# Incremental Maintenance of Materialized XML Views

Leonidas Fegaras
*University of Texas at Arlington, CSE*
*Arlington, TX 76019, USA*
fegaras@cse.uta.edu

## ABSTRACT

We investigate the problem of incremental maintenance of materialized XML views. We are considering the case where the underlying database is a relational database and the view exposed to querying is a materialized XML view. Then, updates to the underlying database should be reflected to the stored XML view, to keep it consistent with the source data, without recreating the entire view from the database after each source update. Unlike related work that uses algebraic methods, we use source-to-source compositional transformations, guided by the database and view schemata. We first translate SQL updates to pure (update-free) XQuery expressions that reconstruct the entire database state, reflecting the updated values in the new state. Then, we synthesize the right-inverse of the XQuery view function, guided by the view schema. This inverse function is applied to the old view to derive the old database state, which in turn is mapped to the new database state through the update function, and then is mapped to the new view through the view function. The resulting view-to-view function is normalized and translated to XQuery updates that destructively modify the materialized view efficiently to reflect the new view values.

## 1. INTRODUCTION

We address the problem of incremental maintenance of materialized XML views. That is, given a materialized XML view over a database and an update against the database, our goal is to generate view updates so that the new view after the updates is exactly the same as the view we would have gotten if we had applied the view to the updated source data. We are considering the case where the underlying database is a relational database and the view exposed to querying is a materialized XML view, stored in a native XML database. A native XML database is a specialized database that supports storage, indexing methods, and query processing techniques specially tailored to XML data. There are already many native XML database management systems available, including Oracle Berkeley-DB XML, Qizx, Natix, etc. Our framework is targeting those native XML databases that support the XQuery Update Facility (XUF) [21] for updating XML data. Exporting data through materialized XML views is very important to data integration, not only because it speeds up query processing, but also because it provides a light-weight interface to data services, without exposing the main database. The incremental maintenance of materialized views is important for applications that require freshness of view data, but it is too expensive to recompute the view data every time the source data change.

In our framework, SQL updates are translated to plain XQueries applied to a canonical view of the relational database. More specifically, we derive a pure update function that maps the old database to the new database. Then, we synthesize the right-inverse of the XQuery view function, guided by the view schema. This inverse function is applied to the old view to derive the old database state, which in turn is mapped to the new database state through the update function, and then mapped to the new view through the view function. Although correct, the resulting view-to-view function can be very inefficient. We provide methods to optimize this program and translate it into efficient XQuery updates, so that only the relevant parts of the materialized view are updated. More specifically, our framework performs the following tasks:

1. We map the underlying relational database $DB$ to the XML data $db$ using a canonical view of relational data. This wrapper, which is a virtual XML view, allows us to refer to relational data within the XQuery data model without any model extension.

2. We express the source SQL updates in pure (update-free) XQuery code over the virtual XML view, as a function $u$ from the old to the new database state that reconstructs the entire canonical XML view of the relational database, reflecting the updates.

3. Given a view, $view$, expressed in XQuery, we synthesize the XQuery expression $view'$ that is a right-inverse of $view$, such that $view(view'(V)) = V$ for any valid view instance $V$ (that is, $view$ must be a surjective function). Note that, $view'(view(DB))$ is not necessarily equal to $DB$; it is equal to $DB$ for isomorphic views only, which are uncommon.

4. The composition $F(V) = view(u(view'(V)))$ is a function from the old view $V$ to the new view (after the updates), where $view'(V)$ creates an entire source database from the view $V$, then $u$ reconstructs this database to reflect the updates, and finally $view$ maps the new database to the new view. We use normalization techniques to optimize $F(V)$ so that the resulting program avoids the reconstruction of most parts of the intermediate database data.

5. Finally, we rewrite the normalized XQuery expression $F(V)$ into efficient XQuery updates that destructively modify the original view $V$ using the XQuery Update Facility.

The key contribution of our work is in the development of a novel framework for the incremental maintenance of materialized XML views 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 [11, 13], our work can be incorporated into any existing XQuery engine.
- Since it breaks the task of view maintenance into a number of more manageable tasks that are easier to implement and verify their correctness, it has the potential of becoming a general methodology for incremental view maintenance.
- It can also be used for incremental view maintenance when both the underlying database and the view exposed to querying are XML databases. In an earlier work [8], we have described a powerful method for translating XUF updates to XQuery expressions that reconstruct the updated data. Thus, XUF updates to the underlying XML database are mapped to a pure XQuery $u$, which can be used to derive a view-to-view mapping $view(u(view'(V)))$, as it was done for SQL updates.
- It enables other transformations that are important to databases, such as translating updates over virtual XML views into SQL updates over the underlying relational database. Although this application is not addressed in this paper, one way to translate an XQuery view update to SQL is to convert it to a function $U$ from the old view to the new view that reconstructs the view reflecting the updates. Then, the composition $view'(U(view(DB)))$ reconstructs the relational database state to reflect the view updates. Mapping this composition back to SQL updates is a far more challenging task than mapping it to XQuery updates.

The most important limitation of our approach is that both the inverse synthesis (Step 3) and the XUF generation (Step 5) algorithms are schema-based. That is, they require that both the underlying relational and the XML view schemata be known at query translation time. The second limitation is that our algorithms are heuristics that work only in certain cases, such as they can handle one-to-one and one-to-many but they cannot handle many-to-many joins in the view definition. Finally, due to the lack of space, we have not covered some parts the XQuery/XUF syntax, such as descendant-or-self steps, but we believe that there is no fundamental reason why our framework cannot be extended to cover most of the XQuery syntax.

The rest of this paper is organized as follows. Section 2 compares our approach with related work. Section 3 overviews our approach through the use of one example. Section 4 presents rules for translating SQL updates to XQuery (Step 2). Section 5 describes our right-inverse synthesis algorithm (Step 3), whose validity is proved in Appendix A. Section 6 presents some normalization rules to fuse the layers of state transformations generated from our compositional rules (Step 4). Section 7 presents our algorithm for translating XQueries from view-to-view to XUF updates (Step 5) Finally, Section 8 reports on a prototype implementation and discusses efficiency issues,

## 2. RELATED WORK

Although the problem of incremental view maintenance has been extensively studied for relational views (see [12] for a literature survey), there is still little work on XML view maintenance [3, 1]. One of the earliest works on incremental maintenance of materialized views over semi-structured data was for the Lorel query language [2], which was based on the graph-based data model OEM. Their method produces a set of queries that compute the updates to the view based upon an update of the source. Although there are already a number of proposals for XQuery update languages, there is now a W3C candidate recommendation, called XQuery Update Facility (XUF) [21]. El-Sayed *et al* [13] use an algebraic approach for incremental XQuery view maintenance, where an update to the XML source is transformed into a set of update primitives that are propagated through the XML algebra tree and become incremental update primitives to be applied to the result view. The work by Sawires *et al* [17] considers the view maintenance problem for simple XPath views (without value-based predicates). It reduces the frequency of view recomputation by detecting cases where a source update is irrelevant to a view and cases where a view is self-maintainable given a base update. BEA's AquaLogic Data Services Platform allows developers to specify inverse functions that enable its XQuery optimizer to push predicates and perform updates on data services that involve the use of such functions [16].

The problem of bidirectionalization has been recently studied by the functional programming community [18, 14]. Given a function (a view) and updates to the function result, bidirectionalization will map these updates back to the function input. A pair of two partial functions, a 'get' function, which corresponds to the view function, and a 'put' function which is an update strategy, is generally known as a lens. Instead of deriving lenses automatically, the work on Relational Lenses [5] provides a bidirectional query language, based on standard relational operators, in which every expression can be read both from left to right as a view definition and from right to left as an update policy.

## 3. OVERVIEW

In this section, we use an XML view example to describe our approach to the incremental maintenance of XML views. Consider a bibliography database described by the following relational schema:

Article ( aid, title, year )
Author ( aid, pid )
Person ( pid, name, email )

where the attributes Author.aid and Author.pid are foreign keys that reference the keys Article.aid and Person.pid, respectively. An XML view over a relational database can be defined over a canonical XML view of relational data, as is done in SilkRoute [9] and many other systems. The canonical XML view of the bibliography relational schema has the following XML type:

```
element DB {
  element Article {
    element row {
      element aid { xs: int }, element title { xs: string },
      element year { xs: string } }* },
  element Author {
    element row {
      element aid { xs: int }, element pid { xs: int } }* },
  element Person {
    element row {
      element pid { xs: int }, element name { xs: string },
      element email { xs: string } }* } }
```

that is, a relational table is mapped to an XML element whose tag name is the table name and its children are row elements that correspond to the table tuples. In addition, this type satisfies the following key constraints:

```
key { /DB/Article/row, aid/data() }
key { /DB/Author/row, (aid,pid)/data() }
key { /DB/Person/row, pid/data() }
```

As in XML-Schema [19], for a key { $p_1$, $p_2$ }, the selector path $p_1$ defines the applicable elements and the field path $p_2$ defines the data that are unique across the applicable elements. For example, key { /DB/Article/row, aid/data() } indicates that the aid elements are unique for the Article rows.

