>}[\text{Introduction}^{(\texttt{ori}(\star),\texttt{s})}]$

$S_5 = \texttt{<}\text{p}^{(\texttt{ori},\texttt{s})}\texttt{>}[\text{Text}^{(\texttt{ori}(\star),\texttt{s})}]$

$S'_3 = \texttt{<}\text{section}^{(\texttt{ori},\texttt{s})}\texttt{>}[S'_4, S_5]$

$S'_4 = \texttt{<}\text{title}^{(\texttt{ori},\texttt{s})}\texttt{>}[\text{Background}^{(\texttt{mod},\texttt{s})}]$

$S'\ = \texttt{<}\text{book}^{(\texttt{ori},\texttt{s})}\texttt{>}[S_1, S_2, S'_3]$

$S_c = \texttt{<}\text{section}^{(\texttt{ori},\texttt{c})}\texttt{>}[()]$

\$book-or-section	S

$\mathcal{C}_0$

\$section	S_3
\$book-or-section	S

$\mathcal{C}_1$

\$book-or-section	S_3
\$section	S_3
\$book-or-section	S

$\mathcal{C}_2$

\$book-or-section	S	S

$\mathcal{E}_0$

\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}_1$

\$book-or-section	S_3	S_3
\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}_2$

\$book-or-section	S	S'

$\mathcal{E}'_0$

\$section	S_3	S'_3
\$book-or-section	S	S

$\mathcal{E}'_1$

Fig. 10 Data and environments for the XQuery example.

`f1.1`, `f1.2` and `f1.3`, and they are executed in sequence since the transformation `f1.1` is a sequence composition. As another example, the transformation `f1.3` depends on the transformations `f1.3.1`, `f1.3.2` and `f1.3.3`, and their results will be put together as the result of transformation `f1.3` since it is an `xmap` transformation. In our bidirectional language, backward executions always need forward executions to generate intermediary data. For example, the transformation `b1.1` needs the forward transformations `f1.1`, `f1.2` to generate its source data S_1, S_2, S_3.

The organization of executions in the tables with execution indexes provides a convenient way to trace the bidirectional transformations and the intermediate data. Each line in the tables contains all information needed to understand the transformation in that line, and moreover the transformation details can be found by following the indexes. For example, the line `f1.3.3` in Table 1 says under the context C_0 the conditional transformation generates the target $\texttt{<}\text{section}^{(\texttt{ori},\texttt{c})}\texttt{>}[S_4]$ from S_3, and if the details of this conditional transformation is wanted, then we can check the lines with the indexes `f1.3.3.1` and `f1.3.3.2`.

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 $[\![X]\!].(S, T) = (S, \cdot)$.

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 $[\![X]\!]_{\mathcal{E}}(S,T) = (S',\mathcal{E}')$ and $[\![X]\!]_{\mathcal{E}'.2}(S') = T'$, 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.

$$\begin{aligned}&\text{<bib}^{(\texttt{ori,s})}\text{>[<book}^{(\texttt{ori,s})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Database}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <price}^{(\texttt{ori,s})}\text{>[20}^{(\texttt{ori}(\star)\texttt{,s})}\text{]],}\\ &\qquad\text{<book}^{(\texttt{ori,s})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Network}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <price}^{(\texttt{ori,s})}\text{>[10}^{(\texttt{ori}(\star)\texttt{,s})}\text{]]]}\end{aligned}$$

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

$$\begin{aligned}&\text{BibView} = \texttt{xchild}; \texttt{xlet}\ \$\text{books}\ (\texttt{xconst}\ \text{<bibview}^{(\texttt{ori,c})}\text{>[()]};\\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{xsetcnt}\ (\text{MkToc}||\texttt{xvar}\ \$\text{books}))\\ &\text{MkToc} = \texttt{xconst}\ \text{<toc}^{(\texttt{ori,c})}\text{>[()]}; \texttt{xsetcnt}(\texttt{xvar}\ \$\text{books}; \texttt{xmap}\ (\texttt{xchild}; \text{GetTitle}))\\ &\text{GetTitle} = \texttt{xmap}\ (\texttt{xif}\ (\texttt{xwithtag}\ \text{title})\ \texttt{xid}\ \texttt{xconst}\ ())\end{aligned}$$

$$\begin{aligned}&\text{<bibview}^{(\texttt{ori,c})}\text{>[<toc}^{(\texttt{ori,c})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Database}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <title}^{(\texttt{ori,s})}\text{>[Network}^{(\texttt{ori}(\star)\texttt{,s})}\text{]],}\\ &\qquad\text{<book}^{(\texttt{ori,s})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Database}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <price}^{(\texttt{ori,s})}\text{>[20}^{(\texttt{ori}(\star)\texttt{,s})}\text{]],}\\ &\qquad\text{<book}^{(\texttt{ori,s})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Network}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <price}^{(\texttt{ori,s})}\text{>[10}^{(\texttt{ori}(\star)\texttt{,s})}\text{]]]}\end{aligned}$$

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

$$\begin{aligned}&\text{<bibview}^{(\texttt{ori,c})}\text{>[<toc}^{(\texttt{ori,c})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Web Database}^{(\texttt{mod,s})}\text{], <title}^{(\texttt{ori,s})}\text{>[Network}^{(\texttt{ori}(\star)\texttt{,s})}\text{]],}\\ &\qquad\text{<book}^{(\texttt{ori,s})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Web Database}^{(\texttt{mod,s})}\text{], <price}^{(\texttt{ori,s})}\text{>[20}^{(\texttt{ori}(\star)\texttt{,s})}\text{]],}\\ &\qquad\text{<book}^{(\texttt{ori,s})}\text{>[<title}^{(\texttt{ori,s})}\text{>[Network}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <price}^{(\texttt{ori,s})}\text{>[10}^{(\texttt{ori}(\star)\texttt{,s})}\text{]]]}\end{aligned}$$

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 $[\![X]\!].(S,T) = (S',\cdot)$ and $[\![X]\!].(S') = T'$, then $T \sqsubseteq T'$.

$$
\begin{array}{ll}
() & \sqsubseteq () \\
str^{(u,o)} & \sqsubseteq str^{(u,o)} \\
str^{(\mathtt{ori}(\star),\mathtt{s})} & \sqsubseteq str'^{(\mathtt{mod},\mathtt{s})} \\
str^{(\mathtt{ori}(\uparrow),\mathtt{s})} & \sqsubseteq str'^{(\mathtt{mod},\mathtt{s})}, \text{if } str' > str \\
str^{(\mathtt{ori}(\downarrow),\mathtt{s})} & \sqsubseteq str'^{(\mathtt{mod},\mathtt{s})}, \text{if } str' < str \\
str^{(\mathtt{ori}(i),\mathtt{s})} & \sqsubseteq str^{(\mathtt{del},\mathtt{s})}, i \in \{\uparrow,\downarrow,\star,\bullet\} \\
\texttt{<}tag^{(w,o)}\texttt{>}[S] & \sqsubseteq \texttt{<}tag^{(w,o)}\texttt{>}[S'], \text{if } \ S \sqsubseteq S' \\
\texttt{<}tag^{(\mathtt{ori},\mathtt{s})}\texttt{>}[S] & \sqsubseteq \texttt{<}tag^{(\mathtt{del},\mathtt{s})}\texttt{>}[S'], \text{if } S \sqsubseteq S' \\
v, S & \sqsubseteq v', S', \text{if } v \sqsubseteq v' \text{ and } S \sqsubseteq S'
\end{array}
$$

Fig. 11 The update-keeping relation.

That is, we do not require the updated view T and the new view T' from the updated source 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.

- $str^{(\mathtt{ori}(\star),\mathtt{s})} \sqsubseteq str'^{(\mathtt{mod},\mathtt{s})}$. A replica of string $str^{(\mathtt{ori}(\star),\mathtt{s})}$ is changed to $str'^{(\mathtt{mod},\mathtt{s})}$, so the string becomes $str'^{(\mathtt{mod},\mathtt{s})}$ in the view generated from the updated source due to update side-effects.
- $str^{(\mathtt{ori}(\uparrow),\mathtt{s})} \sqsubseteq str'^{(\mathtt{mod},\mathtt{s})}$, if $str' > str$. A replica of string $str^{(\mathtt{ori}(\uparrow),\mathtt{s})}$ is changed to $str'^{(\mathtt{mod},\mathtt{s})}$, which is consistent with the modification indicator in this string, so the string becomes $str'^{(\mathtt{mod},\mathtt{s})}$ in the new view generated from the updated source.
- $str^{(\mathtt{ori}(i),\mathtt{s})} \sqsubseteq str^{(\mathtt{del},\mathtt{s})}$, $i \in \{\uparrow,\downarrow,\bullet,\star\}$. The string $str^{(\mathtt{ori}(i),\mathtt{s})}$ becomes $str^{(\mathtt{del},\mathtt{s})}$ if it has a replica on the view tagged as deleted.
- $<tag^{(\mathtt{ori},\mathtt{s})}>[S] \sqsubseteq <tag^{(\mathtt{del},\mathtt{s})}>[S']$, if $S \sqsubseteq S'$. The $<tag^{(\mathtt{ori},\mathtt{s})}>[S]$ has a replica on the view tagged as deleted, so it becomes $<tag^{(\mathtt{del},\mathtt{s})}>[S']$ after the transformation is applied to the updated source.

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 *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

$$
\begin{array}{lll}
\mathit{Var} & ::= & \mathit{NCName} \\
\mathit{Expr} & ::= & \mathit{String} \mid () \mid \mathit{Expr}, \mathit{Expr} \mid \$\mathit{Var} \\
 & & \mid \texttt{for } \$\mathit{Var} \texttt{ in } \mathit{Expr} \texttt{ return } \mathit{Expr} \\
 & & \mid \texttt{let } \$\mathit{Var} := \mathit{Expr} \texttt{ return } \mathit{Expr} \\
 & & \mid \texttt{if } (\mathit{Cmp}) \texttt{ then } \mathit{Expr} \texttt{ else } \mathit{Expr} \\
 & & \mid \mathit{Axis}\ \mathit{NodeTest} \\
 & & \mid \texttt{element } \mathit{NCName}\ \{\mathit{Expr}\} \\
 & & \mid \mathit{NCName}\ (\mathit{Expr}_1, ..., \mathit{Expr}_n) \\
\mathit{Cmp} & ::= & \mathit{Expr} < \mathit{Expr} \mid \mathit{Expr} = \mathit{Expr} \mid \mathit{Expr} > \mathit{Expr} \\
\mathit{Axis} & ::= & \texttt{child} :: \mid \texttt{descendant} :: \mid \texttt{self} :: \\
\mathit{NodeTest} & ::= & \mathit{NCName} \mid * \mid \texttt{text}() \mid \texttt{node}() \\
\mathit{FunDec} & ::= & \texttt{function } \mathit{NCName}(\mathit{ArgList})\{\mathit{Expr}\} \\
\mathit{ArgList} & ::= & \$\mathit{Var}_1, ..., \$\mathit{Var}_n
\end{array}
$$

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 `fn : sum` and `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 `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 $[\![expr]\!]_{\mathcal{I}}$ returns the translation result for the XQuery Core expression *expr*. Note that $\mathcal{I}$ 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 `for` expression, the subexpression $Expr_1$ is first translated, and then composed with an `xmap`, which takes the transformation `xlet` $\$Var\ [\![Expr_2]\!]_{\mathcal{I}}$ as its argument. That is, the variable $\$Var$ is bound to each value in the sequence returned by $[\![Expr_1]\!]_{\mathcal{I}}$, and then used within the scope $[\![Expr_2]\!]_{\mathcal{I}}$.

In the XQuery Core, the expression *Axis NodeTest* means that *Axis* first produces a list of nodes from its context node, and then from this list *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 *Axis* and *NodeTest*.

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

```
fun xdes($node) = xvar $node; xif xiselement (xchild;(xid||xmap DeepNodes)) (xconst ())
where DeepNodes = xlet $cnode (xfunapp xdes [xvar $cnode])
```

If the input node is a text node, then it does not have any descendant, so the function `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

