A Schema-Based Translation of XQuery Updates

Leonidas Fegaras

University of Texas at Arlington, CSE
416 Yates Street, P.O. Box 19015
Arlington, TX 76019

fegaras@cse.uta.edu

Abstract

We address the problem of translating XQuery updates to plain XQuery expressions. The resulting XQuery code reconstructs the mutable state (the updated XML data), reflecting the updated values in the new state. The translation is done using source-to-source, compositional transformations only. Unlike related approaches that use framework-specific algebras to achieve a similar goal, our work can be incorporated into any existing XQuery engine. We address data aliasing during updating by detecting expressions that return replicated XML nodes. These expressions are factored out from the state, while updates to replicas are directed to the original data sources.

1. Introduction

A major part of web programming nowadays involves querying and updating XML data, since most data available from web services and other data providers come in the form of XML. To handle these data, there was an increasing need by web developers for a well-supported XML query language, powerful enough to let them do most of the XML processing in a single language, without having to embed it into scripts or to use low-level APIs, such DOM or SAX. XQuery [17], a functional query language for XML data, was introduced by W3C to cover this need. XQuery has recently gained widespread adoption in the web programming and database communities mostly because it is supported by most database and web development software vendors. Its resemblance to SQL, a query language already familiar to all web programmers, and its conformance to W3C standards have also contributed to make XQuery a recent favorite for web programming. Web programmers though, comfortable with the imperative style of scripting languages, such as JavaScript for client-side and PHP for server-side programming, found it hard to do some programming tasks using a purely functional programming language. More specifically, there was a need for imperative language extensions to XQuery that would allow one to transform XML data before they are fed into some other web service or application. Furthermore, there is a recent interest to use XQuery on the Web browser for client-side programming [12] to dynamically modify the Web page content displayed on a Web browser. Finally, people working with databases wanted to use XQuery to actually update the underlying databases. As a result, there is a recent proposal by W3C for extending XQuery with updates, called the XQuery Update Facility (XUF) [18], that addresses these needs.

Although XQuery expressions can be composed in arbitrary ways, XUF imposes many syntactic and semantic restrictions on updates. The most important static restriction is that updates are permitted to appear in certain places only, such as in the return part of a FLWOR (For-Let-Where-Order-Return) expression only. In addition, the target of insert, replace, and rename updates should be a singleton sequence, which can only be checked at run-time. XUF uses snapshot semantics, indicating that the results of the updates are not visible during query evaluation (that is, updates cannot commit during querying). In other words, updates are accumulated during the query evaluation and are executed at the end of the query. The XUF operational semantics [18] uses a pending update list to accumulate updates during the query execution. This update list is a collection of update primitives, which represent node state changes that have not yet been applied [18].

Although the XUF proposal fulfills an important need for web programmers, it also introduces some problems. The most important problem is the optimization of XQuery updates. While there are many optimization techniques and evaluation strategies based on various storage mappings for plain XQuery, there is still much work to be done on XUF query optimization. Current work uses framework-specific algebras for optimization [5, 11], which makes it hard to apply to other frameworks that use different algebras. In addition, one of the intended use of XQuery updates is to use them in various schema mappings and integration scenarios, where XML is used as an integration medium (as a virtual or materialized view) while a relational database is used for storage. Then, view updates expressed in XUF must be reflected to the underlying database [7], while SQL updates must result to the incremental modification of the XML view, if the view is materialized [8]. Both tasks can be greatly facilitated if XQuery updates and view mappings (which are typically expressed in XQuery) are fused and optimized in the same framework.

Currently, many people wrongly believe that there is no straightforward and efficient translation from XUF to XQuery, despite that both XQuery and XUF are computationally complete. One of the reasons for this belief is that XUF updates destructively update XML data while preserving their node identity, which cannot be simulated directly in plain XQuery. Another reason is that it may be very hard to develop a translator from XUF to XQuery that preserves the complex XUF semantics, as is currently described in

terms of pending lists of updates. Naive XUF translators that use recursive XQuery functions to apply the updates to the qualified XML nodes may satisfy the XUF semantics but the resulting recursive programs are difficult to optimize.

In this paper, we address the problem of translating XUF queries to plain (side-effect-free) XQuery expressions. The resulting XQuery reconstructs the mutable XML state, reflecting the updated values in the new state. The mutable state s of an XUF query is an XML tree that consists of all XML data whose nodes are updated by the XUF query. Central to our approach is the concept of the *S-complement*: Let v be a sequence of XML nodes $(v_1, \ldots, v_n)$, s be the mutable state, and u be a function, called a *context mapping function*, that can be applied to each of these v nodes. The S-complement of v is a function f so that the application of f to s and u, that is, the function call $(f\, s\, u)$, returns a copy of the state s in which, for all i, the nodes that have the same node identity as v_i have been replaced with $(u\, v_i)$. Our approach is to compile any XUF expression e to pure XQuery code that calculates the S-complement of the value of e. That is, this code creates a copy of the state that is equal to the original state in all places except those reachable by the expression e. The latter places are transformed by the context mapping function, which allows us to change their values at will. When e is used as an update destination, then updates targeting e can be translated to state transformations by just providing the appropriate context mapping. For example, if we want to insert a new node x before the node retrieved by the XQuery e (specified as **insert node** x **before** e in XUF), then we synthesize the S-complement of e and use the context function $u\, v = (x, v)$. This S-complement returns a copy of the state in all places except at the node v of the result of e, which is replaced by concatenating x with v.

The most important problem associated with updates in a purely functional setting is node sharing caused by aliasing. When nodes are duplicated in the mutable state, updates to a node must also update all the replicas. We address this problem by using our type system to detect those sub-expressions in the mutable state computation that duplicate nodes from an entire data source. These sub-expressions are factored out and the values returned by these expressions are labeled with a new type name. These type names are used during the XUF compilation to redirect operations over the state components under this type name to equivalent updates over the original data source. Then, the new state is calculated by applying the factored out sub-expressions to the modified data source.

The key contribution of our work is in the development of a novel framework for translating XQuery updates to plain XQuery that has the following characteristics:

- It uses source-to-source, compositional transformations only. Unlike related approaches that use framework-specific algebras to achieve a similar goal [5, 11], our work can be incorporated into any existing XQuery engine.
- It addresses data aliasing during updating. Expressions that return replicated XML nodes are detected and factored out during type-checking and updates to replicas are directed to the original data sources.
- It can be used for XQuery update optimization since it maps XUF updates to plain XQuery, which is amenable to optimization. XQuery optimization has already been addressed by our earlier work [7] and by related work [10, 16].
- It can be used for XUF type-checking and validation by simply type-checking the resulting XQuery code, a task that has already been described by XQuery semantics [17].
- It enables other transformations that are important to databases, such as the incremental maintenance of materialized XML views to make them consistent with the source data, without recreating the entire view from the database after each source update. Updates to the source data must be propagated through the view to become view updates, which can be done effectively if we translate updates to state transformations [8]. This was in fact our original motivation for the work presented in this paper.

The most important limitation of our approach is that it is schema-based. There is a good reason for requiring a schema: without a schema, one has to use recursive XQuery functions to apply each update to the qualified nodes. Then, the resulting code would be recursive, which is very hard to optimize. Another limitation is that the framework presented in this paper is not quite based on the XUF semantics [18], as our pending updates are not reordered according to the official semantics (which requires that insertions be done before updates and deletes). Finally, we have not covered some parts the XUF syntax, such as backward steps, but we believe that there is no fundamental reason why our framework cannot be extended to cover the full syntax.

The rest of this paper is organized as follows. Section 2 overviews our approach through the use of one example. Section 3 presents the detail rules of the translation and uses these rules to translate an XUF query step-by-step. Section 4 extends our framework to handle positional updates. Section 5 addresses the problem of data aliasing. Section 6 addresses the problem of reordering the pending updates required by the XUF semantics. Section 7 presents some normalization rules to fuse the layers of state transformations generated from our compositional rules. Section 8 reports on a prototype implementation and discusses efficiency issues, while Section 9 explains how to use our translation framework for XUF optimization. Finally, Section 10 compares our approach with related work.

2. Overview

Consider the following XQuery variable declaration:

```
declare variable $v as tp := doc('bib.xml');
```

where the type tp is equivalent to the XQuery type:

```
element bib { element article {
                  element title string,
                  element year string,
                  element author string* }* }
```

and the XUF query QUERY1

```
for $i in $v/article[author="Smith"]
where contains($i/title, "XQuery")
return replace value of node $i/year with 2009
```

which finds all articles published by Smith whose title contains the word XQuery, and replaces their year with 2009. Because of the snapshot semantics, the query result is:

```
for $i in $v/article[author="Smith"]
where contains($i/title, "XQuery")
return if count($i/year) = 1 then () else error()
```

that is, the replace update is translated to a simple test that imposes the XUF semantic restriction that the update destination must always return a singleton sequence. However, as a side-effect, QUERY1 changes the value of $v and these changes are visible to the follow-up queries. The new value of $v after this update is equivalent to the result of the following XQuery expression:

```
<bib>{ for $z in $v/article
       return if $z/author="Smith"
              then if contains($z/title,"XQuery")
                   then <article>{
                              $z/title,
                              <year>2009</year>,
                              $z/author
                         }</article>
                   else $z else $z }</bib>
```

That is, the new value of $v is the same as the old value, except of the years of the qualified articles, which have been replaced by 2009. There is an important discrepancy though. The XUF semantic requires that all nodes that are not updated should preserve their identity and document order, which is not true for the previous bib, article, and year element constructions. In a purely functional setting though, these properties can be brought into effect using an explicit node ID for each XML node, which is not directly accessible to programs, but is used indirectly for node comparisons. To preserve the node identity though, we expect that some system-generated programs, such as those that translate XUF to plain XQuery, to be the only ones permitted to explicitly modify the node identity during querying. The document order can be taken to be the node identity as long as each newly constructed element is assigned a succeeding ID, such as when an XML parser parses a document and stores it into memory. For instance, our implementation (HXQ [9]) uses the system attribute _id to assign a unique id to an element construction. (Although atomic values need node identity for node comparisons too, their identities do not need to be changed by our translations.) Then, the opening tag of each element construction must bind this attribute to the appropriate value to propagate the article identity:

```
<bib _id="{$v/@_id}">{
    for $z in $v/article
    return if $z/author="Smith"
           then if contains($z/title,"XQuery")
                then <article _id="{$z/@_id}">{
                           $z/title,
                           <year _id="{$z/year/@_id}">
                               2009</year>,
                           $z/author
                     }</article>
                else $z else $z }</bib>
```

To simplify our presentation, although essential, this node identity propagation is not shown in our rules and examples.

From the previous example, it is clear that, to translate an XUF query to a plain XQuery, one has to know the detailed type of the mutable state. It is actually easy to derive the mutable state of a given XUF query: it consists of all global variables (variables defined by a declare variable statement) that are used, directly or indirectly, in the destination of some XUF update in the query. These are the only variables accessible to the follow-up XUF queries; all other side effects are simply ignored since they cannot affect the results of the other queries. If, for example, an XUF query updates doc("bib.xml") directly, these changes would not be visible to the other queries, as long as they do not become persistent in a database and they do not rewrite the document[1], because if another query reads doc("bib.xml"), it will get the document nodes under different IDs. We call these variables, *mutable variables*. For example, the variable $v in the previous example is mutable, since the replace expression uses the variable $i, which depends on $v. Our framework imposes a very important restriction, called *non-interference*: There should be no two nodes in the XML data returned by the mutable variables with the same identity. Node interference is caused by aliasing. For example, the mutable variable $v := let $x := e return ($x,$x) violates non-interference since it duplicates the nodes of e. Essentially, the reason this restriction is imposed is that updates to a node must propagate to all its replicas, which is hard (but not impossible) to do in a purely functional setting. This restriction is not needed for hypothetical queries (transform queries), described in Section 3.5. We will lift this restriction in Section 5.

Suppose that the mutable state of a given XUF query consists of the mutable variables $\$v_1, \ldots, \$v_n$ defined by:

$$\textbf{declare variable } \$v_i \textbf{ as } t_i \ := \ e_i;$$

where the type t_i can be omitted since it can be inferred from e_i. Combining all these mutable variables, we derive the state type S, expressed as follows in the XQuery type system:

$$\text{element state } \{ \text{ element } v_1 \ t_1, \ldots, \text{element } v_n \ t_n \ \}$$

For the previous XUF query, the only mutable variable is $v, which indicates that the mutable state has the type:

```
element state {
   element v {
      element bib {
         element article {
            element title string,
            element year string,
            element author string*
         }* } } }
```

For each XUF query, our framework generates a **declare variable** statement that defines the variable $state to be the new mutable state, while each mutable variable $\$v_i$ is redefined to be $\$state/v_i$.