Given this canonical view $DB of the relational database, an XML view, view($DB), can be defined using plain XQuery code:

```
view($DB) =

<bib>{
  for $i in $DB/Article/row
  return < article aid='{$i/aid/data()}'>{
             $i/ title , $i/year,
             for $a in $DB/Author/row[aid=$i/aid]
             return <author pid='{$a/pid/data()}'>{
                        $DB/Person/row[pid=$a/pid]/name/data()
                    }</author>
         }</ article >
}</bib>
```

This code converts the entire canonical view of the relational database into an XML tree that has the following type:

```
element bib {
   element article {
       attribute aid { xs:int },
       element title { xs:string }, element year { xs:string },
       element author { attribute pid { xs:int },
                        xs:string }* }* }
```

We assume that this view is materialized on a persistent storage, which is typically a native XML database. Our goal is to translate relational updates over the base data to XQuery updates (expressed in XUF) that incrementally modify the stored XML view to make it consistent with the base data. Our framework synthesizes a right-inverse functon view' of the view function, view, such that, for all valid view instances $V$, view(view'($V$)) is equal to $V$. Since an inverted view reconstructs the relational database from the view, it must preserve the key constraints of the relational schema. This is accomplished with an extension of the FLWOR XQuery syntax that enforces these constraints. In its simplest form, our FLWOR syntax may contain an optional 'unique' part:

**for** $v **in** $e$ **where** $p$($v) **unique** $k$($v) **return** $f$($v)

where $p$($v), $k$($v), and $f$($v) are XQuery expressions that depend on $v. The unique $k$($v) part of the FLWOR syntax skips the duplicate nodes $v from $e$, where the equality for duplicate removal is modulo the key function $k$($v). More specifically, let the result of $e$ be the sequence of nodes $(x_1, \ldots, x_n)$. Then the result of this FLWOR expression is the sequence concatenation $(y_1, \ldots, y_n)$, where

$$y_i = \begin{cases} () & \neg p(x_i) \\ () & \exists j{<}i : \ k(x_j) = k(x_i) \\ f(x_i) & (\text{otherwise}) \end{cases}$$

Based on this extended syntax, we can define a function view'($V) that is precisely the right-inverse of the previous view:

```
view'($V) =

<DB><Article>{
        for $ii in $V/article
        unique $ii/@aid/data()
        return <row><aid>{$ii/@aid/data()}</aid>
                    {$ii/ title , $ii/year}</row>
    }</ Article >
    <Author>{
        for $ia in $V/ article ,
            $aa in $ia/author
        unique ($ia/@aid/data(), $aa/@pid/data())
        return <row><aid>{$ia/@aid/data()}</aid>
                    <pid>{$aa/@pid/data()}</pid></row>
    }</Author>
    <Person>{
        for $ip in $V/ article ,
            $ap in $ip/author
        unique $ap/@pid/data()
        return <row><pid>{$ap/@pid/data()}</pid>
                    <name>{$ap/data()}</name>
                    <email>*</email></row>
    }</Person></DB>
```

In fact, as we will show in Section 7, our framework is capable of synthesizing this function automatically from the view function for many different forms of XQuery code. It will also generate the unique constraints automatically from the DB schema key constraints. For example, our system will infer that the applicable key for the first for-loop in view'($V) is key { /DB/Article/row, aid/data() }, given that the for-loop body has the same type as /DB/Article/row and is inside DB and Article element constructions. By applying the field path ./aid/data() to the for-loop body and normalizing it, our system will add the constraint **unique** $ii/@aid/data() to the for-loop. (Note that this heuristic method is applicable to relational key constraints only, where the key selector type corresponds to a relational table.) We can see that the unique parts of the for-loops in the inverted view are necessary in order to reconstruct the relational data without duplicate key values. The star in the email element content indicates that it can be any value, since this value is not used in the view. Stars in the inverted view signify that the view itself is a surjective function, so that we may not be able to reconstruct the complete database from the view. As we will show next, when our framework synthesizes the incremental view updates, the results of the inverted view must always pass through the view function to derive a view-to-view function, which eliminates the star values.

To show that view(view'($V)) is indeed equal to $V, we have to use normalization rules and the properties of the key constraints. More specifically, view(view'($V)) is:

```
<bib>{
  for $i in view'($V)/Article/row
  return < article aid='{$i/aid/data()}'>{
             $i/ title , $i/year,
             for $a in view'($V)/Author/row[aid=$i/aid]
             return <author pid='{$a/pid/data()}'>{
                        view'($V)/Person/row[pid=$a/pid]
                                /name/data()
                    }</author>
         }</ article >
}</bib>
```

XQuery normalization reduces general XQueries to normal forms, such as to for-loops whose domains are simple XPaths. In addition to fusing layers of XQuery compositions and eliminating their intermediate results, normal forms are simpler to analyze than their original forms. Normalization can be accomplished with the help of rules, such as:

$$\textbf{for } \$v \textbf{ in } (\textbf{for } \$w \textbf{ in } e_1 \textbf{ return } e_2) \textbf{ return } e_3 \\ = \quad \textbf{for } \$w \textbf{ in } e_1, \ \$v \textbf{ in } e_2 \textbf{ return } e_3$$

to normalize the for-loop domain. XQuery normalization is described in Section 6. If we expand the view'($V) definition and normalize the resulting code, we get the following code:

```
<bib>{
  for $ii in $V/article
  unique $ii/@aid/data()
  return <article aid='{$ii/@aid/data()}'>{
              $ii/title, $ii/year,
              for $ia in $V/article,
                  $aa in $ia/author
              where $ia/@aid/data()=$ii/@aid/data()
              unique ($ia/@aid/data(), $aa/@pid/data())
              return <author pid='{$aa/@pid/data()}'>{
                          for $ip in $V/article,
                              $ap in $ip/author
                          where $ap/@pid/data()=$aa/@pid/data()
                          unique $ap/@pid/data()
                          return $ap/data()
                        }</author>
         }</article>
}</bib>
```

To simplify this code further, we would need to take into account the key constraints expressed in the **unique** parts of the for-loops. A general rule that eliminates nested for-loops is:

```
for $v1 in f1($v̄), ..., $vn in fn($v̄)
return g( for $w1 in f1($w̄), ..., $wn in fn($w̄)
          where key($w̄) = key($v̄)
          unique key($w̄)
          return h($w̄) )
= for $v1 in f1($v̄), ..., $vn in fn($v̄)
  return g( h($v̄) )
```

where key, $g, h, f_1, \ldots, f_n$ are XQuery expressions that depend on the XQuery variables, $\$\overline{v}$ and $\$\overline{w}$, which are the variables $\$v_1, \ldots, \$v_n$ and $\$w_1, \ldots, \$w_n$, respectively. This rule, which is described in detail in Section 6, indicates that if we have a nested for-loop, where the inner for-loop variables have the same domains as those of the outer variables, modulo variable renaming (that is, $f_i(\$\overline{v})$ and $f_i(\$\overline{w})$, for all $i$), and they have the same key values, for some key function, then the values of $\$\overline{w}$ must be identical to those of $\$\overline{v}$, and therefore the inner for-loop can be eliminated. We can apply this rule to our previous normalized code, by noticing that the outer for-loop is over the variables $ia and $aa and the inner for-loop is over the variables $ip and $ap, where the key is $ap/@pid/data() and the predicate is $ap/@pid/data()=$aa/@pid/data(). This means that the code can be simplified to:

```
<bib>{
  for $ii in $V/article
  unique $ii/@aid/data()
  return <article aid='{$ii/@aid/data()}'>{
              $ii/title, $ii/year,
              for $ia in $V/article,
                  $aa in $ia/author
              where $ia/@aid/data()=$ii/@aid/data()
              unique ($ia/@aid/data(), $aa/@pid/data())
              return <author pid='{$aa/@pid/data()}'>{
                          $aa/data()
                        }</author>
         }</article>
}</bib>
```

Similarly, we can apply the same rule to the resulting XQuery by noticing that the outer for-loop is over the variable $ii and the inner for-loop is over the variable $ia, where the key is $ia/@aid/data(), and simplify the code further:

ID($V) =

```
<bib>{
  for $ii in $V/article
  unique $ii/@aid/data()
  return <article aid='{$ii/@aid/data()}'>{
              $ii/title, $ii/year,
              for $aa in $ii/author
              return <author pid='{$aa/@pid/data()}'>{
                          $aa/data()
                        }</author>
         }</article>
}</bib>
```

ID($V) is precisely the copy function applied to the view $V, which copies every node in the view. That is, ID($V) is equal to $V.

Consider now an SQL update over the base relational tables:

```
update Article set year=2009
where exists ( select *
               from Author a, Person p
               where a.aid = aid
                 and a.pid = p.pid
                 and p.name = 'Smith'
                 and title = 'XQuery')
```

which finds all articles authored by Smith that have XQuery as title and replaces their year with 2009. This update can be written in plain XQuery U($DB) over the canonical XML view $DB of the relational database that reconstructs the database reflecting the updates:

```
<DB><Article>{
      for $iu in $DB/Article/row
      return if some $au in $DB/Author/row,
                     $pu in $DB/Person/row
                satisfies $au/aid = $iu/aid
                      and $au/pid = $pu/pid
                      and $pu/name = 'Smith'
                      and $iu/title = 'XQuery'
             then <row>{$iu/aid, $iu/title}
                       <year>2009</year></row>
             else $iu
  }</Article>{$DB/Person, $DB/Author}</DB>
```

The new view after the update can be reconstructed from the old view $V using view(U(view'($V))). First, after normalization, U(view'($V)) becomes:

```
<DB><Article>{
      for $ii in $V/article
      return if some $aa in $ii/author
                satisfies $aa/data() = 'Smith'
                      and $ii/title = 'XQuery'
             then <row><aid>{$ii/@aid/data()}</aid>
                       {$ii/title,<year>2009</year>}</row>
             else <row><aid>{$ii/@aid/data()}</aid>
                       {$ii/title, $ii/year}</row>
  }</Article>{...}</DB>
```

where ... are the Author and Person parts of view'($V) (they are not affected by the update). Similarly, after normalization, view(U(view'($V))) becomes:

```
<bib>{
  for $ii in $V/article
  return if some $aa in $ii/author
            satisfies $aa/data() = 'Smith'
                  and $ii/title = 'XQuery'
         then <article aid='{$ii/@aid/data()}'>{
                   $ii/title, <year>2009</year>,
                   for $aa in $ii/author
                   return <author pid='{$aa/@pid/data()}'>{
                               $aa/data()
                             }</author>
              }</article>
         else $ii
}</bib>
```

