Bidirectionalizing XQuery

— Updating XML through Materialized XQuery View—

Dongxi LIU, Zhenjiang HU and Masato TAKEICHI

METR 2006–25 April 2006

Bidirectionalizing XQuery

— Updating XML through Materialized XQuery View —

Dongxi LIU, Zhenjiang HU and Masato TAKEICHI

Department of Mathematical Informatics
Graduate School of Information Science and Technology
The University of Tokyo
{liu,hu,takeichi}@mist.t.u-tokyo.ac.jp

April 8th, 2006

Abstract

Although several domain-specific bidirectional transformation languages have been proposed, it remains unclear how general these languages could be. In this paper, we show that XQuery, a powerful functional language used to query XML data, can be made bidirectional. Our approach consists of two steps: designing a new language for bidirectional XML transformation with more practical view updating semantics, and then interpreting XQuery with this language. As a result, an XQuery expression can be executed in two directions: in the forward direction, it generates a materialized view from a source XML document; while in the backward direction, it takes as input the modified view together with the original source document, and returns an updated source document. After backward execution, modifications on the view are reflected back into the source data. Our prototype implementation confirms the usefulness of this approach.

1 Introduction

As increasing amount of information are stored, exchanged, and presented using XML, the ability to intelligently query XML data becomes more and more important. XQuery [1] is such a query language that is designed as a standard for querying a broad spectrum of XML information sources, including both databases and documents. The role of XQuery in XML is just like that of SQL in relation database. However, XQuery still lacks an important feature that SQL has. This feature is *view update* [2, 3, 4], that is, modifications on a view can be reflected back to the underlying relational database that makes up the view. XQuery does not support view update on XML [5], and it is only able to perform *unidirectional* XML transformation.

When a view is changed, the corresponding source XML data cannot be updated automatically.

To see how view update is useful, consider, as an example, the XML data (about a book) stored in the file "book.xml" as given in Appendix A. Suppose we want to compute a table-of-content (toc) view consisting of all sections and their titles in the book. We may write the following query expression in XQuery, which is borrowed from the XQuery use cases in a W3C working draft [6].

```
<toc>
   { for $s in doc("book.xml")/book
     return local:toc($s)}
</toc>
```

Here, the local recursive function `toc` is to compute the table-of-content, and is defined as follows.

```
declare function local:toc (
  $book-or-section as element()) as element()*
{
   for $section in $book-or-section/section
   return
     <section>
        { $section/title ,
           local:toc($section)
        }
     </section>
}
```

This query will produce the following view.

```
<toc>
   <section>
      <title>Introsuction</title>
      <section>
         <title>Audience</title>
      </section>
      ...
   </section>
   <section>
      <title>A Syntax For Data</title>
      <section>
          <title>Base Tupes</title>
      </section>
       ...
   </section>
</toc>
```

You may have found that there are two spelling errors: “Introsuction” should be “Introduction” and “Tupes” should be “Types”. These errors need to be corrected. One way is to find these errors in file book.xml first, and then correct them, but this would be much more difficult if the size of the file is huge or if it is on a remote server that is not convenient to access. It would be nice if we could correct these mistakes directly on the view, and have the system to reflect these corrections to the original source data. However, this is impossible at present since the current XQuery can only generate view from source data.

In this paper, we aim to solve the problem of updating XML documents through view generated by XQuery. Our approach consists of two steps: designing a domain specific language for bidirectional XML transformation, and then interpreting XQuery with this language. With such interpretation, an XQuery expression can execute in two directions: in the forward direction, it generates a materialized view from a source XML document; while in the backward direction, it takes as input the modified view together with the original source XML document, and returns an updated source document. After backward execution, this updated document contains the modifications on view and users do not need to modify the source document directly.

There are two challenges we are facing: the first one is to design the underlying bidirectional transformation language that should be expressive enough to interpret XQuery, and the second is to define a suitable view updating semantics for this language.

There have been several attempts on designing domain-specific bidirectional transformation languages, such as FOCAL [7] for synchronizing tree data and X [8] for implementing a programmable editor, and the technique used to define these languages is to define forward and backward transformation semantics for each language construct. However, it remains open how general or expressive a bidirectional language defined using this technique could be. For our purpose, the question is wether this technique can be used to define an expressive target language to interpret XQuery. The answer is positive in our work, but with this technique enhanced by several new mechanisms. For example, our language supports variable binding, function declaration and call, and in the backward execution of a function call, each argument could be updated.

Most of the existing view updating semantics like that in [7] is basically defined with two functions, one for generating view (*get* function) and another for updating the source data (*putback* function), and these two functions must obey two laws. The first law says if a view generated by a *get* is not modified, then the corresponding *putback* function must not do any change to the source data; the second law says that for a modified view, if a *putback* function generates an updated source document, which is then transformed by the corresponding *get* function to produce a new view, then this two views should be same. However, the second law is too strong. There are mainly two cases that the second law is violated for an expressive language. In the first case, if a query creates a view that includes several copies of one value from the source data (i.e., the local dependency [8]), then only changing one copy will violate the second law, since after *putback* and *get* again, other copies in view also change. In the second case, if a view includes the data used by query conditions, then the second law is violated if these data is changed and no longer satisfies the query conditions.

Our main contributions in this paper can be summarized as follows.

- We design and implement a new bidirectional transformation language. Compared with FOCAL [7] and X [8], the new language supports some new mechanisms and features, such as supporting variable binding, function declaration and call, a type system with regular expression types, dealing with sequence values, supporting XPath expressions, allowing to construct computed elements. Introduction of variable binding and scope makes it easy to interpret XQuery, because in XQuery the evaluation result of an expression can be bound to a variable by `for` or `let`, and may then be used many times by other expressions in a valid scope. A type system with regular expression types is useful when we need to reflect some newly inserted values back into source data. Insertion causes much trouble because the backward transformation does not know for the inserted values what the source data should like after updated. Types annotated on transformation can give some information about the updated source data. So the updated source data is more reasonable than that obtained without such information.

- We introduce a new view updating semantics based on the relations between values in view and their corresponding values in the source data. By studying how an value in the updated source data is changed with respect to the corresponding values in the original source data and its changed view, we can guarantee that a backward transformation is side effect free and reflects all appropriate modifications back into the source data.

  Technically, in order to relate corresponding values between source data and view, each value in our work is annotated with two special flags: one is for uniquely identifying the value, and another is for indicating the changing status of the value, that is, whether it is modified, inserted, deleted or still unchanged. Intuitively, our basic idea is that after a source data is updated, all values in the updated source data must be equivalent to the corresponding values in view if they appear there and are changed, and other values not appeared in view must be kept same as the corresponding ones in the original source data.

- We propose a set of transformation rules to interpret XQuery by trans-

$$
\begin{array}{lcl}
\mathit{Val} & ::= & () \mid str_I^O \mid \; <tag_I^O>[\mathit{ValSeq}] \\
\mathit{ValSeq} & ::= & \mathit{Val}_1, ..., \mathit{Val}_n \\
 & & \\
I & ::= & id \mid \mathtt{x} \\
O & := & \mathtt{mod} \mid \mathtt{ins} \mid \mathtt{del} \mid \mathtt{non}
\end{array}
$$

Figure 1: Syntax of Annotated XML Values

lating XQuery expressions into the code of the underlying language. And as such, XQuery becomes a bidirectional transformation language. In our implementation with OCaml, our approach is based on XQuery Core since its syntax is more compact and can be automatically generated by Galaxy [9], an implementation of XQuery. We have proved that this interpretation preserves the semantics of XQuery.

The organization of the paper is as follows. We start by defining a new view updating semantics based on the relations between values in the source and view in Section 2. Then, after introducing an important updating operation `merge` in Section 3, we define our new bidirectional language in Section 4, and give a set of transformation rules for translating XQuery expressions to those in the new bidirectional language in Section 5. Finally, we discus the related work in Section 7, and make conclusion in Section 8.

2 View Updating Semantics

We shall propose a new updating semantics based on relations between values in the source and its corresponding view. To formalize such relations, we annotate each value with an identifier to identify where it is from, and an editing annotation to indicate how it is changed.

2.1 XML Values and Editing Operations on View

The syntax of XML values in this work is given in Figure 1. An XML value can be either a simple value *Val* including the unit value, string or element with annotations, or *ValSeq* a sequence of simple values. To save space, the end tags of XML elements are omitted and its contents are enclosed by brackets. For example, the element <author>Tom</author> is represented as <author>[Tom]. This notation is borrowed from [10]. Unit values can be inserted into or removed from a value sequence freely without changing its meaning. In this work, we assume that a value sequence does not include unit values except that it is just a unit value.

A string or the tag of an element is annotated with *I* and *O*. *I* is for identifying a value, and it can be `x` or *id*. The special annotation `x` means that the annotated value originates from the transformation code. For example, if a string is used as the parameter of a transformation, then the I annotation of this string is `x`. The annotation *id* is used to uniquely identify values in a source. After a source value is translated into a view, the view probably includes both `x`- or *id*-annotated values, and an *id* annotation may appear several times if the value with this annotation is duplicated in view.

The annotation O specifies the kind of changes on a value, which can be `non`, `mod`, `ins` or `del`. The annotation `non` is for values that are not changed yet, and the last three annotations correspond to three editing operations on view. A value is annotated with `mod` when it is modified, with `ins` when it is inserted, and with `del` when it is deleted. Note that the deleted value is not really removed from source data after backward transformations. They can be removed easily with an eraser procedure, since they have been annotated.

The I annotation is used only to illustrate our view updating semantics; it does not play a role in transformation. For brevity, we always omit I and O annotations on a value if we are not interested in them.

2.2 View Updating Semantics

As a preparation, we first introduce some notations for defining view updating semantics. The operator `anno`(*ValSeq*) returns a bag of annotation pairs (I, O), and each pair is for the I and O annotations on a string or a tag in *ValSeq*. In the following definition, $\uplus$ denotes the union of two bags, and ϕ is for an empty bag.

$$\begin{array}{lcl}
\texttt{anno}(()) & = & \phi \\
\texttt{anno}(str_I^O) & = & \{(I, O)\} \\
\texttt{anno}(<tag_I^O>[\mathit{ValSeq}]) & = & \{(I, O)\} \uplus \texttt{anno}(\mathit{ValSeq}) \\
\texttt{anno}(\mathit{Val}_1, ..., \mathit{Val}_n) & = & \texttt{anno}(\mathit{Val}_1) \uplus ... \uplus \texttt{anno}(\mathit{Val}_n)
\end{array}$$

There are several variants of the operator `anno`:

$$\begin{array}{l}
\texttt{anno}_I(\mathit{ValSeq}) = \{I | (I, O) \in \texttt{anno}(\mathit{ValSeq})\} \\
\qquad\qquad\qquad\quad \text{for } I = id \text{ or } I = \texttt{x} \\
\texttt{anno}_{\texttt{O}}(\mathit{ValSeq}) = \{id | (id, O) \in \texttt{anno}(\mathit{ValSeq})\} \\
\qquad\qquad\qquad\quad \text{for } O \in \{\texttt{mod}, \texttt{del}, \texttt{ins}, \texttt{non}\} \\
\texttt{anno}_{\texttt{chn}}(\mathit{ValSeq}) = \texttt{anno}_{\texttt{mod}}(\mathit{ValSeq}) \uplus \texttt{anno}_{\texttt{del}}(\mathit{ValSeq}) \\
\qquad\qquad\qquad\quad \uplus\, \texttt{anno}_{\texttt{ins}}(\mathit{ValSeq}).
\end{array}$$

Based on an identifier *id*, we can get the tag or the string annotated with *id* in a value *ValSeq* by using the operator `index`(*ValSeq*, *id*). This operator is defined as follows:

$$
\begin{array}{l}
\texttt{index}((), id) = () \\
\texttt{index}(str^{O}_{id}, id) = str \\
\texttt{index}(str^{O}_{id'}, id) = (), \textsf{if } id' \neq id \\
\texttt{index}(<tag^{O}_{id}>[ValSeq], id) = tag \\
\texttt{index}(<tag^{O}_{id'}>[ValSeq], id) = (), \textsf{if } id' \neq id \\
\texttt{index}(Val_1, ..., Val_n, id) = \begin{cases} \texttt{index}(Val_i, id), \\ \quad \textsf{if } \texttt{index}(Val_i, id) \neq () \\ \texttt{index}(Val_2, ..., Val_n, id) \\ \quad \textsf{otherwise} \end{cases}
\end{array}
$$

When an element is deleted, all its contents are thought to be removed, too; and when an element is inserted, all its contents should be also newly inserted. Hence, when an element has annotation `del` or `ins` in its tag, all elements and string values in it should also have the same O annotations. These annotation requirements can be checked using the following two operators: $\texttt{check}_O(ValSeq)$ holds if $\texttt{anno}_{non}(ValSeq)=\phi$ and $\texttt{anno}_O(ValSeq)=$ $\texttt{anno}_{\texttt{chn}}(ValSeq)$, where $O \in \{\texttt{del}, \texttt{ins}\}$.

