LiXQuery: A Formal Foundation for XQuery Research

Jan Hidders Philippe Michiels* Jan Paredaens Roel Vercammen*

University of Antwerp
Middelheimlaan 1
B-2020 Antwerp, Belgium
{jan.hidders, philippe.michiels, jan.paredaens,roel.vercammen}@ua.ac.be

ABSTRACT

XQuery is considered to become the standard query language for XML documents. However, the complete XQuery syntax and semantics seem too complicated for research and educational purposes. By defining a concise backwards compatible subset of XQuery with a complete formal description, we provide a practical foundation for XQuery research. We pay special attention to usability by supporting the most typical XQuery expressions.

1. MOTIVATION

XQuery's popularity for querying XML documents is growing rapidly[1, 13, 9, 11]. The main reason for this is the fact that it is generally believed to become the standard language for querying XML documents. Furthermore, XQuery is a powerful language in which many typical database queries can be written down very compactly compared to, for example, XSLT. This power however, comes at a price. It is very hard to define the full semantics of XQuery in an elegant and concise manner[3, 2, 4].

From an academic point of view, the need arises for a sublanguage of XQuery, which is nearly as powerful as the full language but has an elegant syntax and semantics that can be written down in just a couple of pages. Similar efforts have been made for XPath[18] and XSLT[7] and have played important roles in practical and theoretical research[10, 16, 15, 14].

The definition of such a language enables us to investigate certain aspects of the XQuery language like

- the power of recursion in XQuery and possible syntactical restrictions that allow us to control this power,
- the complexity of deciding query equivalence for purposes like query optimization,
- the functional character of XQuery, compared to functional programming languages like LISP, ML, et cetera
- the roles of XPath expressions inside XQuery in terms of expressive power and query optimization and
- the relationship between XQuery expressions and the classical well-understood concept of generic database queries.

In this paper we define LiXQuery, a sublanguage of XQuery that is usefull for educational and research purposes and that has several interesting properties, which are easy to prove and can be transposed to the full XQuery language.

The remainder of this paper is organized as follows. In Section 2 we discuss the syntax of LiXQuery and some of the important design choices we made. In Section 3 we give some examples to illustrate the LiXQuery syntax and semantics and Section 4 gives a short introduction to the formal semantics of LiXQuery. Finally, in Section 5 we will point out some interesting research directions that may benefit from this work.

2. SYNTAX AND DESIGN CHOICES

2.1 Syntax

The syntax of LiXQuery is given in Fig. 1 as an abstract syntax, i.e., it assumes that extra brackets and precedence rules are added for disambiguation.

All queries in LiXQuery are syntactically correct in XQuery and their LiXQuery semantics is consistent with those of the XQuery counterparts. Built-in functions for manipulation of basic values are omitted. The non-terminal $\langle Name \rangle$ refers to the set of names $\mathcal{N}$ which we will not describe in detail here except that the names are strings that must start with a letter or "_". The non-terminal $\langle String \rangle$ refers to strings that are enclosed in double quotes such as in `"abc"` and $\langle Integer \rangle$ refers to integers such as `100`, `+100`, and `-100`.[1] Therefore the sets associated with $\langle Name \rangle$, $\langle String \rangle$ and $\langle Integer \rangle$ are pairwise disjoint.

The ambiguity between rule [5] and [24] is resolved by giving precedence to [5], and for path expressions we will assume that the operators "`/`" and "`//`" (rule [18]) are

*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.

[1]Integers are the only numeric type that exists in LiXQuery.