Compare now this XQuery with the view copy function, ID($V) (the result of view(view'($V)) found previously). We can see that,

for each $ii, if the if-then-else condition is true, then their article elements are identical, except for the $ii/year component, which is <year>2009</year> in the new view. If we ignore the identical components, we can derive view(U(view'($V))) from the original view $V by just updating those components that differ. This means that the XUF update:

```
for  $ii  in  $V/ article
return  if  some $aa in  $ii / author ,
            satisfies   $aa/ data () = 'Smith'
                 and $ii / title  = 'XQuery'
        then replace node $ii / year  with <year>2009</year>
        else  ()
```

will destructively modify the view to become the modified view after the update U. In Section 7, we present a heuristic, conservative algorithm that, given a view mapping from the old view to the new view, finds those program fragments that differ from the identity view mapping and generates an XUF update for each one. For this algorithm to be effective, it is essential that the view mapping program be normalized to make program equivalence more tractable (which is undecidable in general). It basically compares if the two programs are identical, modulo variable renaming, but it also takes into account the possible alternatives in sequence construction.

# 4. MAPPING SQL UPDATES TO XQUERY

In our framework, SQL updates over the underlying relational database are translated to pure XQuery code that reconstructs the entire relational database reflecting the updates. The resulting code is fused with the view and then translated to XUF updates over the materialized view. Let $DB$ be a relational database schema that consists of the relations $R_i(A_{i1} : t_{i1}, \ldots, A_{in} : t_{in})$, where $t_{ij}$ are SQL base types. The canonical view of $DB$ is the XML schema:

$$\text{element DB } \{ \ \ldots, \ \text{element } R_i \ T_i*, \ \ldots \ \}$$

where the tuple type $T_i$ is:

$$\text{element row } \{ \ \ldots, \ \text{element } A_{ij} \ \{ \ t'_{ij} \ \}, \ \ldots \ \}$$

where $t'_{ij}$ is the XML equivalent of the SQL base type $t_{ij}$. Views like this have been used by many other XML view frameworks, such as SilkRoute [9]. Our task in this section is to map an update over the relational database, expressed in SQL, to a pure XQuery program that reconstructs the entire database, reflecting the update.

- **Update:** The SQL update

  **update** $R_i$ **set** $A_{ij}$=$e_1$ **where** $e_2$

  is mapped to the following XQuery over the canonical XML view $DB:

  ```
  <DB>{$DB/R1, ...,
        <Ri>{ for $ri in $DB/Ri/row return
                    if e2'
                    then <row>{ $ri/Ai1, ...,
                                <Aij>{e1'}</Aij>,
                                ...}</row>
                    else $ri }</Ri>,
        ...}</DB>
  ```

  where $e'_1/e'_2$ are the XQuery translations of the SQL expressions $e_1/e_2$. That is, the data of all the tables except $R_i$ are copied from the old database state as is. For each tuple $\$r_i$ of $R_i$, though, we check if it satisfies the predicate $e_2$. If it does, it is 'updated' by embedding a new row into the new database state whose $A_{ij}$ column is $e_1$ while the other columns are copied from the old row.

- **Delete:** The SQL update

  **delete from** $R_i$ **where** $e$

  is mapped to:

  ```
  <DB>{$DB/R1, ...,
        <Ri>{ for $ri in $DB/Ri/row return
                    if e' then () else $ri }</Ri>,
        ...}</DB>
  ```

  where $e'$ is the XQuery translation of the SQL expression $e$. That is, if a tuple $\$r_i$ of $R_i$ satisfies the predicate $e$, it is not embedded into the new database state.

- **Insert:** Finally, the SQL update

  **insert into** $R_i$ **values** $(e_1, \ldots, e_n)$

  is mapped to:

  ```
  <DB>{$DB/R1, ...,
        <Ri>{$DB/Ri/row}
            <row><Ai1>{e1'}</Ai1>
                 ...<Ain>{en'}</Ain>
                 </row></Ri>,
        ...}</DB>
  ```

  where the new row is appended to the table content $DB/$R_i$/row.

SQL expressions are mapped to XQuery expressions in a straightforward way. For example, the SQL projection $v.A$ is mapped to $\$v/A$. A simple column $A$ is mapped to $\$r_i/A$, where the variable $\$r_i$ ranges over the table $R_i$ and can be inferred from the SQL query context. Embedded SQL queries, such as

**exists (select * from** $R_{i_1}$ $v_1$**, ...,** $R_{i_n}$ $v_n$ **where** $e$**)**

are mapped to FLWOR loops, such as:

**some** $\$v_1$ **in** $DB/$R_{i_1}$/row, ..., $\$v_n$ **in** $DB/$R_{i_n}$/row
**satisfies** $e'$

For example, the SQL update used in our example:

```
update Article  set  year=2009
where exists  ( select  *  from Author a,  Person  p
                where a.aid  = aid  and a.pid  = p.pid
                  and p.name = 'Smith'  and  title  = 'XQuery')
```

is translated to:

```
<DB><Article>{
      for  $article  in  $DB/Article/row
      return  if  some $a in  $DB/Author/row,
                      $p  in  $DB/Person/row
                  satisfies   $a/aid  =  $article /aid
                        and $a/pid  = $p/pid
                        and $p/name = 'Smith'
                        and  $article / title  = 'XQuery'
              then <row>{ $article/aid ,  $article / title }
                        <year>2009</year></row>
              else  $article
  }</ Article >{$DB/Person, $DB/Author}</DB>
```

# 5. SYNTHESIZING THE RIGHT-INVERSE

Given that XQuery is computationally complete, one of hardest tasks for updating materialized views is to synthesize the inverse function of a view expressed in XQuery. This task becomes even harder when the underlying database is relational because the view must be expressed in terms of joins, which in general do not have an inverse function. In this section, we describe a heuristic algorithm

that synthesizes the inverse of a view mapping. It can handle many XQuery forms that are used frequently in real view mapping scenarios, including views that use one-to-one and one-to-many joins. Our program synthesis algorithm is guided by the type (schema) of the view source code, which can be inferred from the input schema of the view (the schema of the underlying relational database).

## 5.1 Preliminaries

Given an XQuery expression, $f(x)$, that depends on the variable $x$ (a shorthand for the XQuery variable $\$x$), the right-inverse $\mathcal{I}_x(f(x), y)$ is an XQuery expression $g(y)$ that depends on $y$, such that $y = f(x) \Rightarrow x = g(y)$. This definition implies that $f(g(y)) = y$, which means that $g(y) = \mathcal{I}_x(f(x), y)$ is the right-inverse of $f(x)$. In this Section, we present the rules for extracting $\mathcal{I}_x(e, y)$ for most forms of XQuery expression $e$. Some $\mathcal{I}_x$ rules may return an error, which is denoted by $\perp$. For example, if the view is

y = **if** x>4 **then** x-1 **else** x+2

then the right-inverse is:

x = **if** y+1>4 **then** (**if** y-2>4 **then** y+1 **else** $\perp$) **else** y-2

that is, if y is equal to 4, 5, or 6, then $x$ must be a $\perp$ value since this y can be produced in two different ways (e.g., both x=2 and x=5 produce y=4). Some rules may also return $*$, which indicates that it can be any XML node of the proper type. For example, if the type of x is element A { element B xs:int, element C xs:int }, then the right-inverse of y = x/C is x = <A>{*,y/self::C}</A>, which indicates that x/B can be any B element.

The most difficult expression to invert is XQuery sequence concatenation, because, in contrast to regular tuples and records, nested sequences are flatten out to sequences of XML nodes or base values. For example, if the type of x is element A { element B xs:int, element C xs:int }, then, for y = (x/B,x/C), we would like to derive x = <A>{y/self::B,y/self::C}</A> in a compositional way. That is, since y1 = x1/B implies x1 = <A>{y1/self::B,*}</A> and y2 = x2/C implies x1 = <A>{*,y2/self::C}</A>, the goal is to derive the previous solution for y = (x/B,x/C) = (y1,y2). But we must have x=x1=x2, in order to have a valid solution for x. By looking at x1 and x2, this can only be done if we match x1 with x2, since $*$ matches with any expression. In fact, as we will see next, we would need a unification algorithm that also matches variables with expressions by binding these variables to these expressions.

Consider now the inversion of a for-loop, such as

y = **for** $v **in** x/C/data() **return** $v+1

We can see that y must be equal to the sequence of integers $v+1. Thus, each value $v must be equal to an element of y minus one, which implies that

(**for** $w **in** y **return** $w-1) = x/C/data()

which can be solved for x since it has the form y' = x/C/data(). Therefore, to invert a for-loop, we must first invert the for-loop body with respect to the for-loop variable, and then invert the for-loop domain with respect to x. This rule works correctly if the for-loop body does not refer to a non-local variable (a variable other than the for-loop variable). But references to non-local variables are very common in a view mapping over a relational database, as we can see from the view($DB) example in Section 3. In fact, these non-local references correspond to joins. Consider, for example, the following non-local reference to $x in the body of a for-loop on $y:

**for** $x **in** e1
**return** <A>{... **for** $y **in** e2
**return** <B>{... $x ...}</B>...}</A>

When we invert the body of the inner loop <B>... $x ... </B> with respect to the loop variable $y, we encounter the non-local variable $x. Then, at this point of the inversion process, instead of a solution $y = ..., we get a solution $x = .... This solution must be matched (unified) with the solution for $x found in other places in the body of the outer for-loop. We use a special binding list, called the Inverted Vars ($\mathcal{IV}$) that binds variables to their inverted expressions. When we find a new non-local contribution, such as the previous $x = ..., we unify it with the existing binding of the variable in $\mathcal{IV}$, if exists. This contribution to a for-loop variable from $\mathcal{IV}$ is added to the solution found by inverting the for-loop body with respect to this variable.

