Bidirectional Interpretation of XQuery

Dongxi Liu, Zhenjiang Hu and Masato Takeichi
Department of Mathematical Informatics, University of Tokyo
{liu,hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract

XQuery is a powerful functional language to query XML data. This paper presents a bidirectional interpretation of XQuery to address the problem of updating XML data through materialized XQuery views. We first design an expressive bidirectional transformation language, and then translate XQuery into this language. The programs of this bidirectional language can be executed in two directions: in the forward direction, they generate materialized views from XML source data; while in the backward direction, they update the source data by reflecting back the updates on views. Hence, with such an interpretation, XQuery can not only query, but also update XML source data through views by being executed forward or backward.

We prove that this language satisfy two well-behaved criteria, the stability property and the extended round-tripping property. The stability property is already used by the existing view updating methods, while the extended round-tripping property is proposed by us for the expressive bidirectional language in this work. It is a challenging task to design suitable well-behaved criteria for this expressive bidirectional language and prove that it satisfies these criteria. We design a type system with regular expression types for this bidirectional language, and give it a novel use to provide guiding information for transforming back the updated views with insertions. We prove that this type system is sound with respect to the forward semantics of this language, and the type of source data is preserved by the backward executions of well-typed programs. We have implemented our approach and applied it to some XQuery use cases from a W3C draft, which confirms the practicability of this approach.

Key words:

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], that is, 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.

The underlying bidirectional language is inspired by the lens language proposed in [5], which includes a collection of combinators, called *lenses*, for tree transformations. The technique used by the lens language is to define both the forward and backward semantics for each combinator, and the backward semantics is responsible for yielding the updated source data. However, as stated in [5], it is not clear how expressive the combinators defined in the lens language could be, or what limits the bidirectional languages defined with this technique will have. For our work, the question becomes whether this technique can be used to define an expressive bidirectional language to interpret XQuery. In this paper, we give a positive answer to this question by designing a bidirectional language that is expressive enough to interpret XQuery. The bidirectional language we designed provides a way of treating the variable binding mechanism in a bidirectional context and defines a set of combinators suitable for constructing and deconstructing XML data. These features are critical to interpret XQuery. For example, the variable binding mechanism provides the basis for interpreting function calls, `for` and `let` expressions in XQuery.

We design a type system with regular expression types for this bidirectional language. Given a program of this language and the type of source data, the type system can check whether this program is well-typed, and if yes, it generates the corresponding view type and annotates this program with appropriate type information. The soundness property of this type system is proved with respect to the forward semantics of this language, so that 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, that cannot be detected statically by this type system. For the successful backward executions of well-typed programs, we prove that they preserve the types of their source data.

We consider 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 annotated on programs by the type system. These annotated types provide guiding information for putting inserted values back in a reasonable way.

The well-behavedness of view updating methods [2,3,6,5] is generally characterized by two criteria, the stability property and the round-tripping property, which may be called differently in different systems. The stability says the source data should not be changed if it is updated by the view updating methods with unchanged views. The round-tripping property says if the source data is updated with respect to an updated view,

then executing the same query on the updated source data should get the same view as the updated view. The stability property is suitable and will be used as one criterion for the well-behavedness of our bidirectional language. However, the second criterion is not suitable for our language any more. For example, if a program of our language creates a view containing several copies of one value from the source data (i.e., the dependency in view [7]), then modifying one copy will violate this criterion even if the value in the source data is correctly updated. This is because executing this program on the updated source data will generate a different view where all copies of the value become the modified one. In this work, the second well-behaved criterion for our bidirectional language is called extended round-tripping property. In this criterion, we do not require the updated view and the view generated from the updated source data be the exactly same. Instead, we require all updates in the former view be kept in the latter view, and all unchanged values in the former view be kept or changed reasonably in the latter view.

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

- We design a bidirectional language expressive enough to interpret XQuery, and prove that it satisfies the stability property and the extended round-tripping property. We present the translation rules from XQuery Core to the bidirectional language, and prove the correctness of this translation.
- We design a type system with regular expression types for this language. We prove that this type system is sound with respect to the forward semantics of this language, and the backward executions of well-typed programs preserve the types of their source data. After type-ckecing programs, this type system annotates the programs with types. These annotated types provide guiding information for transforming backward views with insertions. We illustrate the difficulties of processing insertions on views and present our type-based solution.
- We have implemented our approach and applied it to some XQuery use cases from a W3C draft [8]. The implementation and application examples on XQuery use cases are available at [9].

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 gives 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 XQuery use case from the W3C draft [8]. 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

```
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

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 [9].

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

Fig. 3. Syntax of Data

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 $<\mathsf{author}>\mathsf{Tom}</\mathsf{author}>$ is represented as $\texttt{<}\mathsf{author}\texttt{>}[\mathsf{Tom}]$. This form is borrowed from [10]. Two sequences S_1 and S_2 can be concatenated as a bigger sequence, written as S_1, S_2, or sometimes $\overline{S_1, S_2}$ for clarity. We have $\overline{(), S} = S$ and $\overline{S, ()} = S$.

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 u can be: `ori`, `non`, `mod`, `ins` or `del`, which indicate respectively strings that are in their original state, not allowed to change, modified or inserted ones, or strings to be deleted. An element can only be deleted or inserted, so the updating annotation w is `ori`, `ins` or `del`.

In this work, after backward executions, the annotation `del` is propagated from values on views back to their origins (called provenance in [11]) in the source data. These values can then be removed by an independent procedure like 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.

$$
\begin{aligned}
X \ ::= \ & \texttt{xid} \mid \texttt{xconst}\ T \mid \texttt{xvar}\ \mathit{Var} \mid \texttt{xchild} \mid \texttt{xsetcnt}\ X \\
& \mid X_1; X_2 \mid X_1 || X_2 \mid \texttt{xmap}\ X \mid \texttt{xif}\ P\ X_1\ X_2 \\
& \mid \texttt{xlet}\ \mathit{Var}\ X \mid \texttt{xfunapp}\ \mathit{fname}\ [X_1, ..., X_n] \\
P \ ::= \ & \texttt{xeq}\ X \mid \texttt{xwithtag}\ \mathit{str} \mid \texttt{xiselement} \\
G \ ::= \ & \cdot \mid G, \texttt{fun}\ \mathit{fname}(\mathit{Var}_1, ..., \mathit{Var}_n) = X
\end{aligned}
$$

Fig. 4. Syntax

3.2. *Syntax of the Language*

The syntax of this 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 result of X_1 as the input of X_2, while the transformation $X_1 || X_2$ executes X_1 and X_2 independently with their results combined together as the view. The transformation `xmap` applies its argument transformation X to each item in its source data, corresponding to the map function in function 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 omitted in this paper.

3.3. *Evaluation Contexts*

This language has forward and backward semantics, so we need two evaluation contexts, one for forward semantics, and the other for backward semantics. The context for forward semantics is denoted by $\mathcal{C}$, which maps variables to values; the context for backward semantics is denoted by $\mathcal{E}$, which maps variables to pairs of values. If in the context $\mathcal{E}$ 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 context is represented by a period $\cdot$. We can build new contexts by concatenating contexts and variable-binndings with the comma operator. For example, the new context $C_1, \mathit{Var} \mapsto S, C_2$ is the result of concatenating the context C_1, the binding $\mathit{Var} \mapsto S$, and the context C_2. For clarity, this new context is also written as $\overline{C_1, \mathit{Var} \mapsto S, C_2}$.

For a context $\mathcal{E}$, the notation $\mathcal{E}.1$ denotes a context 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 $\mathcal{E}.1 = \cdot$; or 2) if $\mathcal{E} = \overline{\mathcal{E}', \mathit{Var} \mapsto (S, S')}$, then $\mathcal{E}.1 = \overline{\mathcal{E}'.1, \mathit{Var} \mapsto S}$. Similarly, the notation $\mathcal{E}.2$ denotes a context which maps every variable in $\mathcal{E}$ to the second component of the pair mapped by $\mathcal{E}$ for this variable. $\mathit{Dom}(\mathcal{E})$ (or $\mathit{Dom}(\mathcal{C})$) means the domain of $\mathcal{E}$ (or $\mathcal{C}$).

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

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

Note that the above two forms are for the successful forward and backward executions of X. If its executions get stuck, X returns the special value `fail`.

3.4. *Semantics of the Language*

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

Identity transformation: This transformation keeps the (updated) source data and the (updated) view identical in the both directions. It is just the identity lens in [5] except for the evaluation contexts.

$$[\![\texttt{xid}]\!]_{\mathcal{C}}(S) = S$$
$$[\![\texttt{xid}]\!]_{\mathcal{E}}(S, T) = (T, \mathcal{E})$$

Constant transformation: This transformation returns its argument T_c for any source data in the forward execution. T_c must be (), $str^{(\texttt{ori}}$ or $\texttt{<}tag^{(\texttt{ori,c})}\texttt{>}[()]$. 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 context. When the special value `fail` is generated, the being executed program also terminates with the value `fail`.

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

$$[\![\texttt{xconst } T_c]\!]_{\mathcal{E}}(S, T) = \begin{cases} (S, \mathcal{E}), & \text{if } T_c = T \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

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 context $\mathcal{E}'$, the `mg` operator, defined in Figure 5, is used to merge the updates within S_2 and T'.

$$[\![\texttt{xvar } Var]\!]_{\mathcal{C}}(S) = \begin{cases} T, & \text{if } \mathcal{C} = \overline{\mathcal{C}_1, Var \mapsto T, \mathcal{C}_2} \text{ and } Var \notin Dom(\mathcal{C}_2) \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

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

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

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,

$$
\begin{array}{ll}
\mathtt{mg}((),()) & = () \\
\mathtt{mg}(str^{(u,o)}, str^{(u,o)}) & = str^{(u,o)} \\
\mathtt{mg}(str^{(\mathtt{ori,s})}, str^{(u,\mathtt{s})}) & = str^{(u,\mathtt{s})}, \text{where } u \in \{\mathtt{non}, \mathtt{del}\} \\
\mathtt{mg}(str^{(\mathtt{ori,s})}, str'^{(\mathtt{mod,s})}) & = str'^{(\mathtt{mod,s})} \\
\mathtt{mg}(str^{(u,\mathtt{s})}, str^{(\mathtt{ori,s})}) & = str^{(u,\mathtt{s})}, \text{where } u \in \{\mathtt{non}, \mathtt{del}\} \\
\mathtt{mg}(str'^{(\mathtt{mod,s})}, str^{(\mathtt{ori,s})}) & = str'^{(\mathtt{mod,s})} \\
\mathtt{mg}(\texttt{<}tag^{(w,o)}\texttt{>}[S_1], \texttt{<}tag^{(w,o)}\texttt{>}[S_2]) & = \texttt{<}tag^{(w,o)}\texttt{>}[S'], \text{where } S' = \mathtt{mg}(S_1, S_2) \\
\mathtt{mg}(\texttt{<}tag^{(\mathtt{ori,s})}\texttt{>}[S_1], \texttt{<}tag^{(\mathtt{del,s})}\texttt{>}[S_2]) & = \texttt{<}tag^{(\mathtt{del,s})}\texttt{>}[S'], \text{where } S' = \mathtt{mg}(S_1, S_2) \\
\mathtt{mg}(\texttt{<}tag^{(\mathtt{del,s})}\texttt{>}[S_1], \texttt{<}tag^{(\mathtt{ori,s})}\texttt{>}[S_2]) & = \texttt{<}tag^{(\mathtt{del,s})}\texttt{>}[S'], \text{where } S' = \mathtt{mg}(S_1, S_2) \\
\mathtt{mg}(\overline{v_1, S_1}, \overline{v_2, S_2}) & = \mathtt{mg}(v_1, v_2), \mathtt{mg}(S_1, S_2) \\
\mathtt{mg}(S_1, S_2) & = \mathtt{fail}, \text{if no other case applies}
\end{array}
$$

Fig. 5. The mg Operator

merging elements <Title$^{(\mathtt{mod},s)}$>[Xquery$^{(\mathtt{ori,s})}$] and <title$^{(\mathtt{ori,s})}$>[XQuery$^{(\mathtt{mod,s})}$] will generate the element <Title$^{(\mathtt{mod,s})}$>[XQuery$^{(\mathtt{mod,s})}$], while merging <price$^{(\mathtt{ori,s})}$>[30$^{(\mathtt{mod,s})}$] and <price$^{(\mathtt{ori,s})}$>[25$^{(\mathtt{mod,s})}$] will cause a conflict.

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, it replaces the contents of the source element with the updated contents.

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

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 argument 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.

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

Sequential composition: This transformation takes two argument 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 contexts. Note that the backward execution of X_2 needs to invoke the forward execution of X_1 under the context $\mathcal{E}.1$ to generate the intermediate source data.

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

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

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

Parallel composition: In the forward execution, this transformation executes its argument transformations X_1 and X_2 independently, and composes their views as the final view. Correspondingly, in the backward execution, the updated final view needs to be split into two parts for the updated views of X_1 and X_2, respectively. This is done with 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 $\texttt{len}(T_i) = l_i$. The operator `len` returns the length of a sequence. 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 lis is represented by $[]$, and the concatenation of an item x with a list xs is represented by $x{:}xs$.

$$
\begin{aligned}
&[\![X_1 || X_2]\!]_{\mathcal{C}}(S) &&= [\![X_1]\!]_{\mathcal{C}}(S), [\![X_2]\!]_{\mathcal{C}}(S) \\
&[\![X_1 || X_2]\!]_{\mathcal{E}}(S, T) &&= (\texttt{mg}(S_1', S_2'), \mathcal{E}') \\
&\quad \text{where} \\
&\qquad [T_1, T_2] = \texttt{split}(T, [\texttt{len}([\![X_1]\!]_{\mathcal{E}.1}(S)), \texttt{len}([\![X_2]\!]_{\mathcal{E}.1}(S)]) \\
&\qquad (S_2', \mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}(S, T_2) \\
&\qquad (S_1', \mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}(S, T_1)
\end{aligned}
$$

Mapping transformation: This transformation applies its argument transformation X to each string or element in the source data. If the source data is (), then the target data is also ().In the backward execution, we first split the updated view T into a list of subsequences, each of which is the updated view for a string or an element in the original source data. And then, the `iter` operator, defined in Figure 6, is used to iterate the backward execution of X on each source item and its updated view.

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

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 context $\mathcal{E}'$, 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 $\texttt{true}^{(\texttt{ori,c})}$ (or $\texttt{false}^{(\texttt{ori,c})}$) under the context $\mathcal{E}$ 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 the other

transformations. This condition is necessary for the well-behavedness of bidirectional transformations, as discussed in the next section. The predicate P is defined in the next subsection and more illustrations are given there by examples.

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

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.

$$
\begin{aligned}
&[\![\texttt{xlet}\ \mathit{Var}\ X]\!]_{\mathcal{C}}(S) = [\![X]\!]_{\mathcal{C}'}((\)), \text{where } \mathcal{C}' = \mathcal{C}, \mathit{Var} \mapsto S \\
&[\![\texttt{xlet}\ \mathit{Var}\ X]\!]_{\mathcal{E}}(S,T) = (S',\mathcal{E}') \\
&\quad \text{where } ((), \overline{\mathcal{E}', \mathit{Var} \mapsto (S,S')}) = [\![X]\!]_{\mathcal{E}''}((),T) \text{ and } \mathcal{E}'' = \mathcal{E}, \mathit{Var} \mapsto (S,S)
\end{aligned}
$$

The forward semantics of this construct is the same as that of the let in conventional functional programming languages. Its backward semantics is defined by executing backward the transformation X under the context $\overline{\mathcal{E}, \mathit{Var} \mapsto (S,S)}$, where the variable Var is bound to a pair of the original source data S, since the value of this variable has not been updated at this point. After the backward execution of X, the generated context $\overline{\mathcal{E}', \mathit{Var} \mapsto (S,S')}$ contains the updated source data S' in its top binding.

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

$$\texttt{fun}\ \mathit{fname}(\mathit{Var}_1, ..., \mathit{Var}_n) = X$$

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

$$
\begin{aligned}
&\texttt{xfunapp}\ \mathit{fname}\ [X_1, ..., X_n] = \texttt{xconst}\ (); X'_1 \\
&\quad \text{where} \\
&\qquad X'_1 = X_1; \texttt{xlet}\ \mathit{Var}_1\ X'_2 \\
&\qquad X'_2 = X_2; \texttt{xlet}\ \mathit{Var}_2\ X'_3 \\
&\qquad ... \\
&\qquad X'_n = X_n; \texttt{xlet}\ \mathit{Var}_n\ X
\end{aligned}
$$

In this definition, all argument transformations are first evaluated, and then their results are bound to the corresponding variables. And then, the function body X is executed. Note that in this definition, the source data for the function body is always the empty sequence () due to the definition of `xlet`. That is, it 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.

$$\texttt{guard}(S, str') = \begin{cases} str^{(\texttt{non,s})}, & \text{if } S = str^{(\texttt{ori,s})} \text{ and } str = str' \\ S, & \text{else if } S \in \{str^{(u,\texttt{s})}, str^{(\texttt{ori,c})}\}, u \neq \texttt{ori} \text{ and } str = str' \\ \text{fail}, & \text{otherwise} \end{cases}$$

Fig. 7. The `guard` Operator

3.5. *Predicates*

Each predicate is also defined with both forward and backward semantics. The target data produced by predicates is either $\texttt{true}^{(\texttt{ori,c})}$ or $\texttt{false}^{(\texttt{ori,c})}$, 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 $\texttt{true}^{(\texttt{ori,c})}$ or $\texttt{false}^{(\texttt{ori,c})}$.

Comparison: In the forward direction, this predicate returns $\texttt{true}^{(\texttt{ori,c})}$ if the source data and the view of its argument X contain the same string even with different annotations, or $\texttt{false}^{(\texttt{ori,c})}$ if they contain different strings. In the backward direction, the updated source data returned is generated by the `guard` operator, defined in Figure 7, which makes sure the updated source data has not changed by the branch transformations of `xif` and will not be changed by other transformations, either. In other words, in the definition below if the updated source S' has been modified to a different string, then `guard` will fail; if S' is $str^{(\texttt{ori,s})}$, then it has the chance to be changed by other transformations, so the `guard` operator will change it to $str^{(\texttt{non,s})}$. Thus, if other transformations change one of its replicas into $str'^{(\texttt{mod,s})}$, then finally the `mg` operator will detect such conflicting updates and fail. Similarly, the view of the argument X is also guarded, and propagated back into the context $\mathcal{E}'$ by its backward execution.

$$[\![\texttt{xeq}\ X]\!]_{\mathcal{C}}(S) = \begin{cases} \texttt{true}^{(\texttt{ori,c})}, & \text{if } S = str^{(u,o)} \text{ and } [\![X]\!]_{\mathcal{C}}(()) = str^{(u',o')} \\ \texttt{false}^{(\texttt{ori,c})}, & \text{else if } S = str^{(u,o)}, [\![X]\!]_{\mathcal{C}}(()) = str'^{(u',o')} \text{ and } str \neq str' \\ \text{fail}, & \text{otherwise} \end{cases}$$

$$[\![\texttt{xeq}\ X]\!]_{\mathcal{E}}(S, S') = \begin{cases} (\texttt{guard}(S', str), \mathcal{E}'), & \text{if } S = str^{(u,o)} \text{ and } [\![X]\!]_{\mathcal{E}.1}(()) = str'^{(u',o')} \\ \text{fail}, & \text{otherwise} \end{cases}$$

$$\text{where } ((), \mathcal{E}') = [\![X]\!]_{\mathcal{E}}((), \texttt{guard}(str'^{(u',o')}, str'))$$

For example, suppose we have the source data: $\texttt{10}^{(\texttt{ori,s})}$, and the transformation: `xid||xif` P `xid xconst` $()$, where $P =$ `xeq xconst` $\texttt{10}^{(\texttt{ori,c})}$. After the forward transformation, the view is $\overline{\texttt{10}^{(\texttt{ori,s})}, \texttt{10}^{(\texttt{ori,s})}}$. If we change the view into $\overline{\texttt{11}^{(\texttt{mod,s})}, \texttt{10}^{(\texttt{ori,s})}}$, then the backward execution of `xif` will return the updated source data $\texttt{10}^{(\texttt{non,s})}$, while the left-most `xid` return the updated source data $\texttt{11}^{(\texttt{mod,s})}$. Thus, the mg operator in the parallel composition will fail to merge these two updated strings. On the other hand, if we change the view into $\overline{\texttt{10}^{(\texttt{ori,s})}, \texttt{11}^{(\texttt{mod,s})}}$, the `guard` operator in `xeq` will detect this illegal modification. Similarly, we can define `xlt` and `xgt` for less-than and greater-than comparisons.

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 context $\mathcal{E}'$ directly since the tags of elements are not allowed to change and nothing is needed to guard.

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

$$[\![\texttt{xwithtag}\ str]\!]_{\mathcal{E}}(S, S') = \begin{cases} (S', \mathcal{E}), & \text{if } S = \texttt{<}tag^{(w,o)}\texttt{>}[S_1] \\ \text{fail}, & \text{otherwise} \end{cases}$$

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 context $\mathcal{E}'$ directly in its backward execution.

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

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

3.6. *Programming Examples*

To help understand this language, we give a programming example, where this bidirectional language is used to implement the recursive `toc` function in Figure 1. The program is given in Figure 8, which is divided into several pieces for the convenience of reading. 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. The id attribute is omitted in this implementation. It is similar to the code `XTitle` except that the construct `xchild` should be replaced by `xattribute` in our implementation.

4. Well-Behaved Bidirectional Transformations

In this section, we will give two properties that well-behaved bidirectional transformations should have, and then prove that the transformations defined in the previous section are well-behaved.