[1] ⟨*Query*⟩ → (⟨*FunDef*⟩ "`;`")* ⟨*Expr*⟩
[2] ⟨*FunDef*⟩ → "`declare` `function`" ⟨*Name*⟩ "`(`" (⟨*Var*⟩ ("`,`" ⟨*Var*⟩)*)? "`)`" "`{`" ⟨*Expr*⟩ "`}`"
[3] ⟨*Expr*⟩ → ⟨*Var*⟩ | ⟨*BuiltIn*⟩ | ⟨*IfExpr*⟩ | ⟨*ForExpr*⟩ | ⟨*LetExpr*⟩ | ⟨*Concat*⟩ | ⟨*AndOr*⟩ | ⟨*ValCmp*⟩ | ⟨*NodeCmp*⟩ | ⟨*AddExpr*⟩ | ⟨*MultExpr*⟩ | ⟨*Union*⟩ | ⟨*Step*⟩ | ⟨*Filter*⟩ | ⟨*Path*⟩ | ⟨*Literal*⟩ | ⟨*EmpSeq*⟩ | ⟨*Constr*⟩ | ⟨*TypeSw*⟩ | ⟨*FunCall*⟩
[4] ⟨*Var*⟩ → "`$`" ⟨*Name*⟩
[5] ⟨*BuiltIn*⟩ → "`doc(`" ⟨*Expr*⟩ "`)`" | "`name(`" ⟨*Expr*⟩ "`)`" | "`string(`" ⟨*Expr*⟩ "`)`" | "`xs:integer(`" ⟨*Expr*⟩ "`)`" | "`root(`" ⟨*Expr*⟩ "`)`" | "`concat(`" ⟨*Expr*⟩, ⟨*Expr*⟩ "`)`" | "`true()`" | "`false()`" | "`not(`" ⟨*Expr*⟩ "`)`" | "`count(`" ⟨*Expr*⟩ "`)`" | "`position()`" | "`last()`"
[6] ⟨*IfExpr*⟩ → "`if (`" ⟨*Expr*⟩ "`)`" "`then`" ⟨*Expr*⟩ "`else`" ⟨*Expr*⟩
[7] ⟨*ForExpr*⟩ → "`for`" ⟨*Var*⟩ ("`at`" ⟨*Var*⟩)? "`in`" ⟨*Expr*⟩ "`return`" ⟨*Expr*⟩
[8] ⟨*LetExpr*⟩ → "`let`" ⟨*Var*⟩ "`:=`" ⟨*Expr*⟩ "`return`" ⟨*Expr*⟩
[9] ⟨*Concat*⟩ → ⟨*Expr*⟩ "`,`" ⟨*Expr*⟩
[10] ⟨*AndOr*⟩ → ⟨*Expr*⟩ ("`and`" | "`or`") ⟨*Expr*⟩
[11] ⟨*ValCmp*⟩ → ⟨*Expr*⟩ ("`=`" | "`<`") ⟨*Expr*⟩
[12] ⟨*NodeCmp*⟩ → ⟨*Expr*⟩ ("`is`" | "`<<`") ⟨*Expr*⟩
[13] ⟨*AddExpr*⟩ → ⟨*Expr*⟩ ("`+`" | "`-`") ⟨*Expr*⟩
[14] ⟨*MultExpr*⟩ → ⟨*Expr*⟩ ("`*`" | "`idiv`") ⟨*Expr*⟩
[15] ⟨*Union*⟩ → ⟨*Expr*⟩ "`|`" ⟨*Expr*⟩
[16] ⟨*Step*⟩ → "`.`" | "`..`" | ⟨*Name*⟩ | "`@`" ⟨*Name*⟩ | "`*`" | "`@*`" | "`text()`"
[17] ⟨*Filter*⟩ → ⟨*Expr*⟩ "`[`" ⟨*Expr*⟩ "`]`"
[18] ⟨*Path*⟩ → ⟨*Expr*⟩ ("`/`" | "`//`") ⟨*Expr*⟩
[19] ⟨*Literal*⟩ → ⟨*String*⟩ | ⟨*Integer*⟩
[20] ⟨*EmpSeq*⟩ → "`()`"
[21] ⟨*Constr*⟩ → "`element`" "`{`" ⟨*Expr*⟩ "`}`" "`{`" ⟨*Expr*⟩ "`}`" | "`attribute`" "`{`" ⟨*Expr*⟩ "`}`" "`{`" ⟨*Expr*⟩ "`}`" | "`text`" "`{`" ⟨*Expr*⟩ "`}`" | "`document`" "`{`" ⟨*Expr*⟩ "`}`"
[22] ⟨*TypeSw*⟩ → "`typeswitch (`" ⟨*Expr*⟩ "`)`" ("`case`" ⟨*Type*⟩ "`return`" ⟨*Expr*⟩)+ "`default`" "`return`" ⟨*Expr*⟩
[23] ⟨*Type*⟩ → "`xs:boolean`" | "`xs:integer`" | "`xs:string`" | "`element()`" | "`attribute()`" | "`text()`" | "`document-node()`"
[24] ⟨*FunCall*⟩ → ⟨*Name*⟩ "`(`" (⟨*Expr*⟩ ("`,`" ⟨*Expr*⟩)*)? "`)`"