The case that really needs unification instead of matching is when inverting equality predicates in FLWOR or XPath conditions. Consider, for example, the inversion y = $x/Person/row[pid=$a/pid] taken from the view($x) definition in Section 3. The condition pid=$a/pid is very important because it specifies a join between Author and Person. This predicate should provide a new contribution to the $a solution that has its $a/pid equal to the Person's pid. This can be done by solving y = $x/Person/row, then solving w = $a/pid and w = y/pid, where w is a new (fresh) unification variable, and then unifying the two solutions to yield a binding for w that makes the predicate y/pid=$a/pid true. That is, y = $x/Person/row will give us a solution for y, then w = y/pid will give us a solution for w, and finally w = $a/pid will give us a contributing solution for $a.

Our unification algorithm for XQuery expressions algorithm is based on XQuery code equality. Given that code equality is undecidable in general, our unification algorithm uses heuristics to check for identical XQuery program structures, modulo variable renaming. For example, a for-loop can be unified with another for-loop as long as their corresponding for-loop domains, predicates, and bodies unify. In addition, $*$ unifies with anything while the unification of a variable with an expression binds the variable to the expression (as long as there are no cyclic references). The following are some of the unification rules. The outcome of the unification unify$(e_1, e_2)$ is either $\perp$ (failure to unify) or the unified expression along with possible new bindings from XQuery variables (which are the unification variables) to their bindings (XQuery expressions).

$$\text{unify}(e, e) = e \tag{U-1}$$

$$\text{unify}(*, e) = e \tag{U-2}$$

$$\text{unify}(v, e) = e \quad \text{and bind } v \text{ to } e \tag{U-3}$$

$$\text{unify}(<A>\{e_1\}</A>, <A>\{e_2\}</A>) = <A>\{\text{unify}(e_1, e_2)\}</A> \tag{U-4}$$

$$\text{unify}((e_1, e_2), e_3) = \text{unify}(e_1, \text{unify}(e_2, e_3)) \tag{U-5}$$

$$\text{unify}(e_1, e_2) = \textbf{if } e_1 = e_2 \textbf{ then } e_1 \textbf{ else } \perp \quad (\text{if } e_1 \text{ and } e_2 \text{ are of base type}) \tag{U-6}$$

## 5.2 The Inversion Algorithm

We are now ready to give some of the rules for $\mathcal{I}_x$. The validity of these rules (ie, that they generate XQuery code that satisfies the right-inverse property) is proved in Appendix A. The program synthesis $\mathcal{I}_x(e, y)$ is described by considering different cases for the XQuery expression $e$. The program synthesis rules given below can be easily implemented as a simple tree traversal algorithm over the abstract syntax tree of the XQuery expression $e$. Consequently, the time complexity of this algorithm is proportional to the query size (the number of nodes in the abstract syntax tree).

### 5.2.1 Variables

The simplest case of an XQuery expression is a variable. If this variable is equal to the inversion variable $x$, then $x = y$:

$$\mathcal{I}_x(\$x, y) = y \tag{I-1}$$

If the variable is different from $x$, then

$$\mathcal{I}_x(\$v, y) = * \tag{I-2}$$

that is, the solution for $x$ is $*$, which does not contribute any information about $x$. But, as a side-effect, the Inverted Vars, $\mathcal{IV}$, is extended with the binding from $v$ to $y$, if $v$ is not already bound in $\mathcal{IV}$; if $v$ has already a binding in $\mathcal{IV}$, which is accessed by $\mathcal{IV}[v]$, it is replaced with unify$(\mathcal{IV}[v], y)$. That is, we are contributing a new solution $y$ to $v$.

### 5.2.2 Simple XQueries

XQuery constants do not contribute any solution to $x$ but they pose a restriction on the $y$ value:

$$\mathcal{I}_x(c, y) = \textbf{if } y = c \textbf{ then } * \textbf{ else } \bot \tag{I-3}$$

Simple arithmetic expressions with a constant operand can be easily inverted in a straightforward way. For example:

$$\mathcal{I}_x(e + c, y) = \mathcal{I}_x(e, y - c) \tag{I-4}$$

For an if-then-else expression **if** $p$ **then** $e_1$ **else** $e_2$, we invert both the true and false parts: $x_1 = \mathcal{I}_x(e_1, y)$ and $x_2 = \mathcal{I}_x(e_2, y)$. If $x_1$ unifies with $x_2$, then

$$\mathcal{I}_x(\textbf{if } p \textbf{ then } e_1 \textbf{ else } e_2, y) = \text{unify}(x_1, x_2) \tag{I-5}$$

otherwise, we would have to take into account the value of the predicate for $x = x_1$ and $x = x_2$:

$$\begin{aligned}&\mathcal{I}_x(\textbf{if } p(\$x) \textbf{ then } e_1 \textbf{ else } e_2, y) \\ &= \textbf{if } p(x_1) \textbf{ then } (\textbf{if } p(x_2) \textbf{ then } x_1 \textbf{ else } \bot) \textbf{ else } x_2\end{aligned} \tag{I-6}$$

That is, if $p(x_1)$ is true, then $y$ must be equal to $e_1$, but if in addition $p(x_2)$ is false, then $y$ must also be equal to $e_2$, which means that $x$ must be equal to both $x_1$ and $x_2$, which is assumed false.

### 5.2.3 Paths and Elements

The inverse of an XPath step of the form $e$/axis::test can be found by considering the inferred type of the XQuery expression $e$. If the type of $e$ is inferred to be: element $B$ { ..., element $A$ $t$, ... }, then:

$$\begin{aligned}&\mathcal{I}_x(e/A, y) \\ &= \mathcal{I}_x(e, <B>\{*, \ldots, y/\text{self::}A, \ldots, *\}</B>)\end{aligned} \tag{I-7}$$

That is, $e$ must be an element construction whose all but the $A$ children are $*$, while the $A$ children must satisfy $e/A = y$. The other forward XPath steps of the form $e$/axis::test can be handled in the same way: since we can infer the type of $e$, we can embed $y$ into a number of element constructions, $c(y)$, so that $c(y)/\text{axis::test} = y$. In addition, if the schema is not recursive, backward steps can always be translated to equivalent forward steps.

For the element content, we have:

$$\mathcal{I}_x(e/\text{data}(), y) = \mathcal{I}_x(e, <B>\{y\}</B>) \tag{I-8}$$

given that $e$ is inferred to be of type: element $B$ $t$. Finally, for an element construction, we have:

$$\mathcal{I}_x(<A>\{e\}</A>, y) = \mathcal{I}_x(e, y/\text{self::}A/\text{node}()) \tag{I-9}$$

which imposes the restriction that $y$ be an element tagged $A$ and gives the solution that $e$ be the $y$ element content.

### 5.2.4 FLWOR Blocks

As we discussed at the beginning of this section, we can invert a for-loop by inverting the loop body with respect to the for-loop variable, and then invert the for-loop domain with respect to x, given that y is now the result of the inverted loop body:

$$\begin{aligned}&\mathcal{I}_x(\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2, y) \\ &= \mathcal{I}_x(e_1, \textbf{for } \$v' \textbf{ in } y \textbf{ return } \mathcal{I}_v(e_2, \$v'))\end{aligned} \tag{I-10}$$

For example, if y = **for** $v **in** x **return** $v+1, then we invert $v'=$v+1 with respect to $v to get $v=$v'-1 and we get x = **for** $v' **in** y **return** $v'-1. Note that, by definition, $\$v = \mathcal{I}_v(e_2, \$v')$. After this solution for $\$v$ is calculated, we may have a binding $\mathcal{IV}[v]$ in Inverted Vars that accumulates non-local references to the variable $\$v$ in $e_2$, as we discussed earlier (for example, when $\$v$ is referenced in the body of an inner for-loop in $e_2$). These contributions must be merged using unify$(\mathcal{I}_v(e_2, \$v'), \mathcal{IV}[v])$, which unifies all solutions to $\$v$. Then, Rule (I-10) becomes:

$$\begin{aligned}&\mathcal{I}_x(\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2, y) \\ &= \mathcal{I}_x(e_1, \textbf{for } \$v' \textbf{ in } y \textbf{ return } \text{unify}(\mathcal{I}_v(e_2, \$v'), \mathcal{IV}[v]))\end{aligned} \tag{I-10'}$$

Also note that Rule (I-10) can only apply if the loop body is not of a sequence type. For example, it cannot invert y = **for** $v **in** x **return** ($v,$v+1). Finally, if the for-loop has a predicate, $p(\$v)$, that depends on the loop variable, we move the predicate to the loop domain and use Rule (I-12) described next:

$$\begin{aligned}&\mathcal{I}_x(\textbf{for } \$v \textbf{ in } e_1 \textbf{ where } p(\$v) \textbf{ return } e_2, y) \\ &= \mathcal{I}_x(\textbf{for } \$v \textbf{ in } e_1\textbf{[}p(.)\textbf{]} \textbf{ return } e_2, y)\end{aligned} \tag{I-11}$$

