Expressive Power of Recursion and Aggregates in XQuery

Technical Report UA 2005-05

Jan Hidders   Stefania Marrara   Jan Paredaens
Roel Vercammen

Abstract

The expressive power of languages has been widely studied in Computer Science literature. In this technical report we investigate the expressive power of XQuery, trying to focus on fragments of the language itself in order to outline which features are really necessary, and which ones simplify queries already expressible and could hence be omitted. The core of the report is the study of the effect of recursion, aggregates, sequence generators, node constructors, and position information on the expressiveness of XQuery, starting from a simple subset called XQ and then adding, step by step, new features, in order to discover what can be defined as a minimal set of syntax rules for each degree of desired expressive power.

Contents

Chapter 1

Introduction

The literature on programming languages contains many studies about expressive power of languages. These studies would like to formally prove that a certain language is more or less expressive than another language. Determining such a relation between languages objectively rather than subjectively seems to be somewhat problematic, a phenomenon that Paul Graham has discussed in "The Blub Paradox" [6]. Arguments in this context typically assert the expressibility or non-expressibility of programming constructs relative to a language. Unfortunately, programming language theory does not provide a formal framework for specifying and verifying this statement as described in [5], as the languages are usually universal. However, this approach can be useful if used to study the expressive power of one single language, in order to separate an essential core of programming structures from the syntactic sugar. In this work we study the expressive power of the XML query language proposed by the W3C, XQuery [2], which is a powerful and convenient language designed for querying XML data. The drawback with this language is that XQuery is rather complex and with a well defined but troublesome semantics. A question that rises is: what queries can be expressed by a certain fragment of XQuery and is the entire syntax of the language, with its complexity, really necessary? After all, it is the inability to express certain properties that motivates language designers to add new features. The point is: which are the redundancies of XQuery as it is going to be the standard query language for XML data? The aim of our work is to investigate the expressive power of this language, trying to focus on fragments of the language itself in order to outline which features really add expressive power and which ones simplify queries already expressible and could be omitted, for example, in a prototype engine or to create a simple application for users who do not need the whole complexity of XQuery. This work can be also useful for theoretical studies, such as studying typing problems or the equivalence of expressions (maybe for a new query optimization approach). As an example of what could be syntactic sugar, consider the XQuery core definition: one possible definition says that it should be an essential group of rules, but it contains constructs that could be easily omitted; for example the "`case`" clause can be simulated by means of a set of "`if`" clauses, while two axes (child and descendant) have enough expressive power to simulate all the others. On the other hand, there are queries that cannot be expressed without certain constructs. As an example, given a sequence of integers "`seq`" consider the following query that uses the “`at`” clause:

```
for $i at $pos in $seq
  return ($i + $pos)
```

It turns out that this query cannot be expressed without “`at`” or node construction. There are many queries that can be simulated using only a small part of the syntax of XQuery, and our aim is to point out which are the main fragments of the language and the containment that hold among these fragments, in order to discover what can be defined as a minimal set of syntax rules for each degree of desired expressive power.

The core of this report is the study of the effect of recursion, aggregates, sequence generators, node constructors, and position information on the expressiveness of XQuery starting from a simple subset called XQ and then adding, step by step, new features. Section 2 defines the syntax and the semantics of the different XQuery fragments that we are going to analyze. Chapter 3 presents the classes to which fragments with the same expressive power belong and their characteristics. Chapter 4 presents a small review of other work about the problem of query languages expressive power. Finally, Chapter 5 outlines the conclusions of our work.

Chapter 2

XQuery Fragments

This section formally introduces the XQuery fragments for which we study the expressive power in this report. We will use LiXQuery [7] as a formal foundation. LiXQuery is a light-weight sublanguage of XQuery which is fully downwards compatible with XQuery and includes the typical expressions of XQuery. The sublanguage XQ of LiXQuery will be defined together with a set of attributes (such as count, sum, recursion, at, to, and node constructors), which we use to extend XQ. We can combine XQ and the attributes to construct a number of XQuery fragments. The syntax of each of these fragments is defined in Section 2.1. In Section 2.2 we briefly describe the semantics of a query.

2.1 Syntax

The syntax of the fragment XQ is shown in Figure 2.1, by rules [1-19] [1]. This syntax is an abstract syntax in the sense that it assumes that extra brackets and precedence rules are added for disambiguation. 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. It does not include user defined functions, the "`count`" and "`sum`" functions, the "`to`" sequence generator, and node constructors. 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 2.2 for the syntax of the attributed fragments). The fragment XQ^R denotes XQ augmented with (recursive) functions definitions. If a fragment is attributed by "C", it also contains the "`count`" function; the attribute "S" denotes the inclusion of the "`sum`" function; the attribute "*at*" denotes the "`at`" clause in a for expression; "*ctr*" indicates the inclusion of the node constructors, and finally the "*to*" attribute 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 work is to investigate and to compare the expressive power of these fragments. With XQ^* we denote the fragment $XQ^{R,to,ctr}_{C,S,at}$ expressed by rules [1-26]. Following

[1]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.

auxiliary definitions will be used throughout the technical report:

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 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 2.2.*

