On the Expressive Power of Node Construction in XQuery

Wim Le Page Jan Hidders Philippe Michiels* Jan Paredaens Roel Vercammen*
University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium
{wim.lepage, jan.hidders, philippe.michiels, jan.paredaens, roel.vercammen}@ua.ac.be

ABSTRACT

In the relational model it has been shown that the flat relational algebra has the same expressive power as the nested relational algebra, as far as queries over flat relations and with flat results are concerned [6]. Hence, for each query that uses the nested relational model and that, with a flat table as input always has a flat table as output, there exists an equivalent flat query that only uses the flat relational model. In analogy, we study a related flat-flat problem for XQuery: for each expression containing operations that construct new nodes and whose XML result contains only original nodes, there exists an equivalent "flat" expression in XQuery that does not construct new nodes.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management; H.2.3 [**Database Management**]: Languages—*Query languages*

Keywords

XQuery,XML,Expressive power

1. INTRODUCTION

As XQuery [1] is becoming the standard language for querying XML documents, it is important to study the properties of this powerful query language. In XQuery, a query can have a result containing nodes not occurring in the input. These new nodes are constructed during the evaluation of the expression. Nevertheless, it is still possible that only original nodes occur in the final result. We call such expressions *node-conservative*. For example, the query in Example 1.1 creates new nodes not occuring in the result. In this example we perform a join and a projection of two XML documents in XQuery.

Example 1.1 Node-Conservative Expression

The following XQuery expression

```
let $jointtable :=
        element {"table"}{
            for $b1 in doc("table.xml")/table/row
            for $b2 in doc("table2.xml")/table/row
            where $b1/a = $b2/a
            return element{"row"}{$b1/*,$b2/*} }
return
    for $b in $jointtable/row/b return string($b)
```

has the result sequence "one","two"when given the input documents table.xml and table2.xml which look as follows

```
<table>                               <table>
<row><a>1</a><b>one</b></row>         <row><a>1</a><c>red</c></row>
<row><a>2</a><b>two</b></row>         <row><a>2</a><c>blue</c></row>
<row><a>3</a><b>three</b></row>       </table>
</table>
```

In this paper we show that for each deterministic node-conservative expression there exists an expression without node construction that essentially always returns the same store and result sequence. For example, the query in Example 1.1 can be rewritten to the query shown in Example 1.2. In this work we will show how to generate automatically equivalent constructor-free expressions for node-conservative expressions. This result gives an indication of the expressive power of the node construction. Furthermore it can be interesting for query optimization, since optimizing node construction can be hard. For example, in [3] a translation fom a subset if XQuery to SQL is given, where the construction of new elements yields SQL statements with special numbering operations which are relatively hard to optimize.

Example 1.2 Constructor-Free Expression

The following XQuery expression

```
for $b1 in doc("table.xml")/table/row
for $b2 in doc("table2.xml")/table/row
where $b1/a = $b2/a
return
   for $b in ($b1/*, $b2/*)/b return string($b)
```

is equivalent to the query of Example 1.1 and does not contain node constructors.

The work in this paper was inspired by simular results for the nested relational algebra [6, 7]. In [6] it is shown that each nested algebra expression that has a flat relation as output when applied to a flat relation, is equivalent to a flat algebra expression. In [7] a very direct proof is given of this fact using a simulation technique. Other work studied the effect of adding object creation to query languages on the expressive power of these languages. For example, in [2] the effect of object identity on the power of query

*Philippe Michiels and Roel Vercammen are supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant numbers 31016 and 31581.

Copyright is held by the author/owner. Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

languages is studied and a notion of determinate transformations is introduced as a generalization of the standard domain-preserving transformations. However, obvious extensions of complete database programming languages with object creation are not complete for determinate transformations. In [8] this mismatch is solved by introducing the notion of constructive transformations, a special kind of determinate transformations which are precisely the transformations that can be expressed by these obvious extensions.

This paper is structured as follows. In Section 2 we discuss LiXQuery, which we will use as a formal model for XQuery and for proving our theorems. Section 3 contains the theorem and proof for the elimination of node construction in expressions that do not contain newly constructed nodes in their results. Finally, the conclusion of this work is presented in Section 4.