### 5.2.5 Equality Predicates

Inverting an equality predicate, $y = e\textbf{[}e_1 = e_2\textbf{]}$, could be as easy as inverting $y = e$. The equality $e_1 = e_2$ though may give us more information about the inverse code since it relates data produced by two different places in the view code. The most common example is a nested for-loop, which corresponds to a relational join, as we saw in Section 3. Thus, in addition to the solution $\mathcal{I}_x(e\textbf{[}e_1 = e_2\textbf{]}, y) = \mathcal{I}_x(e, y)$, we have more contributions from the predicate. To calculate these contributions using our inversion algorithm, we let $\$z$ be the current context of the predicate (the XQuery dot), where $\$z$ is an XQuery variable, we let $\$w$ be the result of each branch of the equality, where $\$w$ is an another XQuery variable, and we invert $\$w = e_1$ and $\$w = e_2$ with respect to $\$z$ using $\mathcal{I}_z(e_1, \$w)$ and $\mathcal{I}_z(e_2, \$w)$. All these solutions are unified, which provide a binding for $\$w$, and which in turn is used to eliminate $\$w$ from the solutions. More specifically:

$$\begin{aligned}&\mathcal{I}_x(e\textbf{[}e_1 = e_2\textbf{]}, y) \\ &= \mathcal{I}_x(e, \text{unify}(\text{unify}(y, \mathcal{I}_z(e_1', \$w)), \mathcal{I}_z(e_2', \$w)))\end{aligned} \tag{I-12}$$

where $e_1'/e_2'$ is equal to $e_1/e_2$ with the current context (the dot) replaced with $\$z$.

Consider, for example, inverting a part of the view defined in Section 3:

$\mathcal{I}_x$( y, $x/Author/row[aid=$i/aid] )

Based on Rule (I-12), it is equal to

$\mathcal{I}_x$( $x/Author/row,
unify( unify( y, $\mathcal{I}_z$($z/aid,$w) ), $\mathcal{I}_z$($i/aid,$w) ) )

for some new variables $z and $w. Using our inversion rules, $\mathcal{I}_z$($z/aid,$w) gives the solution $z = <row>{$w/self::aid,*}</row> (an Author row), while $\mathcal{I}_z$($i/aid,$w) gives the solution $z = $*$, but accumulates the contribution $i = <row>{$w/self::aid,*,*}</row> (an

$$\text{split}(\text{element } A\, t_1, \text{element } A\, t_2, y) \quad \text{(S-1)}$$
$$= \text{split}(t_1, t_2, y/\text{self::}A/\text{node}())$$
$$\text{split}(\text{element } A\, t_1, \text{element } B\, t_2, y) \quad \text{(S-2)}$$
$$= (y/\text{self::}A,\ y/\text{self::}B)$$
$$\text{split}((t_1, t_2), t, y) = (p_1|p_2,\ p) \quad \text{(S-3)}$$
*where* $\text{split}(t_1, t, y) = (p_1, p)$
*and* $\text{split}(t_2, t, y) = (p_2, p)$
$$\text{split}(t, (t_1, t_2), y) = (p,\ p_1|p_2) \quad \text{(S-4)}$$
*where* $\text{split}(t, t_1, y) = (p, p_1)$
*and* $\text{split}(t, t_2, y) = (p, p_2)$
$$\text{split}(\text{xs:string}, \text{xs:string}, y) = (y[1],\ y[2]) \quad \text{(S-5)}$$
$$\text{split}(t_1, t_2, y) = \bot \quad \textit{(otherwise)} \quad \text{(S-6)}$$

**Figure 1: The Split Function**

Article row). Given that y=<row>{y/aid,y/pid}</row>, if the solutions for $z are unified with y, we get $w = y/aid and the contribution to $i becomes <row>{y/aid,*,*}</row>.

### 5.2.6 *Sequence Concatenation*

As we discussed earlier, the most difficult case to handle is XQuery sequence concatenation because nested sequences are flatten out to sequences of XML nodes or base values. Let $y = (e_1, e_2)$ (recall that in XQuery a sequence $(e_1, e_2, \ldots, e_n)$ can be written using binary sequence concatenations $((e_1, e_2), \ldots, e_n)$). If the types of $e_1$ and $e_2$ are basic types (such as strings), then $y$ must be a sequence of two values, $(y[1], y[2])$. In that case, let $x_1$ be the solution of $y[1] = e_1$ and $x_2$ be the solution of $y[2] = e_2$. Then, the solution for $x$ must be $\text{unify}(x_1, x_2)$, which will fail if $x_1$ and $x_2$ are incompatible. In general, $e_1$ and $e_2$ can be of any type, including sequence types. What is needed is a method to split $y$ into two components $y_1$ and $y_2$ so that $y_1/y_2$ have the same type as $e_1/e_2$, respectively. Then the inverse of $y = (e_1, e_2)$ would be the unification of the inverses for $y_1 = e_1$ and $y_2 = e_2$. This splitting is accomplished with the function $\text{split}(t_1, t_2, y)$ that derives two predicates, $p_1$ and $p_2$, to brake $y$ into two components $(y_1, y_2)$ so that $y_1 = y$**[**$p_1$**]** and $y_2 = y$**[**$p_2$**]**, and the types of $y_1/y_2$ match those of $e_1/e_2$, respectively. The inverse rule for sequences is:

$$\mathcal{I}_x((e_1, e_2), y) = \text{unify}(\mathcal{I}_x(e_1, y\textbf{[}p_1\textbf{]}), \mathcal{I}_x(e_2, y\textbf{[}p_2\textbf{]})) \quad \text{(I-13)}$$

where $e_1/e_2$ have types $t_1/t_2$ and $(p_1, p_2) = \text{split}(t_1, t_2, .)$. Some of the rules for the split function are given in Figure 1. Rule (S-1) indicates that if the types we try to discriminate are elements with the same tag, then we can only split $y$ if we consider the content of $y$. On the other hand, Rule (S-1) indicates that if the types are elements with different tags, $A$ and $B$, then we can simply split $y$ to $y/\text{self::}A$ and $y/\text{self::}B$. For example, from Rules (I-13) and (S-5), the inverse of y = (x-1,x*2) is unify($\mathcal{I}_x$(y[1],x-1),$\mathcal{I}_x$(y[2],x*2)), which is equal to unify(y[1]+1,y[2]/2). Finally, from Rule (U-6), we get:

x = **if** y[1]+1=y[2]/2 **then** y[1]+1 **else** $\bot$

## 5.3 A Complete Example

Suppose that the type of $x is:

element A { element B int, element C int,
element D int* }

and that y is equal to:

<M>{ for $v in $x/D return <T>{$v/data()+1}</T>,
<C>1</C>,
<N>{$x/B/data()}</N>}</M>

We want to synthesize the XQuery that computes x, based on the inversion rules. From Rule (I-10), we get:

$\mathcal{I}_x$( for $v in $x/D return <T>{$v+1}</T>, z )
= $\mathcal{I}_x$( $x/D, for $v' in z
return $\mathcal{I}_v$(<T>{$v+1}</T>,$v') )

Since the type of $x/D is element D int* (the type qualifier * is ignored), we can use Rules (I-7) and (I-1) to get:

$\mathcal{I}_x$( $x/D, z ) = $\mathcal{I}_x$( $x, <A>{ $*$, $*$, z/self::D }</A>)
= <A>{ $*$, $*$, z/self::D }</A>

From Rules (I-9), (I-4), (I-8), and (I-1), we have:

$\mathcal{I}_v$( <T>{$v/data()+1}</T>, z )
= $\mathcal{I}_v$( $v/data()+1, z/self::T/node() )
= $\mathcal{I}_v$( $v/data(), z/self::T/node()-1 )
= $\mathcal{I}_v$( $v, <D>{z/self::T/node()-1}</D>)
= <D>{z/self::T/node()-1}</D>

After substituting these results in the for-loop, we get:

$\mathcal{I}_x$( for $v in $x/D return <T>{$v+1}</T>, z )
= $\mathcal{I}_x$( $x/D, for $v' in z return
$\mathcal{I}_v$(<T>{$v+1}</T>,$v') )
= <A>{ $*$, $*$, (for $v' in z return
$\mathcal{I}_v$(<T>{$v+1}</T>,$v'))/self::D }</A>
= <A>{ $*$, $*$, for $v' in z return
<D>{$v'/self::T/node()-1}</D> }</A>

Using the Rules (I-9) and (I-3), we get:

$\mathcal{I}_x$(<C>1</C>,z) = $\mathcal{I}_x$(1,z/self::C/node())
= if z/self::C/node() = 1 then $*$ else $\bot$

From Rule (I-7), we have:

$\mathcal{I}_x$( $x/B, w ) = $\mathcal{I}_x$( $x, <A>{w/self::B,*,*}</A> )
= <A>{w/self::B,*,*}</A>

Therefore, using Rules (I-9), (I-8), (I-7), and (I-1), we have:

$\mathcal{I}_x$( <N>{$x/B/data()}</N>, z )
= $\mathcal{I}_x$( $x/B/data(), z/self::N/node() )
= $\mathcal{I}_x$( $x/B, <B>{z/self::N/node()}</B> )
= $\mathcal{I}_x$( $x, <A>{(<B>{z/self::N/node()}</B>)/self::B,
*,*}</A> )
= <A>{<B>{z/self::N/node()}</B>,*,*}</A>

Finally, using Rule (I-9), the solution for $x is:

$\mathcal{I}_x$( ( for $v in $x/D return <T>{$v/data()+1}</T>,
<C>1</C>, <N>{$x/B/data()}</N>),
y/self::M/node() )

These sequence elements have types element T int*, element C int, and element M int, which means that, using theRules (S-2) and (S-3), their split predicates are ./self::T, ./self::C, and ./self::M, respectively. To invert this sequence using Rule (I-13), we need to unify the following three components:

<A>{ $*$, $*$, for $v' in z/self::T return
<D>{$v'/self::T/node()-1}</D> }</A>
if z/self::C/node() = 1 then $*$ else $\bot$
<A>{<B>{z/self::N/node()}</B>,*,*}</A>

for z = y/self::M/node(). Based on the rules for match, we get the following solution for $x:

<A>{ <B>{z/self::N/node()}</B>,
if z/self::C/node() = 1 then $*$ else $\bot$,
for $v' in z/self::T return
<D>{$v'/self::T/node()-1}</D> }</A>

Therefore, the final solution for $x is:

```
if y/self::M/C/node() = 1
then <A>{ <B>{y/self::M/N/node()}</B>, *,
          for $v' in y/self::M/T return
              <D>{$v'/self::T/node()-1}</D> }</A>
else ⊥
```

# 6. NORMALIZING THE VIEW MAPPINGS

When composing layers of XML mappings and updates, such as view(U(view'($V))) that captures view updates, each mapping traverses its input XML tree and constructs a new XML tree as output, which is immediately consumed by the next mapping, which in turn constructs a new XML tree, and so on. We would like to fuse these layers into a single program that does not produce these intermediate XML trees. In addition, one of our goals is to translate the view functions that reconstruct the view to efficient XUF updates that destructively modify the materialized view. This goal can be accomplished if we recognize certain patterns in the view code that copy the view data as is, in order to eliminate this code. These patterns can only be identified if the view code be normalized. In contrast to scripting languages, XQuery is amenable to normalization and optimization since it is purely functional. Such optimizations have already been addressed by related work [15, 10] and have also been described in our earlier work [7].

Normalization can be done with the help of 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** $e_1$, $\$v$ **in** $e_2$ **return** $e_3$

to partial evaluate operations on constructions and sequences, and eventually eliminate some of their components. In addition, as we saw in Section 3, we can eliminate nested for-loops using the following general rule:

**for** $\$v_1$ **in** $f_1(\$\overline{v})$, ..., $\$v_n$ **in** $f_n(\$\overline{v})$
**return** g( **for** $\$w_1$ **in** $f_1(\$\overline{w})$, ..., $\$w_n$ **in** $f_n(\$\overline{w})$
**where** key($\$\overline{w}$) = key($\$\overline{v}$)
**unique** key($\$\overline{w}$)
**return** h($\$\overline{w}$) )
= **for** $\$v_1$ **in** $f_1(\$\overline{v})$, ..., $\$v_n$ **in** $f_n(\$\overline{v})$
**return** g( h($\$\overline{v}$) )

This rule indicates that if we have a nested for-loop, where the inner for-loop variables have the same domains as those of the outer variables, modulo variable renaming (that is, $f_i(\$\overline{v})$ and $f_i(\$\overline{w})$, for all $i$), and they have the same key values, for some key function, then the values of $\$\overline{w}$ must be identical to those of $\$\overline{v}$, and therefore the inner for-loop can be eliminated. The validity of this rule can be asserted by considering the body of the outer for-loop for one instance of $\$\overline{v}$ from the domains $f_i(\$\overline{v})$. Then, the values of $\$\overline{w}$ of the inner for-loop must be equal to the $\$\overline{v}$ values because 1) their mappings through the key function are equal and 2) since the key function is the key of the inner for-loop, there are no other $\$\overline{w}$ values from these domains with the same key.

# 7. SYNTHESIZING XQUERY UPDATES FROM THE VIEW MAPPING

The final task is, given a pure XQuery expression $F(V)$ from view to view, to rewrite it into an efficient program (expressed in XUF) that destructively updates the materialized view $V$ (the input view) to transform it to the output view. This is the hardest task among all others in updating materialized views and, to the best of our knowledge, no related work has addressed it. Obviously, one may update the materialized view $V$ using the update **replace node** $V$**with** $F(V)$, but our goal is to update only those parts of the new view $F(V)$ that are different from the old view. Although this task is far harder than the tasks addressed in the earlier sections, the update generation algorithm does not have to be complete. When we recognize that a part of the input view is propagated to the output view as is, then we discard the code that generates this part; otherwise we brake the code further into subparts and apply this method recursively until we have no choice but to generate an XUF update. In fact, one heuristic to recognize those parts that remain unchanged is to compare the $F(V)$ code with the copy function over $V$ that creates an exact copy of $V$, by traversing and reconstructing all nodes of $V$. 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 view schema. The comparison of $F(V)$ to the copy function can only be effective if $F(V)$ be normalized, as described in Section 6, so that for-loops that generate the view output constructions are over the same paths as the corresponding paths in the copy function (modulo variable renaming). Our method also takes into account the possible alternatives in sequence construction. (Note that, although program equivalence is undecidable, we are only interested in effective heuristics.)

Figure 2 gives some of the rules for deriving the XUF updates. Function $\mathcal{U}[\![e]\!]\, t\, d\, q\, x$ focuses on a fragment of the view of type $t$. Let $d$ be the fragment of the initial view that corresponds to the type $t$ and let $e$ be the XQuery code that maps $d$ to a new view fragment of type $t$. Then, $\mathcal{U}[\![e]\!]\, t\, d\, q\, x$ derives an XUF expression over $d$ that destructively modifies $d$ in such a way that the new $d$ is equal to the result of $e$. In a way, $d$ brings out the copy function discussed earlier by expanding its copying components on demand, when is necessary to compare the subparts of $e$ and $d$. The way $d$ is expanded does not require variable renaming for code equivalence since the variables used in $d$ are taken from $e$. The parameter $q$ is a flag the indicates whether a node insertion ($q$ = true) or a replacement ($q$ = false) should take place when $e$ does not match $p$. For insertions, $x$ is the insertion destination. If $x = \perp$ (no insertion destination), then $e$ is inserted as the first child into $p$'s parent. For the view mapping $F(V)$ over the view $V$ of type $t_v$, the XUF code generation is accomplished using $\mathcal{U}[\![F(V)]\!]\, t_v\, V$ **false** $\perp$.

Rule (X-1) indicates that if the view mapping $e$ matches the associated unmodified view part $p$, then no update is generated. The operation $e \equiv p$ is XQuery code equivalence, which returns true if and only if the code $e$ is identical to the code of $p$. Rules (X-6) through (X-8), which apply when none of the other rules apply, generate XUF updates to modify $d$ so that the new $d$ after the updates is equal to the result of $e$. By default (when $q$ = false), $p$ is replaced by $e$ (Rule (X-6)); the last two rules apply when we handle sequences (as will be shown next). Rule (X-2) indicates that if $e$ is an element construction, we should apply this algorithm recursively to the element content and the $p$ content, which is $p/\text{self}::A/\text{node}()$. Rule (X-4) applies when the for-loop domain $e_1$ is identical to $p$. In that case, since $p$ is equal to **for** $\$v$ **in** $p$ **return** $\$v$, we must apply the algorithm recursively to the for-loop bodies, $e_2$ and $\$v$. The most important rule is Rule (X-5), which handles sequences using the following function:

$$
\begin{array}{lcll}
\mathcal{U}[\![e]\!]\,t\,p\,q\,x & = & () & \text{if } e \equiv p \quad \text{(X-1)}\\
\mathcal{U}[\![\texttt{<A>\{e\}</A>}]\!]\,[\![\text{element } A\ t]\!]\,p\,q\,x & = & \mathcal{U}[\![e]\!]\,t\,(p/\text{self::}A/\text{node()})\,q\,x & \quad \text{(X-2)}\\
\mathcal{U}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\,t\,p\,q\,x & = & \textbf{if } e_1 \textbf{ then } (\mathcal{U}[\![e_2]\!]\,t\,p\,q\,x) \textbf{ else } (\mathcal{U}[\![e_3]\!]\,t\,p\,q\,x) & \quad \text{(X-3)}\\
\mathcal{U}[\![\textbf{for } \$v \textbf{ in } e_1 \textbf{ return } e_2]\!]\,[\![t*]\!]\,p\,q\,x & = & \textbf{for } \$v \textbf{ in } e_1 \textbf{ return } (\mathcal{U}[\![e_2]\!]\,t\ \$v\,q\,x) & \text{if } e_1 \equiv p \quad \text{(X-4)}\\
\mathcal{U}[\![(e_1,\ldots,e_n)]\!]\,t\,p\,q\,x & = & \mathcal{UL}\,(e_1,\ldots,e_n)\,t\,t\,p \text{ empty } \bot & \text{where } t=(t_1,\ldots,t_m) \quad \text{(X-5)}\\
\mathcal{U}[\![e]\!]\,t\,p\,q\,x & = & \textbf{replace node } p \textbf{ with } e & \text{if } \neg q \quad \text{(X-6)}\\
\mathcal{U}[\![e]\!]\,t\,p\,q\,x & = & \textbf{insert node } e \textbf{ as first into } (p/..) & \text{if } q \wedge x = \bot \quad \text{(X-7)}\\
\mathcal{U}[\![e]\!]\,t\,p\,q\,x & = & \textbf{insert node } e \textbf{ after } x & \text{if } q \wedge x \neq \bot \quad \text{(X-8)}
\end{array}
$$