$$
\begin{array}{lcl}
\llbracket \mathit{String} \rrbracket_{\mathcal{I}} & = & \texttt{xconst}\ \mathit{String}^{(\texttt{ori}(\bullet),\texttt{c})} \\
\llbracket () \rrbracket_{\mathcal{I}} & = & \texttt{xconst}\ () \\
\llbracket \mathit{Expr}_1, \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}} || \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}} \\
\llbracket \$\mathit{Var} \rrbracket_{\mathcal{I}} & = & \texttt{xvar}\ \$\mathit{Var} \\
\llbracket \texttt{for}\ \$\mathit{Var}\ \texttt{in}\ \mathit{Expr}_1\ \texttt{return}\ \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}; \texttt{xmap}\ (\texttt{xlet}\ \$\mathit{Var}\ \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}) \\
\llbracket \texttt{let}\ \$\mathit{Var} := \mathit{Expr}_1\ \texttt{return}\ \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}; \texttt{xlet}\ \$\mathit{Var}\ \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}} \\
\llbracket \texttt{if}\ (\mathit{Expr'}_1 < \mathit{Expr'}_2)\ \texttt{then}\ \mathit{Expr}_1\ \texttt{else}\ \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{xif}\ (\texttt{xlt}\ \llbracket \mathit{Expr'}_1 \rrbracket_{\mathcal{I}}\ \llbracket \mathit{Expr'}_2 \rrbracket_{\mathcal{I}}) \\
 & & \qquad \texttt{xconst}\ (); \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}\ \ \texttt{xconst}\ (); \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}} \\
\llbracket \texttt{if}\ (\mathit{Expr'}_1 = \mathit{Expr'}_2)\ \texttt{then}\ \mathit{Expr}_1\ \texttt{else}\ \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{xif}\ (\texttt{xeq}\ \llbracket \mathit{Expr'}_1 \rrbracket_{\mathcal{I}}\ \llbracket \mathit{Expr'}_2 \rrbracket_{\mathcal{I}}) \\
 & & \qquad \texttt{xconst}\ (); \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}\ \ \texttt{xconst}\ (); \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}} \\
\llbracket \texttt{if}\ (\mathit{Expr'}_1 > \mathit{Expr'}_2)\ \texttt{then}\ \mathit{Expr}_1\ \texttt{else}\ \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{xif}\ (\texttt{xgt}\ \llbracket \mathit{Expr'}_1 \rrbracket_{\mathcal{I}}\ \llbracket \mathit{Expr'}_2 \rrbracket_{\mathcal{I}}) \\
 & & \qquad \texttt{xconst}\ (); \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}\ \ \texttt{xconst}\ (); \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}} \\
\llbracket \mathit{Axis}\ \mathit{NodeTest} \rrbracket_{\mathcal{I}} & = & \llbracket \mathit{Axis} \rrbracket_{\mathcal{I}}; \llbracket \mathit{NodeTest} \rrbracket_{\mathcal{I}} \\
\llbracket \texttt{child}\ :: \rrbracket_{\mathcal{I}} & = & \texttt{xvar}\ \$\texttt{fs:dot}; \texttt{xchild} \\
\llbracket \texttt{descendant}\ :: \rrbracket_{\mathcal{I}} & = & \texttt{xfunapp}\ \texttt{xdes}\ [\texttt{xvar}\ \$\texttt{fs:dot}] \\
\llbracket \texttt{self}\ :: \rrbracket_{\mathcal{I}} & = & \texttt{xvar}\ \$\texttt{fs:dot} \\
\llbracket \mathit{NCName} \rrbracket_{\mathcal{I}} & = & \texttt{xmap}\ (\texttt{xif}\ \texttt{xiselement}\ \texttt{xid}\ (\texttt{xconst}\ ())); \\
 & & \texttt{xmap}\ ((\texttt{xif}\ (\texttt{xwithtag}\ \mathit{NCName})\ \texttt{xid}\ (\texttt{xconst}\ ()))) \\
\llbracket * \rrbracket_{\mathcal{I}} & = & \texttt{xmap}\ (\texttt{xif}\ \texttt{xiselement}\ \texttt{xid}\ (\texttt{xconst}\ ())) \\
\llbracket \texttt{text}() \rrbracket_{\mathcal{I}} & = & \texttt{xmap}\ (\texttt{xif}\ \texttt{xiselement}\ (\texttt{xconst}\ ())\ \texttt{xid}) \\
\llbracket \texttt{node}() \rrbracket_{\mathcal{I}} & = & \texttt{xid} \\
\llbracket \texttt{element}\ \mathit{NCName}\ \{\mathit{Expr}\} \rrbracket_{\mathcal{I}} & = & \texttt{xconst}\ <\mathit{NCName}^{(\texttt{ori},\texttt{c})}>[()]; \texttt{xsetcnt}\ \llbracket \mathit{Expr} \rrbracket_{\mathcal{I}} \\
\llbracket \mathit{NCName}\ (\mathit{Expr}_1, ..., \mathit{Expr}_n) \rrbracket_{\mathcal{I}} & = & \texttt{xfunapp}\ \mathit{NCName}\ [\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, ..., \llbracket \mathit{Expr}_n \rrbracket_{\mathcal{I}}]
\end{array}
$$

Fig. 13 Translation of XQuery core expression.

XQuery function:

$$\texttt{function}\ \mathit{NCName}(\$\mathit{Var}_1, ..., \$\mathit{Var}_n)\{\mathit{Expr}\}$$

is translated into the following function in the bidirectional language:

$$\texttt{fun}\ \mathit{NCName}(\$\mathit{Var}_1, ..., \$\mathit{Var}_n) = \llbracket \mathit{Expr} \rrbracket_{\mathcal{I}}$$

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.

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

The `erase` operator is extended to an environment C by the following definition.

$$
\texttt{erase}(C) \quad = \quad \begin{cases} \cdot, \text{if } C = \cdot \\ \texttt{erase}(C'), \mathit{Var} \mapsto \texttt{erase}(S), \text{if } C = \overline{C', \mathit{Var} \mapsto S} \end{cases}
$$

$$\tau ::= a \mid () \mid \mathtt{string}^{o} \mid \texttt{<}tag^{o}\texttt{>}[\tau] \mid \tau* \mid \tau,\tau \mid \tau|\tau \mid \mu a.\tau \mid \lceil\tau\rceil$$

Fig. 14 Syntax of recursive regular expression types.

$$\frac{}{() \in ()} \qquad \frac{}{str^{(u,o)} \in \mathtt{string}^{o}} \qquad \frac{S \in \tau}{\texttt{<}tag^{(u,o)}\texttt{>}[S] \in \texttt{<}tag^{o}\texttt{>}[\tau]} \qquad \frac{S \in ()|\tau,\tau*}{S \in \tau*}$$

$$\frac{S_1 \in \tau_1 \quad S_2 \in \tau_2}{S_1, S_2 \in \tau_1, \tau_2} \qquad \frac{S \in \tau_i \quad i=1 \ \text{ or } \ i=2}{S \in \tau_1|\tau_2} \qquad \frac{S \in \tau[\mu a.\tau/a]}{S \in \mu a.\tau} \qquad \frac{S \in \tau}{\lceil S\rceil \in \lceil\tau\rceil}$$

Fig. 15 Semantics of types.

Theorem 3 Let C be an environment that maps variables to values. If an XQuery Core expression *Expr* has the value *Val* under `erase`(C), then the forward transformation $[\![[\![Expr]\!]_{\mathcal{I}}]\!]_C(())$ must return a value T and `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 $\lceil\tau\rceil$, and the origin annotation o on 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 `xmap`. The notation $S \in \tau$, defined in Figure 15, means S has the type τ. 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 where possible since it appears only in the annotations of `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 τ 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 $\Gamma \vdash X : \tau \leftrightarrow \tau' \Rightarrow X'$, meaning that under the typing context Γ, if the source data has the type τ, then the transformation X will generate a view with the type τ' after forward executions, and on the other hand, if the updated view has the type τ', then X will generate an updated source data with the type τ after backward executions. The typing context Γ 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 `xconst` S, the operation `mkty`(S) makes a type, say τ', from S: if $S = ()$, then $\tau' = ()$; if $S = str^{(\mathrm{ori}(\bullet),\mathrm{c})}$, then $\tau' = \mathtt{string}^{\mathrm{c}}$; if $S = \texttt{<}tag^{(\mathrm{ori},\mathrm{c})}\texttt{>}[()]$, then $\tau' = \texttt{<}tag^{\mathrm{c}}\texttt{>}[()]$. Hence, $S \in$ `mkty`(S). For the typing of `xsetcnt`, its argument X can be applied to source data with any one of types τ_1,τ_2,..., and τ_n. 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: `xvar`, `xchild`, the parallel composition transformation $\|$, `xmap`, `xif`, and `xfunapp`. The first six constructs need type information to process updated views with insertions. The last construct `xfunapp` uses annotated information to dynamically annotate the function body to be called. A function

$$\frac{}{\Gamma \vdash \texttt{xid} : \tau \leftrightarrow \tau \Rightarrow \texttt{xid}}$$

$$\frac{}{\Gamma \vdash \texttt{xconst}\ S : \tau \leftrightarrow \texttt{mkty}(S) \Rightarrow \texttt{xconst}\ S}$$

$$\frac{\Gamma = \Gamma_1, \mathit{Var} \mapsto \tau', \Gamma_2 \qquad \mathit{Var} \notin \mathit{Dom}(\Gamma_2)}{\Gamma \vdash \texttt{xvar}\ \mathit{Var} : \tau \leftrightarrow \tau' \Rightarrow \texttt{xvar}^{\tau'}\ \mathit{Var}}$$

$$\frac{\tau = \texttt{<}\mathit{tag}_1^{o1}\texttt{>}[\tau_1] | ... | \texttt{<}\mathit{tag}_n^{on}\texttt{>}[\tau_n]}{\Gamma \vdash \texttt{xchild} : \tau \leftrightarrow \tau_1 | ... | \tau_n \Rightarrow \texttt{xchild}^{\tau}}$$

$$\frac{\tau = \texttt{<}\mathit{tag}_1^{o1}\texttt{>}[\tau_1] | ... | \texttt{<}\mathit{tag}_n^{on}\texttt{>}[\tau_n] \quad \Gamma \vdash X : \tau_i \leftrightarrow \tau_i' \Rightarrow X' \quad \Gamma \vdash X : \tau_1 | ... | \tau_n \leftrightarrow \tau'' \Rightarrow X''}{\Gamma \vdash \texttt{xsetcnt}\ X : \tau \leftrightarrow \texttt{<}\mathit{tag}_1^{o1}\texttt{>}[\tau_1'] | ... | \texttt{<}\mathit{tag}_n^{on}\texttt{>}[\tau_n'] \Rightarrow \texttt{xsetcnt}\ X''}$$

$$\frac{\Gamma \vdash X_1 : \tau \leftrightarrow \tau_1 \Rightarrow X_1' \quad \Gamma \vdash X_2 : \tau_1 \leftrightarrow \tau_2 \Rightarrow X_2'}{\Gamma \vdash X_1 ; X_2 : \tau \leftrightarrow \tau_2 \Rightarrow X_1' ; X_2'}$$

$$\frac{\Gamma \vdash X_1 : () \leftrightarrow \tau_1 \Rightarrow X_1' \quad \Gamma \vdash X_2 : () \leftrightarrow \tau_2 \Rightarrow X_2' \quad \tau' = \lceil \tau_1 \rceil, \lceil \tau_2 \rceil}{\Gamma \vdash X_1 \| X_2 : \tau \leftrightarrow \tau_1, \tau_2 \Rightarrow X_1' \|_{\tau}^{\tau'} X_2'}$$

$$\frac{\texttt{singlety}(\tau) = \emptyset}{\Gamma \vdash \texttt{xmap}\ X : \tau \leftrightarrow () \Rightarrow \texttt{xmap}^{()}\ X}$$

$$\frac{\texttt{singlety}(\tau) \neq \emptyset \quad \Gamma; \cdot \vdash_{\texttt{m}} \texttt{xmap}\ X : \tau \leftrightarrow \tau' \quad \Gamma \vdash X : \texttt{mkchoicety}((\texttt{singlety}(\tau)) \leftrightarrow \tau'' \Rightarrow X'}{\Gamma \vdash \texttt{xmap}\ X : \tau \leftrightarrow \texttt{rmbox}(\tau') \Rightarrow \texttt{xmap}^{\tau'}\ X'}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\texttt{ok}} P : \tau \quad \texttt{T}(\tau, P) \neq \emptyset \quad \texttt{F}(\tau, P) \neq \emptyset \\ \Gamma \vdash X_1 : \texttt{mkchoicety}(\texttt{T}(\tau, P)) \leftrightarrow \tau_1 \Rightarrow X_1' \quad \Gamma \vdash X_2 : \texttt{mkchoicety}(\texttt{F}(\tau, P)) \leftrightarrow \tau_2 \Rightarrow X_2' \end{array}}{\Gamma \vdash \texttt{xif}\ P\ X_1\ X_2 : \tau \leftrightarrow \tau_1 | \tau_2 \Rightarrow \texttt{xif}_{\tau_1}^{\tau_2}\ P\ X_1'\ X_2'}$$

$$\frac{\Gamma \vdash_{\texttt{ok}} P : \tau \quad \texttt{T}(\tau, P) = \emptyset \quad \texttt{F}(\tau, P) \neq \emptyset \quad \Gamma \vdash X_2 : \texttt{mkchoicety}(\texttt{F}(\tau, P)) \leftrightarrow \tau_2 \Rightarrow X_2'}{\Gamma \vdash \texttt{xif}\ P\ X_1\ X_2 : \tau \leftrightarrow \tau_2 \Rightarrow \texttt{xif}_{()}^{\tau_2}\ P\ X_1'\ X_2'}$$

$$\frac{\Gamma \vdash_{\texttt{ok}} P : \tau \quad \texttt{T}(\tau, P) \neq \emptyset \quad \texttt{F}(\tau, P) = \emptyset \quad \Gamma \vdash X_1 : \texttt{mkchoicety}(\texttt{T}(\tau, P)) \leftrightarrow \tau_1 \Rightarrow X_1'}{\Gamma \vdash \texttt{xif}\ P\ X_1\ X_2 : \tau \leftrightarrow \tau_1 \Rightarrow \texttt{xif}_{\tau_1}^{()}\ P\ X_1'\ X_2'}$$

$$\frac{\Gamma, \mathit{Var} \mapsto \tau \vdash X : () \leftrightarrow \tau' \Rightarrow X'}{\Gamma \vdash \texttt{xlet}\ \mathit{Var}\ X : \tau \leftrightarrow \tau' \Rightarrow \texttt{xlet}\ \mathit{Var}\ X'}$$

$$\frac{\begin{array}{l} \texttt{fun}\ \mathit{fname}(Var_1, ..., Var_n) = X \in G \\ \Gamma \vdash X_i : () \leftrightarrow \tau_i \Rightarrow X_i' \ \ (1 \leq i \leq n) \quad \mathit{fname}(\tau_1, ..., \tau_n) \notin \mathit{Dom}(\Gamma) \\ [\mathit{Var}_1 \mapsto \tau_1, ..., \mathit{Var}_n \mapsto \tau_n, \mathit{fname}(\tau_1, ..., \tau_n) \mapsto a] \vdash X : () \leftrightarrow \tau' \Rightarrow X' \quad a \text{ is fresh} \end{array}}{\Gamma \vdash \texttt{xfunapp}\ \mathit{fname}\ [X_1, ..., X_n] : \tau \leftrightarrow \mu a.\tau' \Rightarrow \texttt{xfunapp}^{[\tau_1, \cdots, \tau_n]}\ \mathit{fname}\ [X_1', ..., X_n']}$$

$$\frac{\begin{array}{l} \texttt{fun}\ \mathit{fname}(Var_1, ..., Var_n) = X \in G \quad \Gamma \vdash X_i : () \leftrightarrow \tau_i \Rightarrow X_i' \ \ (1 \leq i \leq n) \\ \Gamma = \Gamma_1, \mathit{fname}(\tau_1, ..., \tau_n) \mapsto a, \Gamma_2 \quad \mathit{fname}(\tau_1, ..., \tau_n) \notin \mathit{Dom}(\Gamma_2) \end{array}}{\Gamma \vdash \texttt{xfunapp}\ \mathit{fname}\ [X_1, ..., X_n] : \tau \leftrightarrow a \Rightarrow \texttt{xfunapp}^{[\tau_1, \cdots, \tau_n]}\ \mathit{fname}\ [X_1', ..., X_n']}$$

Fig. 16 Typing rules.