We can check whether a value *ValSeq′* is properly updated or changed with respect to another value *ValSeq*, written $ValSeq \equiv ValSeq'$. Intuitively, if $ValSeq'$ is properly updated or changed, then all changes are correctly annotated, and the structure of $ValSeq'$ is still same as that of except the inserted values *ValSeq*. In the following, *str* and str' are different values, also for *tag* and tag'.

$$str^{\texttt{non}}_{I} \equiv str^{\texttt{non}}_{I}$$

$$str^{\texttt{non}}_{I} \equiv str'^{\texttt{mod}}_{I}$$

$$str^{\texttt{non}}_{I} \equiv str'^{\texttt{del}}_{I}$$

$$\frac{ValSeq \equiv ValSeq'}{<tag^{\texttt{non}}_{I}>[ValSeq] \equiv <tag^{\texttt{non}}_{I}>[ValSeq']}$$

$$\frac{ValSeq \equiv ValSeq'}{<tag^{\texttt{non}}_{I}>[ValSeq] \equiv <tag'^{\texttt{mod}}_{I}>[ValSeq']}$$

$$\frac{ValSeq \equiv ValSeq' \quad \texttt{check}_{\texttt{del}}(ValSeq) \textsf{ holds}}{<tag^{\texttt{non}}_{I}>[ValSeq] \equiv <tag'^{\texttt{del}}_{I}>[ValSeq']}$$

$$\frac{\begin{array}{c} Val''_1, ..., Val''_n = \texttt{rc}(Val'_1, ..., Val'_m) \\ Val_i \equiv Val''_i (1 \leq i \leq n) \end{array}}{Val_1, ..., Val_n \equiv Val'_1, ..., Val'_m \ (n \leq m)}$$

The above relation depends on the operator $\texttt{rc}$, which removes all inserted values from a sequence value, and also checks whether they are properly annotated.

$$\begin{array}{lcl}
\texttt{rc}(()) & = & () \\
\texttt{rc}(str_I^{\texttt{ins}}) & = & () \\
\texttt{rc}(str_I^{O}) & = & str_I^{O}, \texttt{if } O \neq \texttt{ins} \\
\texttt{rc}(<tag_I^{\texttt{ins}}>[ValSeq]) & = & (), \texttt{if check}_{\texttt{ins}}(ValSeq) \text{ holds} \\
\texttt{rc}(<tag_I^{O}>[ValSeq]) & = & <tag_I^{O}>[ValSeq], \\
 & & \quad \texttt{if}\, O \neq \texttt{ins} \\
\texttt{rc}(Val_1, ..., Val_n) & = & \texttt{rc}(Val_1), ..., \texttt{rc}(Val_n)
\end{array}$$

A value *ValSeq* is said to have no *conflict* changes provided that $\texttt{anno}_{\texttt{chn}}(ValSeq)$ does not contain repeated identifiers, that is, it is a set. A value *ValSeq* is said to a *proper initial* source value, if $\texttt{anno}_{\texttt{chn}}(ValSeq)=\phi$, $\texttt{anno}_{\texttt{x}}(ValSeq) = \phi$ and $\texttt{anno}_{id}(ValSeq)$ does not contain repeated identifiers.

For a bidirectional transformation X, $[\![X]\!]_F(S)$ indicates the forward transformation, which means to transform source value S into some view, while $[\![X]\!]_B(S, V')$ indicates the backward transformation, which takes as input the original source data S and the changed view V', and returns the updated source data S'.

Definition 1 (Consistent View Updating Semantics) Suppose S is a proper initial document, and

$$\begin{array}{lcl}
[\![X]\!]_F(S) & = & V \\
[\![X]\!]_B(S, V') & = & S'.
\end{array}$$

Then, X has a *consistent view updating semantics*, if $V \equiv V'$ and V' contains no conflict changes, then the following conditions hold.

1) $S \equiv S'$;

2) $\texttt{anno}_{\texttt{mod}}(S) = \texttt{anno}_{\texttt{mod}}(S')$;

3) $\texttt{anno}_{\texttt{del}}(S) = \texttt{anno}_{\texttt{del}}(S')$;

4) $\texttt{anno}_{\texttt{ins}}(S) = \texttt{anno}_{\texttt{ins}}(S')$;

5) $\forall id \in \texttt{anno}_{\texttt{chn}}(S').\texttt{index}(S', id) = \texttt{index}(S, id)$.

□

3 Merging Updated Values

In this section, we consider how to merge two XML values, which are supposed to be two replicas of one source value, and probably contain different changes. The merge operation will be used to define the bidirectional transformation language in next section, where some parts of the source are duplicated in view, and changes in each copy are required to be reflected back into the source.

The `merge` operation combines two values into a new value. It is defined as follows.

```
$\texttt{merge}(\underline{Val_1, ..., Val_n}, \underline{Val'_1, ..., Val'_m}) =$
    if $\texttt{isIns}(Val_1)$ and $\texttt{isIns}(Val'_1)$ then
        $Val_1, Val'_1, \texttt{merge}(ValSeq, ValSeq')$
    else if $\texttt{isNotIns}(Val_1)$ and $\texttt{isIns}(Val'_1)$ then
        $Val'_1, \texttt{merge}(\underline{Val_1, ValSeq}, ValSeq')$
    else if $\texttt{isIns}(Val_1)$ and $\texttt{isNotIns}(Val'_1)$ then
        $Val_1, \texttt{merge}(ValSeq, \underline{Val'_1, ValSeq'})$
    else $\texttt{mergeS}(Val_1, Val'_1), \texttt{merge}(ValSeq, ValSeq')$
  where $ValSeq = Val_2, ..., Val_n$
        $ValSeq' = Val'_2, ..., Val'_m$
```

Here, `mergeS` is to merge two single basic values.

```
$\texttt{mergeS}(str^O, str'^{O'}) =$ if $O \neq$ non then $str^O$ else $str'^{O'}$
$\texttt{mergeS}(<tag^O>[ValSeq], <tag'^{O'}>[ValSeq']) =$
    if $O \neq$ non then $<tag^O>[\texttt{merge}(ValSeq, ValSeq')]$
    else $<tag^{O'}>[\texttt{merge}(ValSeq, ValSeq')]$
```

In the above definition, the predicate $\texttt{isIns}(Val)$ is true if Val is annotated with `ins`, otherwise $\texttt{isNotIns}(Val)$ is true. Note that a sequence of values is marked with an underline for clear separation between two sequences.

For two properly updated replicas of one source value, the `merge` operator can reflect all changes among them into one new updated source value, if there is no conflict changes. This property is described by the following proposition.

Proposition 1 Suppose that there are three values S, S_1 and S_2, satisfying $S \equiv S_1$ and $S \equiv S_2$. Let $S' = \texttt{merge}(S_1, S_2)$. If $\texttt{anno}_{\texttt{chn}}(S_1) \cap \texttt{anno}_{\texttt{chn}}(S_2) = \phi$, and both S_1 and S_2 contain no conflict changes, then the following conditions hold.

1) $S \equiv S'$;

2) $\texttt{anno}_{\texttt{mod}}(S') = \texttt{anno}_{\texttt{mod}}(S_1) \uplus \texttt{anno}_{\texttt{mod}}(S_2)$;

$$
\begin{array}{lcl}
X & ::= & \mathit{BX} \mid \mathit{XC} \mid \mathit{EM} \mid \mathit{FC} \\
\mathit{BX} & ::= & \texttt{<xid>}[] \mid \texttt{<xconst>}[\mathit{Val}] \mid \texttt{<xchild>}[] \\
\mathit{XC} & ::= & \texttt{<xseq>}[X_1, ..., X_n] \mid \texttt{<xchcont>}[X_1, ..., X_n] \\
 & & \mid \texttt{<xmap>}[X] \mid \texttt{<xif>}[P, X_1, X_2] \\
 & & \mid \texttt{<xrename>}[X] \mid \texttt{<xlt>}[X_1, X_2] \\
\mathit{EM} & ::= & \texttt{<xstore>}[\mathit{Var}] \mid \texttt{<xload>}[\mathit{Var}] \mid \texttt{<xfree>}[\mathit{Var}] \\
\mathit{FC} & ::= & <\!\mathit{funname}\!>[X_1, ..., X_n] \\
 & & \\
P & ::= & \texttt{<xwithtag>}[\mathit{str}] \mid \texttt{<xistext>}[] \mid X
\end{array}
$$

Figure 2: Syntax of the Target Language

3) $\texttt{anno}_{\texttt{del}}(S') = \texttt{anno}_{\texttt{del}}(S_1) \uplus \texttt{anno}_{\texttt{del}}(S_2)$;

4) $\texttt{anno}_{\texttt{ins}}(S') = \texttt{anno}_{\texttt{ins}}(S_1) \uplus \texttt{anno}_{\texttt{ins}}(S_2)$;

5) $\forall I \in \texttt{anno}_{\texttt{chn}}(S_1).\texttt{index}(S_1, I) = \texttt{index}(S', I)$;

6) $\forall I \in \texttt{anno}_{\texttt{chn}}(S_2).\texttt{index}(S_2, I) = \texttt{index}(S', I)$.

$\square$

4 The Bidirectional Transformational Language

This section introduce the bidirectional transformational language which will be used to interpret XQuery. To simplify our presentation, the semantics of the language in this section does not take account of inserted values in the view. Insertion will be discussed in Section 6 with the help of types.

4.1 Syntax

The syntax of this language is defined in Figure 2, where each expression is represented as an XML element. It is a functional language. We choose XML to represent our language for two reasons. First, we implemented the language in Java, and the XML representation save us from writing its parser, Second, this language serves mainly as an intermediate language rather than as a language for users to write programs, so verbose notation is not a problem.

There are two main syntax categories: transformation X and predicate P. Transformations are divided into four kinds: *BX* for basic transformations, *XC* for transformation combinators, *EM* for transformation environment management and *FC* for function calls. Unlike the language FOCAL [7] or X [8], our language works on forest, not just on tree. A transformation can be used as a predicate, but not vice versa. In addition, each transformation in this language can be checked upon the conditions required by consistent view updating semantics.

This language supports variable bindings, which is essential to interpret XQuery, since XQuery can bind variables using `let` or `for` expressions, and then use them in subexpressions. All transformations in this language execute with a transformation environment $\mathcal{E}$, which is used to store all bindings. For the transformation X, its bidirectional semantics is defined with the following two functions.

- Forward-Direction Semantics: $[\![X]\!]_F^{\mathcal{E}}(S) = (V, \mathcal{E}')$ means transforming the source data S using X in the environment $\mathcal{E}$, and generating the view V as well as a new environment $\mathcal{E}'$.