```
fun toc($book-or-section) =
  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
where
  XSec = xlet $section (
           xconst <section>[()];   //builds a section element with (ori,c) omitted
           xsetcnt (XTitle||XSubTitles) //sets the 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. 8. The Motivating Example in Bidirectional Language

4.1. *Stability Property*

This property says if the view is not updated, then after backward transformation, the source data 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 the $\texttt{expand}(\mathcal{C})$ operator builds a new context for backward execution from the context $\mathcal{C}$. It is defined as: 1) if $\mathcal{C} = \cdot$, then $\texttt{expand}(\mathcal{C}) = \cdot$; or 2) if $\mathcal{C} = \overline{\mathcal{C}', \mathit{Var} \mapsto S}$, then $\texttt{expand}(\mathcal{C}) = \texttt{expand}(\mathcal{C}'), \mathit{Var} \mapsto (S, S)$.

Theorem 1. If $[\![X]\!]_{\mathcal{C}}(S) = T$, then $[\![X]\!]_{\mathcal{E}}(S, T) = (S, \mathcal{E})$, where $\mathcal{E} = \texttt{expand}(\mathcal{C})$.

Proof. The proof proceeds by structural induction on X. All cases of X' can be proved straightforwardly. The proof for some cases is shown below.

- Case $X = \texttt{xvar}\ \mathit{Var}$. If $[\![X]\!]_{\mathcal{C}}(S) = T$, then $\mathcal{C}$ must have the form $\overline{\mathcal{C}_1, \mathit{Var} \mapsto T, \mathcal{C}_2}$, where $\mathit{Var} \notin \mathit{Dom}(\mathcal{C}_2)$. Since $\mathcal{E} = \texttt{expand}(\mathcal{C})$, we know $\mathcal{E} = \mathcal{E}_1, \mathit{Var} \mapsto (T, T), \mathcal{E}_2$, where $\mathcal{E}_1 = \texttt{expand}(\mathcal{C}_1)$, $\mathcal{E}_2 = \texttt{expand}(\mathcal{C}_2)$ and $\mathit{Var} \notin \mathit{Dom}(\mathcal{E}_2)$. Suppose $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$. Then, according to the backward semantics of `xvar`, we have $S = S'$ and $\mathcal{E}' = \mathcal{E}_1, \mathit{Var} \mapsto (T, \texttt{mg}(T, T)), \mathcal{E}_2$. By Lemma 2 below, $T = \texttt{mg}(T, T)$, so $\mathcal{E}' = \mathcal{E}$.
- Case $X = \texttt{xsetcnt}\ X'$. If $[\![X]\!]_{\mathcal{C}}(S) = T$, then S must have the form $\texttt{<}\mathit{tag}^{(w,o)}\texttt{>}[S']$, and T has the form $\texttt{<}\mathit{tag}^{(w,o)}\texttt{>}[T']$, where $T' = [\![X']\!]_{\mathcal{C}}(S')$. By the induction hypothesis on X', we get $[\![X']\!]_{\mathcal{E}}(S', T') = (S', \mathcal{E})$, so $[\![X]\!]_{\mathcal{E}}(S, T) = (S, \mathcal{E})$, where $\mathcal{E} = \texttt{expand}(\mathcal{C})$.
- Case $X = X_1; X_2$. If $[\![X]\!]_{\mathcal{C}}(S) = T$, then there must exist S', such that $[\![X_1]\!]_{\mathcal{C}}(S) = S'$ and $[\![X_2]\!]_{C}(S') = T$. By the induction hypotheses on X_2 and X_1, respectively, the proof for this case is done.

□

Lemma 2. If $S = \texttt{mg}(S', S')$, then $S = S'$.

Proof. The proof proceeds by structural induction on the structure of S'. For each case of S', the proof is straightforward. Two cases are proved below as examples.

- Case $S' = \mathit{str}^{(u,o)}$. If $S = \texttt{mg}(S', S')$, then $S = \mathit{str}^{(u,o)}$, so $S = S'$.
- Case $S' = \texttt{<}\mathit{tag}^{(u,o)}\texttt{>}[S'']$. If $S = \texttt{mg}(S', S')$, then $S = \texttt{<}\mathit{tag}^{(u,o)}\texttt{>}[S''']$, where $S''' = \texttt{mg}(S'', S'')$. By the induction hypothesis on S'', we get $S''' = S''$, so $S' = S$.

□

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 called `PUTGET` property in [5,6], and `consistent condition` in [2].

4.2.1. *Restrictions 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 restricted for our bidirectional language. This restriction has been recognized in [7]. The intuitiveness behind this restriction is that if one value in the source data S has several replicas in T, where only one replica of this value is modified, then after updating the source data and producing the new view T' again, all replicas of this value will be changed to the updated value. Hence, $T \neq T'$. The following example explains this restriction. Suppose we have the following source data.

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

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.

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

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

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

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

Thus, we can see the old updated view and the new updated view is not identical and the existing round-trpping property fails to accommodate the above transformation of our bidirectional language. In the new updated view, the change of title in the `book` element is called an update side-effect.

4.2.2. *Extended Round-Tripping Property*

The extended round-tripping property is defined as: if $[\![X]\!]_{\mathcal{E}}(S,T) = (S',\mathcal{E}')$ and $[\![X]\!]_{\mathcal{E}'.2}(S') = T'$, then $T \sqsubseteq T'$. That is, we do not require the old updated view T

$$
\begin{array}{ll}
() & \sqsubseteq S \\
str^{(u,o)} & \sqsubseteq str^{(u,o)} \\
str^{(\texttt{ori},\texttt{s})} & \sqsubseteq str^{(u,\texttt{s})}, \text{where } u \in \{\texttt{non}, \texttt{del}, \texttt{mod}\} \\
str^{(\texttt{ins},\texttt{c})} & \sqsubseteq str^{(\texttt{ori},\texttt{c})} \\
\texttt{<}tag^{(w,o)}\texttt{>}[S] & \sqsubseteq \texttt{<}tag^{(w,o)}\texttt{>}[S'], \text{where } S \sqsubseteq S' \\
\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S] & \sqsubseteq \texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S'], \text{where } S \sqsubseteq S' \\
\texttt{<}tag^{(\texttt{ins},\texttt{c})}\texttt{>}[S] & \sqsubseteq \texttt{<}tag^{(\texttt{ori},\texttt{c})}\texttt{>}[S'], \text{where } S \sqsubseteq S' \\
v, S & \sqsubseteq v', S', \text{where } v \sqsubseteq v' \text{ and } S \sqsubseteq S' \\
v, S & \sqsubseteq v', S', \text{where } v \not\sqsubseteq v' \text{ and } v, S \sqsubseteq S'
\end{array}
$$

Fig. 9. The Update-Keeping Relation

and the new updated view T' be identical. Instead, we relate T to T' with the update-keeping relation $\sqsubseteq$, which is defined in Figure 9. Informally, this criterion requires all updates made to T are still kept on T', and T' includes more updates than T due to update side-effects. In the following, we explain the rationale behind the update-keeping relation by discussing each case in its definition.

- $() \sqsubseteq S$. In this case, all values in the old updated view have been related to some parts of the new updated view, and S contains only the values as the side-effects of updates.
- $str^{(u,o)} \sqsubseteq str^{(u,o)}$. In this case, the string $str^{(u,o)}$ is related to itself, that means this string is not affected by the side-effects of updates.
- $str^{(\texttt{ori},\texttt{s})} \sqsubseteq str^{(u,\texttt{s})}$, where $u \in \{\texttt{non}, \texttt{del}, \texttt{mod}\}$. In this case, the string $str^{(\texttt{ori},\texttt{s})}$ must have other replicas, which have been updated.
- $str^{(\texttt{ins},\texttt{c})} \sqsubseteq str^{(\texttt{ori},\texttt{c})}$. In this case, both strings come from `xconst` since they are annotated by `c`. The string $str^{(\texttt{ins},\texttt{c})}$ is inserted into the view as an update, but the `xconst` can only generate the same string with `ori` annotation. So these two strings are related.
- $\texttt{<}tag^{(w,o)}\texttt{>}[S] \sqsubseteq \texttt{<}tag^{(w,o)}\texttt{>}[S']$, where $S \sqsubseteq S'$. In this case, the element $\texttt{<}tag^{(w,o)}\texttt{>}[S]$ is not affected by the side-effects of updates except for its contents. Its contents S is related to the contents of its replicas by further check.
- $\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S] \sqsubseteq \texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S']$, where $S \sqsubseteq S'$. In this case, the element $\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S]$ has other replicas and some replicas are being deleted.
- $\texttt{<}tag^{(\texttt{ins},\texttt{c})}\texttt{>}[S] \sqsubseteq \texttt{<}tag^{(\texttt{ori},\texttt{c})}\texttt{>}[S']$, where $S \sqsubseteq S'$. In this case, both elements come from `xconst`, so they are related as the constant strings above. Their contents are related by further check.
- $v, S \sqsubseteq v', S'$, where $v \sqsubseteq v'$ and $S \sqsubseteq S'$. In this case, v and v' have been related, so we ask the remaining parts S and S' are related.
- $v, S \sqsubseteq v', S'$, where $v \not\sqsubseteq v'$ and $v, S \sqsubseteq S'$. In this case, v and v' have not been related yet, so we try to relate $\overline{v, S}$ to S' by skipping v'.

For example, the old updated view and the new updated view in the previous section (Section 4.2.1) are related by the update-keeping relation.

The update-keeping relation is reflexive by Lemma 3, so the round-tripping property is covered by the extended round-tripping property. That is, if a view-updaing method satisfies the round-tripping property, then it also satisfies the the extended round-tripping property.

Lemma 3. If $S_1 = S_2$, then $S_1 \sqsubseteq S_2$.

Proof. We perform structural induction on the structure of S_1. Each case of S_1 can be proved straightforwardly according to the definition of the $\sqsubseteq$ relation. The proof is sketched as below.

- Case $S_1 = \texttt{<}tag^{(w,o)}\texttt{>}[S']$. If $S_1 = S_2$, then $S_2 = \texttt{<}tag^{(w,o)}\texttt{>}[S']$. By the induction hypothesis on S', we have $S' \sqsubseteq S'$, and hence $\texttt{<}tag^{(w,o)}\texttt{>}[S'] \sqsubseteq \texttt{<}tag^{(w,o)}\texttt{>}[S']$.
- Case $S_1 = v, S'$. If $S_1 = S_2$, then $S_2 = v, S'$. By the induction hypotheses on v and S', we get $v \sqsubseteq v$ and $S \sqsubseteq S'$, and hence $v, S' \sqsubseteq v, S'$.

□

Lemma 4. If $v_1, S_1 \sqsubseteq S$, then there must exit a subsequence v_2, S_2 of S, such that $v_1 \sqsubseteq v_2$ and $S_1 \sqsubseteq S_2$.

Proof. The proof proceeds by structural induction on the derivation of $v_1, S_1 \sqsubseteq S$. There are two cases to derive $v_1, S_1 \sqsubseteq S$. In each case, S must have the form v', S', otherwise $v_1, S_1 \not\sqsubseteq S$, contradicting the assumption. We prove for each case that the claim holds.

In the first case, we have $v_1 \sqsubseteq v'$ and $S_1 \sqsubseteq S'$, so the claim holds by letting $v_2 = v'$ and $S_2 = S'$.

In the second case, we have $v_1 \not\sqsubseteq v'$ and $v_1, S_1 \sqsubseteq S'$. By the induction hypothesis on the derivation of $v_1, S_1 \sqsubseteq S'$, there must exist a subsequence of S', say v'', S'', such that $v_1 \sqsubseteq v''$ and $S_1 \sqsubseteq S''$, and hence the claim holds with $v_2 = v''$ and $S_2 = S''$.

□

Lemma 5. If S is a subsequence of S_1 and $S_1 \sqsubseteq S_2$, then $S \sqsubseteq S_2$.

Proof. If $S = ()$, then the claim follows from $() \sqsubseteq S_2$. For the case where $S \neq ()$, we prove by contradiction. Suppose $S \not\sqsubseteq S_2$. Then there must exist v in S, such that $v \not\sqsubseteq S_2$. Since v is also in S_1, we have $S_1 \not\sqsubseteq S_2$, contradicting the assumption of this lemma.

□

Lemma 6. If $S \sqsubseteq S_1$ and S_1 is a subsequence of S_2, then $S \sqsubseteq S_2$.

Proof. The lemma is proved similarly as Lemma 5.

□

The update-keeping relation is transitive by Lemma 7, which is used to prove the bidirectional transformations defined in Section 3 satisfy the extended round-tripping property.

Lemma 7. If $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_3$, then $S_1 \sqsubseteq S_3$.

Proof. The proof proceeds by structural induction on the last rule applied to derive $S_1 \sqsubseteq S_2$.

- Rule $() \sqsubseteq S$. By this rule, $S_1 = ()$ and $S_2 = S$. The proof is trivial since for any S_3, we have $() \sqsubseteq S_3$.
- Rule $str^{(u,\texttt{s})} \sqsubseteq str^{(u,\texttt{s})}$, where $u \neq \texttt{ori}$. We have $S_1 = S_2$ for this rule. So if $S_2 \sqsubseteq S_3$, then $S_1 \sqsubseteq S_3$. It is similar to prove the claim for the rule $str^{(u,\texttt{c})} \sqsubseteq str^{(u,\texttt{c})}$, where $u \neq \texttt{ins}$.

– Rule $str^{(\texttt{ori},\texttt{s})} \sqsubseteq str^{(u,\texttt{s})}$, where $u \in \{\texttt{ori}, \texttt{non}, \texttt{mod}, \texttt{del}\}$. If this rule applies, then $S_1 = str^{(\texttt{ori},\texttt{s})}$, and S_2 has four possibilities: $str^{(\texttt{ori},\texttt{s})}$, $str^{(\texttt{non},\texttt{s})}$, $str^{(\texttt{del},\texttt{s})}$ and $str'^{(\texttt{mod},\texttt{s})}$. For each possibility of S_2, by assuming that $S_2 \sqsubseteq S_3$, we prove $S_1 \sqsubseteq S_3$. If $S_2 = str^{(\texttt{ori},\texttt{s})}$, then $S_1 \sqsubseteq S_3$ follows from $S_2 \sqsubseteq S_3$ since $S_1 = S_2$. If $S_2 = str^{(\texttt{non},\texttt{s})}$ and $S_2 \sqsubseteq S_3$, then by Lemma 4 there must be v_3 in S_3, such that $str^{(\texttt{non},\texttt{s})} \sqsubseteq v_3$, which implies v_3 must be $str^{(\texttt{non},\texttt{s})}$. Hence, $S_1 \sqsubseteq v_3$ and we get $S_1 \sqsubseteq S_3$ by Lemma 6. The other two possibilities are proved similarly. For the rule $str^{(\texttt{ins},\texttt{c})} \sqsubseteq str^{(u,\texttt{c})}$, where $u \in \{\texttt{ori}, \texttt{ins}\}$, the proof is also done similarly.
– Rule $\texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_1] \sqsubseteq \texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_2]$, where $S'_1 \sqsubseteq S'_2$ and $w \neq \texttt{ori}$. For this rule, $S_1 = \texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_1]$ and $S_2 = \texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_2]$. If $S_2 \sqsubseteq S_3$, then by Lemma 4 there must be v_3 in S_3, such that $\texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_2] \sqsubseteq v_3$, which implies v_3 must have the form $\texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_3]$ and $S'_2 \sqsubseteq S'_3$. By the induction hypothesis on $S'_1 \sqsubseteq S'_2$, we get $S'_1 \sqsubseteq S'_3$, so $S_1 \sqsubseteq v_3$. By Lemma 6, we get $S_1 \sqsubseteq S_3$. It is similar to prove the rule $\texttt{<}tag^{(w,\texttt{c})}\texttt{>}[S'_1] \sqsubseteq \texttt{<}tag^{(w,\texttt{c})}\texttt{>}[S'_2]$, where $S'_1 \sqsubseteq S'_2$ and $w \neq \texttt{ins}$.
– Rule $\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S'_1] \sqsubseteq \texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_2]$, where $S'_1 \sqsubseteq S'_2$ and $w \in \{\texttt{ori}, \texttt{del}\}$. For this rule, $S_1 = \texttt{<}tag^{(w,\texttt{s})}\texttt{>}[S'_1]$ and S_2 is either $\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S'_2]$ or $\texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S'_2]$. Let $S_2 = \texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S'_2]$. If $S_2 \sqsubseteq S_3$, then by Lemma 4 there must be v_3 in S_3, such that $\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S'_2] \sqsubseteq v_3$, which means v_3 must be $\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S'_3]$ or $\texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S'_3]$, where $S'_2 \sqsubseteq S'_3$. By the induction hypothesis on $S'_1 \sqsubseteq S'_2$, we get $S'_1 \sqsubseteq S'_3$, so $S_1 \sqsubseteq v_3$, and by Lemma 6, $S_1 \sqsubseteq S_3$. Simiarly, we can prove the claim for another possibility of S_2, that is $S_2 = \texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S'_2]$. The proof is also done similarly for the rule $\texttt{<}tag^{(\texttt{ins},\texttt{c})}\texttt{>}[S'_1] \sqsubseteq \texttt{<}tag^{(w,\texttt{c})}\texttt{>}[S'_2]$, where $S'_1 \sqsubseteq S'_2$ and $w \in \{\texttt{ori}, \texttt{ins}\}$.
– Rule $v_1, S'_1 \sqsubseteq S_2$ (either of last two rules in the definition of $\sqsubseteq$ relation). Let $S_1 = v_1, S'_1$. If $S_1 \sqsubseteq S_2$, then by Lemma 4 there must be a subsequence of S_2, say v_2, S'_2, such that $v_1 \sqsubseteq v_2$ and $S'_1 \sqsubseteq S'_2$. Moreover, if $S_2 \sqsubseteq S_3$, then by Lemma 5 we have $v_2, S'_2 \sqsubseteq S_3$. By Lemma 4, there must be a subsequence v_3, S'_3 of S_3 such that $v_2 \sqsubseteq v_3$ and $S'_2 \sqsubseteq S'_3$. By the induction hypotheses on v_1 and S'_1, we get $v_1 \sqsubseteq v_3$, $S'_1 \sqsubseteq S'_3$, and hence by Lemma 6, we get $S_1 \sqsubseteq S_3$.

