Bidirectional Interpretation of XQuery

A Case Study towards Bidirectionalizing Functional Languages

Dongxi Liu Zhenjiang Hu Masato Takeichi

Department of Mathematical Informatics, University of Tokyo, Japan

{liu,hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract

XQuery is a powerful functional language to query XML data. This paper gives 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 expressions into the code of this language. As a result, an XQuery expression can execute in two directions: in the forward direction, it generates a materialized view from the source XML data; while in the backward direction, it updates the source data by putting back the updates on the view. we have implemented our approach and applied it to some XQuery use cases from a W3C draft, which confirms the practicability of this approach.

Keywords Bidirectional programming, XQuery, XML

1. Introduction

XQuery [4] is a powerful functional language designed to query XML data. The role of XQuery to XML is just like that of SQL to relational database tables. However, XQuery still lacks an important feature that SQL has. This feature is *view update* [3, 9, 11], 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 the source XML 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 execute 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 also execute in two directions, and the backward execution will put the updates on views back into the source data.

The underlying language is inspired by the bidirectional language proposed in [10, 5], which includes a collection of combinators, called *lenses*, for tree transformations. The technique used by this 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 [10], it is not clear what the limits of bidirectional programming with this technique are, or how expressive the combinators defined in this language could be. 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 destructing 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 also design a type system 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 type-correct, and if yes, it also generates the corresponding view type. The soundness property of this type system characterizes both the forward and backward behaviors of well-typed programs. For a type-correct program, its forward execution does not get stuck and will generate the view with the correct type. However, its backward execution probably terminates with a special value `fail` and fail to update the source data even if the updated view conforms to the view type. This happens when updates on the view contain conflicts or improper insertions. The successful backward execution guarantees the updated source data is valid against the given source type.

For XQuery views, we consider three kinds of updates: modifications to text contents, insertions or deletions of elements. The insertions on views are more tricky to deal with 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. This problem is illustrated more by examples in Section 6. We solve this problem by annotating the language constructs with types, which provide guidance information for putting inserted data back in a reasonable way. We, however, do not need users to annotate programs manually. This is done by the type system.

The property of view updating is generally stated by the condition that after the source data is updated according to an updated view, executing the same query on the updated source data should get the same view as the updated view again [3, 9, 5, 10]. However, this condition is not suitable for the view updating problem of XQuery, which is a quite general functional language. For example, if an XQuery expression creates a view containing several copies of one value from the source data (i.e., the dependency in view [13]), then modifying one copy will violate the above condition even the value in the source data is correctly updated. This is because executing this XQuery expression on the updated source data will generate a different view where all copies of the value be-

```
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>
```

Figure 1. An XQuery Expression

come the modified one. In this work, the property of view updating is studied by relating the updates on views with those in the source data. The well-behaved bidirectional programs are required to put all view updates back into the related values in the source data if their backward executions are successful.

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

- We design a bidirectional language, which is expressive enough to interpret XQuery. This language also has a sound type system to check the type correctness of bidirectional programs.
- We define the translational semantics of XQuery by giving the translation rules from XQuery Core to the target bidirectional language.
- We illustrate the difficulties of processing insertions on views and propose type-directed approaches to processing insertions on views.
- The view updating semantics is defined by relating the updates between the source data and view, which is more suitable for the view updating problem of XQuery.
- Our approach has been implemented and is available publicly at [1].

The remainder of the paper is organized as follows. Section 2 gives an example to illustrate our motivation. Section 3 defines the target bidirectional language. Section 4 interprets XQuery. Section 5 presents the type system. Section 6 discusses the insertion problems and approaches. Section 7 introduces our implmentation. Section 8 gives the related work and Section 9 concludes the paper.

2. An Example

Our motivation can be explained 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 following data:

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

$$
\begin{array}{lcl}
\mathit{value} & ::= & v \mid S \\
v & ::= & \mathit{str}^u \mid \texttt{<}\mathit{tag}^u\texttt{>}[S] \\
S & ::= & () \mid v_1, ..., v_n \\
u & ::= & \texttt{non} \mid \texttt{mod} \mid \texttt{ins} \mid \texttt{del}
\end{array}
$$

Figure 2. Syntax of XML Data

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

Figure 3. Syntax

Then, in a bidirectional context, the forward execution of the query in Figure 1 will generate the following view, which is the table-of-contents of the book:

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

On this view, users can modify titles or id attributes, insert or delete sections. These updates will be put back into the source file "book.xml" automatically by executing backward this query. For example, if we change the id attribute on this view from "intro" into "introduction" and insert a new section after the first section, then after the backward execution of this query the value of the corresponding attribute in the source data is also changed into the same value and the new section will appear between the first and second sections. This example can be found at [1].

3. The Bidirectional Language

This section introduces the bidirectional language for interpreting XQuery. The backward semantics of the language in this section does not consider insertions on views.

3.1 XML Values

The syntax of XML values is given in Figure 2. An XML value is either a single value *v* or a sequence *S* of single values. An empty sequence is written as `()`. To save space, the end tags of XML elements are omitted and their contents are enclosed by brackets.

Strings or elements are annotated with the flag *u*, which indicates their updating status. The `non` flag means the strings or the tags of elements are not modified, otherwise the `mod` flag should be used. The `ins` flags are for inserted values, and `del` for deleted values. In addition, if an element has the `ins` or `del` flag, then all strings and elements in its contents also have the same flag.

In this work, deleted values in the updated source data are still kept, but flagged with the `del` flag. They can be removed easily by an independent procedure like the database trigger, which can take into account some application-specific constraints on the source data when removing values. As an example, for the source data in Section 2, if an element `title` is indicated by `del`, then the `section` element containing it can be removed since a section should have a title.

3.2 Syntax

The syntax of this language is defined in Figure 3. In this syntax, *Var* and *fname* represent the variable names and function names, respectively. The metavariable X represents bidirectional transformations. The transformations `xconst` and `xvar` correspond to the constant or variable expressions in general programming languages. The transformations `xchild` and `xsetcnt` are used to get or set the contents of elements. This language also includes other transformation constructs, such as those to deal with element attributes or name spaces, which are not presented in this paper. The sequential composition of transformations $X_1; X_2$ is to execute two transformations X_1 and X_2 in sequence, while the parallel composition of transformations $X_1||X_2$ executes X_1 and X_2 in parallel. The construct `xfunapp` applies the function *fname*, defined globally in *G*, onto a list of arguments.

3.3 Semantics

This language supports variable bindings, so we need evaluation contexts or environments to maintain the values of variables in both forward and backward executions. The context for forward executions is denoted by $\mathcal{C}$, which maps variables to their values; the context for backward executions is denoted by $\mathcal{E}$, which maps variables to pairs of values. Suppose for a variable *Var*, $\mathcal{E}(Var)$ = (S, S'). Then, S is the original value of *Var*, and S' is the updated value of *Var* during backward executions. The notation $\mathcal{E}(Var).1$ is used for the first component of the pair $\mathcal{E}(Var)$, and $\mathcal{E}(Var).2$ for the second component; the notation $\mathcal{E}.1$ denotes a new evaluation context, say $\mathcal{C}'$, defined as $Dom(\mathcal{E}) = Dom(\mathcal{C}')$ and $\forall\, Var \in Dom(\mathcal{E})$, $\mathcal{C}'(Var) = \mathcal{E}(Var).1$, where $Dom(\mathcal{E})$ (or $Dom(\mathcal{C}')$) means the domain of $\mathcal{E}$ (or $\mathcal{C}'$). These contexts can be processed like stacks. The notation $\mathcal{C} \oplus [\,Var \mapsto S]$ denotes a new context where a new binding of variable *Var* is pushed onto the top of $\mathcal{C}$, and similarly for pushing new bindings onto $\mathcal{E}$. The notation $\mathcal{E}[\,Var \mapsto S]$ means the bound value of the least recent variable Var in $\mathcal{E}$ is changed to S. When we concern the top variable binding in the context $\mathcal{E}$, we use the notation $\mathcal{E}_1 \bullet [Var \mapsto (S, S')]$ to represent $\mathcal{E}$, where $\mathcal{E}_1$ represents the bottom part of $\mathcal{E}$.