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

The transformation `xvar` is annotated with the type of the variable it references, `xchild` is annotated with the type of its source data, and the parallel composition $\|$ is annotated with the type of its source data and a pair of the boxed view types of its two argument transformations.

There two rules for typing `xmap` X. The first rule applies when the source type τ does not contain any string type

$$\frac{\Gamma \vdash X : () \leftrightarrow \tau' \Rightarrow X' \quad \tau' \in \{\mathtt{string}^{\mathtt{s}}|\mathtt{string}^{\mathtt{c}}, \mathtt{string}^{\mathtt{s}}, \mathtt{string}^{\mathtt{c}}\} \quad \tau \in \{\mathtt{string}^{\mathtt{s}}|\mathtt{string}^{\mathtt{c}}, \mathtt{string}^{\mathtt{s}}, \mathtt{string}^{\mathtt{c}}\}}{\Gamma \vdash_{\mathtt{ok}} \mathtt{xeq}\ X : \tau}$$

$$\frac{\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]}{\Gamma \vdash_{\mathtt{ok}} \mathtt{xwithtag}\ str : \tau}$$

$$\frac{\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_k^{o_k}\texttt{>}[\tau_k]|\tau' \quad \tau' \in \{\mathtt{string}^{\mathtt{s}}|\mathtt{string}^{\mathtt{c}}, \mathtt{string}^{\mathtt{s}}, \mathtt{string}^{\mathtt{c}}\}}{\Gamma \vdash_{\mathtt{ok}} \mathtt{xiselement} : \tau}$$

Fig. 17 The typing rules for predicates.

$$\frac{}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : a \leftrightarrow a}$$

$$\frac{}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : () \leftrightarrow ()}$$

$$\frac{\tau \in \{\mathtt{string}^{o}, \texttt{<}tag^{o'}\texttt{>}[\tau']\} \quad \Gamma \vdash X : \tau\Theta \leftrightarrow \tau'' \Rightarrow X'}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau \leftrightarrow \lceil \tau'' \rceil}$$

$$\frac{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau \leftrightarrow \tau_1}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau* \leftrightarrow \tau_1*}$$

$$\frac{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_1 \leftrightarrow \tau_1' \quad \Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_2 \leftrightarrow \tau_2'}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_1, \tau_2 \leftrightarrow \tau_1', \tau_2'}$$

$$\frac{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_1 \leftrightarrow \tau_1' \quad \Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_2 \leftrightarrow \tau_2'}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_1|\tau_2 \leftrightarrow \tau_1'|\tau_2'}$$

$$\frac{\Gamma; \Theta, [\mu a.\tau/a] \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau \leftrightarrow \tau'}{\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \mu a.\tau \leftrightarrow \mu a.\tau'}$$

Fig. 18 The rules for xmap typing.

or element type, that is, when $\mathtt{singlety}(\tau) = \emptyset$, 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 $\mathtt{singlety}(\tau) \neq \emptyset$. 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 $\mathtt{singlety}(\tau) \neq \emptyset$, the rules in Figure 18 are used to generate the view type of xmap. These rules derive the judgment of the form $\Gamma; \Theta \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau \leftrightarrow \tau'$, 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 $\mu a.\tau/a$. These substitutions can be applied to a type τ', written as $\tau'\Theta$, to replace all free variables a in τ' with $\mu a.\tau$. If Θ does not include any substitution, it is represented as $\cdot$.

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 $\mathtt{rmbox}(\tau')$. 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 $\Gamma; \Theta \vdash_m X : \tau \leftrightarrow \tau'$, then τ has the identical structure as τ' at the box level modulo typing of X under Γ, written as $\tau \cong^X_{\Gamma;\Theta} \tau'$. That is,

- if $\tau = a$, then $\tau' = a$;
- if $\tau = ()$, then $\tau' = ()$;

$$\begin{array}{lcl}
\texttt{singlety}(a) & = & \emptyset \\
\texttt{singlety}(()) & = & \emptyset \\
\texttt{singlety}(\texttt{string}^o) & = & \{\texttt{string}^o\} \\
\texttt{singlety}(\texttt{<}tag^o\texttt{>}[\tau]) & = & \{\texttt{<}tag^o\texttt{>}[\tau]\} \\
\texttt{singlety}(\tau*) & = & \texttt{singlety}(\tau) \\
\texttt{singlety}(\tau_1, \tau_2) & = & \texttt{singlety}(\tau_1) \cup \texttt{singlety}(\tau_2) \\
\texttt{singlety}(\tau_1|\tau_2) & = & \texttt{singlety}(\tau_1) \cup \texttt{singlety}(\tau_2) \\
\texttt{singlety}(\lceil\tau\rceil) & = & \texttt{singlety}(\tau) \\
\texttt{singlety}(\mu a.\tau) & = & \texttt{singlety}(\tau)
\end{array}
\qquad
\begin{array}{lcl}
\texttt{rmbox}(a) & = & a \\
\texttt{rmbox}(()) & = & () \\
\texttt{rmbox}(\texttt{string}^o) & = & \texttt{string}^o \\
\texttt{rmbox}(\texttt{<}tag^o\texttt{>}[\tau]) & = & \texttt{<}tag^o\texttt{>}[\tau] \\
\texttt{rmbox}(\tau*) & = & \texttt{rmbox}(\tau)* \\
\texttt{rmbox}(\tau_1, \tau_2) & = & \texttt{rmbox}(\tau_1), \texttt{rmbox}(\tau_2) \\
\texttt{rmbox}(\tau_1|\tau_2) & = & \texttt{rmbox}(\tau_1)|\texttt{rmbox}(\tau_2) \\
\texttt{rmbox}(\lceil\tau\rceil) & = & \tau \\
\texttt{rmbox}(\mu a.\tau) & = & \mu a.\tau', \text{where } \tau' = \texttt{rmbox}(\tau)
\end{array}$$

Fig. 19 The `rmbox` and `singlety` operators for `xmap` typing.

- if $\tau \in \{\texttt{string}^o, \texttt{<}tag^{o'}\texttt{>}[\tau_1]\}$, then $\tau' = \lceil\tau''\rceil$ and $\Gamma \vdash X : \tau\Theta \leftrightarrow \tau'' \Rightarrow X'$;
- if $\tau = \tau_1*$, then $\tau' = \tau_1'*$ and $\tau_1 \cong^X_{\Gamma;\Theta} \tau_1'$;
- if $\tau = \overline{\tau_1, \tau_2}$, then $\tau' = \overline{\tau_1', \tau_2'}$, $\tau_1 \cong^X_{\Gamma;\Theta} \tau_1'$ and $\tau_2 \cong^X_{\Gamma;\Theta} \tau_2'$;
- if $\tau = \tau_1|\tau_2$, then $\tau' = \tau_1'|\tau_2'$, $\tau_1 \cong^X_{\Gamma;\Theta} \tau_1'$ and $\tau_2 \cong^X_{\Gamma;\Theta} \tau_2'$;
- if $\tau = \mu a.\tau_1$, then $\tau' = \mu a.\tau_1'$, and $\tau_1 \cong^X_{\Gamma;\Theta,\mu a.\tau_1/a} \tau'$.

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

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

$$\texttt{mkchoicety}(TSet) \quad = \quad \begin{cases} \tau, \text{if } TSet = \{\tau\} \\ \tau|\texttt{mkchoicety}(TSet'), \text{if } TSet = \{\tau\} \cup TSet' \end{cases}$$

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 `xmap xchild`. If the source type is $\texttt{<bag}^s\texttt{>[<apple}^s\texttt{>[string}^s\texttt{]*]}$, then the view type annotation of `xmap` is $\lceil\texttt{<apple}^s\texttt{>[string}^s\texttt{]*}\rceil$; if the source type is $\texttt{<bag}^s\texttt{>[<apple}^s\texttt{>[string}^s\texttt{]]*}$, then its view type annotation is $\lceil\texttt{<apple}^s\texttt{>[string}^s\texttt{]}\rceil*$. The different position of box can tell us whether the `apple` elements in a view come from the same `bag` element or from different `bag` elements. If we insert a new `apple` element on the view already containing some `apple` elements, then in the first case, the new `apple` element should be used together with other existing `apple` elements by `xchild` to update the source data, resulting in a new `bag` element containing all `apple` elements; while in the second case, the inserted `apple` element should be processed independently by `xchild`, and a new `bag` element for this new `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 `xif` checks its two branches with its source type τ refined by $\texttt{T}(\tau, P)$ and $\texttt{F}(\tau, P)$, 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 $\texttt{T}(\tau, \texttt{xwithtag}\ str)$ selects in τ the element types with the tag str, while $\texttt{F}(\tau, \texttt{xwithtag}\ str)$ returns the element types with the tag other than str; the operator $\texttt{T}(\tau, \texttt{xiselement})$ selects the element types in τ, while the operator $\texttt{F}(\tau, \texttt{xiselement})$ selects the string types; both operators $\texttt{T}(\tau, \texttt{xeq})$ and $\texttt{F}(\tau, \texttt{xeq})$ return the source type without further selection since the source type τ for `xeq` can only a string type or a choice of strings. The set of types returned by $\texttt{T}(\tau, P)$ and $\texttt{F}(\tau, P)$ are combined into a choice type by using `mkchoicety`. An empty set returned by $\texttt{T}(\tau, P)$ means there is no source data of type τ that makes P true. At this

$$
\begin{array}{lcl}
\mathtt{T}(\tau, \mathtt{xeq}\ X') & = & \{\tau\} \\
\mathtt{F}(\tau, \mathtt{xeq}\ X') & = & \{\tau\} \\
\\
\mathtt{T}(\mathtt{string}^o, \mathtt{xiselement}) & = & \emptyset \\
\mathtt{T}(\mathtt{<}tag^o\mathtt{>}[\tau], \mathtt{xiselement}) & = & \{\mathtt{<}tag^o\mathtt{>}[\tau]\} \\
\mathtt{T}(\mathtt{string}^o|\tau', \mathtt{xiselement}) & = & \mathtt{T}(\tau', \mathtt{xiselement}) \\
\mathtt{T}(\mathtt{<}tag^o\mathtt{>}[\tau]|\tau', \mathtt{xiselement}) & = & \{\mathtt{<}tag^o\mathtt{>}[\tau]\} \cup \mathtt{T}(\tau', \mathtt{xiselement}) \\
\mathtt{F}(\mathtt{string}^o, \mathtt{xiselement}) & = & \{\mathtt{string}^o\} \\
\mathtt{F}(\mathtt{<}tag^o\mathtt{>}[\tau], \mathtt{xiselement}) & = & \emptyset \\
\mathtt{F}(\mathtt{string}^o|\tau', \mathtt{xiselement}) & = & \{\mathtt{string}^o\} \cup \mathtt{F}(\tau', \mathtt{xiselement}) \\
\mathtt{F}(\mathtt{<}tag^o\mathtt{>}[\tau]|\tau', \mathtt{xiselement}) & = & \mathtt{F}(\tau', \mathtt{xiselement}) \\
\\
\mathtt{T}(\tau, \mathtt{xwithtag}\ str) & = & \{\mathtt{<}tag^o\mathtt{>}[\tau'] | \mathtt{<}tag^o\mathtt{>}[\tau'] \in \mathtt{T}(\tau, \mathtt{xiselement}), tag = str\} \\
\mathtt{F}(\tau, \mathtt{xwithtag}\ str) & = & \{\mathtt{<}tag^o\mathtt{>}[\tau'] | \mathtt{<}tag^o\mathtt{>}[\tau'] \in \mathtt{T}(\tau, \mathtt{xiselement}), tag \neq str\}
\end{array}
$$

Fig. 20 The operator T and F.

case, the true branch will not be type-checked, since it is never be executed. It is also similar for the false branch when $\mathtt{F}(\tau, P)$ returns an empty set. It is a typing error when both sets $\mathtt{T}(\tau, P)$ and $\mathtt{F}(\tau, P)$ are empty based on the semantics of the predicates. Hence, there is no the typing rule of `xif` where both $\mathtt{T}(\tau, P)$ and $\mathtt{F}(\tau, P)$ are empty.