2.1.1 Comparison with LiXQuery

Since XQ^* is a sublanguage of LiXQuery, we ignore typing and do not consider namespaces[2], comments, programming instructions, and entities. There are some features left out from LiXQuery in the definition of XQ^*. The first feature that is left out is the union. This can be easily simulated in the following way:

e_1 "`|`" e_2 $\rightarrow$ `(` e_1 `,` e_2 `)/.`

The "`/.`" expression removes the duplicates and sort the result sequence by document order. A second feature that is in LiXQuery, but not in XQ^* is the filter expression and the functions "`position()`" and "`last()`". We can simulate these as follows:

e_1 "`[`" e_2 "`]`" $\rightarrow$

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

Where e_2' is the same test as e_2, except that "`position()`" and "`last()`" are replaced by "`$pos`" and "`$last`". The last feature that is not in XQ^* is the parent step ("`..`"), which we can simulate as follows:

e_1 "`/..`" $\rightarrow$

```
(
  for $dot1 in e1
  return
    for $dot2 in root($dot1)//.
    return
      if ($dot1 = ($dot2/*, $dot2/text(), $dot2/@*))
      then ($dot2)
      else ()
)/.
```

Again, "`/.`" performs a distinct-doc-order operation. The variable "`$dot2`" runs over all nodes of the input document trees and is in the result sequence if one of its children (element, attribute, or text nodes) equals a node from the input sequence. 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.

2.1.2 Simulation of (other) XQuery features

We can simulate many XQuery features that are not in XQ^* by using (a sublanguage of) XQ^*. For example the emptiness test and the quantifiers `some` and `every` can be simulated by following XQ expressions:

[2] 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

[1] $\langle Query\rangle \rightarrow (\langle FunDecl\rangle\,\text{“\texttt{;}”})^*\langle Expr\rangle$

[2] $\langle Expr\rangle \rightarrow \langle Var\rangle \mid \langle BuiltIn\rangle \mid \langle IfExpr\rangle \mid \langle ForExpr\rangle \mid \langle LetExpr\rangle \mid \langle Concat\rangle \mid$
$\langle AndOr\rangle \mid \langle ValCmp\rangle \mid \langle NodeCmp\rangle \mid \langle AddExpr\rangle \mid \langle MultExpr\rangle \mid$
$\langle Step\rangle \mid \langle Path\rangle \mid \langle Literal\rangle \mid \langle EmpSeq\rangle \mid \langle Constr\rangle \mid \langle TypeSw\rangle \mid$
$\langle FunCall\rangle \mid \langle Count\rangle \mid \langle Sum\rangle$

[3] $\langle Var\rangle \rightarrow \text{“\texttt{\$}”}\langle Name\rangle$

[4] $\langle Literal\rangle \rightarrow \langle String\rangle \mid \langle Integer\rangle$

[5] $\langle EmpSeq\rangle \rightarrow \text{“\texttt{()}”}$

[6] $\langle BuiltIn\rangle \rightarrow \text{“\texttt{doc(}”}\langle Expr\rangle\text{“\texttt{)}”} \mid \text{“\texttt{name(}”}\langle Expr\rangle\text{“\texttt{)}”} \mid \text{“\texttt{string(}”}\langle Expr\rangle\text{“\texttt{)}”} \mid$
$\text{“\texttt{xs:integer(}”}\langle Expr\rangle\text{“\texttt{)}”} \mid \text{“\texttt{root(}”}\langle Expr\rangle\text{“\texttt{)}”} \mid$
$\text{“\texttt{concat(}”}\langle Expr\rangle, \langle Expr\rangle\text{“\texttt{)}”} \mid \text{“\texttt{true()}”} \mid \text{“\texttt{false()}”} \mid$
$\text{“\texttt{not(}”}\langle Expr\rangle\text{“\texttt{)}”} \mid \text{“\texttt{distinct-values(}”}\ \langle Expr\rangle\ \text{“\texttt{)}”}$

[7] $\langle IfExpr\rangle \rightarrow \text{“\texttt{if} ”“\texttt{(}”}\langle Expr\rangle\text{“\texttt{)}” “\texttt{then}”}\langle Expr\rangle\ \text{“\texttt{else}”}\langle Expr\rangle$

[8] $\langle ForExpr\rangle \rightarrow \text{“\texttt{for}”}\langle Var\rangle(\langle AtExpr\rangle)?\ \text{“\texttt{in}”}\langle Expr\rangle\ \text{“\texttt{return}”}\langle Expr\rangle$

[9] $\langle LetExpr\rangle \rightarrow \text{“\texttt{let}”}\langle Var\rangle\text{“\texttt{:=}”}\langle Expr\rangle\ \text{“\texttt{return}”}\langle Expr\rangle$

[10] $\langle Concat\rangle \rightarrow \langle Expr\rangle\text{“\texttt{,}”}\langle Expr\rangle$

[11] $\langle AndOr\rangle \rightarrow \langle Expr\rangle(\text{“\texttt{and}”} \mid \text{“\texttt{or}”})\langle Expr\rangle$

[12] $\langle ValCmp\rangle \rightarrow \langle Expr\rangle(\text{“\texttt{=}”} \mid \text{“\texttt{<}”})\langle Expr\rangle$

[13] $\langle NodeCmp\rangle \rightarrow \langle Expr\rangle(\text{“\texttt{is}”} \mid \text{“\texttt{<<}”})\ \langle Expr\rangle$

[14] $\langle AddExpr\rangle \rightarrow \langle Expr\rangle\ (\text{“\texttt{+}”} \mid \text{“\texttt{-}”})\ \langle Expr\rangle$

[15] $\langle MultExpr\rangle \rightarrow \langle Expr\rangle\ (\text{“\texttt{*}”} \mid \text{“\texttt{idiv}”})\ \langle Expr\rangle$