Let V be a sequence of single XML values. The forward and backward semantics of each language construct is defined by the following forms:

- The forward semantics: $[\![X]\!]_{\mathcal{C}}(S)$ = V, which means the transformation X generates the view V from the source data S under the context $\mathcal{C}$.
- The backward semantics: $[\![X]\!]_{\mathcal{E}}(S, V') = (S', \mathcal{E}')$, which means under the environment $\mathcal{E}$, the transformation X generates the updated source data S' from the updated view V' and the original source S. In addition, a new environment $\mathcal{E}'$ is also generated. The backward execution of X probably fails, and the form $[\![X]\!]_{\mathcal{E}}(S, V')$ = `fail` is used for such case.

In what follows, we will define the forward and backward semantics for each language construct in Figure 3.

Identity transformation: This transformation returns the source data as the view in the forward direction, and returns the updated view as the updated source data in the backward direction.

$$\begin{array}{lcl} [\![\texttt{xid}]\!]_{\mathcal{C}}(S) & = & S \\ [\![\texttt{xid}]\!]\mathcal{E}(S, V) & = & (V, \mathcal{E}) \end{array}$$

Constant transformation: This transformation returns a constant view V for any source data in the forward direction and returns the original source data in the backward direction without allowing updates on V. When the special value `fail` is generated, the being executed program terminates immediately.

$$\begin{array}{lcl} [\![\texttt{xconst}\ V]\!]_{\mathcal{C}}(S) & = & V \\ [\![\texttt{xconst}\ V]\!]_{\mathcal{E}}(S, V') & = & \begin{cases} (S, \mathcal{E}), & \text{if } V = V' \\ \texttt{fail}, & \text{otherwise} \end{cases} \end{array}$$

This transformation provides a template to implement other non-invertible functions in XQuery, such as the sum and comparison operations. Their backward executions do not update the source data and their views cannot be changed.

Variable reference: The forward execution of this transformation 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. The `mg` operation merges the updates in two values and will be defined later.

$$\begin{array}{lcl} [\![\texttt{xvar}\,Var]\!]_{\mathcal{C}}(S) & = & \mathcal{C}(Var) \\ [\![\texttt{xvar}\,Var]\!]_{\mathcal{E}}(S, V') & = & (S, \mathcal{E}') \\ \multicolumn{3}{l}{\text{where}} \\ \multicolumn{3}{l}{\quad \mathcal{E}' = \mathcal{E}[\,Var \mapsto (\mathcal{E}(Var).1, \texttt{mg}(V', \mathcal{E}(Var).2))]} \end{array}$$

Element destructing: This construct corresponds to the `child` axis in XPath. It returns the contents of the source element in the forward execution, and replaces the contents with the updated view in the backward execution.

$$\begin{array}{lcl} [\![\texttt{xchild}]\!]_{\mathcal{C}}(\texttt{<}tag^u\texttt{>}[S]) & = & S \\ [\![\texttt{xchild}]\!]_{\mathcal{E}}(\texttt{<}tag^u\texttt{>}[S], S') & = & (\texttt{<}tag^u\texttt{>}[S'], \mathcal{E}) \end{array}$$

Element constructing: Suppose S is an element of the form `<`tag^u`>`$[S']$. In the forward direction, the contents of this element are replaced by the result of executing X, and in the backward direction the original contents are restored. The updates on V are reflected back to both the tag of the source element and the values of variables in $\mathcal{E}$. The argument transformation X is applied to an empty source, so its backward execution only updates the context.

$$\begin{array}{lcl} [\![\texttt{xsetcnt}\ X]\!]_{\mathcal{C}}(S) & = & \texttt{<}tag^u\texttt{>}[[\![X]\!]_{\mathcal{C}}(())] \\ [\![\texttt{xsetcnt}\ X]\!]_{\mathcal{E}}(S, \texttt{<}tag'^{u'}\texttt{>}[V]) & = & (\texttt{<}tag'^{u'}\texttt{>}[S'], \mathcal{E}') \\ \multicolumn{3}{l}{\text{where}} \\ \multicolumn{3}{l}{\quad ((), \mathcal{E}') = [\![X]\!]_{\mathcal{E}}((), V)} \end{array}$$

Sequential composition: This composed transformation applies its argument transformations one by one. This definition is same as that in [10] except that this definition considers the evaluation contexts. Note that the backward execution of X_2 needs to invoke the forward execution of X_1 to generate the intermediate source data.

$$\begin{array}{lcl} [\![X_1; X_2]\!]_{\mathcal{C}}(S) & = & [\![X_2]\!]_{\mathcal{C}}([\![X_1]\!]_{\mathcal{C}}(S)) \\ [\![X_1; X_2]\!]_{\mathcal{E}}(S, V) & = & [\![X_1]\!]_{\mathcal{E}'}(S, V') \\ \multicolumn{3}{l}{\text{where}} \\ \multicolumn{3}{l}{\quad (V', \mathcal{E}') = [\![X_2]\!]_{\mathcal{E}}([\![X_1]\!]_{\mathcal{E}.1}(S), V)} \end{array}$$

Parallel composition: This composed transformation executes each of its argument transformations, and puts their views in parallel. The operator `len` returns the length of a sequence, and the operator $\texttt{split}(V, [l_1, ..., l_n])$ divides the value V into n subsequences V_i' $(1 \leq i \leq n)$, where $\texttt{len}(V_i') = l_i$. For example, $\texttt{split}(\underbrace{v_1, v_2, v_3}, [2, 1])$ generates two subsequences: $\underbrace{v_1, v_2}$ and v_3. For clarity, a sequence value is sometimes underbraced.

$$\begin{array}{lcl} [\![X_1||X_2]\!]_{\mathcal{C}}(S) & = & [\![X_1]\!]_{\mathcal{C}}(()), [\![X_2]\!]_{\mathcal{C}}(()) \\ [\![X_1||X_2]\!]_{\mathcal{E}}(S, V) & = & (S, \mathcal{E}'') \\ \multicolumn{3}{l}{\text{where}} \\ \multicolumn{3}{l}{\quad V_1', V_2' = \texttt{split}(V, [\texttt{len}([\![X_1]\!]_{\mathcal{E}.1}(())), \texttt{len}([\![X_2]\!]_{\mathcal{E}.1}(()))])} \\ \multicolumn{3}{l}{\quad ((), \mathcal{E}') = [\![X_2]\!]_{\mathcal{E}}((), V_2')} \\ \multicolumn{3}{l}{\quad ((), \mathcal{E}'') = [\![X_1]\!]_{\mathcal{E}'}((), V_1')} \end{array}$$

Mapping: Suppose $S = v_1, ..., v_n$. This transformation applies its argument transformation X to each single value $v_i (1 \leq i \leq n)$ in the source data S.

$$\begin{array}{lcl} [\![\texttt{xmap}\ X]\!]_{\mathcal{C}}(S) & = & [\![X]\!]_{\mathcal{C}}(v_1), ..., [\![X]\!]_{\mathcal{C}}(v_n) \\ [\![\texttt{xmap}\ X]\!]_{\mathcal{E}}(S,V) & = & (\underbrace{v'_1, ..., v'_n}, \mathcal{E}') \end{array}$$

where

$$V'_1, ..., V'_n = \texttt{split}(V, [\texttt{len}([\![X]\!]_{\mathcal{E}.1}(v_1)), ..., \texttt{len}([\![X]\!]_{\mathcal{E}.1}(v_n))])$$
$$(v'_n, \mathcal{E}_{n-1}) = [\![X]\!]_{\mathcal{E}}(v_n, V'_n)$$
$$(v'_{n-1}, \mathcal{E}_{n-2}) = [\![X]\!]_{\mathcal{E}_{n-1}}(v_{n-1}, V'_{n-1})$$
$$...$$
$$(v'_1, \mathcal{E}') = [\![X]\!]_{\mathcal{E}_1}(v_1, V'_1)$$