□

Lemma 8. If $S_1 \sqsubseteq S'_1$ and $S_2 \sqsubseteq S'_2$, then $S_1, S_2 \sqsubseteq S'_1, S'_2$.

Proof. We perform induction on the structure of S_1.

If $S_1 = ()$, then $S_1, S_2 = S_2$. Since $S_2 \sqsubseteq S'_2$ and S'_2 is a subsequence of S'_1, S'_2, by Lemma 6, we have $S_2 \sqsubseteq S'_1, S'_2$, so the claim holds.

If $S_1 = v$ and $S_1 \sqsubseteq S'_1$, then by Lemma 4, there must be v' in S'_1, such that $v \sqsubseteq v'$. Let S'_1 have the form S'_{11}, v', S'_{12}, so $S'_1, S'_2 = S'_{11}, v', S'_{12}, S'_2$. By Lemma 6, $S_2 \sqsubseteq S'_{12}, S'_2$, so $v, S_2 \sqsubseteq v', S'_{12}, S'_2$. By using Lemma 6 again, we get $v, S_2 \sqsubseteq S'_{11}, v', S'_{12}, S'_2$, that is, $S_1, S_2 \sqsubseteq S'_1, S'_2$.

if $S_1 = v, S_{11}$ and $S_1 \sqsubseteq S'_1$, then by Lemma 4 there must be a subsequence of S'_1, say v', S'_{11}, such that $v \sqsubseteq v'$ and $S_{11} \sqsubseteq S'_{11}$. Let S'_1 have the form $S'_{10}, v', S'_{11}, S'_{12}$, so $S'_1, S'_2 = S'_{10}, v', S'_{11}, S'_{12}, S'_2$. By Lemma 6, $S_2 \sqsubseteq S'_{12}, S'_2$. By the induction hypothesis on S_{11}, we get $S_{11}, S_2 \sqsubseteq S'_{11}, S'_{12}, S'_2$. Hence, using Lemma 6 again, we get $v, S_{11}, S_2 \sqsubseteq S_{10}, S'_{11}, v', S'_{12}, S'_2$, that is, $S_1, S_2 \sqsubseteq S'_1, S'_2$. □

The `mg` operator combines updates within two replicas of one value. An update side-effect is caused, for instance, when a string with `ori` annotation is merged with a string

with `mod` annotation. The following lemma says all updates within replicas are kept in the merging result.

Lemma 9. If $S = \mathtt{mg}(S_1, S_2)$, then $S_1 \sqsubseteq S$ and $S_2 \sqsubseteq S$.

Proof. The proof proceeds by structural induction on the structure of S_1. In the following, we prove some cases as examples.

- Case $S_1 = ()$. If $S = \mathtt{mg}(S_1, S_2)$, then S_2 must be (), and S is also (). Hence, $S_1 \sqsubseteq S_3$.
- Case $S_1 = str^{(\mathtt{ori,s})}$. If $S = \mathtt{mg}(S_1, S_2)$, then S_2 can be $str^{(\mathtt{ori,s})}$, $str^{(\mathtt{non,s})}$, $str'^{(\mathtt{mod,s})}$ or $str^{(\mathtt{del,s})}$. For each possibility of S_2, we prove the claim holds. If $S_2 = str^{(\mathtt{ori,s})}$, then S is also the same string as S_1 and S_2, so the claim holds. If $S_2 = str'^{(\mathtt{mod,s})}$ (or $str^{(\mathtt{non,s})}$, $str^{(\mathtt{del,s})}$), then $S = S_2$, and hence the claim holds.
- Case $S_1 = \texttt{<}tag^{(\mathtt{ori,s})}\texttt{>}[S_1']$. If $S = \mathtt{mg}(S_1, S_2)$, then S_2 is either $\texttt{<}tag^{(\mathtt{ori,s})}\texttt{>}[S_2']$ or $\texttt{<}tag^{(\mathtt{del,s})}\texttt{>}[S_2']$. If $S_2 = \texttt{<}tag^{(\mathtt{ori,s})}\texttt{>}[S_2']$, then $S = \texttt{<}tag^{(\mathtt{ori,s})}\texttt{>}[S']$, where $S' = \mathtt{mg}(S_1', S_2')$. By the induction hypothesis on S_1', we have $S_1' \sqsubseteq S'$ and $S_2' \sqsubseteq S'$, and hence $S_1 \sqsubseteq S$ and $S_2 \sqsubseteq S$. Similarly, we can prove the claim for $S_2 = \texttt{<}tag^{(\mathtt{del,s})}\texttt{>}[S_2']$.
- Case $S_1 = v_1, S_1'$. If $S = \mathtt{mg}(S_1, S_2)$, then S_2 must have the form v_2, S_2', and $S = v', S'$, where $v' = \mathtt{mg}(v_1, v_2)$ and $S' = \mathtt{mg}(S_1', S_2')$. By the induction hypotheses on v_1 and S_1', respectively, we get $v_1 \sqsubseteq v'$, $v_2 \sqsubseteq v'$, $S_1' \sqsubseteq S'$ and $S_2' \sqsubseteq S'$. Hence, $v_1, S_1' \sqsubseteq v', S'$ and $v_2, S_2' \sqsubseteq v', S'$.

□

We need to extend the updating-keeping relation into the backward execution context for proving that our bidirectional language satisfies the extended round-tripping property.

Definition 1. For two environments $\mathcal{E}$ and $\mathcal{E}'$, we say $\mathcal{E} \sqsubseteq \mathcal{E}'$, if 1) $\mathcal{E} = \cdot$ and $\mathcal{E}' = \cdot$; or 2) $\mathcal{E} = \mathcal{E}_1, Var \mapsto (S, S_1)$ and $\mathcal{E}' = \mathcal{E}_1', Var \mapsto (S, S_2)$, such that $\mathcal{E}_1 \sqsubseteq \mathcal{E}_1'$ and $S_1 \sqsubseteq S_2$.

Lemma 10. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then $\mathcal{E} \sqsubseteq \mathcal{E}'$.

Proof. The proof proceeds by structural induction on X. For the cases where X is `xid`, `xconst` or `xchild`, the backward execution of X does not change the environment $\mathcal{E}$, that is, $\mathcal{E} = \mathcal{E}'$, so by Lemma 3 the claim holds. Other cases are proved as below.

- Case $X = \mathtt{xvar}\ Var$. Suppose $\mathcal{E} = \mathcal{E}_1, Var \mapsto (S_1, S_2), \mathcal{E}_2$ and $Var \notin Dom(\mathcal{E}_2)$. Then, if $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, we know $\mathcal{E}' = \mathcal{E}_1, Var \mapsto (S_1, \mathtt{mg}(S_2, T)), \mathcal{E}_2$. By Lemma 9, we have $S_2 \sqsubseteq \mathtt{mg}(S_2, T))$, so the claim follows.
- Case $X = \mathtt{xsetcnt}\ X'$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (\texttt{<}tag^{(w',o)}\texttt{>}[S''], \mathcal{E}')$, then we know S must have the form $\texttt{<}tag^{(w,o)}\texttt{>}[S']$, T the form $\texttt{<}tag^{(w',o)}\texttt{>}[T']$, and $[\![X']\!]_{\mathcal{E}}(S', T') = (S'', \mathcal{E}')$. By the induction hypothesis on X', we get the claim $\mathcal{E} \sqsubseteq \mathcal{E}'$.
- Case $X = X_1; X_2$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then there must exist S'' and $\mathcal{E}''$, such that $[\![X_2]\!]_{\mathcal{E}}([\![X_1]\!]_{\mathcal{E}.1}(S), T) = (S'', \mathcal{E}'')$ and $[\![X_1]\!]_{\mathcal{E}''}(S, S'') = (S', \mathcal{E}')$. By the induction hypotheses on X_2 and X_1, we have $\mathcal{E} \sqsubseteq \mathcal{E}''$ and $\mathcal{E}'' \sqsubseteq \mathcal{E}'$. According to Lemma 7, we get $\mathcal{E} \sqsubseteq \mathcal{E}'$.
- Case $X = X_1 || X_2$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then there must exist $\mathcal{E}''$, T_1 and T_2, such that $T = T_1, T_2$, $[\![X_2]\!]_{\mathcal{E}}(S, T_2) = (S_2', \mathcal{E}'')$, $[\![X_1]\!]_{\mathcal{E}''}(S, T_1) = (S_1', \mathcal{E}')$ and $S' = \mathtt{mg}(S_1', S_2')$. By applying the induction hypotheses on X_2 and X_1, we get $\mathcal{E} \sqsubseteq \mathcal{E}'$.

– Case $X =$ `xmap` X'. If S and T are both (), then $\mathcal{E} = \mathcal{E}'$, so the claim holds. Otherwise, for all subsequences generated by the operator `split`, we iteratively apply the induction hypothesis on X. That yields the claim.
– Case $X =$ `xif` P X_1 X_2. We do a case analysis on P. If P is `xiselement` or `xwithtag` *tag*,then the claim is proved by applying the induction hypotheses on X_1 and X_2; if $P =$ `xeq` X', the claim is proved by applying the induction hypotheses on X' and then X_1, or X' and then X_2.
– Case $X =$ `xlet` *Var* X'. If $[\![$`xlet` *Var* $X']\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$, then $[\![X']\!]_{\mathcal{E}''}((),T) = ((), \mathcal{E}', Var \mapsto (S, S'))$, where $\mathcal{E}'' = \mathcal{E}, Var \mapsto (S,S)$. By the induction hypothesis on X', we get $\mathcal{E}, Var \mapsto (S,S) \sqsubseteq \mathcal{E}', Var \mapsto (S,S')$, so $\mathcal{E} \sqsubseteq \mathcal{E}'$.
– Case $X =$ `xfunapp` *fname* $[X_1, ..., X_n]$. The proof for the case $X = X_1; X_2$ applies here, since this construct is implemented as a sequential composition transformation. □

The following theorem says the bidirectional transformation language defined in Section 3 satisfies the extended round-tripping property.

Theorem 11. If $[\![X]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$ and $[\![X]\!]_{\mathcal{E}'.2}(S') = T'$, then $T \sqsubseteq T'$.

Proof. The proof proceeds by structural induction on the structure of X.
– Case $X =$ `xid`. We have $[\![X]\!]_{\mathcal{E}}(S,T) = (T, \mathcal{E})$, so $[\![X]\!]_{\mathcal{E}.2}(T) = T$. The claim holds trivially since $T \sqsubseteq T$.
– Case $X =$ `xconst` T_c. If $[\![X]\!]_{\mathcal{E}}(S,T) = (S, \mathcal{E})$, then $T = T_c$ due to the successful backward execution of `xconst`. Hence, $T \sqsubseteq [\![X]\!]_{\mathcal{E}.2}(S)$, since $[\![X]\!]_{\mathcal{E}.2}(S) = T_c$.
– Case $X =$ `xvar` *Var*. Suppose $\mathcal{E} = \mathcal{E}_1, Var \mapsto (S_1, S_2), \mathcal{E}_2$ and $Var \notin Dom(\mathcal{E}_2)$. Then, $[\![X]\!]_{\mathcal{E}}(S,T) = (S, \mathcal{E}')$, where $\mathcal{E}' = \mathcal{E}_1, Var \mapsto (S_1, \mathtt{mg}(S_2,T)), \mathcal{E}_2$. Hence, $[\![X]\!]_{\mathcal{E}'.2}(S) = T'$, where $T' = \mathtt{mg}(S_2, T)$. Thus, the claim holds by Lemma 9.
– Case $X =$ `xchild`. Suppose $S = \texttt{<}tag^{(w,o)}\texttt{>}[S_1]$. Then, $[\![X]\!]_{\mathcal{E}}(S,T) = (\texttt{<}tag^{(w,o)}\texttt{>}[T], \mathcal{E})$. Thus, $[\![X]\!]_{\mathcal{E}.2}(\texttt{<}tag^{(w,o)}\texttt{>}[T]) = T$, and the claim holds since $T \sqsubseteq T$.
– Case $X =$ `xsetcnt` X'. Suppose $S = \texttt{<}tag^{(w,o)}\texttt{>}[S_1]$. If $[\![X]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$, then T must have the form $\texttt{<}tag^{(w',o)}\texttt{>}[T_1]$, and $[\![X']\!]_{\mathcal{E}}(S_1,T_1) = (S_1', \mathcal{E}')$. So $S' = \texttt{<}tag^{(w',o)}\texttt{>}[S_1']$. Let $[\![X']\!]_{\mathcal{E}'.2}(S_1') = T_1'$. By the induction hypothesis on X', $T_1 \sqsubseteq T_1'$. If $T' = [\![X]\!]_{\mathcal{E}'.2}(S')$, that is, $T' = \texttt{<}tag^{(w',o)}\texttt{>}[T_1']$, then $T \sqsubseteq T'$.
– Case $X = X_1; X_2$. Suppose $[\![X_1; X_2]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$. Then, there must exist S'' and $\mathcal{E}''$, such that $[\![X_2]\!]_{\mathcal{E}}([\![X_1]\!]_{\mathcal{E}.1}(S), T) = (S'', \mathcal{E}'')$ and $[\![X_1]\!]_{\mathcal{E}''}(S, S'') = (S', \mathcal{E}')$. By the induction hypothesis on X_2, we have $T \sqsubseteq [\![X_2]\!]_{\mathcal{E}''.2}(S'')$. Let $[\![X_1]\!]_{\mathcal{E}'.2}(S') = S'''$. By the induction hypothesis on X_1, we have $S'' \sqsubseteq S'''$. According to Lemma 10, we have $\mathcal{E}'' \sqsubseteq \mathcal{E}'$, so if $[\![X_2]\!]_{\mathcal{E}''.2}(S'') \sqsubseteq [\![X_2]\!]_{\mathcal{E}'.2}(S''')$, then the claim holds for this case by Lemma 7. We will prove $[\![X_2]\!]_{\mathcal{E}''.2}(S'') \sqsubseteq [\![X_2]\!]_{\mathcal{E}'.2}(S''')$ under the assumptions $S'' \sqsubseteq S'''$ and $\mathcal{E}'' \sqsubseteq \mathcal{E}'$ by induction on the structure of X_2. This result is also used by `xmap` and parallel composition cases below.
  ⋆ Case $X_2 =$ `xid`. $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = S''$ and $[\![X_2]\!]_{\mathcal{E}'.2}(S''') = S'''$. The claim follows the assumption $S'' \sqsubseteq S'''$.
  ⋆ Case $X_2 =$ `xconst` T_c. We have $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = T_c$ and $[\![X_2]\!]_{\mathcal{E}'.2}(S''') = T_c$. The claim holds by Lemma 3.
  ⋆ Case $X_2 =$ `xvar` *Var*. Suppose $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = S_1$. Then, $\mathcal{E}''$ must have the form $\mathcal{E}_1'', Var \mapsto (S_0, S_1), \mathcal{E}_2''$ and $Var \notin Dom(\mathcal{E}_2'')$. Since $\mathcal{E}'' \sqsubseteq \mathcal{E}'$, $\mathcal{E}'$ has the form

$\mathcal{E}'_1, Var \mapsto (S_0, S_2), \mathcal{E}'_2$, where $S_1 \sqsubseteq S_2$ and $Var \notin Dom(\mathcal{E}'_2)$. Hence, $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = S_2$, and the claim follows.

