A Light but Formal Introduction to XQuery

Jan Hidders, Jan Paredaens, Roel Vercammen*, and Serge Demeyer

University of Antwerp, Middelheimlaan 1, BE-2020 Antwerp, Belgium
{jan.hidders,jan.paredaens,roel.vercammen,serge.demeyer}@ua.ac.be

Abstract. We give a light-weight but formal introduction to XQuery by defining a sublanguage of XQuery. We ignore typing, and don't consider namespaces, comments, programming instructions, and entities. To avoid confusion we call our version LiXQuery (Light XQuery). LiXQuery is fully downwards compatible with XQuery. Its syntax and its semantics are far less complex than that of XQuery, but the typical expressions of XQuery are included in LiXQuery. We claim that LiXQuery is an elegant and simple sublanguage of XQuery that can be used for educational and research purposes. We give the complete syntax and the formal semantics of LiXQuery.

1 Introduction

XQuery is considered to become the standard query language for XML-documents [1,10,7,9]. However, this language is rather complex and its semantics, although well defined (see [3,2,4]), is not easily defined in a precise and concise manner. There seems therefore to be a need for a sublanguage of XQuery that has almost the same expressive power as XQuery and that has an elegant syntax and semantics that can be written down in a few pages. Similar proposals were made for XPath 1.0 [12] and XSLT [6], and have subsequently played important roles in practical and theoretical research [8,11]

Such a language would enable us to investigate more easily certain aspects of XQuery such as the expressive power of certain types of expressions found in XQuery, the expressive power of recursion in XQuery and possible syntactical restrictions that let us control this power, the complexity of deciding equivalence of expressions for purposes such as query optimization, the functional character of XQuery in comparison with functional languages such as LISP and ML, the role of XPath 1.0 and 2.0 in XQuery in terms of expressive power and query optimization, and finally the relationship between XQuery queries and the classical well-understood concept of generic database queries.

The contribution of this paper is the definition of LiXQuery, a sublanguage of XQuery with a relatively simple syntax and semantics that is appropriate both for educational and research purposes. Indeed, we are convinced that LiXQuery has a number of interesting properties, that can be proved formally, and that can be transposed to XQuery.

* Roel Vercammen is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant number 31581.

Z. Bellahsène et al. (Eds.): XSym 2004, LNCS 3186, pp. 5–20, 2004.

Section 2 contains an example of a query in LiXQuery , while section 3 explains the design choices we made. In Section 4 we give the complete syntax and its informal meaning. Section 5 further illustrates LiXQuery with more examples and finally in Section 6 the formal semantics of LiXQuery is given. In the remainder of this paper the reader is assumed to be acquainted with XML and to have some notions about XQuery.

2 Design Choices

We designed LiXQuery with two audiences in mind. First of all, we target researchers investigating the expressive power and the computational complexity of XQuery. From experience, we know that the syntax and semantics in the XQuery standard is unwieldy for proving certain properties, hence we dropped a number of language features which are important for practical purposes but not essential from a theory perspective. The second audience for LiXQuery are teachers. Here as well we learned from experience that the XQuery standard contains features which are important for designing an efficient and practical language, but not essential to understand the 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 specified LiXQuery so that all syntactically valid LiXQuery expressions do also satisfy the XQuery syntax. Moreover, the LiXQuery semantics is defined in such a way that the result of a query evaluated using our semantics will be a proper subset of the same query evaluated by XQuery. 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 indeed important for certain query optimizations because they enable to catch certain mistakes at compile time. Moreover, type coercion is quite convenient when dealing with semi-structured data, as it allows for shorter expressions. Unfortunately, types –especially 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 sublanguage.

Secondly, we removed most of the axes of navigation namely the horizontal ones (‘following’, ‘following-sibling’, ‘preceding’, ‘preceding-sibling’) and half of the vertical ones (‘ancestor’) preserving only the ‘descendant-or-self’ and ‘child’ directions. Indeed, it has been shown formally that all other axes of navigation can be reduced to the ones we preserved, thus from a theory perspective such a simplification makes sense. From an educational perspective, it is sufficient to observe that the extra navigation axes are rarely needed, hence add to the cognitive overhead.