Conditional transformation: The argument transformation X_1 is chosen if the predicate P holds, otherwise X_2 is chosen. A predicate holds if it does not return the empty sequence.

$$[\![\texttt{xif}\ P\ X_1\ X_2]\!]_{\mathcal{C}}(S) = \begin{cases} [\![X_1]\!]_{\mathcal{C}}(S), & \text{if } [\![P]\!]_{\mathcal{C}}(S) \neq () \\ [\![X_2]\!]_{\mathcal{C}}(S), & \text{otherwise} \end{cases}$$
$$[\![\texttt{xif}\ P\ X_1\ X_2]\!]_{\mathcal{E}}(S,V) = \begin{cases} [\![X_1]\!]_{\mathcal{E}}(S,V), & \text{if } [\![P]\!]_{\mathcal{E}.1}(S) \neq () \\ [\![X_2]\!]_{\mathcal{E}}(S,V), & \text{otherwise} \end{cases}$$

Predicates: Predicates are only used as the condition of `xif`, where only their forward semantics are concerned. This means that the predicates are not essential to the expressiveness of our language, and the language can include other needed predicates, such as the existential predicate in XQuery Core, without affecting the definition of `xif`. The predicates used in this paper will be introduced informally.

The predicate `xwithtag` *str* holds if the input data is an element with the tag *str*; the predicates `xiselement` and `xistext` judge whether the input data is an element or a string, respectively. When these three predicates hold, they can return any nonempty value as their results. We let them return the string "true". A transformation X can also be used as a predicate, and its value is determined by its forward semantics.

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.

$$\begin{array}{lcl} [\![\texttt{xlet}\ \mathit{Var}\ X]\!]_{\mathcal{C}}(S) & = & [\![X]\!]_{\mathcal{C}\oplus[\mathit{Var}\mapsto S]}(()) \\ [\![\texttt{xlet}\ \mathit{Var}\ X]\!]_{\mathcal{E}}(S,V) & = & (S', \mathcal{E}') \end{array}$$

where

$$((), \mathcal{E}' \bullet [\mathit{Var} \mapsto (S, S')]) = [\![X]\!]_{\mathcal{E}\oplus[\mathit{Var}\mapsto(S,S)]}((), V)$$

The forward semantics of this construct is same as that of the let in general functional programming languages. Its backward semantics is defined by executing backward the transformation X under the context $\mathcal{E} \oplus [\mathit{Var} \mapsto (S, S)]$, where the variable *Var* is bound to a pair of the original source data S. After the backward execution of X, the generated context $\mathcal{E}' \bullet [\mathit{Var} \mapsto (S, S')]$ 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 $X_1, ..., X_n$ can be defined by using the constructs defined before.

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

where

$$X'_1 = X_1; \texttt{xlet}\ \mathit{Var}_1\ X'_2$$
$$X'_2 = X_2; \texttt{xlet}\ \mathit{Var}_2\ X'_3$$
$$...$$
$$X'_n = X_n; \texttt{xlet}\ \mathit{Var}_n\ X$$

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 the transformation `xfunapp`. Hence, any data to be processed by the function body should be passed as the arguments of the function call.

```
fun toc($book-or-section) =
  xvar $book-or-section; xchild;
  xmap (xif (xwithtag ''section'') X0 (xconst ()))
where
  X0 = xlet $section
         (xconst <section>[]; xsetcnt (X1||X2))
  X1 = xvar $section; xchild;
       xmap (xif (xwithtag ''title'') xid (xconst ()))
  X2 = xfunapp toc [xvar $section]
```

Figure 4. A Programming Example

3.4 Merging Updates

For view updating of XQuery, it is common that one source value has several replicas, which may contain different updates. The merging operation `mg` is to combine all updates in two replicas if there are no conflicts. For example, merging the elements <Title$^{\text{mod}}$>[Xquery$^{\text{non}}$] and <title$^{\text{non}}$>[XQuery$^{\text{mod}}$] will generate a new element <Title$^{\text{mod}}$>[XQuery$^{\text{mod}}$], and merging the elements <price$^{\text{non}}$>[30$^{\text{mod}}$] and <price$^{\text{non}}$>[25$^{\text{mod}}$] will cause a conflict.

The `mg` operation in this section only merges the values without insertions. It will be extended in Section 6 to consider inserted values. This operation is defined as follows.

$$\texttt{mg}((), ()) = ()$$
$$\texttt{mg}(\underbrace{v, S}, \underbrace{v', S'}) = \texttt{mg}'(v, v'), \texttt{mg}(S, S')$$

The operation $\texttt{mg}'$ merges two strings or elements, defined as follows.

$$\texttt{mg}'(\mathit{str}^u, \mathit{str}'^{u'}) = \begin{cases} \mathit{str}^u, \text{if } u \neq \texttt{non} \text{ and } u' = \texttt{non} \\ \mathit{str}'^{u'}, \text{if } u = \texttt{non} \text{ and } u' \neq \texttt{non} \\ \mathit{str}^u, \text{if } u = u' \text{ and } \mathit{str} = \mathit{str}' \\ \texttt{fail}, \text{otherwise} \end{cases}$$
$$\texttt{mg}'(\texttt{<}\mathit{tag}^u\texttt{>}[S], \texttt{<}\mathit{tag}'^{u'}\texttt{>}[S']) = \texttt{<}\mathit{tag}''^{u''}\texttt{>}[S'']$$
$$\text{where } S'' = \texttt{mg}(S, S')$$
$$\mathit{tag}''^{u''} = \texttt{mg}'(\mathit{tag}^u, \mathit{tag}'^{u'})$$

The $\texttt{mg}'$ operation fails if the updates of two strings are conflicting or two elements contain conflicting updates on tags or some text contents. In this case, the backward execution terminates immediately with the special value `fail`.

3.5 Programming Examples

To help understand this language, we give two programming examples in this section. The first example uses this language to implement the recursive `toc` function in Figure 1, which is divided into several pieces for the convenience of reading. The program is given in Figure 4. The function body first gets the contents of the input element. The contents consist of the author, title, section and other elements. Next, only section elements are chosen, and for each section element, the code X0 is used to construct the section element in the view with the help of code X1 and X2, 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 X1 except that the construct `xchild` should be replaced by `xattribute` in our implementation.

The `child` axis of XPath is primitively defined by `xchild` in the bidirectional language. In the second example, we define another useful axis in XPath, the `descendant` axis. This axis returns all descendant nodes of the input element. The function `xdes` below is for this axis in the bidirectional language. It is not difficult to define other XPath axes, such as `descendant-or-self`, in this language.

```
fun xdes($elm) =
  xvar $elm; xchild;
  xlet $cnt (xvar $cnt || (xvar $cnt; X))
where
  X = xmap (xlet $cnt1 (xfunapp xdes [xvar $cnt1]))
```

3.6 Property of Bidirectional Execution

For the language in this section, a successful backward execution will yield the updated source data which contains all modifications or deletions made on views. To state this property precisely, we assume that all strings and element tags in the source data are annotated with unique identifiers *id*; all strings and element tags in the arguments of `xconst` (or the results of non-invertible functions) have the special identifier `c`. A single value with the identifier I is written as str_I^u or <tag_I^u>$[S]$, where S is also annotated with appropriate identifiers. The identifiers are kept unchanged during transformations and updating views.

Based on these assumptions, the views produced by forward executions also contain strings or elements annotated with identifiers. If a value has the identifier `c`, then it origins from the arguments of `xconst`; if it has the identifier *id*, then it comes from the source data.

