On the Expressive Power of XQuery Fragments

Jan Hidders[1], Stefania Marrara[2], Jan Paredaens[1], and Roel Vercammen*[1]

[1] University of Antwerp, Dept. Math and Computer Science,
Middelheimlaan 1, BE-2020 Antwerp, Belgium
[2] Universitá degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione,
Via Bramante 65, I-26013 Crema (CR), Italy

Abstract. XQuery is known to be a powerful XML query language with many bells and whistles. For many common queries we do not need all the expressive power of XQuery. We investigate the effect of omitting certain features of XQuery on the expressive power of the language. We start from a simple base fragment which can be extended by several optional features being aggregation functions such as count and sum, sequence generation, node construction, position information in for loops, and recursion. In this way we obtain 64 different XQuery fragments which can be divided into 17 different equivalence classes such that two fragments can express the same functions iff they are in the same equivalence class. Moreover, we investigate the relationships between these equivalence classes.

1 Introduction

XQuery [2], the W3C standard query language for XML, is a very powerful query language which is known to be Turing Complete [8]. As the language in its entirety is too powerful and complex for many queries, there is a need to investigate the different properties of frequently used fragments. Most existing theoretical work focuses on XPath, a rather limited subset of XQuery. For example, Benedikt, Fan, and Kuper studied structural properties of XPath fragments [1], the computational complexity of query evaluation for a number of XPath fragments was investigated by Gottlob, Koch, and Pichler in [4], and Marx increased the expressive power of XPath by extending it in order to be first order complete. It was not until recently that similar efforts were made for XQuery: Koch studies the computational complexity of nonrecursive XQuery [9], Vansummeren looks into the well-definedness problem for XQuery fragments [13] and the expressive power of the node construction in XQuery is studied in [10]. In this paper we will investigate the expressive power of XQuery fragments in a similar fashion as was done for the relational algebra [12] and SQL [11]. In order to do this, we establish some interesting properties for these fragments. We start from a small base fragment in which we can express many commonly used features such as some built-in functions, arithmetic, boolean operators, node and

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

value comparisons, path expressions, simple for-loops and XPath set operations. This base fragment can be extended by a number of features that are likely to increase the expressive power such as recursion, aggregate functions, sequence generators, node constructors, and position information. The central question is which features of XQuery are really necessary in these fragments and which ones are only syntactic sugar, simplifying queries that were already expressible without this feature. Our most expressive fragment corresponds to LiXQuery [5], which is conjectured to be as expressive as XQuery.

This paper is organized as follows. Section 2 introduces the syntax and the semantics of the different XQuery fragments that we are going to analyze. In Section 3 we present some expressibility results for these fragments and in Section 4 we show some properties that hold for some of the fragments. These results are combined in Section 5, where we partition the set of fragments into classes of fragments with the same expressive power. Finally, Section 6 outlines the conclusions of our work.

2 XQuery Fragments

This section formally introduces the XQuery fragments for which we study the expressive power in this paper. We will use LiXQuery [5] as a formal foundation, which is a light-weight sublanguage of XQuery, fully downwards compatible with XQuery. The syntax of each of the XQuery fragments is defined in Subsection 2.1. In Subsection 2.2 we briefly describe the semantics of a query.

2.1 Syntax

The syntax of the fragment XQ is shown in Figure 1, by rules [1-19] [3]. This syntax is an abstract syntax [4]. The XQuery fragment XQ contains simple arithmetic, path expressions, "`for`" clauses (without "`at`"), the "`if`" test, "`let`" variable bindings, the existential semantics for comparison, typeswitches and some built-in functions. Adding non-recursive function definitions to XQ would clearly not augment the expressive power of XQ. We use 6 attributes for fragments: C, S, at, ctr, to and R (cf. Figure 2 for the syntax of the attributed fragments). The fragment XQ^R denotes XQ augmented with (recursive) functions definitions, XQ_C is XQ plus the "`count`" function, XQ_S denotes the inclusion of the "`sum`" function, XQ_{at} includes the "`at`" clause in a for expression, XQ^{ctr} indicates the inclusion of the node constructors, and finally the XQ^{to} denotes the sequence generator "`to`". The fragment XQ can be attributed by a set of these attributes. In this way, we obtain 64 fragments of XQuery. The aim of this paper is to investigate and to compare the expressive power of these fragments. With

[3] Note that expressions which are not allowed in a fragment definition must be considered as not occurring in the right hand side of a production rule. As an example *FunCall* and *Count* do not occur in rule [2] for XQ.

[4] It assumes that extra brackets and precedence rules are added for disambiguation.

XQ^* we denote the fragment $XQ^{R,to,ctr}_{C,S,at}$ expressed by rules [1-26]. Following auxiliary definitions will be used throughout the paper:

Definition 1. *The language* $\mathbf{L}(XF)$ *of an XQuery fragment* XF *is the (infinite) set of all expressions that can be generated by the grammar rules for this fragment with* $\langle Query \rangle$ *as start symbol. The set* Φ *is the set of all 64 XQuery fragments defined in Figure 2.*

Similar to LiXQuery, we ignore static typing and do not consider namespaces[5], comments, processing instructions, and entities. There are some features left out from LiXQuery in the definition of XQ^*, such as the union, the filter expression, the functions "`position()`" and "`last()`", and the parent step ("`..`"), but they can easily been simulated in XQ^* (see the details in [6]). From these considerations, we can claim that XQ^* has the same expressive power as LiXQuery.