- Case $X_2 = \texttt{xchild}$. Suppose $S'' = \texttt{<}tag^{(w,o)}\texttt{>}[T'']$. Then, since $S'' \sqsubseteq S'''$, S'' must be $\texttt{<}tag^{(w',o')}\texttt{>}[T''']$, where $T'' \sqsubseteq T'''$. Since $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = T''$ and $[\![X_2]\!]_{\mathcal{E}'.2}(S''') = T'''$, the proof is done.
- Case $X_2 = \texttt{xsetcnt}\ X'$. Let $S'' = \texttt{<}tag^{(w'',o'')}\texttt{>}[T'']$ and $S''' = \texttt{<}tag^{w',o')}\texttt{>}[T']$. Since $S'' \sqsubseteq S'''$, we have $T'' \sqsubseteq T'$. Then, $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = \texttt{<}tag^{(w'',o'')}\texttt{>}[[\![X']\!]_{\mathcal{E}''.2}(T'')]$ and $[\![X_2]\!]_{\mathcal{E}'.2}(S''') = \texttt{<}tag^{(w',o')}\texttt{>}[[\![X']\!]_{\mathcal{E}'.2}(T')]$. By the induction hypothesis on X', we get $[\![X']\!]_{\mathcal{E}''.2}(T'') \sqsubseteq [\![X']\!]_{\mathcal{E}'.2}(T')$. Thus, the claim holds, that is $\texttt{<}tag^{(w'',o'')}\texttt{>}[[\![X']\!]_{\mathcal{E}''.2}(T'')] \sqsubseteq \texttt{<}tag^{(w',o')}\texttt{>}[[\![X']\!]_{\mathcal{E}'.2}(T')]$, where the relation between $tag^{(w'',o'')}$ and $tag^{(w',o')}$ is derived from $S'' \sqsubseteq S'''$.
- Case $X_2 = X'_1; X'_2$. Our goal is to prove $[\![X'_1; X'_2]\!]_{\mathcal{E}''.2}(S'') \sqsubseteq [\![X'_1; X'_2]\!]_{\mathcal{E}'.2}(S''')$. This proof is performed by applying the induction hypothesis on X'_1, and then on X'_2.
- Case $X_2 = X'_1 || X'_2$. Let $[\![X'_1]\!]_{\mathcal{E}''.2}(S'') = T''_1$ and $[\![X'_1]\!]_{\mathcal{E}'.2}(S''') = T'_1$. By the induction hypothesis on X'_1, we have $T''_1 \sqsubseteq T'_1$. Let $[\![X'_2]\!]_{\mathcal{E}''.2}(S'') = T''_2$ and $[\![X'_2]\!]_{\mathcal{E}'.2}(S''') = T'_2$. By the induction hypothesis on X_2, we have $T''_2 \sqsubseteq T'_2$. So by Lemma 8, we have $T''_1, T''_2 \sqsubseteq T'_1, T'_2$.
- Case $X_2 = \texttt{xmap}\ X'$. If $S'' = ()$, then $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = ()$, which implies the claim; if $S'' = v'', S''_1$, then by Lemma 4, there must be a subsequence of S''', say v', S'_1, such that $v'' \sqsubseteq v'$ and $S''_1 \sqsubseteq S'_1$. Let S''' has the form S'_0, v', S'_1, S'_2. By the induction hypothesis on X', we have $[\![X']\!]_{\mathcal{E}''.2}(v'') \sqsubseteq [\![X']\!]_{\mathcal{E}'.2}(v')$ and $[\![X']\!]_{\mathcal{E}''.2}(S''_1) \sqsubseteq [\![X']\!]_{\mathcal{E}'.2}(S'_1)$. Hence, by Lemma 8, $[\![X']\!]_{\mathcal{E}''.2}(v''), [\![X']\!]_{\mathcal{E}''.2}(S''_1) \sqsubseteq [\![X']\!]_{\mathcal{E}'.2}(v'), [\![X']\!]_{\mathcal{E}'.2}(S'_1)$. Let $[\![X_2]\!]_{\mathcal{E}'.2}(S''') = [\![X']\!]_{\mathcal{E}'.2}(S'_0), [\![X']\!]_{\mathcal{E}'.2}(v'), [\![X']\!]_{\mathcal{E}'.2}(S'_1), [\![X']\!]_{\mathcal{E}'.2}(S'_2)$ and $[\![X_2]\!]_{\mathcal{E}''.2}(S'') = [\![X']\!]_{\mathcal{E}''.2}(v''), [\![X']\!]_{\mathcal{E}''.2}(S''_1)$. Finally, by Lemma 6, $[\![X_2]\!]_{\mathcal{E}''.2}(S'') \sqsubseteq [\![X_2]\!]_{\mathcal{E}'.2}(S''')$.
- Case $X_2 = \texttt{xif}\ P\ X'_1\ X'_2$. For this case, the main point is to prove that $[\![X_2]\!]_{\mathcal{E}''.2}(S'')$ and $[\![X_2]\!]_{\mathcal{E}'.2}(S''')$ choose the same branch to execute as $[\![X_2]\!]_{\mathcal{E}.2}([\![X_1]\!]_{\mathcal{E}.1}(S), T)$. And thus, the claim can be proved by the induction hypotheses on X'_1 and X'_2. To prove this goal, we need to perform a case analysis on P. In the following, only the proof for $\texttt{xeq}\ X'$ is shown. Note that we are performing two proofs both by structural induction, with case analysis of X and X_2, respectively, and the corresponding induction hypothesis is referred to as main induction hypothesis or auxiliary induction hypothesis.

  Suppose $[\![X_2]\!]_{\mathcal{E}.2}([\![X_1]\!]_{\mathcal{E}.1}(S), T)$ chooses the branch X'_1. Then, $[\![P]\!]_{\mathcal{E}.1}([\![X_1]\!]_{\mathcal{E}.1}(S))$ returns $\texttt{true}^{(\texttt{ori},\texttt{c})}$, that is $[\![X_1]\!]_{\mathcal{E}.1}(S) = str^{(u,o)}$ and $[\![X']\!]_{\mathcal{E}.1}(()) = str^{(u',o')}$. Recall that $[\![X_2]\!]_{\mathcal{E}.2}(str^{(u,o)}, T) = (S'', \mathcal{E}'')$, and during this backward execution, X'_1 is executed backward and then its result is guarded by $\texttt{xeq}$. Let $[\![X'_1]\!]_{\mathcal{E}}(str^{(u,o)}, T) = (S''_1, \mathcal{E}''_1))$. Then, $S'' = \texttt{guard}(S''_1, str)$, that is, if not $\texttt{fail}$, S'' must be $str^{(u'',\texttt{s})}$ ($u'' \neq \texttt{ori}$) or $str^{(\texttt{ori},\texttt{c})}$. Since $S'' \sqsubseteq S'''$ and S''' is used as the argument of $\texttt{xeq}$, S''' must be S''. On the other hand, $\mathcal{E}''$ is from $[\![X']\!]_{\mathcal{E}''_1}((), \texttt{guard}(str^{(u',o')}, str)) = ((), \mathcal{E}'')$. By the main induction hypothesis on X', we have $\texttt{guard}(str^{(u',o')}, str) \sqsubseteq [\![X']\!]_{\mathcal{E}''.2}(())$. Since $\texttt{guard}(str^{(u',o')}, str)$ is either $str^{(u'',\texttt{s})}$ ($u'' \neq \texttt{ori}$) or $str^{(\texttt{ori},\texttt{c})}$ and $[\![X']\!]_{\mathcal{E}''.2}(())$ is used as the argument of $\texttt{xeq}$, $[\![X']\!]_{\mathcal{E}''.2}(())$ must equal to $\texttt{guard}(str^{(u',o')}, str)$. Since $\mathcal{E}'' \sqsubseteq \mathcal{E}'$, by the auxiliary induction hypothesis on X', $[\![X']\!]_{\mathcal{E}''.2}(()) \sqsubseteq [\![X']\!]_{\mathcal{E}'.2}(())$, that is $[\![X']\!]_{\mathcal{E}'.2}(())$ must equal to $[\![X']\!]_{\mathcal{E}''.2}(())$ since $[\![X']\!]_{\mathcal{E}'.2}(())$ is used as the argument of $\texttt{xeq}$. Hence, both $[\![P]\!]_{\mathcal{E}''.2}(S'')$ and $[\![P]\!]_{\mathcal{E}'.2}(S''')$ return $\texttt{true}^{(\texttt{ori},\texttt{c})}$. Similarly, if $[\![X_2]\!]_{\mathcal{E}.2}([\![X_1]\!]_{\mathcal{E}.1}(S), T)$ chooses X'_2, we can prove $[\![P]\!]_{\mathcal{E}''.2}(S'')$ and $[\![P]\!]_{\mathcal{E}'.2}(S''')$ return

$\texttt{false}^{(\texttt{ori},\texttt{c})}$.

⋆ Case $X_2 = \texttt{xlet}\ \mathit{Var}\ X'$. Let $\mathcal{E}_1 = \mathcal{E}'', \mathit{Var} \mapsto (S'', S'')$ and $\mathcal{E}_2 = \mathcal{E}', \mathit{Var} \mapsto (S''', S''')$. Since $S'' \sqsubseteq S'''$ and $\mathcal{E}'' \sqsubseteq \mathcal{E}'$, we have $\mathcal{E}_1 \sqsubseteq \mathcal{E}_2$. By the induction hypothesis on X', we get $[\![X']\!]_{\mathcal{E}_1.2}(()) \sqsubseteq [\![X']\!]_{\mathcal{E}_2.2}(())$, that yields the claim.

⋆ Case $X_2 = \texttt{xfunapp}\ \mathit{fname}\ [X'_1, ..., X'_n]$. This construct is defined as a sequential composition of transformation constructs, so the proof for the sequential composition construct applies.

– Case $X = X_1||X_2$. Suppose $[\![X_1||X_2]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$. Then, there must exist T_1, T_2 and $\mathcal{E}''$, such that $T = \overline{T_1, T_2}$, $[\![X_2]\!]_{\mathcal{E}}(S, T_2) = (S'_2, \mathcal{E}'')$ and $[\![X_1]\!]_{\mathcal{E}''}(S, T_1) = (S'_1, \mathcal{E}')$, where $\mathcal{E}'' \sqsubseteq \mathcal{E}'$ from Lemma 10. Thus, $S' = \texttt{mg}(S'_1, S'_2)$, and by Lemma 9, $S'_1 \sqsubseteq S'$ and $S'_2 \sqsubseteq S'$. By the induction hypothesis on X_1, we have $T_1 \sqsubseteq [\![X_1]\!]_{\mathcal{E}'.2}(S'_1)$, and by the induction hypothesis on X_2, we have $T_2 \sqsubseteq [\![X_2]\!]_{\mathcal{E}''.2}(S'_2)$. The current goal is to prove $[\![X_1]\!]_{\mathcal{E}'.2}(S'_1) \sqsubseteq [\![X_1]\!]_{\mathcal{E}'.2}(S')$ and $[\![X_2]\!]_{\mathcal{E}''.2}(S'_2) \sqsubseteq [\![X_2]\!]_{\mathcal{E}'.2}(S')$. If this goal is proved, then by Lemma 7 and Lemma 8, we can prove the claim $T \sqsubseteq [\![X_1||X_2]\!]_{\mathcal{E}'.2}(S')$. The proof for the goal has been done at the case where $X = X_1; X_2$.

– Case $X = \texttt{xmap}\ X'$. We do a case analysis on the structure of S. If $S = ()$, then $T = ()$ and $[\![X]\!]_{\mathcal{E}}(S,T) = ((), \mathcal{E})$, so $T \sqsubseteq [\![X]\!]_{\mathcal{E}.2}(())$.

If $S = v_1, ..., v_n$, then T must have the form $T_1, ..., T_n$, where T_i is the updated view of v_i $(1 \leq i \leq n)$. Suppose $[\![\texttt{xmap}\ X']\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$. Then the `iter` operator implements $[\![X']\!]_{\mathcal{E}_{i-1}}(v_i, T_i) = (v'_i, \mathcal{E}_i)$, where $\mathcal{E}_0 = \mathcal{E}$, $\mathcal{E}_n = \mathcal{E}'$ and $S' = v'_1, ..., v'_n$. By Lemma 10, $\mathcal{E}_{i-1} \sqsubset \mathcal{E}_i$. Let $[\![X']\!]_{\mathcal{E}_i.2}(v'_i) = T'_i$. By the induction hypothesis on X', we get $T_i \sqsubseteq T'_i$. The remaining work is to prove $[\![X']\!]_{\mathcal{E}_i.2}(v'_i) \sqsubseteq [\![X']\!]_{\mathcal{E}_n.2}(v'_i)$. If this proof can be done, then $T_i \sqsubseteq [\![X']\!]_{\mathcal{E}_n.2}(v'_i)$, so $T_1, ..., T_n \sqsubseteq [\![X']\!]_{\mathcal{E}_n.2}(v'_1), ..., [\![X']\!]_{\mathcal{E}_n.2}(v'_n)$, that is, $T \sqsubseteq [\![X]\!]_{\mathcal{E}'.2}(S')$. Proving $[\![X']\!]_{\mathcal{E}_i.2}(v'_i) \sqsubseteq [\![X']\!]_{\mathcal{E}_n.2}(v'_i)$ under the assumption that $calE_i \sqsubseteq calE_n$ has been done at the case where $X = X_1; X_2$.

– Case $X = \texttt{xif}\ P\ X_1\ X_2$. We perform a case analysis on P. In the following, only the case where $P = \texttt{xeq}\ X'$ is given, and other cases of P are proved similarly. Let $[\![X]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$. The point for this case is to prove $[\![X]\!]_{\mathcal{E}'.2}(S')$ chooses the same branch X_1 or X_2 as $[\![X]\!]_{\mathcal{E}}(S,T)$. If this proof is done, then the claim follows from the induction hypotheses on X_1 and X_2.

Suppose $[\![X]\!]_{\mathcal{E}}(S,T)$ chooses X_1. Then, $S = \mathit{str}^{(u,o)}$ and $[\![X']\!]_{\mathcal{E}.1}(()) = \mathit{str}^{(u',o')}$. Let $[\![X_1]\!]_{\mathcal{E}}(S,T) = (S'_1, \mathcal{E}'_1)$. Then, $S' = \texttt{guard}(S'_1, \mathit{str})$, that is, if not `fail`, S' must be $\mathit{str}^{(u'',\texttt{s})}$ $(u'' \neq \texttt{ori})$ or $\mathit{str}^{(\texttt{ori},\texttt{c})}$, and $\mathcal{E}'$ comes from $[\![X']\!]_{\mathcal{E}'_1}((), \texttt{guard}(\mathit{str}^{(u',o')}, \mathit{str})) = ((), \mathcal{E}')$. By the induction hypothesis on X', $\texttt{guard}([\![X']\!]_{\mathcal{E}'_1.1}(()), \mathit{str}) \sqsubseteq [\![X']\!]_{\mathcal{E}'.2}(())$. Note that $[\![X']\!]_{\mathcal{E}'_1.1}(()) = [\![X']\!]_{\mathcal{E}.1}(())$. Since $\texttt{guard}([\![X']\!]_{\mathcal{E}'_1.1}(()), \mathit{str})$ is either $\mathit{str}^{(u'',\texttt{s})}$ $(u'' \neq \texttt{ori})$ or $\mathit{str}^{(\texttt{ori},\texttt{c})}$, $[\![X']\!]_{\mathcal{E}'.2}(())$ must be $\mathit{str}^{(u'',\texttt{s})}$ $(u'' \neq \texttt{ori})$ or $\mathit{str}^{(\texttt{ori},\texttt{c})}$. Hence, S' and $[\![X']\!]_{\mathcal{E}'.2}(())$ still contain the same string *str*. Similarly, we can prove the case where $[\![X]\!]_{\mathcal{E}}(S,T)$ chooses the branch X_2.

– Case $X = \texttt{xlet}\ \mathit{Var}\ X'$. If $[\![\texttt{xlet}\ \mathit{Var}\ X']\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$, then $[\![X']\!]_{\mathcal{E}, \mathit{Var} \mapsto (S,S)}((), T) = ((), \mathcal{E}'')$, where $\mathcal{E}'' = \mathcal{E}', \mathit{Var} \mapsto (S, S')$. By the induction hypothesis on X', we have $T \sqsubseteq [\![X']\!]_{\mathcal{E}'.2, \mathit{Var} \mapsto S'}(())$. Hence, the claim $T \sqsubseteq [\![\texttt{xlet}\ \mathit{Var}\ X']\!]_{\mathcal{E}'.2}(S')$ follows.

– Case $X = \texttt{xfunapp}\ \mathit{fname}\ [X'_1, ..., X'_n]$. The claim holds for this case following the proof for the case where $X = X_1; X_2$.

□

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

Fig. 10. Syntax of the XQuery Core

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 [12]. The syntax of XQuery core is more compact. Hence, like the work [13], we implement bidirectional XQuery based on the XQuery Core syntax.

5.1. *Syntax of XQuery Core*

The syntax of the XQuery Core presented in this paper is given in Figure 10. In this syntax, the XPath axes, `child`, `descendant` and `self`, implicitly use the reserved variable `$fs:dot` to refer to their context nodes. This syntax does not include the 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 its ancestor node. But this is not a limitation to our approach. The technique in [14] provides a way to rewrite 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 make sure the length of its argument, which is a sequence, and all items in its argument cannot be changed, otherwise the extended round-tripping property will be violated. Implementing these built-in functions is our future work.

$$
\begin{array}{ll}
[\![String]\!]_{\mathcal{I}} & = \texttt{xconst}\ String^{(\texttt{ori,c})} \\
[\![()]\!]_{\mathcal{I}} & = \texttt{xconst}\ () \\
[\![Expr_1, Expr_2]\!]_{\mathcal{I}} & = [\![Expr_1]\!]_{\mathcal{I}} || [\![Expr_2]\!]_{\mathcal{I}} \\
[\![\$Var]\!]_{\mathcal{I}} & = \texttt{xvar}\ \$Var \\
[\![\texttt{for}\ \$Var\ \texttt{in}\ Expr_1\ \texttt{return}\ Expr_2]\!]_{\mathcal{I}} & = [\![Expr_1]\!]_{\mathcal{I}}; \texttt{xmap}\ (\texttt{xlet}\ \$Var\ [\![Expr_2]\!]_{\mathcal{I}}) \\
[\![\texttt{let}\ \$Var := Expr_1\ \texttt{return}\ Expr_2]\!]_{\mathcal{I}} & = [\![Expr_1]\!]_{\mathcal{I}}; \texttt{xlet}\ \$Var\ [\![Expr_2]\!]_{\mathcal{I}} \\
[\![\texttt{if}\ (Cmp)\ \texttt{then}\ Expr_1\ \texttt{else}\ Expr_2]\!]_{\mathcal{I}} & = \texttt{xif}\ [\![Cmp]\!]_{\mathcal{I}}\ [\![Expr_1]\!]_{\mathcal{I}}\ [\![Expr_2]\!]_{\mathcal{I}} \\
[\![Expr_1 < Expr_2]\!]_{\mathcal{I}} & = \texttt{xlt}\ [\![Expr_1]\!]_{\mathcal{I}}\ [\![Expr_2]\!]_{\mathcal{I}} \\
[\![Expr_1 = Expr_2]\!]_{\mathcal{I}} & = \texttt{xeq}\ [\![Expr_1]\!]_{\mathcal{I}}\ [\![Expr_2]\!]_{\mathcal{I}} \\
[\![Expr_1 > Expr_2]\!]_{\mathcal{I}} & = \texttt{xgt}\ [\![Expr_1]\!]_{\mathcal{I}}\ [\![Expr_2]\!]_{\mathcal{I}} \\
[\![Axis\ NodeTest]\!]_{\mathcal{I}} & = [\![Axis]\!]_{\mathcal{I}}; [\![NodeTest]\!]_{\mathcal{I}} \\
[\![\texttt{child}::]\!]_{\mathcal{I}} & = \texttt{xvar}\ \$\texttt{fs:dot}; \texttt{xchild} \\
[\![\texttt{descendant}::]\!]_{\mathcal{I}} & = \texttt{xfunapp}\ \texttt{xdes}\ [\texttt{xvar}\ \$dot] \\
[\![\texttt{self}::]\!]_{\mathcal{I}} & = \texttt{xvar}\ \$\texttt{fs:dot} \\
[\![NCName]\!]_{\mathcal{I}} & = \texttt{xmap}\ (\texttt{xif}\ \texttt{xiselement}\ \texttt{xid}\ (\texttt{xconst}\ ())); \\
 & \quad \texttt{xmap}\ ((\texttt{xif}\ (\texttt{xwithtag}\ NCName)\ \texttt{xid}\ (\texttt{xconst}\ ()))) \\
[\![*]\!]_{\mathcal{I}} & = \texttt{xmap}\ (\texttt{xif}\ \texttt{xiselement}\ \texttt{xid}\ (\texttt{xconst}\ ())) \\
[\![\texttt{text}()]\!]_{\mathcal{I}} & = \texttt{xmap}\ (\texttt{xif}\ \texttt{xiselement}\ (\texttt{xconst}\ ())\ \texttt{xid}) \\
[\![\texttt{node}()]\!]_{\mathcal{I}} & = \texttt{xid} \\
[\![\texttt{element}\ NCName\ \{Expr\}]\!]_{\mathcal{I}} & = \texttt{xconst}\ \texttt{<}NCName^{(\texttt{ori,c})}\texttt{>}[()]; \texttt{xsetcnt}\ [\![Expr]\!]_{\mathcal{I}} \\
[\![NCName\ (Expr_1, ..., Expr_n)]\!]_{\mathcal{I}} & = \texttt{xfunapp}\ NCName\ [[\![Expr_1]\!]_{\mathcal{I}}, ..., [\![Expr_n]\!]_{\mathcal{I}}]
\end{array}
$$

Fig. 11. Translation of XQuery Core Expression

5.2. *The Translation*

Figure 11 gives the rules for translating XQuery Core into the bidirectional language. 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 XQuery function:

$$\texttt{function}\ NCName(\$Var_1, ..., \$Var_n)\{Expr\}$$

is translated into the following function in the bidirectional language:

$$\texttt{fun}\ NCName(\$Var_1, ..., \$Var_n) = [\![Expr]\!]_{\mathcal{I}}$$

The translation defined in Figure 11 satisfies the following property, 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 are ignored in the following theorem.

