On the Expressibility of Functions in XQuery Fragments

Jan Hidders [a] Stefania Marrara [b] Jan Paredaens [a]
Roel Vercammen [a,1]

[a] *University of Antwerp, Dept. Math and Computer Science, Middelheimlaan 1, BE-2020 Antwerp, Belgium*

[b] *Universitá degli Studi di Milano, Dipartimento di Tecnologie dell'Informazione, Via Bramante 65, I-26013 Crema (CR), Italy*

Abstract

XQuery is a powerful XML query language with many features and syntactic constructs. 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 and derive some properties of the fragments within these equivalence classes.

1 Introduction

XQuery [3], the W3C standard query language for XML, is a very powerful query language which is known to be Turing Complete [7]. 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

Email addresses: `jan.hidders@ua.ac.be` (Jan Hidders), `marrara@dti.unimi.it` (Stefania Marrara), `jan.paredaens@ua.ac.be` (Jan Paredaens), `roel.vercammen@ua.ac.be` (Roel Vercammen).

[1] Roel Vercammen is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant number 33581.

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 [5], and Marx [12] 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 query evaluation for nonrecursive XQuery fragments without aggregation functions, position information in `for` loops and sequence generation [8], Koch and Benedikt study the relationship between similar fragments and fragments of first-order logic with counters [2], Vansummeren looks into the well-definedness problem for XQuery fragments [14], and the expressive power of the node construction in XQuery is studied in [9].

In this article we investigate the expressive power of XQuery fragments in a similar fashion as was done for the relational algebra [13] 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 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 what 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 [6], which is conjectured to be as expressive as XQuery.

This article is organized as follows. Section 2 introduces the syntax and the semantics of the different XQuery fragments that we are going to analyze. Section 3 presents the main result of the article, i.e., we partition the set of fragments into classes of fragments with the same expressive power. In Section 4 we present some expressibility results for these fragments and in Section 5 we show some properties that hold for some of the fragments. These results are combined in Section 6, where we prove the main result of the paper, which was presented in Section 3. Finally, Section 7 outlines the conclusions of our work.

2 XQuery Fragments

In this section we introduce the XQuery fragments whose expressive power we study in the article. We use LiXQuery [6] as a formal foundation, which is a light-weight sublanguage of XQuery, fully downwards compatible with

XQuery. The syntax and an informal description of the semantics of each of the XQuery fragments is given in Subsection 2.1. In Subsection 2.2 we introduce the essential notions of the formal semantics of LiXQuery that we use throughout the paper. Finally, in Subsection 2.3 we show how some typical XQuery constructs can be expressed in XQ^*.

2.1 Syntax and Informal Semantics

The syntax of the fragment XQ is shown in Fig. 1, by rules S1 to S18, to which we will refer from now on as (S1-S18). Expressions that 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 (S2) for XQ. This syntax is an abstract syntax, i.e., it assumes that extra brackets and precedence rules are added for disambiguation.

We briefly and informally discuss the semantics of the constructs in XQ. In rule (S5) the expression `()` returns the empty sequence. In rule (S6) the built-in functions are declared. 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 `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 `root()` returns for a node the root of the tree it belongs to. The function `concat()` concatenates strings. The functions `true()` and `false()` return the boolean values **true** and **false**, respectively. The function `not()` inverts the the boolean value of its argument.

In rule (S7) the expression `if (`e_1`) then` e_2 `else` e_3 denotes the usual conditional expression. In rule (S8) the expression `for $`x `in` e_1 `return` e_2 expresses iteration where the result is computed by iterating over each element in the sequence that is the result of e_1, binding this element to `$x` and evaluating e_2, and finally concatenating all the sequences that resulted from the evaluation of e_2. For example, the expression `for $x in (1, 2, 3) return ($x, $x)` returns the sequence $\langle 1, 1, 2, 2, 3, 3\rangle$. In rule (S9) the expression `let $`x `:=` e_1 `return` e_2 returns the result of evaluating e_2 with `$`x bound to the result of e_1. In rule (S10) the expression e_1`,`e_2 denotes sequence concatenation that returns the concatenation of the results of e_1 and e_2. For example, if e_1 returns $\langle 1, 2\rangle$ and e_2 returns $\langle 3, 4\rangle$, then e_1`,`e_2 returns $\langle 1, 2, 3, 4\rangle$. In the semantics of XQuery single values such as numbers strings and nodes are assumed to

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

(S1) ⟨*Query*⟩ → (⟨*FunDecl*⟩ “`;`”)* ⟨*Expr*⟩

(S2) ⟨*Expr*⟩ → ⟨*Var*⟩ | ⟨*BuiltIn*⟩ | ⟨*IfExpr*⟩ | ⟨*ForExpr*⟩ | ⟨*LetExpr*⟩ | ⟨*Concat*⟩ | ⟨*AndOr*⟩ | ⟨*ValCmp*⟩ | ⟨*NodeCmp*⟩ | ⟨*ArithmExpr*⟩ | ⟨*Step*⟩ | ⟨*Path*⟩ | ⟨*Literal*⟩ | ⟨*EmpSeq*⟩ | ⟨*Constr*⟩ | ⟨*TypeSw*⟩ | ⟨*FunCall*⟩ | ⟨*Count*⟩ | ⟨*Sum*⟩ | ⟨*SeqGen*⟩

(S3) ⟨*Var*⟩ → “`$`” ⟨*Name*⟩

(S4) ⟨*Literal*⟩ → ⟨*String*⟩ | ⟨*Integer*⟩

(S5) ⟨*EmpSeq*⟩ → “`()`”

(S6) ⟨*BuiltIn*⟩ → “`string(`” ⟨*Expr*⟩ “`)`” | “`xs:integer(`” ⟨*Expr*⟩ “`)`” | “`doc(`” ⟨*Expr*⟩ “`)`” | “`name(`” ⟨*Expr*⟩ “`)`” | “`root(`” ⟨*Expr*⟩ “`)`” | “`concat(`” ⟨*Expr*⟩, ⟨*Expr*⟩ “`)`” | “`true()`” | “`false()`” | “`not(`” ⟨*Expr*⟩ “`)`”

(S7) ⟨*IfExpr*⟩ → “`if (`” ⟨*Expr*⟩ “`)`” “`then`” ⟨*Expr*⟩ “`else`” ⟨*Expr*⟩

(S8) ⟨*ForExpr*⟩ → “`for`” ⟨*Var*⟩ (⟨*AtExpr*⟩)? “`in`” ⟨*Expr*⟩ “`return`” ⟨*Expr*⟩

(S9) ⟨*LetExpr*⟩ → “`let`” ⟨*Var*⟩ “`:=`” ⟨*Expr*⟩ “`return`” ⟨*Expr*⟩

(S10) ⟨*Concat*⟩ → ⟨*Expr*⟩ “`,`” ⟨*Expr*⟩

(S11) ⟨*AndOr*⟩ → ⟨*Expr*⟩ (“`and`” | “`or`”) ⟨*Expr*⟩

(S12) ⟨*ValCmp*⟩ → ⟨*Expr*⟩ (“`=`” | “`<`”) ⟨*Expr*⟩

(S13) ⟨*NodeCmp*⟩ → ⟨*Expr*⟩ (“`is`” | “`<<`”) ⟨*Expr*⟩

(S14) ⟨*ArithmExpr*⟩ → ⟨*Expr*⟩ (“`+`” | “`-`” | “`*`” | “`idiv`”) ⟨*Expr*⟩

(S15) ⟨*Step*⟩ → “`.`” | ⟨*Name*⟩ | “`@`” ⟨*Name*⟩ | “`*`” | “`@*`” | “`text()`”

(S16) ⟨*Path*⟩ → ⟨*Expr*⟩ (“`/`” | “`//`”) ⟨*Expr*⟩

(S17) ⟨*TypeSw*⟩ → “`typeswitch (`” ⟨*Expr*⟩ “`)`” (“`case`” ⟨*Type*⟩ “`return`” ⟨*Expr*⟩)+ “`default`” “`return`” ⟨*Expr*⟩

(S18) ⟨*Type*⟩ → “`xs:boolean`” | “`xs:integer`” | “`xs:string`” | “`element()`” | “`attribute()`” | “`text()`” | “`document-node()`”

(S19) ⟨*Count*⟩ → “`count(`” ⟨*Expr*⟩ “`)`”

(S20) ⟨*Sum*⟩ → “`sum(`” ⟨*Expr*⟩ “`)`”

(S21) ⟨*AtExpr*⟩ → “`at`” ⟨*Var*⟩

(S22) ⟨*SeqGen*⟩ → ⟨*Expr*⟩ “`to`” ⟨*Expr*⟩

(S23) ⟨*FunCall*⟩ → ⟨*Name*⟩ “`(`” (⟨*Expr*⟩ (“`,`” ⟨*Expr*⟩)*)? “`)`”

(S24) ⟨*FunDecl*⟩ → “`declare function`” ⟨*Name*⟩ “`(`” (⟨*Var*⟩ (“`,`” ⟨*Var*⟩)*)? “`)`” “{” ⟨*Expr*⟩ “}”

(S25) ⟨*Constr*⟩ → “`document`” “{” ⟨*Expr*⟩ “}” | “`element`” “{” ⟨*Expr*⟩ “}” “{” ⟨*Expr*⟩ “}” | “`attribute`” “{” ⟨*Expr*⟩ “}” “{” ⟨*Expr*⟩ “}” | “`text`” “{” ⟨*Expr*⟩ “}” |

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

be identical with a singleton sequence that contains them. So the expression `(1, 2)` in fact denotes the concatenation of the sequences $\langle 1 \rangle$ and $\langle 2 \rangle$, which is indeed the sequence $\langle 1, 2 \rangle$.