[16] $\langle Step\rangle \rightarrow \text{“\texttt{.}”} \mid \langle Name\rangle \mid \text{“\texttt{@}”}\langle Name\rangle \mid \text{“\texttt{*}”} \mid \text{“\texttt{@*}”} \mid \text{“\texttt{text()}”}$

[17] $\langle Path\rangle \rightarrow \langle Expr\rangle(\text{“\texttt{/}”} \mid \text{“\texttt{//}”})\langle Expr\rangle$

[18] $\langle TypeSw\rangle \rightarrow \text{“\texttt{typeswitch} ”“\texttt{(}”}\langle Expr\rangle\text{“\texttt{)}”}\ (\text{“\texttt{case}”}\ \langle Type\rangle\ \text{“\texttt{return}”}\langle Expr\rangle)^+$
$\text{“\texttt{default}” “\texttt{return}”}\langle Expr\rangle$

[19] $\langle Type\rangle \rightarrow \text{“\texttt{xs:boolean}”} \mid \text{“\texttt{xs:integer}”} \mid \text{“\texttt{xs:string}”} \mid \text{“\texttt{element()}”} \mid$
$\text{“\texttt{attribute()}”} \mid \text{“\texttt{text()}”} \mid \text{“\texttt{document-node()}”}$

---

[20] $\langle Count\rangle \rightarrow \text{“\texttt{count(}”}\ \langle Expr\rangle\ \text{“\texttt{)}”}$

[21] $\langle Sum\rangle \rightarrow \text{“\texttt{sum(}”}\ \langle Expr\rangle\ \text{“\texttt{)}”}$

[22] $\langle AtExpr\rangle \rightarrow \text{“\texttt{at}”}\ \langle Var\rangle$

[23] $\langle SeqGen\rangle \rightarrow \langle Expr\rangle\ \text{“\texttt{to}”}\ \langle Expr\rangle$

[24] $\langle FunCall\rangle \rightarrow \langle Name\rangle\text{“\texttt{(}”}((\langle Expr\rangle(\text{“\texttt{,}”}\langle Expr\rangle)^*)?\text{“\texttt{)}”}$

[25] $\langle FunDecl\rangle \rightarrow \text{“\texttt{declare}” “\texttt{function}”}\ \langle Name\rangle\ \text{“\texttt{(}”}\ ((\langle Var\rangle\ (\ \text{“\texttt{,}”}\ \langle Var\rangle)^*)?\ \text{“\texttt{)}”}$
$\text{“\{”}\ \langle Expr\rangle\ \text{“\}”}$

[26] $\langle Constr\rangle \rightarrow \text{“\texttt{element}”“\{”}\langle Expr\rangle\text{“\}” “\{”}\langle Expr\rangle\text{“\}”} \mid$
$\text{“\texttt{attribute}”“\{”}\langle Expr\rangle\text{“\}” “\{”}\langle Expr\rangle\text{“\}”} \mid$
$\text{“\texttt{text}”“\{”}\langle Expr\rangle\text{“\}”} \mid \text{“\texttt{document}”“\{”}\langle Expr\rangle\text{“\}”}$

Figure 2.1: Syntax for XQ^* queries and expressions

XQ	[1-19]
$_{C}$	+ [20]
$_{S}$	+ [21]
$_{at}$	+ [22]
^{to}	+ [23]
^{R}	+ [24-25]
^{ctr}	+ [26]

Figure 2.2: Definition of XQ fragments

```
"empty(" e1 ")" →                      if (1 = (for $y in e1  return 1))
                                       then true()
                                       else false()
"every" $x "in" e1 "satisfies" e2 →    not(empty(
                                           for $x in  e1  return
                                             if (e2) then $x  else ()
                                       ))
"some" $x "in" e1 "satisfies" e2 →      empty(
                                           for $x in  e1  return
                                           if (e2) then () else $x
                                         )
```

Another example is the aggregate function "max", which can even be simulated in XQ as we will now illustrate. The maximum of the result sequence of e_1 can be computed as follows:

```
"max("e1")" →   let $v1 := e1 return
                distinct-values(
                  for $v2 in $v1
                  return
                    if ($v2 < $v3)
                    then ()
                    else $v2
                )
```

A third XQuery feature that can be simulated in XQ are all XPath axes. We illustrate this claim by giving the simulation of the "following-sibling" axis:

```
e1"/following-sibling::node()" →   (
                                     for $v1 in e1
                                     return
                                       for $v2 in ($v1/../*, $v1/../text())
                                       return
                                         if ($v1 << $v2)
                                         then $v2
                                         else ()
                                   )/.
```

The last XQuery feature that we use to illustrate the claim that most typical XQuery expressions can be expressed in XQ^* is the "order by" clause. The simulation of this feature can be done by implementing the insertion sort algorithm, which can obviously be done in XQ_{at}^{R}

2.2 Semantics

The semantics of the XQuery fragments that we have just defined is downwards compatible with the XQuery Formal Semantics[4] defined by the W3C. However, we need a more formal and precise notion of the result of a query for examining the expressive power. Therefore we first introduce the notion of an XML store and an environment, as described in [8]. Then we illustrate briefly how we can define the result of XQ^* expressions by using reasoning rules. Finally, we define the semantics of a query by means of the semantics of its subexpressions. Most definitions in this subsection originate from LiXQuery[7].