In our framework, given a state s of type S, an XUF expression e that returns a sequence of nodes of type t is translated to a pure XQuery expression of type S that reconstructs the state s, reflecting the updates in e. This is done in a compositional way, using source-to-source transformations. Our translation function $\mathcal{C}[\![e]\!]\,\sigma\,s\,u$ is parameterized by the *context mapping* u of type $t \rightarrow t'$ that maps the current context of e (the XQuery 'dot' value) to a new value. An identity mapping leaves the context as is; if e needs to be changed, then u must return a new value that replaces the current context of e with the new value. Basically, if e does not contain side-effects, $\mathcal{C}[\![e]\!]\,\sigma\,s\,u$ creates an exact copy of the state s but with the values associated with the results of e wrapped inside u. That way, if e is used as an update destination, then one simply needs to provide the proper mapping u to update e.

For example, the XPath expression

```
$v/article[author="Smith"]/year
```

is translated to

```
<state><v><bib>{
       for $w in $state/v/bib/article
       return if $w/author="Smith"
              then <article>{ $w/title,
                              u($w/year),
                              $w/author }</article>
              else $w
}</bib></v></state>
```

which allows us to 'update' the values returned by the XPath expression by providing an appropriate mapping u. This is exactly the S-complement of the XPath result. Note that, even though the XPath expression may return multiple years (the years of articles published by Smith), the context mapping u is applied to each individual year.

The environment σ in $\mathcal{C}[\![e]\!]\,\sigma\,s\,u$ is a binding list that maps XQuery variables to functions that, given a state s and a con-

[1] Persistent XQuery updates, which are translated to SQL updates, have been addressed in our earlier work [7].

text mapping u, calculate a new state, thus encapsulating the context mappings associated with the variable. That is, an XQuery variable in our mappings is bound to the S-complement of the variable value. For example, since $i in QUERY1 iterates over $v/article[author="Smith"], it is bound to the translation of this expression, which is

```
λs.λu. <state><v><bib>{
          for $w in s/v/bib/article
          return if $w/author="Smith"
                 then u($w) else $w
       }</bib></v></state>
```

where an anonymous function, such as $\lambda x.\lambda y.\, x + y$, is equivalent to the function f with $f\, x\, y = x + y$. (Note that, although we use higher-order mappings for convenience, the resulting XQuery code is first-order. In fact, the mapping algorithm can be implemented in a language that does not support anonymous functions, such as C, provided that we implement function substitution explicitly.)

When $i is used in the path step $i/year, we simply replace u in the $i binding with

```
λx. <article>{ x/title, u(x/year), x/author }</article>
```

which is derived from the type of $i. Then, updating $i/year in 'replace value of node $i/year with 2009', is simply using $u = \lambda x.$ <year>2009</year> to replace the current context with a new value.

Consider now the following update:

```
insert node a as first
into $v//article[author="Smith"]
```

which inserts a new article a before any paper written by Smith. Based on our translations, the update destination will use the u mapping as many times as the number of articles written by Smith. We cannot simply use $u = \lambda x.\, (a, x)$ because this mapping will insert a before every article x written by Smith. Fortunately, this update is not permitted in the XUF semantics: the update destination must always return a singleton sequence. Our framework enforces this restriction by generating the following code for this query:

```
if count($v//article[author="Smith"])=1 then () else error()
```

One may use $v//article[author="Smith"][0] as the update destination to make it a valid update. If this restriction were not imposed by XUF, our context mapping function should have included two more parameters: the sequence position and the sequence size, so that each one of the article mappings would have gotten a different position.

3. Translating XQuery Updates to Plain XQuery

In this section, we provide a source-to-source translation of XUF to plain XQuery. Since many of our translation rules are guided by data types, we impose certain limitations on XML types: we can only handle non-recursive XML types of the following form:

$$
\begin{array}{lll}
t, t_1, t_2 \in \text{Type} & & \\
t & ::= & \text{element } A\ t \mid \text{attribute } A\ t \mid (t_1, t_2) \mid (t_1 \,|\, t_2) \\
& \mid & () \mid t* \mid t+ \mid t? \mid \text{`an XQuery base type'}
\end{array}
$$

The types and type-checking rules for the XQuery data model (XDM) are described in detail by the XQuery Formal Semantics recommendation [17]. Furthermore, we do not allow ambiguous types, such as element A { element B t*, element B t* }. More specifically, for every type t, we should not get duplicate tags in tags(t), defined as follows:

$$
\begin{aligned}
\text{tags}(\text{element } A\ t) &= [\,A\,] \\
\text{tags}(\,(t_1, t_2)\,) &= \text{tags}(t_1) +\!\!+ \text{tags}(t_2) \\
\text{tags}(\,(t_1 \,|\, t_2)\,) &= \text{tags}(t_1) +\!\!+ \text{tags}(t_2) \\
\text{tags}(\,t*) &= \text{tags}(\,t+) \;=\; \text{tags}(\,t?) \;=\; \text{tags}(t) \\
\text{tags}(t) &= [\,] \qquad \textit{(otherwise)}
\end{aligned}
$$

Based on these restrictions, our framework excludes few valid types too, such as (element B t, element B t). We plan to address these restrictions in a future work.

Our framework handles the following XUF expressions:

$$
\begin{array}{lll}
\$v, \$v_1, \ldots, \$v_n & \in & \text{Variables} \\
e, e', e_1, \ldots, e_n & \in & \text{Expressions} \\
\text{axis} & \in & \{\text{ child, descendant, self, attribute }\} \\
\text{test} & ::= & * \mid tag
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & \$v \qquad \text{variable} \\
& \mid & . \mid () \mid \text{constant} \\
& \mid & e/\text{axis::test} \qquad \text{path step} \\
& \mid & e_1[e_2] \qquad \text{path predicate} \\
& \mid & e/\text{node}() \qquad \text{element content} \\
& \mid & (e_1, e_2) \qquad \text{sequence concat.} \\
& \mid & \textbf{element}\ \{e_1\}\ \{e_2\} \qquad \text{element constr.} \\
& \mid & f(e_1, \ldots, e_n) \qquad \text{function call} \\
& \mid & e_1\ \text{opr}\ e_2 \qquad \text{binary operation} \\
& \mid & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \qquad \text{conditional} \\
& \mid & \textbf{for}\ \$v\ \textbf{in}\ e_1\ [\textbf{where}\ e_2]\ \textbf{return}\ e_3 \qquad \text{for-loop} \\
& \mid & \textbf{let}\ \$v\ :=\ e_1\ \textbf{return}\ e_2 \qquad \text{let-binding} \\
& \mid & \textbf{replace}\ [\textbf{value of}]\ \textbf{node}\ e_1\ \textbf{with}\ e_2 \qquad \text{replacement} \\
& \mid & \textbf{delete node}\ e \qquad \text{deletion} \\
& \mid & \textbf{insert node}\ e_1\ \textbf{into}\ e_2 \qquad \text{insertion} \\
& \mid & \textbf{insert node}\ e_1\ \textbf{as last into}\ e_2 \qquad \text{insert-last} \\
& \mid & \textbf{insert node}\ e_1\ \textbf{as first into}\ e_2 \qquad \text{insert-first} \\
& \mid & \textbf{insert node}\ e_1\ \textbf{before}\ e_2 \qquad \text{insert-before} \\
& \mid & \textbf{insert node}\ e_1\ \textbf{after}\ e_2 \qquad \text{insert-after} \\
& \mid & \textbf{rename node}\ e_1\ \textbf{with}\ e_2 \qquad \text{renaming} \\
& \mid & \textbf{copy}\ \$v_1 := e_1,\ \ldots,\ \$v_n := e_n\ \textbf{modify}\ e\ \textbf{return}\ e' \\
& & \qquad \text{transform}
\end{array}
$$

plus the usual abbreviations: e/A for $e/\text{child}::A$, <A>{e}</A> for **element** {"A"} {e}, etc.

Our goal in this section is to translate an XUF expression Q to a plain XQuery. We assume that the mutable state of Q, which consists of the mutable variables $\$v_1, \ldots, \$v_n$, satisfies the non-interference constraint. Given an XUF expression e and a state s of type S, the $\mathcal{C}[\![e]\!]\,\sigma\, s\, u$ derives a pure XQuery expression that reconstructs the state s, reflecting the updates in e. Essentially, the function $\mathcal{C}[\![e]\!]\,\sigma$ synthesizes an XQuery that calculates the S-complement of e. It is of type $S \to (t \to t') \to S$, where t is the type of the FLWOR variable $\$v$ in **for** $\$v$ **in** e **return** e'. That is, based on the XQuery type system [17], t can be derived from the type of e by stripping out the outer qualifiers and converting sequence types to union types:

$$
\begin{aligned}
\text{strip}(\,(t_1, t_2)\,) &= \text{strip}(t_1) \,|\, \text{strip}(t_2) \\
\text{strip}(\,(t_1 \,|\, t_2)\,) &= \text{strip}(t_1) \,|\, \text{strip}(t_2) \\
\text{strip}(\,t*) &= \text{strip}(\,t+) = \text{strip}(\,t?) = \text{strip}(t) \\
\text{strip}(t) &= t \qquad \textit{(otherwise)}
\end{aligned}
$$

The context mapping $u : t \to t'$ maps the current context of e to a new value. Basically, $\mathcal{C}[\![e]\!]\,\sigma\, s\, u$ reconstructs the entire state s by ignoring all side-effect-free expressions and by incorporating side-effects into the context mapping function u. The environment

σ is a binding list that maps XQuery variables to functions that, given a state s and a context mapping function u, calculate a new state, thus mapping the variable to the S-complement of its value. The notation $\sigma[\$v]$ extracts the binding of the variable $\$v$, while $\sigma[\$v/f]$ extends the environment with a new binding from $\$v$ to f. Initially, e is the XUF query Q, u is the identity function **id** that leaves the current context unchanged, s is the state before the XUF query:

$$<\text{state}><v_1>\{\$v_1\}</v_1>\cdots<v_n>\{\$v_n\}</v_n></\text{state}>$$

and σ contains the bindings from each mutable variable $\$v_i$ to

$$\lambda s.\lambda u.\ <\text{state}>\{\ s/v_1,\ldots,<v_i>\{u(s/v_i/\text{node}())\}</v_i>,\\ \ldots,s/v_n\ \}</\text{state}>$$

based on the previously defined type S. For the XUF query Q, our framework generates the new state, $\$ns$:

declare variable $\$ns := \mathcal{C}[\![Q]\!]\,\sigma\ s$ **id**;

which is used to redefine each mutable variable:

declare variable $\$v_i := \$ns/v_i/\text{node}()$;

The actual value of the XUF query Q is $\mathcal{E}[\![Q]\!]\,(\text{error}())$, where the function $\mathcal{E}[\![e]\!]\,x$ returns the expression e, where all XUF update expressions (except the transform updates) have been removed and the current context (dot) has been replaced with x. The following are some of the rules for $\mathcal{E}[\![e]\!]$:

$$\begin{aligned}
&\mathcal{E}[\![.]\!]\,x = x\\
&\mathcal{E}[\![\$v]\!]\,x = \$v\\
&\mathcal{E}[\![(e_1,e_2)]\!]\,x = (\mathcal{E}[\![e_1]\!]\,x, \mathcal{E}[\![e_2]\!]\,x)\\
&\mathcal{E}[\![\textbf{delete node}\ e]\!]\,x = ()\\
&\mathcal{E}[\![\textbf{replace node}\ e_1\ \textbf{with}\ e_2]\!]\,x\\
&\qquad = \textbf{if}\ \text{count}(\mathcal{E}[\![e_1]\!]\,x) = 1\ \textbf{then}\ ()\ \textbf{else}\ \text{error}()
\end{aligned}$$

For example, $\mathcal{E}[\![A/B]\!]\,x$ is equal to $x/A/B$ since the path expression A/B is equivalent to $./A/B$. Note that a mutable variable $\$v$ is bound to its original value before the XUF query. The translation of transform updates (hypothetical queries) is given in Section 3.5.