Figure 1: Syntax for LiXQuery queries and expressions

left associative and are preceded by the filter operation (rule [17]) in priority.

2.2 Informal Semantics

Since we assume that the reader is already somewhat familiar with XQuery we only describe here the semantics of some of the less common expressions. More precisely, we will discuss the semantics of rules [5], [11], [12], [15], [21], [22], and [23].

In rule [5] the built-in functions are declared. The function `doc()` returns the document of the XML document with the name that was given as its argument, e.g., `doc("file.xq")` indicates the document root of the content of the XML document `file.xq`. The function `name()` returns the name of an element node or attribute node. The function `string()` gives the string value of an attribute node or text node, and converts integers to strings. The function `xs:integer()`[2] converts strings to integers. The function `root()` gives for a node the root of its tree. The function `concat()` concatenates strings. Rule [11] introduces the comparison operators for basic values. These comparison operators have existential semantics, i.e., they are true for two sequences if there is a basic value in one sequence and a basic value in the other sequence such that the comparison holds between these two basic values. Rule [12] gives the comparison operators for nodes where "`is`" detects the equality of nodes and "`<<`" compares nodes in document order. Rule [15] expresses the union of two node sequences, i.e., it returns a sequence of nodes that contains exactly all the nodes in the operands, contains no duplicates and is sorted in document order. Rule [21] gives the constructors for each type of node. The semantics of "`element` $\{e_1\}\{e_2\}$" is that an element node with name e_1 and content e_2 is created. The semantics of "`attribute` $\{e_1\}\{e_2\}$" is that an attribute node with name e_1 and value e_2 is created. The semantics of "`text` $\{e\}$" is that a text node with value e is created. The semantics of "`document` $\{e\}$" is that a document node with attributes and content as in e is created. Rules [22] and [23] define the typeswitch-expression that checks whether a value belongs to certain types and for the first type that matches returns a certain value.

[2] "`xs:`" indicates a namespace. Although we do not handle namespaces we use them here to be compatible with XQuery.

2.3 Syntactic Sugar

To allow for a shorter notation of certain very common expressions we introduce some short-hands.

2.3.1 The Empty Function

The function `empty()` is assumed to be declared as follows:

```
declare function empty($sequence){
  count($sequence) = 0
};
```

2.3.2 Quantified Formulas

The expression "`some` v `in` e_1 `satisfies` e_2" is introduced as a shorthand for "`not(empty(for` v `in` e_1 `return if (`e_2`) then` v `else ()))`", and "`every` v `in` e_1 `satisfies` e_2" is introduced as a shorthand for "`empty(for` v `in` e_1 `return if (`e_2`) then () else` v`)`".

2.3.3 FLWOR Expression

When `for` and `let` expressions are nested we allow that the intermediate "`return`" is removed. E.g., "`for` v_1

`in` e_1 `return let` $\$v_2$ `:=` e_2 `return` e_3" may be written as "`for` $\$v_1$ `in` e_1 `let` $\$v_2$ `:=` e_2 `return` e_3". Furthermore we allow in `for` and `let` expressions the shorthand "`where` e_1 `return` e_2" for "`return if` e_1 `then` e_2 `else ()`".

2.3.4 Coercion

Let e_1 (or e_2) have the form "`string(`e`)`" where the result of e is a sequence containing a single text node or a single attribute node. Then e_1 (or e_2) can be replaced by e in the following expressions: "`xs:integer(`e_1`)`", "`concat(`e_1`,`e_2`)`", "e_1`=`e_2", "e_1`<`e_2" and "`attribute{`e_1`}{`e_2`}`".