2. LIXQUERY

We use LiXQuery [4, 5] as a basis for studying the expressive power of node construction in XQuery. LiXQuery is a sublanguage of XQuery that has a semantics that is consistent with that of XQuery, has the same expressive power as XQuery and has a compact and well defined syntax and semantics. The LiXQuery language was designed with the audience of researchers investigating the expressive power of XQuery in mind. The XQuery features that are omitted in LiXQuery are only those that are not essential from a theoretical perspective. We claim that the results that we show for LiXQuery also hold for XQuery.

LiXQuery has only a few built-in functions and no primitive data-types, order by clause, namespaces, comments, programming instructions and entities. Furthermore it ignores typing and only provides `descendant-or-self` and `child` as navigational axes. The other navigational axes can be simulated using these 2 axes. Although the features that LiXQuery lacks, are important for practical purposes, they are not relevant to our problem. Note that LiXQuery does support recursive functions, positional predicates and atomic values, which are essential in our approach.

We define LQE as the set of LiXQuery expressions. In LiXQuery, *expressions* are evaluated against an *XML store* and an *evaluation environment*. The XML store contains the fragments that are created as intermediate results, as well as the entire web. The store that only contains the entire web is called the *initial XML store*. The evaluation environment essentially contains mapping information for function names, variable names and the context item (including context position in the context sequence and the context sequence size). Formally, the XML store is a 6-tuple[1] $St = (V, E, \ll, \nu, \sigma, \delta)$ where: V is the set of available nodes; (V, E) forms an acyclic directed graph to represent the tree-structures; $\ll$ defines a total order over the nodes in V; ν labels element and attribute nodes with their node name; σ labels the attribute and text nodes with their string value; δ is a partial function that uniquely associates with an URI or a file name, a document node.

[1]This tuple is the same as in [5] except that the sibling order $<$ is replaced by the document order $\ll$.

The environment in LiXQuery is denoted by a tuple $Env = (a, b, v, x, k, m)$ where a is a partial function that maps a function name to its formal argument; b is a partial function that maps a function name to the body of the function; v is a partial function that maps variable names to their values; x is an item of St and indicates the context item or x is undefined; k is an integer denoting the position of the context item in the context sequence or k is undefined; m is an integer denoting the size of the context sequence, or m is undefined.

The result of an expression evaluated against an XML store and environment is a (possibly expanded) XML store (*result store*) and a sequence of one or more items over the result store (*result sequence*). Items in the result sequence can either be atomic values or nodes. The semantics of a LiXQuery expression is defined by statements of the form $St, Env \vdash e \Rightarrow St', v$, which state that when e is evaluated against a store St and an environment Env then St' is the result store and v is the result sequence over St. We derive such statement by using inference rules, which are given in [5].

We denote the empty sequence by $\langle\rangle$, non-empty sequences by, for example, $\langle 1, 2, 3\rangle$ and the concatenation of two sequences l_1 and l_2 by $l_1 \circ l_2$. Last but not least, each node has a unique identity. It is important to note that atomic values do not have an identity.

3. ELIMINATING NODE CONSTRUCTION

We will show that some XQuery expressions that contain node constructors can be simulated by another XQuery expression that does not use node construction.

3.1 Node Conservative Expressions

Clearly an expression cannot be simulated by an expression without constructors if it returns newly created nodes, so we introduce the notion of *node conservative expressions*.

Definition 1. A *node-conservative expression* (NCE) is an expression $e \in LQE$ such that for all stores St and environments Env it holds that if $St, Env \vdash e \Rightarrow St', v$ then all nodes in v are nodes in St.

In Example 1.1 we considered a join and a projection of two XML documents in LiXQuery. This expression is an example of a NCE.

Another restriction we make is that we only consider deterministic expressions. Node creation is a source of nondeterminism in LiXQuery (and XQuery) because the fragment that is created by a constructor is placed at an arbitrary position in document order between the already existing trees in the store. Since node construction is the only source of non-determinism in LiXQuery, it is clear that we cannot simulate that there are many possible results without it. This is however not a fundamental feature of XQuery so we ignore non-deterministic expressions.