In addition to type correctness:

$$e : t_e \wedge t = \text{strip}(t_e)\ \Rightarrow\ \mathcal{C}[\![e]\!]\,\sigma : S \to (t \to t') \to S$$

we have provided a correctness proof sketch for our XUF translation algorithm. More specifically, we have proved that the XQuery expressions composed by our translation algorithm modify the XML data in the way described by the XUF update semantics. In the standard XUF semantics [18], an XUF update results to a pending list of update primitives, which is reordered before is evaluated, so that insertions are done before updates and deletes. Our translation algorithm generates programs that are precisely equivalent to the results of these pending lists but without the reordering.

Theorem 1. *The resulting state calculated by $\mathcal{C}[\![e]\!]\,\sigma\ s$ **id** is equivalent to the result of applying (without reordering) the pending update list of e to the state s.*

The proof is given in Appendix A. Section 6 addresses the reordering of the pending updates (using multiple state transformers).

3.1 Update Translation

We first describe the rules for $\mathcal{C}[\![e]\!]\,\sigma\,s\,u$ in which e is an XUF update expression.

Replacing Nodes and Values. Replace updates may take two forms: The first form replaces a single node (the result of the update destination e_1) with a new sequence of zero or more nodes (from e_2):

$$\begin{aligned}
&\mathcal{C}[\![\textbf{replace node}\ e_1\ \textbf{with}\ e_2]\!]\,\sigma\,s\,u \qquad \text{(repl)}\\
&\quad = \mathcal{C}[\![e_1]\!]\,\sigma\,s\,(\lambda x.\,u(\mathcal{E}[\![e_2]\!]\,x))
\end{aligned}$$

The XUF semantics requires that neither e_1 nor e_2 may contain an update expression (that is, they must be plain XQueries). This is also true for the other update expressions. The resulting code reconstructs the state by copying all its nodes except those nodes x returned by e_1, which are replaced by the value of e_2, which is $\mathcal{E}[\![e_2]\!]\,x$. The second form of the replace expression modifies the value of a node e_1 while preserving its node identity:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{replace value of node}\ e_1\ \textbf{with}\ e_2]\!]\,\sigma\,s\,u \qquad \text{(repl-val)}\\
&\quad = \mathcal{C}[\![e_1]\!]\,\sigma\,s\,(\lambda x.\,u(<A>\{\mathcal{E}[\![e_2]\!]\,x\}</A>))
\end{aligned}$$

given that e_1 is of type, element $A\,t$. There is a similar rule for attribute values. Recall that the node identity can be propagated to newly constructed elements as a special attribute value derived from the original nodes.

Deleting Nodes. A delete expression deletes zero or more nodes from e:

$$\mathcal{C}[\![\textbf{delete node}\ e]\!]\,\sigma\,s\,u\ =\ \mathcal{C}[\![e]\!]\,\sigma\,s\,(\lambda x.\,u(()))\qquad\text{(del)}$$

Unlike the other updates, the update destination e does not have to return a singleton sequence. The generated code replaces the current context with $()$, which, when embedded in a sequence, will result to the removal of the element.

Inserting Nodes. There are five variations of the insert expression. Using the insert-last expression, the inserted nodes e_1 become the last children of the target node e_2:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{insert node}\ e_1\ \textbf{as last into}\ e_2]\!]\,\sigma\,s\,u \qquad \text{(insl)}\\
&\quad = \mathcal{C}[\![e_2]\!]\,\sigma\,s\,(\lambda x.\,u(<A>\{x/\text{node}(),\ \mathcal{E}[\![e_1]\!]\,x\}</A>))
\end{aligned}$$

given that e_2 is of type, element $A\,t$. The resulting code reconstructs each node from e_1 by appending the node sequence e_2 to the children nodes. Using the insert-first expression, the inserted nodes e_1 become the first children of the target node e_2:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{insert node}\ e_1\ \textbf{as first into}\ e_2]\!]\,\sigma\,s\,u \qquad \text{(insf)}\\
&\quad = \mathcal{C}[\![e_2]\!]\,\sigma\,s\,(\lambda x.\,u(<A>\{\mathcal{E}[\![e_1]\!]\,x,\ x/\text{node}()\}</A>))
\end{aligned}$$

given that e_2 is of type, element $A\,t$. The **insert node** e_1 **into** e_2 expression simply makes the nodes e_1 children of the target node e_2. The positions of the inserted nodes among the children of the target node are implementation-dependent, so we may implement this operation as an insert-last expression. Using the insert-before expression, the inserted nodes e_1 become the preceding siblings of the target node e_2:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{insert node}\ e_1\ \textbf{before}\ e_2]\!]\,\sigma\,s\,u \qquad \text{(insb)}\\
&\qquad = \mathcal{C}[\![e_2]\!]\,\sigma\,s\,(\lambda x.\,u((\mathcal{E}[\![e_1]\!]\,x,\ x)))
\end{aligned}$$

Finally, using the insert-after expression, the inserted nodes e_1 become the following siblings of the target node e_2:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{insert node}\ e_1\ \textbf{after}\ e_2]\!]\,\sigma\,s\,u \qquad \text{(insa)}\\
&\qquad = \mathcal{C}[\![e_2]\!]\,\sigma\,s\,(\lambda x.\,u((x,\ \mathcal{E}[\![e_1]\!]\,x)))
\end{aligned}$$

Renaming Nodes. A rename expression replaces the name property of a data node e with a new name q:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{rename node}\ e\ \textbf{with}\ q]\!]\,\sigma\,s\,u \qquad \text{(renm)}\\
&\quad = \mathcal{C}[\![e]\!]\,\sigma\,s\,(\lambda x.\,u(\textbf{element}\ \{\mathcal{E}[\![q]\!]\,x\}\ \{x/\text{node}()\}))
\end{aligned}$$

This rule applies when e is of an element type. There is a similar rule when e is of an attribute type.

3.2 The Translation of Plain XQuery Expressions

We now present the rules for reconstructing the mutable state for the XUF expressions that are plain XQuery expressions. If none of the rules apply, then $\mathcal{C}[\![e]\!]\,\sigma\,s\,u\;=\;s$. Variables are introduced by FLWOR expressions (for-loops and let-bindings). Based on the XUF semantics, a FLWOR expression can only have updates in its return body. A for-loop is translated as follows:

$$\mathcal{C}[\![\textbf{for}\;\$v\;\textbf{in}\;e_1\;\textbf{where}\;p\;\textbf{return}\;e_2]\!]\,\sigma\,s\,u\;=\;\mathcal{C}[\![e_2]\!]\,\sigma'\,s\,u \quad \text{(for)}$$

where σ' extends σ with a new binding: $\sigma' = \sigma\textbf{[}\$v/f\textbf{]}$ so that

$$f = \lambda s.\lambda u.\,\mathcal{C}[\![e_1]\!]\,\sigma\,s\,(\lambda x.\,\text{subst}\,\$v\,x\,(\textbf{if}\;\mathcal{E}[\![p]\!]\,x\;\textbf{then}\;u(x)\;\textbf{else}\;x))$$

where subst $\$v\,x\,e$ substitutes all occurrences of the variable $\$v$ in the term e with the term x. This rule indicates that the resulting state is equal to the state of the for-loop return expression, e_2, under the binding of the for-loop variable $\$v$ to the state of e_1. The predicate p is moved inside the mapping function so that if the predicate is true, the context is mapped through u; otherwise the context is returned unchanged. Let-bindings are translated in a similar way:

$$\mathcal{C}[\![\textbf{let}\;\$v\;:=\;e_1\;\textbf{return}\;e_2]\!]\,\sigma\,s\,u\;=\;\mathcal{C}[\![e_2]\!]\,\sigma'\,s\,u \quad \text{(let)}$$

where $\sigma' = \sigma\textbf{[}\$v/(\lambda s.\lambda u.\,\text{subst}\,\$v\,x\,(\mathcal{C}[\![e_1]\!]\,\sigma\,s\,u))\textbf{]}$. Let-bindings, such as **let** $v:=e return ($v,$v), may apply the same update in e multiple times, if expanded, but such expressions are forbidden by XUF. The variable bindings are used in variable translations:

$$\mathcal{C}[\![\$v]\!]\,\sigma\,s\,u = \sigma\textbf{[}\$v\textbf{]}\,s\,u \quad \text{(subst)}$$

XPath predicates are translated in a way similar to FLWOR predicates:

$$\begin{aligned}&\mathcal{C}[\![e\textbf{[}p\textbf{]}]\!]\,\sigma\,s\,u \quad \text{(cond)}\\ &=\;\mathcal{C}[\![e]\!]\,\sigma\,s\,(\lambda x.\,\textbf{if}\;\mathcal{E}[\![p]\!]\,x\;\textbf{then}\;u(x)\;\textbf{else}\;x)\end{aligned}$$

The e/node() expression extracts the child nodes of e:

$$\begin{aligned}&\mathcal{C}[\![e/\text{node()}]\!]\,\sigma\,s\,u \quad \text{(node)}\\ &=\;\mathcal{C}[\![e]\!]\,\sigma\,s\,(\lambda x.\,\text{<A>}\{u\,(x/\text{node()})\}\text{</A>})\end{aligned}$$

given that e is of type, element $A\,t$. Element constructions are translated as follows:

$$\begin{aligned}&\mathcal{C}[\![\text{<A>}\{e\}\text{</A>}]\!]\,\sigma\,s\,u \quad \text{(constr)}\\ &=\;\mathcal{C}[\![e]\!]\,\sigma\,s\,(\lambda x.\,(u\,(\text{<A>}\{x\}\text{</A>}))/\text{node()})\end{aligned}$$

A sequence concatenation is handled by using the resulting state of the first sequence component as the initial state of the second component:

$$\mathcal{C}[\![(e_1,e_2)]\!]\,\sigma\,s\,u = \mathcal{C}[\![e_2]\!]\,\sigma\,(\mathcal{C}[\![e_1]\!]\,\sigma\,s\,u)\,u \quad \text{(seq)}$$

Note that, if $e_1 : t_1$ and $e_2 : t_2$, then the domain of u in Rule (seq) is equal to $\text{strip}(t_1,t_2) = t_1\,|\,t_2$, that is, u can apply to the contexts of both e_1 and e_2. Given a non-recursive updating function $f(\$v_1,\ldots,\$v_n) = \mathit{body}$, a call to f is translated to:

$$\begin{aligned}&\mathcal{C}[\![f(e_1,\ldots,e_n)]\!]\,\sigma\,s\,u \quad \text{(appl)}\\ &=\;\mathcal{C}[\![\mathit{body}]\!]\;\textbf{[}\$v_1/(\mathcal{C}[\![e_1]\!]\,\sigma),\ldots,\$v_n/(\mathcal{C}[\![e_n]\!]\,\sigma)\textbf{]}\,s\,u\end{aligned}$$

It binds the function parameters to the mappings derived from the call arguments and uses these bindings to derive the mapping of the function body. In XUF, functions that contain update expressions must be defined using the declare updating function syntax.