2.4 Design Choices

We have learned from experience that the XQuery standard contains numerous features that are important for designing a practical and efficient query language. However, many of these features do not add any expressive power to the language, while others are not essential for understanding typical queries written in XQuery.

Therefore, we choose to omit a number of standard XQuery features. However, to ensure the validity of LiXQuery, we designed it as a proper sublanguage. Specifically, we made sure that all syntactically valid LiXQuery expressions also satisfy the XQuery syntax. Moreover, the LiXQuery semantics is defined in such a way that the set of possible results of a query evaluated using our semantics will be a proper subset of the result of the set of possible results of the same query evaluated with the full XQuery semantics. Of course, the lack of a complete formal semantics for XQuery does not allow us to prove that relation.

The most visible feature we dropped from XQuery are *types* (and consequently *type coercion*). Types are very useful in XQuery for numerous reasons. But unfortunately, types –esp ecially type coercions– add lots of complexity to a formal semantic definition of a language. And since types are optional in XQuery anyway, we decided to omit them for our sub-language.

Secondly, we removed most of the navigational axes, keeping only the `child`, `parent`, `self` and `descendant-or-self` axes. It has been proven that all other axes can be simulated using the aforementioned ones. Additionally, these extra axes are rarely used in common path expressions.

Finally, we omitted primitive data-types, the `order by` clause, namespaces, comments, programming instructions, entities, and almost all of the built-in functions and operators. For these features we argue that they are necessary to specify a full-fledged query language, yet add too much overhead to incorporate them in a concise, yet formal semantics description.

3. INSTRUCTIVE EXAMPLES

In this section we discuss LiXQuery informally and provide a short example. The query in Fig. 2(a) restructures a list of parts, containing information about their containing parts, to an embedded list of the parts with their subparts [5]. For instance, the document of Fig. 2(b) will be transformed into that of Fig. 2(c). The query starts with the definition of the function `oneLevel`. This is followed by the let-clause that defines the variable `$list` whose value is the `partList` element on the file `partList.xml`. Then a new element is returned with name `intList` and which has as content the result of the function `oneLevel` that is called for each `part`-element `$p` in the `$list` element that has no `partOf`-attribute. The function `oneLevel` constructs a new `part`-element, with one attribute. It is named `partId` and its value is the string of the `partId` attribute of the element `$p` (the second parameter of `oneLevel`). Furthermore the element `part` has a child-element `$s` for each of the parts in the first parameter `$l` and which is part of `$p`. For each such an `$s` the function `oneLevel` is called recursively. If the file `partList.xml` contains Fig. 2(b) the result is shown in Fig. 2(c).

The following example shows how the `ancestor` axis can be simulated in LiXQuery.

```
declare function ancestor($s) {
(: retrieves all anc's of the nodes in $s :)
for $node in $s
  for $anc in root($node)//.
  where some $v in $anc//(*,@*,text())
        satisfies $v is $node
  return $anc
};
```

4. FORMAL SEMANTICS

Due to space limitations, we can only provide a short introduction to the formal semantics of LiXQuery. We refer to [12] for the full description. We will use following notations: the set $\mathcal{A}$ denotes the set of all atomic values, $\mathcal{V}$ is the set of all nodes, $\mathcal{S} \subseteq \mathcal{A}$ is the set of all strings, and $\mathcal{N} \subseteq \mathcal{S}$ is the set of strings that may be used as tag names. The set $\mathcal{V}$ is partitioned into the sets of document nodes ($\mathcal{V}^d$), element nodes ($\mathcal{V}^e$), attribute nodes ($\mathcal{V}^a$), and text nodes ($\mathcal{V}^t$).

4.1 Store

Expressions will be evaluated against an *XML store* which contains XML fragments. This store contains the fragments that are created as intermediate results, but also the web documents that are accessed by the expression. Although in practice these documents are materialized in the store when they are accessed for the first time, we will assume here that all documents are in fact already in the store when the expression is evaluated.