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

```
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)
  }
```

Fig. 1. A LiXQuery query

3 An Example

The query in Fig. 1 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 will be transformed into that of Fig. 3. 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 the result is shown in Fig. 3.

```
<?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>
```

Fig. 2. Content of the file `partList.xml`

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

Fig. 3. Result of the query in Fig. 1

4 Syntax and Informal Description of LiXQuery

We first give the syntax and the informal semantics of LiXQuery, and then extend it with some syntactic sugar.

4.1 Basic Syntax

The syntax of LiXQuery is given in Fig. 4 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 their XQuery syntax. 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 without quotes 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 syntax contains 24 rules. Their informal semantics is mostly straightforward. Some of the rules were illustrated in the introductory example.

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

4.2 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.

In rule [5] the built-in functions are declared. The function `doc()` returns the document node that is the root of the tree that corresponds to the content of the file with the name that was given as its argument, e.g., `doc("file.xq")` indicates the document root of the content of the file `file.xq`. The function `name()` gives the tag name of an element node or the attribute name of an attribute node. The function `string()` gives the string value of an attribute

[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⟩)*)?“)”
```

Fig. 4. Syntax for LiXQuery queries and expressions

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. Note that “`2 < 10`” and “`"10" < "2"`” both hold. 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

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

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.

4.3 Syntactic Sugar

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

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

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

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`)`".

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 ()`".

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`$\{e_3\}\{e_2\}$".

5 More Examples

In this section we demonstrate the expressive power of LiXQuery.

5.1 Simulating Deep Equality

The first example shows that we can express deep equality of two sequences. This essentially means that we have to check whether two fragments are isomorphic except that we have to take into account that attributes are unordered. For a more formal definition see Definition 9.

```
declare function deepat1($e,$f) {
(: detects whether the attributes of $e are equal in name and value with those of $f :)
  every $ae in $e/@* satisfies
    some $af in $f/@* satisfies
      ( name($ae)=name($af) and string($ae)=string($af) )
  and
  every $af in $f/@* satisfies
    some $ae in $e/@* satisfies
      ( name($ae)=name($af) and string($ae)=string($af) )
};

declare function typetext($e) {
(: verifies whether $e is a textnode :)
  typeswitch ($e) case text() return true() default return false()
};

declare function deepequal($se,$sf) {
(: detects whether $se and $sf are sequences of pairwise deep equal items :)
  if (empty($se) and empty($sf)) then true()
    else
  if (empty($se) or empty($sf)) then false()
    else
  if (typetext($se[1]))
    then if (typetext($sf[1]))
           then ( string($se[1])=string($sf[1]) and
                  deepequal($se[1 < position()], $sf[1 < position()]) )
           else false()
    else if (typetext($sf[1]))
           then false()
           else ( name($se[1])=name($sf[1])  and
                  deepat1($se[1],$sf[1])  and
                  deepequal($se[1]/(*|text()), $sf[1]/(*|text())) and
                  deepequal($se[1 < position()], $sf[1 < position()])
                  )
};
```

5.2 Simulation of Other Axes

We can simulate all the axes that are not already directly supported in the syntax of LiXQuery. To demonstrate this we show here the `following-sibling` and `ancestor` axis.

```
declare function following-sibling($s) {
 (: retrieves all fs’s of the nodes in $s :)
   for $node in $s
     for $sib in $node/../*
     where $sib >> $node
     return $sib
 };
```

```
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//* satisfies $v is $node
    return $anc
};
```

5.3 Simulation of the Full `string()` Function

In LiXQuery the `string()` function is only defined for integers, attribute nodes and text nodes, but in XQuery it is defined for all items. We can simulate this more general function as follows.

```
declare function concatAll($x) {
(: concatenate all strings in $x :)
  if ( empty( $x ) )
  then ""
  else concat($x[position()=1], concatAll($x[position()>1]))
};
```

```
declare function xqString($x) {
(: simulates full xquery string function :)
  if ( empty( $x ) )
  then ""
  else typeswitch ($x)
         case document-node() return concatAll($x/text())
         case element() return concatAll($x/text())
         default return string($x)
};
```

5.4 Turing Completeness

It is easy to see that the amount of arithmetic and recursion in LiXQuery allows us to express all partial recursive functions over numbers. It is also possible to simulate LISP. For this purpose we represent a LISP list `((b c) d)` as shown in Fig. 5. Given this representation we simulation the `car`, `cdr` and `cons` functions:

```
declare function car($x) { $x/*[1] };
declare function cdr($x) { element{ "list" }{ $x/*[1 < position()] } };
declare function cons($x,$y) { element{ "list" }{ $x,$y/* } };
```

Since we can also compare strings and have conditional expressions, it is easy to see that by using recursion we can define all partial recursive functions over LISP lists.