Definition 2. An expression $e \in LQE$ is said to be *deterministic* if for every store St and environment Env it holds that if $St, Env \vdash e \Rightarrow St', v$ and $St, Env \vdash e \Rightarrow St'', w$ then $v = w$.

Note that this is a very strict definition of determinism which, in fact, only allows node-conservative expressions. We could have allowed multiple results that were equivalent up to isomorphism over the nodes, but this would make things unnecessarily complex.

Next to restricting the types of expressions we consider we also allow a simulation to differ in its semantics from the the original in two ways. The first is that a simulation may have a defined result where the original does not. Note that we still require that whenever an expression has a defined result then the simulation has the same defined result, but not necessarily the reverse. We conjecture that the theorem also holds when we also require the reverse but proving this would add a lot of overhead to this paper without adding much extra insight in the expressive power of node construction.

The second way in which the semantics of a simulation differs from that of the original is that resulting stores only have to be the same up to garbage collection, i.e., after removing the trees that are not reachable by the δ function (the `fn:doc()` function) or contain nodes from the result sequence. If we denote the store that results from garbage collection on a store St and a result sequence v as $[St]_v$ then this leads to the following definition:

Definition 3. Given two expression $e, e' \in LQE$ we say that e' is a *simulation of* e if for all stores St and environments Env with undefined x, k and m it holds that if $St, Env \vdash e \Rightarrow St', v$ then there exists a store St'' such that $St, Env \vdash e' \Rightarrow St'', v$ and $[St'']_v = [St']_v$.

We use this definition for the following theorem, which is the main result of this paper:

THEOREM 1. *For every deterministic node-conservative*[2] *expression* $e \in LQE$ *there exists a simulation* $e' \in LQE$ *that does not contain constructors.*

3.2 Outline of the Simulation

Our goal is to transform an expression into a semi-equivalent expression which does not use node constructions. We are going to eliminate the construction by simulating it. To simulate construction we will need to simulate the store, because it is there that the information concerning the newly constructed nodes will reside. In short, the simulation performs the following steps:

1. We use a few special variables in the environment to encode a part of the store. This part will contain the newly created nodes but also parts of the old store that are retrieved with the `doc()` function;
2. Whenever a `doc()` call occurs in the original expression, the simulation will add the encoding of the document tree to the simulated store on the condition that it is not already there;
3. Accessing nodes in the store is simulated by accessing the encoded store;
4. Nodes are simulated by node identifiers which are numbers that refer to the encoded nodes in the store;
5. In order to be able to distinguish encoded atomic values from node identifiers within sequences, we let the normal atomic values be preceded by a `0` and the node identifiers by a `1`. Note that this means that in the simulation, a sequence will be twice as long and every item that was at position i will now be at position $2i$;
6. Finally, the simulation replaces the node identifiers with the corresponding nodes from the store. If the original expression is indeed a deterministic node conservative expression, the result – and thus also the result of the simulation – will contain no newly constructed nodes. Consequently, this last step is always possible if the original expression is node conservative.

[2]Since every deterministic expression is also node-conservative we can strictly speaking drop the second requirement.

The transformation of an expression to a constructor-free expression that simulates it, is expressed by a transformation function. A transformation function is a function $\epsilon : LQE \rightarrow LQE$. The commuting diagram in Figure 1 illustrates what should hold for such a transformation function ϵ for it to be correct. We show this by induction on the subexpressions e'' of an expression e.

$$\begin{array}{ccc} (St, Env) & \xrightarrow{\tau} & (\widehat{St}, \widehat{Env}) \\ e'' \downarrow & & \downarrow \epsilon(e'') \\ (St', v) & \xrightarrow{\tau'} & (\widehat{St}, \widehat{v}) \end{array}$$

Figure 1: This diagram depicts the relations between the several components in the translation.

On the left-hand side we see that starting from a store St and an environment Env, the evaluation of the expression e'', which may add new nodes to St, will result in a new store $St' \supseteq St$ and a result v. On the right-hand side we see that starting from a store $\widehat{St}$ and an environment $\widehat{Env}$, the evaluation of the transformed constructor-free expression $\epsilon(e'')$, which will not add new nodes to $\widehat{St}$, will result in the same store $\widehat{St}$ and a result $\widehat{v}$.