DEFINITION 4.1 (XML STORE). *An* XML store *is a 6-tuple* $St = (V, E, <, \nu, \sigma, \delta)$ *with*

- V *is a finite subset of* $\mathcal{V}$*; we write* V^d *for* $V \cap \mathcal{V}^d$ *(resp.* V^e *for* $V \cap \mathcal{V}^e$, V^a *for* $V \cap \mathcal{V}^a$, V^t *for* $V \cap \mathcal{V}^t$*);*
- (V, E) *is an acyclic directed graph (with nodes* V *and directed edges* E*) where each node has an in-degree of at most one, and hence it is composed of trees; if* $(m, n) \in E$ *then we say that* n *is a child*

```
declare function oneLevel($l,$p) {
  element { "part" } {
    attribute { "partId" }{ $p/@partId },
    for $s in $l//part
    where $s/@partOf=$p/@partId
    return oneLevel($l,$s)
  }
};

let $list := doc("partList.xml")/partList
return
  element { "intList" } {
    for $p in $list//part[empty(@partOf)]
    return oneLevel($list,$p)
  }
```

(a)

```
<?xml version ="1.0"?>
<partList>
  <part partId="1"/>
  <part partId="2" partOf="1"/>
  <part partId="3" partOf="1"/>
  <part partId="4" partOf="3"/>
  <part partId="5"/>
  <part partId="6" partOf="5"/>
</partList>
```

(b)

```
<intList>
  <part partId="1">
    <part partId="2"/>
    <part partId="3">
      <part partId="4"/>
    </part>
  </part>
  <part partId="5">
    <part partId="6"/>
  </part>
</intList>
```

(c)

Figure 2: A LiXQuery query (a), a document (b) and the result of the query applied to the document (c).

of m;[3] *we denote by E^* the reflexive transitive closure of E;*

- *$<$ is a strict partial order on V that compares exactly the different children of a common node, hence for two distinct nodes n_1 and n_2 it holds that $((n_1 < n_2) \vee (n_2 < n_1)) \Leftrightarrow \exists m \in V((m, n_1) \in E \wedge (m, n_2) \in E)$*
- *$\nu : V^e \cup V^a \rightarrow \mathcal{N}$ labels the element and attribute nodes with their* node name*;*
- *$\sigma : V^a \cup V^t \rightarrow \mathcal{S}$ labels the attribute and text nodes with their* string value*;*
- *$\delta : \mathcal{S} \rightarrow \mathcal{V}^d$ a partial function that associates with an URI or a file name, a document node. It is called the* document function*. This function represents all the URIs of the Web and all the names of the files, together with the documents they contain. We suppose that all these documents are in the store.*

The following properties have to hold for an XML store:

- *each document node of V^d is the root of a tree and has only one child element;*
- *attribute nodes of V^a and text nodes of V^t do not have any children;*
- *in the $<$-order attribute children precede the element and text children, i.e. if $n_1 < n_2$ and $n_2 \in V^a$ then $n_1 \in V^a$;*
- *there are no adjacent text children, i.e. if $n_1, n_2 \in V^t$ and $n_1 < n_2$ then there is an $n_3 \in V^e$ with $n_1 < n_3 < n_2$;*
- *for all text nodes n_t of V^t holds $\sigma(n_t) \neq$ "";*
- *all the attribute children of a common node have a different name, i.e. if $(m, n_1), (m, n_2) \in E$ and $n_1, n_2 \in V^a$ then $\nu(n_1) \neq \nu(n_2)$.*

Similarly to XQuery there exists a total order over all nodes in the store, called document order. This order is uniquely defined for nodes within the same tree. However, the XQuery Formal Semantics states that each implementation can choose how the root nodes are ordered, as long as the document order is stable during the evaluation of a query.

[3] As opposed to the terminology of XQuery, we consider attribute nodes as children of their associated element node. The definitions of parent, descendant and ancestor are straightforward.