Theorem 12. Let $\mathcal{C}$ be a context that maps variables to values. If an XQuery Core expression *Expr* is evaluated to a value under $\mathcal{C}$, then the expression $[\![[\![Expr]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$ is also evaluated to the same value.

Proof. This theorem is proved by induction on the structure of the XQuery Core expression *Expr*.

- Case $Expr = String$. The expression *String* is translated into $\texttt{xconst}\ String^{(\texttt{ori,c})}$, and $[\![\texttt{xconst}\ String^{(\texttt{ori,c})}]\!]_{\mathcal{C}}(()) = String^{(\texttt{ori,c})}$. Hence, the claim follows directly.
- Case $Expr = ()$. This case is proved similarly as the above one.
- Case $Expr = Expr_1, Expr_2$. This expression is translated into $[\![Expr_1]\!]_{\mathcal{I}}||[\![Expr_2]\!]_{\mathcal{I}}$, and $[\![[\![Expr_1]\!]_{\mathcal{I}}||[\![Expr_2]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(()) = [\![[\![Expr_1]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(()), [\![[\![Expr_2]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$. By the induction hypothesis on $Expr_1$, we know $Expr_1$ and $[\![[\![Expr_1]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$ have the same value, and similarly $Expr_2$ and $[\![[\![Expr_2]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$ also have the same value. So $[\![[\![Expr_1]\!]_{\mathcal{I}}||[\![Expr_2]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$ and *Expr* have the same value.
- Case $Expr = \$Var$. The claim holds since both $\$Var$ and $\texttt{xvar}\ \$Var$ returns the value of the variable $\$Var$ in the context $\mathcal{C}$.
- Case $Expr = \texttt{for}\ \$Var\ \texttt{in}\ Expr_1\ \texttt{return}\ Expr_2$. This expression is translated into $[\![Expr_1]\!]_{\mathcal{I}}; \texttt{xmap}\ (\texttt{xlet}\ \$Var\ [\![Expr_2]\!]_{\mathcal{I}})$. By the induction hypothesis on $Expr_1$, we know $Expr_1$ and $[\![[\![Expr_1]\!]_{\mathcal{I}}]\!]_{C}(())$ have the same value, say S. If S is an empty sequence, then both *Expr* and its translated expression returns an empty sequence. Otherwise, suppose $S = v_1, ..., v_n$. Then, each v_i $(1 \leq i \leq n)$ will cause $Expr_2$ to execute one time under the context $\overline{\mathcal{C}, \$Var \mapsto v_i}$, or $\texttt{xlet}\ \$Var\ [\![Expr_2]\!]_{\mathcal{I}}$ to execute one time under the context $\mathcal{C}$ with the source data v_i. Both executions will return the same value according to the forward semantics of `xlet` and the induction hypothesis on $Expr_2$.
- Case $Expr = \texttt{let}\ \$Var := Expr_1\ \texttt{return}\ Expr_2$. This case is proved similarly as the `for` expression.
- Case $Expr = \texttt{if}\ (Cmp)\ \texttt{then}\ Expr_1\ \texttt{else}\ Expr_2$. This expression is translated into $\texttt{xif}\ [\![Cmp]\!]_{\mathcal{I}}\ [\![Expr_1]\!]_{\mathcal{I}}\ [\![Expr_2]\!]_{\mathcal{I}}$. We do a case analysis on *Cmp*. Suppose *Cmp* is

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

Fig. 12. Syntax of Types

$Expr'_1 = Expr'_2$, which is translated into $\texttt{xeq}\ [\![Expr'_1]\!]_{\mathcal{I}}\ [\![Expr'_2]\!]_{\mathcal{I}}$. By the induction hypotheses on $Expr'_1$ and $Expr'_2$, we know $Expr'_1 = Expr'_2$ and $[\![\texttt{xeq}\ [\![Expr'_1]\!]_{\mathcal{I}}\ [\![Expr'_2]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$ have the same value. Hence, the `if` expression and its translated expression choose the same branch to execute. Then, by the the induction hypotheses on $Expr_1$ and $Expr_2$, respectively, we know on each branch the `if` expression and its translated expression have the same execution result. It is proved similarly for the cases where *Cmp* is $Expr'_1 < Expr'_2$ and *Cmp* is $Expr'_1 > Expr'_2$.

- Case $Expr = Axis\ NodeTest$. We prove the claim by assuming that *Axis* is `child` and *NodeTest* is *NCName*, and other combinations of *Axis* and *NodeTest* are proved similarly. The expression `xchild`::*NCName* first gets the content of the current context item, and then returns all elements with the tag *NCName* from the content. Suppose the current context item is bound to the variable `$fs:dot`. The expression `xchild`::*NCName* is translated into $\texttt{xvar}\ \$\texttt{fs:dot}; \texttt{xchild}; X_1; X_2$, where $X_1 = \texttt{xmap}\ (\texttt{xif xiselement xid}\ (\texttt{xconst}\ ()))$ and $X_2 = \texttt{xmap}\ ((\texttt{xif}\ P\ \texttt{xid}\ (\texttt{xconst}\ ())))$ with $P = \texttt{xwithtag}\ NCName$. In the translated expression, $\texttt{xvar}\ \$\texttt{fs:dot}; \texttt{xchild}$ first returns the content of the current context item, and then X_1 filters out those content other than elements, and X_2 is to keep only elements with the tag *NCName*. So the translated expression returns the same value as the expression `xchild`::*NCName*.
- Case $Expr = \texttt{element}\ NCName\ \{Expr'\}$. The expression *Expr* is translated into the expression $\texttt{xconst}\ \texttt{<}NCName^{(\texttt{ori,c})}\texttt{>}[]; \texttt{xsetcnt}\ [\![Expr']\!]_{\mathcal{I}}$. By the induction hypothesis on $Expr'$, we can see the `element` expression and its translated expression construct the same element.
- Case $Expr = NCName\ (Expr_1, ..., Expr_n)$. The translated expression is also a function application $\texttt{xfunapp}\ NCName\ [[\![Expr_1]\!]_{\mathcal{I}}, ..., [\![Expr_n]\!]_{\mathcal{I}}]$. By the induction hypotheses on $Expr_i$ $(1 \leq i \leq n)$, the argument expression $Expr_i$ has the same value, say S_i, as its translated expression. On the other hand, suppose the function *NCName* is defined as $\texttt{function}\ NCName(\$Var_1, ..., \$Var_n)\{Expr'\}$. By the induction hypothesis on $Expr'$, the expression $Expr'$ has the same value as its translated expression under the context $\mathcal{C}, \$Var_1 \mapsto S_1, ..., \$Var_n \mapsto S_n$. That is, both function applications have the same value under the context $\mathcal{C}$.

□

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.

$$
\begin{array}{ll}
() & \in () \\
str^{(u,o)} & \in \texttt{string}^{o} \\
\texttt{<}tag^{(u,o)}\texttt{>}[S] & \in \texttt{<}tag^{o}\texttt{>}[\tau], \text{if } S \in \tau \\
S & \in \tau*, \text{if } S \in ()|\tau, \tau* \\
S_1, S_2 & \in \tau_1, \tau_2, \text{if } S_1 \in \tau_1 \text{ and } S_2 \in \tau_2 \\
S & \in \tau_1|\tau_2, \text{if } S \in \tau_1 \text{ or } S \in \tau_2 \\
S & \in \mu a.\tau, \text{if } S \in \tau[\mu a.\tau/a] \\
S & \in \lceil \tau \rceil, \text{if } S \in \tau
\end{array}
$$

Fig. 13. Syntax of Types

6.1. *Syntax of Types*

The syntax of types is given in Figure 12, which is almost the regular expression types in [15] except for the boxed type $\lceil \tau \rceil$ and the origin annotation o on the string type and element type. This origin annotation is still either `s` or `c`. The boxed type will be introduced later when we discuss the typing rule for `xmap`. The notation $S \in \tau$ means S has the type τ, defined in Figure 13. The recursive type $\mu a.\tau$ is regarded as equivalent to its unfolded form $\tau[\mu a.\tau/a]$, where 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 this paper. The boxed type does not affect the relation between values and types, so it is also ignored where possible.

6.2. *Typing Rules*

The typing rules for the bidirectional transformation language are defined in Figure 14. 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.

In the typing rule for `xconst` S, the operation $\texttt{mkty}(S)$ makes a type, say τ', from S: if $S = ()$, then $\tau' = ()$; if $S = str^{(\texttt{ori,c})}$, then $\tau' = \texttt{string}^{\texttt{c}}$; if $S = \texttt{<}tag^{(\texttt{ori,c})}\texttt{>}[()]$, then $\tau' = \texttt{<}tag^{\texttt{c}}\texttt{>}[()]$. Hence, $S \in \texttt{mkty}(S)$.

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 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. Some typing rules are explained below.

The typing rule of `xmap` depend on the typing procedure defined in Figure 15. The

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

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

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

$$\frac{\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]}{\Gamma \vdash \texttt{xchild} : \tau \leftrightarrow \tau_1|...|\tau_n \Rightarrow \texttt{xchild}^{\tau}}$$

$$\frac{\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau']|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau'] \qquad \Gamma \vdash X : \tau' \leftrightarrow \tau'' \Rightarrow X'}{\Gamma \vdash \texttt{xsetcnt}\ X : \tau \leftrightarrow \texttt{<}tag_1^{o_1}\texttt{>}[\tau'']|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau''] \Rightarrow \texttt{xsetcnt}\ X'}$$

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

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

$$\frac{\Gamma \vdash_{\texttt{m}} \texttt{xmap}\ X : \tau \leftrightarrow \tau' \Rightarrow \tau'' \quad \Gamma \vdash X : \tau'' \leftrightarrow \tau''' \Rightarrow X'}{\Gamma \vdash \texttt{xmap}\ X : \tau \leftrightarrow \tau' \Rightarrow \texttt{xmap}_{\tau}^{\tau'}\ X'}$$

$$\frac{\Gamma \vdash P : \tau \leftrightarrow \tau_P \Rightarrow P' \quad \Gamma \vdash X_1 : \texttt{T}(\tau, P) \leftrightarrow \tau_1 \Rightarrow X_1' \quad \Gamma \vdash X_2 : \texttt{F}(\tau, P) \leftrightarrow \tau_2 \Rightarrow X_2'}{\Gamma \vdash \texttt{xif}\ P\ X_1\ X_2 : \tau \leftrightarrow \tau_1|\tau_2 \Rightarrow \texttt{xif}_{\tau_1}^{\tau_2}\ P\ X_1'\ X_2'}$$

$$\frac{\tau = \texttt{string}^{o_1}|...|\texttt{string}^{o_n} \qquad \Gamma \vdash X : () \leftrightarrow \texttt{string}^{o_1'}|...|\texttt{string}^{o_n'} \Rightarrow X'}{\Gamma \vdash \texttt{xeq}\ X : \tau \leftrightarrow \texttt{true}^{\texttt{c}}|\texttt{false}^{\texttt{c}} \Rightarrow \texttt{xeq}\ X'}$$

$$\frac{\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n\texttt{>}[\tau_n^{o_n}]}{\Gamma \vdash \texttt{xwithtag}\ str : \tau \leftrightarrow \texttt{true}^{\texttt{c}}|\texttt{false}^{\texttt{c}} \Rightarrow \texttt{xwithtag}\ str}$$

$$\frac{\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n\texttt{>}[\tau_n^{o_n}]|\texttt{string}^{o_1'}|...|\texttt{string}^{o_m'}}{\Gamma \vdash \texttt{xiselement} : \tau \leftrightarrow \texttt{true}^{\texttt{c}}|\texttt{false}^{\texttt{c}} \Rightarrow \texttt{xiselement}}$$

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

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

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

Fig. 14. Typing Rules

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

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

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

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

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

Fig. 15. Typing Rules for `xmap`

transformation `xmap` applies its argument transformation X to each single value in the source data. Correspondingly, the procedure in Figure 15 identifies each `string` and element type in the source-data type of `xmap`, and then uses this `string` or element type as the source-data type to check X, resulting in a boxed type. That is, all values matched with this boxed types have the same string or element as their source data. The view type of `xmap` is its source-data type with each `string` or element type replaced with its corresponding boxed type. The rules in Figure 15 also collect all top-level `string` and element types in the source-data type of `xmap` and represent them by a choice type. This choice type will then be used to check X again, so that X is annotated with the type information from all possible `string` and element types.

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-data type is `<bag>[<apple>[string]*]`, then the view type of `xmap` is $\lceil$`<apple>[string]*`$\rceil$; if the source-data type is `<bag>[<apple>[string]]*`, then its view type is $\lceil$`<apple>[string]`$\rceil$`*`. 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.

In the typing rule for the transformation $X_1||X_2$, the view types of X_1 and X_2 are boxed first, and then composed together as the view type of $X_1||X_2$. So the boxed types in this typing rule can be used to determine whether values are computed by X_1 or X_2. The revised split operator in the next section depends on boxed types to split views for the backward executions of `xmap` X and $X_1||X_2$.

The typing rule of `xif` checks its two branches under the source-data type computed by $\mathtt{T}(\tau, P)$ and $\mathtt{F}(\tau, P)$, respectively, which are defined in Figure 16. These operators generate more precise source-data types for each branch. The following example shows

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

Fig. 16. The operator T and F

this feature is useful. In this example, suppose we have the code `xif` (`xwithtag` "book") `xchild` (`xconst` ()) and the source-data type `<book>`[`string`]|`string`. If the source-data 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 source data. Actually, if the true branch is chosen at runtime, we know the `xwithtag` predicate must hold, so the source data of this branch must be an element. Here are some brief explanation of these two operators: the operator $\mathtt{T}(\tau, \mathtt{xwithtag}\ str)$ selects in τ the element types with the tag *str*, and the operator $\mathtt{F}(\tau, \mathtt{xwithtag}\ str)$ does the reverse selection; the operator $\mathtt{T}(\tau, \mathtt{xiselement})$ selects the element types in τ, while the operator $\mathtt{F}(\tau, \mathtt{xiselement})$ selects the string types; both operators $\mathtt{T}(\tau, \mathtt{xeq})$ and $\mathtt{F}(\tau, \mathtt{xeq})$ return the source-data type without further selection since the source-data type of `xeq` contains only string types.

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.

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 context $\mathcal{C}$. In the following definition, we define the well-typed evaluation context $\mathcal{C}$ with respect to a typing context Γ, represented by $\mathcal{C} \in \Gamma$, which is needed by the type-soundness property.

Definition 2. Given the evaluation context $\mathcal{C}$ and the typing context Γ, we say $\mathcal{C} \in \Gamma$, if 1) $\mathcal{C} = \cdot$ and $\Gamma = \cdot$; or 2) $\mathcal{C} = \mathcal{C}', Var \mapsto S$ and $\Gamma = \Gamma', Var \mapsto \tau$, such that $\mathcal{C}' \in \Gamma'$ and $S \in \tau$.

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

Theorem 13. Given a transformation X, a context $\mathcal{C}$ and a source value S, if $\Gamma \vdash X : \tau \leftrightarrow \tau' \Rightarrow X'$, $\mathcal{C} \in \Gamma$ and $S \in \tau$, then $[\![X']\!]_{\mathcal{C}}(S) = T$ and $T \in \tau'$.

Proof. This proof is performed by structural induction on X. X and X' differ only in type annotations, which do not affect the forward execution result. For simplicity, we take the simplified typing judgment $\Gamma \vdash X : \tau \leftrightarrow \tau'$, and prove $[\![X]\!]_{\mathcal{C}}(S) = T$ and $T \in \tau'$.

- Case $X = \texttt{xid}$. By the typing rule for `xid`, we have $\tau' = \tau$. So $[\![X]\!]_{\mathcal{C}}(S) = S$ and $S \in \tau$.
- Case $X = \texttt{xconst}\ T_c$. By the typing rule for `xconst`, $\tau' = \texttt{mkty}(T_c)$. Hence, $[\![X]\!]_{\mathcal{C}}(S) = T_c$ and $T_c \in \tau'$.
- Case $X = \texttt{xvar}\ Var$. If $\Gamma \vdash X : \tau \leftrightarrow \tau'$, then we know $\Gamma = \Gamma_1, Var \mapsto \tau', \Gamma_2$, where $Var \notin Dom(\Gamma_2)$. Since $\mathcal{C} \in \Gamma$, we have $\mathcal{C} = \mathcal{C}_1, Var \mapsto T, \Gamma_2$, where $Var \notin Dom(\Gamma_2)$ and $T \in \tau'$. Hence, $[\![X]\!]_{\mathcal{C}}(S) = T$.
- Case $X = \texttt{xchild}$. If $\Gamma \vdash X : \tau \leftrightarrow \tau'$, then τ has the form $\texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]$ and τ' is in the form $\tau_1|...|\tau_n$. By assumption, $S \in \tau$. Then, S must be an element, and there exists i $(1 \le i \le n)$, such that $S \in \texttt{<}tag_i^{o_i}\texttt{>}[\tau_i]$. Let S be $\texttt{<}tag_i^{(w,o_i)}\texttt{>}[S']$. Hence, we get $[\![X]\!]_{\mathcal{C}}(S) = S'$ and $S' \in \tau_i$.
- Case $X = \texttt{xsetcnt}\ X''$. If $\Gamma \vdash X : \tau \leftrightarrow \tau'$, then $\tau = \texttt{<}tag_1^{o_1}\texttt{>}[\tau_s]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_s]$ and $\tau' = \texttt{<}tag_1^{o_1'}\texttt{>}[\tau'']|...|\texttt{<}tag_n^{o_n'}\texttt{>}[\tau'']$. Since $S \in \tau$, it must have the form $\texttt{<}tag_i^{(w,o_i)}\texttt{>}[S']$ for some i $(1 \le i \le n)$, and $S' \in \tau_s$. By the induction hypothesis on X'', we know if $\Gamma \vdash X'' : \tau_s \leftrightarrow \tau''$, $S' \in \tau_s$ and $\mathcal{C} \in \Gamma$, then $[\![X'']\!]_{\mathcal{C}}(S') = T''$ and $T'' \in \tau''$. Hence, $[\![X]\!]_{\mathcal{C}}(S) = \texttt{<}tag_i^{(w,o_i)}\texttt{>}[T'']$ and $\texttt{<}tag_i^{o_i}\texttt{>}[T''] \in \tau'$.
- Case $X = X_1; X_2$. If $\Gamma \vdash X : \tau \leftrightarrow \tau'$, then we have $\Gamma \vdash X_1 : \tau \leftrightarrow \tau_1$ and $\Gamma \vdash X_2 : \tau_1 \leftrightarrow \tau'$. Suppose $S \in \tau$ and $\mathcal{C} \in \Gamma$. Then, the proof is carried out by proving $[\![X_1]\!]_{\mathcal{C}}(S) =$

S_1' and $S_1' \in \tau_1$ with the induction hypothesis on X_1, and then proving $[\![X_2]\!]_{\mathcal{C}}(S_1') = T$ and $T \in \tau'$ with the induction hypothesis on X_2.