3.3 Translation of XPath Steps

The rules for translating XPath steps are the most complex ones. Let e be a sequence of nodes of type t (that is, if e has type t', then $t = \text{strip}(t')$), then:

$$\begin{aligned}&\mathcal{C}[\![e/\text{axis::test}]\!]\,\sigma\,s\,u \quad \text{(step)}\\ &=\;\mathcal{C}[\![e]\!]\,\sigma\,s\,(\lambda x.\,\mathcal{P}[\![t]\!]\,x\,(\text{axis::test})\,0\,u)\end{aligned}$$

This rule uses the function $\mathcal{P}[\![t]\!]\,x\,(\text{axis::test})\,n\,u$ to handle forward XPath steps, where the XML nesting level n is initially zero (must be 1 to apply the child step) and x is the current context. It generates an expression of type t that creates a copy of the value of x (which is of type t), except of the nodes that can be reached by the path, which are wrapped by the function u. Currently, the rules for $\mathcal{P}[\![t]\!]$, which are guided by the type t of e, can handle child, descendant, self, and attribute steps only. First, if the type t is an element type and the step is not applicable to the type t (that is, if it always returns $()$), then:

$$\begin{aligned}&\mathcal{P}[\![\text{element}\,A\,t]\!]\,x\;\text{step}\;n\,u \quad \text{(p-expr)}\\ &=\;\mathcal{P}[\![\text{element}\,A\,t*\,]\!]\,x\;\text{step}\;n\,u\;=\;x/\text{self::}A\end{aligned}$$

For the self step, we have to be at the nesting level 0 to check whether the step tag matches the element tag. If it does, we wrap u around x, if not, we simply return x:

$$\begin{aligned}&\mathcal{P}[\![\text{element}\,A\,t]\!]\,x\,(\text{self::}B)\,0\,u \quad \text{(p-self)}\\ &=\;\text{if}\;A = B \vee B = *\;\text{then}\;u(x/\text{self::}A)\;\text{else}\;x/\text{self::}A\end{aligned}$$

For a child step though, we have to be at the nesting level 1:

$$\begin{aligned}&\mathcal{P}[\![\text{element}\,A\,t]\!]\,x\,(\text{child::}B)\,1\,u \quad \text{(p-child)}\\ &=\;\text{if}\;A = B \vee B = *\;\text{then}\;u(x/\text{self::}A)\;\text{else}\;x/\text{self::}A\end{aligned}$$

For a descendant step, we have to apply u recursively to all applicable nodes at any level, generating code to reconstruct the elements as we proceed:

$$\begin{aligned}&\mathcal{P}[\![\text{element}\,A\,t]\!]\,x\,(\text{descendant::}B)\,n\,u \quad \text{(p-desc)}\\ &=\;\text{if}\;n > 0 \wedge (A = B \vee B = *)\;\text{then}\;u(z)\;\text{else}\;z\end{aligned}$$

where z is equal to:

$$\text{<A>}\{\mathcal{P}[\![t]\!]\,(x/\text{self::}A/\text{node()})\,(\text{descendant::}B)\,(n{+}1)\,u\}\text{</A>}$$

Attribute steps can only apply at level 0:

$$\begin{aligned}&\mathcal{P}[\![\text{attribute}\,A\,t]\!]\,x\,(\text{attribute::}B)\,0\,u \quad \text{(p-attr)}\\ &=\;\text{if}\;A = B \vee B = *\;\text{then}\;u(x/@A)\;\text{else}\;x/@A\end{aligned}$$

Otherwise (for child step at level 0 or descendant step at any level), we generate code that reconstructs the node and we apply our method recursively:

$$\begin{aligned}&\mathcal{P}[\![\text{element}\,A\,t]\!]\,x\;\text{step}\;n\,u \quad \text{(p-step)}\\ &=\;\text{<A>}\{\mathcal{P}[\![t]\!]\,(x/\text{self::}A/\text{node()})\;\text{step}\;(n+1)\,u\}\text{</A>}\end{aligned}$$

Sequence types are simple to handle:

$$\begin{aligned}&\mathcal{P}[\![(t_1,t_2)]\!]\,x\;\text{step}\;n\,u \quad \text{(p-seq)}\\ &=\;(\mathcal{P}[\![t_1]\!]\,x\;\text{step}\;n\,u,\;\mathcal{P}[\![t_2]\!]\,x\;\text{step}\;n\,u)\end{aligned}$$

For a star type, we generate a for-loop. The most common form is a star over an element type:

$$\begin{aligned}&\mathcal{P}[\![\,\text{element}\,A\,t*\,]\!]\,x\;\text{step}\;n\,u \quad \text{(p-star)}\\ &=\;\textbf{for}\;\$z\;\textbf{in}\;x/\text{self::}A\\ &\quad\;\textbf{return}\;\text{<A>}\{\mathcal{P}[\![t]\!]\,(\$z/\text{node()})\;\text{step}\;(n+1)\,u\}\text{</A>}\end{aligned}$$

This rule is a special case of the following rule:

$$\begin{aligned}&\mathcal{P}[\![\,t*\,]\!]\,x\;\text{step}\;n\,u \quad \text{(p-star')}\\ &=\;\textbf{for}\;\$z\;\textbf{in}\;x\textbf{[}\text{self::}A_1\,|\cdots|\text{self::}A_k\textbf{]}\;\textbf{return}\;\mathcal{P}[\![t]\!]\,\$z\;\text{step}\;n\,u\end{aligned}$$

where $A_1, \ldots, A_k$ are the tag names in tags(t), defined in Section 3. Union types require special treatment:

$$\begin{aligned}&\mathcal{P}[\![(t_1 \mid t_2)]\!]\, x \text{ step } n\, u \qquad \text{(p-sum)}\\ &\quad = \textbf{ if } \mathcal{D}[\![t_1]\!]\,[\![t_2]\!]\, x \textbf{ then } \mathcal{P}[\![t_1]\!]\, x \text{ step } n\, u\\ &\qquad\qquad \textbf{else } \mathcal{P}[\![t_2]\!]\, x \text{ step } n\, u\end{aligned}$$

The discriminator $\mathcal{D}[\![t_1]\!]\,[\![t_2]\!]\, x$, where x is of type $t_1|t_2$, returns an expression that checks whether x is of type t_1 or t_2, using a minimal number of checks. This expression evaluates to a non-empty sequence if x is of type t_1 and to () if it is of type t_2. If it cannot discriminate t_1 from t_2, it returns $\bot$. It is defined as follows. If both types are elements of the same tag, we have no choice but to check the children of the node:

$$\mathcal{D}[\![\text{element } A\, t_1]\!]\,[\![\text{element } A\, t_2]\!]\, y = \mathcal{D}[\![t_1]\!]\,[\![t_2]\!]\,(y/\text{self::}A/\text{node}())$$

Otherwise (if the tags are different), we simply enforce the node to be of the first tag:

$$\mathcal{D}[\![\text{element } A\, t_1]\!]\,[\![\text{element } B\, t_2]\!]\, y = y/\text{self::}A$$

For sequence types, any sequence component can be used to discriminate:

$$\mathcal{D}[\![(t_1, t_2)]\!]\,[\![t']\!]\, y = \text{if } \mathcal{D}[\![t_1]\!]\,[\![t']\!] = \bot \text{ then } \mathcal{D}[\![t_2]\!]\,[\![t']\!] \text{ else } \mathcal{D}[\![t_1]\!]\,[\![t']\!]$$

$$\mathcal{D}[\![t]\!]\,[\![(t'_1, t'_2)]\!]\, y = \text{if } \mathcal{D}[\![t]\!]\,[\![t'_1]\!] = \bot \text{ then } \mathcal{D}[\![t]\!]\,[\![t'_2]\!] \text{ else } \mathcal{D}[\![t]\!]\,[\![t'_1]\!]$$

while for union types, every union alternative must be considered:

$$\begin{aligned}\mathcal{D}[\![(t_1|t_2)]\!]\,[\![t']\!]\, y &= \mathcal{D}[\![t_1]\!]\,[\![t']\!] \textbf{ and } \mathcal{D}[\![t_2]\!]\,[\![t']\!]\\ \mathcal{D}[\![t]\!]\,[\![(t'_1|t'_2)]\!]t'\, y &= \mathcal{D}[\![t]\!]\,[\![t'_1]\!] \textbf{ and } \mathcal{D}[\![t]\!]\,[\![t'_2]\!]\end{aligned}$$

given that none of these $\mathcal{D}$ terms is $\bot$. Otherwise, $\mathcal{D}[\![t_1]\!]\,[\![t_2]\!]\, y = \bot$, which indicates that we cannot discriminate between t_1 and t_2.

3.4 Example

As an example, we translate the following XUF query, QUERY2, which adds one more update to QUERY1 to illustrate sequence concatenation:

```
for $i in $v/article[author="Smith"]
where contains($i/title,"XQuery")
return ( replace value of node $i/year with 2009,
         insert node <author>Jones</author>
              as last into $i )
```

For clarity, the environment σ is omitted. Initially, the mutable variable $v has type: element bib {...}, given in Section 2, and is bound to λs.λu.<state><v>{u (s/v/node())}</v></state> in σ. From Rules (step), (p-step), (p-star), and (p-child), we can deduce that:

```
C[[$v/article]] s u
= C[[$v]] s (λx. P[[element bib {...}]] x (child::article) 0 u)
= <state><v>{
      P[[element bib {...}]] (s/v/node()) (child::article) 0 u
  }</v></state>
= <state><v>{
      P[[element article {...}*]] (s/v/node()/self::bib/node())
        (child::article) 1 u
  }</v></state>
= <state><v>{
      for $z in P[[element article {...}]]
                  (s/v/bib/node()/self::article)
                  (child::article) 1
      return u($z)
  }</v></state>
= <state><v>{
      for $z in s/v/bib/article return u($z)
  }</v></state>
```

Using this result and Rule (cond), we get:

```
C[[ $v/article[author="Smith"] ]] s u
= C[[$v/article]] s (λx. if E[[author="Smith"]] x
                         then u(x) else x)
= <state><v>{ for $z in s/v/bib/article
              return if E[[author="Smith"]] $z
                     then u($z) else $z }</v></state>
= <state><v>{ for $z in s/v/bib/article
              return if $z/author="Smith"
                     then u($z) else $z }</v></state>
```

Using Rule (for), QUERY2 is translated to:

```
C[[( replace value of node $i/year with 2009,
     insert node <author>Jones</author>
          as last into $i )]] s u
```

where variable $i is bound to

```
λs.λu. C[[$v/article[author="Smith"]]] s
       (λz. if E[[contains(z/title,"XQuery")]] z
            then u(z) else z)
= λs.λu. C[[$v/article[author="Smith"]]] s
         (λz.if contains(z/title,"XQuery") then u(z) else z)
= λs.λu. <state><v>{
           for $z in s/v/bib/article
           return if $z/author="Smith"
                  then if contains($z/title,"XQuery")
                       then u($z) else $z
                  else $z }</v></state>
```

Letting u' x = u(<year>2009</year>) and using Rule (repl-val), we have:

```
C[[ replace value of node $i/year with 2009 ]] s u
= C[[$i/year]] s (λx. u(<year>2009</year>))
= C[[$i/year]] s u'
= C[[$i]] s (λx.P[[element article {...}]] x (child::year) 0 u')
```

As before:

```
P[[element article {...}]] x (child::year) 0 u'
= <article>{ P[[(element title string, ...)]]
               (x/self::article/node()) 1 u' }</article>
= <article>{
      x/self::article/node()/self::title,
      u'(x/self::article/node()/self::year),
      x/self::article/node()/self::author }</article>
= <article>{
      x/self::article/title, u'(x/self::article/year),
      x/self::article/author }</article>
```

Therefore, using Rule (subst), we get:

```
C[[replace value of node $i/year with 2009]] s u
= C[[$i]] s (λx. <article>{
                   x/self::article/title,
                   u(<year>2009</year>),
                   x/self::article/author }</article>)
= <state><v>{
    for $z in s/v/bib/article
    return if $z/author="Smith"
           then if contains($z/title,"XQuery")
                then <article>{
                       $z/self::article/title,
                       u(<year>2009</year>),
                       $z/self::article/author
                     }</article>
                else $z
           else $z }</v></state>
```

Using Rule (insl), the second update is translated to:

```
C⟦ insert node <author>Jones</author>
        as last into $i ⟧ s u
= C⟦$i⟧ s (λx. u(<article>{ x/node(),
                    <author>Jones</author>
                }</article>))
= <state><v>{
      for $z in s/v/bib/article
      return if $z/author="Smith"
             then if contains($z/title,"XQuery")
                  then u(<article>{ $z/node(),
                                    <author>Jones</author>
                                }</article>)
                  else $z else $z }</v></state>
```

Finally, using Rule (seq), the translation of QUERY2 is derived by passing the resulting state of the first update as an input state to the second update:

```
C⟦ insert node <author>Jones</author>
        as last into $i ⟧
   (C⟦replace value of node $i/year with 2009⟧ s id) id
```

which is normalized to:

```
<state><v>{
   for $z in s/v/bib/article
   return if $z/author="Smith"
          then if contains($z/title,"XQuery")
               then <article>{
                        $z/title,
                        <year>2009</year>,
                        $z/author,
                        <author>Jones</author>
                    }</article>
               else $z else $z }</v></state>
```

3.5 Hypothetical Queries

A hypothetical XUF update (a transform expression) creates modified copies of XML data. It binds a number of variables to exact copies of XML data that are assigned new node identities and it performs updates against these variables. Simple transform expressions that create a copy of one variable only can be easily handled by our system as follows:

$$\begin{aligned}&\mathcal{E}[\![\textbf{copy}\ \$v_1 := e_1\ \textbf{modify}\ e\ \textbf{return}\ e']\!]\ x\\ &=\ \textbf{let}\ \$v_1 := \mathcal{C}[\![e]\!]\ \textbf{[}\$v_1/(\lambda s.\lambda u.\ u\ s)\textbf{]}\ (\mathcal{E}[\![e_1]\!]\ x)\ \textbf{id}\\ &\quad\ \textbf{return}\ \mathcal{E}[\![e']\!]\ (\text{error}())\end{aligned}$$

This transform expression creates a copy of the value of e_1 (with new node identities), binds it to $\$v_1$, and modifies it using updates in e. Variable $\$v_1$ can only be accessed in e'. Neither e_1 nor e' may contain updates. This expression is translated by $\mathcal{C}$ using the value of e_1 as an initial state.