Suppose S is the original source data or view, S' the updated source data or view. The operation $S \rhd S' \Rightarrow U$ returns all updates U in S' with respect to S. The updates in the set U has the form $(I, str, str', \texttt{mod})$ (or $(I, tag, tag', \texttt{mod})$) meaning that the string *str* (or the tag *tag*) with identifier I is modified to *str* (or *tag'*), or the form $(I, \texttt{del})$ meaning that the string or element with identifier I is deleted.

$$str_I^{\texttt{non}} \rhd str_I^{\texttt{non}} \Rightarrow \phi$$

$$str_I^{\texttt{non}} \rhd str'^{\texttt{mod}}_I \Rightarrow \{(I, str, str', \texttt{mod})\}$$

$$str_I^{\texttt{non}} \rhd str_I^{\texttt{del}} \Rightarrow \{(I, \texttt{del})\}$$

$$\frac{S \rhd S' \Rightarrow U}{\texttt{<}tag_I^{\texttt{non}}\texttt{>}[S] \rhd \texttt{<}tag_I^{\texttt{non}}\texttt{>}[S'] \Rightarrow U}$$

$$\frac{S \rhd S' \Rightarrow U}{\texttt{<}tag_I^{\texttt{non}}\texttt{>}[S] \rhd \texttt{<}tag'^{\texttt{mod}}_I\texttt{>}[S'] \Rightarrow \{(I, tag, tag', \texttt{mod})\} \cup U}$$

$$\frac{S \rhd S' \Rightarrow U}{\texttt{<}tag_I^{\texttt{non}}\texttt{>}[S] \rhd \texttt{<}tag_I^{\texttt{del}}\texttt{>}[S'] \Rightarrow \{(I, \texttt{del})\} \cup U}$$

$$\frac{v_i \rhd v_i' \Rightarrow U_i \quad (1 \le i \le n)}{v_1, ..., v_n \rhd v_1', ..., v_n' \Rightarrow U_1 \cup ... \cup U_n}$$

Note that the above operation $\rhd$ requires that the modified strings or tags in S' should be annotated with the `mod` flag and that the deleted values should not be modified at the same time, otherwise this operation cannot return the set of updates since no rule above can be applied. If the set U of a view contains updates with the identifier `c`, then the backward execution with this view will fail since such values are not allowed to change. The property of bidirectional execution is stated below.

THEOREM 1 (Property of Bidirectional Execution). Suppose X is a bidirectional program, S is the source data and ϕ is the empty context $\mathcal{C}$ or $\mathcal{E}$. If $[\![X]\!]_\phi(S) = V$, $V \rhd V' \Rightarrow U_v$ and $[\![X]\!]_\phi(S, V') = (S', \mathcal{E}')$, then $\mathcal{E}' = \phi$ and $U_s = U_v$, where $S \rhd S' \Rightarrow U_s$. □

This theorem can be proved by induction over each language construct with the help of the following two lemmas: LEMMA 2 is used when proving the construct `xvar`, which depends on the `mg` operation in its backward semantics; LEMMA 3 is used when proving those constructs, such as `xlet` and `xsetcnt`, which update their contexts after backward executions.

Var	::=	*NCName*
Expr	::=	*String* | () | *Expr*, *Expr* | $*Var*
		| `for` $*Var* `in` *Expr* `return` *Expr*
		| `let` $*Var* := *Expr* `return` *Expr*
		| `if` (*Expr*) `then` *Expr* `else` *Expr*
		| *Expr op Expr* | *Axis NodeTest*
		| `element` *NCName* {*Expr*}
		| *NCName* ($Expr_1$, ..., $Expr_n$)
op	::=	+ | < | = | >
Axis	::=	`child` :: | `descendant` :: | `self` ::
NodeTest	::=	*NCName* | ∗ | `text()` | `node()`
FunDec	::=	`function` *NCName*(*ArgList*){*Expr*}
ArgList	::=	$$Var_1$, ..., $$Var_n$

Figure 5. Syntax of the XQuery Core

LEMMA 2 Suppose S is the source data, and S_1 and S_2 are two updated replicas of S. If $S \rhd S_1 \Rightarrow U_1$, $S \rhd S_2 \Rightarrow U_2$, S'= $\texttt{mg}(S_1, S_2)$ and $S' \neq \texttt{fail}$, then $U_1 \cup U_2 = U_{s'}$, where $S \rhd S' \Rightarrow U_{s'}$. □

LEMMA 3 Suppose X is a bidirectional program, S is the source data and $\mathcal{E}$ is a evaluation context where each variable is mapped to a pair of same values. If $[\![X]\!]_{\mathcal{E}.1}(S) = V$, $V \rhd V' \Rightarrow U_v$ and $[\![X]\!]_{\mathcal{E}}(S, V') = (S', \mathcal{E}')$, then $Dom(\mathcal{E}) = Dom(\mathcal{E}')$ and $U_\mathcal{E} \cup U_s = U_v$, where $S \rhd S' \Rightarrow U_s$, and $U_\mathcal{E}$ is the union of all U in the set $\{U | \mathcal{E}'(Var).1 \rhd \mathcal{E}'(Var).2 \Rightarrow U, Var \in Dom(\mathcal{E}')\}$. □

4. Interpreting XQuery

The expressions of XQuery can be normalized to the equivalent expressions in XQuery Core, for instance, by Galax XQuery engine [2]. The syntax of the XQuery core is more compact. Hence, like the work [15], we implement bidirectional XQuery based on XQuery Core syntax.

4.1 Syntax of XQuery Core

The syntax of the XQuery Core presented in this paper is given in Figure 5. In this syntax, the XPath axes, `child`, `descendant` and `self`, implicitly use the reserved variable $*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, the technique used in the previous section is difficult to implement reverse axes 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 [16] can be used to rewrite path expressions with reverse axes into equivalent reverse-axis-free ones.

XQuery also includes a lot of predefined functions, such as `fn:data` and `fn:subsequence`. In order to process all XQuery expressions, we must define the bidirectional versions of these functions in the underlying bidirectional language. The functions `fn:data` and `fn:subsequence` have been supported in our implementation. Although we have not implemented all predefined functions in XQuery, we believe that it is possible to achieve this goal. The basic idea is that if a function is invertible, then we define its forward and backward semantics, otherwise its backward semantics does not change the original source data, just like the `xconst` transformation.

4.2 The Translation

Figure 6 gives the rules for translating XQuery Core expressions into the code of the bidirectional language. With such interpretation, XQuery Core expressions can also execute in two directions: generating the view in the forward direction and putting view updates back in the backward direction. The translation is not diffi-