- Case $X = X_1||X_2$. The case is proved similarly as the case where $X = X_1; X_2$.
- Case $X = \texttt{xmap}\ X''$. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, by the typing rule for `xmap`, we know $\Gamma \vdash_{\texttt{m}} \texttt{xmap}\ X : \tau \leftrightarrow \tau'$. In the following, we prove $[\![X]\!]_{\mathcal{C}}(S) \in \tau'$ by induction on the source type τ.

  If $\tau = ()$, then $\tau' = ()$ and S must be (). Hence, $[\![X]\!]_{\mathcal{C}}(S) = ()$, and $[\![X]\!]_{\mathcal{C}}(S) \in ()$.

  If $\tau \in \{\texttt{string}^o, \texttt{<}tag^{o'}\texttt{>}[\tau_1]\}$, then the judgment $\Gamma \vdash X'' : \tau \leftrightarrow \tau'$ can be derived. By the induction hypothesis on X'', we get $[\![X'']\!]_{\mathcal{C}}(S) \in \tau'$, and since S is a string or an element, we know $[\![X]\!]_{\mathcal{C}}(S) = [\![X'']\!]_{\mathcal{C}}(S)$, so $[\![X]\!]_{\mathcal{C}}(S) \in \tau'$ and $[\![X]\!]_{\mathcal{C}}(S) \in \lceil\tau'\rceil$.

  If $\tau = \tau''*$, then $\tau' = \tau'''*$, where τ''' comes from the judgment $\Gamma \vdash_{\texttt{m}} \texttt{xmap}\ X : \tau'' \leftrightarrow \tau'''$. If $S = ()$, then the claim holds as proved in the case where $\tau = ()$. Otherwise, we assume S is not an empty sequence and has the form S_1,...,S_n. Since $S \in \tau''*$, $S_i \in \tau''$. For each i $(1 \leq i \leq n)$, by the induction hypothesis on τ'', we get $[\![X]\!]_{\mathcal{C}}(S_i) = T_i$ and $T_i \in \tau'''$. Hence, $T_1, ..., T_n \in \tau'''*$.

  If $\tau = \tau_1, \tau_2$, then we have $\tau' = \tau_1', \tau_2'$, where τ_1' and τ_2' are obtained from the judgments $\Gamma \vdash_{\texttt{m}} \texttt{xmap}\ X : \tau_1 \leftrightarrow \tau_1'$ and $\Gamma \vdash_{\texttt{m}} \texttt{xmap}\ X : \tau_2 \leftrightarrow \tau_2'$. Since $S \in \overline{\tau_1, \tau_2}$, it must have the form $\overline{S_1, S_2}$, such that $S_1 \in \tau_1$ and $S_2 \in \tau_2$. Let $T_1 = [\![X]\!]_{\mathcal{C}}(S_1)$ and $T_2 = [\![X]\!]_{\mathcal{C}}(S_2)$. By the induction hypotheses on τ_1 and τ_2, respectively, we get $T_1, T_2 \in \tau_1', \tau_2'$. Similarly, we can prove the claim for $\tau = \tau_1|\tau_2$.
- $X = \texttt{xif}\ P\ X_1\ X_2$. This case is proved with a case analysis on P. We show the proof for $P = \texttt{xwithtag}\ tag$. For this case, if $\Gamma \vdash X : \tau \leftrightarrow \tau'$, then τ must have the form $\texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]$. If $S \in \tau$, $[\![P]\!]_{\mathcal{C}}(S)$ must return correctly. And thus $[\![X]\!]_{\mathcal{C}}(S)$ will be $[\![X_1]\!]_{\mathcal{C}}(S)$ or $[\![X_2]\!]_{\mathcal{C}}(S)$. The proof is done by the induction hypotheses on X_1 and X_2.
- $X = \texttt{xlet}\ Var\ X'$. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, $\Gamma, Var \mapsto \tau \vdash X' : () \leftrightarrow \tau'$. By the induction hypothesis on X', we get $[\![X']\!]_{\mathcal{C}'}(()) \in \tau'$, where $\mathcal{C}' = \mathcal{C}, Var \mapsto S$. The proof is done since $[\![X']\!]_{\mathcal{C}'}(()) = [\![X]\!]_{\mathcal{C}}(S)$.
- $X = \texttt{xfunapp}\ fname\ [X_1, ..., X_n]$. The case is proved similarly as the `xlet` case.

□

6.4. *Backward Type Preservation*

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

Definition 3. For the evaluation context $\mathcal{E}$ and the typing context Γ, we say $\mathcal{E} \in \Gamma$, if 1) $\mathcal{E} = \cdot$ and $\Gamma = \cdot$; or 2) $\mathcal{E} = \mathcal{E}', Var \mapsto (S, S')$ and $\Gamma = \Gamma', Var \mapsto \tau$, such that $\mathcal{E}' \in \Gamma'$, $S \in \tau$ and $S' \in \tau$.

During backward transformations, the updated source data may contain values generated by the `mg` operator, which combines updates made to different replicas of same values. The following lemma says this operator does not change the type of data being merged.

Lemma 14. Suppose $S_1 \in \tau$ and $S_2 \in \tau$. If $S = \texttt{mg}(S_1, S_2)$, then $S \in \tau$.

Proof. The proof proceeds by structural induction on the structure of S_1. Some cases are proved as examples.

- Case $S_1 = ()$. If $S = \texttt{mg}(S_1, S_2)$, then S_2 must be (), and S is also (). Hence, $S \in \tau$ since $S_1 \in \tau$.
- Case $S_1 = str^{(\texttt{ori,s})}$. If $S = \texttt{mg}(S_1, S_2)$, then S_2 can be $str^{(\texttt{ori,s})}$, $str^{(\texttt{non,s})}$, $str'^{(\texttt{mod,s})}$ or $str^{(\texttt{del,s})}$. For each possibility of S_2, we prove the claim holds. If $S_2 = str^{(\texttt{ori,s})}$, then S is also the same string as S_1, so $S \in \tau$. On the other hand, if $S_2 = str'^{(\texttt{mod,s})}$ ($str^{(\texttt{non,s})}$ or $str^{(\texttt{del,s})}$), then $S = S_2$, and hence $S \in \tau$ since $S_2 \in \tau$.
- Case $S_1 = \texttt{<}tag^{(\texttt{ori,s})}\texttt{>}[S_1']$. If $S = \texttt{mg}(S_1, S_2)$, then S_2 is $\texttt{<}tag^{(\texttt{ori,s})}\texttt{>}[S_2']$, $\texttt{<}tag^{(\texttt{non,s})}\texttt{>}[S_2']$ or $\texttt{<}tag^{(\texttt{del,s})}\texttt{>}[S_2']$. Since $S_1 \in \tau$ and $S_2 \in \tau$, we know τ must have the form $\texttt{<}tag^{o}\texttt{>}[\tau']$, such that $S_1' \in \tau'$ and $S_2' \in \tau'$. We do case analysis on each possibility of S_2. If $S_2 = \texttt{<}tag^{(\texttt{ori,s})}\texttt{>}[S_2']$, then $S = \texttt{<}tag^{(\texttt{ori,s})}\texttt{>}[S']$, where $S' = \texttt{mg}(S_1', S_2')$. By the induction hypothesis on S_1', we have $S' \in \tau'$. Hence, $\texttt{<}tag^{(\texttt{ori,s})}\texttt{>}[S'] \in \tau$. For the possibility of $S_2 = \texttt{<}tag^{(\texttt{non,s})}\texttt{>}[S_2']$ (or $\texttt{<}tag^{(\texttt{del,s})}\texttt{>}[S_2']$), we have $S = \texttt{<}tag^{(\texttt{non,s})}\texttt{>}[S']$ (or $\texttt{<}tag^{(\texttt{del,s})}\texttt{>}[S']$), where $S' = \texttt{mg}(S_1', S_2')$. Similarly, by the induction hypothesis on S_1', we get $S' \in \tau'$, and hence $S \in \tau$.
- Case $S_1 = v_1, V_1$. If $S = \texttt{mg}(S_1, S_2)$, then S_2 must have the form v_2, V_2, and $S = v', V'$, where $v' = \texttt{mg}(v_1, v_2)$ and $V' = \texttt{mg}(V_1, V_2)$. Since $S_1 \in \tau$ and $S_2 \in \tau$, we know there must be three forms for τ: $\tau = \tau_1, \tau_2$, where $v_i \in \tau_1$ and $V_i \in \tau_2$ ($i \in \{1, 2\}$); $\tau = \tau'*$, where $v_i \in \tau'$ and $V_i \in \tau'*$; $\tau = \tau_1|\tau_2$, where $v_i, V_i \in \tau_1$ or $v_i, V_i \in \tau_2$. For the first form of τ, we can prove $v', V' \in \tau_1, \tau_2$ by the induction hypotheses on v_1 and V_1. Similarly, $v', V' \in \tau'*$ can be proved. The third form is proved by the induction hypotheses on τ_1 and τ_2.

□

The property of backward type preservation is stated in Theorem 15. That is, the type of the source data is respected after it is updated by backward executions of well-typed programs. This is a specific feature for the type system designed in this work. 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 15. Let $\mathcal{E} \in \Gamma$ and $S \in \tau$. If $\Gamma \vdash X : \tau \leftrightarrow \tau' \Rightarrow X'$, $T \in \tau'$ and $[\![X']\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then $S' \in \tau$ and $\mathcal{E}' \in \Gamma$.

Proof. The proof proceeds by induction on X. In this section, we prove this theorem with the assumption that updates to T are only modifications or deletions. When T include insertions, the backward semantics of some language constructs, such as `xmap` and `xchild`, will be revised, and we will discuss this theorem for insertion in the next section. If T does not include insertions, the type annotations on X' will not affect its backward execution. Hence, in the following proof, we take the simplified typing judgment $\Gamma \vdash X : \tau \leftrightarrow \tau'$, and prove the claim about $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$.

- Case $X = \texttt{xid}$. By the typing rule for `xid`, we have $\tau' = \tau$. So $[\![X]\!]_{\mathcal{E}}(S, T) = (T, \mathcal{E})$, and the claim holds by assumptions.
- Case $X = \texttt{xconst}\ T_c$. The backward execution of `xconst` does not change the source data S and the environment $\mathcal{E}$, so the claim holds.
- Case $X = \texttt{xvar}\ Var$. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, $\Gamma = \Gamma_1, Var \mapsto \tau', \Gamma_2$, where $Var \notin Dom(\Gamma)$. Since $\mathcal{E} \in \Gamma$, we know $\mathcal{E} = \mathcal{E}_1, Var \mapsto (S_1, S_2), \mathcal{E}_2$, where $Var \notin Dom(\mathcal{E}_2)$, and $S_1 \in \tau'$ and $S_2 \in \tau'$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then $S' = S$ and $\mathcal{E}' =$

$\mathcal{E}_1, Var \mapsto (S_1, \mathtt{mg}(S_2, T)), \mathcal{E}_2$. By Lemma 14, $\mathtt{mg}(S_2, T) \in \tau'$ since $S_2 \in \tau'$ and $T \in \tau'$, so $\mathcal{E}' \in \Gamma$.

- Case $X = \mathtt{xchild}$. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, $\tau = <tag_1^{o_1}>[\tau_1]|...|<tag_n^{o_n}>[\tau_n]$ and $\tau' = \tau_1|...|\tau_n$. By assumption, $S \in \tau$. Then, S must be an element, and there exists i $(1 \leq i \leq n)$, such that $S \in <tag_i^{o_i}>[\tau_i]$. Let S be $<tag_i^{(w,o_i)}>[T_s]$, where $T_s \in \tau_i$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then $\mathcal{E}' = \mathcal{E}$ and $S' = <tag_i^{(w,o_i)}>[T]$. Now we need to prove that $<tag_i^{(w,o_i)}>[T] \in <tag_i^{o_i}>[\tau_i]$. We know that T is the updated T_s, that is, it is obtained by modifying strings in T_s or deleting strings or elements in T_s by changing annotations. Hence, T has the same type τ_i as T_s, and $<tag_i^{(w,o_i)}>[T] \in <tag_i^{(w,o_i)}>[\tau_i^{o_i}]$.
- Case $X = \mathtt{xsetcnt}\ X'$. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, $\tau = <tag_1^{o_1}>[\tau_s]|...|<tag_n^{o_n}>[\tau_s]$ and $\tau' = <tag_1^{o_1}>[\tau'']|...|<tag_n^{o_n}>[\tau'']$, where τ'' is from $\Gamma \vdash X' : \tau_s \leftrightarrow \tau''$. Since $S \in \tau$, it must have the form $<tag_i^{(w,o_i)}>[S_s]$ for some i $(1 \leq i \leq n)$ and $S_s \in \tau_s$. T must have the form $<tag_i^{(w',o_i)}>[T']$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then $S' = <tag_i^{(w',o_i)}>[S'_s]$ and $[\![X']\!]_{\mathcal{E}}(S_s, T') = (S'_s, \mathcal{E}')$. By the induction hypothesis on X', we get $\mathcal{E}' \in \Gamma$ and $S'_s \in \tau_s$, so $S' \in <tag_i^{o_i}>[\tau_s]$.
- Case $X = X_1; X_2$. If $\Gamma \vdash X : \tau \leftrightarrow \tau'$, then we have $\Gamma \vdash X_1 : \tau \leftrightarrow \tau_1$ and $\Gamma \vdash X_2 : \tau_1 \leftrightarrow \tau'$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then there must S'' and $\mathcal{E}''$, such that $[\![X_2]\!]_{\mathcal{E}}([\![X_1]\!]_{\mathcal{E}.1}(S), T)$ $= (S'', \mathcal{E}'')$ and $[\![X_1]\!]_{\mathcal{E}''}(S, S'') = (S', \mathcal{E}')$. By Theorem 13, $[\![X_1]\!]_{\mathcal{E}.1}(S) \in \tau_1$, so by the induction hypothesis on X_2, we get $S'' \in \tau_1$ and $\mathcal{E}'' \in \Gamma$. Thus, the proof is done by the induction hypothesis on X_1.
- Case $X = X_1||X_2$. This case is proved similarly as the case where $X = X_1; X_2$.
- Case $X = \mathtt{xmap}\ X''$. This case is proved by induction on the source type τ. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, $\Gamma \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau \leftrightarrow \tau'$.

  If $\tau = ()$, then $\tau' = ()$, $S = ()$ and $T = ()$. Thus, $[\![X]\!]_{\mathcal{E}}((), ()) = ((), \mathcal{E})$. Hence, the claim follows directly.

  If $\tau \in \{\mathtt{string}^o, <tag^{o'}>[\tau_1]\}$, then $\Gamma \vdash X'' : \tau \leftrightarrow \tau'$. Let $[\![X'']\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$. By the induction hypothesis on X'', we get $S' \in tau$ and $\mathcal{E}' \in \Gamma$. Since S is a string or an element, we know $[\![X]\!]_{\mathcal{E}}(S, T) = [\![X'']\!]_{\mathcal{E}}(S, T)$, so the claim holds.

  If $\tau = \tau''*$, then $\tau' = \tau'''*$, where τ''' comes from $\Gamma \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau'' \leftrightarrow \tau'''$. Since $S \in \tau''*$, it is either empty or have the form S_1, S_2, where $S_1 \in \tau''$ and $S_1 \in \tau''*$. If $S = ()$, then the claim holds as proved when $\tau = ()$. Otherwise, suppose the target data for S_1 and S_2 are respective T_1 and T_2, where $T_1 \in \tau'''$, $T_2 \in \tau'''*$ and $T = T_1, T_2$. Then, $[\![X]\!]_{\mathcal{E}}(S_1, T_1) = (S'_1, \mathcal{E}'_1)$. By the induction hypothesis on τ'', we have $S'_1 \in \tau''$ and $\mathcal{E}'_1 \in \Gamma$, and then for $[\![X]\!]_{\mathcal{E}'_1}(S_2, T_2) = (S'_2, \mathcal{E}')$, by the induction hypothesis on $\tau''*$, we get $\mathcal{E}' \in \Gamma$ and $S'_2 \in \tau''*$, and also $S'_1, S'_2 \in \tau''*$.

  If $\tau = \tau_1, \tau_2$, then we have $\tau' = \tau'_1, \tau'_2$, where τ'_1 and τ'_2 are obtained from $\Gamma \vdash_{\mathtt{m}}$ $\mathtt{xmap}\ X : \tau_1 \leftrightarrow \tau'_1$ and $\Gamma \vdash_{\mathtt{m}} \mathtt{xmap}\ X : \tau_2 \leftrightarrow \tau'_2$. Since $S \in \tau_1, \tau_2$, it must have the form S_1, S_2, such that $S_1 \in \tau_1$ and $S_2 \in \tau_2$. Suppose the target data for S_1 and S_2 are respective T_1 and T_2, where $T_1 \in \tau'_1$, $T_2 \in \tau'_2$ and $T = T_1, T_2$. Then, $[\![X]\!]_{\mathcal{E}}(S_1, T_1)$ $= (S'_1, \mathcal{E}'_1)$. By the induction hypothesis on τ'', we have $S'_1 \in tau''$ and $\mathcal{E}'_1 \in \Gamma$, and then for $[\![X]\!]_{\mathcal{E}'_1}(S_2, T_2) = (S'_2, \mathcal{E}')$, by the induction hypothesis on τ_2, we get $\mathcal{E}' \in \Gamma$ and $S'_2 \in \tau_2$, and also $S'_1, S'_2 \in \tau_1, \tau_2$. The case for $\tau = \tau_1|\tau_2$ is proved similarly.
- Case $X = \mathtt{xif}\ P\ X_1\ X_2$. This case is proved by case analysis of P. We only show the proof for $P = \mathtt{xwithtag}\ tag$. For this case, $[\![X]\!]_{\mathcal{E}}(S, T)$ will be $[\![X_1]\!]_{\mathcal{E}}(S, T)$ or $[\![X_2]\!]_{\mathcal{E}}(S, T)$. The proof is done by the induction hypotheses on X_1 and X_2.