If the transform expression uses multiple variables, we can construct a state from all these variables, as we did for regular updates:

$$\begin{aligned}&\mathcal{E}[\![\textbf{copy}\ \$v_1 := e_1,\ \ldots,\ \$v_n := e_n\ \textbf{modify}\ e\ \textbf{return}\ e']\!]\ x\\ &=\ \textbf{let}\ \$s := \mathcal{C}[\![e]\!]\ \sigma\ S\ \textbf{id},\ \$v_1 := \$s/v_1,\ \ldots,\ \$v_n := \$s/v_n\\ &\quad\ \textbf{return}\ \mathcal{E}[\![e']\!]\ (\text{error}())\end{aligned}$$

where S is:

$$\begin{aligned}&\text{<state><}v_1\text{>}\{\mathcal{E}[\![e_1]\!]\ x\}\text{</}v_1\text{>}\\ &\quad\ldots\text{<}v_n\text{>}\{\mathcal{E}[\![e_n]\!]\ x\}\text{</}v_n\text{></state>}\end{aligned}$$

and σ contains the bindings from each variable $\$v_i$ to

$$\begin{aligned}&\lambda s.\lambda u.\ \text{<state>}\{\ s/v_1,\ldots,\text{<}v_i\text{>}\{u(s/v_i/\text{node}())\}\text{</}v_i\text{>},\\ &\qquad\ldots, s/v_n\ \}\text{</state>}\end{aligned}$$

Note that, unlike regular updates, there is no interference among the variables $\$v_i$ since every copy has new node identities.

4. Handling Positional Queries

The context mappings described in the previous section are oblivious to the element position. Consequently, they cannot handle positional expressions, such as position(), last(), sequence indexing e**[**i**]** for an integer i, and 'at' bindings in FLWOR expressions. This section extends our translations to cope with positional expressions. Our new state transformations compile an expression e that returns a sequence of nodes of type t to a function $S \to (t \to \text{int} \to t') \to S$, where the integer parameter of the context mapping provides the position of the current context node within its parent. If e_2 is of type int, then the XPath indexing e_1**[**e_2**]** is translated to:

$$\begin{aligned}&\mathcal{C}[\![e_1\textbf{[}e_2\textbf{]}]\!]\ \sigma\ s\ u \qquad\qquad (\text{index})\\ &=\ \mathcal{C}[\![e_1]\!]\ \sigma\ s\ (\lambda x.\lambda p.\ \textbf{if}\ p = \mathcal{E}[\![e_2]\!]\ x\ p\ \textbf{then}\ u\ x\ p\ \textbf{else}\ x)\end{aligned}$$

where the function $\mathcal{E}[\![e]\!]\ x\ p$ returns the expression e, where all XUF update expressions in e (except the transform updates) have been replaced with $()$, the current context (dot) has been replaced with x, and position() has been replaced with p. The XPath step translations must now be parameterized by the current position p:

$$\begin{aligned}&\mathcal{C}[\![e/\text{axis::test}]\!]\ \sigma\ s\ u \qquad\qquad (\text{step})\\ &=\ \mathcal{C}[\![e]\!]\ \sigma\ s\ (\lambda x.\lambda p.\ \mathcal{P}[\![t]\!]\ x\ p\ (\text{axis::test})\ 0\ u)\end{aligned}$$

and apply the context mapping to the qualified elements using both the context x and the position p:

$$\begin{aligned}&\mathcal{P}[\![\text{element}\ A\ t]\!]\ x\ p\ (\text{child::}B)\ 1\ u \qquad\qquad (\text{p-child})\\ &=\ \text{if}\ A = B \vee B = *\ \text{then}\ u\ (x/\text{self::}A)\ p\ \text{else}\ x/\text{self::}A\end{aligned}$$

The positions are set when the value has a star type:

$$\begin{aligned}&\mathcal{P}[\![\ \text{element}\ A\ t*\ ]\!]\ x\ \text{step}\ n\ u \qquad\qquad (\text{p-star})\\ &=\ \textbf{for}\ \$z\ \textbf{at}\ \$i\ \textbf{in}\ x/\text{self::}A\\ &\quad\ \textbf{return}\ \text{<}A\text{>}\{\mathcal{P}[\![t]\!]\ (\$z/\text{node}())\ \$i\ \text{step}\ (n+1)\ u\}\text{</}A\text{>}\end{aligned}$$

For example,

```
C⟦ $v/article[2] ⟧ s u
= C⟦$v/article⟧ s (λx.λp. if p=2 then u x p else x)
= <state><v>{
      for $z at $i in s/v/bib/article
      return if $i=2
             then u $z $i else $z }</v></state>
```

FLWOR expressions with 'at' bindings are translated as follows:

$$\begin{aligned}&\mathcal{C}[\![\textbf{for}\ \$v\ \textbf{at}\ \$i\ \textbf{in}\ e_1\ \textbf{where}\ pred\ \textbf{return}\ e_2]\!]\ \sigma\ s\ u \qquad (\text{for'})\\ &=\ \mathcal{C}[\![e_2]\!]\ \sigma'\ s\ u\end{aligned}$$

where $\sigma' = \sigma\textbf{[}\$v/f_v, \$i/f_i\textbf{]}$ so that

$$\begin{aligned}&f_v = \lambda s.\lambda u.\ \mathcal{C}[\![e_1]\!]\ \sigma\ s\ (\lambda x.\lambda p.\ \text{subst}\ \$v\ x\ (\textbf{if}\ \mathcal{E}[\![pred]\!]\ x\ p\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ u\ x\ p\ \textbf{else}\ x))\\ &f_i = \lambda s.\lambda u.\ \mathcal{C}[\![e_1\textbf{[}pred'\textbf{]}]\!]\ \sigma\ s\ (\lambda x.\lambda p.\ p)\\ &\text{where}\ pred' = \text{subst}\ \$v\ (.)\ pred\end{aligned}$$

5. Addressing Interference Caused by Aliasing

As explained in Section 2, in order for the translations to be correct, our framework requires that there is no interference among the nodes of a mutable state. This is a very serious limitation and is addressed in this section. Consider for example the following QUERY3:

```
declare variable $x := doc('bib.xml');
declare variable $y := <A>{$x/article[author='Smith'],
                           doc('a.xml')/article}</A>;
( insert node <article>{...}</article> into $x,
  for $a in $y/article[contains(title,'XQuery')]
  return replace value of node $a/year with 2009 )
```

The state of this query consists of the two variables $x and $y, since they are both used in update destinations. But many nodes in $y are copies of nodes in $x (they have the same identity) and, therefore, updates to the former nodes must be reflected to the latter, and vice versa. This cannot be done if we use a state that has two detached components, one for each variable.

Our approach to address interference is to statically factor out the largest sub-expressions of a mutable variable binding which return nodes that overlap with the nodes of other mutable variables. That is, a mutable variable declaration $\$v := e$ is transformed to $\$v := f(e_1, \ldots, e_n)$, where each e_i is an XQuery expression that extracts nodes from some other mutable variable v_i used in the query but does not have new element constructions, and f does not access nodes from other mutable variables. We call these sub-expressions e_i, *aliased expressions*. For our example, $\$y := f(\$x/article[author='Smith'])$, where $f(z) =$ <A>{ z, doc('a.xml')/article }</A>. Then, the initial state consists of the mutable variable values but with empty factored out values: $f((), \ldots, ())$. That is, the state for our example query is:

```
<state><x>{doc('bib.xml')}</x>
       <y><A>{ (), doc('a.xml')/article }</A></y>
</state>
```

Note that this state does not include nodes with the same identity anymore. For each aliased expression e_i of type t_i that depends on the mutable variable v_i, we create a new named type $T_i = t_i$ with a unique name T_i. The XQuery type inference algorithm does not need to be changed, since the XQuery type system already supports named types. The translation algorithm $\mathcal{C}[\![e]\!]\,\sigma$ though must include an extra environment parameter δ that binds these named types to functions similar to those used in variable bindings. More specifically, T_i, which is associated to the aliased expression e_i, is bound to $\mathcal{C}[\![e_i]\!]\,\sigma\,\delta$. The translation algorithm is extended with the following rule:

$$e : T_i \;\Rightarrow\; \mathcal{C}[\![e]\!]\,\sigma\,\delta\,s\,u \;=\; \delta[\![T_i]\!]\,s\,u \qquad \text{(alias)}$$

That is, when the algorithm reaches an expression that retrieves aliased nodes (which must be of type T_i), it retrieves these nodes from the primary copy, $\mathcal{C}[\![e_i]\!]\,\sigma\,\delta$, since the replicas are not part of the state any more. The path translation algorithm must also be extended with the following rule:

$$\mathcal{P}[\![T_i]\!]\,x\,\text{step}\,n\,u \;=\; \delta[\![T_i]\!]\,s\,(\lambda z.\,\mathcal{P}[\![t_i]\!]\,z\;\text{step}\,n\,u)$$

(the current state s must be added as parameter to $\mathcal{P}[\![t]\!]$). For QUERY3, we introduce one named type T = element article {...}, which is bound in δ to:

$\mathcal{C}[\![\$v/article[author='Smith']]\!]\,\sigma\,\delta$

which, based on the example in Section 3.4, is equal to

```
λs.λu. <state><x>{ for $z in s/x/bib/article
                   return if $z/author='Smith'
                          then u($z) else $z }</x>
               <y>{s/y/node()}</y></state>
```

Based on Rule (alias), expression $w/article is translated to

$\mathcal{P}[\![$element A {T, t'}$]\!]$ (s/y) (child::article) 0 u

where t' is the type of doc('a.xml')/article. This is reduced by Rules (p-step) and (p-seq) to:

($\mathcal{P}[\![$T$]\!]$ (s/y/self::A/node()) (child::article) 1 u, ...)

The first sequence component, based on our new rules, is equal to

```
<state><x>{ for $z in s/x/bib/article
            return if $z/author='Smith'
                   then u'($z) else $z }</x>
       <y>{s/y/node()}</y></state>
```

where u' is equal to

λz. $\mathcal{P}[\![$element article {...}$]\!]$ z (child::article) 1 u
= λz. u (z/self::article)

based on Rule (p-child). Thus, the update context, u, is now applied to the x component of the state.

The only problem that remains to be addressed is factoring out the maximal sub-expressions that replicate nodes from a mutable variable and assigning it a new type name. This can be done with the help of the type inference system. These named types are treated specially by the type system: when a mutable variable is found in the expression of a variable declaration, it is assigned a new type name T. A path expressions, such as e/axis::test, has type T if e has type T (the special type name). Similarly, $e_1[e_2]$ has type T if e_1 has type T. These are unconventional rules that allow us to detect the maximal expression of type T. For constructions, such as $<A>\{e\}</A>$, of course, we can use the standard typing rules, since these constructions do not duplicate nodes.

6. Reordering the Pending Updates

The XUF semantics requires that the primitive updates in the pending list of an XUF query be reordered in the following order [18]: *"First, all insertInto, insertAttributes, replaceValue, and rename primitives are applied. Next, all insertBefore, insertAfter, insertIntoAsFirst, and insertIntoAsLast primitives are applied. Next, all replaceNode primitives are applied. Next, all replaceElementContent primitives are applied. Next, all delete primitives are applied."* Since there is no explanation given, presumably, the authors' intention was to make XUF side-effects easier to predict. Unfortunately, this reordering complicates many other aspects of semantics: it makes unnecessarily hard to type-check updates (the type-checking rules would not be compositional but would depend on context) and to optimize queries. In this section, we extend our framework to match these requirements.