$$
\begin{array}{lcl}
[\![\mathit{String}]\!]_{\mathcal{I}} & = & \texttt{xconst}\ \mathit{String}^{\mathrm{non}} \\
[\![()]\!]_{\mathcal{I}} & = & \texttt{xconst}\ () \\
[\![\mathit{Expr}_1, \mathit{Expr}_2]\!]_{\mathcal{I}} & = & [\![\mathit{Expr}_1]\!]_{\mathcal{I}} || [\![\mathit{Expr}_2]\!]_{\mathcal{I}} \\
[\![\$\mathit{Var}]\!]_{\mathcal{I}} & = & \texttt{xvar}\ \$\mathit{Var} \\
[\![\texttt{for}\ \$\mathit{Var}\ \texttt{in}\ \mathit{Expr}_1\ \texttt{return}\ \mathit{Expr}_2]\!]_{\mathcal{I}} & = & [\![\mathit{Expr}_1]\!]_{\mathcal{I}}; \texttt{xmap}\ (\texttt{xlet}\ \$\mathit{Var}\ [\![\mathit{Expr}_2]\!]_{\mathcal{I}}) \\
[\![\texttt{let}\ \$\mathit{Var} = \mathit{Expr}_1\ \texttt{in}\ \mathit{Expr}_2]\!]_{\mathcal{I}} & = & [\![\mathit{Expr}_1]\!]_{\mathcal{I}}; \texttt{xlet}\ \$\mathit{Var}\ [\![\mathit{Expr}_2]\!]_{\mathcal{I}} \\
[\![\texttt{if}\ (\mathit{Expr})\ \texttt{then}\ \mathit{Expr}_1\ \texttt{else}\ \mathit{Expr}_2]\!]_{\mathcal{I}} & = & \texttt{xif}\ [\![\mathit{Expr}]\!]_{\mathcal{I}}\ [\![\mathit{Expr}_1]\!]_{\mathcal{I}}\ [\![\mathit{Expr}_2]\!]_{\mathcal{I}} \\
[\![\mathit{Expr}_1\ \mathit{op}\ \mathit{Expr}_2]\!]_{\mathcal{I}} & = & \mathit{xop}\ [\![\mathit{Expr}_1]\!]_{\mathcal{I}}\ [\![\mathit{Expr}_2]\!]_{\mathcal{I}} \\
[\![\mathit{Axis}\ \mathit{NodeTest}]\!]_{\mathcal{I}} & = & [\![\mathit{Axis}]\!]_{\mathcal{I}}; [\![\mathit{NodeTest}]\!]_{\mathcal{I}} \\
[\![\texttt{child}\ ::]\!]_{\mathcal{I}} & = & \texttt{xvar}\ \$\mathit{dot}; \texttt{xchild} \\
[\![\texttt{descendant}\ ::]\!]_{\mathcal{I}} & = & \texttt{xfunapp}\ \texttt{xdes}\ [\texttt{xvar}\ \$\mathit{dot}] \\
[\![\texttt{self}\ ::]\!]_{\mathcal{I}} & = & \texttt{xvar}\ \$\mathit{dot} \\
[\![\mathit{NCName}]\!]_{\mathcal{I}} & = & \texttt{xmap}\ (\texttt{xif}\ (\texttt{xwithtag}\ \mathit{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{xistext}\ \texttt{xid}\ (\texttt{xconst}\ ())) \\
[\![\texttt{node}()]\!]_{\mathcal{I}} & = & \texttt{xid} \\
[\![\texttt{element}\ \mathit{NCName}\ \{\mathit{Expr}\}]\!]_{\mathcal{I}} & = & \texttt{xconst}\ \texttt{<}\mathit{NCName}^{\mathrm{non}}\texttt{>}[]; \texttt{xsetcnt}\ [\![\mathit{Expr}]\!]_{\mathcal{I}} \\
[\![\mathit{NCName}\ (\mathit{Expr}_1, ..., \mathit{Expr}_n)]\!]_{\mathcal{I}} & = & \texttt{xfunapp}\ \mathit{NCName}\ [[\![\mathit{Expr}_1]\!]_{\mathcal{I}}, ..., [\![\mathit{Expr}_n]\!]_{\mathcal{I}}]
\end{array}
$$

Figure 6. Translation of XQuery Core Expression

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

Figure 7. Syntax of Types

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

$$
\frac{}{\Gamma; \tau \vdash \texttt{xconst}\ S : \texttt{mkTy}(S) \Rightarrow \texttt{xconst}\ S}
$$

$$
\frac{\tau' = \Gamma(\mathit{Var})}{\Gamma; \tau \vdash \texttt{xvar}\ \mathit{Var} : \tau' \Rightarrow \texttt{xvar}^{\tau'}\ \mathit{Var}}
$$

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

$$
\frac{\tau = \texttt{<}\mathit{tag}_1\texttt{>}[\tau_1] | ... | \texttt{<}\mathit{tag}_n\texttt{>}[\tau_n] \quad \Gamma; () \vdash X : \tau' \Rightarrow X'}{\Gamma; \tau \vdash \texttt{xsetcnt}\ X : \texttt{<}\mathit{tag}_1\texttt{>}[\tau'] | ... | \texttt{<}\mathit{tag}_n\texttt{>}[\tau'] \Rightarrow \texttt{xsetcnt}\ X'}
$$

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

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

$$
\frac{\Gamma; \tau \vdash_{\mathtt{m}} \texttt{xmap}\ X : \tau' \Rightarrow \tau'' \quad \Gamma; \tau'' \vdash X : \tau''' \Rightarrow X'}{\Gamma; \tau \vdash \texttt{xmap}\ X : \texttt{rmbar}(\tau') \Rightarrow \texttt{xmap}^{\tau'}\ X'}
$$

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

$$
\frac{P \in \{\texttt{xiselement}, \texttt{xistext}, \texttt{xwithtag}\ \mathit{str}\}}{\Gamma; \tau \vdash P : \texttt{string} | () \Rightarrow P}
$$

$$
\frac{\Gamma[\mathit{Var} \mapsto \tau]; () \vdash X : \tau' \Rightarrow X'}{\Gamma; \tau \vdash \texttt{xlet}\ \mathit{Var}\ X : \tau' \Rightarrow \texttt{xlet}\ \mathit{Var}\ X'}
$$

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

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

Figure 8. Typing Rules

cult due to the expressiveness of the target language. Some of these 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 an `xlet` with the arguments the variable $\$\mathit{Var}$ and the translation result of the subexpression Expr_2. That is, the variable $\$\mathit{Var}$ is bound to each value in the sequence returned by $[\![\mathit{Expr}_1]\!]_{\mathcal{I}}$, and then used in $[\![\mathit{Expr}_2]\!]_{\mathcal{I}}$.

The operator *xop* represents the sum (+) and the comparison operators ($<, =$ and $>$) in the bidirectional language. They are all defined in a style similar to `xconst`.

In the XQuery Core, the expression $\mathit{Axis}\ \ \mathit{NodeTest}$ means the axis *Axis* first produces a list of nodes from its context node, and then from this list the node test *NodeTest* selects the nodes satisfying some condition. 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 $\$\mathit{dot}$, and then the translation results of *Axis* and *NodeTest* are composed.

An XQuery function declaration of the form:

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

is translated into the following function declaration in the bidirectional language:

$$\texttt{fun}\ \mathit{NCName}(\$\mathit{Var}_1, ..., \$\mathit{Var}_n) = [\![\mathit{Expr}]\!]_{\mathcal{I}}$$

The translation defined in Figure 6 satisfies the following property, which says that the translation preserves the semantics of XQuery Core.

THEOREM 4 (Correctness of Translation). Let $\mathcal{C}$ be a context that maps variables to XML values. If an XQeury Core expression *Expr* is evaluated to a value under $\mathcal{C}$, then the expression $[\![[\![\mathit{Expr}]\!]_{\mathcal{I}}]\!]_{\mathcal{C}}(())$ is also evaluated to the same value. □

This theorem can be proved by induction over each translation rule.

5. The Type System

This type system serves two purposes. The first is, as usual, to guarantee the bidirectional programs are type-correct. For example, the `xchild` cannot be applied to a text node. The second is to annotate

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

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

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

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

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

Figure 9. Typing Rules for `xmap`

$$\text{let } P = \mathtt{xwithtag}\ str$$

$$\begin{array}{lcl}
\mathtt{T}(\tau, P) & = & (), \text{where } \tau \in \{(), \mathtt{string}, \tau*\} \\
\mathtt{T}(\mathtt{<}tag\mathtt{>}[\tau], P) & = & \begin{cases} \mathtt{<}tag\mathtt{>}[\tau], & \text{if } str = tag \\ (), & \text{otherwise} \end{cases} \\
\mathtt{T}(\underbrace{\tau_1, \tau_2}_{}, P) & = & \begin{cases} \mathtt{T}(\tau_1, P), & \text{if } \tau_2 = () \\ \mathtt{T}(\tau_2, P), & \text{if } \tau_1 = () \\ (), & \text{otherwise} \end{cases} \\
\mathtt{T}(\tau_1 | \tau_2, P) & = & \mathtt{T}(\tau_1, P) | \mathtt{T}(\tau_2, P)
\end{array}$$

Figure 10. The operator `T` for `xwithtag`

some program constructs with types, and the type information will be used to process insertions in next section.

The syntax of types is given in Figure 7, which is just the regular expression types in [12]. As the definition there, the notation $S \in \tau$ means the value S has the type τ. 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 later development.