– $X =$ `xlet` *Var* X'. Suppose $\Gamma \vdash X : \tau \leftrightarrow \tau'$. Then, Γ, *Var* $\mapsto \tau \vdash X' : () \leftrightarrow \tau'$. Let $\mathcal{E}_1 = \mathcal{E}$, *Var* $\mapsto (S, S)$. Since $\mathcal{E} \in \Gamma$, we have $\mathcal{E}_1 \in \Gamma$, *Var* $\mapsto \tau$. If $[\![X]\!]_{\mathcal{E}}(S, T) = (S', \mathcal{E}')$, then $[\![X']\!]_{\mathcal{E}_1}((), T) = ((), \mathcal{E}_2)$, where $\mathcal{E}_2 = \mathcal{E}'$, *Var* $\mapsto (S, S')$. The proof is done by the induction hypothesis on X'.
– $X =$ `xfunapp` *fname* $[X_1, ..., X_n]$. The case is proved similarly as the `xlet` case.

□

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`.

<book(ori,s)>[<title(ori,s)>[a(ori,s)], <author(ori,s)>[b(ori,s)]],
<book(ori,s)>[<title(ori,s)>[c(ori,s)], <author(ori,s)>[d(ori,s)], <author(ori,s)>[e(ori,s)]]

7.1. *Missing Source Data*

The bidirectional transformations defined before 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 possible that some inserted values do not have the 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 it happens that 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 examples later), and apply it to the source data `BookList`. Then we obtain a view identical to the source data. The following updated view is the result of inserting a new book on the view.

<book(ori,s)>[<title(ori,s)>[a(ori,s)], <author(ori,s)>[b(ori,s)]],
<book(ori,s)>[<title(ori,s)>[c(ori,s)], <author(ori,s)>[d(ori,s)], <author(ori,s)>[e(ori,s)]]
<book(ins,s)>[<title(ins,s)>[f(ins,s)], <author(ins,s)>[g(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`. Obviously, the first book and the second book in the updated view have the first book and the second book in `BookList`, respectively, as their source data. The third book in the updated view misses its source data.

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. This is the only case where missing source data can happen,

as shown by the revised backward semantics of `xmap` later. The missing source data is denoted by Ω and its length is 0.

7.2. *Revising Forward Semantics*

Although missing source data happens at the backward execution of X in `xmap` X, we need to revise the forward semantics of the language since the backward execution of X may invoke forward executions of its constituent transformations with Ω as the source data. For example, suppose we apply the transformation

$$\texttt{xmap}\ (\texttt{xchild}; \texttt{xconst}\ \texttt{<}\text{book}^{(\texttt{ori,c})}\texttt{>}[()])$$

to the source data `BookList`. Then, we get the view $\overline{\texttt{<}\text{book}^{(\texttt{ori,c})}\texttt{>}[()], \texttt{<}\text{book}^{(\texttt{ori,c})}\texttt{>}[()]}$. For the updated view, $\texttt{<}\text{book}^{(\texttt{ori,c})}\texttt{>}[()], \texttt{<}\text{book}^{(\texttt{ori,c})}\texttt{>}[()], \texttt{<}\text{book}^{(\texttt{ins,c})}\texttt{>}[()]$, when performing backward transformation, we need to execute $\texttt{xchild}; \texttt{xconst}\ \texttt{<}\text{book}^{(\texttt{ori,c})}\texttt{>}[()]$ with the source data Ω and the view $\texttt{<}\text{book}^{(\texttt{ins,c})}\texttt{>}[()]$. Thus, by the backward semantics of the sequential composition transformation, `xchild` needs to be executed forward with the source data Ω.

There are four transformations and one predicate that have revised forward semantics, defined in Figure 17. For these transformations, if their source data is Ω, then they return Ω as views, otherwise their forward semantics is the same as before. In addition, for the `xif` transformation, if its predicate `P` returns Ω, then its view is also Ω. For the predicate `xeq` X, if its argument X returns Ω, then it returns Ω. Since if `xif` has Ω as its source data, it will not invoke its predicates, so `xeq` needs not to consider the case where its source data is Ω, and for the same reason the forward semantics of `xwithtag` and `xiselement` need not revision.

7.3. *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.

7.3.1. *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, so we use $T \sqsubseteq T_c$ rather than $T = T_c$ to check the validity of T.

$$[\![\texttt{xconst}\ T_c]\!]_{\mathcal{E}}(S, T) = \begin{cases} (S, \mathcal{E}), & \text{if } S \neq \Omega \text{ and } T \sqsubseteq T_c \\ \texttt{fail}, & \text{otherwise} \end{cases}$$

The example in Section 7.2 uses `xconst`, and its backward execution will fail since the `xconst` in that example takes Ω as its source data. Here is another example of `xconst` with with an inserted element as its view, and this time it has a successful its backward execution. The code for this example is given below.

$$
\begin{array}{lcl}
[\![\texttt{xchild}]\!]_{\mathcal{C}}(S) & = & \begin{cases} S', & \text{if } S = \texttt{<}tag^{(w,o)}\texttt{>}[S'] \\ \Omega, & \text{else if } S = \Omega \\ \texttt{fail}, & \text{otherwise} \end{cases} \\
[\![\texttt{xsetcnt } X]\!]_{\mathcal{C}}(S) & = & \begin{cases} \texttt{<}tag^{(w,o)}\texttt{>}[[\![X]\!]_{\mathcal{C}}(S')], & \text{if } S = \texttt{<}tag^{(w,o)}\texttt{>}[S'] \\ \Omega, & \text{else if } S = \Omega \\ \texttt{fail}, & \text{otherwise} \end{cases} \\
[\![\texttt{xmap } X]\!]_{\mathcal{C}}(()) & = & () \\
[\![\texttt{xmap } X]\!]_{\mathcal{C}}(\Omega) & = & \Omega \\
[\![\texttt{xmap } X]\!]_{\mathcal{C}}(\overline{v_1, ..., v_n}) & = & [\![X]\!]_{\mathcal{C}}(v_1), ..., [\![X]\!]_{\mathcal{C}}(v_n) \\
[\![\texttt{xif } P\ X_1\ X_2]\!]_{\mathcal{C}}(S) & = & [\![X_1]\!]_{\mathcal{C}}(S), \text{if } [\![P]\!]_{\mathcal{C}}(S) = \texttt{true}^{(\texttt{ori},c)} \\
[\![\texttt{xif } P\ X_1\ X_2]\!]_{\mathcal{C}}(S) & = & [\![X_2]\!]_{\mathcal{C}}(S), \text{if } [\![P]\!]_{\mathcal{C}}(S) = \texttt{false}^{(\texttt{ori},c)} \\
[\![\texttt{xif } P\ X_1\ X_2]\!]_{\mathcal{C}}(S) & = & \Omega, \text{if } S = \Omega \text{ or } [\![P]\!]_{\mathcal{C}}(S) = \Omega \\
[\![\texttt{xeq } X]\!]_{\mathcal{C}}(S) & = & \begin{cases} \texttt{true}^{(\texttt{ori},c)}, & \text{if } S = str^{(u,o)} \text{ and } [\![X]\!]_{\mathcal{C}}(()) = str^{(u',o')} \\ \texttt{false}^{(\texttt{ori},c)}, & \text{else if } S = str^{(u,o)}, [\![X]\!]_{\mathcal{C}}(()) = str'^{(u',o')} \text{ and } str \neq str' \\ \Omega, & \text{else if } [\![X]\!]_{\mathcal{C}}(()) = \Omega \\ \text{fail}, & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 17. Revised Forward Semantics

```
xmap (xlet $b (<pack^(ori,c)>[()]; xsetcnt (xvar $b)))
```

Applying the above code to the source data `BookList`, we get a view that includes two `pack` elements, each of which contains inside a book from the source data. Now if we insert a new pack element containing a new book element into the view as the last element, then the backward execution of the `xconst` above will succeed since its source data is () and the inserted book element will appear as the last element in the updated source data.

7.3.2. *Element Deconstruction*

If the transformations `xchild` does not have source data, it will 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 $\texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]$. If the view T has the type τ_i, then this view is supposed to come from a source element with the type $\texttt{<}tag_i^{o_i}\texttt{>}[\tau_i]$, so we use $tag_i^{(\texttt{ins},o_i)}$ as the tag of the updated source data.

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

It should mention that if the source data of `xchild` is not missing, we have a check to make sure the updated source data still has the annotated source-data type. 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 $\texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]$. 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 $\texttt{<}tag_i^{o_i}\texttt{>}[\tau_i]$, 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 $\tau_1|...|\tau_2$, T not necessarily has the type τ_i. Hence, the updated source data may not have the type $\texttt{<}tag_i^{o_i}\texttt{>}[\tau_i]$, or even not the type $\texttt{<}tag_1^{o_1}\texttt{>}[\tau_1]|...|\texttt{<}tag_n^{o_n}\texttt{>}[\tau_n]$.

7.3.3. *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 addition, the variable *Var* in the evaluation context $\mathcal{E}$ may be bound to a pair of Ωs by the most recent enclosed `xlet`. At this case, the context $\mathcal{E}$ is updated by setting the updated value of *Var* to the updated view T'. Otherwise, the context $\mathcal{E}$ is updated by setting the updated value of *Var* as the result of merging T' and the current updated value of *Var* in $\mathcal{E}$. Note that in the revised backward semantics a new `mg` operator, defined in Figure 18, is used to merge values. This new operator is directed by the type annotated on `xvar`, which characterizes the structure of the expected merging result, and it can merge two values that may not have identical structures due to insertions. This new `mg` operator is also used in the revised backward semantics of parallel composition transformation.

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

where $\mathcal{E}' = \overline{\mathcal{E}_1,\ \mathit{Var} \mapsto (S_1, \texttt{mg}(S_2,T',\tau)),\mathcal{E}_2}$ and $\mathcal{E}'' = \overline{\mathcal{E}_1,\ \mathit{Var} \mapsto (\Omega,T'),\mathcal{E}_2}$

The definition of new `mg` operator depends on two operators `allins` and `typeel`. The operation $\texttt{allins}(S)$ returns true if all top-level strings or elements in S have the updating annotation `ins`. The operation $\texttt{typeel}(S_1,\tau_1,S_2,\tau_2)$, defined in Figure 19, splits S_1 into two subsequences, say S_{11} and S_{12}, such that $S_{11} \in \tau_1$ and $\overline{S_{12},S_2} \in \tau_2$.

Here is an example for explaining the new `mg` operator. Suppose `Title` and `Author` denote the types of the title element and the author element in the source data `BookList`, respectively. Let S_1 and S_2 be respectively the following data

S_1 : $\texttt{<}\text{title}^{(\text{ori,s})}\texttt{>}[\text{t}^{(\text{mod,s})}], \texttt{<}\text{author}^{(\text{ins,s})}\texttt{>}[\text{h}^{(\text{ins,s})}], \texttt{<}\text{author}^{(\text{ori,s})}\texttt{>}[\text{b}^{(\text{ori,s})}]$

S_2 : $\texttt{<}\text{title}^{(\text{ori,s})}\texttt{>}[\text{a}^{(\text{ori,s})}], \texttt{<}\text{author}^{(\text{ins,s})}\texttt{>}[\text{g}^{(\text{ins,s})}], \texttt{<}\text{author}^{(\text{ori,s})}\texttt{>}[\text{b}^{(\text{ori,s})}], \texttt{<}\text{author}^{(\text{ins,s})}\texttt{>}[\text{k}^{(\text{ins,s})}]$

Then, the operation $\texttt{mg}(S_1,S_2,\overline{\texttt{Title},\texttt{Author}*})$ returns the following data, that merges insertions in S_1 and S_2.

$$\texttt{<}\text{title}^{(\text{ori,s})}\texttt{>}[\text{t}^{(\text{mod,s})}], \texttt{<}\text{author}^{(\text{ins,s})}\texttt{>}[\text{h}^{(\text{ins,s})}], \texttt{<}\text{author}^{(\text{ins,s})}\texttt{>}[\text{g}^{(\text{ins,s})}],$$
$$\texttt{<}\text{author}^{(\text{ori,s})}\texttt{>}[\text{b}^{(\text{ori,s})}], \texttt{<}\text{author}^{(\text{ins,s})}\texttt{>}[\text{k}^{(\text{ins,s})}]$$

$$
\begin{array}{lll}
\texttt{mg}((),(),()) & = & () \\
\texttt{mg}(str^{(u,o)}, str^{(u,o)}, \texttt{string}) & = & str^{(u,o)} \\
\texttt{mg}(str^{(\texttt{ori},\texttt{s})}, str^{(u,\texttt{s})}, \texttt{string}) & = & str^{(u,\texttt{s})}, \text{where } u \in \{\texttt{non}, \texttt{del}\} \\
\texttt{mg}(str^{(\texttt{ori},\texttt{s})}, str'^{(\texttt{mod},\texttt{s})}, \texttt{string}) & = & str'^{(\texttt{mod},\texttt{s})} \\
\texttt{mg}(str^{(u,\texttt{s})}, str^{(\texttt{ori},\texttt{s})}, \texttt{string}) & = & str^{(u,\texttt{s})}, \text{where } u \in \{\texttt{non}, \texttt{del}\} \\
\texttt{mg}(str'^{(\texttt{mod},\texttt{s})}, str^{(\texttt{ori},\texttt{s})}, \texttt{string}) & = & str'^{(\texttt{mod},\texttt{s})} \\
\texttt{mg}(\texttt{<}tag^{(w,o)}\texttt{>}[S_1], \texttt{<}tag^{(w,o)}\texttt{>}[S_2], \texttt{<}tag\texttt{>}[\tau]) & = & \texttt{<}tag^{(w,o)}\texttt{>}[S'], \text{where } S' = \texttt{mg}(S_1, S_2, \tau) \\
\texttt{mg}(\texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S_1], \texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S_2], \texttt{<}tag\texttt{>}[\tau]) & = & \texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S'], \text{where } S' = \texttt{mg}(S_1, S_2, \tau) \\
\texttt{mg}(\texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S_1], \texttt{<}tag^{(\texttt{ori},\texttt{s})}\texttt{>}[S_2], \texttt{<}tag\texttt{>}[\tau]) & = & \texttt{<}tag^{(\texttt{del},\texttt{s})}\texttt{>}[S'], \text{where } S' = \texttt{mg}(S_1, S_2, \tau) \\
\texttt{mg}(S_1, S_2, \tau*) & = & S_{11}, S_{21}, \texttt{mg}(S_{12}, S_{22}, \tau*) \\
\quad \text{where } \texttt{typeel}(S_1, \tau, (), \tau*) = (S_{11}, S_{12}), \texttt{typeel}(S_2, \tau, (), \tau*) = (S_{21}, S_{22}), & & \\
\quad\quad \texttt{allins}(S_{11}) = \texttt{true}, \text{and } \texttt{allins}(S_{21}) = \texttt{true} & & \\
\texttt{mg}(S_1, S_2, \tau*) & = & S_{11}, \texttt{mg}(S_{12}, S_2, \tau*) \\
\quad \text{where } \texttt{typeel}(S_1, \tau, (), \tau*) = (S_{11}, S_{12}), \texttt{typeel}(S_2, \tau, (), \tau*) = (S_{21}, S_{22}), & & \\
\quad\quad \texttt{allins}(S_{11}) = \texttt{true}, \text{and } \texttt{allins}(S_{21}) = \texttt{false} & & \\
\texttt{mg}(S_1, S_2, \tau*) & = & S_{21}, \texttt{mg}(S_1, S_{22}, \tau*) \\
\quad \text{where } \texttt{typeel}(S_1, \tau, (), \tau*) = (S_{11}, S_{12}), \texttt{typeel}(S_2, \tau, (), \tau*) = (S_{21}, S_{22}), & & \\
\quad\quad \texttt{allins}(S_{11}) = \texttt{false}, \text{and } \texttt{allins}(S_{21}) = \texttt{true} & & \\
\texttt{mg}(S_1, S_2, \tau*) & = & \texttt{mg}(S_{11}, S_{21}), \texttt{mg}(S_{12}, S_{22}, \tau*) \\
\quad \text{where } \texttt{typeel}(S_1, \tau, (), \tau*) = (S_{11}, S_{12}), \texttt{typeel}(S_2, \tau, (), \tau*) = (S_{21}, S_{22}), & & \\
\quad\quad \texttt{allins}(S_{11}) = \texttt{false}, \text{and } \texttt{allins}(S_{21}) = \texttt{false} & & \\
\texttt{mg}(S_1, S_2, \overline{\tau_1, \tau_2}) & = & \texttt{mg}(S_{11}, S_{21}, \tau_1), \texttt{mg}(S_{12}, S_{22}, \tau_2) \\
\quad \text{where } \texttt{typeel}(S_1, \tau_1, (), \tau_2) = (S_{11}, S_{12}), \texttt{typeel}(S_2, \tau_1, (), \tau_2) = (S_{21}, S_{22}) & & \\
\texttt{mg}(S_1, S_2, \tau_1|\tau_2) & = & \texttt{mg}(S_1, S_2, \tau_1), \text{where } S_1 \in \tau_1 \text{ and } S_2 \in \tau_1 \\
\texttt{mg}(S_1, S_2, \tau_1|\tau_2) & = & \texttt{mg}(S_1, S_2, \tau_2), \text{where } S_1 \in \tau_2 \text{ and } S_2 \in \tau_2 \\
\texttt{mg}(S_1, S_2, \tau) & = & \texttt{fail}, \text{if no other case applies}
\end{array}
$$

Fig. 18. The New `mg` Operator

$$
\begin{array}{lll}
\texttt{typeel}((), \tau_1, S_2, \tau_2) & = & \texttt{fail}, \text{if } () \notin \tau_1 \text{ or } S_2 \notin \tau_2 \\
\texttt{typeel}(S_1, \tau_1, S_2, \tau_2) & = & (S_1, S_2), \text{if } S_1 \in \tau_1 \text{ and } S_2 \in \tau_2 \\
\texttt{typeel}(S_1, \tau_1, S_2, \tau_2) & = & \texttt{typeel}(S_1', \tau_1, \overline{v, S_2}, \tau_2), \text{if } S_1 \notin \tau_1 \text{ and } S_1 = S_1', v
\end{array}
$$

Fig. 19. The `typeel` Operator

The following two lemmas about the new mg operator are needed to show the revised backward semantics of `xvar` satisfies the extended round-tripping property and the backward type preservation property.

Lemma 16. If $S = \texttt{mg}(S_1, S_2, \tau)$, then $S_1 \sqsubseteq S$ and $S_2 \sqsubseteq S$.

Proof. The proof proceeds by structural induction on the type τ.In the following, we prove the case where $\tau = \tau'*$. Other cases can be proved similarly.