Expressions are evaluated against an *XML store* which contains XML fragments created as intermediate results, and all the web documents. The latter assumption is a simplification, since in practice these documents are materialized in the store when they are accessed for the first time. In the following definitions we will use some 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 2.2. *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 document nodes* V^d*, element nodes* V^e*, attribute nodes* V^a*, and text nodes* V^t*;*
- (V, E) *is an acyclic directed graph (with nodes* V *and directed edges* E*), and hence it is composed of trees; if* $(m, n) \in E$ *then we say that* n *is a child of* m*;*
- $<$ *is a strict partial order on* V *that compares exactly the different children of a common node;*
- $\nu : V^e \cup V^a \to \mathcal{N}$ *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 and text children;*
- *there are no adjacent text children;*
- *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 report, we will use the function ξ, which maps a sequence of items to its serialization, as defined in [9].

Example 2.2.1. *Let* $St = (V, E, <, \nu, \sigma, \delta)$ *be an XML store that is shown in Figure 2.3.*

- *The set of nodes* V *consists of* $V^e = \{n_1^e, n_2^e, n_3^e, n_5^e, n_7^e\}$, $V^t = \{n_4^t, n_6^t, n_8^t\}$, $V^d = V^a = \emptyset$.

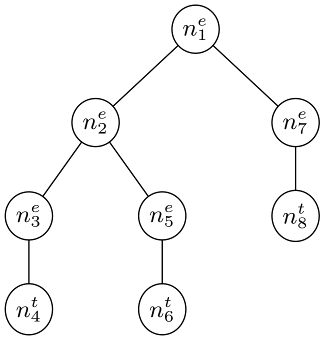

Figure 2.3: XML tree of Example 2.2.1

- *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*"*.* [3]

In this example $\xi(n_1^e) =$ "<a><c>t1</c><c>t2</c>t3" *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 2.3. *An* Environment *of an XML store St is a 4-tuple* $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ *with*

- *a partial function* $\mathbf{a} : \mathcal{N} \to \mathcal{N}^*$ *that maps a function name to its formal arguments; it is used in rule [1,24,25];*
- *a partial function* $\mathbf{b} : \mathcal{N} \to \mathbf{L}(XQ^*)$ *that maps a function name to the body of the function; it is also used in rules [1,24,25];*
- *a partial function* $\mathbf{v} : \mathcal{N} \to (\mathcal{V} \cup \mathcal{A})^*$ *that 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* $(\forall f \in \mathrm{rng}(\mathbf{b})).(f \in \mathbf{L}(XF))$*.*

Note that the definition of an environment is slightly different from the definition in [7]. The original definition also included the position of the context item in the context sequence ($\mathbf{k}$) and the size of the context sequence ($\mathbf{m}$).

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

They have been omitted because no rule of XQ^* uses them in creating the result sequence. The only LiXQuery constructs that use **k** and **m** values in their semantics are "`last()`" and "`position()`", but they are not included in XQ^* syntax. 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 **v** maps n to y. This gives us the necessary formal foundation to write down what the result of the evaluation of an expression is. 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. We will use reasoning rules to define the semantics of XQ^* expressions. Since the definition of the semantics is not the main purpose of this work, we give only one example of such a rule. A more detailed discussion on the formal semantics containing all semantic rules, can be found in [7]. As an example of a semantic rule, consider the rule for the slash operator that occurs in path expressions. The semantics of (e' `/` e'') is defined by following rule.

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

First e' is evaluated. Then for each item in its result we bind in the environment **x** to this item, and with this environment we evaluate e''. The results of all these evaluations are concatenated and finally this sequence is sorted by document order (**Ord**) and the duplicates are removed (**Set**). The result is only defined if all the evaluations of e'' contain only nodes.

So far, we have only discussed the semantics of XQ^* expressions, which are evaluated against a store and an environment. But in this paper we want to study the expressive power of queries. For this we will need to specify the semantics of a query in a certain fragment.

Definition 2.4. *Let $XF \in \Phi$ be an XQuery fragment. An XF query q is an XF expression that is evaluated against an initial store (containing the web) and an initial environment of the set $EN[XF]$. The semantics of this query q is the same as the semantics of the XF expression evaluated against the initial store and the initial environment.*

This definition can have a few implications w.r.t. expressive power. The input of a query is not only a store, but also an environment has to be taken into account. This can be justified by the XQuery Processing Model[4], which allows the user to set an initial environment. Furthermore, this guarantees modularity, since each query can now be expressed in a function and vice versa (the parameters of a function are just variables of the environment). The output of a query is supposed not to be serialized, i.e., a new store and the result sequence are returned. Again, this can be justified by the Processing Model which states the serialization of the result sequence is optional and that the result can sometimes be processed directly via a DOM interface. Moreover, serialization is not an information-preserving operation, since it discards information about node identity. Hence when queries q_1 and q_2 are composed, we could get a different result for q_2 as we would get by inlining q_1 into q_2, assuming that serialization would be done after evaluating q_1. For these reasons, we will look at the expressive power of expressions instead of queries. From our point of view a query has the same semantics as an expression.

Chapter 3

Expressive power of the fragments

In this chapter we study the expressive power of the XQuery fragments defined in Chapter 2. First we prove some expressibility results in Section 3.1. In Section 3.2 some inexpressibility results are shown. Finally in Section 3.3 we demonstrate the equivalence classes of fragments and their relationship.