In rule (S11) we find the boolean conjunction and disjunction. The rule (S12) 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 (S13) gives the comparison operators for nodes where “`is`” tests the equality of nodes and “`<<`” compares nodes in

document order. Rule (S14) defines basic arithmetic operations.

In rule (S15) we find the basic steps in path expressions that navigate starting from a so-called *context item* which is either a node or a basic value. The expression `.` simply returns the context item. An expression of the form N with N a valid element name returns all children of the context item which are element nodes with the name N. Likewise the expression `@`N retrieves the attribute nodes under the context item with name N. If the wildcard `*` is used for N then these operations return respectively all element nodes and all attribute nodes under the context item. Finally the `text()` expression returns all text nodes under the context item. In rule (S16) the expressions for defining composing path expressions are defined. An expression e_1`/`e_2 returns the sequence of nodes that is obtained by evaluating e_1 and then for each node in its result evaluate e_2 with this node as the context item, and finally take the union of all nodes in the results of e_2. The resulting sequence is without duplicates and sorted in document order, i.e., as the associates elements, attributes, etc. are encountered in the document. The expression e_1`//`e_2 has the same semantics except that we evaluate e_2 not only for all the nodes in the result of e_1 but also for all their descendants.

Finally in rule (S17) the type-switch expression is defined, which allows us to check the type of a node or basic value. For an example consider the following expression:

```
for $x in ($y/@*, $y/*, $y/text())
return (
  typeswitch ($x)
    case element() return string($x)
    case attribute() return concat("@",string($x))
    default return "text" )
```

If it is evaluated while `$y` is bound to the root node of an XML fragment `<test id="5"> <result/> scheduled </test>` then the result will be the sequence $\langle$`"@id"`, `"result"`, `"text"`$\rangle$.

The rules (S19-S25) define the constructs that we will use to define the different fragments of XQuery that we will consider. The rules (S19) and (S20) introduce the operations for counting and adding the elements of a sequence. The extension of the fragment XQ with these operations is denoted as XQ_C and XQ_S respectively, and as $XQ_{C,S}$ if both operations are allowed. The motivation for making these operations optional is that although they are in practice often used their added expressive power is often not easy to establish. See for example [11] for a discussion of this for SQL. In XQ these operations seem to add expressive power because they are the only ones that can distinguish sequences with the same elements. The distinction between `count()` and

`sum()` is also interesting because the first ignores the numbers in a sequence whereas the second takes them into account.

The rule (S21) defines the `at` clause which can be used to bind a variable to the position of the current item in a sequence during the iteration of the `for` expression. As an example the expression `for $x at $y in ("a", "b", "c") return ($y, $x)` returns the sequence $\langle 1, \texttt{"a"}, 2, \texttt{"b"}, 3, \texttt{"c"} \rangle$. The extension of XQ with this construct is denoted as XQ_{at}. The reason to make this construction optional is that it seem to be the only construct that can refer to the position of an item in a sequence.

The rule (S22) defines expressions of the form e_1 `to` e_2 which construct a sequence of numbers beginning from the result of e_1 up to and including the result of e_2. For example the expression `1 to 4` returns the sequence $\langle 1, 2, 3, 4 \rangle$. The extension of XQ with this construct is denoted XQ_{at}. It is an interesting operation since it is one of the few operations that allows the construction of large results where the size depends not so much on the size of the input but on the magnitude of the numbers in the input.

The rules (S23-S24) allow the application and definition of functions, and specifically recursive functions. An expression of the form N`(`e_1`, ..., `e_n`)` applies the function with name N to the results of $e_1, \ldots, e_n$. An expression of the form `declare function` N`(`v_1`,...,`v_n`)`$\{e\}$ declares a function with name N, formal arguments $v_1, \ldots, v_n$ and body e. Observe that if we disallow recursion these constructs would not increase the expressive power of the language since all function calls could be in-lined. The extension of XQ with these constructs is denoted as XQ^R. Adding recursion will obviously greatly increase the expressive power and make the language computationally complete for operations over strings and integers but, as will be shown later, there may still be operations over XML data and/or sequences that cannot be expressed. For example, XQ^R cannot express the `count()` function, which can be informally explained by saying that there is no operation in XQ^R that can distinguish sequences that contain the same elements. There are for example no operations in XQ that allow us to select the head or the tail of a list, as is possible in LISP. Clearly with such operations the `count()` could have been expressed with recursive functions. This will be discussed more formally later on.

Finally the rule (S25) introduces expressions for creating new nodes. An expression `document`$\{e\}$ creates a new document node with as its contents a deep copy of the resulting sequence of e. An expression `element`$\{e_1\}\{e_2\}$ creates a new element node with a name computed by e_1 and a deep copy of the result of e_2 as its contents. Note that these contents include any attribute nodes that should be associated with the new node. Finally, an expression `text`$\{e\}$ constructs a new text node with the string computed by e as its string value. The extension of XQ with these expressions is denoted as XQ^{ctr}.

XQ	(S1-S18)	*basic fragment*
XQ_C	+ (S19)	`count(`e`)`
XQ_S	+ (S20)	`sum(`e`)`
XQ_{at}	+ (S21)	`for $`x `at $`y `in` e_1 `return` e_2
XQ^{to}	+ (S22)	e_1 `to` e_2
XQ^R	+ (S23-S24)	*recursive functions*
XQ^{ctr}	+ (S25)	*node construction*

Fig. 2. Definition of XQuery fragments

Clearly these operations add expressive power since without them one cannot construct new nodes, but they also do so if we only consider functions that return no new nodes in the final result. For example, in XQ we cannot distinguish the sequences $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$, but if we have node construction then we can convert them to the fragments `<a> <b>1</b> <b>2</b> </b>` and `<a> <b>2</b> <b>1</b> </b>` which can be distinguished. Informally this might be explained by saying that node construction can give us access to the sequence order by converting it into document order. This will be discussed in more detail later on.

Summarizing, we use 6 attributes for XQ fragments, namely C, S, at, to, R and ctr (cf. Fig. 2 for the syntax of the attributed fragments). These annotations may be freely combined such that, for example, $XQ^{ctr}_{C,S,at}$ denotes the fragment with node construction, `count()`, `sum()` and the `at` clause in `for` expressions. For the largest fragment, $XQ^{R,to,ctr}_{C,S,at}$ expressed by rules (S1-S25), we additionally introduce the short-hand XQ^*. Since there are 6 attributes they define 64 fragments of XQuery. The main goal of this article is to investigate and to compare the expressive power of these fragments.

The following auxiliary definitions are used throughout the article:

Definition 2.1 *The language* $\mathbf{L}(XF)$ *of an XQuery fragment* XF *is the (infinite) set of all expressions that can be generated by the syntax rules for this fragment with* $\langle Query\rangle$ *as start symbol. The set* Φ *is the set of all 64 XQuery fragments defined in Fig. 2.*

Similar to LiXQuery, we ignore static typing and do not consider namespaces [3], comments, processing instructions, and entities.

[3] 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

The semantics of our XQuery fragments is the same as that of LiXQuery and downwards compatible with the XQuery Formal Semantics [4]. Expressions are evaluated against an *XML store* which contains XML fragments created as intermediate results, and all the web documents. 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. We now introduce some notations that we use in this article. The set $\mathcal{S}$ is the set of all strings, $\mathcal{N} \subseteq \mathcal{S}$ is the set of strings that may be used as tag names, $\mathbb{Z}$ is the set of all integers, $\mathbb{N} \subset \mathbb{Z}$ is the set of all positive integers, $\mathcal{B}$ is the set $\{\mathbf{true}, \mathbf{false}\}$ of boolean values, and $\mathcal{A} = \mathbb{Z} \cup \mathcal{B} \cup \mathcal{S}$ is the set of all atomic values. The set $\mathcal{V}$ is the set of all nodes. The domain and the range of a function f are denoted by respectively $\mathbf{dom}(f)$ and $\mathbf{rng}(f)$.