2.2 Semantics

We will now introduce the semantics of our XQuery fragments which is the same as that of LiXQuery and downwards compatible with the XQuery Formal Semantics[3].

Expressions are evaluated against an *XML store* which contains XML fragments created as intermediate results, and all the web documents[6]. First we need some definitions of sets for the formal specification of the LiXQuery semantics. The set $\mathcal{A}$ is 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.

Definition 2. *An* **XML Store** *is a 6-tuple* $St = (V, E, <, \nu, \sigma, \delta)$ *where* $V = V^d \cup V^e \cup V^a \cup V^t$ *is a finite countable set of nodes* ($V \subseteq \mathcal{V}$) *consisting of the pairwise disjoint sets of document nodes* V^d*, element nodes* V^e*, attribute nodes* V^a*, and text nodes* V^t*;* (V, E) *is a forest (with nodes* V *and directed edges* E*); if* $(m, n) \in E$ *then we say that* n *is a child of* m*;* $<$ *is the sibling order for the trees in* (V, 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 attribute and text nodes with their string value;* $\delta : S \to V^d$ *is a partial function that associates with a 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 the documents are in the store.*

Moreover, for each store, each document node is the root of a tree and contains exactly one child, which is an element node; attribute nodes and text nodes do not have any children; in the $<$*-order attribute children precede the element*

[5] In types and built-in functions, such as "`xs:integer`", the "`xs:`" part indicates a namespace. Although we do not handle namespaces we use them here to be compatible with XQuery

[6] This assumption models correctly the formal semantics since each time a "`doc`" function is called for the same document, the same document node is returned.

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

---

[20] ⟨*Count*⟩ → "`count(`" ⟨*Expr*⟩ "`)`"
[21] ⟨*Sum*⟩ → "`sum(`" ⟨*Expr*⟩ "`)`"
[22] ⟨*AtExpr*⟩ → "`at`" ⟨*Var*⟩
[23] ⟨*SeqGen*⟩ → ⟨*Expr*⟩ "`to`" ⟨*Expr*⟩
[24] ⟨*FunCall*⟩ → ⟨*Name*⟩ "`(`" (⟨*Expr*⟩ ("`,`" ⟨*Expr*⟩)*)? "`)`"
[25] ⟨*FunDecl*⟩ → "`declare`" "`function`" ⟨*Name*⟩ "`(`" (⟨*Var*⟩ ("`,`" ⟨*Var*⟩)*)? "`)`" "`{`" ⟨*Expr*⟩ "`}`"
[26] ⟨*Constr*⟩ → "`element`" "`{`" ⟨*Expr*⟩ "`}`" "`{`" ⟨*Expr*⟩ "`}`" | "`attribute`" "`{`" ⟨*Expr*⟩ "`}`" "`{`" ⟨*Expr*⟩ "`}`" | "`text`" "`{`" ⟨*Expr*⟩ "`}`" | "`document`" "`{`" ⟨*Expr*⟩ "`}`"

Fig. 1. Syntax for XQ^* queries and expressions

XQ	[1-19]
$_C$	+ [20]
$_S$	+ [21]
$_{at}$	+ [22]
$_{to}$	+ [23]
$_R$	+ [24-25]
$_{ctr}$	+ [26]

Fig. 2. Definition of XQuery fragments

and text children; two sibling text nodes are separated by at least one non-text sibling node; for all text nodes n_t of V^t holds $\sigma(n_t) \neq$ ""; all attribute children of a common node have a different name.

The set ST is the set of all (valid) XML Stores.

We now give an example to illustrate this definition. In both this example and the rest of the paper, we will use the function ξ, which maps a sequence of items and a store to its serialization, as defined in [7].

Example 1. Let $St = (V, E, <, \nu, \sigma, \delta)$ be an XML store that is shown in Figure 3.

- The set of nodes V consists of $V^e = \{n_1^e, n_2^e, n_3^e, n_5^e, n_7^e\}$, $V^t = \{n_4^t, n_6^t, n_8^t\}$, $V^d = V^a = \emptyset$.
- The set of edges is $E = \{(n_1^e, n_2^e), (n_1^e, n_7^e), (n_2^e, n_3^e), (n_2^e, n_5^e), (n_3^e, n_4^t), (n_5^e, n_6^t), (n_7^e, n_8^t)\}$.
- The order relation $<$ is defined by $n_2^e < n_7^e, n_3^e < n_5^e$.
- Furthermore $\nu(n_1^e) =$"`a`", $\nu(n_2^e) = \nu(n_7^e) =$"`b`", $\nu(n_3^e) = \nu(n_5^e) =$"`c`", and $\sigma(n_4^t) =$ "`t1`", $\sigma(n_6^t) =$ "`t2`", $\sigma(n_8^t) =$ "`t3`". [7]

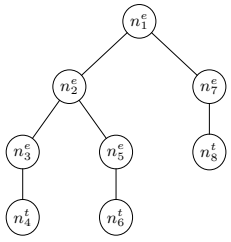

Fig. 3. XML tree of Example 1

In this example $\xi(n_1^e, St) =$ "`<a><b><c>t1</c><c>t2</c></b><b>t3</b></a>`" is the serialization of the node n_1^e.