To illustrate our method, suppose that we have two groups of updates only: insertion and non-insertion updates, so that insertions must be done before the rest. (This method can be generalized to multiple groups, such as those required by XUF semantics.) We develop two versions of the $\mathcal{C}$ algorithm: $\mathcal{C}_1$ that compiles insertion updates in the same way as $\mathcal{C}$ but compiles non-insertions to identity transformations that leave the state unchanged, and $\mathcal{C}_2$ that does the opposite. Then, this ordering of groups can be done using $\mathcal{C}_2[\![e]\!]\,\sigma\,(\mathcal{C}_1[\![e]\!]\,\sigma\,s\,u)\,u$, for a query e. That is, we synthesize a new state that reflects the insertion updates only and then we transform this state to reflect the rest of the updates. One may think that this method could result to inefficient code since we evaluate the query twice, but, after query normalization, the FLWOR loops of these transformations would be fused, yielding a normalized program in which FLWOR domains are simple XPath queries (see Section 7).

7. XQuery Normalization

Since our translation rules are compositional, they generate layers of state transformations, where each layer traverses its input state and constructs a new state, which is immediately consumed by the next layer, which in turn constructs a new state, and so on. We would like to fuse these layers into a single XQuery that does not produce these intermediate states. This can be done with the help

of normalization rules, such as:

$(<A>\{e\}</A>)/\text{child}::B \;=\; e/\text{self}::B$
$(<A>\{e\}</A>)/\text{self}::A \;=\; <A>\{e\}</A>$
$(<A>\{e\}</A>)/\text{node}() \;=\; e$
$e/\text{node}()/\text{self}::A \;=\; e/\text{child}::A$
$(e_1, e_2)/\text{axis::test} \;=\; (e_1/\text{axis::test},\ e_2/\text{axis::test})$
for $\$v$ **in** (e_1, e_2) **return** e_3
$\quad =\; (\textbf{for}\ \$v\ \textbf{in}\ e_1\ \textbf{return}\ e_3,\ \textbf{for}\ \$v\ \textbf{in}\ e_2\ \textbf{return}\ e_3)$
for $\$v$ **in** (**for** $\$w$ **in** e_1 **return** e_2) **return** e_3
$\quad =\;$ **for** $\$w$ **in** e_1 **return for** $\$v$ **in** e_2 **return** e_3

to partial evaluate operations on constructions and sequences, and eventually eliminate some of their components. In contrast to scripting languages, XQuery is amenable to optimization since it is purely functional. Such optimizations have already been addressed by related work [10, 16] and have also been described in our earlier work [7]. Normalization has also been used to translate XQuery to XQuery Core [17].

8. Implementation and Performance Evaluation

All the algorithms described in this paper, except of the treatment of aliasing, have already been implemented in Haskell and the presented examples have been tested. The source code is available at `http://lambda.uta.edu/Updates.hs`. The actual translation algorithm is only 120 lines of self-contained Haskell code and covers a substantial subset of XUF. This code has yet to be incorporated into our XQuery database management system, HXQ [9], which is written in Haskell.

The run-time of our translation algorithm is linear to the query size, since it is a recursive tree algorithm over the abstract syntax tree of the XUF code. On the other hand, query normalization can be exponential and may result to an exponential blowup of the query code, as is evident from the normalization of **let** $\$v :=$ e **return** $(\$v, \$v)$ into (e, e).

The default XUF implementation that preserves the XUF snapshot semantics must extract and accumulate all update primitives from an XUF query into a pending list before it applies them to the XML data. Each pending update may contain the kind of the update, a pointer to the update destination (an XML node), and the update source (an XML sequence). Using its destination pointer, the application of a pending update can be completed in constant time. On the other hand, if the update semantics were eager, then the implementation would not have required the extra space for keeping the pending update list, while the update sources would have been available for garbage collection earlier. The question we address now is how the default implementation of an XUF query compares with its XQuery translation generated by our translation algorithm. This is a hard question to answer because our translation framework is an enabling technology since it makes possible other transformations, such as XUF optimization, and can be fairly evaluated in combination with the transformations it enables. But let us assume that the XUF query is already optimal. Is the default XUF implementation faster than our XQuery translation? Obviously yes, but the overhead is not substantial. The actual evaluation of both implementations traverse the same number of XML nodes but our generated code constructs new XML elements that were not part of the original XML data (although their node identity is identical to existing nodes). The original XML nodes that these new constructions replace must be garbage-collected, a task that imposes additional run-time overhead (although modern automatic garbage collectors are very fast). Consider, for example, the translation of QUERY1. The new bib, article, and year element constructions introduce space and time overheads. Assume, for example, that there are 1000 articles with an average of 2 authors per article, from which 10 articles qualify in QUERY1 (all articles published by Smith whose title contains the word XQuery). The translation of QUERY1, given in the beginning of Section 2, generates extra element and sequence cells, calculated in the following table:

	data element cells	data sequence cells	extra element cells	extra sequence cells
bib	1	1000	1	1000
article	1000	4000	10	40
title	1000	0	0	0
year	1000	0	10	0
author	2000	0	0	0
total	5001	5000	21	1040

That is, there is a space overhead of 21/5001=0.42% for element construction and 1040/5000=20.8% for sequence construction, a total of 1061/10000=10.6% space overhead. These extra cells replace old cells that can be garbage-collected at the end of the XUF query evaluation.

9. Schema-Based XUF Optimization

As is evident from the previous section, our framework translates XUF updates into efficient XQueries, and, thus, there is no need for any further processing. But if the targets of these XUF updates are persistent XML data, one may want to translate these updates to optimized XUF updates that destructively modify the XML database. This task can be done if we translate the generated XQuery that transforms the XML state to an XUF query that updates the state. That is, we can translate the original XUF update to an XQuery using our translation algorithm, optimize it using standard XQuery optimization, and finally translate the resulting XQuery back to XUF. The latter task, translating XQuery to XUF, may seem far harder than translating XUF to XQuery, but the update generation algorithm does not have to be complete. In the worst case, we can always generate a replace update that replaces the entire database with the new. But there is a simple heuristic that results to more precise updates. When we recognize that a part of the state is propagated to the new state as is, then we discard the code that generates this part; otherwise we brake it further into subparts and apply this method recursively until we have no choice but to generate an XUF update. One heuristic to recognize those parts that remain unchanged is to compare the state transformation code with the copy function that creates an exact copy of the state, by traversing and reconstructing all nodes of the state. This is a conservative approach, which, in the worst case, may generate unnecessary (although correct) updates. This copy function can be easily generated from the state schema. The comparison between the state transformation and the copy function can only be effective if the state transformation code is normalized, as described in Section 7, so that for-loops that generate the new state constructions are over the same paths as the corresponding paths in the copy function (modulo variable renaming). This algorithm is described in detail in our work on incremental view maintenance [8].

10. Related Work

Although there are already a number of proposals for XQuery update languages [1, 3, 4, 14], there is now a W3C candidate recommendation, called XQuery Update Facility (XUF) [18]. The work by Fan *et al* [6] implements XQuery transform queries without requiring any change to existing XQuery processors. It uses (among others) a naive method that evaluates XQuery updates using pure XQuery recursive functions. Contrary to our type-guided update translations, recursive functions offer very few opportunities for optimization. The work by Cheney [4] introduces an XML update language similar to XUF, called FLUX, and gives it operational se-

mantics and a sound, decidable static type system. In contrast to XUF, FLUX does not allow aliasing of the mutable store, and thus it avoids the interference problem. The work by Ghelli *et al* [15] gives denotational semantics to XQuery updates, by extending the XQuery data model with stores and store histories, and uses it for a commutativity analysis. The standard XUF semantics requires that update primitives be accumulated in a pending list before they are applied. A naive implementation of this semantics, used by current XQuery processors, can be very inefficient. To improve performance, the work by Benedikt *et al* [2] uses interleaved semantics to pipeline both reads and writes when the update query is 'binding independent', that is, if there is no interference between reads and writes. The work by Foster *et al* [11] on XQuery view maintenance has similar goals as ours but uses the Galax algebra for the update mappings. Unlike these related approaches that use framework-specific algebras, our work can be incorporated into any existing XQuery engine since it translates XUF queries to plain XQuery.

11. Conclusion

The effectiveness of our translation framework can be attributed to the extensive use of type information. Although state transformations can also be generated from untyped XUF queries (using recursive functions), the quality of the generated code would be so low that would offer very few opportunities for optimization. Another characteristic of our approach that makes it different from related work is that it captures node identity explicitly, allowing us to propagate it through state transformations easily, thus satisfying the XUF semantics that expects identity preservation through updates.

References

[1] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Adding Updates to XQuery: Semantics, Optimization, and Static Analysis. In *XIME-P'05*.

[2] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of Tree Updates for Optimization. In *CAV'05*.

[3] J. Cheney. Lux: A Lightweight, Statically Typed XML Update Language. In *PLAN-X'07*.

[4] J. Cheney. FLUX: FunctionaL Updates for XML. In *ICFP'08*.

[5] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views. In *WIDM'02*.

[6] W. Fan, G. Cong, and P. Bohannon. Querying XML with Update Syntax. In *SIGMOD'08*.

[7] L. Fegaras. Propagating Updates through XML Views using Lineage Tracing. In *ICDE'10*. Available at `http://lambda.uta.edu/updates09.pdf`.

[8] L. Fegaras. Incremental Maintenance of Materialized XML Views. Submitted to a conference. Available at `http://lambda.uta.edu/views10.pdf`.

[9] L. Fegaras. HXQ: A Compiler from XQuery to Haskell. `http://hackage.haskell.org/package/HXQ`, 2010.

[10] M. Fernandez, J. Simeon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: the Galax experience. In *VLDB'03*.

[11] J. N. Foster, R. Konuru, J. Simeon, and L. Villard. An Algebraic Approach to XQuery View Maintenance. In *PLAN-X'08*.

[12] G. Fourny, M. Pilman, D. Florescu, D. Kossmann, T. Kraska, and D. McBeath. XQuery in the Browser. In *WWW'09*.

[13] G. Ghelli, N. Onose, K. Rose, and J. Simeon. XML Query Optimization in the Presence of Side Effects. In *SIGMOD'08*.

[14] G. Ghelli, C. Re, and J. Simeon. XQuery!: An XML Query Language with Side Effects. In *EDBT'06*.

[15] G. Ghelli, K. Rose, and J. Simeon. Commutativity analysis for XML updates. In *TODS'08*, 33(4).

[16] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. In *TODS'06*, 31(3).

[17] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/xquery-semantics/`, 2007.

[18] W3C. XQuery Update Facility 1.0. W3C Candidate Recommendation 1. `http://www.w3.org/TR/xquery-update-10/`, June 2009.

A. Correctness Proof

In this Appendix, we give a correctness proof sketch for our XUF translation algorithm. More specifically, we prove that the XQuery expressions composed by our translation algorithm modify the XML data in the way described by the XUF update semantics. In the standard XUF semantics [18], an XUF update results to a pending list of update primitives, which is reordered before is evaluated, so that insertions are done before updates and deletes. Our translation algorithm is equivalent to this pending list production but without the reordering. The reordering is addressed in Section 6, where multiple state transformers, corresponding to multiple pending lists, are used.

Our proof sketch considers five kinds of XUF queries only: variables, let-bindings, for-loops without predicates, sequence concatenation, and replace-node updates. Other kinds of expressions can be treated in the same way. An interpreter that evaluates an XUF query e, before the application of the pending updates, is $\mathcal{I}[\![e]\!]\,\rho$, where ρ is a binding list that maps XUF variables to XML node sequences. The value domain of the interpreter is an XML node sequence (denoted by T^*, where T is the type of XML node). These sequences are constructed using lists operations, $[\,]$ for construction and $+\!\!+$ for concatenation. The interpreter can be described by the following rules:

$$\mathcal{I}[\![\$v]\!]\,\rho \;=\; \rho\mathbf{[}\$v\mathbf{]} \tag{i-1}$$

$$\mathcal{I}[\![\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2]\!]\,\rho \;=\; \mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/(\mathcal{I}[\![e_1]\!]\,\rho)\mathbf{]} \tag{i-2}$$

$$\mathcal{I}[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\,\rho \;=\; \mathrm{concatMap}\,(\lambda z.\,\mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/[z]\mathbf{]})\,(\mathcal{I}[\![e_1]\!]\,\rho) \tag{i-3}$$

$$\mathcal{I}[\![(e_1, e_2)]\!]\,\rho \;=\; (\mathcal{I}[\![e_1]\!]\,\rho)+\!\!+(\mathcal{I}[\![e_2]\!]\,\rho) \tag{i-4}$$

$$\mathcal{I}[\![\mathbf{replace\ node}\ e_1\ \mathbf{with}\ e_2]\!]\,\rho \;=\; [\,] \tag{i-5}$$

where concatMap $f\,[x_1, \ldots, x_n] = f(x_1)+\!\!+\cdots+\!\!+f(x_n)$. Rule (i-3) evaluates the for-loop body e_2 under a new environment $\rho\mathbf{[}\$v/[z]\mathbf{]}$, which is ρ extended with the binding from the variable $\$v$ to the node sequence $[z]$, where z is an element of the sequence from the evaluation of e_1. The value of $\$v$ in ρ is extracted using $\rho\mathbf{[}\$v\mathbf{]}$ (Rule (i-1)).