Definition 2.2 (XML Store) *An XML store is a 6-tuple* $St = (V, E, \ll, \nu, \sigma, \delta)$ *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 directed acyclic 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*;*
- $\ll$ *is a total order on the nodes of* V*;*
- $\nu : V^e \cup V^a \to \mathcal{N}$ *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 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 these documents are in the store.*

The following properties have to hold for an XML store:

(1) each document node of V^d *is the root of a tree and has only one child which is an element node;*
(2) attribute nodes of V^a *and text nodes of* V^t *do not have any children;*

[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.

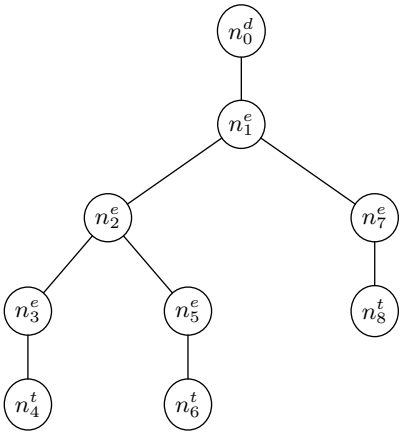

Fig. 3. XML tree of Example 2.1

(3) the $\ll$ -order is the document order over (V, E) such that for all trees it corresponds to its preorder, i.e.:
  (a) if $(n_1, n_2) \in E^*$ and $n_1 \neq n_2$ then $n_1 \ll n_2$;
  (b) if $(m, n_1), (m, n_3) \in E$, $(n_1, n_2) \in E^*$, and $n_1 \ll n_3$ then $n_2 \ll n_3$;

(4) nodes of two different trees are not "mixed" in document order, i.e., if $(n_1, n_2), (n_1, n_4) \in E^*$ and $n_2 \ll n_3 \ll n_4$ then $(n_1, n_3) \in E^*$.

(5) in the $\ll$ -order attribute children precede the element and text children, i.e., if $(m, n_1), (m, n_2) \in E$, $n_1 \ll n_2$ and $n_2 \in V^a$ then $n_1 \in V^a$;

(6) there are no adjacent text children, i.e., if (m, n_1), $(m, n_2) \in E$, $n_1, n_2 \in V^t$, and $n_1 \ll n_2$ then there is an $n_3 \in V^e$ with $n_1 \ll n_3 \ll n_2$;

(7) for all text nodes n_t of V^t holds $\sigma(n_t) \neq$ "";

(8) 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 $\nu(n_1) \neq \nu(n_2)$.

Note that this definition slightly differs from our original definition of an XML Store [6], since we now have included the document order in the store instead of the sibling order.

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 $l_1 \circ l_2$. A sequence over a store St is a sequence of items of St. We now give an example to illustrate the definition of a store.

Example 2.1 *Let $St = (V, E, <, \nu, \sigma, \delta)$ be an XML store with one document "doc.xml", which is shown in Fig. 3.*

- *The set of nodes V consists of $V^d = \{n_0^d\}$, $V^e = \{n_1^e, n_2^e, n_3^e, n_5^e, n_7^e\}$, $V^t = \{n_4^t, n_6^t, n_8^t\}$, $V^a = \emptyset$.*
- *The set of edges is $E = \{(n_0^d, n_1^e), (n_1^e, n_2^e), (n_1^e, n_7^e), (n_2^e, n_3^e), (n_2^e, n_5^e), (n_3^e, n_4^t), (n_5^e, n_6^t), (n_7^e, n_8^t)\}$.*
- *The document order $\ll$ is defined by $n_0^d \ll n_1^e \ll n_2^e \ll n_3^e \ll n_4^t \ll n_5^e \ll$*

$n_6^t \ll n_7^e \ll n_8^t$.

- *Furthermore* $\nu(n_1^e) =$ `"a"`, $\nu(n_2^e) = \nu(n_7^e) =$ `"b"`, $\nu(n_3^e) = \nu(n_5^e) =$ `"c"`, $\sigma(n_4^t) =$ `t1`, $\sigma(n_6^t) =$ `t2`, $\sigma(n_8^t) =$ `t3`, *and* $\delta(\text{"}doc.xml\text{"}) = n_0^d$.

The store in this example contains only one tree, which models the following XML fragment: `<a><b><c>t1</c><c>t2</c></b><b>t3</b></a>`.

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 2.3 (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 ;* $\mathbf{b} : \mathcal{N} \to \mathbf{L}(XQ^*)$ *a partial function that maps a function name to the body of the function;* $\mathbf{v} : \mathcal{N} \to (\mathcal{V} \cup \mathcal{A})^*$ *a partial function that maps variable names to their values; and* $\mathbf{x}$ *which is undefined or an item of St and indicates the context item*[5] *.*

Note that we do not allow multiple functions to have the same name, whereas this is allowed in XQuery as long as these functions have a different arity. 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 over St', i.e., 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 [6].

2.3 Discussion

We conclude this section by giving more motivation for the XQuery sublanguage XQ^* that we consider in this paper. First we compare XQ^* with LiXQuery to show that the few syntactic changes from LiXQuery in XQ^* does not affect the expressive power. Next, we show how some typical XQuery constructs can be expressed in XQ^*.

[5] The context item is a node against which steps in path expressions are evaluated. It is used in rules (S15-S16) of Fig. 1

2.3.1 Comparison with LiXQuery

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 these features can be easily simulated in XQ^*, as we will show now. Hence, we claim that XQ^* has the same expressive power as LiXQuery.

The first feature that is left out is the union e_1`|`e_2 which concatenates the results of e_1 and of e_2, removes any duplicates and sorts the result in document order. This is equivalent to (`(`e_1`,`e_2`)/.`, because the `/.` expression at the end removes the duplicate nodes and sorts the result sequence by document order.

The second feature in LiXQuery, but not in XQ^*, is the filter expression e_1`[`e_2`]`. Its semantics is that we iterate over the result of e_1 and select each node for which e_2 evaluates to **true** while taking this node as the context item. For example, `(1, 2, 3, 4)[. > 2]` returns the sequence $\langle 3, 5\rangle$. A simulation of e_1`[`e_2`]` can be done in two steps. First, we construct e_2' from e_2 by replacing every subexpression that depends upon the context item from e_1, i.e., is not nested in the e_2' of e_1'`[`e_2'`]`, e_1'`/`e_2' or e_1'`//`e_2', is replaced as follows: the expression `.` is replaced with `$dot`, and all expressions e of the forms N, `*`, `@`N, `@*` and `text()` are replaced with `$dot/`e. After this we can simulate e_1`[`e_2`]` with:

```
for $dot in e1 return (
  if (e2') then $dot else ())
```

Note that another possible simulation is e_1`/(if (`e_2`) then . else ())` but only if the result contains only nodes and no basic values.

The third feature are the functions `position()` and/or `last()`. They can be used in every expression that is evaluated for a context item that is taken from some sequence. These are the subexpressions e_2 in expressions of the forms e_1`/`e_2, e_1`//`e_2 and e_1`[`e_2`]`. The meaning of these functions is for e_1`/`e_2 and e_1`[`e_2`]` the same: `position()` refers to the position of the context item in the result sequence of e_1 and `last()` refers to the position of the last item in this sequence, i.e., the length of the sequence. For example, `("a", "b", "c", "d")[position() > 2]` returns $\langle$`"c"`, `"d"`$\rangle$ and `("a", "b", "c", "d")[position() = last()]` returns $\langle$`"d"`$\rangle$. Another example is the expression `a/(if (position() = 2) then . else ())` which returns the second `a` element child of the context item. For expressions of the form e_1`//`e_2 the semantics of `position()` and `last()` are similar except that the context sequence to which `position()` and `last()` refer is differently defined. Recall that for each node n in the result of e_1 the evaluation iterates over n and its descendants and evaluates e_2. During the evaluation of e_2 the sequence consisting of n and its descendants is assumed to be the context sequence.

We can simulate e_1[e_2] with `position()` and/or `last()` in e_2 as follows, assuming e_2' is constructed from e_2 as previously explained for the preceding simulation of e_1[e_2] plus that in addition `position()` and `last()` are replaced by the variables `$pos` and `$last` [6]:

```
let $seq := e1 return
let $last := count($seq) return
for $dot at $pos in e1 return
  if (e2') then $dot else ()
```

The simulation of e_1/e_2 with `position()` and/or `last()` in e_2 is similar except that here the results of e_2 are returned and the final result is sorted in document order by applying `/.`:

```
let $seq := e1 return
let $last := count($seq) return
(for $dot at $pos in e1 return e2')/.
```

The simulation of $e_1//e_2$ with `position()` or `last()` in e_2 is accomplished by rewriting it as e_1`/((.//.)/`e_2`)` and reducing it thus to the previous case.

The last feature that is not in XQ^* is the parent step `..` that retrieves the parent of the context item and which we can simulate as follows:

```
for $dot in root(.)//. return (
  for $chl in ($dot/*, $dot/text(), $dot/@*) return (
    if ($chl is .) then $dot else () ) )
```

The variable `$dot` iterates over all nodes of the tree that contains the context item and selects a node if one of its children (element, attribute, or text nodes) equals the context item.