```
<list>
  <list> <atom> b </atom> <atom> c </atom> </list> <atom> d </atom>
</list>
```

Fig. 5. Simulation of the LISP list `((b c) d)`

6 Formal Semantics

We now proceed with the formal semantics of LiXQuery.

Definition 1 (Atomic Value). *We assume a set of* booleans $\mathcal{B} = \{\mathbf{true}, \mathbf{false}\}$*, a set of* strings $\mathcal{S}$ *and a set of* integers $\mathcal{I}$ *that contains integers.*[3]

Furthermore a set of Names $\mathcal{N} \subseteq \mathcal{S}$ *is identified that contains those strings that may be used as tag names [1]. For each of these sets a strict total ordering, written as $<$, is presumed to exist. The set of all atomic values is $\mathcal{A} = \mathcal{B} \cup \mathcal{S} \cup \mathcal{I}$.*

We also assume the functions $AtValueToString : \mathcal{A} \to \mathcal{S}$ which is a function that maps the atomic values to their string representation, and $StringToInteger : \mathcal{S} \to \mathcal{I}$ which is partial function that maps strings that represent integers to their integer value.

Definition 2 (Node). *We assume four countably infinite sets of nodes $\mathcal{V}^d$, $\mathcal{V}^e$, $\mathcal{V}^a$ and $\mathcal{V}^t$ which respectively represent the set of* document, element, attribute *and* text nodes*. These sets are pairwise disjoint with each other and with the set of atomic values.*

[3] We denote the empty string as “”, non-empty strings as for example “123” and the concatenation of two strings s_1 and s_2 as $s_1 \cdot s_2$.

The set of all nodes is denoted as $\mathcal{V}$, i.e., $\mathcal{V} = \mathcal{V}^d \cup \mathcal{V}^e \cup \mathcal{V}^a \cup \mathcal{V}^t$.

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 3 (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 of m;[4] 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 $((n_1 < n_2) \vee (n_1 = n_2) \vee (n_2 < n_1)) \Leftrightarrow \exists m \in V((m, n_1) \in E \wedge (m, n_2) \in E)$*
- *$\nu : V^e \cup V^a \to \mathcal{N}$ labels the element and attribute nodes with their* node name*;*
- *$\sigma : V^a \cup V^t \to \mathcal{S}$ labels the attribute and text nodes with their* string value*;*
- *$\delta : \mathcal{S} \to \mathcal{V}^d$ 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;*
- *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 $\nu(n_1) \neq \nu(n_2)$.*

Definition 4 (Union of stores). *Two stores $St = (V, E, <, \nu, \sigma, \delta)$ and $St' = (V', E', <', \nu', \sigma', \delta')$ are disjoint, denoted as $St \cap St' = \emptyset$, iff $V \cap V' = \emptyset$. The definition of the union of two disjoint stores St and St', denoted as $St \cup St'$, is straightforward.*

[4] 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 5 (Root of a node). *Given a store St, the root of one of its nodes n, denoted as* $\text{root}(n)$ *is the unique root of the tree of n, i.e.* $\text{root}(n) = r$ *iff* $(r, n) \in E^*$ *and for no node $s \neq r$ of St holds $(s, r) \in E^*$.*

Definition 6 (Item). *An* item *of an XML store St is an atomic value in $\mathcal{A}$ or a node in St.*

We denote the empty sequence as $\langle\rangle$, non-empty sequences as for example $\langle 1, 2, 3\rangle$ and the concatenation of two sequences l_1 and l_2 as $l_1 \circ l_2$. The expression $\langle y_i \in l | \varphi(y, i)\rangle$ with $\varphi(y, i)$ a formula of an item y and position i denotes the subsequence of l that is obtained by selecting from l all items that satisfy $\varphi(y, i)$.