The following example shows the usefulness of this typing feature. In this example, suppose we have the code `xif` (`xwithtag` book) `xchild` (`xconst` ()) and the source type $\mathtt{<book}^s\mathtt{>[string}^s\mathtt{]|string}^s$. If the source type of `xif` is directly applied to check its branches, the true branch will cause a type error since `xchild` can only accept elements as its source. Actually, if the true branch is chosen at runtime, we know the `xwithtag` predicate must hold and the source data of this branch must be an element. Hence, we can use the element type $\mathtt{<book}^s\mathtt{>[string}^s\mathtt{]}$ to check the true branch `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 Γ, 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 Var_i is mapped to the type τ_i, and the function name *funname* together with these argument types is mapped to a fresh type variable a. The view type τ' of the function body X probably contains the free type variable a because of recursive function calls. Therefore the view type of `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 call, $\mathtt{xfunapp}^{[\tau_1,...,\tau_n]}\ fname\ [X_1, ..., X_n]$, is to be executed backward and the function *fname* is defined as

$$\mathtt{fun}\ fname(Var_1, ..., Var_n) = X$$

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

$$[Var_1 \mapsto \tau_1, ..., Var_n \mapsto \tau_n, fname(\tau_1, ..., \tau_n) \mapsto a] \vdash X : () \leftrightarrow \tau' \Rightarrow X'$$

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-

$$\begin{aligned}
X[\mu a.\tau/a] &= X, \text{where } X \in \{\texttt{xid}, \texttt{xconst}\ T\}\\
(\texttt{xvar}^{\tau'}\ \mathit{Var})[\mu a.\tau/a] &= \texttt{xvar}^{\tau'[\mu a.\tau/a]}\ \mathit{Var}\\
\texttt{xchild}^{\tau'}[\mu a.\tau/a] &= \texttt{xchild}^{\tau'[\mu a.\tau/a]}\\
(\texttt{xsetcnt}\ X)[\mu a.\tau/a] &= \texttt{xsetcnt}\ X[\mu a.\tau/a]\\
(X_1; X_2)[\mu a.\tau/a] &= X_1[\mu a.\tau/a]; X_2[\mu a.\tau/a]\\
(X_1 \|_{\tau_3}^{\lceil\tau_1\rceil,\lceil\tau_2\rceil} X_2)[\mu a.\tau/a] &= X_1[\mu a.\tau/a] \|_{\tau_3[\mu a.\tau/a]}^{\lceil\tau_1[\mu a.\tau/a]\rceil,\lceil\tau_2[\mu a.\tau/a]\rceil} X_2[\mu a.\tau/a]\\
(\texttt{xmap}^{\tau'}\ X)[\mu a.\tau/a] &= \texttt{xmap}^{\tau'[\mu a.\tau/a]}\ X[\mu a.\tau/a]\\
(\texttt{xif}_{\tau_1}^{\tau_2}\ P\ X_1\ X_2)[\mu a.\tau/a] &= \texttt{xif}_{\tau_1[\mu a.\tau/a]}^{\tau_2[\mu a.\tau/a]}\ P\ X_1[\mu a.\tau/a]\ X_2[\mu a.\tau/a]\\
(\texttt{xlet}\ \mathit{Var}\ X)[\mu a.\tau/a] &= \texttt{xlet}\ \mathit{Var}\ X[\mu a.\tau/a]\\
(\texttt{xfunapp}\ \mathit{fname}^{[\tau_1,\ldots,\tau_n]}\ [X_1, ..., X_n])[\mu a.\tau/a] &= \texttt{xfunapp}\ \mathit{fname}^{[\tau_1[\mu a.\tau/a],\ldots,\tau_n[\mu a.\tau/a]]}\\
&\qquad [X_1[\mu a.\tau/a], ..., X_n[\mu a.\tau/a]]
\end{aligned}$$

Fig. 21 The type variable resolving operation.

lowing definition, we define the well-typed evaluation environment C with respect to a typing context Γ, 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 Γ, we say $C \in \Gamma$ if for all $\mathit{Var} \in \mathit{Dom}(C)$ we have $\mathit{Var} \in \mathit{Dom}(\Gamma)$ and $C(\mathit{Var}) \in \Gamma(\mathit{Var})$.

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 Γ, an evaluation environment C and a source value S, if $\Gamma \vdash X : \tau \leftrightarrow \tau' \Rightarrow X'$, $C \in \Gamma$ and $S \in \tau$, then $[\![X']\!]_C(S) = T$ 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 Γ.

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

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 $\Gamma \vdash X : \tau \leftrightarrow \tau' \Rightarrow X'$, $T \in \tau'$, and $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$ or $[\![X']\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, 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 `toc` in Figure 9. In this example, a judgment $\Gamma_0 \vdash \text{XBody} : () \leftrightarrow \text{VSecTy}* \Rightarrow \text{XBody}'$ is derived. The annotated function body XBody′ is given in Figure 23, where another annotated function body XBody′′ is the result of type-checking XSubTitles in XBody′. The derivation generating XBody′ is indicated by the index t1 in in Table 5, while the derivation for XBody′′ is indicated by the index t1.3.3.2.2.1.2.2 in Table 6. 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 t1 depends on the sequential derivations of t1.1, t1.2 and t1.3. Typing an `xmap` transformation needs a

$$\begin{aligned}
\text{BookTy} &= \texttt{<book}^{\mathrm{s}}\texttt{>}[\text{TitleTy}, \text{AuthorTy}{*}, \text{SecTy}{*}] \\
\text{TitleTy} &= \texttt{<title}^{\mathrm{s}}\texttt{>}[\text{string}^{\mathrm{s}}] \\
\text{AuthorTy} &= \texttt{<author}^{\mathrm{s}}\texttt{>}[\text{string}^{\mathrm{s}}] \\
\text{SecTy} &= \mu\$sec.\texttt{<section}^{\mathrm{s}}\texttt{>}[\text{TitleTy}, \text{SCntTy}{*}] \\
\text{SCntTy} &= \text{ParaTy}|\$sec
\end{aligned}$$

$$\begin{aligned}
\text{ParaTy} &= \texttt{<p}^{\mathrm{s}}\texttt{>}[\text{string}^{\mathrm{s}}] \\
\text{VSecTy} &= \texttt{<section}^{\mathrm{c}}\texttt{>}[\text{TitleTy}, \text{RSecTy}] \\
\text{RSecTy} &= \mu\$s.(\text{OSecTy}{*}) \\
\text{OSecTy} &= \text{CSecTy}|() \\
\text{CSecTy} &= \texttt{<section}^{\mathrm{c}}\texttt{>}[\text{TitleTy}, \$s]
\end{aligned}$$

`toc`(BookTy)	$b
$book-or-section	BookTy

Γ_0

$section	SecTy
`toc`(BookTy)	$b
$book-or-section	BookTy

Γ_1

`toc`(SecTy)	$s
$book-or-section	SecTy
$section	SecTy
`toc`(BookTy)	$b
$book-or-section	BookTy

Γ_2

$section	SecTy
`toc`(SecTy)	$s
$book-or-section	SecTy
$section	SecTy
`toc`(BookTy)	$b
$book-or-section	BookTy

Γ_3

Fig. 22 The source type and contexts for typing `toc`.

derivation 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 t1.3 needs to annotate its component transformation xif (xwithtag "section") XSec (xconst ()) under Γ_0 with the source type TitleTy|AuthorTy|SecTy (i.e., the result of `singlety`(TitleTy, AuthorTy∗, SecTy∗)). With the help of `T` operator for `xwithtag`, the true branch XSec needs to be checked under Γ_0 with the source type SecTy. This check is already done by the derivation t1.3.3.2. With the `F` operator, the false branch `xconst` () is checked with the source type TitleTy|AuthorTy, which is trivial and similar to the derivations t1.3.1.2 and t1.3.2.2.

Together with the examples in Section 3.6, for the forward transformation, we have $\mathcal{C}_0 \in \Gamma_0, () \in ()$ and hence $<\text{section}^{(\texttt{ori,c})}>[S_4] \in \text{VSecTy}*$; for the backward transformation, we have $\mathcal{E}_0 \in \Gamma_0, () \in (), <\text{section}^{(\texttt{ori,c})}>[S_4'] \in \text{VSecTy}*$, and then $() \in ()$ and $\mathcal{E}_0' \in \Gamma_0$.

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`.

$$\begin{aligned}
&<\text{book}^{(\texttt{ori,s})}>[<\text{title}^{(\texttt{ori,s})}>[\text{a}^{(\texttt{ori}(\star)\texttt{,s})}], <\text{author}^{(\texttt{ori,s})}>[\text{b}^{(\texttt{ori}(\star)\texttt{,s})}]], \\
&<\text{book}^{(\texttt{ori,s})}>[<\text{title}^{(\texttt{ori,s})}>[\text{c}^{(\texttt{ori}(\star)\texttt{,s})}], <\text{author}^{(\texttt{ori,s})}>[\text{d}^{(\texttt{ori}(\star)\texttt{,s})}], <\text{author}^{(\texttt{ori,s})}>[\text{e}^{(\texttt{ori}(\star)\texttt{,s})}]]
\end{aligned}$$

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

Table 5 An example of type derivation (Part 1).

Index	Tran	SrcTy	TyCtx	ViewTy
t1	XBody	()	Γ_0	VSecTy*
t1.1	xvar $book-or-section	()	Γ_0	BookTy
t1.2	xchild	BookTy	Γ_0	TitleTy, AuthorTy∗, SecTy∗
t1.3	xmap (xif (xwithtag section) XSec (xconst ()))	TitleTy, AuthorTy∗, SecTy∗	Γ_0	(),()*, VSecTy*
t1.3.1	xif (xwithtag section) XSec (xconst ()))	TitleTy	Γ_0	()
t1.3.1.1	T(TitleTy, xwithtag section)=$\emptyset$, F(TitleTy, xwithtag section)={TitleTy}			
t1.3.1.2	xconst ()	TitleTy	Γ_0	()
t1.3.2	xif (xwithtag section) XSec (xconst ()))	AuthorTy	Γ_0	()
t1.3.2.1	T(AuthorTy, xwithtag section)=$\emptyset$, F(AuthorTy, xwithtag section)={AuthorTy}			
t1.3.2.2	xconst ()	AuthorTy	Γ_0	()
t1.3.3	xif (xwithtag section) XSec (xconst ()))	SecTy	Γ_0	VSecTy
t1.3.3.1	T(SecTy, xwithtag section)={SecTy}, F(SecTy, xwithtag section)=$\emptyset$			
t1.3.3.2	XSec	SecTy	Γ_0	VSecTy
t1.3.3.2.1	xconst <section$^{(\mathtt{ori,c})}$>[()]	()	Γ_1	<section^{c}>[()]
t1.3.3.2.2	xsetcnt (XTitle‖XSubTitles)	<section^{c}>[()]	Γ_1	VSecTy
t1.3.3.2.2.1	XTitle‖XSubTitles	()	Γ_1	TitleTy,RSecTy
t1.3.3.2.2.1.1	XTitle	()	Γ_1	TitleTy
t1.3.3.2.2.1.1.1	xvar $section	()	Γ_1	SecTy
t1.3.3.2.2.1.1.2	xchild	SecTy	Γ_1	TitleTy, SCntTy∗
t1.3.3.2.2.1.1.3	xmap (xif (xwithtag title) xid (xconst ()))	TitleTy, SCntTy∗	Γ_1	TitleTy, ()*
t1.3.3.2.2.1.1.3.1	xif (xwithtag title) xid (xconst ())	TitleTy	Γ_1	TitleTy
t1.3.3.2.2.1.1.3.1.1	T(TitleTy, xwithtag title)={TitleTy}, F(TitleTy, xwithtag title)=$\emptyset$			
t1.3.3.2.2.1.1.3.1.2	xid	TitleTy	Γ_1	TitleTy
t1.3.3.2.2.1.1.3.2	xif (xwithtag title) xid (xconst ())	SCntTy	Γ_1	()
t1.3.3.2.2.1.1.3.2.1	T(SCntTy, xwithtag title)=$\emptyset$, F(SCntTy, xwithtag title)={ParaTy,SecTy }			
t1.3.3.2.2.1.1.3.2.2	xconst ()	ParaTy|SecTy	Γ_1	()

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 `xmap xid`, where type annotations on `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.

<book$^{(\mathtt{ori,s})}$>[<title$^{(\mathtt{ori,s})}$>[a$^{(\mathtt{ori}(\star)\mathtt{,s})}$], <author$^{(\mathtt{ori,s})}$>[b$^{(\mathtt{ori}(\star)\mathtt{,s})}$]],
<book$^{(\mathtt{ori,s})}$>[<title$^{(\mathtt{ori,s})}$>[c$^{(\mathtt{ori}(\star)\mathtt{,s})}$], <author$^{(\mathtt{ori,s})}$>[d$^{(\mathtt{ori}(\star)\mathtt{,s})}$], <author$^{(\mathtt{ori,s})}$>[e$^{(\mathtt{ori}(\star)\mathtt{,s})}$]],
<book$^{(\mathtt{ins,s})}$>[<title$^{(\mathtt{ins,s})}$>[f$^{(\mathtt{ins,s})}$], <author$^{(\mathtt{ins,s})}$>[g$^{(\mathtt{ins,s})}$]]

Now we transform backward the updated view, that is, transform backward each book in the view and its corresponding book in `BookList` by `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 Ω as in [5]. When `xmap` X is executed backward, if its view includes inserted values, then it is possible that the backward execution of X on the inserted values does not have corresponding source data. At that time, the symbol Ω will be introduced, as shown later by the revised backward semantics of `xmap`.

The backward execution of X in `xmap` X may depend on forward executions of its constituent transformations. If the backward execution of X takes Ω as its source, then Ω may also be the source of the involved forward executions. For this case, we have the convention that a forward transformation maps Ω to Ω. Since Ω represents a nonexistent

Table 6 An example of type derivation (Part 2).

Index	Tran	SrcTy	TyCtx	ViewTy
t1.3.3.2.2.1.2	XSubTitles	()	Γ_1	RSecTy
t1.3.3.2.2.1.2.1	xvar $section	()	Γ_1	SecTy
t1.3.3.2.2.1.2.2	XBody	()	Γ_2	OSecTy*
t1.3.3.2.2.1.2.2.1	xvar $book-or-section	()	Γ_2	SecTy
t1.3.3.2.2.1.2.2.2	xchild	SecTy	Γ_2	TitleTy,SCntTy∗
t1.3.3.2.2.1.2.2.3	xmap (xif (xwithtag section) XSec (xconst ()))	TitleTy, SCntTy∗	Γ_2	(), OSecTy*
t1.3.3.2.2.1.2.2.3.1	xif (xwithtag section) XSec (xconst ()))	TitleTy	Γ_2	()
t1.3.3.2.2.1.2.2.3.1.1	T(TitleTy, xwithtag section)=∅, F(TitleTy, xwithtag section)={TitleTy}			
t1.3.3.2.2.1.2.2.3.1.2	xconst ()	TitleTy	Γ_2	()
t1.3.3.2.2.1.2.2.3.2	xif (xwithtag section) XSec (xconst ()))	SCntTy	Γ_2	OSecTy
t1.3.3.2.2.1.2.2.3.2.1	T(SCntTy, xwithtag section)={SecTy} F(SCntTy, xwithtag section)= {ParaTy}			
t1.3.3.2.2.1.2.2.3.2.2	xconst ()	ParaTy	Γ_2	()
t1.3.3.2.2.1.2.2.3.2.3	XSec	SecTy	Γ_2	CSecTy
t1.3.3.2.2.1.2.2.3.2.3.1	xconst <section$^{(\mathtt{ori,c})}$>[()]	()	Γ_3	<section$^{\mathtt{c}}$>[()]
t1.3.3.2.2.1.2.2.3.2.3.2	xsetcnt (XTitle||XSubTitles)	<section$^{\mathtt{c}}$>[()]	Γ_3	
t1.3.3.2.2.1.2.2.3.2.3.2.1	XTitle||XSubTitles	()	Γ_3	TitleTy, $s
t1.3.3.2.2.1.2.2.3.2.3.2.1.1	XTitle	()	Γ_3	TitleTy
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.1	xvar $section	()	Γ_3	SecTy
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.2	xchild	SecTy	Γ_3	TitleTy,SCntTy∗
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3	xmap (xif (xwithtag title) xid (xconst ()))	TitleTy, SCntTy∗	Γ_3	TitleTy, ()*
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3.1	xif (xwithtag title) xid (xconst ())	TitleTy	Γ_3	TitleTy
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3.1.1	T(TitleTy, xwithtag title)={TitleTy}, F(TitleTy, xwithtag title)=∅			
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3.1.2	xid	TitleTy	Γ_3	TitleTy
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3.2	xif (xwithtag title) xid (xconst ())	SCntTy	Γ_3	()
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3.2.1	T(SCntTy, xwithtag title)=∅, F(SCntTy, xwithtag title)={ParaTy,SecTy}			
t1.3.3.2.2.1.2.2.3.2.3.2.1.1.3.2.2	xconst ()	ParaTy|SecTy	Γ_3	()
t1.3.3.2.2.1.2.2.3.2.3.2.1.2	XSubTitles	()	Γ_3	$s
t1.3.3.2.2.1.2.2.3.2.3.2.1.2.1	xvar $section	()	Γ_3	SecTy

value, its length is regarded as 0 and its type can be any type, that is $\Omega \in \tau$ for any τ.