The typing rules for the bidirectional transformation language is defined in Figure 8. The judgment has the form $\Gamma; \tau \vdash X : \tau' \Rightarrow X'$, meaning that under the typing context Γ, if the source data has the type τ, the transformation X will generate a view with the type τ'. The transformation X' is the result of annotating X with types. The typing context Γ maps variables to their types or function names together with the types of their arguments to their view types.

The transformation `xconst` always returns a constant view, so its type is just this constant with each text content replaced by a `string` type. This is done by the operator `mkTy`. The transformation `xvar` is annotated with the type of the variable it concerns, `xchild` is annotated with the type of its source data, and the parallel composition is annotated with the view types of its two argument transformations.

The transformation `xmap` applies its argument transformation X to each single value in the source data. Therefore, in its typing rule, we need to identify each `string` and element type in the source data type of `xmap`, and then use it as the source data type to check X. The view type of `xmap` is just its source data type with each `string` or element type replaced by the view type resulting from it. This typing procedure is defined by rules in Figure 9. These rules also collect all `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 will be annotated with all possible `string` or element types. The view type of `xmap` generated by the rules in Figure 9 contains a special star $\bar{*}$, which tells that the type component modified by it corresponds to a `string` type or element type, which is also modified by $*$ in the source data type. The view type with $\bar{*}$ is only used to annotate `xmap`, and otherwise $\bar{*}$ is changed into $*$ by the operator `rmbar`. The example below illustrates that the refined $*$ can help update the source data in a more reasonable way for the inserted values. The use of refined $*$ is given in Figure 13.

For example, suppose we have the code `xmap xchild`. If the source data type is `<box>`[`<apple>`[`string`]$*$], then the view type annotated on `xmap` is `<apple>`[`string`]$*$; if the source data type is `<box>`[`<apple>`[`string`]]$*$, then the view type annotated on `xmap` is `<apple>`[`string`]$\bar{*}$. This refined $*$ can tell us whether the `apple` elements in a view come from the same `box` element or from a sequence of different `box` elements. If we insert a new `apple` element on the view, which already contains a list of `apple` elements from the source data, 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 the new `box` element containing both the existing `apple` elements and the new inserted one; while in the second case, the inserted `apple` element should be processed independently by `xchild` as other existing elements, and the result is the updated source contains a new `box` element for this new `apple` element. Anyway, in both cases, the updated source data still has the valid type due to using the information provided by the refined $*$. If both the `box` and `apple` type in the source data type are modified by $*$, then `xmap` will be annotated by `<apple>`[`string`] $* \bar{*}$. For this case, both updating ways, putting the new element back into an existing or a new `box` element, generate the valid source data. In this work, we choose the first updating way since it leads to less changes on the source data.

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. This is to generate more accurate view types for each branch. The following example shows that such accuracy 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 used to check its branches, the true branch will cause a type error since `xchild` can only be applied to elements. 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. The operator $\mathtt{T}(\tau, \mathtt{xwithtag}\ str)$ selects in τ the element types with the tag *str*, which is defined in Figure 10, and the remainder types in τ is returned by the operator $\mathtt{F}(\tau, \mathtt{xwithtag}\ \ str)$. These two operators on `xiselement` and `xistext` are defined similarly. For any transformation X, both $\mathtt{T}(\tau, X)$ and $\mathtt{F}(\tau, X)$ returns τ. After type checking, `xif` is annotated with the view types of its two branches.

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 $\tau_i (1 \leq i \leq n)$, 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, but because we want to avoid the trouble of managing different versions of the same function with different type annotations. Our approach is to annotate function calls with the types of their arguments, and then use these types to type check and annotate function bodies when meeting with function calls at runtime.

The soundness property of this type system is stated as follows. This property concerns both the forward and backward behaviors of well-typed programs. For the backward behavior, the type system cannot guarantee a well-type program will not fail since it cannot check conflicting and improperly updates statically.

THEOREM 5 (Soundness). Given a transformation X and a source value S, if $\phi;\tau \vdash X : \tau' \Rightarrow X'$ and $S \in \tau$, then $[\![X']\!]_\phi(S) = V$ and $V \in \tau'$; and moreover, if $V' \in \tau'$ and $[\![X']\!]_\phi(S, V') = (S', \mathcal{E}')$, then $\mathcal{E}' = \phi$ and $S' \in \tau$. □

This theorem can be proved by induction over each language construct in Figure 3.

6. Insertions

This section discusses several view updating problems caused by insertions, and shows our type-based approaches. In these approaches, values are always needed to be related to the types against which they have been validated, so that we can merge or split these values according to their type information. For this purpose, we annotate each `string` and element type with the unique index I, written as $\texttt{string}_I$ and $\texttt{<}tag_I\texttt{>}[\tau]$. After an XML value is successfully validated against a type, all strings and elements within this value are annotated with the corresponding indexes in this type. For a validated value S or a type τ, the operator $\texttt{Id}(S)$ or $\texttt{Id}(\tau)$ returns a set of indexes at the top-level of S or τ. Hence if $\texttt{Id}(S) \subseteq \texttt{Id}(\tau)$, we know the value S has the value τ.

The source data used in this section is shown below. It contains a list of books, each of which may contain a title and any number of authors and the `non` flags on values are omitted.

```
<book>[<title>[a], <author>[b]],
<book>[<title>[c], <author>[d], <author>[e]]
```

6.1 Merging Inserted Values

We use the following code to illustrate the problem when merging two inserted values. The variable `$books` in this expression is supposed to be bound to the above source data.

```
xvar $books;
xmap (xlet $b (const <item>[] ; xsetcnt (X1||X2)))
  where
    X1 = xvar $b; xchild;
         xmap (xif (xwithtag "title") xid xconst ())
    X2 = xvar $b; xchild;
         xmap (xif (xwithtag "author") xid xconst ())
```

The view of the above code consists of two items. Consider the following updated view.

$$\begin{array}{l}\texttt{<item>}[\texttt{<title>}[\text{a}], \texttt{<author>}[\text{b}]],\\ \texttt{<item>}[\texttt{<title>}[\text{c}], \texttt{<author>}[\text{d}]], \texttt{<author>}[\text{e}],\\ \texttt{<item}^{\texttt{ins}}\texttt{>}[\texttt{<title}^{\texttt{ins}}\texttt{>}[\text{f}^{\texttt{ins}}], \texttt{<author}^{\texttt{ins}}\texttt{>}[\text{g}^{\texttt{ins}}]]\end{array}$$

Then, during the backward execution of the above code, the expression `X1` and `X2` will deal with the `title` and `author` elements in the view, respectively. For the third item, since it is not generated from a book in the original source, the variable `$b` is not bound to any value at the beginning of the backward execution of `xlet` above. The backward execution for processing the third item is sketched as follows. First, `X1` is used to deal with the inserted `title` element. After its backward execution the value of `$b` can be simply set to a new `book` element containing only the inserted `title` since `$b` has no valid value yet. Next, `X2` is used to deal with the inserted `author` element. When executing the backward execution of `xvar` in `X2`, its view is a new `book` element containing only the inserted `author` element. Obviously, the merging operation used by `xvar` should merge this view with the existing value of `$b`, that is, to build a new `book` element containing both the inserted `title` and `author` elements. In this example, it seems that we can merge these two `book` elements just by putting their contents together in the order they appear on the view. However, this is probably not true for other cases. For example, the order of contents on a view perhaps is different from that in the source, and contents are perhaps needed to be merged further into a new content.