Example 1. The XML store that is represented in Fig. 5 is $St = (V, E, <, \nu, \sigma, \delta)$ and is shown in Fig. 6. The set of nodes $V^e = \{n_1^e, n_2^e, n_3^e, n_5^e, n_7^e\}$, $V^t = \{n_4^t, n_6^t, n_8^t\}$, $V^d = V^a = \emptyset$, $E = \{(n_1^e, n_2^e), (n_1^e, n_7^e), (n_2^e, n_3^e), (n_2^e, n_5^e), (n_3^e, n_4^t), (n_5^e, n_6^t), (n_7^e, n_8^t)\}$, the order relation $<$ is defined by $n_2^e < n_7^e, n_3^e < n_5^e$; furthermore $\nu(n_1^e) = \nu(n_2^e) =$ "`list`", $\nu(n_3^e) = \nu(n_5^e) = \nu(n_7^e) =$ "`atom`" and $\sigma(n_4^t) =$ "`b`", $\sigma(n_6^t) =$ "`c`", $\sigma(n_8^t) =$ "`d`".[5]

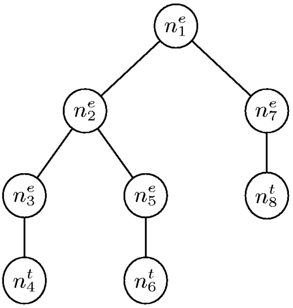

Fig. 6. XML tree of Fig. 5

Definition 7 (Document Order of a Store). *A* document order $\ll$[6] *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* $n_2 < n_3 < n_4$ *then* $(n_1, n_3) \in E^*$*.*

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

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 $y_1 \ll_{St} \cdots \ll_{St} y_m$.

6.1 Evaluation of Expressions

Expressions are evaluated against an environment. Assuming that $\mathcal{X}$ is the set of LiXQuery-expressions this environment is defined as follows.

Definition 8 (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*

[5] We do not mention here the documents on the Web and on files.

[6] A store can have more than one document order, but we choose a fixed document order here that we denote by $\ll_{St}$.

1. *a partial function* $\mathbf{a} : \mathcal{N} \to \mathcal{N}^*$ *that maps a function name to its formal arguments; it is used in rule [1,2,24];*
2. *a partial function* $\mathbf{b} : \mathcal{N} \to \mathcal{X}$ *that maps a function name to the body of the function; it is also used in rules [1,2,24];*
3. *a partial function* $\mathbf{v} : \mathcal{N} \to (\mathcal{V} \cup \mathcal{A})^*$ *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 $y \neq \bot$ 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'.

6.2 Semantic Rules

In what follows we give the reasoning rules that are used to define the semantics of LiXQuery. 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. We will use the following notation: v for values, x for items, n for nodes, r for roots, s for strings and names, f for function names, b for booleans , i for integers and e for expressions.

Query (Rules [1] and [2]) A function declaration extends $\mathbf{a}$ and $\mathbf{b}$ and then the last expression is evaluated with these $\mathbf{a}$ and $\mathbf{b}$. Function declarations are allowed to be mutually recursive.

$$\frac{En' = En[\mathbf{a}(f) \mapsto \langle s_1, \ldots s_m \rangle][\mathbf{b}(f) \mapsto e] \qquad St, En' \vdash e' \Rightarrow (St', v)}{St, En \vdash \texttt{declare function } f(s_1, \ldots, s_m)\{\ e\ \};\ e' \Rightarrow (St', v)}$$

Variable (Rule [4])

$$\frac{}{St, En \vdash \$s \Rightarrow (St, \mathbf{v}_{En}(s))}$$

Built-in Functions (Rule [5])

$$\frac{St, En \vdash e \Rightarrow (St', \langle s \rangle) \qquad \delta_{St'}(s) = n}{St, En \vdash \texttt{doc}(e) \Rightarrow (St', n)} \qquad \frac{St, En \vdash e \Rightarrow (St', \langle n \rangle) \qquad n \in \mathcal{V}^e \cup \mathcal{V}^a}{St, En \vdash \texttt{name}(e) \Rightarrow (St', \langle \nu_{St'}(n) \rangle)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle n \rangle) \qquad n \in \mathcal{V}^a \cup \mathcal{V}^t}{St, En \vdash \texttt{string}(e) \Rightarrow (St', \langle \sigma_{St'}(n) \rangle)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle x \rangle) \qquad x \in \mathcal{A} \qquad \mathit{AtValueToString}(x) = s}{St, En \vdash \texttt{string}(e) \Rightarrow (St', \langle s \rangle)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle s \rangle) \qquad s \in \mathcal{S} \qquad \mathit{StringToInteger}(s) = i}{St, En \vdash \texttt{xs : integer}(e) \Rightarrow (St', \langle i \rangle)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle n \rangle) \qquad n \in V_{St'}}{St, En \vdash \texttt{root}(e) \Rightarrow (St', \langle \mathit{root}(n) \rangle)}$$