3.1 Expressibility Results

Some features that correspond to XQ attributes can be simulated in a fragment that does not have this feature. It is possible that in such a simulation new nodes are constructed that cannot be reached after the termination of the simulation. These nodes can then safely be garbage collected. More precisely, the garbage collection is defined as follows:

Definition 3.1 (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* $\mathrm{rng}(\delta)$ *and for which no node of the tree is in s.*

This garbage collection operation only preserves the nodes in the store that can be accessed through document calls or items in the sequences. We call two expressions equivalent iff they always return the same result upto garbage collection:

Definition 3.2 (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 $e_1 \sim e_2$.*

We say that an expression e can be expressed in a certain fragment F iff there exists an expression $e' \in \mathbf{L}(F)$ such that $e \sim e'$. Note that we do not require the input environments to be in $EN[F]$, since the equality of results of both expressions has to hold for every environment and hence also for all environments in $EN[F]$.

Lemma 3.1. *The "`count`" operator can be expressed in XQ_{at}.*

Proof. From Section 2.1 we know 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 3.2. *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 3.3. *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 3.4. *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 $sequence return
    for $j in (1 to $i) return 1)
```

□

Lemma 3.5. *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 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. More precisely, the function "`count-nodes`" is defined as follows:

```
declare function count-nodes($sequence) {
  if ($sequence) then (
    let $head := (
      for $e1 in $sequence
      let $other := (
```

```
        for $e2 in $sequence
        return (
          if (not($e1 is $e2)) then $e2 else ()
        )
      )
      return
        if (
          for $e3 in $other
          return
            if ($e3 << $e1) then 1 else ()
        ) then $e1 else ()
    ) return
    let $tail := (
      for $e1 in $sequence
      return
        if (not($e1 is $head)) then $e1 else ()
    )
    return (1 + count-nodes($tail))
  )
  else 0
};
```

Each recursion step we filter out the node of the sequence that is first in document order (this node is stored in the variable "`$head`") and we recursively apply the function on the rest of the sequence ("`$tail`"). The recursion stops when applied to an empty sequence. 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 3.6. *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 the item and are in the same order as the corresponding items in the original sequence. If we can define auxiliary (non-recursive) XQ_C^{ctr} functions "`pos`" (to find the position of a node in a sequence of document-ordered nodes), "`encode`" and "`decode`" (to make sure that we do not loose any information in creating a new node for an item in the result sequence of the "`in`" clause) then the following XQ_C^{ctr} be 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 many times. Furthermore the expression e_2 is guaranteed to have the right values for the variables "$\$x$" and "$\$pos$" iff the function "`decode`" is behaves as desired. We only assume that e_2 does not use variables "`$seq`" and "`$newseq`", since they are used in the simulation[1].

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 with its own identity. 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 they have node identity and putting them as a child in a newly constructed node would discard all information we have about the node identity. Therefore we store for a node all information we need to recover 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. If 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) then we can define the functions "`encode`" and "`decode`" as follows:

```
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:boolean return
        element {"boolean"} {if ($e) then 1 else 0}
      case xs:integer return
        element {"integer"} {$e}
      case xs:string return
        element {"string"} {$e}
      case element() return
        element {"node"} {
          attribute {"root"}    {pos(root($e), $rootseq)},
          attribute {"descpos"} {pos($e, root($e)//.)}
        }
      case attribute() return
        element {"node"} {
```

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

```
          attribute {"root"}    {pos(root($e), $rootseq)},
          attribute {"descpos"} {pos($e, root($e)//.)}
        }
      case text() return
        element {"node"} {
          attribute {"root"}    {pos(root($e), $rootseq)},
          attribute {"descpos"} {pos($e, root($e)//.)}
        }
      case document-node() return
        element {"node"} {
          attribute {"root"}    {pos(root($e), $rootseq)},
          attribute {"descpos"} {pos($e, root($e)//.)}
        }
      default return
        element {"string"} {$e}
  }
  return $newseq/*
};

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

To finish the proof, we still need to show how we can express the 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. Their definitions, by means of XQ_C^{ctr} expressions, are:

```
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 ()
};
```

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}. $\square$

3.2 Inexpressibility Results

The previous section provided some expressibility results. In this section we prove that certain features can not be simulated in certain fragments.

The first two inexpressibility results rely on the fact that we cannot distinguish between sequences with the same set or bag representation in some XQuery fragments. To formalize this notion we define set-equivalency and bag-equivalency between evironments and between sequences.

Definition 3.3. *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 the following statements hold:*

- $\mathbf{a} = \mathbf{a}'$*;*
- $\mathbf{b} = \mathbf{b}'$*;*
- $dom(\mathbf{v}) = dom(\mathbf{v}')$ *and* $(\forall s \in dom(\mathbf{v})).(\mathbf{Set}(\mathbf{v}(s)) = \mathbf{Set}(\mathbf{v}'(s)))$*;*
- $\mathbf{x} = \mathbf{x}'$*;*

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 3.7. *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 $St, En' \vdash e \Rightarrow (St, r')$[2], it also holds that $\mathbf{Set}(r) = \mathbf{Set}(r')$.*

Proof. We prove this lemma by induction on the query AST. In this AST the nodes correspond to the $\langle Expr \rangle$ non-terminal of the XQ^R grammar and as a consequence each node corresponds to a construct of rules $[3-18, 24]$ in Figure 2.1. We prove that $\mathbf{Set}(r) = \mathbf{Set}(r')$ for all evaluations $St, En \vdash e \Rightarrow (St, r)$ and $St, En' \vdash e \Rightarrow (St, r')$, with $En, En' \in EN[XQ^R]$ two set-equivalent XQ^R environments and e an expression in XQ^R.