For the evaluation of queries we do not only need an XML store, but also an environment, which contains information about functions, variable bindings, the context sequence, and the context item. This environment is defined as follows:

Definition 3 (Environment). *An* environment *of an XML store St is a 4-tuple $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ where $\mathbf{a} : \mathcal{N} \to \mathcal{N}^*$ is a partial function that maps a function name to its formal arguments (it is used in rule [1,24,25]); $\mathbf{b} : \mathcal{N} \to \mathbf{L}(XQ^*)$ maps a function name to the body of the function (it is also used in rules [1,24,25]); $\mathbf{v} : \mathcal{N} \to (\mathcal{V} \cup \mathcal{A})^*$ maps variable names to their values; $\mathbf{x}$ which is undefined or an item of St and indicates the context item (it is used in rule [16,17]).*

Let $XF \in \Phi$ be an XQuery fragment. The set of XF-environments($EN[XF]$) is the set of all environments for which it holds that $\forall f \in \mathbf{rng}(\mathbf{b}) : f \in \mathbf{L}(XF)$.

If En is an environment, n a name, and y an item then we let $En[\mathbf{v}(n) \mapsto y]$ denote the environment that is equal to En except that the function $\mathbf{v}$ maps n to y. 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 a result sequence v, where v can only contain nodes of St' and atomic values. The semantics of XQ^* expressions is defined by means of reasoning rules, following the notation detailed in [5].

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

We define the expressive power of an XQuery fragment as the set of *XQuery functions* that can be expressed in this fragment. XQuery functions are defined as partial multivalued functions that map a store and a variable assignment over that store to a new store and a result sequence over this result store. We assume that the result store does not contain nodes that are no longer reachable, since such nodes can be safely garbage collected. More precisely, the garbage collection is defined as follows:

Definition 4 (Garbage Collection). *Garbage Collection (Γ_s) maps a store St and a sequence s to a new store St' by removing all trees from St for which the root node is not in* $\mathbf{rng}(\delta)$ *and for which no node of the tree is in s.*

We now define the notion of XQuery function as follows:

Definition 5 (XQuery function). *The XQuery function corresponding to an expression e is* $\{((St, \mathbf{v}), (\Gamma_v(St'), v)) \mid St, (\phi, \phi, \mathbf{v}, \perp) \vdash e \Rightarrow (St', v)\}$. *An element of this set is called an* evaluation pair. *If two expressions e_1 and e_2 have the same corresponding XQuery functions then they are said to be equivalent, denoted as $e_1 \sim e_2$.*

This measure of expressive power can be justified by the XQuery Processing Model[3]. There it is possible to set variables in an initial environment. Moreover, the serialization of the result sequence is optional and an XQuery query can be embedded into another processing environment.

3 Expressibility Results

Adding extra features to XQuery fragments does not always extend the set of XQuery functions expressible in the fragment. In this section we will show how to simulate certain features in fragments that, syntactically, do not include this feature.

Lemma 1. *The "`count`" operator can be expressed in XQ_{at}.*

Proof. It is clear that "`max(`e_1`)`" and "`empty(`e_1`)`" can be expressed in XQ. Hence the following expression is equivalent to "`count(`e_1`)`":

```
let $v := max(for $i at $pos in e1 return $pos)
return
  if (empty($v)) then 0 else $v
```

Lemma 2. *The "`count`" operator can be expressed in XQ_S.*

Proof. Following XQ_S expression is equivalent to "`count(`e_1`)`":

```
sum(for $i in e1 return 1)
```

Lemma 3. *The "`to`" operator can be expressed in XQ^R.*

Proof. We can define a recursive function "`to`" such that "e_1 `to` e_2" is equivalent to "`to(`e_1`,` e_2`)`" as follows:

```
declare function to($i ,$j) {
  if ($j < $i) then () else (to($i, $j - 1), $j)
};
```

Lemma 4. *The "`sum`" operator can be expressed in XQ_C^{to}.*

Proof. Following XQ_C^{to} expression is equivalent to "`sum(`e_1`)`":

```
count(
  for $i in e1 return
    for $j in (1 to $i) return 1)
```

Lemma 5. *The "`count`" operator can be expressed in $XQ^{ctr,R}$.*

Proof. We can define a recursive function "`count-nodes`" such that "`count(`e_1`)`" is equivalent to following $XQ^{ctr,R}$ expression:

```
count-nodes(
  for $e in e1 return element {"e"} {()}
)
```

This expression generates as many new nodes as there are items in the input e_1 and then applies a newly defined function "`count-nodes`" to this sequence, which counts the number of distinct nodes in a sequence. This can be done by decreasing the input sequence of the function call to "`count-nodes`" by exactly one node in each recursion step, which is possible since all items in the input sequence of "`count-nodes`" have a different node identity and hence we can remove each step the first node (in document order) of the newly created nodes. Note that, since the count operator returns only atomic values, none of the newly created nodes that were used to count the number of items in the sequence is reachable after applying garbage collection.