$$\frac{St, En \vdash e_1 \Rightarrow (St_1, \langle s_1 \rangle) \qquad s_1 \in \mathcal{S} \qquad St_1, En \vdash e_2 \Rightarrow (St_2, \langle s_2 \rangle) \qquad s_2 \in \mathcal{S}}{St, En \vdash \texttt{concat}(e_1, e_2) \Rightarrow (St_2, \langle s_1 \cdot s_2 \rangle)}$$

$$\frac{}{St, En \vdash \texttt{true}() \Rightarrow (St, \langle \mathbf{true} \rangle)} \qquad \frac{}{St, En \vdash \texttt{false}() \Rightarrow (St, \langle \mathbf{false} \rangle)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle b \rangle) \qquad b \in \mathcal{B}}{St, En \vdash \texttt{not}(e) \Rightarrow (St', \langle \neg b \rangle)} \qquad \frac{St, En \vdash e \Rightarrow (St', \langle x_1, \ldots, x_m \rangle)}{St, En \vdash \texttt{count}(e) \Rightarrow (St', \langle m \rangle)}$$

$$\frac{}{St, En \vdash \texttt{position}() \Rightarrow (St, \langle \mathbf{k}_{En} \rangle)} \qquad \frac{}{St, En \vdash \texttt{last}() \Rightarrow (St, \langle \mathbf{m}_{En} \rangle)}$$

If-expression (Rule [6]). The semantics of the if-expression is given by two inference rules: one for the case the condition evaluates to **true** and one for **false**. Note that in each case only one of the branches is executed.

$$\frac{St, En \vdash e \Rightarrow (St', \langle \mathbf{true} \rangle) \qquad St', En \vdash e_1 \Rightarrow (St_1, v_1)}{St, En \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow (St_1, v_1)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle \mathbf{false} \rangle) \qquad St', En \vdash e_2 \Rightarrow (St_2, v_2)}{St, En \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rightarrow (St_2, v_2)}$$

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.

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

Let-expression (Rule [8])

$$\frac{St, En \vdash e \Rightarrow (St', v) \qquad St', En[\mathbf{v}(s) \mapsto v] \vdash e' \Rightarrow (St'', v')}{St, En \vdash \texttt{let}\ \$s := e\ \texttt{return}\ e' \Rightarrow (St'', v')}$$

Concatenation (Rule [9])

$$\frac{St, En \vdash e' \Rightarrow (St', v') \qquad St', En \vdash e'' \Rightarrow (St'', v'')}{St, En \vdash e', e'' \Rightarrow (St'', v' \circ v'')}$$

Boolean Operators (Rule [10])

$$\frac{St, En \vdash e' \Rightarrow (St', \langle b' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle b'' \rangle) \qquad b', b'' \in \mathcal{B}}{St, En \vdash e' \ \texttt{and}\ e'' \Rightarrow (St'', \langle b' \wedge b'' \rangle) \qquad St, En \vdash e' \ \texttt{or}\ e'' \Rightarrow (St'', \langle b' \vee b'' \rangle)}$$

Atomic Value Comparisons (Rule [11])

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St', \langle x'_1, \ldots, x'_{m'} \rangle) \\ x'_1, \ldots, x'_{m'} \in \mathcal{A} \qquad St', En \vdash e'' \Rightarrow (St'', \langle x''_1, \ldots, x''_{m''} \rangle) \qquad x''_1, \ldots, x''_{m''} \in \mathcal{A} \\ b_{=} \Leftrightarrow \exists_{1 \le i \le m', 1 \le j \le m''}(x'_i = x''_j) \qquad b_{<} \Leftrightarrow \exists_{1 \le i \le m', 1 \le j \le m''}(x'_i < x''_j) \end{array}}{St, En \vdash e' = e'' \Rightarrow (St'', \langle b_{=} \rangle) \qquad St, En \vdash e' < e'' \Rightarrow (St'', \langle b_{<} \rangle)}$$