- Backward-direction Semantics: $[\![X]\!]_B^{\mathcal{E}''}(S, V') = (S', \mathcal{E}''')$ means transforming the source data S and the changed view V' using X in the environment $\mathcal{E}''$, and generating the updated source data S' and a new environment $\mathcal{E}'''$.

Most transformations neither change nor use the environment directly, so we will omit the environment when defining transformations except for those needing this environment.

4.2 Basic Transformations

A basic transformation transforms the input data in one particular way. There are three basic transformations defined here. Others can be added to the language according to the domain-specific requirements.

xid: Let $X =$ <`xid`>[].

$$\begin{array}{lcl} [\![X]\!]_F(S) & = & S \\ [\![X]\!]_B(S, V') & = & V' \end{array}$$

`xid` is the identify transformation, where the view is identical to the source document, and also the updated source document is identical to the changed view.

xconst: Let $X =$ <`xconst`>[*Val*], where $\texttt{anno}_{id}(\mathit{Val}) = \phi$ and $\texttt{anno}_{\texttt{chn}}(\mathit{Val}) = \phi$.

$$\begin{array}{lcl} [\![X]\!]_F(S) & = & \mathit{Val} \\ [\![X]\!]_B(S, \mathit{Val}') & = & S \end{array}$$

`xconst` transforms any source document into a constant value *Val*, which is the argument of `xconst`. *Val* is required to have I annotation `x` and O annotation `non`. Backward transformation abandons all changes on view.

xchild: Let $X = $ <`xchild`>[].

$$\begin{array}{lcl} [\![X]\!]_F(<tag_I^O>[\mathit{ValSeq}]) & = & \mathit{ValSeq} \\ [\![X]\!]_B(<tag_I^O>[\mathit{ValSeq}], \mathit{ValSeq}') & = & <tag_I^O>[\mathit{ValSeq}'] \end{array}$$

`xchild` corresponds to the `child` axis in XPath. This transformation returns the contents of the input XML value. If we need to get the contents of a sequence value, we can pass `xchild` as the argument of `xmap`, a transformation combinator, which will be discussed later.

4.3 Transformation Combinators

Transformation combinators are used to build complex transformations by gluing simpler transformations together.

xseq: Let $X = $ <`xseq`>$[X_1, ..., X_n]$.

$$\begin{array}{lcl} [\![X]\!]_F(SV_0) & = & SV_n \\ [\![X]\!]_B(SV_0, SV'_n) & = & SV'_0 \\ \multicolumn{3}{l}{\texttt{where}} \\ \quad SV_i & = & [\![X_i]\!]_F(S)\ (1 \leq i \leq n) \\ \quad SV'_{j-1} & = & [\![X_j]\!]_B(SV_{j-1}, SV'_j)\ \ (1 \leq j \leq n) \end{array}$$

`xseq` takes as arguments a sequence of transformations, which are applied in sequence. In the intermediate states, SV'_i is the result of updating SV_i $(1 \leq i \leq n-1)$, which is the view for transformation X_i and the source data for transformation X_{i+1}.

xchcont: Let $X = $ <`xchcont`>$[X_1, ..., X_n]$ and $S = <tag_I^O>[\mathit{ValSeq}]$.

$$\begin{array}{lcl} [\![X]\!]_F(S) & = & <tag_I^O>[NC] \\ [\![X]\!]_B(S, <tag'^{O'}_I>[NC']) & = & <tag'^{O'}_I>[\mathit{ValSeq}] \\ \multicolumn{3}{l}{\texttt{where}} \\ \quad NC & = & V_1, ..., V_n \\ \quad V_i & = & [\![X_i]\!]_F(())\quad (1 \leq i \leq n) \\ \quad V'_1,\ \dots\ , V'_n & = & \texttt{split}(NC', [|V_1|, ..., |V_n|]) \\ \quad S'_i & = & [\![X_i]\!]_B((), V'_i)\quad (1 \leq i \leq n) \end{array}$$

`xchcont` creates the new contents NC of the source document by executing each argument transformation $X_i (1 \leq i \leq n)$. In backward transformation, the result of executing X_i does not be used, but it is still be executed for possible effect on the transformation environment. For a sequence value NC' and a list with n integers, operator $\texttt{split}(NC', [|V_1|, ..., |V_n|])$ returns n subsequences $V'_i(1 \leq i \leq n)$, such that $V'_1, ..., V'_n = NC'$ and $|V'_i| = |V_i|$ (1≤ i ≤ n), where $|V_i|$ denotes the length of the sequence value V_i.

xmap: Let $X =$ <xmap>$[X']$ and $S = Val_1, ..., Val_n$.

$$\begin{array}{lcl}
[\![X]\!]_F(S) & = & V \\
[\![X]\!]_B(S, V') & = & Val'_1, ..., Val'_n \\
\multicolumn{3}{l}{\texttt{where}} \\
\quad V & = & V_1 \ ... \ V_n \\
\quad V_i & = & [\![X']\!]_F(Val_i)(1 \le i \le n) \\
\quad V'_1, ..., V'_n & = & \texttt{split}(V', [|V_1|, ..., |V_n|]) \\
\quad Val'_i & = & [\![X']\!]_B(Val_i, V'_i)(1 \le i \le n)
\end{array}$$

`xmap` applies its argument transformation X' to each simple value $Val_i(1 \le i \le n)$ in the source value S, and puts each result V_i together as the view. Note that the number of simple values in view V is probably different from n, the number of simple values in the source value S. Hence, in backward transformation, we need to divide the changed view V' into a list of subsequences $V'_i(1 \le i \le n)$ using operator `split`, where V'_i is the changed view of the source value $Val_i(1 \le i \le n)$. When inserted elements are considered, this way of splitting a sequence value used in `xchcont` and `xmap` will be different.

xif: Let $X =$ <xif>$[P, X_1, X_2]$.

$$\begin{array}{lcl}
[\![X]\!]_F(S) & = & \begin{cases} [\![X_1]\!]_F(S) & \text{if } [\![P]\!](S) \neq () \\ [\![X_2]\!]_F(S), & \text{otherwise} \end{cases} \\
[\![X]\!]_B(S, V') & = & \begin{cases} [\![X_1]\!]_B(S, V'), & \text{if } [\![P]\!](S) \neq () \\ [\![X_2]\!]_B(S, V'), & \text{otherwise} \end{cases}
\end{array}$$

`xif` chooses X_1 or X_2 based on the result of applying predicate P to the source data S. Here, the unit value () is regarded as false (same as XQuery). Note that the result of $[\![P]\!](S)$ does not appear in the view of `xif`.

xrename: Let $X =$ <xrename>$[X']$ and $S = \texttt{<}tag_I^O\texttt{>}[ValSeq]$.

$$\begin{array}{lcl}
[\![X]\!]_F(S) & = & \texttt{<}newtag_{I'}^{O'}\texttt{>}[ValSeq] \\
[\![X]\!]_B(S, V') & = & \texttt{<}tag_I^{O'''}\texttt{>}[ValSeq'''] \\
\multicolumn{3}{l}{\texttt{where}} \\
\quad newtag_{I'}^{O'} & = & [\![X']\!]_F(S) \\
\quad V' & = & \texttt{<}newtag'^{O''}_{I'}\texttt{>}[ValSeq'] \\
\quad \texttt{<}tag'^{O'''}_I\texttt{>}[ValSeq''] & = & [\![X']\!]_B(S, newtag'^{O''}_{I'}) \\
\quad ValSeq''' & = & \texttt{merge}(ValSeq', ValSeq'')
\end{array}$$

`xrename` changes the tag of the source value S from tag_I^O to $newtag_{I'}^{O'}$, which is the result of applying the argument X' to S. In the backward transformation, the changed tag $newtag'^{O''}_{I'}$ is first used to compute an intermediate updated source $< tag'^{O'''}_{I} >[\mathit{ValSeq}'']$, and then the final updated source is constructed by using the tag $tag'^{O'''}_{I}$ and the contents ValSeq''' obtained by merging ValSeq'' and ValSeq' in the changed view.

xlt: Let $X = \texttt{<xlt>}[X_1, X_2]$.

$$\begin{array}{lcl} [\![X]\!]_F(S) & = & \left\{ \begin{array}{l} \text{“}\texttt{true}\text{”}^{\texttt{non}}_{\texttt{x}}, \\ \quad \text{if } [\![X_1]\!]_F(S) < [\![X_2]\!]_F(S) \\ (), \quad \texttt{otherwise} \end{array} \right. \\ [\![X]\!]_B(S, str^O_{\texttt{x}}) & = & S \end{array}$$

In the backward direction, like `xconst`, `xlt` just returns the original source value since the changes in view cannot be reflected back in a meaningful way. Other functions, such as `count` in XQuery, can be implemented in such way if they do not concern about changes on their views.

4.4 Transformation Environment

The environment $\mathcal{E}$ behaves like a stack (not exactly a like). There are several operators generating a new $\mathcal{E}$ by changing an old one. The operator $\texttt{push}(\mathcal{E}, [\mathit{Var} \mapsto \mathit{ValSeq}])$ pushes a new binding to the top of $\mathcal{E}$; $\texttt{pop}(\mathcal{E}, \mathit{Var})$ removes the least recent binding of variable Var from $\mathcal{E}$; $\texttt{update}(\mathcal{E}, [\mathit{Var} \mapsto \mathit{ValSeq}])$ changes the least recent mapping of variable Var such that it is mapped to the new value ValSeq.

There are three transformation constructs `xstore`, `xload` and `xfree` for changing or using the environment explicitly. To use a variable Var, there should be a pair of `xstore` and `xfree` to specify a scope, in which `xload` can be used to access the value of this variable.

xstore: Let $X = \texttt{<xstore>}[\mathit{Var}]$.

$$\begin{array}{lcl} [\![X]\!]^{\mathcal{E}}_F(S) & = & (S, \texttt{push}(\mathcal{E}, [Var \mapsto S])) \\ [\![X]\!]^{\mathcal{E}'}_B(S, V') & = & (\texttt{merge}(V', \mathcal{E}'(\mathit{Var})), \texttt{pop}(\mathcal{E}', \mathit{Var})) \end{array}$$

In the forward direction, `xstore` behaves just like the identity transformation, and besides the transformation, it also pushes a new binding of Var into the environment $\mathcal{E}$. In the backward direction, the updated source data is the result of merging the changed view V' and the value of Var, and after transformation, the binding of Var is popped off from the environment $\mathcal{E}$. The notation $\mathcal{E}(\mathit{Var})$ denotes the value of Var in $\mathcal{E}$.

xload: Let $X =$ <xload>[*Var*].

$$\begin{array}{lcl}
[\![X]\!]_F^{\mathcal{E}}(S) & = & (\mathcal{E}(\mathit{Var}), \mathcal{E}) \\
[\![X]\!]_B^{\mathcal{E}'}(S, V') & = & (S, \mathtt{update}(\mathcal{E}', [\mathit{Var} \mapsto \mathit{ValSeq}]) \\
\multicolumn{3}{l}{\mathtt{where}} \\
\quad \mathit{ValSeq} & = & \mathtt{merge}(V', \mathcal{E}'(\mathit{Var}))])
\end{array}$$

`xload` helps to use the value of a bound variable. In the forward direction, the value of *Var* is returned as the view. In the backward direction, the source value S is not to be updated, but the value of *Var* is updated.

xfree: Let $X =$ <xfree>[*Var*].

$$\begin{array}{lcl}
[\![X]\!]_F^{\mathcal{E}}(S) & = & (S, \mathcal{E}') \\
[\![X]\!]_B^{\mathcal{E}''}(S, V') & = & (V', \mathtt{push}(\mathcal{E}'', [\mathit{Var} \mapsto S']) \\
\multicolumn{3}{l}{\mathtt{where}} \\
\quad S' & = & \mathcal{E}(\mathit{Var}) \\
\quad \mathcal{E}' & = & \mathtt{pop}(\mathcal{E}, \mathit{Var})
\end{array}$$

In the forward direction, `xfree` removes the binding of *Var* from $\mathcal{E}$, while in the backward direction, this removed binding is put back onto $\mathcal{E}$ again.