At the top of the diagram we see the encoding τ which encodes a store $St_{\widehat{Env}} \subseteq St$ into sequences of atomic values that are bound to special variables in the environment $\widehat{Env}$. Moreover, τ replaces the values of all variables in Env with sequences of atomic values and the bodies of all functions are transformed by ϵ to constructor-free expressions. At the bottom of the diagram we see the encoding τ' which encodes a store $St_{\widehat{v}} \subseteq St'$ and the value v as a sequence of atomic values $\widehat{v}$.

When we use this schema to show by induction that we can correctly translate an expression e to a constructor-free expression $\epsilon(e)$ it will hold for the evaluation of the subexpression e'' that $\widehat{St}$ is the store against which e is evaluated. Moreover, if during the evaluation of e nodes where created before the evaluation of e'' then (1) these nodes have been added to St and (2) in the evaluation of

$\epsilon(e)$ they were added to the encoded store in $\widehat{Env}$. So it will hold that $St = \widehat{St} \cup St_{\widehat{Env}}$. Obviously it has to be shown by induction that this remains true after the evaluation of e'' so it has to be shown that $St' = \widehat{St} \cup St_{\widehat{v}}$. An overview of all these relationships between the involved stores is illustrated in Figure 2.

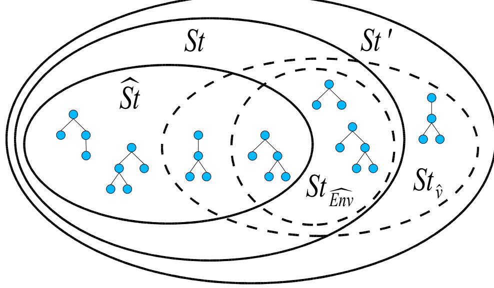

Figure 2: The stores $\widehat{St}$, $St_{\widehat{Env}}$, St, $St_{\widehat{v}}$ and St'

3.3 Encoding the Store and Environment

Before we describe how to translate LiXQuery expressions into their constructor-less simulations, we first have to look into the encodings of the store and environment based on their formal semantics.

We first describe how to encode a store in sequences of atomic values. We will define this given an injective function $id : V \to \mathbb{N}$ that provides the unique node identifier for each node and which will be used to represent the nodes in the encoding.

Definition 4. Given an XML store $St = (V, E, \ll, \nu, \sigma, \delta)$ and an injective function $id : V \to \mathbb{N}$ then we call a tuple of XML values $(\hat{V}, \hat{E}, \hat{\delta})$ a *store encoding of St under id* if

- $\hat{V} = \langle id(v_1), t_1, n_1, s_1 \rangle \circ \ldots \circ \langle id(v_k), t_k, n_k, s_k \rangle$ where (1) $\{v_1, \ldots, v_k\} = V$, (2) $v_1 \ll \ldots \ll v_k$, (3) t_i equals `"text"`, `"doc"`, `"attr"` or `"elem"` if v_i is a text node, a document node, an attribute node or an element node, respectively, (4) n_k is $\nu(v_k)$ if it is defined and `""` otherwise, and (5) s_k is $\sigma(v_k)$ if it is defined and `""` otherwise,
- $\hat{E} = \langle id(v_1), id(v'_1) \rangle \circ \ldots \circ \langle id(v_m), id(v'_m) \rangle$ where $\{(v_1, v'_1), \ldots, (v_m, v'_m)\} = E$,
- $\hat{\delta} = \langle s_1, id(v_1) \rangle \circ \ldots \circ \langle s_p, id(v_p) \rangle$ where $\delta = \{(s_1, v_1), \ldots, (s_p, v_p)\}$.

Note that a store encoding is not uniquely determined given St and id because we can choose the order in $\hat{E}$ and $\hat{\delta}$.

We have to encode sequences of atomic values and nodes as sequences of atomic values. When we directly replace each node v with $id(v)$ we cannot always tell if a number represents itself or encodes a node identifer. Therefore we let atomic values that encode themselves be preceded by `0` and atomic values that are node identifiers be preceded by `1`. For illustration consider the examples in Example 3.1.