Since we have shown that all features that are in LiXQuery but not in XQ^* can be simulated in XQ^*, it follows that XQ^* has the same expressive power as LiXQuery.

2.3.2 Simulation of (other) XQuery Features

We can simulate many XQuery features that are not in XQ^* by using a sublanguage of XQ^*. For example, since the "=" comparison has an existential semantics, the emptiness test `empty(`e_1`)` can be expressed in XQ as follows:

```
if (1 = (for $y in e1  return 1)) then true() else false()
```

[6] We assume, w.l.o.g., that `$pos` and `$last` do not occur in e_2.

Furthermore, all XPath axes can be simulated in XQ. We illustrate this claim by giving the simulation of `following-sibling::node()`:

```
for $dot in ../(*, text()) return (
    if (. << $dot) then $dot else () )
```

The final XQuery feature that we use to illustrate the claim that most typical XQuery expressions can be expressed in XQ^* is the `order by` clause:

`for` $x `at` $y `in` e_1 `order by` e_2 `return` e_3

We show that we can simulate this expression in the fragment $XQ_{C,at}$, i.e., without using recursive functions. Assume that the expression in e_2 yields exactly one item when evaluated against an item in the result of e_1. If the evaluation of e_1 yields n items, then the sequence that we have to order according to the order by clause e_2 has also n items. We will create a permutation of the numbers 1 to n in a variable `$ordByPos` such that the i^{th} item in this sequence is j iff the j^{th} item in the result sequence of e_1 is the smallest item in the result of e_1 with at least $i-1$ items smaller or equal. In order to obtain a stable order, we also have to incorporate the position of the items in the result of e_1. The simulation of the `order by` can then be performed as follows:

```
let $inExpr := e1 return
let $unordBy := (for $x at $y in $inExpr return (e2)) return
let $ordByPos := (
  for $n at $p in $unordBy return (
    for $n2 at $p2 in $unordBy return (
      let $smaller := (for $n3 at $p3 in $unordBy
                       return if ($n3 < $n2) then $n3 else ())
      let $equalBef := (for $n3 at $p3 in $unordBy
                        return if (($n3 = $n2) and ($p3 < $p2))
                               then $n3 else ())
      if (count(($smaller,$equalBef)) = $p) then $p2 else ()
    )
  )
) return
let $ordInExpr := (
  for $x at $pos in $inExpr return
    for $y in $ordByPos return
      if ($y = $pos) then $x else ()
) return
for $x at $pos in $ordInExpr return
let $y := (for $xx at $yy in $ordByPos
           return if ($xx = $pos) then $yy else ())
return e3
```

3 Expressive Power of the Fragments

The main contribution of this paper consists of showing that some $XQ*$ features can be simulated in some fragments that do not contain them and some can not. We 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.

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 3.1 (Garbage Collection) *The garbage collected version* $\Gamma_s(St)$ *of a store* $St = (V, E, \nu, \sigma, \delta)$ *relative to a sequence* s *is the store obtained 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 3.2 (XQuery function) *The XQuery function corresponding to an expression* e *is* $\{((St, \mathbf{v}), (\Gamma_v(St'), v)) \mid St, (\emptyset, \emptyset, \mathbf{v}, \perp) \vdash e \Rightarrow (St', v)\}$*. 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 [4]. 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, which can then inspect the node identities that are returned.

Definition 3.3 (Equivalent Fragments) *Recall that* Φ *is the set of XQuery fragments as defined in Fig. 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)

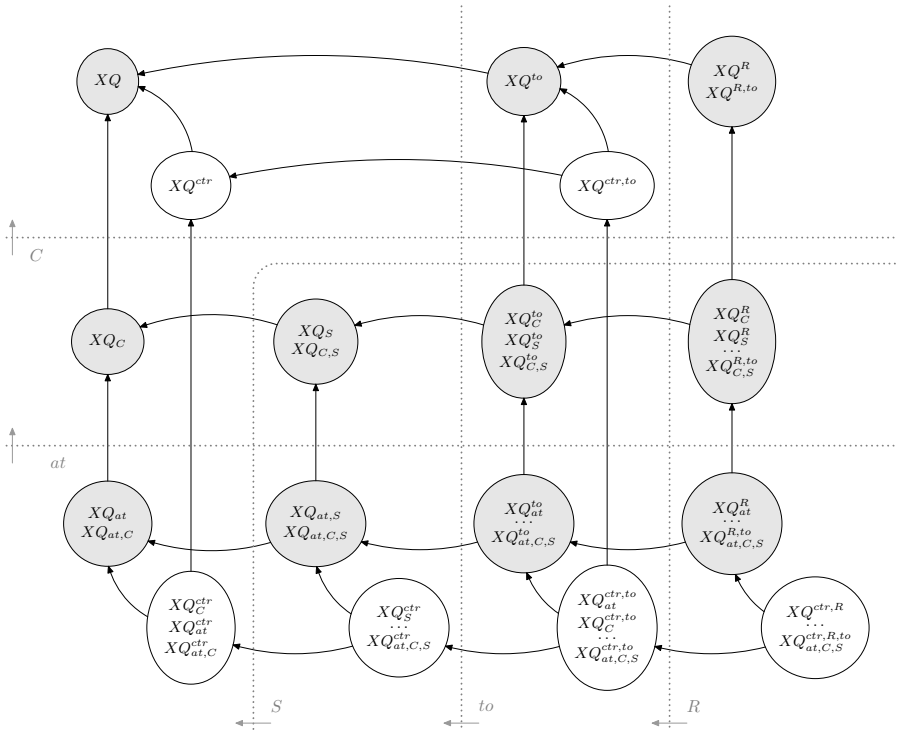

Fig. 4. Equivalence classes of XQuery fragments

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 Fig. 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 3.1 *For the graph in Fig. 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

The proof of this theorem is given in Section 6. The lemmas of Section 4 will be used to show that all fragments that are in the same node have the same expressive power and the lemmas of Section 5 to show that all different nodes in the graph have a different degree of expressive power.

Informally, the dotted borders in Fig. 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 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 something can be expressed in the least expressive fragments of the right-hand side that cannot be expressed in any of the most expressive fragments of the left-hand side. In the following two sections we give the necessary lemmas needed to complete this proof.

4 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 show how to simulate certain features in fragments that, syntactically, do not include this feature.

First we show that we can count the number of items in a sequence in XQ_{at} and XQ_S, which is needed to show we can count in fragments at the right-hand side of the C-line in Fig. 4.

Lemma 4.1 *The* `count` *operator can be expressed in* XQ_{at}.

Proof. From Subsection 2.1 we know that `empty(`e_1`)` can be expressed in XQ. Counting the items of a sequence corresponds to finding the maximal position of an item in a sequence. Hence `count(`e_1`)` is equivalent to :

```
let $positions := (for $i at $pos in (e1) return $pos) return
for $a in (0, $positions) return
  if (empty(
    for $b in $positions return
      if ($b > $a) then 1 else ()
  )) then $a else ()
```

This expression always returns exactly one item, since `(0, $positions)` does not contain duplicate values, and hence there is exactly one item which is the largest.

□

Lemma 4.2 *The* `count` *operator can be expressed in* XQ_S.

Proof. The expression `sum(for $i in (`e_1`) return 1)` is equivalent to `count(`e_1`)`.

□

The following lemma shows that we can simulate the sequence generation from the `to` operator in all fragments that have recursive function definitions and hence shows that we can express it in all fragments on the right-hand side of the *to*-border in Fig. 4.