**Figure 2: Some Heuristic Rules for Update Derivation**

$$
\begin{array}{l}
\mathcal{UL}\,(e_1,e_2,\ldots,e_n)\,(t_1,t_2,\ldots,t_m)\,t\,p\,s\,x\\
= \text{let } d = \text{desc}[\![t_1,t]\!](p)\\
\quad \text{in if } t_1 \text{ is a star type}\\
\quad\quad \text{then if } e_1 : t_1\\
\quad\quad\quad \text{then ( if } d \equiv e_1 \ \wedge\ s \neq \text{present}\\
\quad\quad\quad\quad \text{then } \mathcal{UL}\,(e_2,\ldots,e_n)\,(t_1,t_2,\ldots,t_m)\,t\,p \text{ present } d\\
\quad\quad\quad\quad \text{else ( } \mathcal{U}[\![e_1]\!]\,t_1\,d \textbf{ true } x,\\
\quad\quad\quad\quad\quad\quad \mathcal{UL}\,(e_2,\ldots,e_n)\,(t_1,t_2,\ldots,t_m)\,t\,p \text{ absent } x \text{ )}\\
\quad\quad\quad \text{else ( if } s = \text{empty}\\
\quad\quad\quad\quad\quad \text{then } \textbf{delete node } d \text{ else ()},\\
\quad\quad\quad\quad \mathcal{UL}\,(e_1,e_2,\ldots,e_n)\,(t_2,\ldots,t_m)\,t\,p \text{ empty } x \text{ )}\\
\quad\quad \text{else if } e_1 : t_1 \ \wedge\ s = \text{empty}\\
\quad\quad\quad \text{then ( } \mathcal{U}[\![e_1]\!]\,t_1\,d \textbf{ false } x,\\
\quad\quad\quad\quad \mathcal{UL}\,(e_2,\ldots,e_n)\,(t_2,\ldots,t_m)\,t\,p \text{ empty } d \text{ )}\\
\quad\quad\quad \text{else } \bot\\
\mathcal{UL}\,(\ldots)\,[\,]\,t\,p\,s\,x \ = \ \bot
\end{array}
$$

Rule (X-5) calls $\mathcal{UL}$ with $(e_1,\ldots,e_n)$ and $t=(t_1,\ldots,t_m)$. These two lists are traversed by $\mathcal{UL}$ recursively, until one of the two lists become empty. The path to extract the $t_i$ sequence component from the current path $p$ is calculated using the discriminator $\text{desc}[\![t_i,t]\!](p)$, which is similar to the split function in Figure 1. Given that $p$ has type $t$ (a sequence) and given a component of this sequence $t_i$, $\text{desc}[\![t_i,t]\!](p)$ returns an expression of type $t_i$ that extracts the $t_i$ component of $t$. In the $\mathcal{UL}$ definition, the components of the expression $(e_1,\ldots,e_n)$ must have types from $t_1,\ldots,t_m$, in that order but with some types repeated or missing. If some $e_i$ matches a discriminator, then this value is ignored since it copies the input to the output. But, if it repeats the discriminator value, then all the copies except the first must be inserted in the new view. If no $e_i$ matches a discriminator for some type, then this value must be removed from the view. This information is carried out using the state $s$, which may be empty, present, or absent, indicating that we have not found a new value for $t_1$, we have found at least one and it was a discriminating value, or we have found at least one but none was a discriminating value. The parameter $x$ is the last view component found so that new values are inserted after $x$.

For example, suppose that the view type $t$ is

$t$ = element A { $t_1$, $t_2*$ }
$t_1$ = element B xs:int  $t_2$ = element C { $t_3$, $t_4$ }
$t_3$ = element E xs:int  $t_4$ = element D xs:int

and the view mapping is:

```
<A>{ d/B,
     for $v in d/C
     return if $v/D=1
        then <C>{ <E>{$v/E+1}</E>, $v/D }</C>
        else $v }</A>
```

Then, using Rule (X-2), we get:

$\mathcal{U}[\![$<A>{ d/B, for ... }</A>$]\!]$ $t$ d true $\bot$
= $\mathcal{U}[\![$( d/B, for ... )$]\!]$ $(t_1, t_2*)$ (d/self::A/node()) true $\bot$

The discriminator $\text{desc}[\![t_1,(t_1,t_2*)]\!]$(d/self::A/node()) is equal to d/self::A/node()/self::B = d/self::A/B, which is equivalent to d/B. Thus, based on the $\mathcal{UL}$ algorithm, the first sequence component d/B is skipped. For the second sequence component, $\text{desc}[\![t_2*,(t_1,t_2*)]\!]$ (d/self::A/node()) is d/self::A/C. Therefore, based on Rules (X-4), (X-3), and (X-1), we have:

$\mathcal{U}[\![$for $v in d/C return if $v/D=1 then ... else $v$]\!]$ true $\bot$
  $(t_2*)$ (d/self::A/C)
= for $v in d/C
  return ($\mathcal{U}[\![$if $v/D=1 then ... else $v$]\!]$ $t_2$ $v true $\bot$)
= for $v in d/C
  return (if $v/D=1 then $\mathcal{U}[\![...]\!]$ $t_2$ $v true $\bot$ else ())

Using Rule (X-2), the for-loop body is mapped to:

$\mathcal{U}[\![$<C>{<E>{$v/E+1}</E>,$v/D}</C>$]\!]$ $t_2$ $v true $\bot$
= $\mathcal{U}[\![$( <E>{$v/E+1}</E>, $v/D )$]\!]$ $(t_3, t_4)$
  ($v/self::C/node()) true $\bot$

Based on the $\mathcal{UL}$ algorithm, the second sequence component is ignored (since $v/D is equivalent to $v/self::C/node()/self::D), while the first is mapped to an insertion followed by a deletion. Thus, the view is mapped to the following XUF code:

```
for $v in d/C
return if $v/D=1
       then (insert node <E>{$v/E+1}</E> as first
                    into ($v/self::C/E/..),
             delete node ($v/self::C/E) ) else ()
```

which is a replace, based on snapshot semantics.

# 8. IMPLEMENTATION AND EVALUATION

The algorithms described in this paper have already been implemented in Haskell and the presented examples have been tested. The source code is available at:

http://lambda.uta.edu/MaterializedViews.hs

Although this code has not been incorporated into our XQuery DBMS, HXQ [7], we have used HXQ to evaluate the feasibility of our approach on incremental view maintenance.

HXQ is a fast and space-efficient translator from XQuery to embedded Haskell code. It takes full advantage of Haskell's lazy evaluation to keep in memory only those parts of XML data needed at each point of evaluation, thus performing stream-based evaluation for forward queries. In addition to processing XML files, HXQ can store XML documents in a relational database (currently SQLite or MySQL through ODBC), by shredding XML into relational tuples,

| view # | size (MBs) | tuples ×1000 | view recreation time (secs) | view update time (msecs) |
|---|---|---|---|---|
| 0 | 14 | 130 | 164 | 23 |
| 1 | 29 | 290 | 191 | 22 |
| 2 | 47 | 470 | 233 | 27 |
| 3 | 70 | 380 | 215 | 20 |
| 4 | 98 | 540 | 256 | 19 |
| 5 | 130 | 730 | 310 | 21 |
| 6 | 162 | 910 | 371 | 24 |
| 7 | 185 | 1040 | 416 | 32 |
| 8 | 188 | 1060 | 423 | 33 |
| 9 | 191 | 1070 | 429 | 42 |
| 10 | 194 | 1090 | 439 | 39 |
| 11 | 198 | 1110 | 448 | 47 |
| 12 | 203 | 1140 | 459 | 53 |
| 13 | 209 | 1170 | 472 | 69 |
| 14 | 216 | 1210 | 491 | 68 |
| 15 | 223 | 1250 | 511 | 75 |
| 16 | 230 | 1290 | 530 | 72 |
| 17 | 239 | 1350 | 556 | 78 |
| 18 | 249 | 1410 | 588 | 80 |
| 19 | 262 | 1490 | 625 | 90 |

**Figure 3: Evaluation over a Number of XML Views**

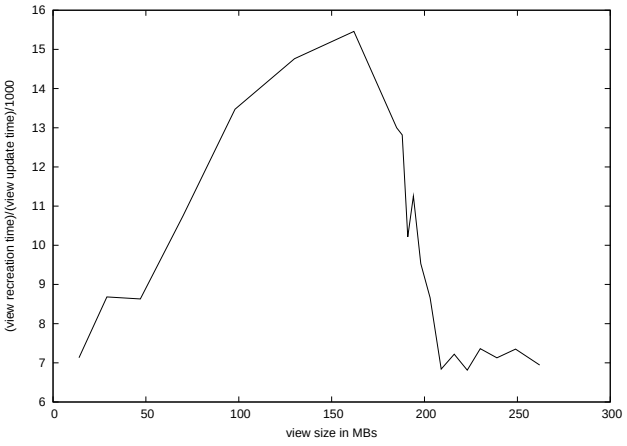


**Figure 4: View Update vs. View Recreation**

and by translating XQueries over the shredded documents into embedded optimized SQL queries. The mapping to relational tables is based on the document's structural summary, which is derived from the document data rather than from a schema. It uses hybrid inlining to inline attributes and non-repeating elements into a single table, thus resulting to a compact relational schema. For each such mapping, HXQ synthesizes an XQuery that reconstructs the original XML document from the shredded data. This XQuery is fused with the user queries and updates using partial evaluation techniques and parts of the resulting query are mapped to SQL queries.