Example 3.1 Encoded values

Given a function $id = \{(v_1, 5), (v_2, 3)\}$:

value	*value encoding*
$\langle 5 \rangle$	$\langle$`0,5`$\rangle$
$\langle v_1 \rangle$	$\langle$`1,5`$\rangle$
$\langle 5, v_1,$`"string"`$, v_2 \rangle$	$\langle$`0,5,1,5,0,"string",1,3`$\rangle$

Definition 5. Given an XML value $v = \langle x_1, \ldots, x_k \rangle$ over a store $St = (V, E, \ll, \nu, \sigma, \delta)$ and an injective function $id : V \to \mathbb{N}$, we call an XML value $\tilde{v}$ the *value encoding of v under id* if $\tilde{v} = \langle m_1, \hat{x}_1 \rangle \circ \ldots \circ \langle m_k, \hat{x}_k \rangle$ where $m_i = 1$ and $\hat{x}_i = id(x_k)$ if x_k is a node and $m_i = 0$ and $\hat{x}_i = x_i$ otherwise.

Note that the encoding of value v is written as $\tilde{v}$ and not as $\hat{v}$ to distinguish it from the $\hat{v}$ in the commuting diagram in Figure 1 which encodes both a store and a value.

We now proceed with formalizing the the τ relationship in Figure 1. Recall that the relations in this diagram hold by induction on the subexpressions e'' of a simulated expression e. The resulting store $\widehat{St}$ is the store against which e is evaluated, because all nodes that are created by e'' are in $\epsilon(e'')$ encoded in $\widehat{Env}$. We will refer to the part of St encoded in $\widehat{Env}$ as $St_{\widehat{Env}}$. Since $St_{\widehat{Env}}$ describes the part of St that is retrieved or created by preceding evaluations it holds that $St = \widehat{St} \cup St_{\widehat{Env}}$ where $\widehat{St} \cap St_{\widehat{Env}}$ contains the documents that were retrieved with the `doc()` function before e'' was evaluated (see Figure 2).

Definition 6. Given a store $St = (V, E, \ll, \nu, \sigma, \delta)$, an environment $Env = (a, b, v, x, k, m)$ over this store and a transformation function ϵ we call a pair $(\widehat{St}, \widehat{Env})$ with store $\widehat{St}$ and environment $\widehat{Env} = (\hat{a}, \hat{b}, \hat{v}, \hat{x}, \hat{k}, \hat{m})$ a *store-environment encoding of St and Env under tr* if there is a store $St_{\widehat{Env}}$ and an injective function $id : V_{\widehat{Env}} \to \mathbb{N}$ such that

- $St = \widehat{St} \cup St_{\widehat{Env}}$,
- all nodes in values of variables in Env are in $St_{\widehat{Env}}$
- $\hat{a} = a$,
- $\hat{b} = \{(s, tr(y)) | (s, y) \in b\}$,
- in $\hat{v}$ (1) all variable names s bound by v are bound to the value encoding of $v(s)$ under id, (2) the variables `tau:E`, `tau:V` and `tau:delta` contain $\hat{V}$, $\hat{E}$ and $\hat{\delta}$, respectively, where $(\hat{V}, \hat{E}, \hat{\delta})$ is the store encoding of $St_{\widehat{Env}}$ under id and (3) the variables `tau:x`, `tau:k` and `tau:m` contain value encodings of x, k and m, respectively, under id, and
- $\hat{x}$, $\hat{k}$ and $\hat{m}$ are all undefined.

In turn, we now define the τ' encoding in Figure 1. Here we refer to the part of the store that is encoded in the environment as $St_{\widehat{v}}$. Since $St_{\widehat{v}}$ describes the part of St that is retrieved or created by preceding evaluations it must hold that $St' = \widehat{St} \cup St_{\widehat{v}}$ where $\widehat{St'} \cap St_{\widehat{v}}$ contains the documents that were retrieved with the `doc()` function before or during e'' was evaluated (see Figure 2).