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`.

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

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` $b (<pack$^{(\mathtt{ori,c})}$>[()]; `xsetcnt` (`xvar` $b)))

```
XBody′ = xvar$^{BookTy}$ $book-or-section; //gets the book or section
        xchild$^{BookTy}$; //gets the contents of the book or section
        xmap$^{\lceil()\rceil,\lceil()\rceil*,\lceil VSecTy\rceil*}$(xif$^{()}_{VSecTy}$ (xwithtag section) XSec (xconst ()))
                  //processes each section with XSec and hides non-section elements
XSec = xlet $section (
         <section$^{(ori,c)}$>[()] //builds a section element
         xsetcnt (XTitle$\|^{\lceil TitleTy\rceil,\lceil RSecTy\rceil}_{()}$XSubTitles) //sets title and subsection titles
       )
XTitle = xvar$^{SecTy}$ $section; //gets a section element
         xchild$^{SecTy}$; //gets its content
         xmap$^{\lceil TitleTy\rceil,\lceil()\rceil*}$(xif$^{()}_{TitleTy}$(xwithtag title) xid (xconst ()))
                                                    //keeps only its title
XSubTitles = xfunapp$^{[SecTy]}$ toc [xvar$^{SecTy}$ $section] //builds toc of subsections


//XBody′′ is the annotated function body generated when typing XSubTitles in XBody′
XBody′′ = xvar$^{SecTy}$ $book-or-section; //gets the book or section
        xchild$^{SecTy}$; //gets the contents of the book or section
        xmap$^{\lceil()\rceil,\overline{\lceil()\rceil|\lceil VSecTy\rceil}*}$(xif$^{()}_{VSecTy}$ (xwithtag section) XSec (xconst ()))
                  //processes each section with XSec and hides non-section elements
XSec = xlet $section (
         <section$^{(ori,c)}$>[()] //builds a section element
         xsetcnt (XTitle$\|^{\lceil TitleTy\rceil,\lceil RSecTy\rceil}_{()}$XSubTitles) //sets title and subsection titles
       )
XTitle = xvar$^{SecTy}$ $section; //gets a section element
         xchild$^{SecTy}$; //gets its content
         xmap$^{\lceil TitleTy\rceil,\lceil()\rceil*}$(xif$^{()}_{TitleTy}$(xwithtag title) xid (xconst ()))
                                                    //keeps only its title
XSubTitles = xfunapp$^{[SecTy]}$ toc [xvar$^{SecTy}$ $section] //builds toc of subsections
```

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$^{(\texttt{ins,c})}$>[()] 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 $<tag_1^{o_1}>[\tau_1]|...|<tag_n^{o_n}>[\tau_n]$. If the view T has the type τ_i, then this view is supposed to come from a source element with the type $<tag_i^{o_i}>[\tau_i]$, so we use $tag_i^{(\text{ins},o_i)}$ as the tag of the updated source data.

$$
[\![\texttt{xchild}^{\tau}]\!]_{\mathcal{E}}(S,T) \;=\; \begin{cases} (<tag^{(w,o)}>[T],\mathcal{E}), & \text{if } S = <tag^{(w,o)}>[S'], \tau = <tag_1^{o_1}>[\tau_1]|...|<tag_n^{o_n}>[\tau_n] \\ & \quad S \in \tau_i \text{ and } T \in \tau_i \\ (<tag_i^{(\text{ins},o_i)}>[T],\mathcal{E}), & \text{if } S = \Omega, \tau = <tag_1^{o_1}>[\tau_1]|...|<tag_n^{o_n}>[\tau_n] \\ & \quad T \in \tau_i \\ \texttt{fail}, & \text{otherwise} \end{cases}
$$

It should mention that if the source data of `xchild` is not missing, we have a check to make sure the updated view 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 $<tag_1^{o_1}>[\tau_1]|...|<tag_n^{o_n}>[\tau_n]$. 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 $<tag_i^{o_i}>[\tau_i]$, so the original view of `xchild` has the type τ_i. However, after updated with insertions, even if the updated view T has the view type $\tau_1|...|\tau_n$, T not necessarily has the type τ_i. Hence, the updated source data may not have the type $<tag_i^{o_i}>[\tau_i]$, or even not the type $<tag_1^{o_1}>[\tau_1]|...|<tag_n^{o_n}>[\tau_n]$.

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 $\mathcal{E}$. 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.

$$[\![\texttt{xvar}^\tau\ Var]\!]_{\mathcal{E}}(S,T') \quad = \quad \begin{cases} (S,\mathcal{E}'), & \text{if } S \neq \Omega, \mathcal{E} = \overline{\mathcal{E}_1, Var \mapsto (S_1,S_2), \mathcal{E}_2}, Var \notin Dom(\mathcal{E}_2) \text{ and} \\ & \qquad \mathcal{E}' = \overline{\mathcal{E}_1, Var \mapsto (S_1, \texttt{mg}(S_2,T',\tau)), \mathcal{E}_2} \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

In the revised backward semantics of `xvar`, the variable *Var* in $\mathcal{E}$ may be bound to a pair of Ωs by the most recent enclosed `xlet`. Then after merging, the binding for *Var* may change from $Var \mapsto (\Omega,\Omega)$ into $Var \mapsto (\Omega, T')$.

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.

$$\begin{aligned} S_1: &\ \ \text{<title}^{(\texttt{ori,s})}\text{>[t}^{(\texttt{mod,s})}\text{], <author}^{(\texttt{ins,s})}\text{>[h}^{(\texttt{ins,s})}\text{], <author}^{(\texttt{ori,s})}\text{>[b}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <year}^{(\texttt{ins,s})}\text{>[2009}^{(\texttt{ins,s})}\text{]} \\ S_2: &\ \ \text{<title}^{(\texttt{ori,s})}\text{>[a}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <author}^{(\texttt{ins,s})}\text{>[g}^{(\texttt{ins,s})}\text{], <author}^{(\texttt{ori,s})}\text{>[b}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <author}^{(\texttt{ins,s})}\text{>[k}^{(\texttt{ins,s})}\text{]} \end{aligned}$$

Let the expected merging result have the type $\tau = \overline{\texttt{Title}, \texttt{Author*}, \overline{\texttt{Year}|()}}$, where `title`, `Author`, and `Year` represent the types for the title, author and year elements, respectively. Then, the operation $\texttt{mg}(S_1,S_2,\tau)$ could return the following data that merges all insertions and modifications in S_1 and S_2.

$$\begin{aligned} &\text{<title}^{(\texttt{ori,s})}\text{>[t}^{(\texttt{mod,s})}\text{], <author}^{(\texttt{ins,s})}\text{>[h}^{(\texttt{ins,s})}\text{], <author}^{(\texttt{ins,s})}\text{>[g}^{(\texttt{ins,s})}\text{],} \\ &\qquad \text{<author}^{(\texttt{ori,s})}\text{>[b}^{(\texttt{ori}(\star)\texttt{,s})}\text{], <author}^{(\texttt{ins,s})}\text{>[k}^{(\texttt{ins,s})}\text{], <year}^{(\texttt{ins,s})}\text{>[2009}^{(\texttt{ins,s})}\text{]} \end{aligned}$$

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 $\text{Bob}^{(\texttt{ori}(\star)\texttt{,s})}$ for an author and the environment C maps a variable \$title for a book tile to $\text{Database}^{(\texttt{ori}(\star)\texttt{,s})}$. The type of title is assumed to be $\texttt{string}^{\text{s}}$, and the source takes type $\texttt{string}^{\text{s}}*$. For the transformation `xmap (xlet $a (xvar $a||xvar $title))`, we get the view $\text{Bob}^{(\texttt{ori}(\star)\texttt{,s})}, \text{Database}^{(\texttt{ori}(\star)\texttt{,s})}$. That is, the transformation joins the book tile $\text{Database}^{(\texttt{ori}(\star)\texttt{,s})}$ to each author in the source. If the view is updated to $\text{Bob}^{(\texttt{ori}(\star)\texttt{,s})}, \text{Database}^{(\texttt{ori}(\star)\texttt{,s})}, \text{Tom}^{(\texttt{ins,s})}, \text{Database}^{(\texttt{ins,s})}$, then the backward execution of `xvar` needs to merge $\text{Database}^{(\texttt{ori}(\star)\texttt{,s})}$ and $\text{Database}^{(\texttt{ins,s})}$ by using the ninth rule of the `mg` operator. If the view is updated to $\text{Bob}^{(\texttt{ori}(\star)\texttt{,s})}, \text{Database}^{(\texttt{ori}(\star)\texttt{,s})}, \text{Tom}^{(\texttt{ins,s})}, \text{WebDB}^{(\texttt{ins,s})}$, the title is changed to $\text{WebDB}^{(\texttt{mod,s})}$ 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 Ω, then `xif` loses the information of how to