4.5 Function Calls

Our language allows users to define transformation functions, which abstract common transformation patterns. A function is declared in the following form.

function declaration:

<function name = "*funname*"
    $\mathtt{arg}_1$ = "Var_1" ... $\mathtt{arg}_n$ = "Var_n">[X']

A function is represented as a `function` element, in which the `name` attribute specifies the function name *funname*, the attribute $\mathtt{arg}_i (1 \le i \le n)$ specifies the ith formal parameter $\mathit{Var}_i (1 \le i \le n)$, and transformation X' is the function body.

The semantics of functional calls are defined using the existing transformations. For a call to the function with the above form, we can define it as follows.

function call: Let $X = \text{<}\mathit{funname}\text{>}[X_1, ..., X_n]$.

$$\begin{array}{rcll} X & = & \texttt{<seq>}[& X_1, \texttt{<xstore>}[\mathit{Var}_1], \\ & & & ..., \\ & & & X_n, \texttt{<xstore>}[\mathit{Var}_n], \\ & & & \texttt{<xconst>}[], \\ & & & X', \\ & & & \texttt{<xfree>}[\mathit{Var}_n], \\ & & & ..., \\ & & & \texttt{<xfree>}[\mathit{Var}_1]\ ] \end{array}$$

That is, we first build the activation record on the stack before executing the function body, and after that, the activation record is removed. Note that the function body begins with the source data (), so it must get other nontrivial values to transform through $\texttt{xload}(\mathit{Var}_i)(1 \leq i \leq n)$. In a function call, all actual parameters are regarded as source data. So each of them may be updated after backward transformation.

4.6 Predicates

Predicates are used as the condition of the transformation `xif`. $[\![P]\!](S)$ is to judge whether P holds on S. In this work, the unit value () is regarded as a false value, and other XML value is regarded as a true value. Since the result of $[\![P]\!](S)$ does not appear in the view generated by `xif`, the predicate P is not essential to the expressiveness of our language, and it can be extended to include any other predicate, such as the type condition in `typeswitch` in XQuery, without affecting the view updating semantics of our language. In this section, we give the following predicates to be used in later sections.

xwithtag: Let $P = \texttt{<xwithtag>}[\mathit{str}_{\texttt{x}}^{\texttt{non}}]$.

$$[\![P]\!](S) \quad = \quad \begin{cases} S, & \text{if } \texttt{tag}(S) = \mathit{str} \\ (), & \texttt{otherwise} \end{cases}$$

`xwithtag` judges whether the source value S has tag *str*. The operator `tag` returns the tag of an XML element.

xiselement: Let $P = \texttt{<xiselement>}[]$.

$$[\![P]\!](S) \quad = \quad \begin{cases} S, & \text{if } S \text{ is an element} \\ (), & \texttt{otherwise} \end{cases}$$

`xiselement` holds if the source value S is an element.

xistext: Let $P = $ <`xistext`>[].

$$[\![P]\!](S) \;=\; \begin{cases} S, & \textsf{if } S \textsf{ is a string} \\ (), & \texttt{otherwise} \end{cases}$$

`xistext` holds if the source document S is a string value.

Note that any transformation X can be used as a predicate, and in this case, only its forward transformation is used. Let $P = X$.

$$[\![P]\!](S) \;=\; [\![X]\!]_F(S)$$

4.7 A Programming Example

To help understand this language, we show how to use it to implement the transformation done by XQuery in Section 1. The program is given in Figure 3. In the main expression, the construct `input` returns the root element in file "book.xml". For brevity, we sometimes omit I and O annotations in this program and other examples in later sections. In this example, all variable names and function names are same as the XQuery example, so it is convenient to compare this program with the XQuery example and make sure that they have the same query result. The difference is that this program allows users to make changes on view, and reflect these changes back into the source data.

5 Interpreting XQuery

In the implementation of XQuery, a query is first normalized to the equivalent expression in XQuery Core. XQuery Core is a subset of XQuery, in which some complex language constructs have been resolved into simpler ones. For example, an XPath expression is mapped into a list of navigation steps in XQuery Core. Our work of bidirectionalizing XQuery is based on XQuery Core, since its syntax is simpler.

5.1 Syntax of XQuery Core

The syntax of the XQuery Core used in this work is given in Figure 4, which includes two main categories: expressions and function declarations. The syntax of expressions contains: string values *String*, unit value (), sequence expressions, `for` and `let` expressions, XPath steps, less-than comparison <, conditional expressions, element constructor, and function calls. In the syntax, the meta-variable *NCName* represents a function name or an element tag. An XPath step consists of an *Axis* and a *NodeTest*. Here, the *Axis* has only `child` and `self`, and other axes can be supported if we provide the corresponding basic transformations in our underlying language. For example,

```
<function name="toc" arg1="$book-or-section">[
    <xseq>[<xload>[$book-or-section], <xchild>[],
           <xmap>[<xif>[<xwithtag>[section],
                        X, <xconst>[] ] ]
    ]
]

X =
  <xseq>[
      <xstore>[$section], <xconst>[<section>[]],
      <xchcont>[
         <xseq>[<xload>[$section], <xchild>[],
                <xif>[<xwithtag>[title],
                      <xid>[],<xconst>[]]],
         <toc>[<xload>[$section]]],
      <xfree>[$section]
  ]

MAIN =
  <xseq>[
    <input>[<source>[book.xml]],
    <xmap>[
      <xseq>[<xstore>[$s], <xconst>[<toc>[]],
        <xchcont>[<toc>[<xload>[$s]]],
        <xfree>[$s] ]
    ]
  ]
```

Figure 3: A Programming Example

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

Figure 4: Syntax of XQuery Core

to interpret the `descendant` axis, we implement a new basic transformation `xdescendant` for this language. In a function declaration, *Ty* denotes the type of each argument or the result, which is specified using XSchema.

5.2 The Translation

In this section, we will interpret XQuery Core by translating it to the target language defined in Section 4. Since the target language describes bidirectional transformations and has well-defined view updating semantics, XQuery Core with such interpretation hence gains this ability of querying source XML documents and reflecting the changes on view back into the source documents.

The translation rules for expressions are defined in Figure 5. A string value *String* and unit value () is translated into `xconst` with *String* and () as argument, and *String* is annotated with `x` and `non`. For a sequence expression $\mathit{Expr}_1, \mathit{Expr}_2$, we first construct a `virtual` element with empty content, and then use `xchcont`, taking as arguments the translation results of Expr_1 and Expr_2, to build the contents of `virtual` element, and finally return these contents by using `xchild`.

In XQeury Core, a bound variable $\$\mathit{Var}$ is referred by its name, while in the target language, it is referred by using `xload`($\$\mathit{Var}$). In the translation of `for` expression, Expr_1 is first translated, and the translation result is followed by a `xmap`, which takes as the argument a sequence of transformations `xstore`($\$\mathit{Var}$), the translation result of Expr_2 and `xfree`($\$\mathit{Var}$). That is, the variable *Var* is bound to each value in a sequence returned by transformation $[\![\mathit{Expr}_1]\!]_{\mathcal{I}}$, and valid in transformation $[\![\mathit{Expr}_2]\!]_{\mathcal{I}}$. The translation of `let` expression is similar, where the `xmap` is removed since the variable $\$\mathit{Var}$ is

$$
\begin{array}{lcl}
\llbracket \mathit{String} \rrbracket_{\mathcal{I}} & = & \texttt{<xconst>}[\mathit{String}_{\texttt{x}}^{\texttt{non}}] \\
\llbracket () \rrbracket_{\mathcal{I}} & = & \texttt{<xconst>}[] \\
\llbracket \mathit{Expr}_1, \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{<xseq>}[\texttt{<xconst>}[<\mathsf{virtual}_{\texttt{x}}^{\texttt{non}}>[]], \\
 & & \texttt{<xchcont>}[[\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}, \texttt{<xchild>}[]] \\
\llbracket \$\mathit{Var} \rrbracket_{\mathcal{I}} & = & \texttt{<xload>}[\$\mathit{Var}] \\
\llbracket \texttt{for } \$\mathit{Var} \texttt{ in } \mathit{Expr}_1 \texttt{ return } \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{<xseq>}[\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, \texttt{<xmap>}[\texttt{<xseq>}[\texttt{<xstore>}[\$\mathit{Var}], \\
 & & \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}, \texttt{<xfree>}[\$\mathit{Var}]]]] \\
\llbracket \texttt{let } \$\mathit{Var} = \mathit{Expr}_1 \texttt{ in } \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{<xseq>}[\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, \texttt{<xstore>}[\$\mathit{Var}], \\
 & & \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}, \texttt{<xfree>}[\$\mathit{Var}]] \\
\llbracket \texttt{if } (\mathit{Expr}) \texttt{ then } \mathit{Expr}_1 \texttt{ else } \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{<xif>}[\llbracket \mathit{Expr} \rrbracket_{\mathcal{I}}, \llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}] \\
\llbracket \mathit{Expr}_1 \ < \ \mathit{Expr}_2 \rrbracket_{\mathcal{I}} & = & \texttt{<xlt>}[\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, \llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}] \\
\llbracket \mathit{Axis}\ \mathit{NodeTest} \rrbracket_{\mathcal{I}} & = & \texttt{<xseq>}[\llbracket \mathit{Axis} \rrbracket_{\mathcal{I}}, \llbracket \mathit{NodeTest} \rrbracket_{\mathcal{I}}] \\
\llbracket \texttt{child} :: \rrbracket_{\mathcal{I}} & = & \texttt{<xchild>}[] \\
\llbracket \texttt{self} :: \rrbracket_{\mathcal{I}} & = & \texttt{<xid>}[] \\
\llbracket \mathit{NCName} \rrbracket_{\mathcal{I}} & = & \texttt{<xmap>}[\texttt{<xif>}[\texttt{<xwithtag>}[\mathit{NCName}], \\
 & & \texttt{<xid>}[], \texttt{<xconst>}[]]] \\
\llbracket * \rrbracket_{\mathcal{I}} & = & \texttt{<xmap>}[\texttt{<xif>}[\texttt{<xiselement>}[], \\
 & & \texttt{<xid>}[], \texttt{<xconst>}[]]] \\
\llbracket \texttt{text}() \rrbracket_{\mathcal{I}} & = & \texttt{<xmap>}[\texttt{<xif>}[\texttt{<xistext>}[], \texttt{<xid>}[], \texttt{<xconst>}[]]] \\
\llbracket \texttt{node}() \rrbracket_{\mathcal{I}} & = & \texttt{<xid>}[] \\
\llbracket \texttt{element } \mathit{NCName}\ \{\mathit{Expr}\} \rrbracket_{\mathcal{I}} & = & \texttt{<xseq>}[\texttt{<xconst>}[<\mathit{NCName}_{\texttt{x}}^{\texttt{non}}>[]], \\
 & & \texttt{<xchcont>}[\llbracket \mathit{Expr} \rrbracket_{\mathcal{I}}]] \\
\llbracket \texttt{element } \{\mathit{Expr}_1\}\ \{\mathit{Expr}_2\} \rrbracket_{\mathcal{I}} & = & \texttt{<xseq>}[\texttt{<xconst>}[<\mathsf{virtual}_{\texttt{x}}^{\texttt{non}}>[]], \\
 & & \texttt{<xchcont>}[\llbracket \mathit{Expr}_2 \rrbracket_{\mathcal{I}}], \texttt{<xrename>}[\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}]] \\
\llbracket \mathit{NCName}\ (\mathit{Expr}_1, ..., \mathit{Expr}_n) \rrbracket_{\mathcal{I}} & = & <\mathit{NCName}>[\llbracket \mathit{Expr}_1 \rrbracket_{\mathcal{I}}, ..., \llbracket \mathit{Expr}_n \rrbracket_{\mathcal{I}}]
\end{array}
$$

Figure 5: Translation of XQuery Core Expression

bound to the whole value returned by $[\![Expr_1]\!]_{\mathcal{I}}$.