In addition to its evaluation, an XUF query causes side-effects in the form of update primitives that are collected in a pending list.

Definition 1 (Update Primitive)**.** *An update primitive is a mapping $v \mapsto u$ from a node sequence $v : T^*$ to a context mapping $u : T \to T^*$.*

The update primitive $v \mapsto u$, when applied to an XML node x, returns a copy of x but with every descendant node of x whose node identity is equal to the node identity of one of the v nodes replaced by the result of applying the context mapping u to this node. More specifically, the following function applies an update primite to a node x:

$$\begin{array}{l}\mathrm{upd}\ (v \mapsto u)\ x\\ =\ \mathbf{if}\ x \equiv v\ \mathbf{then}\ u\,x\\ \quad\ \mathbf{else\ if}\ \exists A, s:\ x = \mathrm{element}\ A\ \ s\\ \qquad\quad \mathbf{then}\ \mathrm{element}\ A\ (\mathrm{map}\ (\lambda z.\,\mathrm{upd}\ (v \mapsto u)\ z)\ s)\\ \qquad\quad \mathbf{else}\ x\end{array}$$

where $\mathrm{element}\ \ A\ \ s$ is an XML element construction with tag A and content s (a node sequence), $x \equiv v$ is true if x has the same node identity as any of the nodes in v, and map $f\,[x_1, \ldots, x_n] = [f(x_1), \ldots, f(x_n)]$. The upd function traverses the XML tree x recursively; if it reaches a node x whose node identity is from the update primitive domain v, then it replaces x with (ux); otherwise, if x is an XML element construction, it applies upd recursively to the element content s; otherwise (if x is a base value), it returns x.

The upd function can be overloaded to work on node sequences: $\mathrm{upd}\ (v \mapsto u)\ s = \mathrm{map}\ (\lambda z.\,\mathrm{upd}\ (v \mapsto u)\ z)\ s$.

Definition 2 (Update List)**.** *An update list δ is a binding list of update primitives*

$$\delta = \mathbf{[}v_1 \mapsto u_1, \ldots, v_n \mapsto u_n\mathbf{]}$$

As for a single update primitive, $(\mathrm{upd}\ \ \delta\ \ x)$ applies the update primitives in δ to x from left to right:

$$\mathrm{upd}\ \delta\ x \;=\; \mathrm{foldl}\ \mathrm{upd}\ x\ \delta$$

where foldl $f\ z\ [x_1, x_2, \ldots, x_n] = f\,x_n\ \ldots\ (f\,x_2\,(f\,x_1\,z))$. That is, $\mathrm{upd}\ \delta\ x = \mathrm{upd}\ (v_n \mapsto u_n)\ (\cdots \mathrm{upd}\ (v_2 \mapsto u_2)\ (\mathrm{upd}\ (v_1 \mapsto u_1)\ x))$. It satisfies the law:

$$\mathrm{upd}\ \delta_2\ (\mathrm{upd}\ \delta_1\ x) = \mathrm{upd}\ (\delta_1 +\!\!+ \delta_2)\ x \tag{p-1}$$

Based on the XUF semantics, an XUF query produces a pending list of updates that accumulates the update primitives in the query:

$$\Delta[\![\$v]\!]\,\rho = \mathbf{[\,]} \tag{d-1}$$

$$\Delta[\![\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2]\!]\,\rho \;=\; \Delta[\![e_2]\!]\ \rho\mathbf{[}\$v/(\mathcal{I}[\![e_1]\!]\,\rho)\mathbf{]} \tag{d-2}$$

$$\Delta[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\,\rho \;=\; \mathrm{concatMap}\,(\lambda z.\,\Delta[\![e_2]\!]\ \rho\mathbf{[}\$v/[z]\mathbf{]})\,(\mathcal{I}[\![e_1]\!]\,\rho) \tag{d-3}$$

$$\Delta[\![(e_1, e_2)]\!]\,\rho \;=\; (\Delta[\![e_1]\!]\,\rho)+\!\!+(\Delta[\![e_2]\!]\,\rho) \tag{d-4}$$

$$\Delta[\![\mathbf{replace\ node}\ e_1\ \mathbf{with}\ e_2]\!]\,\rho \;=\; \mathbf{[}x_1 \mapsto \lambda z.\,x_2\mathbf{]} \quad \text{where}\ \ x_1 = \mathcal{I}[\![e_1]\!]\,\rho\ \ \text{and}\ \ x_2 = \mathcal{I}[\![e_2]\!]\,\rho \tag{d-5}$$

For the replace update, equation (d-5) binds x_1/x_2 to the evaluation of the update destination/source and returns a single update primitive that binds the replace destination node x_1 to the context mapping function $\lambda z.\,x_2$, which ignores the old destination node x and replaces it with the x_2 (the source node sequence).

Based on the XUF semantics, an XUF query Q is evaluated to $\mathcal{I}[\![Q]\!]\,\mathbf{[\,]}$ and generates a pending list of updates $\delta = \Delta[\![Q]\!]\,\mathbf{[\,]}$ such that the the mutable state s (all the updatable XML data) is updated to $(\mathrm{upd}\ \delta\ s)$.

Now we focus on our translation algorithms. Let S be the type of the mutable state (an XML node type T with a predefined type).

Definition 3 (S-Complement)**.** *Given a state of type S, an update list δ, and a node sequence v of type T^*, the S-complement of v is a function $\langle v\rangle_S^\delta$ of type $S \to (T \to T^*) \to S$ such that $\forall s : S$ and $\forall u : T \to T^*$:*

$$\langle v\rangle_S^\delta\ s\ u = \mathrm{upd}\ s\ \delta\mathbf{[}v \mapsto u\mathbf{]}.$$

That is, if v is a sequence of nodes $(v_1, \ldots, v_n)$, then the S-complement of v is the function f so that $(f\,s\,u)$ returns a copy of the state s in which the nodes that have the same node identity as v_i have been replaced with $(u\,v_i)$. It is easy to prove the following properties, for all v, δ, s, and u:

$$\langle v\rangle_S^\delta\ s\ \mathbf{id} \;=\; \mathrm{upd}\ s\ \delta \tag{p-2}$$

$$\langle ()\rangle_S^\delta\ s\,u \;=\; \mathrm{upd}\ s\ \delta \tag{p-3}$$

$$\langle v\rangle_S^{\delta_2}\ (\langle v\rangle_S^{\delta_1}\ s\,u)\,u \;=\; \langle v\rangle_S^{\delta_1 +\!\!+ \delta_2}\ s\,u \tag{p-4}$$

Our original translation algorithm $\mathcal{C}$ described in Section 3 generates plain XQuery code that returns a new XML state that reflects the XUF updates. We first present a variation of our translation algorithm, which actually calculates the state directly, rather than generating an XQuery that returns the state. It is summarized by the following rules, which are taken from Rules (subst), (let), (for)

without a predicate, (seq), and (repl):

$$\mathcal{S}[\![\$v]\!]\,\sigma\,\rho\,s\,u = \sigma\mathbf{[}\$v\mathbf{]}\,s\,u \quad \text{(s-1)}$$

$$\mathcal{S}[\![\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2]\!]\,\sigma\,\rho\,s\,u \quad \text{(s-2)}$$
$$= \mathcal{S}[\![e_2]\!]\,\sigma\mathbf{[}\$v/(\mathcal{S}[\![e_1]\!]\,\sigma\,\rho)\mathbf{]}\,\rho\mathbf{[}\$v/(\mathcal{I}[\![e_1]\!]\,\rho)\mathbf{]}\,s\,u$$

$$\mathcal{S}[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\,\sigma\,\rho\,s\,u \quad \text{(s-3)}$$
$$= \text{foldl}\,(\lambda r.\,\lambda z.\,\mathcal{S}[\![e_2]\!]\,\sigma\mathbf{[}\$v/(f\,z)\mathbf{]}\,\rho\mathbf{[}\$v/[z]\mathbf{]}\,r\,u)\,s\,(\mathcal{I}[\![e_1]\!]\,\rho)$$
where $f\,z\,s\,u = \mathcal{S}[\![e_1]\!]\,\sigma\,\rho\,s\,(\lambda x.\ \mathbf{if}\ x \equiv z\ \mathbf{then}\ u\,x\ \mathbf{else}\ x)$

$$\mathcal{S}[\![(e_1, e_2)]\!]\,\sigma\,\rho\,s\,u = \mathcal{S}[\![e_2]\!]\,\sigma\,\rho\,(\mathcal{S}[\![e_1]\!]\,\sigma\,\rho\,s\,u)\,u \quad \text{(s-4)}$$

$$\mathcal{S}[\![\mathbf{replace\ node}\ e_1\ \mathbf{with}\ e_2]\!]\,\sigma\,\rho\,s\,u \quad \text{(s-5)}$$
$$= \mathcal{S}[\![e_1]\!]\,\sigma\,\rho\,s\,(\lambda x.\,u\,(\mathcal{I}[\![e_2]\!]\,\rho))$$

Note that Rule (s-3) is different from Rule (for). Later, we will prove that $\mathcal{C}$ generates code that calculates the same result as $\mathcal{S}$. We first prove that $\mathcal{S}$ calculates the S-complement.

Lemma 1.

$$\forall v \in \rho:\ \sigma\mathbf{[}\$v\mathbf{]} = \langle\rho\mathbf{[}\$v\mathbf{]}\rangle_S^{\mathbf{[]}} \ \Rightarrow\ \mathcal{S}[\![e]\!]\,\sigma\,\rho = \langle\mathcal{I}[\![e]\!]\,\rho\rangle_S^{\delta}$$

where $\delta = \Delta[\![e]\!]\,\rho$.

That is, given that the variable bindings used by $\mathcal{S}$ are S-complements of the associated bindings used by the interpreter $\mathcal{I}$, then the $\mathcal{S}$ translation of an XUF expression e is the S-complement of the $\mathcal{I}$ interpretation of e.
Proof: We prove this lemma using structural induction over e:

- If $e = \$v$, then from (d-1), $\delta = \Delta[\![\$v]\!]\,\rho = \mathbf{[\,]}$. From (s-1), the premise, and (i-1) we have $\mathcal{S}[\![\$v]\!]\,\sigma\,\rho = \sigma\mathbf{[}\$v\mathbf{]} = \langle\rho\mathbf{[}\$v\mathbf{]}\rangle_S^{\delta} = \langle\mathcal{I}[\![\$v]\!]\,\rho\rangle_S^{\delta}$.
- If $e = \mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2$ and $\delta = \Delta[\![e_2]\!]\rho\mathbf{[}\$v/(\mathcal{I}[\![e_1]\!]\,\rho)\mathbf{]}$, then we have:

  $\mathcal{S}[\![\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2]\!]\,\sigma\,\rho\,s\,u$
  $= \mathcal{S}[\![e_2]\!]\,\sigma\mathbf{[}\$v/(\mathcal{S}[\![e_1]\!]\,\sigma\,\rho)\mathbf{]}\,\rho\mathbf{[}\$v/(\mathcal{I}[\![e_1]\!]\,\rho)\mathbf{]}\,s\,u$ from (s-2)
  $= \langle\mathcal{I}[\![e_2]\!]\ \rho\mathbf{[}\$v/(\mathcal{I}[\![e_1]\!]\,\rho)\mathbf{]}\rangle_S^{\delta}$ (*)
  $= \langle\mathcal{I}[\![\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2]\!]\,\rho\rangle_S^{\delta}$ from (i-2)