Lemma 6. *The "`at`" clause in a for expression can be expressed in XQ_C^{ctr}.*

Proof. The proof is based on the idea that it is possible to transform sequence order into document order by creating new nodes as children of a common parent such that the new nodes will contain all information of each item in the sequence and they are in the same order as the items in the original sequence. It can be shown that we can in XQ_C^{ctr} express the (non-recursive) functions "`pos`" and "`atpos`", which respectively give the position of a node in a document-ordered sequence and returns a node at a certain position in such sequence. If we can define XQ_C^{ctr} functions "`encode`" and "`decode`" (to make sure that we do not lose any information in creating a new node for an item in the result sequence of the "`in`" clause) then the following XQ_C^{ctr} expression is equivalent to the $XQ_{C,at}^{ctr}$ expression "`for` $x `at` $pos `in` e_1 `return` e_2" (where e_1 and e_2 are XQ_C^{ctr} expressions):

```
let $seq    := e1 return
let $newseq := encode($seq) return
for $x in $newseq
return (
  let $pos := pos($x, $newseq) return
  let $x   := decode($x, $seq)
  return e2 )
```

Since the result sequence of e_1, $\$seq$, is used both in the "`in`" clause of the for expression and as actual parameter for the "`decode`" function, we have to assign this result to a new variable, otherwise by simple substitution a node construction that is done in e_1 would be evaluated more than once. Furthermore the expression e_2 is guaranteed to have the right values for the variables "$\$x$" and "$\$pos$" iff the function "`decode`" behaves as desired. We assume that e_2 does not use variables "`$seq`" and "`$newseq`", since they are used in the simulation[8].

We now take a closer look at how to define the functions "`decode`" and "`encode`". The function "`encode`" needs to create a new sequence in which we simulate all items by creating a new node for each item. By adding these nodes as children of a newly constructed element (named "`newseq`") we ensure that the original sequence order is reflected in the document order for the newly constructed sequence. Atomic values are simulated by putting their value as text node in an element which denotes the type of atomic value. Encoding nodes cannot be done by making a copy of them, since this would discard all information we have about the node identity. Therefore we store for a node all information we need to retrieve the node later using the function "`decode`". We do this by storing the root of the node and the position where the node is located in the descendant-or-self list of its root node. We assume that we can define the (non-recursive) XQ_C^{ctr} functions "`pos`" (which we already assumed earlier in this proof), and "`atpos`" (to find the n^{th} node in a sequence of nodes ordered by document order).

Note that none of the previous functions used recursion. Hence we do not actually need functions since we could inline the function definitions in the expressions. Hence the simulation of the "`at`" clause can be written in XQ_C^{ctr}. Furthermore there is no newly created node in the result sequence of the simulation, so all newly created nodes are garbage collected and hence "`at`" can be expressed in XQ_C^{ctr}.

4 Properties of the Fragments

The previous section provided some expressibility results. In this section we prove that certain fragments do not have certain properties, hence they have different degrees of expressive power. For most of the lemmas we do not have

[8] This issue can of course easily be solved by choosing two unused variables to replace these variables.

enough space here to present the complete proofs. We refer the reader for these proofs to our technical report [6].

The first two properties just claim that there are fragments in which it is not possible to distinguish between sequences with the same set or bag representation. To formalize this notion we define set-equivalence and bag-equivalence between evironments and between sequences. In this definition **Set** (**Bag**) maps a sequence to the set (bag) of its items.

Definition 6. *Consider a store St and two environments* $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ *and* $En' = (\mathbf{a}', \mathbf{b}', \mathbf{v}', \mathbf{x}')$ *over the store St. We call En and En′* set-equivalent *iff it holds that* $\mathbf{a} = \mathbf{a}'$, $\mathbf{b} = \mathbf{b}'$, $dom(\mathbf{v}) = dom(\mathbf{v}')$ *and* $\forall s \in dom(\mathbf{v}) : \mathbf{Set}(\mathbf{v}(s)) = \mathbf{Set}(\mathbf{v}'(s))$, *and finally* $\mathbf{x} = \mathbf{x}'$.
The environments En and En′ are called bag-equivalent *iff they are set-equivalent and it holds that* $\forall s \in dom(\mathbf{v}) : \mathbf{Bag}(\mathbf{v}(s)) = \mathbf{Bag}(\mathbf{v}'(s))$

Lemma 7. *Let St be a store,* $En, En' \in EN[XQ^R]$ *two set-equivalent* XQ^R *environments, and e an expression in* XQ^R. *If the result of e is defined for both En and En′, then for each sequence r and r′ for which it holds that* $St, En \vdash e \Rightarrow (St, r)$ *and* $St, En' \vdash e \Rightarrow (St, r')$[9], *it also holds that* $\mathbf{Set}(r) = \mathbf{Set}(r')$.

Proof. (Sketch) This lemma can be proven by induction on the query syntax tree in which each node corresponds to a construct of rules [3−18, 24] in Figure 1.

Lemma 8. *The fragment* XQ_C *does not have the property of Lemma 7.*

Proof. If we consider an environment $En \in EN[XQ^R]$, then $En_1 = En[\mathbf{v}(\text{“}seq\text{”}) \mapsto \langle 1, 1\rangle]$ and $En_2 = En[\mathbf{v}(\text{“}seq\text{”}) \mapsto \langle 1\rangle]$ are two set-equivalent XQ^R environments. The expression “`count($seq)`” returns $\langle 2\rangle$ in the evaluation against En_1 and $\langle 1\rangle$ against En_2.