The `if` expression is translated into `xif` with each subexpression translated into the corresponding subexpression. Similarly, the less-than comparison is translated into `xlt`.

The `child::` axis is translated into `xchild` directly. That is, we have to implement both the forward and backward transformations of `child::` in the underlying language. It is the same for some other axes, such as `descendant::` and `descendant-or-self::`. The `self::` axis is simply translated into `xid`. The node test is to filter the sequence value returned by an path axis. The node test `node()` chooses all values, so it is translated into `xid`; the translation results of *NCName*, $*$ and `text()` are the same except for the conditional in `xif`. Generally speaking, these translation results filer values that do not satisfy the specified conditionals. For example, the translation result of *NCName* filters all those elements that do not have the tag *NCName*.

The element constructor `element` *NCName* $\{Expr\}$ constructs an elements with tag *NCName* and contents computed by *Expr*. In the translation, we first build en element with tag *NCName*, and build it contents using `xchcont` with the translation result of $Expr_2$ as the argument. There is another element constructor `element` $\{Expr_1\}$ $\{Expr_2\}$, which build an element with the tag computed by $Expr_1$. So the translation result of $Expr_1$ is used by `xrename` to change the tag of the virtual element after its content is built.

The function call $NCName(Expr_1, ..., Expr_n)$ is translated into an element with the function name *NCName* as tag and the translation results of arguments $Expr_i (1 \leq i \leq n)$ as contents. At last, the function declaration of the following form:

$$\texttt{function}\ NCName(\$Var_1\ \texttt{as}\ Ty_1, ..., \$Var_n\ \texttt{as}\ Ty_n)\ \texttt{as}\ Ty \\ \{Expr\}$$

is translated into a function declaration in the target language, as follows:

$$<\texttt{function}\ \texttt{name} = \text{“}NCName\text{”} \\ \texttt{arg}_1 = \text{“}Var_1\text{”}\ ...\ \texttt{arg}_n = \text{“}Var_n\text{”}>[[\![Expr]\!]_{\mathcal{I}}]$$

Now we can prove that the above translation preserve the semantics of XQuery Core.

Theorem 2 (Correctness of Translation) Let $\mathcal{E}$ be an environment that maps variables to XML values. If an XQeury Core expression *Expr* is evaluated to a value under $\mathcal{E}$, then the expression $[\![Expr]\!]_{\mathcal{I}}$ is also evaluated to the same value under the same environment without considering annotations.

Proof Sketch: By induction on each translation rule. Some cases are given below.

- *String* is already a value, and <`xconst`>[$\mathit{String}_{\mathtt{x}}^{\mathtt{non}}$] also returns the value *String*.

- If Expr_1 and Expr_2 have the values Val_1 and Val_2, then the sequence expression $\mathit{Expr}_1, \mathit{Expr}_2$ has value Val_1, Val_2. Assume the semantics of Expr_1 and Expr_2 is preserved. That is, $[\![\mathit{Expr}_1]\!]_{\mathcal{I}}$ and $[\![\mathit{Expr}_1]\!]_{\mathcal{I}}$ have the same values Val_1 and Val_2. Then $[\![\mathit{Expr}_1, \mathit{Expr}_2]\!]_{\mathcal{I}}$ generates values <$\mathsf{virtual}_{\mathtt{x}}^{\mathtt{non}}$>[Val_1, Val_2] after using `xchcont`, and returns the same value after using `xchild`.

- \$*Var* and <`xload`>[\$*Var*] have the same value $\mathcal{E}(\$\mathit{Var})$.

- If Expr_1 has the value $\mathit{Val}_1, ..., \mathit{Val}_n$, then the `for` expression has the value $\mathit{ValSeq}_1, ..., \mathit{ValSeq}_n$, where $\mathit{ValSeq}_i (1 \leq i \leq n)$ is the value of Expr_2 under the environment $\mathcal{E}$ extended with [$\mathit{Var}_i \mapsto \mathit{Val}_i$]. Assume the semantics of Expr_1 and Expr_2 is preserved. That is, $[\![\mathit{Expr}_1]\!]_{\mathcal{I}}$ has the value $\mathit{Val}_1, ..., \mathit{Val}_n$, and $[\![\mathit{Expr}_2]\!]_{\mathcal{I}}$ has the same value as Expr_2 under the same environment. In the translation of `for` expression, `xmap` will deal with $\mathit{Val}_i (1 \leq i \leq n)$ one by one, and for each Val_i, $[\![\mathit{Expr}_2]\!]_{\mathcal{I}}$ is evaluated under the environment $\mathcal{E}$ extended with [$\mathit{Var}_i \mapsto \mathit{Val}_i$] because of `xload` and `xfree` before and after it. Therefor, the `for` expression and its translation are evaluated to the same value.

- Assume the semantics of the body of the function *NCName* and the argument expressions $\mathit{Expr}_i (1 \leq i \leq n)$ is preserved. Then, the function call $\mathit{NCName}(\mathit{Expr}_1, ..., \mathit{Expr}_n)$ and its translation have the same value because their bodies are both evaluated under the environment $\mathcal{E}$ extended with the mapping from Var_i to the value of Expr_i or $[\![\mathit{Expr}_i]\!]_{\mathcal{I}}$.

□

5.3 A Translation Example

As an example, we consider translation of the following XQuery Core expression:

```
for $section in
    let $sequence := $book-or-section return
        for $dot in $sequence
        return child::section
return
    element section {
        let $sequence := $section return
            for $dot in $sequence
            return child::title,
        let $v4 := $section return toc($v4)
     }
```

which corresponds to the body of the function `toc` in Section 1 and is adapted from the the normalized function generated by Galax [9]. The above expression is translated into the target language code in Figure 6. To help read, the translated code is divided into several pieces. X1 corresponds to the XQuery Core subexpression from the second line to the fourth line, X2 the seventh line to the ninth line, and X3 corresponds to the last line. Obviously, this code from the translation rules is verbose, but it has much room for optimization, which will be one of our future work.

6 Insertion

In this section, we discuss the view updating problems caused by insertion on view, and show our approach of using a type system to solve them.

6.1 Problems of Insertion

Insertion causes several problems for updating source data. We will use examples to explain these problems. In these examples, only `ins` annotations are given, and others are omitted. Suppose that we have the following source data:

```
<book>[<title>[Network],
       <author>[Tom]],
<book>[<title>[Algorithm],
       <author>[Peter],
       <author>[Kevin]]
```

where each book may contain a title and one or more authors. Consider the following transformation (i.e. equivalent to `child::title` in XQuery):

```
<xmap>[<xseq>[<xchild>[],
              <xmap>[<xif>[<xwithtag>[title],
                           <xid>[],
                           <xconst>[]]]]]
```

When applying this transformation to the above source document, we get a view including two `title` elements from the source document. Now inserting a new title to the view yields the following view.

<title>[Network],
<title>[Algorithm],
<title$^{\mathtt{ins}}$>[Database$^{\mathtt{ins}}$]

Some problems come when we update the source data with this changed view and the above transformation, and they are caused by the fact that the inserted `title` element does not have its corresponding value of `book`

```
<xseq>[ X1,
  <xmap>[
    <xseq>[
      <xstore>[$section],
      <xseq>[
        <xconst>[<section>[]],
        <xchcont>[<xseq>[<xconst>[<virtual>[]],
                         <xchcont>[X2,X3],
                         <xchild>[]]]],
     <xfree>[$section]]
  ]
]

X1:
  <xseq>[<xload>[$book-or-section],
    <xstore>[$sequence],
    <xseq>[<xload>[$sequence],
      <xmap>[
        <xseq>[
          <xstore>[$dot],
          <xseq>[<xchild>[],
                 <xmap>[<xif>[<xwithtag>[section],
                              <xid>[],<xconst>[]]]],
          <xfree>[$dot]]]],
    <xfree>[$sequence]
  ]

X2:
  <xseq>[<xload>[$section],
    <xstore>[$sequence],
    <xseq>[<xload>[$sequence],
    <xmap>[
      <xseq>[
        <xstore>[$dot],
        <xseq>[<xchild>[],
                <xmap>[<xif>[<xwithtag>[title],
                             <xid>[],<xconst>[]]]],
        <xfree>[$dot]]]],
    <xfree>[$sequence]
  ]

X3:
  <xseq>[
    <xload>[$section],
    <xstore>[$v4],<toc>[<xload>[$v4]],<xfree>[$v4]
  ]
```

Figure 6: Translation of the Body of Function toc

element in the source. In the backward transformation, the inserted `title` element is first transformed (in the backward direction) by `xif`. However, we do not know which branch of `xif` should be chosen, since its conditional is evaluated on the original source. And then, after `xif`, its result will be used by `xchild` to put back the removed tag. `xchild` needs to know the tag of the original source is, but it cannot get this information in this example because no source value is available.

For another example, consider a variant of the above transformation.

```
<xmap>[<xseq>[<xchild>[],
              <xmap>[<xif>[<xwithtag>[author],
                           <xid>[],
                           <xconst>[]]]]]
```

After this transformation is applied, the generated view includes three `author` elements from two books. Suppose we insert a new `author` element in the second place. Then, the changed view is the following one:

$$\begin{array}{l}<\texttt{author}>[\texttt{Tom}],\\ <\texttt{author}^{\texttt{ins}}>[\texttt{Alice}^{\texttt{ins}}],\\ <\texttt{author}>[\texttt{Peter}],\\ <\texttt{author}>[\texttt{Kevin}]\end{array}$$

In this example, it is reasonable to put the inserted author back into the first book as a new author or into the second book, or even construct a new book element containing only this author. That is, the way of updating the source document is not unique. Choosing which way is determined by the operator `split` in `xmap`, which divides a sequence value into subsequences to transform. In Section 4, the operator `split` divides a sequence value according to a list of integers for the length of each subsequence. This does not work when we consider insertion, and it needs extension. A naive method is to assign the inserted values always into a subsequence with the values before it (or after it), or a independent subsequence. But this is not a good approach as shown by the following transformation:

```
<xmap>[<xchild>[]]
```

The view from this transformation is a sequence including the `title` and `author` elements of each book. Suppose we the following changed view, where a new author, a new title, and another new author are inserted in the third place.

$$\begin{array}{l}<\texttt{title}>[\texttt{Network}], <\texttt{author}>[\texttt{Tom}],\\ <\texttt{author}^{\texttt{ins}}>[\texttt{Alice}^{\texttt{ins}}], <\texttt{title}^{\texttt{ins}}>[\texttt{Database}^{\texttt{ins}}],\\ <\texttt{author}^{\texttt{ins}}>[\texttt{Paul}^{\texttt{ins}}], <\texttt{title}>[\texttt{Algorithm}],\\ <\texttt{author}>[\texttt{Peter}], <\texttt{author}>[\texttt{Kevin}]\end{array}$$

$$\begin{array}{rcl} \mathit{ValTy} & ::= & t \mid () \mid \texttt{string} \mid \mathit{str} \mid \ <\!\mathit{tag}\!>[\mathit{Ty}] \mid \mathit{Ty}* \\ & & \mid \mathit{Ty}, \mathit{Ty} \mid \mathit{Ty}|\mathit{Ty} \mid \ \mid \texttt{Rec}\ t.\mathit{Ty} \end{array}$$

Figure 7: Syntax of Types

In this example, according to the above naive approach, the three inserted values will be assigned into a subsequence with the first two values in view by operator `split`. After backward transformation, the first `book` element will include one title, two authors, another title and another author. This is not a reasonable update for this book. Actually, if the operator `split` assigns the inserted `title` element as well as its following inserted `author` element into a independent subsequence, and then after backward transformation, the updated source data will include three `book` elements (with assumption that `xchild` knowing the removed tag for an inserted element) and the second is a newly inserted one including the inserted title and the second inserted author. That is, we need a more clever `split` operator.