Definition 7. Given a store $St' = (V, E, \ll, \nu, \sigma, \delta)$ and a value v over this store then a pair $(\widehat{St}, \widehat{v})$ with a store $\widehat{St}$ and an XML value $\widehat{v}$ is called a *store-value encoding of St and v* if there is a store $St_{\widehat{v}}$ and an injective function $id : V_{\widehat{v}} \to \mathbb{N}$ such that (1) $St' = \widehat{St} \cup St_{\widehat{v}}$, (2) all nodes in v are in $St_{\widehat{v}}$ and (3) $\hat{v} = \langle|V|\rangle \circ \hat{V} \circ \langle|E|\rangle \circ \hat{E} \circ \langle|\delta|\rangle \circ \hat{\delta} \circ \tilde{v}$ where $(\hat{V}, \hat{E}, \hat{\delta})$ is the store encoding of $St_{\widehat{v}}$ under id, and $\tilde{v}$ is the value encoding of v under id.

Based on this input/output encoding we can give the formal meaning of the diagram in Figure 1 and define when a transformation function defines a correct simulation.

Definition 8. A transformation function ϵ is said to be a *correct transformation* if it holds for every store St and environment Env that if $St, Env \vdash e \Rightarrow St', v$ and $(\widehat{St}, \widehat{Env})$ is store-environment encoding of St and Env under tr then it holds that $\widehat{St}, \widehat{Env} \vdash \epsilon(e) \Rightarrow \widehat{St}, \widehat{v}$ where $(\widehat{St}, \widehat{v})$ is a store-value encoding of St' and v.

3.4 A Correct Transformation Function

In this section we construct a transformation function $\epsilon : LQE \to LQE$ and show that the following theorem holds.

THEOREM 2. *The transformation function ϵ is a correct transformation function.*

The result of $\epsilon(e)$ is defined by induction upon the structure of e. Because of space limitations we will only show some typical translations for some types of LiXQuery expressions. Helper functions will be defined in the `eps` namespace which is assumed to be distinct from all the used namespaces in e.

We begin with the translation of the `name()` function. Here and in the following we will assume the existence of the functions `eps:V()`, `eps:E()`, `eps:delta()` and `eps:val()` which respectively extract $\widehat{V}$, $\widehat{E}$, $\widehat{\delta}$, and $\widetilde{v}$ from a store-value encoding. For computing the store-value encoding give $\hat{V}$, $\hat{E}$, $\hat{\delta}$ and $\tilde{v}$ we assume the existence of a function `eps:stValEnc()` with formal arguments `$V, $E, $delta` and `$val`. We also introduce the shorthand

```
let $eps:V, E, delta, val := getStVal($eps:res)
```

to denote

```
let $eps:V := eps:V($eps:res)
let $eps:E := eps:E($eps:res)
let $eps:delta := eps:delta($eps:res)
let $eps:val := eps:val($eps:res)
```

The translation of the `name()` function is defined as follows:

```
ϵ(name(e′)) =
  let $eps:res := ϵ(e′)
  let $eps:V, E, delta, val := getStVal($eps:res)
  return eps:epsStValEnc($eps:V, $eps:E, $eps:delta,
          eps:nu($eps:val[2], $eps:V) )
```

The function `eps:nu()` returns the name of the specified node using the information encoded in $\widehat{V}$.

The `doc()` function loads new documents into our encoded store.

```
ϵ(doc(e)) = let $eps:res := ϵ(e)
            return eps:doc($eps:res)
```

Here the function `eps:doc()` checks if the document is already in the encoded store by comparing the URI's tot the URI's already present in $\widehat{\delta}$. If this is the case it just returns the associated simulated node id as found in $\widehat{\delta}$, else the `eps:doc()` function compares the real document node obtained with the given URI, to the real documents obtained via the URI's that are already present in $\widehat{\delta}$. If this is the case, only a new entry is added to $\widehat{\delta}$ linking the new URI to the node identifier of the encoded document. If the document is not present in $\widehat{\delta}$ the document is encoded. First a document node is added to the encoded store and with the resulting node identifier a new entry is added in $\widehat{\delta}$. Then, also using this identifier, the nodes of the document are encoded and added after this document node in $\widehat{V}$. The `eps:doc()` function finally returns a store-value encoding containing the (new) node identifier as the result sequence and the (updated) store, environment and delta.