- Case $\tau = \tau'*$. Suppose $\texttt{typeel}(S_1, \tau', (), \tau'*) = (S_{11}, S_{12})$ and $\texttt{typeel}(S_2, \tau', (), \tau'*) = (S_{21}, S_{22})$. There are four possibilities to generate merging results for $\texttt{mg}(S_1, S_2, \tau'*)$. If the merging result S is $S_{11}, S_{21}, \texttt{mg}(S_{12}, S_{22}, \tau'*)$, then by induction hypothesis, we get $S_{12} \sqsubseteq \texttt{mg}(S_{12}, S_{22}, \tau'*)$ and $S_{22} \sqsubseteq \texttt{mg}(S_{12}, S_{22}, \tau'*)$. We now prove $S_1 \sqsubseteq S$. By Lemma 5, we have $S_{12} \sqsubseteq S_{21}, \texttt{mg}(S_{12}, S_{22}, \tau'*)$, and then by Lemma 8, we get $S_{11}, S_{12} \sqsubseteq S_{11}, S_{21}, \texttt{mg}(S_{12}, S_{22}, \tau'*)$, that is, $S_1 \sqsubseteq S$. By applying Lemma 8 and then Lemma 5, $S_2 \sqsubseteq S$ can be proved. Similarly, we can prove the other three possibilities.

□

Lemma 17. Suppose $S_1 \in \tau$ and $S_2 \in \tau$. If $S = \texttt{mg}(S_1, S_2, \tau)$, then $S \in \tau$.

Proof. The proof proceeds by structural induction on the type τ.

- Case $\tau = ()$. The merging result for this case is (), so the claim holds trivially.
- Case $\tau = str$. The merging result is still the same string *str*, so the claim follows. The cases where $\tau = \texttt{string}$ are also proved similarly.
- Case $\tau = \texttt{<}tag\texttt{>}[\tau']$. For this case, we have $S_1 = \texttt{<}tag^{(u,o)}\texttt{>}[S_1']$ and $S_2 = \texttt{<}tag^{(u,o)}\texttt{>}[S_2']$. Let $S' = \texttt{mg}(S_1', S_2', \tau')$. By the induction hypothesis on τ', we get $S' \in \tau'$. Hence, $\texttt{<}tag^{(u,o)}\texttt{>}[S'] \in \texttt{<}tag\texttt{>}[\tau']$.
- Case $\tau = \tau'*$. Suppose $\texttt{typeel}(S_1, \tau', (), \tau'*) = (S_{11}, S_{12})$ and $\texttt{typeel}(S_2, \tau', (), \tau'*) = (S_{21}, S_{22})$. By the definition of `typeel`, we know $S_{11} \in \tau'$, $S_{12} \in \tau'*$, $S_{21} \in \tau'$ and $S_{22} \in \tau'*$. There are four possibilities to generate merging results for $\texttt{mg}(S_1, S_2, \tau'*)$. If the merging result S is $S_{11}, S_{21}, \texttt{mg}(S_{12}, S_{22}, \tau'*)$, then by induction hypothesis, we get $\texttt{mg}(S_{12}, S_{22}, \tau'*) \in \tau'*$, so $S \in \tau'*$. Similarly, we can prove the other three possibilities.
- Case $\tau = \tau_1, \tau_2$. Suppose $\texttt{typeel}(S_1, \tau_1, (), \tau_2) = (S_{11}, S_{12})$ and $\texttt{typeel}(S_2, \tau_1, (), \tau_2) = (S_{21}, S_{22})$. By the definition of `typeel`, we know $S_{11} \in \tau_1$, $S_{12} \in \tau_2$, $S_{21} \in \tau_1$ and $S_{22} \in \tau_2$. By the induction hypotheses on τ_1 and τ_2, respectively, we get $\texttt{mg}(S_{11}, S_{21}, \tau_1) \in \tau_1$ and $\texttt{mg}(S_{12}, S_{22}, \tau_2) \in \tau_2$. Hence, the claim follows directly.
- Case $\tau = \tau_1 | \tau_2$. The merging result for this case is either $\texttt{mg}(S_1, S_2, \tau_1)$ or $\texttt{mg}(S_1, S_2, \tau_2)$. the claim holds following the induction hypotheses on τ_1 and τ_2.

□

7.3.4. *Conditional Transformation*

If the source data of `xif` is not missing and its predicate has normal values ($\texttt{true}^{(\texttt{ori,c})}$ or $\texttt{false}^{(\texttt{ori,c})}$), 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 or its predicate returns Ω, then `xif` loses the information of how to advance its backward executions. At this case, the types annotated on `xif` provide such information. If the updated view has 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 data and the updated evaluation context must make sure the predicate of `xif` has the corresponding value. That is, if X_1 is chosen, then the predicate must have the value $\texttt{true}^{(\texttt{ori,c})}$ under the updated evaluation context with the

$$\texttt{lenpeel}(S_1, S_2, l) = \texttt{fail}, \text{if } \texttt{orilen}(S_1) < l$$
$$\texttt{lenpeel}(S_1, S_2, l) = (S_1, S_2), \text{if } \texttt{orilen}(S_1) = l$$
$$\texttt{lenpeel}(S_1, S_2, l) = \texttt{lenpeel}(S_1', \overline{v, S_2}, l), \text{if } \texttt{orilen}(S_1) > l \text{ and } S_1 = S_1', v$$

Fig. 20. The `lenpeel` Operator

updated source data, and similarly if X_2 is chosen. This requirement is necessary for the revised backward semantics satisfying the extended round-tripping property.

$$[\![\texttt{xif}^{\tau_2}_{\tau_1}\ P\ X_1\ X_2]\!]_{\mathcal{E}}(S,T) = [\![P]\!]_{\mathcal{E}'}(S,S'), \text{if } S \neq \Omega, [\![P]\!]_{\mathcal{E}.1}(S) = \texttt{true}^{(\texttt{ori,c})} \text{ and } [\![X_1]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$$
$$[\![\texttt{xif}^{\tau_2}_{\tau_1}\ P\ X_1\ X_2]\!]_{\mathcal{E}}(S,T) = [\![P]\!]_{\mathcal{E}'}(S,S'), \text{if } S \neq \Omega, [\![P]\!]_{\mathcal{E}.1}(S) = \texttt{false}^{(\texttt{ori,c})} \text{ and } [\![X_2]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$$
$$[\![\texttt{xif}^{\tau_2}_{\tau_1}\ P\ X_1\ X_2]\!]_{\mathcal{E}}(S,T) = [\![P]\!]_{\mathcal{E}'}(S,S'), \text{if } S = \Omega \text{ or } [\![P]\!]_{\mathcal{E}.1}(S) = \Omega, T \in \tau_1, [\![X_1]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$$
$$\text{and } [\![P]\!]_{\mathcal{E}'.2}(S') = \texttt{true}^{(\texttt{ori,c})}$$
$$[\![\texttt{xif}^{\tau_2}_{\tau_1}\ P\ X_1\ X_2]\!]_{\mathcal{E}}(S,T) = [\![P]\!]_{\mathcal{E}'}(S,S'), \text{if } S = \Omega \text{ or } [\![P]\!]_{\mathcal{E}.1}(S) = \Omega, T \in \tau_2, [\![X_2]\!]_{\mathcal{E}}(S,T) = (S', \mathcal{E}')$$
$$\text{and } [\![P]\!]_{\mathcal{E}'.2}(S') = \texttt{false}^{(\texttt{ori,c})}$$

7.3.5. *Mapping Transformation*

The revision of `xmap` mainly focuses on the `split` operator. The operator `split` is used by `xmap` (also by the parallel composition transformation) to divide their views into subsequences, and then each subsequence is transformed backward to update the corresponding string or element in the source data. 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 an elegant splitting mechanism is needed for `xmap` if its views include insertions. For example, for the source data `BookList`, the code `xmap xchild` produces a view consisting of a sequence of titles and authors of each book. Then, consider the following updated view.

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

where three authors and one title elements 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` for generating an updated or new book element. For this example, it is reasonable to split the updated view into three subsequences: the first three elements, the next two, and the last four. Thus, in the updated source data, 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, and the third book element has an inserted author.

The revised `split` operator is given in Figure 21. 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 source data of this subsequence is

$$
\begin{array}{lcl}
\texttt{split}((), [], ()) & = & [] \\
\texttt{split}(T, [l], \lceil \tau \rceil) & = & [(T, \texttt{e})], \text{where } \texttt{orilen}(T) = l \\
\texttt{split}(T, [], \lceil \tau \rceil) & = & [(T, \texttt{m})], \text{where } \texttt{allins}(T) = \texttt{true} \\
\texttt{split}(T, ls, \tau*) & = & \texttt{split}(T, ls, ()|\overline{\tau, \tau*}) \\
\texttt{split}(T, ls, \tau_1|\tau_2) & = & \texttt{split}(T, ls, \tau_1), \text{where } T \in \tau_1 \\
\texttt{split}(T, ls, \tau_1|\tau_2) & = & \texttt{split}(T, ls, \tau_2), \text{where } T \in \tau_2 \\
\texttt{split}(T, l{:}ls, \overline{\lceil \tau_1 \rceil, \tau_2}) & = & (T_{11}, \texttt{e}){:}\texttt{split}(T_{12}, ls, \tau_2) \\
\quad \text{where } \texttt{lenpeel}(T, (), l) = (T_1, T_2), \texttt{typeel}(T_1, \tau_1, T_2, \tau_2) = (T_{11}, T_{12}) & & \\
\quad\quad \text{and } \texttt{orilen}(T_{11}) = l & & \\
\texttt{split}(T, l{:}ls, \overline{\lceil \tau_1 \rceil, \tau_2}) & = & (T_{11}, \texttt{m}){:}\texttt{split}(T_{12}, l{:}ls, \tau_2) \\
\quad \text{where } \texttt{lenpeel}(T, (), l) = (T_1, T_2), \texttt{typeel}(T_1, \tau_1, T_2, \tau_2) = (T_{11}, T_{12}), & & \\
\quad\quad \texttt{allins}(T) = \texttt{true}, \text{and } \texttt{orilen}(T_{11}) \neq l & & \\
\texttt{split}(T, [], \overline{\lceil \tau_1 \rceil, \tau_2}) & = & (T_1, \texttt{m}){:}\texttt{split}(T_2, [], \tau_2) \\
\quad \text{where } \texttt{allins}(T) = \texttt{true}, \text{and } \texttt{typeel}(T, \tau_1, (), \tau_2) = (T_1, T_2) & & \\
\texttt{split}(T, ls, \tau) & = & \texttt{fail}, \text{if no other case applies}
\end{array}
$$

Fig. 21. The `split` Operator

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

Fig. 22. The `iter` Operator

missing or existing, respectively. The definition of `split` depends on the operators `orilen` and `lenpeel`. The operation `orilen`(S) returns from S the number of top-level strings or elements, which do not have the `ins` annotation, and the operation `lenpeel`(S_1, S_2, l), defined in Figure 20, divides S_1 into two parts, say S_{11} and S_{12}, such that `orilen`(S_{11}) = l, and returns S_{11} and $\overline{S_{12}, S_2}$ in a pair.

Let `Title` and `Author` be the types for title and author elements in the source data `BookList`, and T represent the above view. Then, the source data `BookList` has the type <book$^{\texttt{s}}$>$[$`Title`, `Author`$*]*$, and T has the type $\lceil$`Title`, `Author`$*\rceil*$. The first book element in `BookList` generates two elements in T, and the second generates three, so we can use the operation `split`(T, $[2, 3]$, $\lceil$`Title`, `Author`$*\rceil*$) to split the above view T. The result is three subsequences as expected, with the first and third subsequences flagged by `e` and the second flagged by `m`.

The revised backward semantics of `xmap` X is given below, where the new `iter` operator is defined in Figure 22. For a subsequence, if its source data is missing, then in the `iter` operator, the backward execution of X takes Ω as the source data. The updated source data of `xmap` is checked against the annotated source-data type to make sure it is well-typed. The example below shows that this check is necessary for the revised backward semantics of `xmap` to satisfy the backward type preservation property.

$$[\![\texttt{xmap}_\tau^{\tau'}\ X]\!]_{\mathcal{E}}(S,T) \ = \ (S',\mathcal{E}'), \text{if } S = () \text{ or } S = \Omega$$
$$\text{where } ST = \texttt{split}(T, [], \tau'), (S',\mathcal{E}') = \texttt{iter}(X, ST, S, (), \mathcal{E}), \text{and } S' \in \tau$$
$$[\![\texttt{xmap}_\tau^{\tau'}\ X]\!]_{\mathcal{E}}(S,T) \ = \ (S',\mathcal{E}'), \text{if } S = \overline{v_1, ..., v_n}$$
$$\text{where } ST = \texttt{split}(T, [\texttt{len}([\![X]\!]_{\mathcal{E}.1}(v_1)), ..., \texttt{len}([\![X]\!]_{\mathcal{E}.1}(v_n))], \tau')$$
$$(S',\mathcal{E}') = \texttt{iter}(X, ST, S, (), \mathcal{E}), \text{and } S' \in \tau$$

In this example, suppose the source data has the type $\overline{\texttt{Title}, \texttt{Author}}*$, where `Title` and `Author` are the types for title and author elements, respectively. Then, applying the code `xmap` (`xif` (`xwithtag` title) `xid` `xconst` ()) to a well-typed source data, we get a view consisting of title elements with the view type $\lceil\texttt{Title}\rceil*$. If we insert a new title element into the view, the updated view still has the correct type, but after the backward execution of the example code, the updated source data is not well-typed since the new title element does not have the corresponding author element. Hence, in this example, the backward execution fails according to the revised backward semantics of `xmap`.

7.3.6. *Parallel composition*

With the annotated types and the new operators `mg` and `split`, the revision of the parallel composition transformation is straightforward.

$$[\![X_1||_\tau^{\tau'} X_2]\!]_{\mathcal{E}}(S,T) \ = \ (\texttt{mg}(S_1', S_2', \tau), \mathcal{E}')$$
$$\text{where}$$
$$[T_1, T_2] = \texttt{split}(T, [\texttt{len}([\![X_1]\!]_{\mathcal{E}.1}(S)), \texttt{len}([\![X_2]\!]_{\mathcal{E}.1}(S)], \tau')$$
$$(S_2', \mathcal{E}'') = [\![X_2]\!]_{\mathcal{E}}(S, T_2)$$
$$(S_1', \mathcal{E}') = [\![X_1]\!]_{\mathcal{E}''}(S, T_1)$$

7.3.7. *Function call*

When a function call, $\texttt{xfunapp}^{[\tau_1,...,\tau_n]}$ *fname* $[X_1, ..., X_n]$, is executed backward, it needs to annotate the body of the function *fname* with respect to the types of its arguments, that is, $\tau_1, ..., \tau_n$, respectively. Suppose the function *fname* is defined as

$$\texttt{fun}\ \mathit{fname}(\mathit{Var}_1, ..., \mathit{Var}_n) = X$$

Then, its body X is annotated by type-checking X with the following judgment.

$$[\mathit{Var}_1 \mapsto \tau_1, ..., \mathit{Var}_n \mapsto \tau_n, \mathit{fname}(\tau_1, ..., \tau_n) \mapsto a] \vdash X : () \leftrightarrow \tau' \Rightarrow X'$$

where a is fresh.

8. Implementation

The approach proposed in this work has been implemented in Java with JDOM. Our system is available at [9], where several XQuery Core examples are provided. Most of these examples are obtained by normalizing XQuery use cases from the W3C draft [8] with the Galax XQuery engine [12].

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. More interestingly, our 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 [16], 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 [8]. 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 languages in [5–7] are most closely related. These languages 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. 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. In addition, the languages in [5–7] do not have a type system or do not take regular expression types in their type systems.

The injective language in [17] and the reversible language in [18] can also be executed in bidirectional ways. These languages express only injective functions, so their programs can be inverted. Our bidirectional language can express functions not necessarily injective, with the cost that the backward execution may fail due to, for instance, conflicting updates. The work [19] 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 [19] is simple in that a bound variable can only be used one time 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 regular expression

types, either.

The work [20,21] 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 [22] 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 in [23] also uses programming language technique to solve the view updating problem. But the view definition language in [23] is not bidirecitonal, so when defining a view, users have to write the code for putting back possible updates into the source XML data.

10. Conclusion

In this paper, we have proposed an approach to address the view updating problem of XQuery. In this approach, we first designed an expressive bidirectional XML transformation language, and then used it to interpret XQuery. The backward executions of XQuery expressions reflect back the updates to views into the XML source data making up the views. Although we are motivated by interpreting XQuery, we believe that our work provides a potential technique to define general-purpose bidirectional functional languages since the bidirectional semantics of functions and some constructors for algebraic data types can be defined in this technique.

We proposed the extended round-tripping property as one criterion for well-behaved bidirectional transformations. This criterion is more suitable for the expressive view-definition languages, like XQuery. We have proved that the bidirectional language proposed in this paper satisfies this property.

The structure of XML data is generally described with regular expression types. We designed a type system with regular expression types for our bidirectional language. We have proved that this type system is sound with respect to the forward semantics of this language and the types of source data are preserved by backward executions of well-typed programs.

The insertions on views are difficult to transform backward. We illustrated why they are difficult and proposed a type-based solution to this problem. The annotated types on transformations provide guiding information for backward executions to update source data more reasonably.

One interesting future work is to analyze bidirectional programs and determine what are valid updates on views, such that valid updates do not lead to failure during backward executions.

References

[1] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome 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 Transactions on Database Systems*, 6(4):557–575, 1981.

[3] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM TODS*, 7(3):381–416, 1982.

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

[5] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246. ACM Press, 2005.

[6] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *Proceedings of the 25th ACM symposium on Principles of Database Systems*, 2006.

[7] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2004.

[8] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases, 2006. http://www.w3.org/TR/xquery-use-cases/.

[9] Bidirectional XQuery. http://www.ipl.t.u-tokyo.ac.jp/~liu/BiXQuery.html.

[10] Veronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[11] Peter Buneman, Sanjeev Khanna, and Wang-Chiew 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*, pages 150–158, New York, NY, USA, 2002. ACM Press.

[12] Galax: An Implementation of Query. http://www.galaxquery.org/.

[13] A. Marian and J. Simeon. Projecting XML documents. In *Proceedings of VLDB 2003*, 2003.

[14] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking forward. In *Proceedings of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[15] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[16] Malcolm Wallace and Colin Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, 1999.

[17] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *APLAS*, volume 3302, pages 2–20, 2004.

[18] Tetsuo Yokoyama and Robert 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*, pages 144–153. ACM Press, 2007.

[19] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the ACM International Conference on Functional Programming*, 2007.

[20] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. PATAXO: A framework to allow updates through xml views. *ACM Trans. Database Syst.*, 31(3):839–886, 2006.

[21] Vanessa Braganholo, Susan Davidson, and Carlos Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *Proceedings of VLDB 2004*, 2004.

[22] Ling Wang and Elke A. Rundensteiner. On the updatability of XML views published over relational data. In *International Conference on Conceptual Modeling*, 2004.

[23] H. Kozankiewicz, J. Leszczylowski, and K. Subieta. Updatable XML views. In *ADBIS*, pages 381–399, 2003.