Node Comparisons (Rule [12])

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St', \langle n' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle n'' \rangle) \\ n', n'' \in \mathcal{V} \qquad b_{\texttt{is}} \Leftrightarrow (n' = n'') \qquad b_{\ll} \Leftrightarrow (n' \ll_{St''} n'') \end{array}}{St, En \vdash e' \ \texttt{is}\ e'' \Rightarrow (St'', \langle b_{\texttt{is}} \rangle) \qquad St, En \vdash e' << e'' \Rightarrow (St'', \langle b_{\ll} \rangle)}$$

Additions (Rule [13])

$$\frac{St, En \vdash e' \Rightarrow (St', \langle d' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle d'' \rangle) \qquad d', d'' \in \mathcal{I}}{St, En \vdash e' + e'' \Rightarrow (St'', \langle d' + d'' \rangle) \qquad St, En \vdash e' - e'' \Rightarrow (St'', \langle d' - d'' \rangle)}$$

Multiplications (Rule [14])

$$\frac{St, En \vdash e' \Rightarrow (St', \langle d' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle d'' \rangle) \qquad d', d'' \in \mathcal{I}}{St, En \vdash e' * e'' \Rightarrow (St'', \langle d' \times d'' \rangle) \qquad St, En \vdash e' \ \texttt{idiv}\ e'' \Rightarrow (St'', \langle d'/d'' \rangle)}$$

Union (Rule [15])

$$\frac{St, En \vdash e' \Rightarrow (St', v') \qquad St', En \vdash e'' \Rightarrow (St'', v'') \qquad v', v'' \in \mathcal{V}^*}{St, En \vdash e' \mid e'' \Rightarrow (St'', \mathbf{Ord}_{St''}(\mathbf{Set}(v') \cup \mathbf{Set}(v'')))}$$

Axis Steps (Rule [16]) The semantics of a step consisting of an element name s is that all element children of the context node (indicated in the envorment by $\mathbf{x}$) with name s are returned in document order. The semantics of the step consisting of the wild-card `*` is the same except that all element children of the context node are returned.

$$\frac{\mathbf{x}_{En} \ \textit{is defined}}{St, En \vdash . \Rightarrow (St, \langle \mathbf{x}_{En} \rangle)} \qquad \frac{(n, \mathbf{x}_{En}) \in E_{St}}{St, En \vdash .. \Rightarrow (St, \langle n \rangle)} \qquad \frac{\nexists n(n, \mathbf{x}_{En}) \in E_{St}}{St, En \vdash .. \Rightarrow (St, \langle \rangle)}$$

$$\frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^e \wedge \nu_{St}(n) = s\}}{St, En \vdash s \Rightarrow (St, \mathbf{Ord}_{St}(W))}$$

$$\frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^a \wedge \nu_{St}(n) = s\}}{St, En \vdash @s \Rightarrow (St, \mathbf{Ord}_{St}(W))}$$

$$\frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^e\}}{St, En \vdash * \Rightarrow (St, \mathbf{Ord}_{St}(W))} \qquad \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^a\}}{St, En \vdash @* \Rightarrow (St, \mathbf{Ord}_{St}(W))}$$

$$\frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^t\}}{St, En \vdash \texttt{text()} \Rightarrow (St, \mathbf{Ord}_{St}(W))}$$

Filter-expression (Rule [17]) The semantics of e' `[` e'' `]` is that first e' is evaluated, then for each item in the result of e' the expression e'' is evaluated with $\mathbf{x}$ bound to this item, $\mathbf{k}$ to the position of the item in the result of e' and $\mathbf{m}$ to the number of items in the result of e'. The result of e'' is a boolean or an integer, in which case it is converted to **true** if this integer is equal to $\mathbf{k}$ and to **false** otherwise. Finally, the result is the subsequence of the result of e' that contains exactly all items for which e'' evaluated to **true**.