$$
\begin{aligned}
&\mathtt{mg}((),(),()) &&= ()\\
&\mathtt{mg}(\mathit{str}^{(u,o)}, \mathit{str}^{(u,o)}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}^{(u,o)}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ori}(\star),\mathtt{s})}, \mathit{str}'^{(u,\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}'^{(u,\mathtt{s})}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ori}(\uparrow),\mathtt{s})}, \mathit{str}'^{(\mathtt{mod},\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}'^{(\mathtt{mod},\mathtt{s})}, \text{if } \mathit{str}' > \mathit{str}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ori}(\downarrow),\mathtt{s})}, \mathit{str}'^{(\mathtt{mod},\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}'^{(\mathtt{mod},\mathtt{s})}, \text{if } \mathit{str}' < \mathit{str}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ori}(\uparrow),\mathtt{s})}, \mathit{str}^{(\mathtt{ori}(\downarrow),\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}^{(\mathtt{ori}(\bullet),\mathtt{s})}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ori}(\bullet),\mathtt{s})}, \mathit{str}^{(\mathtt{ori}(i),\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}^{(\mathtt{ori}(\bullet),\mathtt{s})}, i \in \{\star,\bullet,\uparrow,\downarrow\}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{del},\mathtt{s})}, \mathit{str}^{(\mathtt{ori}(i),\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}^{(\mathtt{del},\mathtt{s})}, i \in \{\star,\bullet,\uparrow,\downarrow\}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ins},\mathtt{s})}, \mathit{str}^{(\mathtt{ori}(i),\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathit{str}^{(\mathtt{ori}(i),\mathtt{s})}, i \in \{\star,\bullet,\uparrow,\downarrow\}\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ins},\mathtt{s})}, \mathit{str}'^{(\mathtt{ori}(i),\mathtt{s})}, \mathtt{string}^{\mathtt{s}}) &&= \mathtt{mg}(\mathit{str}^{(\mathtt{mod},\mathtt{s})}, \mathit{str}'^{(\mathtt{ori}(i),\mathtt{s})}, \mathtt{string}^{\mathtt{s}}), \text{if } \mathit{str} \neq \mathit{str}'\\
&\mathtt{mg}(\mathit{str}^{(\mathtt{ins},\mathtt{c})}, \mathit{str}^{(\mathtt{ori}(\bullet),\mathtt{c})}, \mathtt{string}^{\mathtt{c}}) &&= \mathit{str}^{(\mathtt{ori}(\bullet),\mathtt{c})}\\
&\mathtt{mg}(\mathtt{<}\mathit{tag}^{(w,o)}\mathtt{>}[S_1], \mathtt{<}\mathit{tag}^{(w,o)}\mathtt{>}[S_2], \mathtt{<}\mathit{tag}^{o}\mathtt{>}[\tau]) &&= \mathtt{<}\mathit{tag}^{(w,o)}\mathtt{>}[\mathtt{mg}(S_1,S_2,\tau)]\\
&\mathtt{mg}(\mathtt{<}\mathit{tag}^{(\mathtt{ori},\mathtt{s})}\mathtt{>}[S_1], \mathtt{<}\mathit{tag}^{(\mathtt{del},\mathtt{s})}\mathtt{>}[S_2], \mathtt{<}\mathit{tag}^{\mathtt{s}}\mathtt{>}[\tau]) &&= \mathtt{<}\mathit{tag}^{(\mathtt{del},\mathtt{s})}\mathtt{>}[\mathtt{mg}(S_1,S_2,\tau)]\\
&\mathtt{mg}(\mathtt{<}\mathit{tag}^{(\mathtt{ori},o)}\mathtt{>}[S_1], \mathtt{<}\mathit{tag}^{(\mathtt{ins},o)}\mathtt{>}[S_2], \mathtt{<}\mathit{tag}^{o}\mathtt{>}[\tau]) &&= \mathtt{<}\mathit{tag}^{(\mathtt{ori},o)}\mathtt{>}[\mathtt{mg}(S_1,S_2,\tau)]\\
&\mathtt{mg}(S_1,S_2,\tau*) &&= \mathtt{mg}(S_1,S_2,()|\overline{\tau,\tau*})\\
&\mathtt{mg}(S_1,S_2,\overline{\tau_1,\tau_2}) &&= S', \mathtt{mg}(S_{12},S_{22},\tau_2)\\
&\quad \text{if for some } S', S_1 = \overline{S',S_{12}}, S_2 = \overline{S',S_{22}}, S' \in \tau_1, S_{12} \in \tau_2, S_{22} \in \tau_2\\
&\mathtt{mg}(S_1,S_2,\overline{\tau_1,\tau_2}) &&= S_{11}, \mathtt{mg}(S_{12},S_2,\overline{\tau_1,\tau_2})\\
&\quad \text{if } S_1 = \overline{S_{11},S_{12}} \text{ for some } S_{11} \text{ and } S_{12}, S_{11} \in \tau_1, S_{12} \in \overline{\tau_1,\tau_2}, \mathtt{allins}(S_{11}) = \mathtt{true},\\
&\qquad S_2 \neq \overline{S_{11},S'_{22}} \text{ for any } S'_{22}\\
&\mathtt{mg}(S_1,S_2,\overline{\tau_1,\tau_2}) &&= \mathtt{mg}(S_{11},S_{21},\tau_1), \mathtt{mg}(S_{12},S_{22},\tau_2)\\
&\quad \text{if } S_1 = \overline{S_{11},S_{12}} \text{ for some } S_{11} \text{ and } S_{12}, S_{11} \in \tau_1, S_{12} \in \tau_2, S_{12} \notin \overline{\tau_1,\tau_2}, \mathtt{allins}(S_{11}) = \mathtt{true},\\
&\qquad S_2 = \overline{S_{21},S_{22}} \text{ for some } S_{21} \text{ and } S_{22}, S_{21} \in \tau_1, S_{22} \in \tau_2, S_{22} \notin \overline{\tau_1,\tau_2},\\
&\qquad S_2 \neq \overline{S_{11},S'_{22}} \text{ for any } S'_{22}\\
&\mathtt{mg}(S_1,S_2,\overline{\tau_1,\tau_2}) &&= \mathtt{mg}(S_{11},S_{21},\tau_1), \mathtt{mg}(S_{12},S_{22},\tau_2)\\
&\quad \text{if } S_1 = \overline{S_{11},S_{12}} \text{ for some } S_{11} \text{ and } S_{12}, S_{11} \in \tau_1, S_{12} \in \tau_2, \mathtt{allins}(S_{11}) = \mathtt{false},\\
&\qquad S_2 = \overline{S_{21},S_{22}} \text{ for some } S_{21} \text{ and } S_{22}, S_{21} \in \tau_1, S_{22} \in \tau_2, \mathtt{allins}(S_{21}) = \mathtt{false}, \mathtt{orilen}(S_{11}) = \mathtt{orilen}(S_{21}),\\
&\qquad S_2 \neq \overline{S_{11},S'_{22}} \text{ for any } \mathrm{S}'_{22}\\
&\mathtt{mg}(S_1,S_2,\tau_1|\tau_2) &&= \mathtt{mg}(S_1,S_2,\tau_1), \text{if } S_1 \in \tau_1, S_1 \notin \tau_2 \text{ and } S_2 \in \tau_1\\
&\mathtt{mg}(S_1,S_2,\tau_1|\tau_2) &&= \mathtt{mg}(S_1,S_2,\tau_2), \text{if } S_1 \in \tau_2, S_1 \notin \tau_1 \text{ and } S_2 \in \tau_2\\
&\mathtt{mg}(S_1,S_2,\tau_1|\tau_2) &&= \mathtt{mg}(S_1,S_2,\tau_1)\\
&\quad \text{if } S_1 \in \tau_1, S_1 \in \tau_2, S_2 \in \tau_1, S_2 \in \tau_2, \mathtt{mg}(S_1,S_2,\tau_1) = \mathtt{mg}(S_1,S_2,\tau_2) \text{ or } \mathtt{mg}(S_1,S_2,\tau_2) = \mathtt{fail}\\
&\mathtt{mg}(S_1,(),\tau_1|\tau_2) &&= S_1, \text{if } S_1 \in \tau_1, S_1 \notin \tau_2, () \in \tau_2 \text{ and } () \notin \tau_1\\
&\mathtt{mg}(\Omega,S,\tau) &&= S\\
&\mathtt{mg}(S_1,S_2,\tau) &&= \mathtt{mg}(S_2,S_1,\tau), \text{if one case above applies to } \mathtt{mg}(S_2,S_1,\tau)\\
&\mathtt{mg}(S_1,S_2,\tau) &&= \mathtt{fail}, \text{if no other case applies}
\end{aligned}
$$

Fig. 24 The new mg operator.

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 X_1, then it is supposed to be generated by X_1, so X_1 will be chosen; if the updated view has the view type of X_2, then X_2 is chosen. After backward executions of X_1 or X_2, 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.

$$[\![\texttt{xif}^{\tau_2}_{\tau_1}\ P\ X_1\ X_2]\!]_{\mathcal{E}}(S,T) = \begin{cases} [\![P]\!]_{\mathcal{E}'}(S,S'), & \text{if } S \neq \Omega, [\![P]\!]_{\mathcal{E}.1}(S) = \texttt{true} \text{ and } [\![X_1]\!]_{\mathcal{E}}(S,T) = (S',\mathcal{E}') \\ [\![P]\!]_{\mathcal{E}'}(S,S'), & \text{if } S \neq \Omega, [\![P]\!]_{\mathcal{E}.1}(S) = \texttt{false} \text{ and } [\![X_2]\!]_{\mathcal{E}}(S,T) = (S',\mathcal{E}') \\ [\![P]\!]_{\mathcal{E}'}(S,S'), & \text{if } S = \Omega, T \in \tau_1, [\![X_1]\!]_{\mathcal{E}}(\Omega,T) = (S',\mathcal{E}') \\ & \quad \text{and } [\![P]\!]_{\mathcal{E}'.2}(S') = \texttt{true} \\ [\![P]\!]_{\mathcal{E}'}(S,S'), & \text{if } S = \Omega, T \in \tau_2, [\![X_2]\!]_{\mathcal{E}}(\Omega,T) = (S',\mathcal{E}') \\ & \quad \text{and } [\![P]\!]_{\mathcal{E}'.2}(S') = \texttt{false} \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

7.2.5 Mapping transformation

The revision of `xmap` mainly focuses on the `split` operator. The operator `split` is used by `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 `xmap` if its views include insertions. For example, for the source `BookList`, the transformation `xmap xchild` produces a view consisting of a sequence of titles and authors of each book. Then, consider the following updated view.

<title$^{(\texttt{ori,s})}$>[a$^{(\texttt{ori}(\star)\texttt{,s})}$], <author$^{(\texttt{ori,s})}$>[b$^{(\texttt{ori}(\star)\texttt{,s})}$], <author$^{(\texttt{ins,s})}$>[f$^{(\texttt{ins,s})}$], <title$^{(\texttt{ins,s})}$>[g$^{(\texttt{ins,s})}$],
<author$^{(\texttt{ins,s})}$>[h$^{(\texttt{ins,s})}$], <title$^{(\texttt{ori,s})}$>[c$^{(\texttt{ori}(\star)\texttt{,s})}$], <author$^{(\texttt{ori,s})}$>[d$^{(\texttt{ori}(\star)\texttt{,s})}$],
<author$^{(\texttt{ins,s})}$>[i$^{(\texttt{ins,s})}$], <author$^{(\texttt{ori,s})}$>[e$^{(\texttt{ori}(\star)\texttt{,s})}$], <title$^{(\texttt{ins,s})}$>[j$^{(\texttt{ins,s})}$]

where three authors and two titles are inserted. In the backward execution of `xmap xchild`, this view is first divided into subsequences and then each of them is used as the updated view of `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 `xmap` will provide extra information for the new `split` operator, which needs to determine how many subsequences will be generated and how long a subsequence should be.

The new `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 `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 `split` operator. Let `Title` and `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 <book$^{\texttt{s}}$>[`Title`, `Author`∗]∗, and hence T has the type ⌈`Title`, `Author`∗⌉∗. The first book element in `BookList` generates two elements in T, and the second generates three, so `split`(T, [2, 3], ⌈`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 $\overline{\texttt{Title}, \texttt{Author}*}$. Then, for the code `xmap (xif (xwithtag title) xid xconst ())`, the annotated view type will be ⌈`Title`⌉, ⌈()⌉∗. Let the source for this example be (). Then, the original view is also (). If we change the view into <title$^{(\texttt{ins,s})}$>[k$^{(\texttt{ins,s})}$], then the splitting

$$
\begin{array}{ll}
\mathtt{split}((),[],()) & = [] \\
\mathtt{split}(T,[l],\lceil\tau\rceil) & = [(T,\mathtt{e})], \text{if } T\in\tau, \mathtt{orilen}(T)=l \text{ and } \mathtt{allins}(T)=\mathtt{false} \\
\mathtt{split}(T,[],\lceil\tau\rceil) & = [(T,\mathtt{m})], \text{if } T\in\tau \text{ and } \mathtt{allins}(T)=\mathtt{true} \\
\mathtt{split}(T,ls,\tau*) & = \mathtt{split}(T,ls,\overline{\tau,\tau*}), \text{if } ls\neq [] \\
\mathtt{split}(T,[],\tau*) & = \mathtt{split}(T,[],()|\overline{\tau,\tau*}) \\
\mathtt{split}(T,ls,\tau_1|\tau_2) & = \mathtt{split}(T,ls,\tau_1), \text{if } T\in\mathtt{rmbox}(\tau_1) \text{ and } T\notin\mathtt{rmbox}(\tau_2) \\
\mathtt{split}(T,ls,\tau_1|\tau_2) & = \mathtt{split}(T,ls,\tau_2), \text{if } T\in\mathtt{rmbox}(\tau_2) \text{ and } T\notin\mathtt{rmbox}(\tau_1) \\
\mathtt{split}(T,ls,\tau_1|\tau_2) & = \mathtt{split}(T,ls,\tau_1), \text{if } T\in\mathtt{rmbox}(\tau_1), T\in\mathtt{rmbox}(\tau_2) \text{ and } \mathtt{split}(T,ls,\tau_2)=\mathtt{fail} \\
\mathtt{split}(T,ls,\tau_1|\tau_2) & = \mathtt{split}(T,ls,\tau_2), \text{if } T\in\mathtt{rmbox}(\tau_1), T\in\mathtt{rmbox}(\tau_2) \text{ and } \mathtt{split}(T,ls,\tau_1)=\mathtt{fail} \\
\mathtt{split}(T,ls,\tau_1|\tau_2) & = \mathtt{split}(T,ls,\tau_1) \\
\multicolumn{2}{l}{\quad \text{if } T\in\mathtt{rmbox}(\tau_1), T\in\mathtt{rmbox}(\tau_2), \mathtt{split}(T,ls,\tau_1)\neq\mathtt{fail} \text{ and } \mathtt{split}(T,ls,\tau_1)=\mathtt{split}(T,ls,\tau_2)} \\
\mathtt{split}(T,l{:}ls,\overline{\tau_1,\tau_2}) & = \mathtt{split}(T_1,[l],\tau_1)+\!\!+\mathtt{split}(T_2,ls,\tau_2) \\
\multicolumn{2}{l}{\quad \text{if } T=\overline{T_1,T_2} \text{ for some } T_1 \text{ and } T_2, T_1\in\mathtt{rmbox}(\tau_1), T_2\in\mathtt{rmbox}(\tau_2), \mathtt{orilen}(T_1)=l \text{ and } \mathtt{allins}(T_1)=\mathtt{false}} \\
\mathtt{split}(T,ls,\overline{\tau_1,\tau_2}) & = \mathtt{split}(T_1,[],\tau_1)+\!\!+\mathtt{split}(T_2,ls,\tau_2) \\
\multicolumn{2}{l}{\quad \text{if } T=\overline{T_1,T_2} \text{ for some } T_1 \text{ and } T_2, T_1\in\mathtt{rmbox}(\tau_1), T_2\in\mathtt{rmbox}(\tau_2) \text{ and } \mathtt{allins}(T_1)=\mathtt{true}} \\
\mathtt{split}(T,ls,\tau) & = \mathtt{fail}, \text{if no other case applies}
\end{array}
$$

Fig. 25 The `split` operator.

$$
\begin{array}{lll}
\mathtt{iter}(X,[],S,S',\mathcal{E}) & = (S',\mathcal{E}) & \\
\mathtt{iter}(X,(T,\mathtt{m}):ls,S,S',\mathcal{E}) & = \mathtt{iter}(X,ls,S,\overline{S',v'},\mathcal{E}'), \text{where } [\![X]\!]_{\mathcal{E}}(\Omega,T) & = (v',\mathcal{E}') \\
\mathtt{iter}(X,(T,\mathtt{e}):ls,\overline{v,S},S',\mathcal{E}) & = \mathtt{iter}(X,ls,S,\overline{S',v'},\mathcal{E}'), \text{where } [\![X]\!]_{\mathcal{E}}(v,T) & = (v',\mathcal{E}')
\end{array}
$$

Fig. 26 The `iter` operator.

operation will finally get struck on the case $\mathtt{split}((),[],\lceil()\rceil)$. Note that $\mathtt{allins}(()) = $ false. The reason is that the type $\lceil()\rceil$ means some source data is hidden from the view, so for the type $\lceil()\rceil$ 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` X 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 Ω at the backward execution of X.

$$
\begin{array}{lll}
[\![\mathtt{xmap}^{\tau}\ X]\!]_{\mathcal{E}}(S,T) & = & (S',\mathcal{E}'), \text{if } S = () \text{ or } S = \Omega \\
\multicolumn{3}{l}{\quad \text{where } ST = \mathtt{split}(T,[],\tau) \text{ and } (S',\mathcal{E}') = \mathtt{iter}(X,ST,S,(),\mathcal{E})} \\
[\![\mathtt{xmap}^{\tau}\ X]\!]_{\mathcal{E}}(S,T) & = & (S',\mathcal{E}'), \text{if } S = \overline{v_1,...,v_n} \\
\multicolumn{3}{l}{\quad \text{where } ST = \mathtt{split}(T,[\mathtt{len}([\![X]\!]_{\mathcal{E}.1}(v_1)),...,\mathtt{len}([\![X]\!]_{\mathcal{E}.1}(v_n))],\tau)} \\
\multicolumn{3}{l}{\qquad\quad \text{and } (S',\mathcal{E}') = \mathtt{iter}(X,ST,S,(),\mathcal{E})}
\end{array}
$$

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 $\mathtt{xmap}^{\tau}\ X$ and a sequence T, if $\mathtt{split}(T,ls,\tau) = [(T_1,k_1),...,(T_n,k_n)]$, where $k_i \in \{\mathtt{m},\mathtt{e}\}$, then $\overline{\lceil T_1\rceil,...,\lceil T_n\rceil} \in \tau$.