In summary, in backward transformation, the following constructs in the target language encounter problems: the transformation `xchild` lacks information of the removed tag for building a source value; the transformation combinator `xif` does not know which branch transformation should be chosen without the original source data; the transformation combinator `xmap` and `xchcont` do not know how to split a sequence value into subsequences such that the updated source data is more reasonable.

6.2 Our Approach with Types

In this section, we explain how to use type-annotated transformations to solve problems caused by insertion on view. The general idea is that some problematic transformations will be annotated with types, and these types can provide information of how to perform the backward transformation when the original source value is missing. For example, if the source element type of `xchild` is known, then the tag of new source data can be gotten from this type when an inserted value is transformed by `xchild` in backward direction.

The types used by our work are given in Figure 7, which are almost same as the regular expression types in [11] except that a string *str* is also regarded as a type. Hence, an annotated value can be regard as a type if all annotations are removed. The recursive type $\texttt{Rec}\ t.\mathit{Ty}$ is regarded as equivalent to its unfolded form $\mathit{Ty}[\mathrm{t}/\texttt{Rec}\ t.\ \mathit{Ty}]$. The types `string`, *str* and the element type $<\!\mathit{tag}\!>[\mathit{Ty}]$ are called *basic types*.

The semantics of the types is defined in Figure 8. $\mathit{ValSeq} \in \mathit{Ty}$ means the value *ValSeq* has type *Ty*. This semantics is weaker than the usual semantics of regular expression types in [11] because of the last rule, since

$$\frac{}{()\underline{\in}()}$$

$$\frac{}{str^{O}\underline{\in}\texttt{string}}$$

$$\frac{}{str^{O}\underline{\in}str}$$

$$\frac{ValSeq \in Ty \qquad tag = tag' \text{ or } tag = \texttt{any}}{<tag'^{O}>[ValSeq]\underline{\in}<tag>[Ty]}$$

$$\frac{ValSeq_i\underline{\in}Ty \ (1 \leq i \leq n)}{ValSeq_1, ..., ValSeq_n\underline{\in}Ty*}$$

$$\frac{ValSeq_1\underline{\in}Ty_1 \qquad ValSeq_2\underline{\in}Ty_2}{ValSeq_1, ValSeq_2\underline{\in}Ty_1, Ty_2}$$

$$\frac{ValSeq\underline{\in}Ty_i \ (i = 1 \text{ or } i = 2)}{ValSeq\underline{\in}Ty_1|Ty_2}$$

$$\frac{ValSeq\underline{\in}Ty[\texttt{Rec}\ t.Ty/t]}{ValSeq\underline{\in}\texttt{Rec}\ t.Ty}$$

$$\frac{\begin{array}{l} ValSeq = Val_1, ..., Val_n \qquad \texttt{check}_{\texttt{ins}}(ValSeq) \text{ holds} \\ Ty = Ty_1\bullet_1 ... \bullet_{m-1} Ty_m \bullet_m \\ \quad \text{where } \bullet \in \{*, ,, *,\} \text{ and } Ty_i \text{ is a basic type} (1 \leq i \leq m). \\ \forall j : 1 \leq j \leq n. \exists i : 1 \leq i \leq m. Val_j\underline{\in}Ty_i \end{array}}{ValSeq\underline{\in}Ty}$$

Figure 8: Semantics of Types

$\texttt{split}((), [0], ()) = ()$
$\texttt{split}(str, [1], \texttt{string}) = str$
$\texttt{split}(str, [1], str) = str$
$\texttt{split}(<tag>[ValSeq], [1], <tag>[Ty]) = <tag>[ValSeq]$
$\texttt{split}(ValSeq, [l_1, ..., l_n], Ty_1, Ty_2) = \texttt{split}(ValSeq_1, [l_1, ..., l_k], Ty_1),$
$\qquad\qquad \texttt{split}(ValSeq_1, [l_{k+1}, ..., l_n], Ty_2)$
$\quad \texttt{where } ValSeq_1 \in Ty_1; ValSeq_2 \in Ty_2 \ \texttt{len}(ValSeq_1) = l_1 + ... + l_k;$
$\qquad \texttt{len}(ValSeq_2) = l_{k+1} + ... + l_n \ (1 \leq k \leq n)$
$\texttt{split}(ValSeq, [l_1, ..., l_n], Ty_1|Ty_2) = \texttt{split}(ValSeq, [l_1, ..., l_n], Ty_i)$
$\quad \texttt{where } ValSeq \in Ty_i (i = 1 \text{ or } i = 2)$
$\texttt{split}(ValSeq, [l_1, ..., l_n], Ty*) = ValSeq_1, ..., ValSeq_m (n \leq m)$
$\quad \texttt{where}$
$\qquad \forall\ i : 1 \leq i \leq m. ValSeq_i \in Ty$
$\qquad$ and if $ValSeq_i$ is the kth subsequence including not only inserted values,
$\qquad\quad$ then $\texttt{len}(ValSeq_i) = l_k$

Figure 9: The Operator `split`

this rule allows a type to include probably some inserted values that are not in the type according to the usual semantics. For example, a type requires a book to have a title and a price element, our weaker semantics allows a newly inserted book to contain only a title. In the case where the usual semantics is expected, the notation $\mathit{ValSeq} \in \mathit{Ty}$ is used and the last rule is not included.

The problematic transformations are annotated with types in the following way: $<$ `xchild` $>$ $[]^{\lhd tag_1 \rhd [Ty_1] | \ldots | \lhd tag_n \rhd [Ty_n]}$ is annotated by its source type, required to be a choice type consisting only element types; $<$`xif`$>$ $[P, X_1, X_2]^{Ty_2}_{Ty_1}$ is annotated both the view type Ty_1 of the true brach X_1 and the view type Ty_2 of the false branch X_2; both <`xchcont`>$[X_1, ..., X_n]^{Ty}$ and <`xmap`>$[X']^{Ty}$ are annotated by their view types.

It is boring to annotate transformations with types, and moreover, the types annotated by people is not necessarily correct. In our work, we have designed a sound type system. Given an transformation and the type of the source data, this type system can infer the view type and generate a type-annotated transformation. Due to space limitation, this type system is given in Appendix B.

With annotated types, the backward transformations of `xchild`, `xif`, `xchcont`, `xmap` need to be revised to deal with insertion. In the following definition, the inserted values or the changed views are required to have the corresponding view types.

`xchild`: Let $X = $ <`xchild`>$[]^{\lhd tag_1 \rhd [Ty_1] | \ldots | \lhd tag_n \rhd [Ty_n]}$.

$$[\![X]\!]_B((), V') = <tag_i{}^{\mathtt{ins}}_{I}>[V'] \quad \text{if } V' \Subset Ty_i (1 \le i \le n) \text{ and } I \text{ is a fresh } id$$

The unit value () is used for the missing source. If the inserted value V' has the type $Ty_i (1 \le i \le n)$, then the tag in the ith source type will be chosen for the new source. Note that the new source is also annotated as an inserted value. And we ask I is fresh just for checking the view updating semantics of `xchild`, and it can be removed without affecting transformation behavior. If V' can belong to several types Ty_i, then any one can be chosen.

`xif`: Let $X = $ <`xif`>$[P, X_1, X_2]^{Ty_2}_{Ty_1}$.

$$[\![X]\!]_B((), V') = \begin{cases} [\![X_1]\!]_B((), V'), & \text{if } V' \in Ty_1 \\ [\![X_2]\!]_B((), V'), & \text{if } V' \in Ty_2 \\ [\![X_1]\!]_B((), V'), & \text{if } V' \Subset Ty_1 \\ [\![X_2]\!]_B((), V'), & \text{if } V' \Subset Ty_2 \end{cases}$$

`xif` chooses the most precise view type for V', which is then transformed by the corresponding branch. `xif` determines the type of V'

by first using the usual semantics, and if both view types can not be matched, then the weaker semantics of regular expression types is applied.

xchcont: Let $X = $ <xchcont>$[X_1, ..., X_n]^{Ty}$, and $S = $ <tag_I^O>$[\mathit{ValSeq}]$.

$$
\begin{array}{l}
[\![X]\!]_B(S, \text{<}tag'^{O'}_I\text{>}[NC']) = \text{<}tag'^{O'}_I\text{>}[\mathit{ValSeq}] \\
\quad \texttt{where} \\
\quad \begin{array}{lcl}
V_i & = & [\![X_i]\!]_F(())(1 \le i \le n) \\
V'_1, \ldots, V'_n & = & \texttt{split}(NC', [|V_1|, ..., |V_n|], Ty) \\
S'_i & = & [\![X_i]\!]_B((), V'_i)(1 \le i \le n)
\end{array}
\end{array}
$$

For `xchcont`, the problem is how to split the changed contents NC' when it includes inserted values. Here, we design a new operator `split` to do this work. Compared with the old `split`, this new one divides NC' into subsequences using the type information about NC'. Note that the number of subsequences $V'_i(1 \le i \le n)$ is still n, the length of the second argument of the `split` operator, even if there are inserted values. This means that the inserted values must belong to the view type of one transformation X_i if the changed view is well typed.

xmap: Let $X = $ <xmap>$[X']^{Ty}$, and $S = \mathit{Val}_1, ..., \mathit{Val}_n$.

$$
\begin{array}{l}
[\![X]\!]_B(S, V') = Val'_1, ..., Val'_n \\
\quad \texttt{where} \\
\quad \begin{array}{lcl}
V & = & V_1 \ldots V_n \\
V_i & = & [\![X']\!]_F(Val_i)(1 \le i \le n) \\
V'_1, ..., V'_m & = & \texttt{split}(V', [|V_1|, ..., |V_n|], Ty)(n \le m) \\
\mathit{Val}'_i & = & [\![X']\!]_B(\mathit{Val}_i, V'_i)(1 \le i \le m)
\end{array}
\end{array}
$$

This revision of `xmap` also uses the new `split` operator. Unlike `xchcont`, the number of subsequences can be greater than n, the length of the second argument of the `split` operator. That is, some new inserted values will create independent subsequences to be transformed by X' if the changed view is well typed. Note that if Val_i includes only inserted values, then a unit value for missing source data is used in $[\![X']\!]_B((), V'_i)$, and the Val_i will be paired with V'_{i+1} to do backward transformation if Val'_{i+1} contains not only inserted values, otherwise Val_i will be paired with the remaining subsequences of V', and so on.

The new `split` operator is defined in Figure 9. It takes three arguments: the first is the sequence to be divided; the second is a list of integers, each of which indicates the expecting length of a subsequence; the third one is the type of the first argument. In the definition, `len` is to compute the length of a sequence without considering inserted values. As an example for illustration, the last rule says that a sequence ValSeq with type $Ty*$ is split into a list of subsequences $\mathit{ValSeq}_i (1 \le i \le m)$, and each subsequence ValSeq_i is required to have type Ty, and if it is the kth subsequence that contains not only inserted values, then its length without counting inserted values should be l_k.