  (*) from the induction hypothesis for e_2 because the premise $\sigma\mathbf{[}\$v\mathbf{]} = \mathcal{S}[\![e_1]\!]\,\sigma\,\rho = \langle\mathcal{I}[\![e_1]\!]\,\rho\rangle_S^{\mathbf{[]}} = \langle\rho\mathbf{[}\$v\mathbf{]}\rangle_S^{\mathbf{[]}}$ is true (induction hypothesis for e_1).
- For $e = \mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2$, let $\mathcal{I}[\![e_1]\!]\,\rho = [v_1, \ldots, v_n]$. The premise $f\,v_i = \langle v_i\rangle_S^{\mathbf{[]}}$ is true for all i, because from Rule (s-3) and the induction hypothesis (since $\Delta[\![e_1]\!]\rho = \mathbf{[\,]}$), we have:

  $$\begin{aligned} f\,v_i\,s\,u &= \mathcal{S}[\![e_1]\!]\,\sigma\,\rho\,s\,(\lambda x.\ \mathbf{if}\ x \equiv v_i\ \mathbf{then}\ u\,x\ \mathbf{else}\ x) \\ &= \langle\mathcal{I}[\![e_1]\!]\,\rho\rangle_S^{\mathbf{[]}}\,s\,(\lambda x.\ \mathbf{if}\ x \equiv v_i\ \mathbf{then}\ u\,x\ \mathbf{else}\ x) \\ &= \langle[v_1, \ldots, v_n]\rangle_S^{\mathbf{[]}}\,s\,(\lambda x.\ \mathbf{if}\ x \equiv v_i\ \mathbf{then}\ u\,x\ \mathbf{else}\ x) \\ &= \langle v_i\rangle_S^{\mathbf{[]}}\,s\,u \end{aligned}$$

  Since the premise is true, from the induction hypothesis and for $\delta_i = \Delta[\![e_2]\!]\rho\mathbf{[}\$v/[v_i]\mathbf{]}$, we conclude:

  $$\mathcal{S}[\![e_2]\!]\,\sigma\mathbf{[}\$v/(f\,v_i)\mathbf{]}\,\rho\mathbf{[}\$v/[v_i]\mathbf{]} = \langle\mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/[v_i]\mathbf{]}\rangle_S^{\delta_i}$$

  From (d-3), we have $\delta = \Delta[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\rho$
  $= \Delta[\![e_2]\!]\rho\mathbf{[}\$v/[v_1]\mathbf{]} +\!\!+ \cdots +\!\!+ \Delta[\![e_2]\!]\rho\mathbf{[}\$v/[v_n]\mathbf{]}$
  $= \delta_1 +\!\!+ \cdots +\!\!+ \delta_n$. Then, we have:

  $\mathcal{S}[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\,\sigma\,\rho\,s\,u$
  $= \mathcal{S}[\![e_2]\!]\,\sigma\mathbf{[}\$v/(f\,v_n)\mathbf{]}\,\rho\mathbf{[}\$v/[v_n]\mathbf{]}\ \ldots$
  $\quad (\mathcal{S}[\![e_2]\!]\,\sigma\mathbf{[}\$v/(f\,v_1)\mathbf{]}\,\rho\mathbf{[}\$v/[v_1]\mathbf{]}\,s\,u)\,u$ from (s-3)
  $= \langle\mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/[v_n]\mathbf{]}\rangle_S^{\delta_n}\ \ldots$
  $\quad (\langle\mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/[v_1]\mathbf{]}\rangle_S^{\delta_1}\,s\,u)\,u$ proved
  $= \langle\mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/[v_1]\mathbf{]} +\!\!+ \cdots$
  $\quad +\!\!+ \mathcal{I}[\![e_2]\!]\,\rho\mathbf{[}\$v/[v_n]\mathbf{]}\rangle_S^{\delta}\,s\,u$ from (p-4)
  $= \langle\mathcal{I}[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\,\rho\rangle_S^{\delta}\,s\,u$ from (i-3).

- If $e = (e_1, e_2)$, then from (d-4), $\delta = \Delta[\![(e_1, e_2)]\!]\,\rho = \delta_1 +\!\!+ \delta_2$, where $\delta_1 = \Delta[\![e_1]\!]\,\rho$ and $\delta_2 = \Delta[\![e_2]\!]\,\rho$. Then:

  $\mathcal{S}[\![(e_1, e_2)]\!]\,\sigma\,\rho\,s\,u$
  $= \mathcal{S}[\![e_2]\!]\,\sigma\,\rho\,(\mathcal{S}[\![e_1]\!]\,\sigma\,\rho\,s\,u)\,u$ from (s-4)
  $= \langle\mathcal{I}[\![e_2]\!]\,\rho\rangle_S^{\delta_2}\,(\langle\mathcal{I}[\![e_1]\!]\,\rho\rangle_S^{\delta_1}\,s\,u)\,u$ hypotheses
  $= \text{upd}\,(\text{upd}\,s\,\delta_1\mathbf{[}\mathcal{I}[\![e_1]\!]\,\rho \mapsto u\mathbf{]})\,\delta_2\mathbf{[}\mathcal{I}[\![e_2]\!]\,\rho \mapsto u\mathbf{]}$ from Def. 3
  $= \text{upd}\,s\,(\delta_1\mathbf{[}\mathcal{I}[\![e_1]\!]\,\rho \mapsto u\mathbf{]} +\!\!+ \delta_2\mathbf{[}\mathcal{I}[\![e_2]\!]\,\rho \mapsto u\mathbf{]})$ from (p-1)
  $= \text{upd}\,s\,\delta\mathbf{[}(\mathcal{I}[\![e_1]\!]\,\rho +\!\!+ \mathcal{I}[\![e_2]\!]\,\rho) \mapsto u\mathbf{]}$ (*)
  $= \langle\mathcal{I}[\![e_1]\!]\,\rho +\!\!+ \mathcal{I}[\![e_2]\!]\,\rho\rangle_S^{\delta}\,s\,u$ from Def. 3
  $= \langle\mathcal{I}[\![(e_1, e_2)]\!]\,\rho\rangle_S^{\delta}\,s\,u$ from (i-4).

  (*) because of the XUF restrictions, either $\delta = \mathbf{[\,]}$ or $u = \mathbf{id}$ (i.e., we cannot have updates in an update destination).
- If $e = \mathbf{replace\ node}\ e_1\ \mathbf{with}\ e_2$, then $\delta = \Delta[\![e]\!]\,\rho = \mathbf{[}\mathcal{I}[\![e_1]\!]\,\rho \mapsto \lambda z.\,\mathcal{I}[\![e_2]\!]\,\rho\mathbf{]}$ and $\Delta[\![e_1]\!] = \mathbf{[\,]}$ (no updates in the update destination). Since this update cannot appear inside another update, $u = \mathbf{id}$ and:

  $\mathcal{S}[\![\mathbf{replace\ node}\ e_1\ \mathbf{with}\ e_2]\!]\,\sigma\,\rho\,s\ \mathbf{id}$
  $= \mathcal{S}[\![e_1]\!]\,\sigma\,\rho\,s\,(\lambda x.\,\mathcal{I}[\![e_2]\!]\,\rho)$ from (s-5)
  $= \langle\mathcal{I}[\![e_1]\!]\,\rho\rangle_S^{\mathbf{[]}}\,s\,(\lambda x.\,\mathcal{I}[\![e_2]\!]\,\rho)$ hypothesis
  $= \text{upd}\,s\,\mathbf{[}\mathcal{I}[\![e_1]\!]\,\rho \mapsto \lambda x.\,\mathcal{I}[\![e_2]\!]\,\rho\mathbf{]}$ from Def. 3
  $= \langle()\rangle_S^{\delta}\,s\ \mathbf{id}$ from (p-3)
  $= \langle\mathcal{I}[\![\mathbf{replace\ node}\ e_1\ \mathbf{with}\ e_2]\!]\,\rho\rangle_S^{\delta}\,s\ \mathbf{id}$ from (i-5).

□

Lemma 2. *The $\mathcal{C}$ algorithm generates XQuery code that calculates $\mathcal{S}$. That is, $\forall e, \sigma, \rho, s, u$:*

$$\mathcal{I}[\![\mathcal{C}[\![e]\!]\,\sigma\,\rho\,s\,u]\!]\,\rho = \mathcal{S}[\![e]\!]\,\sigma\,\rho\,s\,u.$$

Proof: We need to prove that the Rules (subst), (let), (for) without a predicate, (seq), and (repl) produce code that calculates the results of Rules (s-1), (s-2), (s-3), (s-4), and (s-5), respectively. We can immediately see that Rules (subst), (let), (seq), and (repl) are exactly the same as Rules (s-1), (s-2), (s-4), and (s-5), respectively. Rule (s-3) though is different from Rule (for), which, without a predicate, becomes:

$$\mathcal{C}[\![\mathbf{for}\ \$v\ \mathbf{in}\ e_1\ \mathbf{return}\ e_2]\!]\,\sigma\,s\,u = \mathcal{C}[\![e_2]\!]\,\sigma\mathbf{[}\$v/g\mathbf{]}\,s\,u \quad \text{(for)}$$
where $g\,s\,u = \mathcal{C}[\![e_1]\!]\,\sigma\,s\,(\lambda x.\,\text{subst}\,\$v\,x\,(u\,x))$

which is exactly the same as $\mathcal{C}[\![\mathbf{let}\ \$v := e_1\ \mathbf{return}\ e_2]\!]\,\sigma\,s\,u$ (Rule (let)). We need to prove that Rule (s-3) is equivalent to Rule (for). We will first prove that $g = \langle v\rangle_S^{\mathbf{[]}}$, where $v = \mathcal{I}[\![e_1]\!]\,\rho\,s\,u$. Let $v = [v_1, \ldots, v_n]$. From Lemma 1 for (s-3), $f\,v_i = \langle[v_i]\rangle_S^{\mathbf{[]}} = \langle v_i\rangle_S^{\mathbf{[]}}$. Thus, $\langle v\rangle_S^{\mathbf{[]}}\,s\,u = \langle v_1 +\!\!+ \cdots +\!\!+ v_n\rangle_S^{\mathbf{[]}}\,s\,u =$

$f\ v_n\ (\ldots\ (f\ v_1\ s\ u))\ u$. But, from (s-3), we have:

$$
\begin{aligned}
&f\ v_n\ (\ldots\ (f\ v_1\ s\ u))\ u \\
&= \mathcal{S}[\![e_1]\!]\ \sigma\ \rho\ (\ldots\ (\mathcal{S}[\![e_1]\!]\ \sigma\ \rho\ s\ (\lambda x.\ \textbf{if}\ x \equiv v_1\ \textbf{then}\ u\ x\ \textbf{else}\ x)))\\
&\qquad (\lambda x.\ \textbf{if}\ x \equiv v_n\ \textbf{then}\ u\ x\ \textbf{else}\ x)\\
&= \mathcal{S}[\![e_1]\!]\ \sigma\ \rho\ s\ u = g\ s\ u \qquad\qquad \text{(modulo the substitutions)}
\end{aligned}
$$

From $g = \langle v \rangle_S^{\textbf{[ ]}}$, we have $\rho\textbf{[}\$v\textbf{]} = \langle \sigma\textbf{[}v\textbf{]} \rangle_S^{\textbf{[ ]}}$ for Lemma 1 for the case $e = \textbf{let}\ \$v\ :=\ e_1\ \textbf{return}\ e_2$, which proves that Rule (s-3) is equivalent to Rule (let), which is equal to Rule (for). □

Theorem 2. *The $\mathcal{C}$ and Δ state transformations of a top-level XUF query are equivalent.*

That is, the resulting state calculated by $\mathcal{C}[\![e]\!]\ \sigma\ s$ **id** is equivalent to the result of applying (without reordering) the pending update list of e to the state s.

Proof: Given a top-level XUF query Q and a mutable state s, let $\delta = \Delta[\![Q]\!]\ \textbf{[ ]}$. Then:

$$\mathcal{S}[\![Q]\!]\ \textbf{[ ]}\ \textbf{[ ]}\ s\ \textbf{id} = \langle \mathcal{I}[\![Q]\!]\ \textbf{[ ]} \rangle_S^{\delta}\ s\ \textbf{id} = \mathrm{upd}\ s\ \delta = \mathrm{upd}\ s\ (\Delta[\![Q]\!]\ \textbf{[ ]})$$

Since $\mathcal{C}$ generates code that calculates $\mathcal{S}$, the $\mathcal{C}$ state transformation of s is equivalent to that of Δ. □