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) \\ En' = En[\mathbf{m} \mapsto m] \qquad St_0, En'[\mathbf{x} \mapsto x_1][\mathbf{k} \mapsto 1] \vdash e'' \Rightarrow (St_1, \langle x'_1 \rangle) \\ \ldots \qquad St_{m-1}, En'[\mathbf{x} \mapsto x_m][\mathbf{k} \mapsto m] \vdash e'' \Rightarrow (St_m, \langle x'_m \rangle) \\ x'_1, \ldots, x'_m \in \mathcal{B} \cup \mathcal{I} \qquad v = \langle x_i | (x'_i \in \mathcal{I} \wedge x'_i = i) \vee (x'_i \in \mathcal{B} \wedge x'_i) \rangle \end{array}}{St, En \vdash e' \ [\ e''\ ] \Rightarrow (St_m, v)}$$

Path Expression (Rule [18]) The semantics of (e' `/` e'') is as follows. First e' is evaluated. Then for each item in its result we bind in the environment $\mathbf{x}$ to this item, $\mathbf{k}$ to the position of $\mathbf{x}$ in the result of e', and $\mathbf{m}$ to the number of items in the result of e', and with this environment we evaluate e''. The results of all these evaluations are concatenated and finally this sequence is sorted by document order and the duplicates are removed. The result is only defined if all the evaluations of e'' contain only nodes.

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) \\ En' = En[\mathbf{m} \mapsto m] \qquad St_0, En'[\mathbf{x} \mapsto x_1][\mathbf{k} \mapsto 1] \vdash e'' \Rightarrow (St_1, v_1) \\ \ldots \qquad St_{m-1}, En'[\mathbf{x} \mapsto x_m][\mathbf{k} \mapsto m] \vdash e'' \Rightarrow (St_m, v_m) \qquad v_1, \ldots, v_m \in \mathcal{V}^* \end{array}}{St, En \vdash e' \ /\ e'' \Rightarrow (St_m, \mathbf{Ord}_{St_m}(\cup_{1 \leq i \leq m} \mathbf{Set}(v_i)))}$$

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) \qquad W_1 = \{x \in V_{St_0} | (x_1, x) \in (E_{St_0})^*\} \\ \ldots \qquad W_m = \{x \in V_{St_0} | (x_m, x) \in (E_{St_0})^*\} \qquad \langle x'_1, \ldots, x'_{m'} \rangle = \mathbf{Ord}_{St_0}(\cup_{1 \leq i \leq m} W_i) \\ En' = En[\mathbf{m} \mapsto m'] \qquad St_0, En'[\mathbf{x} \mapsto x'_1][\mathbf{k} \mapsto 1] \vdash e'' \Rightarrow (St_1, v_1) \\ \ldots \qquad St_{m'-1}, En'[\mathbf{x} \mapsto x'_{m'}][\mathbf{k} \mapsto m'] \vdash e'' \Rightarrow (St_{m'}, v_{m'}) \qquad v_1, \ldots, v_{m'} \in \mathcal{V}^* \end{array}}{St, En \vdash e' \ //\ e'' \Rightarrow (St_{m'}, \mathbf{Ord}_{St_{m'}}(\cup_{1 \leq i \leq m'} \mathbf{Set}(v_i)))}$$

Literal (Rule [19]) The result of a literal is simply a sequence with one element, viz., the atomic value the literal represents.

Empty Sequence (Rule [20])

$$\frac{}{St, En \vdash () \Rightarrow (St, \langle\rangle)}$$

Constructors (Rule [21]) 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 9 (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 $h : C_{n_1} \to C_{n_2}$ 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 `element`$\{e'\}\{e''\}$ is the 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.

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

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_1, \langle s \rangle) \\ s \in \mathcal{N} \qquad St_1, En \vdash e'' \Rightarrow (St_2, \langle s' \rangle) \qquad s' \in \mathcal{S} \qquad St_4 = St_2 \cup St_3 \qquad V_{St_3} = \{r\} \\ r \in \mathcal{V}^a \qquad \nu_{St_3}(r) = s \qquad \sigma_{St_3}(r) = s' \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n')) \end{array}}{St, En \vdash \texttt{attribute}\{e'\}\{e''\} \Rightarrow (St_4, \langle r \rangle)}$$

$$\frac{\begin{array}{c} St, En \vdash e \Rightarrow (St_1, \langle s \rangle) \qquad s \in \mathcal{S} - \{\text{``''}\} \qquad St_3 = St_1 \cup St_2 \qquad V_{St_2} = \{r\} \\ r \in \mathcal{V}^t \qquad \sigma_{St_2}(r) = s \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_1} n') \Rightarrow (n \ll_{St_3} n')) \end{array}}{St, En \vdash \texttt{text}\{e\} \Rightarrow (St_3, \langle r \rangle)}$$