Lemma 9. *Let St be a store,* $En, En' \in EN[XQ^R_C]$ *two bag-equivalent* XQ^R_C *environments and e be an expression in* XQ^R_C. *If the result of e is defined for both En and En′, then for each sequence r and r′ for which it holds that* $St, En \vdash e \Rightarrow (St, r)$ *and* $St, En' \vdash e \Rightarrow (St, r')$, *it also holds that* $\mathbf{Bag}(r) = \mathbf{Bag}(r')$.

Proof. For all XQ^R expressions we can show similar to the proof of Lemma 7 that evaluations against bag-equivalent environments result in bag-equivalent result sequences.

Lemma 10. *The fragment* XQ_{at} *does not have the property of Lemma 9.*

Proof. If we consider an environment $En \in EN[XQ^R_C]$, then $En_1 = En[\mathbf{v}(\text{“}seq\text{”}) \mapsto \langle 1, 2\rangle]$ and $En_2 = En[\mathbf{v}(\text{“}seq\text{”}) \mapsto \langle 2, 1\rangle]$ are two bag-equivalent XQ^R_C environments, but the evaluation of the expression

[9] Since e does not contain node constructors in its subexpressions, it is easy to see that all subexpressions are evaluated against the same store St and that the result store of all these subexpressions will also be St.

```
for $i at $pos in $seq
return if ($pos=1) then $i else ()
```

returns $\langle 1 \rangle$ when evaluated against environment En_1 and $\langle 2 \rangle$ when evaluated against En_2.

The maximum size of the output for all queries in certain XQuery fragments can be identified as being bounded by a class of functions w.r.t. the input size. For proving the inexpressibility results related to the output size, we introduce following notions for the maximal input and output size for both sequences and items:

Definition 7 (Auxiliary Notations). *Let $St = (V, E, <, \nu, \sigma, \delta)$ be a store, $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ an environment over St and s a sequence over St. The set of atomic values in a sequence s is defined as $A_s = \mathbf{Set}(s) \cap \mathcal{A}$, the set of atomic values in a store St is $A^{St} = (\mathbf{rng}(\nu) \cup \mathbf{rng}(\sigma)) \cap \mathcal{A}$, while the set of atomic values in the environment En is $A^{En} = \bigcup_{s \in \mathbf{rng}(\mathbf{v})} A_s$.*
The size Δ_{St}^{forest} is the size of the forest in St, i.e., $\Delta_{St}^{forest} = |V|$ and Δ_{St}^{tree} is the size of the largest tree of the forest in St, i.e., $\Delta_{St}^{tree} = \max(\bigcup_{n_1 \in V} \{c | c = |\{n_2 | (n_1, n_2) \in E^*\}|\})$[10].
The function **size** *maps an atomic value to the number of cells needed to represent this item on the tape of a Turing Machine.*

Definition 8 (Largest Sequence/Item Sizes). *Consider the evaluation pair $((St, En), (St'', v))$ of a query e, where $St = (V, E, <, \nu, \sigma, \delta)$, $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$, and $\Gamma(St'', \{v\}) = St' = (V', E', <', \nu', \sigma', \delta')$. The* largest input sequence size *is defined as $d_I^s = \max(\{|s| \, | s \in \mathbf{rng}(\mathbf{v})\} \cup \{\Delta_{St}^{tree}\})$. The* largest input item size *is $d_I^i = \max(\{\mathbf{size}(a) | a \in (A^{St} \cup A^{En})\} \cup \{\lceil \log(\Delta_{St}^{forest} + 1) \rceil\})$. The* largest output sequence size *is $d_O^s = \max(\{|v|, \Delta_{St'}^{tree})$. Finally, the* largest output item size *is $d_O^i = \max(\{\mathbf{size}(a) | a \in (A^{St'} \cup A_v)\} \cup \{\lceil \log(\Delta_{St'}^{forest} + 1) \rceil\})$.*

In the definition of the largest sequence sizes we include the size of the largest tree in the store, since one can generate such a sequence by using the descendant-or-self axis. Note that in the definition of the largest item sizes the first set of the union contains all sizes needed to represent the atomic values that occur in the store (or environment) and the second set contains only one value which indicates how much space we need to represent a pointer to a node in the store. Furthermore, we consider in the definition the maximal size for the entire store (including the entire web). This is a theoretical simplification, but it does not have an influence on the input/output size results: if we have to show that the result of a certain evaluation has an upper bound $f(n)$ where n is the input size, then we have to show that this upper bound holds for all input stores and hence also for the "minimal input store", i.e., the store that only contains these input nodes that are actually accessed during the evaluation. Furthermore, the inclusion of the nodes of the output store in the output size is allowed for two

[10] E^* denotes the reflexive and transitive closure of E

reasons. The first reason is that all upper bound functions that we use in our lemmas are at least linear functions and the input nodes that occur in the output store just add a linear factor to the upper bound function. The second reason is that the nodes of the output store that do not occur in the input store have to be reachable by nodes in the result sequence since for each fragment applied garbage collection.
The following inexpressibility results use the observation that the maximum item and/or sequence output size can be bounded by a certain class of functions in terms of the input size.