7.2.6 Parallel composition

The backward semantics of $X_1\|_{\tau}^{\lceil\tau_1\rceil,\lceil\tau_2\rceil}X_2$ 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 τ_1 and τ_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 Ω as the updated source.

$$[\![X_1\|_{\tau}^{\lceil\tau_1\rceil,\lceil\tau_2\rceil}X_2]\!]_{\mathcal{E}}(S,T) \quad = \quad \begin{cases} (S,\mathcal{E}'), & \text{if } S \neq \Omega \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

$$\text{where } T = \overline{T_1, T_2}, T_i \in \tau_i, \texttt{orilen}(T_i) = \texttt{len}([\![X_i]\!]_{\mathcal{E}.1}(S))$$
$$((), \mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}(S,T_2), ((), \mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}(S,T_1)$$

7.2.7 Variable binding and element construction

If the source is Ω, `xlet` will bind its variable to (Ω,Ω) in the evaluation environment. However, the second Ω 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 Ω. For example, in the backward transformation `xlet` $\$x$ `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, $\texttt{clean}(\Omega, S') = S'$.

$$[\![\texttt{xlet}\ Var\ X]\!]_{\mathcal{E}}(S,T) \quad = \quad \begin{cases} (S',\mathcal{E}'), \text{if } \$\texttt{inlet} \in Dom(\mathcal{E}) \text{ and } S' \neq \Omega \\ (\text{clean}(S,S'),\mathcal{E}'), \text{if } \$\texttt{inlet} \notin Dom(\mathcal{E}) \text{ and } S' \neq \Omega \\ \texttt{fail}, \text{otherwise} \end{cases}$$

$$\text{where } ((), \overline{\mathcal{E}', Var \mapsto (S,S')}) = [\![X]\!]_{\mathcal{E}''}((),T) \text{ and } \mathcal{E}'' = \overline{\mathcal{E}, Var \mapsto (S,S)}, \quad \text{if } \$\texttt{inlet} \in Dom(\mathcal{E})$$
$$((), \overline{\mathcal{E}', Var \mapsto (S,S'), \$\texttt{inlet} \mapsto (1,1)}) = [\![X]\!]_{\mathcal{E}''}((),T) \text{ and}$$
$$\mathcal{E}'' = \overline{\mathcal{E}, Var \mapsto (S,S), \$\texttt{inlet} \mapsto (1,1)}, \quad \text{if } \$\texttt{inlet} \notin Dom(\mathcal{E})$$

If the source is Ω, `xsetcnt` needs to use Ω 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`.

$$[\![\texttt{xsetcnt}\ X]\!]_{\mathcal{E}}(S,T) \quad = \quad \begin{cases} (<\!tag^{(w',o)}\!>[S''],\mathcal{E}'), & \text{if } S = <\!tag^{(w,o)}\!>[S'], T = <\!tag^{(w',o)}\!>[T'] \text{ and} \\ & \quad (S'',\mathcal{E}') = [\![X]\!]_{\mathcal{E}}(S',T') \\ (<\!tag^{(w',o)}\!>[S''],\mathcal{E}'), & \text{if } S = \Omega, T = <\!tag^{(w',o)}\!>[T'] \text{ and} \\ & \quad (S'',\mathcal{E}') = [\![X]\!]_{\mathcal{E}}(\Omega,T') \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

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 (`ins`, `s`) tag on strings might be changed into (`mod`, `s`) or (`ori`(i), `s`), as shown by the second example in Section 7.2.3. Similarly, the (`ins`, `s`) tag on elements is changed into (`ori`, `s`) when they are merged with non-inserted elements.

For the values tagged with (`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 (`ori`, `c`) or (`ori`(•), `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 $\texttt{string}^{\texttt{s}}*$.

(`xvar` \$src; `xmap` `xlet` \$s (<item$^{(\texttt{ori},\texttt{c})}$>[()]; `xsetcnt` `xvar` \$s))||(`xvar` \$src; `xmap` `xid`).

Then, the above transformation produces the view (), which has the type <item$^{(\texttt{ori},\texttt{c})}$>[`string`]$^{\texttt{s}}*$, $\texttt{string}^{\texttt{s}}*$. If we change the view into <item$^{(\texttt{ins},\texttt{c})}$>[hello$^{(\texttt{ins},\texttt{s})}$], then after backward transformation the variable \$src is updated to

$$\begin{array}{lll}
() & \sqsubseteq & S \\
str^{(\texttt{ins,s})} & \sqsubseteq & str^{(\texttt{ori}(i)\texttt{,s})}, i \in \{\star, \bullet, \uparrow, \downarrow\} \\
str^{(\texttt{ins,s})} & \sqsubseteq & str^{(\texttt{mod,s})} \\
str^{(\texttt{ins,c})} & \sqsubseteq & str^{(\texttt{ori}(\bullet)\texttt{,c})} \\
\texttt{<}tag^{(\texttt{ins},o)}\texttt{>}[S] & \sqsubseteq & \texttt{<}tag^{(\texttt{ori},o)}\texttt{>}[S'], \text{if } S \sqsubseteq S' \\
v, S & \sqsubseteq & v', S', \text{if } v \not\sqsubseteq v' \text{ and } S \sqsubseteq S'
\end{array}$$

Fig. 27 Extensions to the update-keeping relation.

Table 7 A backward execution with insertion (Part 1).

Index	Tran	Src	UTar	Ctx	USrc	UCtx
b1	XBody	()	T'	$\mathcal{E}_0$	()	$\mathcal{E}'_0$
b1.3	xvar$^{\texttt{BookTy}}$ \$book-or-section	()	S$'$	$\mathcal{E}_0$	()	$\mathcal{E}'_0$
b1.2	xchild$^{\texttt{BookTy}}$	S$_0$	S$_1$, S$_2$, S$'''_3$	$\mathcal{E}_0$	S$'$	$\mathcal{E}_0$
b1.1	xmap$^{\lceil()\rceil,\lceil()\rceil*,\lceil\texttt{VSecTy}\rceil*}$(xif$^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S$_1$, S$_2$, S$_3$	(),(), T'	$\mathcal{E}_0$	S$_1$, S$_2$, S'''_3	$\mathcal{E}_0$
b1.1.1	xif$^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S$_1$	()	$\mathcal{E}_0$	S$_1$	$\mathcal{E}_0$
b1.1.1.1	$[\![$xwithtag section$]\!]_{\mathcal{E}_0.1}(S_1)$ = false					
b1.1.1.2	xconst ()	S$_1$	()	$\mathcal{E}_0$	S$_1$	$\mathcal{E}_0$
b1.1.2	xif$^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S$_2$	()	$\mathcal{E}_0$	S$_2$	$\mathcal{E}_0$
b1.1.2.1	$[\![$xwithtag section$]\!]_{\mathcal{E}_0.1}(S_2)$ = false					
b1.1.2.2	xconst ()	S$_2$	()	$\mathcal{E}_0$	S$_2$	$\mathcal{E}_0$
b1.1.3	xif$^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S$_3$	T'	$\mathcal{E}_0$	S$'''_3$	$\mathcal{E}_0$
b1.1.3.1	$[\![$xwithtag section$]\!]_{\mathcal{E}_0.1}(S_3)$ = true					
b1.1.3.2	XSec	S$_3$	T'	$\mathcal{E}_0$	S$'''_3$	$\mathcal{E}_0$
b1.1.3.2.2	xconst S_c	()	S_c	$\mathcal{E}''_1$	()	$\mathcal{E}''_1$
b1.1.3.2.1	xsetcnt (XTitle$\|\|^{\lceil\texttt{TitleTy}\rceil,\lceil\texttt{RSecTy}\rceil}_{()}$XSubTitles)	S_c	T'	$\mathcal{E}_1$	S_c	$\mathcal{E}''_1$
b1.1.3.2.1.1	XTitle$\|\|^{\lceil\texttt{TitleTy}\rceil,\lceil\texttt{RSecTy}\rceil}_{()}$XSubTitles	()	S$'_4$, S$'_6$	$\mathcal{E}_1$	()	$\mathcal{E}''_1$
b1.1.3.2.1.1.1	XTitle	()	S$'_4$	$\mathcal{E}'_1$	()	$\mathcal{E}''_1$
b1.1.3.2.1.1.1.3	xvar$^{\texttt{SecTy}}$ \$section	()	S$''_3$	$\mathcal{E}'_1$	()	$\mathcal{E}''_1$
b1.1.3.2.1.1.1.2	xchild$^{\texttt{SecTy}}$	S$_3$	S$'_4$, S$_5$	$\mathcal{E}'_1$	S$''_3$	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.1	xmap$^{\lceil\texttt{TitleTy}\rceil,\lceil()\rceil*}$(xif$^{()}_{\texttt{TitleTy}}$ (xwithtag title) xid (xconst ()))	S$_4$, S$_5$	S$'_4$,()	$\mathcal{E}'_1$	S$'_4$, S$_5$	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.1.1	xif$^{()}_{\texttt{TitleTy}}$ (xwithtag title) xid (xconst ())	S$_4$	S$'_4$	$\mathcal{E}'_1$	S$'_4$	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.1.1.1	$[\![$xwithtag title$]\!]_{\mathcal{E}'_1.1}(S_4)$ = true					
b1.1.3.2.1.1.1.1.1.2	xid	S$_4$	S$'_4$	$\mathcal{E}'_1$	S$'_4$	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.1.2	xif$^{()}_{\texttt{TitleTy}}$ (xwithtag title) xid (xconst ())	S$_5$	()	$\mathcal{E}'_1$	S$_5$	$\mathcal{E}'_1$
b1.1.3.2.1.1.1.1.2.1	$[\![$xwithtag title$]\!]_{\mathcal{E}'_1.1}(S_5)$ = false					
b1.1.3.2.1.1.1.1.2.2	xconst ()	S$_5$	()	$\mathcal{E}'_1$	S$_5$	$\mathcal{E}'_1$

hello$^{(\texttt{ins,s})}$. Based on the updated environment, executing the transformation forward again generates the new view <item$^{(\texttt{ori,c})}$>[hello$^{(\texttt{ins,s})}$], hello$^{(\texttt{ins,s})}$. 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 `ins` into `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'_6 is a section newly inserted and S'_4 is a title modified. After backward transformation, the updated data S' includes a new subsection S''_6, which is generated from S'_6. The following are the results of `split` operations used by `xmap` at b1.1, b1.1.3.2.1.1.1.1, b1.1.3.2.1.1.2.1.1 and b1.1.3.2.1.1.2.1.1.3.2.1.1.1.1.

Table 8 A backward execution with insertion (Part 2).

Index	Tran	Src	UTar	Ctx	USrc	UCtx
b1.1.3.2.1.1.2	XSubTitles	()	S'_6	$\mathcal{E}_1$	()	$\mathcal{E}'_1$
b1.1.3.2.1.1.2.2	$\text{xvar}^{\texttt{SecTy}}$ \$section	()	S'_3	$\mathcal{E}_1$	()	$\mathcal{E}'_1$
b1.1.3.2.1.1.2.1	xlet \$book-or-section XBody	S_3	S'_6	$\mathcal{E}_1$	S'_3	$\mathcal{E}_1$
b1.1.3.2.1.1.2.1.3	$\text{xvar}^{\texttt{SecTy}}$ \$book-or-section	()	S'_3	$\mathcal{E}_2$	()	$\mathcal{E}'_2$
b1.1.3.2.1.1.2.1.2	$\text{xchild}^{\texttt{SecTy}}$	S_3	S_4, S_5, S''_6	$\mathcal{E}_2$	S'_3	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1	$\text{xmap}^{\lceil()\rceil,\overline{\lceil()\rceil\Vert\lceil\texttt{VSecTy}\rceil}*}(\text{xif}^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S_4, S_5	$(),(),S'_6$	$\mathcal{E}_2$	S_4, S_5, S''_6	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.1	$\text{xif}^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S_4	()	$\mathcal{E}_2$	S_4	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.1.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_2.1}(S_4) = \text{false}$					
b1.1.3.2.1.1.2.1.1.1.2	xconst ()	S_4	()	$\mathcal{E}_2$	S_4	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.2	$\text{xif}^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	S_5	()	$\mathcal{E}_2$	S_5	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.2.1	$[\![\text{xwithtag section}]\!]_{\mathcal{E}_2.1}(S_5) = \text{false}$					
b1.1.3.2.1.1.2.1.1.2.2	xconst ()	S_5	()	$\mathcal{E}_2$	S_5	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.3	$\text{xif}^{()}_{\texttt{VSecTy}}$ (xwithtag section) XSec (xconst ()))	Ω	S'_6	$\mathcal{E}_2$	S''_6	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.3.1	$S'_6 \in \texttt{VSecTy}$					
b1.1.3.2.1.1.2.1.1.3.2	XSec	Ω	S'_6	$\mathcal{E}_2$	S''_6	$\mathcal{E}_2$
b1.1.3.2.1.1.2.1.1.3.2.2	xconst S_c	()	S'_c	$\mathcal{E}'_3$	()	$\mathcal{E}'_3$
b1.1.3.2.1.1.2.1.1.3.2.1	xsetcnt $(\text{XTitle}\Vert^{\lceil\texttt{TitleTy}\rceil,\lceil\texttt{RSecTy}\rceil}_{()}\text{XSubTitles})$	S_c	S'_6	$\mathcal{E}_3$	S'_c	$\mathcal{E}'_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1	$\text{XTitle}\Vert^{\lceil\texttt{TitleTy}\rceil,\lceil\texttt{RSecTy}\rceil}_{()}\text{XSubTitles}$	()	$S'_7, ()$	$\mathcal{E}_3$	()	$\mathcal{E}'_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1.1	XTitle	()	S'_7	$\mathcal{E}_3$	()	$\mathcal{E}'_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1.1.3	$\text{xvar}^{\texttt{SecTy}}$ \$section	()	S''_6	$\mathcal{E}_3$	()	$\mathcal{E}'_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1.1.2	$\text{xchild}^{\texttt{SecTy}}$	Ω	S'_7	$\mathcal{E}_3$	S''_6	$\mathcal{E}_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1.1.1	$\text{xmap}^{\lceil\texttt{TitleTy}\rceil,\lceil()\rceil*}(\text{xif}^{()}_{\texttt{TitleTy}}$ (xwithtag title) xid (xconst ()))	Ω	S'_7	$\mathcal{E}_3$	S'_7	$\mathcal{E}_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1.1.1.1	$\text{xif}^{()}_{\texttt{TitleTy}}$ (xwithtag title) xid (xconst ())	Ω	S'_7	$\mathcal{E}_3$	S'_7	$\mathcal{E}_3$
b1.1.3.2.1.1.2.1.1.3.2.1.1.1.1.1.1	$S'_7 \in \texttt{TitleTy}$					
b1.1.3.2.1.1.2.1.1.3.2.1.1.1.1.1.2	xid	Ω	S'_7	$\mathcal{E}_3$	S'_7	$\mathcal{E}_3$