First, consider the leafs of the AST. The result of literals [4,5] is fixed and does not depend on the environment. Hence all evaluations of such an expression yield the same result. A step [16] returns nodes starting from the context item ($\mathbf{x}$) of the environment. Since the sets of context items of set-equivalent environments are equal, these expressions return the same result in both evaluations. Variables [3] return a value from the environment. Since En and En' are set-equivalent, both evaluations return set-equivalent result sequences.

Second, consider expressions that contain subexpressions and assume that the lemma holds for all evaluations of the subexpressions. If e is evaluated

[2]Since e does not contain node constructors in its subexpressions, it's 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.

against En and e_i is the i^{th} subexpression e then we denote the result sequences of the k^{th} evaluation of e_i by $r_{i,k}$ and the environments against which it is evaluated by $En_{i,k}$. If a subexpression is evaluated only once, then we denote this environment by En_i and the result sequence by r_i.

- The subexpressions of a built-in function [6] are evaluated against set-equivalent environments iff the built-in function itself is evaluated against set-equivalent environments. Because two set-equivalent sequences of length one are always the same, we know by the induction hypothesis that built-in functions that only take sequences of one item return the same result when applied to set-equivalent sequences. The function "`distinct-values()`" return the set-representation of the result sequence of the subexpression and is therefore also equal for evaluations against set-equivalent environments. Similar to literals, the functions "`true()`" and "`false()`" always return the same value, no matter against which environment it gets evaluated.

- The if expression [7] first evaluates the clause e_1 before evaluating one of its two subexpressions. Since evaluations of e_1 against set-equivalent environments yield set-equivalent result sequences ($\mathbf{Set}(r_1) = \mathbf{Set}(r'_1)$), we know that both evaluations either evaluate to true or false, and hence the same e_i ($i = 2, 3$) is evaluated for both evaluations of e. Since e_i is evaluated against the same environment as e, we know by induction that they yield set-equivalent result sequences and hence also both evaluations of the if expression return set-equivalent result sequences.

- The for expression [8] evaluates the "`in`" clause e_1. The subexpression e_2 (the "`return`" clause) of the for expression is then evaluated against environments $En_{2,k}$ which are equal to En except for the value of the variable v that is used as iteration variable and which equals the k^{th} item of r_1. Since the results of both evaluations of e_1 are set-equivalent ($\mathbf{Set}(r_1) = \mathbf{Set}(r'_1)$), we know that for each $En_{2,k}$ there exists an $En'_{2,k'}$ such that the variable v has the same value, and vice versa. More precisely, if E_2 is the set of all $(En_{2,k}, r_{2,k})$ pairs and E'_2 the set of all $(En'_{2,k'}, r'_{2,k'})$ pairs then the relation "has the same value for variable v" from E_2 to E'_2 is total and surjective. The only difference between En and the environments in E_2 is the value of the variable v. Since the same also holds for En' and E'_2, we know that the relation "has a set-equivalent environment" from E_2 to E'_2 is also total and surjective. From the induction hypothesis then follows that also the relation "has a set-equivalent result sequence" is total and surjective. Because r and r' are the concatenations of the result sequences of respectively E_2 and E'_2 it hence obviously holds that $\mathbf{Set}(r) = \mathbf{Set}(r')$.

- The let expression [9] binds the result sequence r_1 of clause e_1 to a variable v. From the fact that $En_1 = En$ follows by induction that $\mathbf{Set}(r_1) = \mathbf{Set}(r'_1)$. This value is added to the environment against which e_2 is evaluated. Hence En_2 and En'_2 are also set-equivalent, by induction the results of the return clause e_2 are also set-equivalent ($\mathbf{Set}(r_2) = \mathbf{Set}(r'_2)$) and therefore the result sequences of e are set-equivalent.

- For the concatenation [10] of two sequences it simply holds that $\mathbf{Set}(r_1\texttt{,}\ r_2) = \mathbf{Set}(r_1) \cup \mathbf{Set}(r_2)$ and hence by induction (both e_1 and e_2 are evaluated against the same environment as e) the concatenation returns set-equivalent result sequences when evaluated against set-equivalent environments.

- The binary expressions [11-15] that take two sequences of one item from their subexpressions obviously return the same result when evaluated against set-equivalent environments because, by induction hypothesis, their subexpressions return set-equivalent sequences which are both of length 1 (since we only consider well-defined evaluations). From this we know that $r_1 = r'_1$ and $r_2 = r'_2$ and therefore obviously $\mathbf{Set}(r) = \mathbf{Set}(r')$.

- Path expressions [17] are comparable to for loops, i.e., they compute the concatenation of the results of e_2, which is evaluated against the items of r_1. The result sequence is then the result of sorting the set representation of the concatenated sequence by document order. Hence path expressions return the same result sequences when evaluated against set-equivalent environments.

- The evaluation of typeswitches [18] is similar to the evaluation of the "`if`" clause and hence the same conclusions can be drawn. Note that the result of a type test [19] depends on the environment that contains only one item. Since these enviroments are also set-equivalent, it follows that the item is the same for all the evaluations of this expression. Hence all evaluations of such an expression yield the same result.

- Finally, the evaluation of a function call e [24] against a store St and an environment En is the same as the evaluation of the function body f against the same store St (since we do not have constructors in XQ^R) and a new environment En_f in which we have bound variables (corresponding to the formal parameters) to the actual parameters, specified by the subexpressions of e. Since the values of the actual parameters are by induction set-equivalent, we know that also the environments En_f and En'_f are set-equivalent and hence that the result of both evaluations of f yield set-equivalent result sequences.