DEFINITION 4.2 (DOCUMENT ORDER OF A STORE). *A* document order $\ll$ *of a store St is a total order on V such that*

1. *if $(n_1, n_2) \in E^*$ and $n_1 \neq n_2$ then $n_1 \ll_{St} n_2$;*
2. *if $(n_1, n_2) \in E^*$ and $n_1 < n_3$ then $(n_2 \ll_{St} n_3)$;*
3. *if $(n_1, n_2), (n_1, n_4) \in E^*$ and $n_2 \ll_{St} n_3 \ll_{St} n_4$ then $(n_1, n_3) \in E^*$.*

1. and 2. define the preorder in a tree. 3. says that the nodes of a tree are clustered.

From this definition follows that we can have more than one document order of a store St, but we choose a fixed document order here that we denote by $\ll_{St}$.

The set of items in a sequence l is denoted as $\mathbf{Set}(l)$. Given a sequence of nodes l in an XML store St we let $\mathbf{Ord}_{St}(l)$ denote the unique sequence $l' = \langle y_1, \ldots, y_m \rangle$ such that $\mathbf{Set}(l) = \mathbf{Set}(l')$ and $y_1 \ll_{St} \ldots \ll_{St} y_m$.

4.2 Environment

Expressions are also evaluated against an environment. The environment contains variable bindings that are currently in scope, function definitions, and the context used for the evaluation of path expressions. Assuming that $\mathcal{X}$ is the set of LiXQuery-expressions this environment is defined as follows.

DEFINITION 4.3 (ENVIRONMENT). *An* environment *of an XML store St is a tuple $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x}, \mathbf{k}, \mathbf{m})$ with*

1. *a partial function $\mathbf{a} : \mathcal{N} \rightarrow \mathcal{N}^*$ that maps a function name to its formal arguments; it is used in rule [1,2,24];*

2. *a partial function* $\mathbf{b} : \mathcal{N} \rightarrow \mathcal{X}$ *that maps a function name to the body of the function; it is also used in rules [1,2,24];*
3. *a partial function* $\mathbf{v} : \mathcal{N} \rightarrow (\mathcal{V} \cup \mathcal{A})^*$ *that maps variable names to their values;*
4. $\mathbf{x}$ *which is undefined or an item of St and indicates the context item – it is used in rule [16,17,18];*
5. $\mathbf{k}$ *which is undefined or an integer and gives the position of the context item in the context sequence; it is used in rule [5,17,18];*
6. $\mathbf{m}$ *which is undefined or an integer and gives the size of the context sequence; it is used in rule [5, 17,18].*

If En is an environment, n a name and y an item then we let $En[\mathbf{a}(n) \mapsto y]$ ($En[\mathbf{b}(n) \mapsto y]$, $En[\mathbf{v}(n) \mapsto y]$) denote the environment that is equal to En except that the function $\mathbf{a}$ ($\mathbf{b}$, $\mathbf{v}$) maps n to y. Similarly, we let $En[\mathbf{x} \mapsto y]$ ($En[\mathbf{k} \mapsto y]$, $En[\mathbf{m} \mapsto y]$) denote the environment that is equal to En except that $\mathbf{x}$ ($\mathbf{k}$, $\mathbf{n}$) is defined as y if $y \neq \perp$ and undefined otherwise.

We write $St, En \vdash e \Rightarrow (St', v)$ to denote that the evaluation of expression e against the XML store St and environment En of St may result in the new XML store St' and value v of St'.

4.3 Semantic Rules

In what follows we discuss some of the reasoning rules that are used to define the semantics of LiXQuery. For the full set of rules, we refer to [12]. Each rule consists of a set of premises and a conclusion of the form $St, En \vdash e \Rightarrow (St', v)$. The free variables in the rules are always assumed to be universally quantified.