Following the above idea of annotating those problematic transformations with types, if they appear in a function declaration, we should also annotate the function body, and if this function is applied with different types, then we have to annotate it with each type, and hence, get several copies of the same function with different type annotations. Obviously, annotating function body is not a good idea. Our approach is to annotate the function call $<\mathit{funname}>[X_1, ..., X_n]_{\mathit{Ty}_1,...,\mathit{Ty}_n}$ with its argument types. And then, we can type check the function body at runtime using the type system in Appendix B, and thus, a type-annotated version of function body can be obtained dynamically with context sensitivity. Suppose X' is the body of the function *funname*. The revised function call is defined as follows:

function call: Let $X = <\mathit{funname}>[X_1, ..., X_n]_{\mathit{Ty}_1,...,\mathit{Ty}_n}$.

$$\begin{array}{rcll} X & = & <\texttt{seq}>[& X_1, <\texttt{xstore}>[\mathit{Var}_1], \\ & & & ..., \\ & & & X_n, <\texttt{xstore}>[\mathit{Var}_n], \\ & & & <\texttt{xconst}>[], \\ & & & X'', \\ & & & <\texttt{xfree}>[\mathit{Var}_n], \\ & & & ..., \\ & & & <\texttt{xfree}>[\mathit{Var}_1]\ ] \end{array}$$

where X'' is obtained by the following judgement:

$$\left[\begin{array}{l} \mathit{Var}_1 \mapsto \mathit{Ty}_1, ..., \mathit{Var}_n \mapsto \mathit{Ty}_n, \\ \mathit{funname}(\mathit{Ty}_1, ..., \mathit{Ty}_n) \mapsto t \end{array}\right]; () \vdash X' : \mathit{Ty}' \Rightarrow \Gamma, X''$$

6.3 Revisit the Insertion Problems

For the source data used in Section 6.1, it is supposed to have the following type.

```
<book>[<title>[string],<author>[string]*]*
```

In the following, we will use `Book` for the above book element type, `Title` and `Author` for the title element type and the author element type respecitvely, and `Ty` for the sequence type `Title`, `Author`$*$. The changed views in Section 6.1 are still used here.

For the first transformation, which returns a sequence of title elements, it can be annotated with the following type annotations.

$$\begin{array}{l} <\texttt{xmap}>[<\texttt{xseq}>[<\texttt{xchild}>[]^{\texttt{Book}}, \\ \quad <\texttt{xmap}>[<\texttt{xif}>\ [<\texttt{xwithtag}>[\texttt{title}], \\ \quad\ <\texttt{xid}>[], <\texttt{xconst}>[]]^{()}_{\texttt{Title}}]^{\texttt{Title}|()}]]^{\texttt{Title}*} \end{array}$$

In this example, when the changed view is transformed, the inserted title element is first transformed by `xif`, and it is sent to the `xid` branch, since it is in the view type of `xid`. And then, it is transformed by `xchild`, which now knows the `book` tag should be put on the inserted title according to the annotated source type `Book`.

In the third example, the transformation is annotated as the following.

$$\texttt{<xmap>}[\texttt{<xseq>}[\texttt{<xchild>}[]^{\texttt{Book}}]]^{\texttt{Ty}*}$$

In this example, we are interested in how to split the changed view into subsequences, such that each subsequence will be transformed by `xchild`. This work is done by using the `split` operator with three arguments: the changed view, the integer list [2,3] for the length of two originally existing subsequences (i.e., the contents of the first book and the second book, respectively), and the annotated type `Ty`∗ of `xmap`. According to the last case in Figure 9, the changed view is divided into three subsequences: the first includes the first three elements, the second includes the fourth and the fifth inserted elements, and the third includes the remaining elements. All these subsequences have type `Ty`, and moreover, the first and the third subsequence have length of 2 and 3 (without considering the inserted values), respectively.

7 Related Work

Our bidirectional transformation language learns a lot from the existing ones. In [7], a semantic foundation and a domain-specific programming language FOCAL for bidirectional transformations are given. They form the core of the data synchronisation system Harmony [12]. Another related language was given by Meertens [13] to specify constraints in the design of user-interfaces. In the work of implementing the programmable XML editor [8], a domain-specific XML processing language, called X, is developed, which is a point-free functional language closely related to the languages in [13] and [7].

Our work is motived by many useful applications of view updating. View-updating, which is to correctly reflect the modifications on view back to the database [2, 3, 4, 5], is an old problem in the database community. In recent years, however, the need to synchronize data related by some transformations starts to be recognized by researchers from different fields. In tools for aspect-oriented programming it is helpful to have multiple views of the same program [14]. In editors such as [15] the user edits a view computed from the source by a transformation. Recent research on code clone [16] argues that a certain proportion of code in a software resembles each other, and it may help to develop software maintenance tools that keep the resembling pieces of code updated when one of them is altered. It is argued in [17] that such *coupled transformation* problems are widespread and diverse.

8 Conclusion

In this paper, we have done the work of bidirectionalize XQuery, that is, making XQuery to support view updating just like its counterpart SQL in relational database. The result is that changes on XQuery view can be reflected back into the source XML data just by executing the XQuery expressions in backward direction. This feature makes XQuery more useful in the scenario of exchanging data on web.

During this work, we first give a more flexible view updating semantics, and based on this semantics, we design the underlying language that is expressive enough to interpret XQuery. And then, we give the translation rules from XQuery Core to this underlying language, and we have proved that this translation preserves the semantics of XQuery. Insertion on view causes much trouble because it lacks enough information about what the source data should like after updated, so the backward transformation of inserted values probably produces unreasonable results. We provide a type system to the underlying language, and after type checking, this expression will be annotated with types. This type information can help put the inserted values back into the source data in a more sensible way. Our prototype implementation confirms our idea in this paper to solve the view updating problem of XQuery.

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] Serge Abiteboul. On views and XML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1999.

[6] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases, 2005. http://www.w3.org/TR/xquery-use-cases/.

[7] 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.

[8] 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*, pages 178–189. ACM Press, 2004.

[9] The Galax Team. Galax: An Implementation of Query. http://www.galaxquery.org/.

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

[12] Benjamin C. Pierce, Alan Schmitt, and Michael B. Greenwald. Bringing harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data. Technical Report, MS-CIS-03-42, University of Pennsylvania, March 18, 2004.

[13] Lambert Meertens. Designing constraint maintainers for user interaction. `ftp://ftp.kestrel.edu/ pub/papers/meertens/dcm.ps`, 1998.

[14] Doug Janzen and Kris de Volder. Programming with crosscutting effective views. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, number 3086 in Lecture Notes in Computer Science, pages 195–218. Springer-Verlag, June 14-18, 2004.

[15] Martijn M. Schrage. *Proxima - A presentation-oriented editor for structured documents.* PhD thesis, Utrecht University, The Netherlands, 2004.

[16] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. ARIES: refactoring support environment based on code clone analysis. In *The 8th IASTED International Conference on Software Engineering*

and Applications(SEA 2004), pages 222–229, Cambrdige, USA, November 9-11, 2004. ACTA Press.

[17] Ralf Lämmel. Coupled software transformations (extended abstract). In *First International Workshop on Software Evolution Transformations*, 2004.

A The Source Data File book.xml

```
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <section>
    <title>Introsuction</title>
    <p>Text ... </p>
    <section>
      <title>Audience</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Web Data and the Two Cultures</title>
      <p>Text ... </p>
      <figure>
        <title>
          Traditional client/server architecture
        </title>
        <image/>
      </figure>
      <p>Text ... </p>
    </section>
  </section>
  <section>
    <title>A Syntax For Data</title>
    <p>Text ... </p>
    <figure>
      <title>
        Graph representations of structures
      </title>
      <image/>
    </figure>
    <p>Text ... </p>
    <section>
```

```
         <title>Base Tupes</title>
         <p>Text ... </p>
      </section>
      <section>
        <title>
          Representing Relational Databases
        </title>
        <p>Text ... </p>
        <figure>
          <title>Examples of Relations</title>
          <image/>
        </figure>
      </section>
      <section>
        <title>
          Representing Object Databases
        </title>
        <p>Text ... </p>
      </section>
    </section>
  </book>
```

B The Type System

The typing rules for the bidirectional transformation language is defined in Figure 10. A judgment has the form $\Gamma; \mathit{Ty} \vdash X : \mathit{Ty}' \Rightarrow \Gamma', X'$, which means that if the source type is *Ty*, the transformation X will generate a view having type *Ty'* under the typing context Γ, and after type checking, a new typing context Γ' and a new transformation X' are generated. The new transformation X' is the result of annotating X with types. This type system annotates only five transformations `xchild`, `xchcont`, `xmap`, `xif` and function calls. The typing context Γ maps a variable to a type or a function name with the types of its arguments to its view type.

In the typing rule for `xseq`, the last typing context $\Gamma_n$is required to be equal to Γ, which means that the variables bound in a `xseq` can only be accessed by transformations inside this `xseq`. To meet this requirement, if there is a `xstore`(*Var*) to bind *Var* in a `xseq`, then there must be a following `xfree`(*Var*) to release *Var*. This requirement is also required on the transformation arguments of other transformation combinators, such as `xchcont` and `xmap`, and function calls.

The view type of the true branch in `xif` can be inferred more accurately if the predicate is `xwithtag`, which help restrict the type of the source data, that is, this type system is path sensitive.

There are two typing rules for function calls. If a function and the types of its arguments is not mapped by 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 function arguments $Var_i (1 \leq i \leq n)$ is mapped to the types $Ty_i (1 \leq i \leq n)$, and the function name *funname* together with these argument types is mapped to a fresh type variable t. The type Ty' of function body X probably contains the free type variable t for the function calls with same argument types (due to using the second typing rule for function calls). So the view type of the first typing rule is a recursive type $\texttt{Rec}\, t.Ty'$. In the second rule, the function body will not be checked since its resulting type is already available. In this case, the function is a recurse function. Note that the type-annotated function body is abandoned, and the function declaration does not be changed in type checking. This does not mean that we do not need the type annotation in function body, but because we want to avoid the trouble of managing different versions of the same function with different types. Our approach is to recompute the type-annotated function body when needed at runtime. On the other hand, our type checking for a function call is context sensitive since its body is checked with its argument types determined at the program point of the function call. This kind of type checking can help annotate the function body accurately. But the accuracy is not gotten for free, and the cost is that it will fail to type check some recursive functions if the structure of the type variable t is needed by other transformation in the function body.

This type system is sound with respect to the transformation semantics of the target language.

Theorem 3 (Soundness) For a transformation X and a source value S, if $\phi; Ty \vdash X : Ty' \Rightarrow \phi, X'$, and $S \in Ty$, then $[\![X']\!]_F(S) = V$, and $V \in Ty'$; and moreover, if $V' \in Ty'$ and $V \equiv V'$, then $[\![X']\!]_B(S, V') = S'$ and $S' \underline{\in} Ty$.

Proof. By induction on the typing rules. Two sample cases are given below.

- Let $X = <\texttt{xchild}>[]$ and $Ty = <tag_1>[Ty_1]|...|<tag_n>[Ty_n]$. Hence, $X' = <\texttt{xchild}>[]^{Ty'}_{Ty}$ and $Ty' = Ty_1|...|Ty_n$ according to the typing rule for `xchild`. If S is an element $<tag_i>[ValSeq]$ with the type $<tag_i>[Ty_i] (1 \leq i \leq n)$, then $[\![X']\!]_F(S)$ returns $ValSeq$, which has type Ty_i, a substype of Ty'; $[\![X']\!]_B(S, ValSeq')$ returns $<tag_i>[ValSeq']$. On the other hand, if S is missing, and $ValSeq' \in Ty_i (1 \leq i \leq n)$, $[\![X']\!]_B((), ValSeq')$ returns $<tag_i>[ValSeq']$.