Lemma 4.3 *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)
};
```

□

We now show that we can compute the sum of a list of numbers using the `to` operator and the `count` function, which is used to show that in all fragments on the right-hand side of the S-border in Fig. 4 we can simulate `sum`.

Lemma 4.4 *The `sum` operator can be expressed in XQ^{to}_C.*

Proof. The following expression is equivalent to `sum(`e_1`)`:

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

□

The following Lemma gives another way to simulate `count`. In this simulation we use node construction and recursive function definitions. Together with Lemma 4.1 and Lemma 4.2 this shows that we can simulate `count` in all fragments at the right-hand side of the C-border in Fig. 4.

Lemma 4.5 *The `count` operator can be expressed in $XQ^{ctr,R}$.*

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

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

This expression generates as many new nodes as there are items in the in-

put e_1 and then applies a newly defined function `count-distinct-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-distinct-nodes` by exactly one node in each recursion step, which is possible since all items in the input sequence of `count-distinct-nodes` have a different node identity and hence we can remove each step the first node (in document order) of the newly created nodes. More precisely, the function `count-distinct-nodes` can be defined as follows:

```
declare function count-distinct-nodes($seq) {
  if (empty($seq)) then 0
  else (
    let $newseq := (
      for $e1 in $seq return
        if (empty(
          for $e2 in $seq return
            if ($e2 << $e1) then 1 else ()
        )) then () else $e1
    )
    return (1 + count-distinct-nodes($newseq))
  )
}
```

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.

□

Finally, we show how to simulate the `at` clause of a `for` expression by using node construction and `count`. This shows that we can simulate `at` in all fragments at the right-hand side of the *at*-border in Fig. 4.

Lemma 4.6 *The* `at` *clause in a* `for` *expression can be expressed in* XQ_C^{ctr}.

Proof. We transform sequence order into document order by creating new nodes as children of a common parent such that the new nodes contain all information of each item in the sequence and they are in the same order as the items in the original sequence. First, in XQ_C^{ctr} we can express the (non-recursive) functions `pos` and `atpos`, which respectively give the position of a node in a document-ordered duplicate-free sequence and return a node at a certain position in such sequence. This can be done as follows:

```
declare function pos($node, $seq) {
  count(for $e in $seq return
    if ($e << $node) then 1 else ()
  ) + 1
```

```
};

declare function atpos($seq, $pos) {
  for $node in $seq return
    if (pos($node, $seq) = $pos) then $node else ()
};
```

Let us assume that we can define XQ^{ctr}_C functions `encode` and `decode` such that `encode` translates an arbitrary sequence to an ordered and duplicate-free sequence of nodes while encoding each item in the original sequence into one node at the same position and the function `decode` can retrieve the original item given this node and the original sequence. Then the following XQ^{ctr}_C expression is equivalent to the $XQ^{ctr}_{C,at}$ expression "`for` $\$x$ `at` $\$pos$ `in` e_1 `return` e_2" (where e_1 and e_2 are XQ^{ctr}_C 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))
```

Because 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, since 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 can assume, w.l.o.g., that e_2 does not use variables `$seq` and `$newseq`, since they are used in the simulation.

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.

For example, consider the Store St of Example 2.1, which only has one tree.

Encoding the sequence $\langle 1, n_2^e, \text{“}c\text{”} \rangle$ over this store results into the creation of the following new element:

```
<newseq>
  <int>1</int>
  <node root="1" descpos="3"/>
  <str>c</str>
</newseq>
```

The encoding and decoding is performed by the following two functions:

```
declare function encode($seq) {
  let $rootseq := (
    for $e in $seq return
      typeswitch($e)
      case element() return root($e)
      case attribute() return root($e)
      case document-node() return root($e)
      default return ()
  )/. return
  let $newseq := element {"newseq"} {
    for $e in $seq
    return
      typeswitch($e)
      case xs:integer return element {"int"}  {$e}
      case xs:string  return element {"str"}  {$e}
      case xs:boolean return element {"bool"} {if ($e) then 1 else 0}
      default         return element {"node"} {
        attribute {"root"}    {pos(root($e), $rootseq)},
        attribute {"descpos"} {pos($e, root($e)//.)}
      }
  }
  return $newseq/*
};

declare function decode($node, $seq) {
let $rootseq := (
    for $e in $seq return
      typeswitch($e)
      case element() return root($e)
      case attribute() return root($e)
      case document-node() return root($e)
      default return ()
  )/. return
  if (name($node) = "int")            then xs:integer($node/text())
  else if (name($node) = "str")       then string($node/text())
  else if (name($node) = "bool")      then
```

```
    if (xs:integer($node/text()) = 1) then true() else false()
  else if (name($node) = "node")    then (
    let $root := atpos(rootseq($seq), xs:integer($node/@root))
    return atpos($root//., xs:integer($node/@descpos))
  )
  else ()
};
```

Note that none of the previous functions used recursion, so we do not actually need functions since we could inline the function definitions in the expressions. 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}.

□

5 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.

5.1 Set Equivalence and Bag Equivalence

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 environments and between sequences. In this definition **Set** (**Bag**) maps a sequence to the set (bag) of its items.

Definition 5.1 *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* $\mathbf{a} = \mathbf{a}'$, $\mathbf{b} = \mathbf{b}'$, $\mathbf{dom}(\mathbf{v}) = \mathbf{dom}(\mathbf{v}')$ *and* $\forall s \in \mathbf{dom}(\mathbf{v}) : \mathbf{Set}(\mathbf{v}(s)) = \mathbf{Set}(\mathbf{v}'(s))$, *and finally* $\mathbf{x} = \mathbf{x}'$.
The environments En and En′ are called bag-equivalent *iff they are set-equivalent and it holds that* $\forall s \in \mathbf{dom}(\mathbf{v}) : \mathbf{Bag}(\mathbf{v}(s)) = \mathbf{Bag}(\mathbf{v}'(s))$.

Example 5.1 shows an expression for which it holds that all result sequences in an evaluation against an environment in which `$seq` is bound to a sequence v have the same set representation. This result holds because it holds for all result sequences in an evaluation against another environment in which `$seq` is bound to a sequence v' for which it holds that $\mathbf{Set}(v) = \mathbf{Set}(v')$. This observation is generalized in Lemma 5.1.

Example 5.1 *Consider the following XQ expression:*

```
for $x in $seq return
  for $y in $seq return (if ($x > $y) then $x else ($x * $y))
```

If `$seq` *is* $\langle 1, 2\rangle$ *then the result is* $\langle 1, 2, 2, 4\rangle$*, and if* `$seq` *is* $\langle 2, 1, 2\rangle$ *then the result is* $\langle 4, 2, 4, 2, 1, 2, 4, 2, 4\rangle$*. Both result values have the same set representation* $\{1, 2, 4\}$.

Lemma 5.1 *Let St be a store, En, En′ two set-equivalent environments that have only function bodies which are XQ^R expressions, 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 $St, En' \vdash e \Rightarrow (St, r')$* [7] *, it also holds that* $\mathbf{Set}(r) = \mathbf{Set}(r')$.

Proof. This lemma is proven by induction on the derivation tree in which each node corresponds to a construct of rules (S3-S17,S23-S24) in Fig. 1. Obviously, variables, literals, and empty sequences return sequences with the same set representation when evaluated against set-equivalent environments. If we apply a comparison between two sequences v_1 and v_2 then the result is the same as when applied to v_1' and v_2' if v_1' and v_2' have the same set representation as respectively v_1 and v_2. This is due to the existential semantics of the value comparison operators and the fact that node comparison operators are only defined for single nodes.

We now consider the `for` expression. By induction we know that the set of items i in the result sequence of the `in` clause of a `for` expression is the same when evaluated against set-equivalent environments En and En'. Let E and E' denote the set of environments which contain all extensions of respectively En and En' with a binding of the loop variable to an item in the result sequence of the evaluation of the in-clause against respectively En and En'. It can be shown that the relation "is a set-equivalent environment" between E and E' is a total and surjective function, that is, for each evaluation against an environment En_1 in E there is an evaluation against a set-equivalent environment En_2 in E' and vice versa. This follows intuitively from the fact that En_1 is En but with the loop variable bound to a certain item i and En_2 is En' but with the loop variable also bound to i. By induction we then know that En_1 yields a result sequence with the same set representation as the result of En_2 and vice versa. Hence it follows that the concatenation of the result sequences of all evaluations yields result sequences with the same set representation.

[7] 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.

In a similar way, we can show that all other expressions in this XQuery fragment also return sequences with the same set representation when applied to set-equivalent environments, since the result sequences of their subexpressions have the same set representation.

□

The previous lemma is combined with the following lemma for proving that fragments on the left-hand side of the C-border in Fig. 4 cannot express `count`.

Lemma 5.2 *The fragment XQ_C does not have the property of Lemma 5.1.*

Proof. Consider an environment En, 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.

□

Note that the previous lemma implies that we cannot define full list or tree equality in XQ^R. Similar to the result of Lemma 5.1, there is also a fragment in which we cannot distinguish between lists with the same bag representation. The following lemma states this more precisely.

Lemma 5.3 *Let St be a store, En, En' two bag-equivalent environments that have only function bodies which are XQ^R_C expressions, 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_C expressions we can show similar to the proof of Lemma 5.1 that evaluations against bag-equivalent environments result in bag-equivalent result sequences. The only new feature is the `count()` function, which returns the same value when applied to sequences with the same bag representation. Moreover, we have to show for the `for` expression that there is a bijection between the sets E and E', as defined in the proof of Lemma 5.1.

□

The previous lemma is combined with the following lemma for proving that in fragments on the left-hand side of the *at*-border in Fig. 4 we cannot simulate the `at` clause in `for` expressions.