For-expression (Rule [7]) The rule for

$$\texttt{for } \$s \texttt{ at } \$s' \texttt{ in } e \texttt{ return } e'$$

specifies that first e is evaluated and then e' for each item in the result of e but with $\$s$ and $\$s'$ in the environment bound to the respectively the item in question and its position in the result of e. Finally the results for each item are concatenated to a single sequence.

$$\frac{\begin{array}{c} St, En \vdash e \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) \\ St_0, En[\mathbf{v}(\$s) \mapsto x_1][\mathbf{v}(\$s') \mapsto 1] \vdash e' \Rightarrow (St_1, v_1) \\ \ldots \\ St_{m-1}, En[\mathbf{v}(\$s) \mapsto x_m][\mathbf{v}(\$s') \mapsto m] \vdash e' \Rightarrow (St_m, v_m) \end{array}}{St, En \vdash \texttt{for } \$s \texttt{ at } \$s' \texttt{ in } e \texttt{ return } e' \Rightarrow (St_m, v_1 \circ \ldots \circ v_m)}$$

Constructors (Rule [21]) Constructors are the only operations that create a new store. More precisely, the inference rules for constructors are the only rules that have a result store in the conclusion that is not the input store or a result store of one of the subexpressions.

Before we proceed with the presentation of the rule for the element constructor, we first introduce the notion of *deep equality*. This defines what it means for two nodes in an XML store to represent the same XML fragment.

DEFINITION 4.4 (DEEP EQUAL). *Given the XML store* $St = (V, E, <, \nu, \sigma, \delta)$ *and two nodes* n_1 *and* n_2 *in* St. n_1 *and* n_2 *are said to be* deep equal, *denoted as* $\mathbf{DpEq}_{St}(n_1, n_2)$, *if* n_1 *and* n_2 *refer to two isomorphic trees, i.e., there is a one-to-one function* $h : C_{n_1} \rightarrow C_{n_2}$ *with* $C_{n_i} = \{n | (n_i, n) \in E^*\}$ *for* $i = 1, 2$, *such that for each* $n, n' \in C_{n_1}$ *it holds that (1) if* $n \in \mathcal{V}^d$ ($\mathcal{V}^e$, $\mathcal{V}^a$, $\mathcal{V}^t$) *then* $h(n) \in \mathcal{V}^d$ ($\mathcal{V}^e$, $\mathcal{V}^a$, $\mathcal{V}^t$), *(2) if* $\nu(n) = s$ *then* $\nu(h(n)) = s$, *(3) if* $\sigma(n) = s'$ *then* $\sigma(h(n)) = s'$, *(4)* $(n, n') \in E$ *iff* $(h(n), h(n')) \in E$ *and (5) if* $n, n' \notin \mathcal{V}^a$ *then* $n < n'$ *iff* $h(n) < h(n')$.

The semantics of the element constructor

$$\texttt{element}\{e'\}\{e''\}$$

is defined as follows. First e' is evaluated and assumed to result in a single legal element name. The e'' is evaluated and for the result we create a new store St_3 that contains the new element with the result of e' as its name and with contents that are deep-equivalent with the result of e'' if we compare them item by item. Finally we add St_3 to the original store and return the newly created element node.

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_1, \langle s \rangle) \\ s \in \mathcal{N} \qquad St_1, En \vdash e'' \Rightarrow (St_2, \langle n_1, \ldots, n_m \rangle) \\ n_1, \ldots, n_m \in \mathcal{V} \qquad St_4 = St_2 \cup St_3 \\ n \in V_{St_3} \Rightarrow (r, n) \in E^*_{St_3} \qquad r \in \mathcal{V}^e \qquad \nu_{St_3}(r) = s \\ \mathbf{Ord}_{St_3}(\{n' | (r, n') \in E_{St_3}\}) = \langle n'_1, \ldots, n'_m \rangle \\ \mathbf{DpEq}_{St_4}(n_1, n'_1) \qquad \ldots \qquad \mathbf{DpEq}_{St_4}(n_m, n'_m) \\ \forall\, n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n')) \end{array}}{St, En \vdash \texttt{element}\{e'\}\{e''\} \Rightarrow (St_4, \langle r \rangle)}$$

Similar rules exist for attribute and text node construction. Note that the constructor rules are the only nondeterministic rules in LiXQuery, since there are several possible document orders for the new store St_4, which are only partially restricted by earlier choices of a document order St_2.