□

Lemma 3.8. *It is impossible to simulate a "`count`" function in XQ^R.*

Proof. Clearly the `count` function has not the property of Lemma 3.7. Indeed, 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 3.9. *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 3.7 that evaluations against bag-equivalent environments result into bag-equivalent result sequences. The only real difference in this proof is that we now have to show that there exists a bijection between the "subevaluations" of for expressions. More precisely (using the notation from the proof of Lemma 3.7) the relations "has a bag-equivalent environment" and "has a bag-equivalent result sequence" from E_2 to E'_2 have to be bijections. Finally, the count function obviously returns the same number for evaluations against bag-equivalent environments.

$\square$

Lemma 3.10. *It is impossible to simulate the "`at`" expression in XQ^R_C.*

Proof. Clearly the "`at`" expression has not the property of Lemma 3.9. Indeed, 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 $En_2 = En[\mathbf{v}(\text{"}se\text{"}) \mapsto \langle 2, 1\rangle]$ 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.

$\square$

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 3.4 (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 sets of atomic values A_s, A^{St}, and A^{En} are defined as follows:*

- $A_s = \mathbf{Set}(s) \cap \mathcal{A}$ *(atomic values in a sequence s);*
- $A^{St} = (\mathrm{rng}(\nu) \cup \mathrm{rng}(\sigma)) \cap \mathcal{A}$ *(atomic values in the store St);*
- $A^{En} = \bigcup_{s \in \mathrm{rng}(\mathbf{v})} A_s$ *(atomic values in the environment En).*

The sizes Δ^{forest}_{St} and Δ^{tree}_{St} for the store St are defined as follows:

- Δ^{forest}_{St} *is the size of the forest in St, i.e., $\Delta^{forest}_{St} = |V|$*
- Δ^{tree}_{St} *is the size of the largest tree of the forest in St, i.e., $\Delta^{tree}_{St} = \max(\bigcup_{n_1 \in V}\{c | c = |\{n_2 | (n_1, n_2) \in E^*\}|\})$*[3]

The function **size** *maps an atomic value to the number of cells needed to represent this item on the tape of a Turing Machine.*

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

Definition 3.5 (Largest Sequence/Item Sizes). *Consider the evaluation* $St, En \vdash e \Rightarrow (St'', v)$ *of a query* e*, where* $St = (V, E, <, \nu, \sigma, \delta)$*,* $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$*, and* $\Gamma_v(St'') = St' = (V', E', <', \nu', \sigma', \delta')$*. The largest input and output sizes for sequences and items are defined as follows:*

- *The* largest input sequence size *is* $d_I^s = \max(\{|s|\,|s \in \mathrm{rng}(\mathbf{v})\} \cup \{\Delta_{St}^{tree}\})$.
- *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* $d_O^s = \max(\{|v|, \Delta_{St'}^{tree})$.
- *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 upperbound $f(n)$ where n is the input size, then we have to show that this upperbound 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 reasons. The first reason is that all upperbound 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 upperbound 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. Following example illustrates the definition of largest input and output sequence/item size of a query.

Example 3.2.1. *Consider the following stores* St_1, St_2*, environment* En*, expression* e*, and result sequence* v *in the evaluation* $St_1, En \vdash e \Rightarrow (St_3, v)$ *with* $\Gamma_v(St_3) = St_2$*:*

- $St_1 = (V_1, E_1, <_1, \nu_1, \sigma_1, \delta_1)$ *with*
  - $V_1 = V_1^d \cup V_1^e \cup V_1^t \cup V_1^a$ *with* $V_1^d = \{n_0\}$*,* $V_1^e = \{n_1, n_2, n_4\}$*,* $V_1^t = \{n_3, n_4\}$*,* $V_1^a = \{\}$
  - $E_1 = \{(n_0, n_1), (n_1, n_2), (n_1, n_4), (n_2, n_3), (n_4, n_5)\}$
  - $<_1 = \{(n_2, n_4)\}$
  - $\nu_1 = \{(n_1, \text{“}a\text{”}), (n_2, \text{“}b\text{”}), (n_4, \text{“}b\text{”})\}$
  - $\sigma_1 = \{(n_3, \text{“123”}), (n_5, \text{“}Brussels\text{”}\}$
  - $\delta_1 = \{(\text{“}text.xml\text{”}, n_0)\}$

The serialization of the root nodes of the store St_1 is "document{<a>123Brussels},". *The largest input item size is 8 (since the largest input item is "Brussels") and the largest input sequence size is 6 (since there are 6 items in the store and the store has only one root).*

- $En = (\{\}, \{\}, \{\}, \bot)$

- $e =$
```
let $x := doc("text.xml")
return ($x/a, $x/a,
        element{"res"}{$x/a,$x/a},
        element{"res"}{$x/a/b},"Antwerp")
```