Lemma 5.4 *The fragment XQ_{at} does not have the property of Lemma 5.3.*

Proof. If we consider an environment En, 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

```
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.

□

5.2 Relationships between Input Size and Output Size

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 first introduce some 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., $\Delta_{St}^{tree} = \max(\bigcup_{n_1 \in V} \{c | c = |\{n_2 | (n_1, n_2) \in E^*\}|\})$ [8] . The function **size** maps an atomic value to the number of cells needed to represent this item on the tape of a Turing Machine. For example, if every character can be represented in one cell on the tape of a Turing Machine, then $\mathbf{size}(\texttt{"abc"}) = 3$ and $\mathbf{size}(78) = 2$.

Definition 5.2 (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* $d_O^i = \max(\{\mathbf{size}(a) | a \in (A^{St'} \cup A_v)\} \cup \{\lceil \log(\Delta_{St'}^{forest} + 1) \rceil\})$.

We now illustrate the previous definition with an example.

Example 5.2 *Consider the following query:*

```
for $x at $y in doc("doc.xml")//c return ($x, $y*100)
```

The evaluation of this query does not change the store and when evaluated against the store in Example 2.1 and the empty environment $En_0 = (\emptyset, \emptyset, \emptyset, \perp)$

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

we obtain the evaluation pair $((St, En_0), (St, \langle n_3^e, 100, n_5^e, 200\rangle))$.

- *The largest input sequence size is* $d_I^s = 9$*, which is the size of the largest (and only) tree in the input store.*
- *The largest input item size is* $d_I^i = 2$*, since we need* 2 *characters to represent the string* `"t1"` *and all other atomic values need at most* 2 *characters.*
- *The largest output sequence size is* $d_O^s = 9$*, which is the size of the largest (and only) tree in the output store.*
- *The largest output item size is* $d_O^i = 3$*, which is the size needed to represent the number* 100 *and all other atomic values in the output need at most* 3 *characters.*

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, the inclusion of the nodes of the output store in the output size is allowed for two 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 garbage collection is applied.

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. If such a function is a polynomial p that has $\mathbb{N}$ or $\mathbb{N}^2$ as its domain then there always exists an increasing polynomial p' such that p' is an upper bound for p. Therefore we assume that all such functions that are used as an upper bound in the following lemmas to be increasing functions.

Lemma 5.5 *For each evaluation* $St, En \vdash e \Rightarrow (St', v)$ *where* $e \in \mathbf{L}(XQ^{ctr,to})$ *it holds that* $d_O^i \leq p(d_I^i)$ *for some polynomial* p*.*

Proof. We prove the lemma by induction on the size of the derivation tree of the query q. In this tree the nodes correspond to the $\langle Expr\rangle$ non-terminal of the $XQ^{ctr,to}$ syntax and as a consequence each node corresponds to a construct of rules (S3-S17, S22, S25) in Fig. 1, so we prove the induction step for each of these rules. Literals (S4,S5) return constant values, while steps (S16), and variables (S3) return some items from the input (store and environment) of the expression and hence it is obvious that for all "leaf expressions" $d_O^i \leq p(d_I^i)$ hold for some polynomial p (linear function). All other expressions

have subexpressions. We denote the largest input/output item sizes of the k^{th} subexpression by $d^i_{I_k}$ and $d^i_{O_k}$. From the induction hypothesis it follows that for each subexpression it holds that $d^i_{O_k} \leq p_k(d^i_{I_k})$ for some polynomial p_k. Note that many expressions (S6, S7, S10-S15, S17, S18, S23, S26) do not alter the environment or the store before passing them to their subexpressions, so $d^i_{I_k} = d^i_I$ for all their subexpressions, and hence $d^i_{O_k} \leq p_k(d^i_I)$. All items in the result sequence of these expressions are either in the result of their subexpressions, constant values or items polynomially bounded in size by the items in the result of their subexpressions. Moreover, all items in the result store of these expressions are items in the result store and/or sequence of their subexpressions. Hence it holds that $d^i_O \leq p(d^i_I)$ for some polynomial p. The expressions in $XQ^{ctr,to}$ that do change the environment are:

`for` **expressions (S8)** By induction we know, the largest item in $\$x$ needs at most $d^i_{O_1} \leq p_1(d^i_I)$ space, for some polynomial p_1. From the induction hypothesis it follows that for each iteration of e_2 it holds that $d^i_{O_2} \leq p_2(d^i_{I_2})$ for some polynomial p_2, and hence $d^i_{O_2} \leq p_2(p_1(d^i_I))$. Since the result of a for expression contains only items that are in the result of an evaluation of e_2, we know that there exists a polynomial p such that $d^i_O \leq p(d^i_I)$

`let` **expressions (S9)** From the induction hypothesis it follows that the output item sizes for the first expression are bounded as $d^i_{O_1} \leq p_1(d^i_I)$ for some polynomial p_1. This upper bound also applies to $d^i_{O_2}$. Hence $d^i_O = d^i_{O_2} \leq p_2(p_1(d^i_I)) \leq p_3(d^i_I)$ for some polynomial p_3.

□

The previous lemma is combined with the following lemma to show that we cannot express `count` in fragments at the left-hand side of the C-border in Fig. 4.

Lemma 5.6 *The fragment XQ_C does not have the property of Lemma 5.5.*

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 $d^i_O = \lceil \log(k+1) \rceil$.

□

The following lemma gives upperbounds for the largest output sequence and item sizes for evaluations in $XQ^{ctr}_{at,S}$.

Lemma 5.7 *For each evaluation $St, En \vdash e \Rightarrow (St', v)$ where $e \in \mathbf{L}(XQ^{ctr}_{at,S})$ 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. This lemma can be proven by induction on the size of the deriva-

tion tree of the query q. In this derivation tree the nodes correspond to the $\langle Expr \rangle$ non-terminal of the $XQ^{ctr}_{at,S}$ syntax and as a consequence each node corresponds to a construct of rules (S3-S17, S20, S21, S25) in Fig. 1. First, consider the leafs of the derivation tree. Literals (S4,S5) return constant values, while steps (S16), and variables (S3) return some items from the input (store and environment) of the expression and hence it is obvious that for all leaf expressions $d^s_O \leq p_1(d^s_I)$ and $d^i_O \leq p_2(\log(d^s_I), d^i_I)$ hold for some polynomials (linear functions) p_1 and p_2. All other expressions have subexpressions. Similar to the proof of Lemma 5.5, we denote the input/output sizes of the k^{th} subexpression by $d^s_{I_k}$, $d^i_{I_k}$, $d^s_{O_k}$, and $d^i_{O_k}$. From the induction hypothesis it follows that $d^s_{O_k} \leq p_{k_1}(d^s_{I_k})$ and $d^i_{O_k} \leq p_{k_2}(\log(d^s_{I_k}), d^i_{I_k})$ for each subexpression. Note that many expressions (S6, S7, S10-S15, S17, S20, S25) do not alter the environment or the store before passing them to their subexpressions, so $d^s_{I_k} = d^s_I$ and $d^i_{I_k} = d^i_I$ for all subexpressions.

Basic built-in functions (S6), `if` and binary expressions (S7,S10-S15) and typeswitches (S17) All these expressions return results that are directly bound by the sum of output sizes of these subexpressions. Hence their output size is bound by $d^s_O \leq p_1(d^s_I)$ and $d^i_O \leq p_2(\log(d^s_I), d^i_I)$ for some polynomials p_1 and p_2.

`sum` aggregation (S20) This function returns a single number that is the sum of a number of values of the input sequence (output of the subexpression). This result is bounded by $d^s_{O_1} \cdot d^i_{O_1} \leq p_{k_1}(d^s_I) \cdot 2^{p_{k_2}(\log(d^s_I), d^i_I)}$ and hence $O(\log(p_{k_1}(d^s_I)) + p_{k_2}(\log(d^s_I), d^i_I))$ place is needed to represent this result (one item), which is bounded by $p(\log(d^s_I), d^i_I)$ for some polynomial p.

Constructors (S25) These can worst-case copy the entire input store, such that the output sequence size $d^s_O \leq O(2 \cdot d^s_I)$, and $d^i_O \leq O(\log(d^s_I), d^i_I)$, which is still within the bounds that we have to show.

`let` expression (S9) From the induction hypothesis it follows that the output sizes for the first subexpression are bounded as follows: $d^s_{O_1} \leq p_1(d^s_I)$ and $d^i_{O_1} \leq p_2(\log(d^s_I), d^i_I)$ for some increasing polynomials p_1 and p_2. These upper bounds also apply to $d^s_{I_2}$ and $d^i_{I_2}$. From the induction hypothesis it follows that $d^s_{O_2} \leq p_3(d^s_{I_2})$ and $d^i_{O_2} \leq p_4(\log(d^s_{I_2}), d^i_{I_2})$ for some polynomials p_3 and p_4. Hence $d^s_O = d^s_{O_2} \leq p_3(p_1(d^s_I)) \leq p_5(d^s_I)$ and $d^i_O = d^i_{O_2} \leq p_4(p_1(\log(d^s_I)), p_2(\log(d^s_I), d^i_I)) \leq p_6(\log(d^s_I), d^i_I)$ for some increasing polynomials p_5 and p_6.

`for` expressions (S8) The loop variable ($\$x$) and the position variable ($\$y$) are each iteration bound to one item. By induction we know that there are polynomials p_1 and p_2 such that the largest item of $\$y$ needs at most $\log(d^s_{O_1}) \leq \log(p_1(d^s_I))$ space and the largest item of $\$x$ needs at most $d^i_{O_1} \leq p_2(\log(d^s_I), d^i_I)$ space. Hence, for each iteration of e_2 it holds that $d^s_{I_2} \leq \log(p_1(d^s_I))$ and $d^i_{I_2} \leq p_2(\log(d^s_I), d^i_I)$. By induction we know for each iteration of e_2 that $d^s_{O_2} \leq p_3(d^s_{I_2})$ and $d^i_{O_2} \leq p_4(\log(d^s_{I_2}), d^i_{I_2})$ for some polynomials p_3 and p_4. Since the number of iterations is bounded by the result sequence of e_1, we know that at most $d^s_{O_1} \leq \log(p_1(d^s_I))$

iterations can occur. The result sequences of all iterations are concatenated in order to compute the end result and hence the output sizes are bounded as follows: $d^s_O \leq \log(p_1(d^s_I)) \cdot p_3(\log(p_1(d^s_I))) \leq p_5(d^s_I)$ and $d^i_O \leq p_4(\log(\log(p_1(d^s_I))), p_2(\log(d^s_I), d^i_I)) \leq p_6(\log(d^s_I), d^i_I)$ for some polynomials p_5 and p_6.

Path expressions (S16) These also obviously have output sizes within these polynomial bounds, since they are in fact a special kind of `for` expressions with an extra selection at the end, i.e., a node test and removal of duplicate nodes.

Since the number of subexpressions of an expression does not depend on the input store or environment, the previous results suffice to show that $d^s_O \leq p_1(d^s_I)$ and $d^i_O \leq p_2(\log(d^s_I), d^i_I)$ where p_1 and p_2 are some polynomials that only depend on the expression itself and the functions in the environment and not on the values in the store or the environment.

□

The previous lemma is combined with the following lemma to show that we cannot express `to` in fragments at the left-hand side of the *to*-border in Fig. 4.

Lemma 5.8 *The fragment XQ^{to} does not have the property of Lemma 5.7.*

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 $d^s_O = \Omega(k \log(k))$.

□

Finally, we also give upperbounds for the largest output sequence and item sizes for evaluations in $XQ^{ctr,to}_{at}$.

Lemma 5.9 *For each evaluation $St, En \vdash e \Rightarrow (St', v)$ where $e \in \mathbf{L}(XQ^{ctr,to}_{at})$ 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. Similar to the proof of Lemma 5.7 we prove this lemma by induction on the derivation tree. However, in this proof we will omit some details that were discussed earlier. In the proof of Lemma 5.7 we were allowed to use induction since a polynomial applied to a polynomial resulted again into a polynomial. We are also now allowed to use induction for the following reason. Suppose that $d^s_O \leq p_1(d^s_{I_1}, 2^{d^i_{I_1}})$, $d^i_O \leq p_2(\log(d^s_{I_1}), d^i_{I_1})$, $d^s_{I_1} \leq p_3(d^s_I, 2^{d^i_I})$ and $d^i_{I_1} \leq p_4(\log(d^s_I), d^i_I)$. Then it follows that

- $d^s_O \leq p_1(p_3(d^s_I, 2^{d^i_I}), 2^{p_4(\log(d^s_I), d^i_I)}) \leq p_1(p_3(d^s_I, 2^{d^i_I}), p_5(2^{\log(d^s_I)}, 2^{d^i_I}))$ for some polynomial p_5 and hence $d^s_O \leq p_6(d^s_I, 2^{d^i_I})$ for some polynomial p_6
- $d^i_O \leq p_2(\log(p_3(d^s_I, 2^{d^i_I})), p_4(\log(d^s_I), d^i_I)) \leq p_2(p_7(\log(d^s_I), \log(2^{d^i_I}))$,

$p_4(\log(d_I^s), d_I^i))$ for some polynomial p_7 and hence $d_O^i \leq p_8(\log(d_I^s), d_I^i)$ for some polynomial p_8.

Hence we can use induction in order to prove this lemma. We know that for all XQ_{at}^{ctr} expressions there was a polynomial relation between the largest input sequence/item sizes and the largest output sequence/item sizes. Furthermore, 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 it 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.

□

The previous lemma is combined with the following lemma to show that we cannot simulate all recursive functions in fragments at the left-hand side of the R-border in Fig. 4.

Lemma 5.10 *The fragment XQ^R does not have the property of Lemma 5.9.*

Proof. Clearly there are expressions in XQ^R that do not have this property. Indeed, if we consider the empty store St_0, the environment $En = (\{\}, \{\}, \{(\text{“\$}input\text{”}, k)\}, \bot)$, and the expression $e =$