Lemma 11. *For each evaluation* $St, En \vdash e \Rightarrow (St', v)$ *where* $e \in \mathbf{L}(XQ^{ctr,to})$ *and* $En \in EN[XQ^{ctr,to}]$ *it holds that* $d^i_O \leq p(d^i_I)$ *for some polynomial* p*.*

Proof. (Sketch) For each polynomial p that has $\mathbb{N}$ or $\mathbb{N}^2$ as its domain there always exists an increasing polynomial p' such that p' is an upper bound for p. Therefore we assume all functions that are used as an upper bound in this and following proofs to be increasing functions. We then prove the lemma by induction on the size of the abstract syntax tree of the query q. In this tree the nodes correspond to the $\langle Expr \rangle$ non-terminal of the $XQ^{ctr,to}$ grammar and as a consequence each node corresponds to a construct of rules $[3-18, 23, 26]$ in Figure 1, so we prove the induction step for each of these rules.

Lemma 12. *The fragment* XQ_C *does not have the property of Lemma 11.*

Proof. If we consider the empty store St_0, the environment $En = (\{\}, \{\}, \{(\text{“}\$input\text{”}, \langle 1, \ldots, 1 \rangle)\}, \bot)$, and the expression $e =$ “`count($input)`” where the length of the sequence bound to variable $\$input$ equals k, then the evaluation $St_0, En \vdash e \Rightarrow (St', v)$ has largest input item size $d^i_I = 1$ and output item size $d^i_O = \lceil \log(k+1) \rceil$.

Lemma 13. *For each evaluation* $St, En \vdash e \Rightarrow (St', v)$ *where* $e \in \mathbf{L}(XQ^{ctr}_{at,S})$ *and* $En \in EN[XQ^{ctr}_{at,S}]$ *it holds that* $d^s_O \leq p_1(d^s_I)$ *and* $d^i_O \leq p_2(\log(d^s_I), d^i_I)$ *for some polynomials* p_1 *and* p_2*.*

Proof. (Sketch) This lemma can be proven by induction on the size of the abstract syntax tree of the query q. In this syntax tree the nodes correspond to the $\langle Expr \rangle$ non-terminal of the $XQ^{ctr}_{at,S}$ grammar and as a consequence each node corresponds to a construct of rules $[3-18, 21, 26]$ in Figure 1.

Lemma 14. *The fragment* XQ^{to} *does not have the property of Lemma 13.*

Proof. If we consider the empty store St_0, the environment $En = (\{\}, \{\}, \{(\text{“}\$input\text{”}, \langle k \rangle)\}, \bot)$, and the expression $e =$ “`1 to $input`”, then the evaluation $St_0, En \vdash e \Rightarrow (St', v)$ has maximal input sequence size $d^s_I = O(\log(k))$ and maximal output sequence size $d^s_O = O(k \log(k))$.

Lemma 15. *For each evaluation* $St, En \vdash e \Rightarrow (St', v)$ *where* $e \in \mathbf{L}(XQ^{ctr,to}_{at})$ *and* $En \in EN[XQ^{ctr,to}_{at}]$ *it holds that* $d^s_O \leq p_1(d^s_I, 2^{d^i_I})$ *and* $d^i_O \leq p_2(\log(d^s_I), d^i_I)$ *for some polynomials* p_1 *and* p_2*.*

Proof. (Sketch) Similar to the proof of Lemma 13 this lemma can be proven by induction on the query syntax tree. We already know that for all XQ_{at}^{ctr} expressions there is a polynomial relation between the largest input sequence/item sizes and the largest output sequence/item sizes. Furtermore, the "`to`" expression can construct a sequence of size, at worst, $O(2^{d_I^i})$ with values that need at most $O(d_I^i)$ space. As a consequence is can easily be seen that all $XQ_{at}^{ctr,to}$ expressions have output sizes within the bounds specified by this lemma when evaluated against an $XQ_{at}^{ctr,to}$ environment.

Lemma 16. *The fragment XQ^R does not have the property of Lemma 15.*

Proof. (Sketch) Clearly there are expressions in XQ^R that do not have this property. Indeed, there are expressions that can be simulated in the fragment, such as the power function, that can potentially have largest input item size $d_I^i = \lceil \log(k+1) \rceil$, largest input sequence size $d_I^s = 1$ and largest output sequence size $O(k^k)$.

Finally, we show that the number of possible output values is polynomially bounded by the largest input sequence size and the size of the set of possbile atomic values in the input store and environment.

Definition 9 (Possible Results). *Consider an expression e, a (finite) alphabet $\Sigma \subset \mathcal{A}$ and a number S. The set Res of possible results for evaluations of e constrained by Σ and S is defined as the set of all pairs (St', v) for which it holds that there exists an evaluation $St, En \vdash e \Rightarrow (St', v)$ (with En in the same fragment as e) such that for this evaluation $d_I^s \leq S$ and $A^{St} \cup A^{En} \subseteq \Sigma$.*

In other words, given an expression e, an alphabet Σ and a number S, the set Res contains all possible outputs of the evaluations of e restricted to Σ and S. We will now show that the number of (different) atomic values in this set is polynomially bounded by S and the size of Σ.