- Let $X = <\texttt{xmap}>[X'']$ and $Ty = Ty_1*$ (Other cases of Ty are similar). Hence, $X' = <\texttt{xmap}>[]^{Ty'_1*}$ and $\phi; Ty \vdash X'' : Ty'_1, X'''$ according to the typing rule for `xmap`. If $S = Val_1, ..., Val_n$, and $S \in Ty_1*$, that is each $Val_i (1 \leq i \leq n)$ has type Ty_1, then $[\![X']\!]_F(S)$

$= [\![X''']\!]_F(\mathit{Val}_1), ..., [\![X''']\!]_F(\mathit{Val}_n)$, which has type Ty'_1* according to the inductive hypothesis that X'' satisfies the above property; similarly, if V is split into $\mathit{ValSeq}'_1, ..., \mathit{ValSeq}'_m$ $(n \leq m)$, and $V \in \mathit{Ty}'_1*$, that is each $\mathit{Val}'_i(1 \leq i \leq m)$ has type Ty_1, then $[\![X']\!]_B(S, V) = [\![X''']\!]_B(\mathit{Val}_1, \mathit{Val}'_1), ..., [\![X''']\!]_B(\mathit{Val}_1, \mathit{Val}_m)$, which returns an updated source value since each X''' returns one according to the inductive hypothesis that X'' satisfies the above property.

$\square$

$$\frac{}{\Gamma; Ty \vdash \texttt{<xid>}[] : Ty \Rightarrow \Gamma, \texttt{<xid>}[]}$$

$$\frac{Ty = <tag_1>[Ty_1]|...|\ <tag_n>[Ty_n]}{\Gamma; Ty \vdash \texttt{<xchild>}[] : Ty_1|...|Ty_{\mathrm{n}} \Rightarrow \Gamma, \texttt{<xchild>}[]^{Ty}}$$

$$\frac{}{\Gamma; Ty \vdash \texttt{<xconst>}[Val] : \texttt{rmanno}(Val) \Rightarrow \Gamma, \texttt{<xconst>}[Val]}$$

$$\frac{\Gamma; Ty \vdash X_1 : Ty_1 \Rightarrow \Gamma_1, X'_1 \quad ... \quad \Gamma_{n-1}; Ty_{n-1} \vdash X_n : Ty_n \Rightarrow \Gamma_n, X'_n \quad \Gamma_n = \Gamma}{\Gamma; Ty \vdash \texttt{<xseq>}[X_1, ..., X_n] : Ty_n \Rightarrow \Gamma, \texttt{<xseq>}[X'_1, ..., X'_n]}$$

$$\frac{\begin{array}{l} Ty = <tag_1>[Ty_1]|...|\ <tag_n>[Ty_n] \quad \Gamma; () \vdash X_1 : Ty'_1 \Rightarrow \Gamma_1, X'_1 \quad \Gamma_1 = \Gamma \ ... \\ \Gamma; () \vdash X_n : Ty'_n \Rightarrow \Gamma_n, X'_n \quad \Gamma_n = \Gamma \\ Ty' = <tag_1>[Ty'_1, ..., Ty'_n]|...|\ <tag_n>[Ty'_1, ..., Ty'_n] \end{array}}{\Gamma; Ty \vdash \texttt{<xchcont>}[X_1, ..., X_n] : Ty' \Rightarrow \Gamma, \texttt{<xchcont>}[X'_1, ..., X'_n]^{Ty'}}$$

$$\frac{\begin{array}{l} Ty = Ty_1 \bullet_1 ... \bullet_{n-1} Ty_n, \text{where } \bullet \in \{*, |, ,, *|, *,\} \text{ and } Ty_i \text{ is a basic type}(1 \leq \mathrm{i} \leq \mathrm{n}). \\ \Gamma; Ty_1 \vdash X : Ty'_1 \Rightarrow \Gamma_1, X_1 \quad \Gamma_1 = \Gamma \quad ... \quad \Gamma; Ty_n \vdash X : Ty'_n \Rightarrow \Gamma_n, X_n \quad \Gamma_n = \Gamma \\ \Gamma; Ty_1|...|Ty_n \vdash X : Ty' \Rightarrow \Gamma', X' \quad \Gamma' = \Gamma \end{array}}{\Gamma; Ty \vdash \texttt{<xmap>}[X] : Ty'_1 \bullet_1 ... \bullet_n Ty'_n \Rightarrow \Gamma, \texttt{<xmap>}[X']^{Ty'_1 \bullet_1 ... \bullet_n Ty'_n}}$$

$$\frac{\begin{array}{l} P = \texttt{<xwithtag>}[str^{\mathrm{non}}] \quad \Gamma; Ty \vdash P : Ty'_P|() \Rightarrow \Gamma_P, P' \quad \Gamma_P = \Gamma \\ \Gamma; Ty'_P \vdash X_1 : Ty_1 \Rightarrow \Gamma_1, X'_1 \quad \Gamma_1 = \Gamma \quad \Gamma; Ty \vdash X_2 : Ty_2 \Rightarrow \Gamma_2, X'_2 \quad \Gamma_2 = \Gamma \end{array}}{\Gamma; Ty \vdash \texttt{<xif>}[P, X_1, X_2] : Ty_1|Ty_2 \Rightarrow \Gamma, \texttt{<xif>}[P'\ X'_1\ X'_2]^{Ty_2}_{Ty_1}}$$

$$\frac{\begin{array}{l} \Gamma; Ty \vdash P : Ty_P \Rightarrow \Gamma_P, P' \quad \Gamma_P = \Gamma \quad \Gamma; Ty \vdash X_1 : Ty_1 \Rightarrow \Gamma_1, X'_1 \quad \Gamma_1 = \Gamma \\ \Gamma; Ty \vdash X_2 : Ty_2 \Rightarrow \Gamma_2, X'_2 \quad \Gamma_2 = \Gamma \end{array}}{\Gamma; Ty \vdash \texttt{<xif>}[P, X_1, X_2] : Ty_1|Ty_2 \Rightarrow \Gamma, \texttt{<xif>}[P'\ X'_1\ X'_2]^{Ty_2}_{Ty_1}}$$

$$\frac{Ty = <tag_1>[Ty_1]|...|\ <tag_n>[Ty_n] \quad \Gamma; Ty \vdash X : \texttt{string} \Rightarrow \Gamma_1, X' \quad \Gamma_1 = \Gamma}{\Gamma; Ty \vdash \texttt{<xrename>}[X] : \texttt{<any>}[Ty_1]|...|\ \texttt{<any>}[Ty_{\mathrm{n}}] \Rightarrow \Gamma, \texttt{<xrename>}[X']}$$

$$\frac{}{\Gamma; Ty \vdash \texttt{<xstore>}[Var] : Ty \Rightarrow \Gamma[Var \mapsto Ty], \texttt{<xstore>}[Var]}$$

$$\frac{}{\Gamma; Ty \vdash \texttt{<xload>}[Var] : \Gamma(Var) \Rightarrow \Gamma, \texttt{<xload>}[Var]}$$

$$\frac{}{\Gamma; Ty \vdash \texttt{<xfree>}[Var] : Ty \Rightarrow \texttt{pop}(\Gamma, Var), \texttt{<xfree>}[Var]}$$

Figure 10: Typing Rule

$$\frac{\begin{array}{l}\texttt{<function name} = \text{"}\mathit{funname}\text{"}\ \texttt{arg}_1 = \text{"}Var_1\text{"}\ \ldots\ \texttt{arg}_n = \text{"}Var_n\text{"}\texttt{>}[X]\ \text{is defined.}\\ \Gamma; \mathit{Ty} \vdash X_1 : \mathit{Ty}_1 \Rightarrow \Gamma_1, X_1' \quad \Gamma_1 = \Gamma \quad \ldots \quad \Gamma; \mathit{Ty}_{n-1} \vdash X_n : \mathit{Ty}_n \Rightarrow \Gamma_n, X_n' \quad \Gamma_n = \Gamma\\ \mathit{funname}(\mathit{Ty}_1, \ldots, \mathit{Ty}_n) \notin \texttt{dom}(\Gamma)\\ [\mathit{Var}_1 \mapsto \mathit{Ty}_1, \ldots, \mathit{Var}_n \mapsto \mathit{Ty}_n, \mathit{funname}(\mathit{Ty}_1, \ldots, \mathit{Ty}_n) \mapsto t]; () \vdash X : \mathit{Ty}' \Rightarrow \Gamma', X'\\ \Gamma' = \Gamma \quad t\ \text{is fresh.}\end{array}}{\Gamma; \mathit{Ty} \vdash \texttt{<}\mathit{funname}\texttt{>}[X_1, \ldots, X_n] : \texttt{Rec}\ t.\mathit{Ty}' \Rightarrow \Gamma, \texttt{<}\mathit{funname}\texttt{>}[X_1, \ldots, X_n]_{\mathit{Ty}_1, \ldots, \mathit{Ty}_n}}$$

$$\frac{\begin{array}{l}\texttt{<function name} = \text{"}\mathit{funname}\text{"}\ \texttt{arg}_1 = \text{"}Var_1\text{"}\ \ldots\ \texttt{arg}_n = \text{"}Var_n\text{"}\texttt{>}[X]\ \text{is defined.}\\ \Gamma; \mathit{Ty} \vdash X_1 : \mathit{Ty}_1 \Rightarrow \Gamma_1, X_1' \quad \Gamma_1 = \Gamma \quad \ldots \quad \Gamma; \mathit{Ty}_{n-1} \vdash X_n : \mathit{Ty}_n \Rightarrow \Gamma_n, X_n'\\ \Gamma_n = \Gamma \quad \Gamma(\mathit{funname}(\mathit{Ty}_1, \ldots, \mathit{Ty}_n)) = \mathit{Ty}'\end{array}}{\Gamma; \mathit{Ty} \vdash \texttt{<}\mathit{funname}\texttt{>}[X_1, \ldots, X_n] : \mathit{Ty}' \Rightarrow \Gamma, \texttt{<}\mathit{funname}\texttt{>}[X_1, \ldots, X_n]_{\mathit{Ty}_1, \ldots, \mathit{Ty}_n}}$$

$$\frac{\begin{array}{l}\mathit{Ty} = \texttt{<}\mathit{tag}_1\texttt{>}[\mathit{Ty}_1] | \ldots | \texttt{<}\mathit{tag}_n\texttt{>}[\mathit{Ty}_n]\\ \mathit{Ty}'\ \text{is a choice type consisting of}\ \texttt{<}\mathit{tag}_i\texttt{>}[\mathit{Ty}_i] (1 \leq i \leq n), \text{if}\ \mathit{tag}_i = \mathit{str}\end{array}}{\Gamma; \mathit{Ty} \vdash \texttt{<xwithtag>}[\mathit{str}^{\texttt{non}}] : \mathit{Ty}' | () \Rightarrow \Gamma, \texttt{<xwithtag>}[\mathit{str}^{\texttt{non}}]}$$

$$\frac{}{\Gamma; \mathit{Ty} \vdash \texttt{<xiselement>}[] : \mathit{Ty} | () \Rightarrow \Gamma, \texttt{<xiselement>}[]}$$

$$\frac{}{\Gamma; \mathit{Ty} \vdash \texttt{<xistext>}[] : \mathit{Ty} | () \Rightarrow \Gamma, \texttt{<xistext>}[]}$$

$$\frac{\Gamma; \mathit{Ty} \vdash X_1 : \mathit{Ty}_1 \Rightarrow \Gamma_1, X_1' \quad \Gamma_1 = \Gamma \quad \Gamma; \mathit{Ty} \vdash X_2 : \mathit{Ty}_2 \Rightarrow \Gamma_2, X_2' \quad \Gamma_2 = \Gamma}{\Gamma; \mathit{Ty} \vdash \texttt{<xlt>}[X_1, X_2] : \texttt{string} | () \Rightarrow \Gamma, \texttt{<xlt>}[X_1', X_2']}$$

Figure 11: Typing Rule (Continue)