The for-expression is the most fundamental type of expression in LiXQuery. In it's translation we assume a number x that is unique for each for-expression that has to be translated. This is used to define for every for-expression a unique function `eps:for`$_x$`()`. The parameter $vars_x$ represent all free variables in e'. Recursion is used here to simulate the iteration over a sequence where the resulting store of the previous step is passed on to the following step. The translation of the for-expression is then defined as follows.

```
ϵ(for $s at $s′ in e return e′) =
  let $eps:res := ϵ(e)
  let $eps:V,E,delta,val := getStVal($eps:res)
  return eps:for_x(1, $eps:val, $eps:V, $eps:E,
      $eps:delta, vars_x)
```

with `eps:for`$_x$`()` defined as follows:

```
declare function eps:for_x($eps:pos, $eps:seq,
          $tau:V, $tau:E, $tau:delta, vars_x) {
  let $s := $eps:seq[$eps:pos*2-1], $eps:seq[$eps:pos*2]
  let $s' := $eps:pos
  let $eps:res1 := ϵ(e′)
  let $eps:V1,E1,delta1,val1 := getStVal($eps:res1)
  let $eps:res2 := eps:for_x($eps:pos+1, $eps:seq,
          $eps:V1, $eps:E1, $eps:delta1, vars_x)
  let $eps:V2,E2,delta2,val2 := getStVal($eps:res2)
  return $eps:stValEnc($eps:V2, $eps:E2, $eps:delta2,
            ( $eps:val1, $eps:val2 ))
}
```

The translation of node comparison expressions is done by extracting the information of identity and position contained in the store-value encoding.

```
ϵ(e' is e'') =
  let $eps:res := ϵ(e')
  let $tau:V,E,delta,val1 := getStVal($eps:res1)
  let $eps:res2 := ϵ(e'')
  let $tau:V2,E2,delta2,val2 := getStVal($eps:res2)
  return $eps:stValEnc( $tau:V2, $tau:E2, $tau:delta2,
          (0, $tau:val1[2] = $tau:val2[2]))
```

The translation of a construction operator extends the encoded store which is a crucial part of the simulation. To illustrate this we give the translation the element construction.

```
ϵ(element {e′}{e″}) =
  let $eps:res := ϵ(e′)
  let $tau:V,E,delta,val1 := getStVal($eps:res)
  let $eps:res2 := ϵ(e″)
  let $V2,E2,delta2,val2 := getStVal($eps:res2)
  return eps:addElem(V2, E2, delta2, $tau:val1, val2)
```

with `eps:addElem()` declared as follows.

```
declare function eps:addElem($V $E, $delta,
                 $nameEnc, $chEnc) {
  let $res1 := eps:addElemNode($nameEnc[2], $V, $E)
  let $V1,E1,delta1,val1 := getStVal($res1)
  let $res2 := eps:addChl($val1, $chEnc, V1, E1 )
  let $V2,E2,delta2,val2 := getStVal($res2)
  return $eps:stValEnc( $V2, $E2, $delta, $val1)
}
```

Here the function `eps:addElemNode($name, $V, $E)` adds a new element node with name `$name` and returns a store-value encoding with the new store and the new node identifier. The function `eps:addChl($parEnc, $chEnc, $V, $E)` makes deep copies for all the nodes encoded in `chEnc`, adds these under the node encoded in `$parEnc` and returns a store-value encoding with the new store and the parent node. The function uses recursion in the same way as the translation of the for-expression, in order to be able to iterate with side-effects on the store.

3.5 Creating a Constructor-Free Expression

We now sketch how to create constructor-free semi-equivalent expressions for deterministic (node-conservative) ones, i.e., how to generate the expression e' of Theorem 1, based on $\epsilon(e)$, which is working on an encoding of (St, Env). We do so by showing how encoding (St, Env) and afterwards decoding results St', v can be done for node-conservative expressions.