Lemma 17. *Consider a (finite) alphabet $\Sigma \subset \mathcal{A}$ and a number S. If $N = |\Sigma|$ then for each XQ_{at}^{ctr} expression e it holds that if Res is the set of possible results for evaluations of e constrained by Σ and S, then the number of atomic values in the possible outputs is polynomially bounded as follows: $\left|\bigcup_{(St',v)\in Res}(A^{St'} \cup A_v)\right| \leq p(N, S)$ for some polynomial p*

Proof. This lemma can be proven by induction on the query syntax tree where each expression corresponds to the $\langle Expr \rangle$ non-terminal of the XQ_{at} grammar and as a consequence each node corresponds to a construct of rules $[3-18, 22]$ of Figure 1.

Lemma 18. *The fragment $XQ_{at,S}$ does not have the property of Lemma 17.*

Proof. Consider the alphabet $\Sigma = \{1, 2, 4, \ldots, 2^{n-1}\}$ and $S = n$. Since "`$x`" can contain any combination of elements of Σ, the result of the sum can be any number between 1 and $2^n - 1$. However, there exists no polynomial p such that for each n it holds that $2^n - 1 \leq p(n, n)$. Hence we know that we cannot express the sum in XQ_{at}.

5 Expressive Power of the Fragments

As we have shown in the two previous sections, some LiXQuery features can be simulated in some fragments that do not contain them and some can not. We will now study the relationships between all 64 fragments in terms of expressive power. In order to be able to compare fragments, we first have to define what "equivalent" and "more expressive" means for XQuery fragments.

Definition 10 (Equivalent Fragments). *Recall that Φ is the set of 64 XQuery fragments as defined in Figure 2. Consider two XQuery fragments XF_1, $XF_2 \in \Phi$.*

- $XF_1 \succeq XF_2 \iff \forall e_2 \in \mathbf{L}(XF_2) : \exists e_1 \in \mathbf{L}(XF_1) : e_1 \sim e_2$
  (XF_1 can simulate XF_2)
- $XF_1 \equiv XF_2 \iff ((XF_1 \succeq XF_2) \wedge (XF_2 \succeq XF_1))$
  (XF_1 is equivalent to XF_2)
- $XF_1 \succ XF_2 \iff ((XF_1 \succeq XF_2) \wedge (XF_1 \not\equiv XF_2))$
  (XF_1 is more expressive than XF_2)

In this definition, the relation $\succeq$ is a partial order on Φ, and $\equiv$ is an equivalence relation on Φ. We use these relations to investigate the relationships between all XQuery fragments defined in Section 2. We show that the equivalence relation $\equiv$ partitions Φ (containing 64 fragments) into 17 equivalence classes. In Figure 4 we show these 17 equivalence classes and their relationships. Each node of the graph represents an equivalence class, i.e., a class of XQuery fragments with the same expressive power. The white and grey nodes represent classes with and without node construction, respectively. Each edge is directed from a more expressive class C_1 to a less expressive one C_2 and points out that each fragment in C_1 is more expressive than all fragments of C_2 (i.e., $\forall XF_1 \in C_1, XF_2 \in C_2 : XF_1 \succ XF_2$).

Theorem 1. *For the graph in Figure 4 and for all fragments $XF_1, XF_2 \in \Phi$ it holds that*

- $XF_1 \equiv XF_2 \iff$ *XF_1 and XF_2 are within the same node*
- $XF_1 \succ XF_2 \iff$ *there is a directed path from the node containing XF_1 to the node containing XF_2*

Proof. (Sketch) Informally, the dotted borders in Figure 4 divide the set of fragments (Φ) in two parts: one in which the attribute that labels the border can be expressed and one in which this attribute cannot be expressed. The arrows that cross the borders all go in one direction, i.e., from the set of fragments where you can express a certain construct to the the set where you cannot express it. We call the set of fragments that can simulate the construct the right-hand side of the border and the other set the left-hand side of the border. The correctness of the dotted borders can be proven by showing that you can express something in the least expressive fragment of the right-hand side that you cannot express in the most expressive fragment of the left-hand side. In order to prove this, we need the lemmas of Section 3 and 4. All previous results can now be combined to complete the proof:

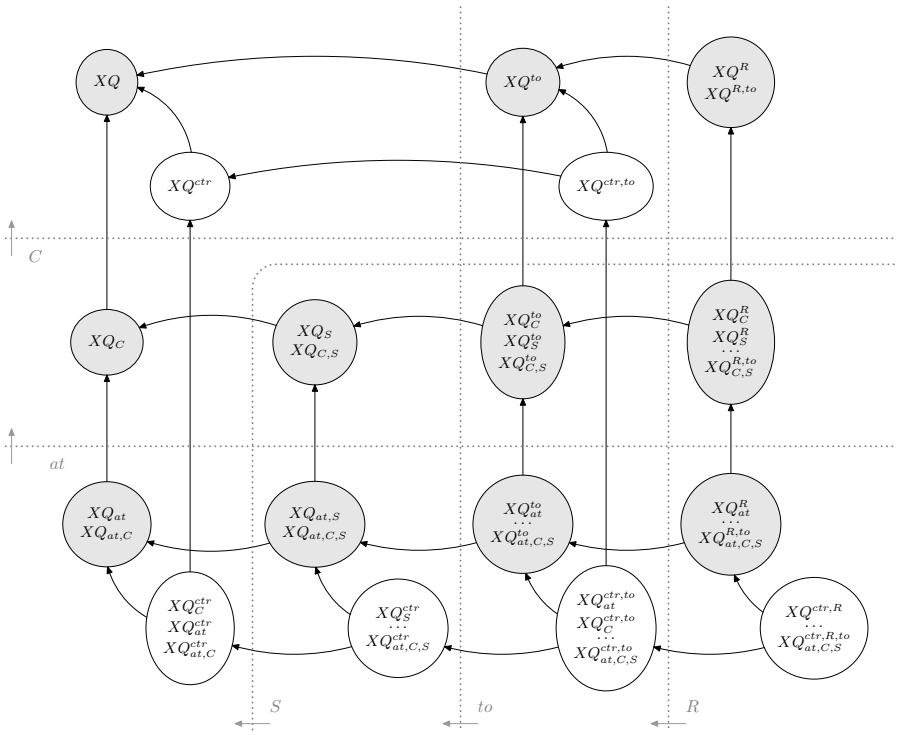

Fig. 4. Equivalence classes of XQuery fragments

- If XF_1 and XF_2 are in the same node then it follows that they are equivalent: This can easily be shown by the lemmas from Section 3.
- If XF_1 and XF_2 are equivalent then they occur in the same node:
  Suppose that XF_1 and XF_2 are not in the same node. There are two possibilities: if one of the two fragments contains a node constructor (suppose XF_1) and the other (XF_2) does not then you obviously cannot simulate the node construction in XF_2. Else it follows from the figure that they are seperated by a dotted border and hence we know that there is something in one fragment that you cannot express in the other fragment, so $XF_1 \not\equiv XF_2$.
- If there is a directed path from the node containing XF_1 to the node containing XF_2 then we know that $XF_1 \succeq XF_2$ and since XF_1 and XF_2 appear in a different node they are not equivalent, so $XF_1 \succ XF_2$:
  This follows from the fact that there is a fragment XF_1' equivalent to XF_1 and XF_2' equivalent to XF_2 such that $\mathbf{L}(XF_2') \subseteq \mathbf{L}(XF_1')$.
- If $XF_1 \succ XF_2$ then there is a directed path from the node containing XF_1 to the node containing XF_2:
  Suppose that $XF_1 \succ XF_2$ and there is no directed path from XF_1 to XF_2. Then either there is a directed path from XF_2 to XF_1 such that $XF_2 \succ XF_1$

and hence $XF_1 \not\succ XF_2$ or there is no directed path at all between the nodes of both fragments. In this case we know by inspecting Figure 4 that there are (at least) two borders seperating the nodes of both fragments where for the first border XF_1 is in the more expressive set of fragments and for the second border XF_2 is in the more expressive set of fragments. Hence XF_1 and XF_2 are incomparable so $XF_1 \not\succ XF_2$.

6 Conclusion

We investigated the expressive power of XQuery fragments in order to outline which features really add expressive power and which ones simplify queries already expressible. The main results of this paper outline that, using six attributes (count, sum, to, at, ctr and recursion), we can define 64 XQuery fragments, which can be divided into 17 equivalence classes, i.e., classes including fragments with the same expressive power. We proved the 17 equivalence classes are really different and own a different degree of expressive power.

References

1. M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *ICDT 2003*, pages 79–95, 2003.
2. S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C Working Draft, 2005. Available at `http://www.w3.org/TR/xquery/`.
3. D. Draper, P. Frankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, 2005. Available at `http://www.w3.org/TR/xquery-semantics/`.
4. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS 2003*, pages 179–190, 2003.
5. J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *XSym 2004*, pages 5–20, 2004.
6. J. Hidders, J. Paredaens, R. Vercammen, and S. Marrara. Expressive power of recursion and aggregates in XQuery. Technical Report TR2005-05, University of Antwerp, 2005. Available at `http://www.adrem.ua.ac.be/pub/TR2005-05.pdf`.
7. M. Kay, N. Walsh, and H. Zongaro. XSLT 2.0 and XQuery 1.0 serialization. W3C Working Draft, 2005. Available at `http://www.w3.org/TR/xslt-xquery-serialization/`.
8. S. Kepser. A simple proof of the Turing-completeness of XSLT and XQuery. In T. Usdin, editor, *Extreme Markup Languages 2004*. IDEAlliance, 2004. Available at `http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2%004Kepser01.html`.
9. C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. In *PODS 2005*, pages 84–97, 2005.
10. W. Le Page, J. Hidders, P. Michiels, J. Paredaens, and R. Vercammen. On the expressive power of node construction in XQuery. In *WebDB 2005*, pages 85–90, 2005. Available at `http://webdb2005.uhasselt.be/webdb05_eproceedings.pdf`.

11. L. Libkin. Expressive power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003.
12. J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.
13. S. Vansummeren. Deciding well-definedness of XQuery fragments. In *PODS 2005*, pages 37–48, 2005.