```
declare function mpowern($m, $n) {
   if ($n = 1) then $m else ($m * mpowern($m, $n - 1))
};
declare function genseq($n) {
  if ($n < 1) then () else (genseq($n - 1), 1)
};
let $n := $input
return genseq(mpowern($n, $n))
```

then the evaluation $St_0, En \vdash e \Rightarrow (St', v)$ has 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 $\Omega(k^k)$.

□

5.3 Upper Bounds for the Number of Different Possible Results

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 possible atomic values in the input store and environment.

Definition 5.3 (Possible Results) *Consider an expression e, a (finite) alphabet $\Sigma \subset \mathcal{A}$ and a number k. The set Res of possible results for evaluations*

of e constrained by Σ and k is defined as the set of all pairs (St', v) for which it holds that there exists a store St and environment En (in the same fragment as e) such that $St, En \vdash e \Rightarrow (St', v)$ and $d_I^s \leq k$ and $A^{St} \cup A^{En} \subseteq \Sigma$.

In other words, given an expression e, an alphabet Σ and a number k, the set *Res* contains all possible outputs of the evaluations of e restricted to Σ and k. We will show that, for expressions the fragment XQ_{at}^{ctr}, the number of different atomic values in this set is polynomially bounded by k and the size of Σ. We first illustrate this claim with an example.

Suppose e is following expression:

```
for $x at $y in $z return ($y, $x + 1)
```

If we assume some Σ and k then we can verify that the number of different atomic values in the output of this expression is bounded by $2 \cdot |\Sigma| + k$. For example, if $\Sigma = \{5, 8\}$ and $k = 2$ then only values in $\{1, 2, 5, 6, 8, 9\}$ can occur in a result, but note that not all these values have to occur in the result of every evaluation constrained by this Σ and k. For example, if the input store is empty and $\mathbf{v}(z) = \langle 5 \rangle$ is the only variable binding in the environment then only the atomic values 1 and 6 occur in the result.

Lemma 5.11 *Consider a (finite) alphabet $\Sigma \subset \mathcal{A}$ and a number k. 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 k, then the number of different atomic values in all possible results is defined as $\left|\bigcup_{(St',v)\in Res}(A^{St'} \cup A_v)\right|$ and bounded by $p(n, k)$ for some polynomial p.*

Proof. This lemma can be proven by induction on the derivation tree where each expression corresponds to the $\langle Expr \rangle$ non-terminal of the XQ_{at} syntax and as a consequence each node corresponds to a construct of rules (S3-S18,S22) of Fig. 1.

First, consider the leafs of the derivation tree. Literals (S4,S5) return for all evaluations the same atomic value, steps (S16) do not return atomic values and variables (S3) only return atomic values originated from the input. All these expressions do not change the input store. Hence it holds that the number of atomic values in the possible results is bounded by $n + 1$.

All other expressions have subexpressions. Note that many expressions (S6, S7, S10-S15, S18, S26) do not alter the environment or the store before passing them to their subexpressions. All these expressions return either only atomic values from their subexpressions or one new atomic value that is a boolean. From the induction hypothesis and the fact that all these expressions have a constant number of subexpressions, which are all evaluated only once during one evaluation of the superexpression, it follows that the number of atomic

values in the possible results is bounded by $p(n,k)$ for some polynomial p.

We now discuss the remaining expressions.

`let` expressions (S9) The second subexpression is evaluated against an alphabet of size $N' < p_1(n,k)$ and a store and environment with a maximal sequence size of $k' < p_2(k)$ (Lemma 5.7) for some polynomials p_1, p_2. From the induction hypothesis then it follows that the number of atomic values in the possible results is bounded by $p'(n',k') < p'(p_1(n,k), p_2(k)) < p(N,k)$ for some polynomials p and p'.

`for` expressions (S8) From Lemma 5.7 we know that the number of different atomic values in the possible results is bounded by $p_1(n,k)$ for some polynomial p_1 and the number of items in the result sequence of the subexpression is bounded by $p_2(k)$ for some polynomial p_2. The expression in the return clause is evaluated at most $p_2(k)$ times against the result store of the first subexpression and environment where two extra variables are set. This in fact means that the subexpression is evaluated against an alphabet of size $n' < p_1(n,k)$ and a store and environment with a maximal sequence size of $k' < p_2(k)$. Hence, from the induction hypothesis it follows that the number of atomic values in the possible results for each evaluation is bounded by $p'(n',k') < p'(p_1(n,k), p_2(k)) < p''(n,k)$ for some polynomial p''. Since the result of the `for` expression is just the concatenation of all results of the return clause, the total number of atomic values in the possible results is bounded by $p_2(k).p''(n,k) < p(n,k)$ for some polynomial p.

Path expressions (S17) Path expressions can be considered as a special kind of `for` expressions with an extra selection at the end, i.e., sorting nodes in document order and removing duplicate nodes. Hence, obviously the lemma also holds for them.

□

The previous and the following lemma are combined in Section 6 to show that we cannot compute the sum of a list of numbers in fragments on the left-hand side of the S-border in Fig. 4.

Lemma 5.12 *The fragment $XQ_{at,S}$ does not have the property of Lemma 5.11.*

Proof. Consider the alphabet $\Sigma = \{1, 2, 4, \ldots, 2^{n-1}\}$ and $k = n$. 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}.

□

6 Proof of the Main Theorem

The results of Section 4 and Section 5 can be combined to complete the proof of Theorem 3.1.

First, we prove that the dotted borders in Fig. 4 are correct by showing that something can be expressed in the least expressive fragments of the right-hand side that cannot be expressed in any of the most expressive fragments of the left-hand side.

to-**border** The most expressive fragment on the left-hand side is $XQ^{ctr}_{at,S}$. The least expressive fragment on the right-hand side is XQ^{to}. From Lemma 5.7 and Lemma 5.8 it follows that `to` cannot be expressed in XQ^{ctr}_{S}.

R-**border** The most expressive fragment on the left-hand side is $XQ^{ctr,to}_{at}$. The least expressive fragment on the right-hand side is XQ^{R}. From Lemma 5.9 and Lemma 5.10 it follows that recursive function definitions cannot be simulated in $XQ^{ctr,to}_{at}$.

C-**border** The most expressive fragments on the left-hand side are XQ^{R} and $XQ^{ctr,to}$. The least expressive fragment on the right-hand side is XQ_{C}. From Lemma 5.1 and Lemma 5.2 it follows that `count()` cannot be expressed in XQ^{R} and from Lemma 5.5 and Lemma 5.6 it follows that `count()` cannot be expressed in $XQ^{ctr,to}$.

at-**border** The most expressive fragments on the left-hand side are XQ^{R}_{C} and $XQ^{ctr,to}$. The least expressive fragment on the right-hand side is XQ_{at}. From Lemma 5.4 it follows that `at` cannot be expressed in XQ^{R}_{C}. From Lemma 5.5 and Lemma 5.6 it follows that `count()` cannot be expressed in $XQ^{ctr,to}$ and hence also `at` cannot be expressed in $XQ^{ctr,to}$, since otherwise we would get a contradiction by simulating `count()` as known from Lemma 5.1 and Lemma 4.1.

S-**border** The most expressive fragments on the left-hand side are XQ^{R}, $XQ^{ctr,to}$ and XQ^{ctr}_{C}. The least expressive fragment on the right-hand side is XQ_{S}. From Lemma 5.5, Lemma 5.6, Lemma 5.1 and Lemma 5.2 it follows that `count()` cannot be expressed in $XQ^{ctr,to}$ and in XQ_{R}. Hence `sum()` cannot be simulated in XQ^{R} nor $XQ^{ctr,to}$. Finally, from Lemma 5.11 and Lemma 5.12 follows that `sum()` cannot be expressed in XQ^{ctr}_{C}.

All previous results can now be combined to complete the proof:

- 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 4.
- 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 separated 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 Fig. 4 that there are (at least) two borders separating 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 $XF_1 \not\succ XF_2$.

7 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 article outline that, using six attributes (the `count()` function, the `sum()` function, `to` expressions, the `at` clause, node construction 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 possess a different degree of expressive power.

This has led to several interesting observations:

- The ability to construct nodes sometimes adds expressive power, even if no new nodes are returned in the result. For example, it is shown that the quite powerful fragment $XQ^{R,to}_{C,S}$, i.e., the basic fragment extended with recursive functions, `to` expressions, the `count()` function and the `sum()` function, still cannot distinguish the sequences $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$ (see Lemma 5.3 and the proof that this fragment is equivalent to XQ^{R}_{C}, i.e., the basic fragment extended with recursive functions and the `count()` function) but it is easy to see that already in XQ^{ctr}, the basic fragment extended with only node construction, these sequences can be distinguished.
- The `order by` clause can already be simulated in XQ_{at}, the basic fragment extended with `at` clauses in `for` expressions, as is shown in Section 2.3.

- Filter expressions, i.e. expressions of the form e_1`[`e_2`]`, can be already simulated in the basic fragment XQ, as shown in Section 2.3.
- The functions `position()` and `last()` that allow the selection of nodes in certain positions can already be simulated in XQ_{at}, the basic fragment extended with only `at` clauses, which is also discussed in Section 2.3.
- On the other hand `at` clauses in `for` expressions can be simulated in XQ_C^{ctr}, the basic fragment extended with node construction and counting, which is shown in Lemma 4.6, and demonstrates the combined expressive power of these two features.
- The smallest fragment that has the expressive power of the full language is $XQ^{ctr,R}$, the basic fragment extended with node construction and recursion (cf. Figure 4). This means that all the other fragment-defining attributes, which are `to` expressions, `at` clauses, the `count()` and `sum()` functions, can be simulated in this fragment.

Finally we briefly discuss some related work by Koch [8] and Benedikt and Koch [2]. In this work also XQuery fragments are defined and studied in terms of computational complexity and compared in expressive power with certain types of first-order logic. Unfortunately the fragments defined in that work have no direct relationship with our fragments. However, we can make some observations on the relationships between their fragments $AtomXQ$ and XQ which seem similar to our fragments XQ^{ctr} and XQ_{at}^{ctr}, respectively.

Their fragment $AtomXQ$ is an XQuery fragment in which one can express path expressions, create new trees, compare nodes/values, test sequences for emptiness and use simple `for` expressions and `if` expressions. In terms of expressive power $AtomXQ$ is a subset of our fragment XQ^{ctr}. The converse clearly does not hold since XQ^{ctr} can do basic arithmetic on values in the XML trees. Even if the arithmetic operations from XQ^{ctr} are removed the relationship is not clear because XQ^{ctr} allows general `let` expressions for which it is not clear if the can be removed without losing expressive power.

Their fragment XQ is basically $AtomXQ$ extended with a deep-equality comparison for trees. In terms of expressive power their fragment XQ (which is different from our XQ) is a subset of our fragment XQ_{at}^{ctr}. The most involved part of the proof is showing that deep equivalence of nodes can be expressed, for which the `at` clause seems to be required. Conversely, it is easily observed that XQ_{at}^{ctr} can express some functions that cannot be expressed by their XQ because with the `at` clause we can write functions that return integers not in the XML tree. This raises the question whether in terms of expressive power their XQ is a subset of our XQ^{ctr}, which seems unlikely, but no proof of a counterexample has been found yet.

Acknowledgements: The authors wish to thank the anonymous referees, whose constructive remarks have helped greatly to improve this paper.

References

[1] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2003.

[2] Michael Benedikt and Christoph Koch. Interpreting tree-to-tree queries. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 552–564. Springer, 2006.

[3] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, 2007. Available at `http://www.w3.org/TR/xquery/`.

[4] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Recommendation, 2007. Available at `http://www.w3.org/TR/xquery-semantics/`.

[5] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 179–190. ACM, 2003.

[6] Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A light but formal introduction to XQuery. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *Database and XML Technologies, Second International XML Database Symposium, XSym 2004, Toronto, Canada, August 29-30, 2004, Proceedings*, volume 3186 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2004.

[7] Stephan Kepser. A simple proof of the Turing-completeness of XSLT and XQuery. In *Proceedings of the Extreme Markup Languages 2004 Conference, 2-6 August 2004, Montréal, Quebec, Canada*. IDEAlliance, 2004. Available at `http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html`.

[8] Christoph Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, 2006.

[9] Wim Le Page, Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen. On the expressive power of node construction in XQuery. In AnHai Doan, Frank Neven, Robert McCann, and Geert Jan Bex, editors, *Proceedings of the Eight International Workshop on the Web & Databases (WebDB 2005), Baltimore, Maryland, USA, Collocated with ACM SIGMOD/PODS 2005, June 16-17, 2005*, pages 85–90, 2005.

[10] Chen Li, editor. *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*. ACM, 2005.

[11] Leonid Libkin. Expressive power of SQL. *Theoretical Computer Science*, 3(296):379–404, 2003.

[12] Maarten Marx. Conditional XPath, the first order complete XPath dialect. In Li [10], pages 13–22.

[13] Jan Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.

[14] Stijn Vansummeren. Deciding well-definedness of XQuery fragments. In Li [10], pages 37–48.