The expression $\epsilon(e)$ will be evaluated against $(St, \widehat{Env})$, where $\widehat{Env}$ contains the encoded store and environment. We construct $St_{\widehat{Env}}$ in such a way that it contains exactly all trees of St for which a node occurs in the variable bindings of Env. Assuming that we can have a sequence that is the concatenation of all variable bindings in Env, we can write an expression to create a new sequence `$roots` that, starting from the former sequence, filters out the nodes, applies the `root` function to each node and finally sorts this result by document order by applying a self-axis step. Since the roots of all trees that have to be in $St_{\widehat{Env}}$ are now in document order in `$roots`, we can write another expression that creates the encoded store $St_{\widehat{Env}}$ starting from an empty encoded store, by simply traversing through the trees under the nodes in `$roots` and extending $St_{\widehat{Env}} = (\widehat{V}, \widehat{E}, \widehat{\delta})$, represented by the variables `$tau:V`, `$tau:E` and `$tau:delta` in the environment $\widehat{Env}$). If this traversal is done in depth-first, left-to-right manner, we visit all nodes of St that will be encoded in $St_{\widehat{Env}}$ in document order. Node identifiers can then be chosen in such a way that they correspond to the position in $St_{\widehat{Env}}$. Starting from Env, we can now create the encoded environment $\widehat{Env}$ by replacing all expressions in b by the simulations $\epsilon(b)$, adding the variables for the encoded store and environment to the function signatures in a, replacing all sequences in the variable bindings with their encoded sequences, and finally, adding the variables `$tau:V`,`$tau:E` and `$tau:delta` to v. Since all nodes that occur in Env are encoded in $St_{\widehat{Env}}$ and node identifiers were assigned based on the position of nodes within the forest under `$roots`, we can easily obtain the encoded sequences for the variable bindings.

The result of the evaluation of $\epsilon(e)$ is the store St and a store-value encoding $St_{\widehat{v}}$. Based on this we can create the result sequence the original expression returned if it was a node-conservative expression. In that case the encoded result sequence will only contain encoded nodes of which the real counterparts were available in the initial XML store St. Therefore we can loop over encoded items in $St_{\widehat{v}}$. Encodings of atomic values are simply replaced by the atomic values itself. For every encoded node we first determine whether it was originally in $St_{\widehat{Env}}$. This can be done by storing (during the encoding phase) all nodes and their chosen node identifiers as pairs in a variable. If the node identifier occurs in this variable then it is an original node and we can easily return the corresponding node. If the root of the encoded node is an encoded document node that is associated to a URI in the variable `$tau:delta` then we can obtain the original document root node by a simple `doc` function call, else it is a newly created node and hence this expression is not a node-conservative expression. By using the position of the encoded node relative to the encoded root node, we can determine the position of the corresponding real node in the document tree and hence we replace the encoded node by the real node in the result sequence.

4. CONCLUSION

In this paper, we showed that deterministic XQuery expressions, always yielding a result with only nodes from the input store, can be rewritten to equivalent expressions that do not contain node constructors. In further research, we plan to investigate whether a similar result can also be obtained for non-recursive XQuery. Furthermore, we intend to investigate how this result can be used to optimize queries by removing or postponing node creation operations in query evaluation plans. Finally, we want to examine whether this result can be used for rewriting `let` expressions in non-recursive XQuery without using XQuery functions. This is not trivial, since simple variable substitution would result in multiple creation of the nodes on the right-hand side of the variable assignment.

5. REFERENCES

[1] XML query (XQuery). http://www.w3.org/XML/Query.

[2] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45:798–842, September 1998.

[3] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *Proceedings of the 30th Int'l Conference on Very Large Databases (VLDB 2004)*, August/September 2004 2004.

[4] J. Hidders, J. Paredaens, P. Michiels, and R. Vercammen. LiXQuery: A formal foundation for XQuery research. *SIGMOD Record*, September 2005.

[5] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *Proceedings of the Second International XML Database Symposium (XSym 2004)*, Toronto, Canada, 2004. Springer.

[6] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems (TODS)*, 17:65–93, 1992.

[7] J. Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254:363–377, 2001.

[8] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44:272–319, March 1997.