$$\texttt{split}(T', [0,0,1], \overline{\lceil()\rceil, \lceil()\rceil*, \lceil\texttt{VSecTy}\rceil*}) = [((), \mathsf{e}), ((), \mathsf{e}), (T', \mathsf{e})] \quad (\text{b1.1 })$$

$$\texttt{split}(S'_4, [1,0], \overline{\lceil\texttt{TitleTy}\rceil, \lceil()\rceil*}) = [(S'_4, \mathsf{e}), ((), \mathsf{e})] \quad (\text{b1.1.3.2.1.1.1.1})$$

$$\texttt{split}(S'_6, [0,0], \lceil()\rceil, \overline{\lceil()\rceil\Vert\lceil\texttt{VSecTy}\rceil*}) = [((), \mathsf{e}), ((), \mathsf{e}), (S'_6, \mathtt{m})] \quad (\text{b1.1.3.2.1.1.2.1.1})$$

$$\texttt{split}(S'_7, [], \overline{\lceil\texttt{TitleTy}\rceil, \lceil()\rceil*}) = [(S'_7, \mathtt{m})] \quad (\text{b1.1.3.2.1.1.2.1.1.3.2.1.1.1.1})$$

Note that the `split` operator is not deterministic. For example, the third `split` below may also return $[((), \mathsf{e}), (S'_6, \mathtt{m}), ((), \mathsf{e})]$. If this splitting result is taken, then the updated source of `xmap` at b1.1.3.2.1.1.2.1.1 will become S_4, S''_6, S_5, rather than S_4, S_5, S''_6. However, both S_4, S_5, S''_6 and S_4, S''_6, S_5 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 Core syntax than we presented in this paper. For example, the `order` expression in XQuery, the existential predicate, the attribute axis, XML name spaces, and the constructors for constructing and destructing sequences (or lists) are supported in our implementation. To support XML data input and output, we implemented the constructs `input` and `output`, which both take a file name as their parameters. The XQuery function `doc`("book.xml") is translated to the `input` with the string "book.xml" as its parameter. The source XML data can specify its type by including the processing instruction `<?import srctype =`"*typefile*`.xml`">. The types are represented in the XML format, following the syntax in Figure 14.

Our implementation also supported the definition of libraries to contain commonly used bidirectional functions. There is a manual [16] that describes the bidirectional language from the perspective of users. In addition, our

$$
\begin{aligned}
S &= \texttt{<}\text{book}^{(\texttt{ori,s})}\texttt{>}[S_1, S_2, S_3] & S'_3 &= \texttt{<}\text{section}^{(\texttt{ori,s})}\texttt{>}[S_4, S_5, S''_6]\\
S_1 &= \texttt{<}\text{title}^{(\texttt{ori,s})}\texttt{>}[\text{Data on the Web}^{(\texttt{ori}(\star)\texttt{,s})}] & S''_3 &= \texttt{<}\text{section}^{(\texttt{ori,s})}\texttt{>}[S'_4, S_5]\\
S_2 &= \texttt{<}\text{author}^{(\texttt{ori,s})}\texttt{>}[\text{Serge}^{(\texttt{ori}(\star)\texttt{,s})}] & S'''_3 &= \texttt{<}\text{section}^{(\texttt{ori,s})}\texttt{>}[S'_4, S_5, S''_6]\\
S_3 &= \texttt{<}\text{section}^{(\texttt{ori,s})}\texttt{>}[S_4, S_5] & S'_4 &= \texttt{<}\text{title}^{(\texttt{ori,s})}\texttt{>}[\text{Background}^{(\texttt{mod,s})}]\\
S_4 &= \texttt{<}\text{title}^{(\texttt{ori,s})}\texttt{>}[\text{Introduction}^{(\texttt{ori}(\star)\texttt{,s})}] & S'_6 &= \texttt{<}\text{section}^{(\texttt{ins,c})}\texttt{>}[S'_7]\\
S_5 &= \texttt{<}\text{p}^{(\texttt{ori,s})}\texttt{>}[\text{Text}^{(\texttt{ori}(\star)\texttt{,s})}] & S''_6 &= \texttt{<}\text{section}^{(\texttt{ins,s})}\texttt{>}[S'_7]\\
S_c &= \texttt{<}\text{section}^{(\texttt{ori,c})}\texttt{>}[()] & S'_7 &= \texttt{<}\text{title}^{(\texttt{ins,s})}\texttt{>}[\text{History of Web}^{(\texttt{ins,s})}]\\
S'_c &= \texttt{<}\text{section}^{(\texttt{ins,c})}\texttt{>}[()] & T' &= \texttt{<}\text{section}^{(\texttt{ori,c})}\texttt{>}[S'_4, S'_6]\\
S' &= \texttt{<}\text{book}^{(\texttt{ori,s})}\texttt{>}[S_1, S_2, S'''_3] & &
\end{aligned}
$$

\$book-or-section	S

$\mathcal{C}_0$

\$section	S_3
\$book-or-section	S

$\mathcal{C}_1$

\$book-or-section	S_3
\$section	S_3
\$book-or-section	S

$\mathcal{C}_2$

\$book-or-section	S	S

$\mathcal{E}_0$

\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}_1$

\$book-or-section	S_3	S_3
\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}_2$

\$book-or-section	S	S'

$\mathcal{E}'_0$

\$section	S_3	S'_3
\$book-or-section	S	S

$\mathcal{E}'_1$

\$book-or-section	S_3	S'_3
\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}'_2$

\$section	Ω	Ω
\$book-or-section	S_3	S_3
\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}_3$

\$section	S_3	S'''_3
\$book-or-section	S	S

$\mathcal{E}''_1$

\$section	Ω	S''_6
\$book-or-section	S_3	S_3
\$section	S_3	S_3
\$book-or-section	S	S

$\mathcal{E}'_3$

Fig. 28 Data and environments for the XQuery examples.

implementation can simulate higher-order functions in functional languages by changing the argument *fname* in `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 `ins` or `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 `xconst` or aggregate functions, such as `sum` and `count`. We reviewed the first forty-one XQuery use cases in [15]. Only six of 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 `xchild` 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

[1] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon, “XQuery 1.0: An XML Query Language,” 2005: http://www.w3.org/TR/xquery/

[2] F. Bancilhon and N. Spyratos, “Updating semantics of relational views,” *ACM Trans. Database Syst.*, vol.6, no.4, pp.557–575, 1981.

[3] U. Dayal and P. A. Bernstein, “On the correct translation of update operations on relational views,” *ACM TODS*, vol.7, no.3, pp.381–416, 1982.

[4] G. Gottlob, P. Paolini, and R. Zicari, “Properties and update semantics of consistent views,” *ACM Trans. Database Syst.*, vol.13, no.4, pp.486–524, 1988.

[5] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, “Combinators for bi-directional tree transformations: a linguistic approach to the view update problem,” In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.233–246, ACM Press, 2005.

[6] A. Bohannon, J. A. Vaughan, and B. C. Pierce, “Relational lenses: A language for updateable views,” In *Proc. of the 25th ACM symposium on Principles of Database Systems*, 2006.

[7] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt, “Boomerang: resourceful lenses for string data,” In *Proc. of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’08, pp.407–419, 2008.

[8] D. M.J. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce, “Matching lenses: alignment and view update,” In *Proc. of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP ’10, pp.193–204, 2010.

[9] J. N. Foster, A. Pilkiewicz, and B. C. Pierce, “Quotient lenses,” In *Proc. of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP ’08, pp.383–396, 2008.

[10] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, “A compositional approach to bidirectional model transformation,” In *31st International Conference on Software Engineering*, pp.235–238, 2009.

[11] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano, “Bidirectionalizing graph transformations,” In *Proc. of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP ’10, pp.205–216, 2010.

[12] M. Hofmann, B. Pierce, and D. Wagner, “Symmetric lenses,” In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’11, pp.371–384, 2011.

[13] D. Liu, Z. Hu, and M. Takeichi, “Bidirectional interpretation of xquery,” In *Proc. of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM ’07, pp.21–30, 2007.

[14] Z. Hu, S.-C. Mu, and M. Takeichi, “A programmable editor for developing structured documents based on bidirectional transformations,” In *Proc. of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2004.

[15] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie, “XML Query Use Cases,” 2006. http://www.w3.org/TR/xquery-use-cases/

[16] “Bidirectional XQuery,” http://www.ipl.t.u-tokyo.ac.jp/˜liu/BiXQuery.html

[17] V. Benzaken, G. Castagna, and A. Frisch, “CDuce: an XML-centric general-purpose language,” In *Proc. of the ACM International Conference on Functional Programming*, 2003.

[18] P. Buneman, S. Khanna, and W.-C. Tan, “On propagation of deletions and annotations through views,” In *PODS ’02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp.150–158, New York, NY, USA, 2002. ACM Press.

[19] “Galax: An Implementation of Query,” http://www.galaxquery.org/

[20] A. Marian and J. Simeon, “Projecting XML documents,” In *Proc. of VLDB 2003*, 2003.

[21] D. Draper, P. Fankhauser, M. Fernndez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler, “XQuery 1.0 and XPath 2.0 Formal Semantics,” 2007. http://www.w3.org/TR/xquery-semantics/

[22] D. Olteanu, H. Meuss, T. Furche, and F. Bry, “XPath: Looking forward,” In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, vol.2490 of *LNCS*, pp.109–127. Springer, 2002.

[23] H. Hosoya and B. C. Pierce, “XDuce: A typed XML processing language,” *ACM Transactions on Internet Technology*, vol.3, no.2, pp.117–148, 2003.

[24] M. Wallace and C. Runciman, “Haskell and XML: generic combinators or type-based translation?” In *Proc. of the fourth ACM SIGPLAN international conference on Functional programming*, 1999.

[25] S.-C. Mu, Z. Hu, and M. Takeichi, “An algebraic approach to bi-directional updating,” In *APLAS*, vol.3302, pp.2–20, 2004.

[26] T. Yokoyama and R. Glück, “A reversible programming language and its invertible self-interpreter,” In *PEPM ’07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp.144–153. ACM Press, 2007.

[27] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi, “Bidirectionalization transformation based on automatic derivation of view complement functions,” In *Proc. of the ACM International Conference on Functional Programming*, 2007.

[28] V. P. Braganholo, S. B. Davidson, and C. A. Heuser, “PATAXO: A framework to allow updates through xml views,” *ACM Trans. Database Syst.*, vol.31, no.3, pp.839–886, 2006.

[29] V. Braganholo, S. Davidson, and C. Heuser, “From XML view updates to relational view updates: old solutions to a new problem,” In *Proc. of VLDB 2004*, 2004.

[30] L. Wang and E. A. Rundensteiner, “On the updatability of XML views published over relational data,” In *International Conference on Conceptual Modeling*, 2004.

[31] I. Boneva, A. Caron, B. Groz, Y. Roos, S. Tison, and S. Staworko, “View update translation for xml,” In *Proc. of the 14th International Conference on Database Theory*, ICDT ’11, pp.42–53, 2011.

[32] H. Kozankiewicz, J. Leszczylowski, and K. Subieta, “Updatable XML views,” In *ADBIS*, pp.381–399, 2003.

[33] B. Choi, G. Cong, W. Fan, and S. Viglas, “Updating recursive xml views of relations,” In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *ICDE*, pp.766–775. IEEE, 2007.

[34] J. Liu, C. Liu, T. Härder, and J. Xu Yu, “Updating typical xml views,” In S. G. Lee, Z. Peng, X. Zhou, Y.-S. Moon, R. Unland, and J. Yoo, editors, *DASFAA (1)*, vol.7238 of *Lecture Notes in Computer Science*, pp.126–140. Springer, 2012.

[35] L. Fegaras, “Propagating updates through xml views using lineage tracing,” In *Proc. of the 26th International Conference on Data Engineering* (*ICDE 2010*), pp.309–320, 2010.

[36] L. Wang, E. A. Rundensteiner, M. Mani, and M. Jiang, “Hux: Handling updates in xml,” In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*, pp.1235–1238, 2006.

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.

```
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` $*$ `fromMarcketMem`, 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`.

```
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`($\star$) or `mod` based on their updating status and the origin `s`.

Table A. 1 The persons table.

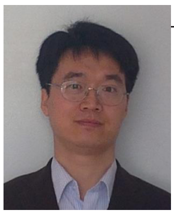
ID	Name	Dept	Phone
1	Tom	Market	9801
2	Peter	Product	9802
3	Kevin	Market	9803

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

ID	Name	DTel	NTel
1	Tom	9801	9801
3	Kevin	9803	9803

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

ID	Name	DTel	NTel
1	Tom	9805	9805
3	Kevin	9803	9803



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.