$$\begin{array}{l}
\texttt{mg}_{\texttt{ins}}(S_1, (), \tau) = S_1\\
\texttt{mg}_{\texttt{ins}}((), S_2, \tau) = S_2\\
\texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau*) = \underbrace{S_1, S_2}\\
\texttt{mg}_{\texttt{ins}}(str_I^{\texttt{ins}}, str'^{\texttt{ins}}_I, \texttt{string}_I) = \begin{cases} str_I^{\texttt{ins}}, & \text{if } str = str' \\ \texttt{fail}, & \text{otherwise}\end{cases}\\
\texttt{mg}_{\texttt{ins}}(\texttt{<}tag_I^{\texttt{ins}}\texttt{>}[S_1], \texttt{<}tag_I^{\texttt{ins}}\texttt{>}[S_2], \texttt{<}tag_I\texttt{>}[\tau]) = \texttt{<}tag_I^{\texttt{ins}}\texttt{>}[S']\\
\quad \text{where } S' = \texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau)\\
\texttt{mg}_{\texttt{ins}}(\underbrace{S_1, S'_1}, \underbrace{S_2, S'_2}, \underbrace{\tau_1, \tau_2}) = \texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau_1), \texttt{mg}_{\texttt{ins}}(S'_1, S'_2, \tau_2)\\
\quad \text{where } \texttt{Id}(S_1) \subseteq \texttt{Id}(\tau_1), \texttt{Id}(S_2) \subseteq \texttt{Id}(\tau_1),\\
\qquad\quad \texttt{Id}(S'_1) \subseteq \texttt{Id}(\tau_2) \text{ and } \texttt{Id}(S'_2) \subseteq \texttt{Id}(\tau_2).\\
\texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau_1|\tau_2) =\\
\quad \begin{cases} \texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau_1), & \text{if } \texttt{Id}(S_1) \subseteq \tau_1 \text{ and } \texttt{Id}(S_2) \subseteq \tau_1\\ \texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau_2), & \text{if } \texttt{Id}(S_1) \subseteq \tau_2 \text{ and } \texttt{Id}(S_2) \subseteq \tau_2\\ \texttt{fail}, & \text{otherwise}\end{cases}
\end{array}$$

Figure 11. The Operator $\texttt{mg}_{\texttt{ins}}$

In our work, the merging operation for inserted values is guided by types, which characterize the structures of the expected merging results. These types can be obtained from the annotations of `xvar`. This merging operation is defined in Figure 11, which has a type argument except the two values to be merged. Before using the operator $\texttt{mg}_{\texttt{ins}}$, the values to be merged should be validated against the type argument, such that all strings and elements in each value can be related to the corresponding type components.

LEMMA 6 Suppose $S_1 \in \tau$ and $S_2 \in \tau$. If $S_3 = \texttt{mg}_{\texttt{ins}}(S_1, S_2, \tau)$ and $S_3 \neq \texttt{fail}$, then $S_3 \in \tau$. □

As an example, if the expected result of merging S_1 and S_2 is described by $\tau*$, then the merging result will be $\underbrace{S_1, S_2}$, which has the expected type $\tau*$ since S_1 and S_2 have been validated against $\tau*$.

6.2 Child Axis and Conditional Transformation

The transformations `xchild` and `xif` also have problems when dealing with inserted values. Recall the definitions of their backward semantics, `xchild` needs the tag of the original element to determine the tag of the updated element, and `xif` needs the source data to determine which branch transformation should be chosen.

These problems are also solved by using the annotated types. The type on `xchild` can provide information about what tag the new source element should have, and the types on `xif` can help choose the branch transformation according to which view type the inserted value has. Note that the element returned by `xchild` in this way is also annotated with the `ins` flag, and if the view types annotated on `xif` are overlapped, the first view type has priority in our implementation. The accuracy policy taken by the typing rule of `xif` can greatly reduce the possibility of overlap in practice.

6.3 Splitting Inserted Values

The operator `split` is used by both `xmap` and the parallel composition to divide their views into subsequences, and then each subsequence is used as a view to perform the backward transformation. When a view does not include inserted values, it can be divided

$$\begin{array}{ll}
\texttt{split}((), [], \tau) & = () \\
\texttt{split}(str_I^u, [1], \texttt{string}_I) & = str_I^u,\ \text{if } u \neq \texttt{ins} \\
\texttt{split}(str_I^{\texttt{ins}}, [], \texttt{string}_I) & = str_I^{\texttt{ins}} \\
\texttt{split}(\texttt{<}tag_I^u\texttt{>}[V], [1], \texttt{<}tag_I\texttt{>}[\tau]) & = \texttt{<}tag_I^u\texttt{>}[V],\ \text{if } u \neq \texttt{ins} \\
\texttt{split}(\texttt{<}tag_I^{\texttt{ins}}\texttt{>}[V], [], \texttt{<}tag_I\texttt{>}[\tau]) & = \texttt{<}tag_I^{\texttt{ins}}\texttt{>}[V]
\end{array}$$

$$\texttt{split}(V, ls, \tau_1|\tau_2) = \begin{cases} \texttt{split}(V, ls, \tau_1), \text{if } \texttt{Id}(V) \subseteq \texttt{Id}(\tau_1) \\ \texttt{split}(V, ls, \tau_2), \text{if } \texttt{Id}(V) \subseteq \texttt{Id}(\tau_2) \end{cases}$$

$$\begin{array}{ll}
\texttt{split}(V, ls, \tau\bar{*}) & = \texttt{split}(V, ls, \underbrace{\tau, \tau\bar{*}}) \\
\texttt{split}(V, [l], \tau*) & = V, \text{if } \texttt{Id}(V) \subseteq \texttt{Id}(\tau*) \text{ and } \texttt{isNotIns}(V) \\
\texttt{split}(V, [], \tau*) & = V, \text{if } \texttt{Id}(V) \subseteq \texttt{Id}(\tau*) \text{ and } \texttt{isIns}(V)
\end{array}$$

$$\texttt{split}(V, ls, \underbrace{\tau_1, \tau_2}) = \begin{cases} \texttt{split}(V, ls, \tau_1), \text{if } \texttt{Id}(V) \subseteq \texttt{Id}(\tau_1) \\ \texttt{split}(V, ls, \tau_2), \text{if } \texttt{Id}(V) \subseteq \texttt{Id}(\tau_2) \end{cases}$$

$$\texttt{split}(V, [], \underbrace{\tau_1, \tau_2}) = \texttt{split}(V_1, [], \tau_1), \texttt{split}(V_2, [], \tau_2), \text{if } \texttt{isIns}(V)$$
$$\text{where } \texttt{Id}(V_1) \subseteq \texttt{Id}(\tau_1), \texttt{Id}(V_2) \subseteq \texttt{Id}(\tau_2) \text{ and } \underbrace{V_1, V_2} = V.$$

$$\texttt{split}(V, ls, \underbrace{\tau_1, \tau_2}) = \begin{cases} \texttt{split}(V_1, [], \tau_1), \texttt{split}(V_2, ls, \tau_2), \text{if } \texttt{isIns}(V_1) \text{ and } \texttt{isNotIns}(V_2) \\ \texttt{split}(V_1, ls, \tau_1), \texttt{split}(V_2, [], \tau_2), \text{if } \texttt{isNotIns}(V_1) \text{ and } \texttt{isIns}(V_2) \end{cases}$$
$$\text{where } \underbrace{V_1, V_2} = V, \texttt{Id}(V_1) \subseteq \texttt{Id}(\tau_1) \text{ and } \texttt{Id}(V_2) \subseteq \texttt{Id}(\tau_2).$$

$$\texttt{split}(V, [l_1, ..., l_n], \underbrace{\tau_1, \tau_2}) = \begin{cases} (), \texttt{split}(V, [l_2, ..., l_{n-1}], \underbrace{\tau_1, \tau_2}), (), & \text{if } l_1 = 0 \text{ and } l_n = 0 \\ (), \texttt{split}(V, [l_2, ..., l_n], \underbrace{\tau_1, \tau_2}), & \text{if } l_1 = 0 \text{ and } l_n \neq 0 \\ \texttt{split}(V, [l_1, ..., l_{n-1}], \underbrace{\tau_1, \tau_2}), (), & \text{if } l_1 \neq 0 \text{ and } l_n = 0 \\ \texttt{split}(V_1, [l_1, ..., l_k], \tau_1), \texttt{split}(V_2, [l_{k+1}, ..., l_n], \tau_2), & \text{if } l_1 \neq 0 \text{ and } l_n \neq 0 \end{cases}$$
$$\text{where } \underbrace{V_1, V_2} = V, \texttt{Id}(V_1) \subseteq \texttt{Id}(\tau_1), \texttt{Id}(V_2) \subseteq \texttt{Id}(\tau_2), \texttt{isNotIns}(V_1), \texttt{isNotIns}(V_2),$$
$$\texttt{lenNoIns}(V_1) = l_1 + ... + l_k \text{ and } \texttt{lenNoIns}(V_2) = l_{k+1} + ... + l_n\ (1 \leq k \leq n).$$
$$\texttt{split}(V, ls, \tau) = \texttt{fail}, \text{if no above case can be applied.}$$