Our experimental setup consists of a 2.2GHz Intel Core 2 Duo with 2GB memory, running Haskell ghc-6.10.3 on a 32-bit Linux 2.6.27 kernel. Our experimental XML data were derived from the DBLP XML document [6] of size 420MB. We fed this document into our HXQ DBMS, which shredded and stored the DBLP data into the MySQL database. The resulting MySQL database consists of 4.3 million tuples. Then, we created 20 materialized XML views of this database, $\text{view}_i$, $i \in [0, 19]$, using the following view mappings:

$\text{view}_i$ =
<dblp>{$DB//inproceedings[year **mod** 20 <= $i$]}</dblp>

that is, each $\text{view}_i$ had progressively more elements, with $\text{view}_{19}$ the largest that contained all inproceedings records of DBLP. Each XML view was materialized into a relational database in MySQL. More specifically, the results of each view mapping was dumped to an XML document, and this document was parsed, shredded, and stored by HXQ into MySQL automatically. Then, for each view, $\text{view}_i$, we evaluated the following XQuery update:

$U_i$ =
**for** $x **in** $\text{view}_i$ // inproceedings [ author='Leonidas Fegaras' ]
**return replace value of node** $x/year **with** 2000

Figure 3 shows our results for each $\text{view}_i$. The second column is the XML document size in MBs generated for each view, while the third column is the number of tuples (in thousands) in the materialized view. The fourth column is the time in seconds needed to create the materialized view from scratch by 1) evaluating the view query against the base relational database, 2) by dumping the view results to an XML document, and 3) by shredding and storing the resulting document into the materialized view. This would also be the time needed to recreate the view after each update. The fifth column is the time in milliseconds needed for each update $U_i$ against the materialized view. Figure 4 draws the ratio between the fourth and fifth column data (divided by 1000) against the second column (view size). We can see that the performance gain of updating the materialized view against recreating the view is between 7,000 and 15,000 times.

# 9. CONCLUSION

At a first glance, it may seem counter-intuitive to transform an efficient program with updates to an inefficient program that reconstructs the entire database, just to translate it back to updates at the end. But, as we show in this paper, this is a very effective way to incorporate source updates into a view, without requiring any substantial modification to the existing XQuery model. Since it is based on compositional transformations that are easy to validate, our approach has the potential of becoming a general methodology for incremental view maintenance.

# 10. REFERENCES

[1] S. Abiteboul, P. Bourhis, and B. Marinoiu. Efficient Maintenance Techniques for Views over Active Documents. In *EDBT'09*.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB'98*.

[3] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured Materialized Views for XML Queries. In *VLDB'07*.

[4] M. Benedikt and J. Cheney Schema-based Independence Analysis of XML Updates. In *VLDB'09*.

[5] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational Lenses: A Language for Updatable Views. In *PODS'06*.

[6] DBLP XML records, the DBLP Computer Science Bibliography. Available at `http://dblp.uni-trier.de/xml/`.

[7] L. Fegaras. Propagating Updates through XML Views using Lineage Tracing. In *ICDE'10*.
[8] L. Fegaras. A Schema-Based Translation of XQuery Updates. In *XSYM'10*. Available at `http://lambda.uta.edu/xuf10.pdf`.
[9] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. In *TODS'02*, 27(4).
[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] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Bulletin on Data Engineering*, 18(2), 1995.
[13] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views. In *WIDM'02*.
[14] D. Liu, Z. Hu, and M. Takeichi. Bidirectional Interpretation of XQuery. A Case Study towards Bidirectionalizing Functional Languages. In *PEPM'07*.
[15] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. In *TODS'06*, 31(3).
[16] N. Onose, V. R. Borkar, and M. Carey. Inverse Functions in the AquaLogic Data Services Platform. In *VLDB'07*.
[17] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. El-Abbadi, and K. S. Candan. Maintaining XPath Views in Loosely Coupled Systems. In *VLDB'06*.
[18] J. Voigtlander. Bidirectionalization for Free!. In *POPL'09*.
[19] W3C. XML Schema. `http://www.w3.org/XML/Schema`, 2000.
[20] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/xquery-semantics/`, 2007.
[21] W3C. XQuery Update Facility 1.0. W3C Candidate Recommendation 1. `http://www.w3.org/TR/xquery-update-10/`, June 2009.

# APPENDIX

# A. A CORRECTNESS PROOF OF THE RIGHT-INVERSE RULES

THEOREM 1. *The $\mathcal{I}_x$ algorithm given in Section 5.2 generates XQuery code that satisfies the right-inverse property.*

PROOF. We will prove that, given an XQuery expression $f(x)$ that depends on the variable $x$, the right-inverse $\mathcal{I}_x(f(x), y)$ is an XQuery expression $g(y)$ that depends on $y$, such that $x = g(y) \Rightarrow y = f(x)$. That is:

$$\forall f(x)\ \exists g(y) = \mathcal{I}_x(f(x), y) : \\ x = g(y) \Rightarrow y = f(x) \qquad \text{(rinv)}$$

Theorem (rinv) implies that $f(g(y)) = y$, which means that $g(y)$ (that is, $\mathcal{I}_x(f(x), y)$) is the right-inverse of $f(x)$, which proves Equation 1. From the algorithm in Section 5.2, we can easily prove that, if $x = g(y)$, then $y$ has the same type as $f(y)$. We will prove Equation (rinv) using structural induction over the structure of the XQuery expression $f(x)$. We will prove the validity of the Rules (I-9), (I-7), (I-10), and (I-12) only. The other rules can be proved in the same way.

Rule (I-9): We will prove that $x = \mathcal{I}_x(e, y/\text{self::}A/\text{node}()) \Rightarrow y = <A>\{e\}</A>$. The induction hypothesis for $h(z) = \mathcal{I}_x(e, z)$ is $x = h(z) \Rightarrow z = e$. Thus, the premise is $x = \mathcal{I}_x(e, y/\text{self::}A/\text{node}()) = h(y/\text{self::}A/\text{node}()) \Rightarrow y/\text{self::}A/\text{node}() = e$. Hence, we have $<A>\{e\}</A> = <A>\{y/\text{self::}A/\text{node}()\}</A> = y/\text{self::}A = y$, since $y$ has the same type as $<A>\{e\}</A>$.

Rule (I-7): Let the type of $e$ be element $B\ \{\ \ldots,$ element $A\ t,\ \ldots\ \}$. We will prove that $x = \mathcal{I}_x(e, <B>\{\ldots, y/\text{self::}A, \ldots\}</B>) \Rightarrow y = e/A$. The induction hypothesis for $h(z) = \mathcal{I}_x(e, z)$ is $x = h(z) \Rightarrow z = e$. Then, $x = \mathcal{I}_x(e, <B>\{\ldots, y/\text{self::}A, \ldots\}</B>) = h(<B>\{\ldots, y/\text{self::}A, \ldots\}</B>)$, which implies that $<B>\{\ldots, y/\text{self::}A, \ldots\}</B> = e$. Therefore, we have $e/A = (<B>\{\ldots, y/\text{self::}A, \ldots\}</B>)/A = y/\text{self::}A = y$, since $y$ has the same type as $e/A$.

Rule (I-10): The induction hypothesis for $h_1(w) = \mathcal{I}_x(e_1, w)$ is $x = h_1(w) \Rightarrow w = e_1$ and the induction hypothesis for $h_2(z) = \mathcal{I}_v(e_2, z)$ is $v = h_2(z) \Rightarrow z = e_2$. We will prove that $x = \mathcal{I}_x(e_1, \textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{return}\ \mathcal{I}_v(e_2, \$v')) \Rightarrow y = \textbf{for}\ \$v\ \textbf{in}\ e_1\ \textbf{return}\ e_2$. The premise is $x = \mathcal{I}_x(e_1, \textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{return}\ \mathcal{I}_v(e_2, \$v')) = h_1(\textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{return}\ h_2(\$v')) \Rightarrow \textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{return}\ h_2(\$v') = e_1$. That is, $\textbf{for}\ \$v\ \textbf{in}\ e_1\ \textbf{return}\ e_2 = \textbf{for}\ \$v\ \textbf{in}\ (\textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{return}\ h_2(\$v'))\ \textbf{return}\ e_2 = \textbf{for}\ \$v'\ \textbf{in}\ y,\ \$v\ \textbf{in}\ h_2(\$v')\ \textbf{return}\ e_2 = \textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{let}\ \$v\ :=\ h_2(\$v')\ \textbf{return}\ e_2 = \textbf{for}\ \$v'\ \textbf{in}\ y\ \textbf{return}\ \$v' = y$, since for $v = h_2(z)$, $z = e_2$.

Rule (I-12): We have two induction hypotheses: $h_1(z_1) = \mathcal{I}_x(e_1, z_1)$ implies $x = h_1(z_1) \Rightarrow z_1 = e_1$ and $h_2(z_2) = \mathcal{I}_x(e_2, z_2)$ implies $x = h_2(z_2) \Rightarrow z_2 = e_2$. Let $e_1 : t_1$ and $e_2 : t_2$, and $p_1$ and $p_2$ be predicates so that $y = (y\textbf{[}p_1\textbf{]}, y\textbf{[}p_2\textbf{]})$ and $y\textbf{[}p_1\textbf{]} : t_1$ and $y\textbf{[}p_2\textbf{]} : t_2$. It is easy to prove that $\text{match}(a, b) = x \Rightarrow a = x \wedge b = x$. From the premise, we have $x = \text{match}(\mathcal{I}_x(e_1, y\textbf{[}p_1\textbf{]}), \mathcal{I}_x(e_2, y\textbf{[}p_2\textbf{]})) = \text{match}(h_1(y\textbf{[}p_1\textbf{]}), h_2(y\textbf{[}p_2\textbf{]})) \Rightarrow h_1(y\textbf{[}p_1\textbf{]}) = x \wedge h_2(y\textbf{[}p_2\textbf{]}) = x \Rightarrow y\textbf{[}p_1\textbf{]} = e_1 \wedge y\textbf{[}p_2\textbf{]} = e_2$. That is, $(e_1, e_2) = (y\textbf{[}p_1\textbf{]}, y\textbf{[}p_2\textbf{]}) = y$. □