5. RESEARCH OPPORTUNITIES

The formal specification of LiXQuery gives us the opportunity to study some aspects of XQuery more formally. In this section we will discuss a few possible research directions in which we can use the LiXQuery language as a formal foundation.

A first application of LiXQuery is that of formally specifying extensions to, or alternative subsets of XQuery. For example, the language XQBE[8] (XQuery By Example) is a powerful graphical query language for XML with a well-defined formal semantics. The implementation of XQBE translates visual queries to expressions within a subset of XQuery. Translating XQBE to LiXQuery expressions instead of XQuery expressions would facilitate the process of formally proving the correspondence between the semantics of XQBE and generated XQuery expressions. A more fundamental extension to XQuery that can be formally described using LiXQuery

are updates in XQuery. In [17] an extension to XQuery to incorporate update operations is defined, but this work lacks a formal semantics, hence it is for example unclear whether node identity is preserved when moving or replacing a node.

Another possible research topic that may benefit from our LiXQuery definition is XQuery optimization, where we could use our formal semantics to prove that we can replace some expressions in a query by equivalent expressions, which are less expensive to evaluate.

Finally, we can use LiXQuery for studying the expressive power of some typical XQuery constructs, since LiXQuery provides us a compact and formal foundation needed to perform this study. Fragments of LiXQuery can be defined and studied to identify a number of structural properties, similar to what was done for XPath fragments by Benedikt, Fan, and Kuper in [6]. These fragments can also be used for studying the complexity of query evaluation in XQuery fragments, comparable to the study of the complexity of XPath query evaluation performed by Gottlob, Koch, and Pichler [10].

6. DISCUSSION

In this work we have introduced a fragment of XQuery called LiXQuery along with a formal and concise description of its semantics that is consistent with the formal semantics of XQuery. We claim that this fragment captures the essence of XQuery as a query language and can therefore be used for educational purposes, e.g., teaching XQuery, and research purposes, e.g., investigating the expressive power of XQuery fragments.

7. REFERENCES

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

[2] XQuery 1.0 and XPath 2.0 data model, W3C working draft 29 october 2004. `http://www.w3.org/TR/2003/WD-xpath-datamodel/`.

[3] XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft 20 february 2004. `http://www.w3.org/TR/2004/WD-xquery-semantics/`.

[4] XQuery 1.0 and XPath 2.0 functions and operators, W3C working draft 29 october 2004. `http://www.w3.org/TR/2003/WD-xpath-functions/`.

[5] XML query use cases, 1.8.4.1. 2003. `http://www.w3.org/TR/xquery-use-cases/`.

[6] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *International Conference on Database Theory*, 2003.

[7] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27:21–39, 2002.

[8] A. Campi, D. Braga, and S. Ceri. XQBE (XQuery by example): a visual interface to the standard XML query language. *ACM Transactions on Database Systems, June 2005*, 2005.

[9] D. Chamberlin. XQuery: An XML query language, tutorial overview. *IBM Systems Journal*, 41(4), 2002.

[10] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.

[11] J. Harbarth. XQuery 1.0 primer. 2003. `http://www.softwareag.com/xml/tools/xquery_primer.pdf`.

[12] 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)*, number 2186 in LNCS, pages 5–20, Toronto, Canada, 2004. Springer. `http://www.adrem.ua.ac.be/pub/lixquery.pdf`.

[13] H. Katz, D. Chamberlin, D. Draper, M. Fernández, M. Kay, J. Robie, M. Rys, J. Siméon, J. Tivy, and P. Wadler, editors. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2004.

[14] S. Kepser. A simple proof of the turing-completeness of XSLT and XQuery. In T. Usdin, editor, *Extreme Markup Languages 2004*, 2004.

[15] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database Systems (PODS)*, Paris, France, 2004.

[16] M. Marx. XCPath, the first order complete XPath dialect. In *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database Systems (PODS)*, Paris, France, 2004.

[17] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. of the ACM SIGMOD 2001 International Conference on Management of Data*, 2001.

[18] P. Wadler. Two semantics for XPath, 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.