$$\frac{\begin{array}{c} St, En \vdash e \Rightarrow (St_1, \langle n_1 \rangle) \\ n_1 \in \mathcal{V}^e \qquad St_3 = St_1 \cup St_2 \qquad n \in V_{St_2} \Rightarrow (r, n) \in E^*_{St_2} \qquad r \in \mathcal{V}^d \\ (r, n_2) \in E_{St_2} \qquad \mathbf{DpEq}_{St_3}(n_1, n_2) \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_1} n') \Rightarrow (n \ll_{St_3} n')) \end{array}}{St, En \vdash \texttt{document}\{e\} \Rightarrow (St_3, \langle r \rangle)}$$

Typeswitch-expression (Rules [22] and [23]) Let $[\![\texttt{xs:boolean}]\!] = \mathcal{B}$, $[\![\texttt{xs:integer}]\!] = \mathcal{I}$, $[\![\texttt{xs:string}]\!] = \mathcal{S}$, $[\![\texttt{document-node()}]\!] = \mathcal{V}^d$, $[\![\texttt{attribute()}]\!] = \mathcal{V}^a$, $[\![\texttt{text()}]\!] = \mathcal{V}^t$ and $[\![\texttt{element()}]\!] = \mathcal{V}^e$.

$$\frac{\begin{array}{c} St, En \vdash e \Rightarrow (St_1, \langle x \rangle) \\ (x \in [\![t_j]\!] \lor j = m + 1) \qquad \forall_{1 \le i < j}(x \notin [\![t_i]\!]) \qquad St_1, En \vdash e_j \Rightarrow (St_2, v) \end{array}}{\begin{array}{c} St, En \vdash \texttt{typeswitch}(e)\ \texttt{case}\ t_1\ \texttt{return}\ e_1 \ldots\ \texttt{case}\ t_m\ \texttt{return}\ e_m \\ \texttt{default return}\ e_{m+1} \Rightarrow (St_2, v) \end{array}}$$

Function Call (Rule [24]) The semantics of $f(e_1, \ldots, e_m)$ is that $e_1, \ldots, e_m$ are consecutively evaluated, and then the expression $\mathbf{b}(f)$ is evaluated with the variable names of $\mathbf{a}(f)$ bound to the results of $e_1, \ldots, e_m$.

$$\frac{\begin{array}{c} St, En \vdash e_1 \Rightarrow (St_1, v_1) \\ \ldots \qquad St_{m-1}, En \vdash e_m \Rightarrow (St_m, v_m) \qquad \mathbf{a}_{En}(f) = \langle s_1, \ldots, s_m \rangle \\ En' = En[\mathbf{v}(s_1) \mapsto v_1] \ldots [\mathbf{v}(s_m) \mapsto v_m][\mathbf{x} \mapsto \bot, \mathbf{k} \mapsto \bot, \mathbf{m} \mapsto \bot] \\ St_m, En' \vdash \mathbf{b}_{En}(f) \Rightarrow (St', v') \end{array}}{St, En \vdash f(e_1, \ldots, e_m) \Rightarrow (St', v')}$$

7 Conclusion

In this paper we have presented a fragment of XQuery called LiXQuery together with a formal and concise but complete 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 and query optimization in XQuery implementations.

References

1. XML query (XQuery). http://www.w3.org/XML/Query.
2. XQuery 1.0 and XPath 2.0 data model, W3C working draft 12 november 2003. http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/.
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-20040220/.
4. XQuery 1.0 and XPath 2.0 functions and operators, W3C working draft 12 november 2003. http://www.w3.org/TR/2003/WD-xpath-functions-20031112/.
5. XML query use cases, 1.8.4.1. 2003. http://www.w3.org/TR/xquery-use-cases/.
6. G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27:21–39, 2002.
7. D. Chamberlin. XQuery: An XML query language, tutorial overview. *IBM Systems Journal*, 41(4), 2002.
8. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB 2002*, Hong Kong, 2002.
9. J. Harbarth. XQuery 1.0 primer. 2003. http://www.softwareag.com/xml/tools/xquery_primer.pdf.
10. 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.
11. M. Marx. XCPath, the first order complete XPath dialect. In *SIGMOD/PODS 2004*.
12. P. Wadler. Two semantics for XPath, 1999. http://www.cs.bell-labs.com/who/wadler/topics/xml.html.