Figure 13. The Operator `split` for `xmap`

$$\texttt{split}(V, [], [\tau_1, \tau_2]) = V_1, V_2$$
$$\text{where } \texttt{isIns}(V), \texttt{Id}(V_1) \subseteq \texttt{Id}(\tau_1), \texttt{Id}(V_2) \subseteq \texttt{Id}(\tau_2) \text{ and } \underbrace{V_1, V_2} = V.$$
$$\texttt{split}(V, [l_1, l_2], [\tau_1, \tau_2]) = V_1, V_2$$
$$\text{where } \texttt{isNotIns}(V), \texttt{Id}(V_1) \subseteq \texttt{Id}(\tau_1), \texttt{Id}(V_2) \subseteq \texttt{Id}(\tau_2), \underbrace{V_1, V_2} = V \text{ and the following } P \text{ holds for } 1 \leq i \leq 2.$$
$$P = \begin{cases} \texttt{lenNoIns}(V_i) = l_i, & \text{if } l_i > 0 \\ V_i = (), & \text{if } l_i = 0 \end{cases}$$
$$\texttt{split}(V, ls, [\tau_1, \tau_2]) = \texttt{fail}, \text{if no above case can be applied.}$$

Figure 12. The Operator `split` for Parallel Composition

precisely according to the expected length for each subsequence computed from the original source data. When the view includes inserted values, the length information tells nothing about how to divide them. The following example illustrates that an elegant splitting mechanism is needed for views with inserted values.

For the above source data, the code `xmap xchild` produces a view consisting of a sequence of titles and authors of each book. Consider the following updated view.

<title>[a], <author>[b],<author$^{\texttt{ins}}$>[], <title$^{\texttt{ins}}$>[],
<author$^{\texttt{ins}}$>[], <title>[c], <author>[d], <author>[e]

In the backward execution of code `xmap xchild`, this view is first divided into subsequences and then each of them is used as the updated view of `xchild`. For this example, it is reasonable to split the updated view into three subsequences: the first three elements, the next two, and the last three. Thus, in the updated source data, the first book is inserted with a new author, the second book is new and contains the inserted title and author, and the third book is unchanged. This example can also be found at [1].

The revised `split` for the parallel transformation is given in Figure 12. It takes three arguments: the first is the updated view to be split; the second is a list of integers, each of which indicates the number of values expected by a subsequence; the third one $[\tau_1, \tau_2]$ contains the view types for the two composed transformations. The first case is applied when V contains only inserted values, guarded by the predicate `isIns`(V). In this case, the second argument is an empty list since there is no original source data available to compute this integer list. The second case,guarded by the predicate `isNotIns`(V), is applied when V contains both the values computed from the original source data and inserted values. The operator `lenNoIns`(V_i) returns the length of V_i without counting inserted values. Before being split, the updated view should be validated against the sequence type τ_1, τ_2. Note that in the second case, if a subsequence is expected to have the zero length, then it must be the empty sequence () and cannot include any inserted values. This is because the zero length means the original source data for this subsequence is hidden by using the code `xconst` (), which does not accept a changed view. This `split` operator only produces two subsequences, corresponding to the two composed transformations.

The revised `split` for `xmap` is given in Figure 13. Its first argument is the updated view to be split; the second also contains integers for the expected length of each subsequence; the third is the view type of `xmap`. The splitting procedure is directed by the view type. To illustrate this operator, we take the cases for the type τ_1, τ_2 as examples: the first case means V belongs to either τ_1 or τ_2; the second case is applied when V contains only inserted values; the third case is used to separate the inserted subsequence at the head or the tail of V; otherwise, the fourth case is applied, which either produces an empty subsequence () for each zero integer in the second argument *ls*, or divides both V and *ls* into two parts such that the first (or second) part of V belongs to τ_1 (or τ_2) and its length without counting inserted values should be same as the sum of integers in the first (or second) part of *ls*. A type-correct view probably cannot be split sensibly. For example, if a new title is inserted between an original title and its following original authors in the view of code `xmap xchild`, then splitting this view will fail. This is an example of improper insertions. A view should be validated against with the view type before splitting. Note that when applying the rule for the type $\tau\bar{*}$, the indexes on the type $\underbrace{\tau, \tau\bar{*}}$ should be re-assigned such that they are unique among τ and $\tau\bar{*}$, and then V needs to be validated against this type to get new indexes. The number of subsequences produced by this `split` can be greater than the length of the original source data, but never less than. This means that `xmap` can insert new values in its source data after backward executions.

7. Implementation

The approach proposed in this work has been implemented in Java with JDOM. Our system is available at [1], where several XQuery Core examples are also provided. Most of these examples are obtained by normalizing XQuery use cases from the W3C draft [8] with the Galax XQuery engine [2].

Our implementation supports more XQuery Core syntax than we present 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 [17], which contains some higher order XML transformation combinators.

In our implementation, users only need to mark the top element of inserted or deleted elements with `ins` or `del` flags, and other flags can be derived by the system automatically. This prototype implementation is not used to benchmark the performance of our approach since the implementation itself can be improved and the code generated directly from XQuery Core has much space to optimize.

8. Related Work

The main 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 related work in the first aspect includes [10, 5, 13]. They cannot be used directly to interpret XQuery for the following reasons. First, these languages do not have variable binding mechanisms, 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, the current languages only allow 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 interpreting XQuery. For example, XPath axes are difficult to interpret in these languages.

The work [6, 7] studies how to update the relational database through XML views, rather than update XML data like our work. They use 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 in [14] also uses programming language technique to solve the view updating problem. But their view definition language is not bidirecitonal, so when defining a view, users have to write the code for putting back possible updates into the source XML data.

9. Conclusion

In this paper, we design an expressive bidirectional XML transformation language, and then use it to interpret XQuery. We use this approach to address the view updating problem of XQuery. This bidirectional language has a type system, which not only guarantees the type-correctness of bidirectional programs, but also annotates the programs with appropriate type information. We also illustrate the difficulties of processing insertions on views, and give type-based approaches to address them. We have confirmed our idea in this paper by the prototype implementation and tests on some recommended XQuery use cases from W3C. Although we are motivated by interpreting XQuery, we believe that our work provides a potential technique to define general bidirectional functional languages since the bidirectional semantics of functions and some constructors for algebraic data types can be defined in this technique.

This work provides a basis for several future work. The first is to analyze bidirectional programs and tell what are valid updates on views, such that valid updates do not lead to failure during backward executions. The second is to optimize the target language code, for instance, by generating the efficient specialized backward code for a particular source data from the forward execution.

References

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

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

[3] F. Bancilhon and N. Spyratos. Updating semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

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

[5] A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational lenses: A language for updateable views. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Database Systems*. ACM Press, 2006.

[6] V. Braganholo, S. Davidson, and C. Heuser. PATAX: a framework to allow updates through XML views. *ACM Trans. Database Syst. (accepted)*.

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

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

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

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

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

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

[13] Z. Hu, S.-C. Mu, and M. 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*, pages 178–189. ACM Press, 2004.

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

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

[16] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proceedings of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

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