- *The result store after garbage collection* $\Gamma_v(St_3) = St_2 = (V_2, E_2, <_2, \nu_2, \sigma_2, \delta_2)$ *with*
  - $V_2 = V_2^d \cup V_2^e \cup\ V_2^t \cup V_2^a$ *with* $V_2^d = V_1^d$, $V_2^e = V_1^e \cup \{n_6,\ n_7,\ n_8,\ n_{10},\ n_{12},\ n_{13},\ n_{15},\ n_{17},\ n_{18},\ n_{20}\}$, $V_2^t = V_1^t \cup \{n_9, n_{11}, n_{14}, n_{16}, n_{19}, n_{21}\}$, $V_2^a = V_1^a$
  - $E_2 = E_1 \cup \{(n_6, n_7), (n_6, n_{12}), (n_7, n_8), (n_7, n_{10}), (n_8, n_9), (n_{10}, n_{11}), (n_{12}, n_{13}), (n_{12}, n_{15}), (n_{13}, n_{14}), (n_{15}, n_{16}), (n_{17}, n_{18}), (n_{17}, n_{20}), (n_{18}, n_{19}), (n_{20}, n_{21})\}$
  - $<_2 = <_1 \cup \{(n_7, n_{12}), (n_8, n_{10}), (n_{13}, n_{15}), (n_{18}, n_{20})\}$
  - $\nu_2 = \nu_1 \cup \{(n_6, \text{“}res\text{”}),\ (n_7, \text{“}a\text{”}),\ (n_8, \text{“}b\text{”}),\ (n_{10}, \text{“}b\text{”}),\ (n_{12}, \text{“}a\text{”}),\ (n_{13}, \text{“}b\text{”}),\ (n_{15}, \text{“}b\text{”}),\ (n_{17}, \text{“}res\text{”}),\ (n_{18}, \text{“}b\text{”}),\ (n_{20}, \text{“}b\text{”})\}$
  - $\sigma_2 = \sigma_1 \cup \{(n_9, \text{“123”}), (n_{11}, \text{“}Brussels\text{”}), (n_{14}, \text{“123”}), (n_{16}, \text{“}Brussels\text{”}), (n_{19}, \text{“123”}), (n_{21}, \text{“}Brussels\text{”})\}$
  - $\delta_2 = \delta_1$

  The garbage collection removed deep-equal nodes of the children of n_6 and $n_1 7$ that could no longer be reached by document loading δ_3 or result sequence v.

- *The result sequence $v = \langle n_1, n_1, n_6, n_{17}, \text{“}Antwerp\text{”}\rangle$.*

The largest output item size is 8 (for "Brussels") and the largest output sequence size is 11 (the size of the tree under node n_6.

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 3.11. *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 $d_O^i \leq p(d_I^i)$ for some polynomial p*

Proof. For each polynomial p that has $\mathbb{N}$ or $\mathbb{N}^2$ as its domain there always exists an increasing polynomial function p' such that p' is an upperbound for p. Therefore we assume all functions that are used as an upperbound in this and following proofs to be increasing functions. This assumption is needed to prove the lemma by induction on the size of the abstract syntax tree of the query q. In this AST 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 2.1.

First, consider the leafs of the query AST. Literals [4,5] return constant values, while steps [16], and variables [3] return some items from the input (store and environment) of the expression and hence it is obvious that for all leaf expressions $d^i_O \leq p(d^i_I)$ 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 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 [6, 7, 10-15, 17, 18, 23, 26] do not alter the environment or the store before passing them to their subexpressions, so $d^i_{I_k} = d^i_I$ for all 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 polynomialy bounded in size by the items in the result of their subexpressions, while all items in the result store of these expressions are items in the result store and/or sequence of their subexpresesions. 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 [8] evaluate their second subexpression e_2 for each result of their first subexpression e_1 with this result bound to a variable $\$x$. 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 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 $d^i_O \leq p(d^i_I)$
- Similarly, the let expression [9] binds a variable $\$x$ to the result of its first subexpression, adds this variable to the environment and passes the new environment and the result store of the first subexpression as input to the second subexpression e_2. The output of e_2 is then the output of the entire expression. From the induction hypothesis 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 upperbound 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.

□

Lemma 3.12. *It is impossible to simulate a "`count`" function in $XQ^{ctr,to}$.*

Proof. Clearly the "`count()`" function has not the property of Lemma 3.11. Indeed, if we consider the empty store St_0, the environment $En = (\{\}, \{\}, \{(\text{“}\$input\text{”}, \langle 1, \ldots, 1\rangle)\}, \perp)$, 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 3.13. *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. We prove this lemma by induction on the size of the abstract syntax tree of the query q. In this AST 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 2.1.

First, consider the leafs of the query AST. Literals [4,5] return constant values, while steps [16], and variables [3] 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 3.11, 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 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 [6, 7, 10-15, 18, 21, 26] 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.

- All basic built-in functions [6], if expressions [7], the binary expressions [10-15], and typeswitch expressions [18] 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.

- The sum function [21] returns a number that is the sum of a number of values of the input sequence (output of the subexpression). This result is bounded by $d^s_{O_1}.d^i_{O_1} \leq p_{k_1}(d^s_I).2^{p_{k_2}(\log(d^s_I), d^i_I)}$ 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 [26] can worst-case copy the entire input store, such that the output sequence size $d^s_O \leq O(2.d^s_I)$, and $d^i_O \leq O(\log(d^s_I), d^i_I)$, which is still within the bounds that we have to show.

- The let expression [9] binds a variable to the result of its first subexpression, adds this variable to the environment and passes the new environment and the result store of the first subexpression as input to the second subexpression. The output of the second expression is the output of the let expression. From the induction hypothesis follows that the output sizes for the first expression 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 upperbounds 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 [8] of the form "`for` $\$x$ `at` $\$y$ `in` e_1 `return` e_2" bind the variables $\$x$ and $\$y$ each iteration to one item. The largest item of $\$y$ needs at most $\log(d^s_I)$ space and the largest item of $\$x$ needs at most d^i_I space. Hence, for each iteration of e_2 it holds that $d^s_{I_2} = d^s_I$ and $d^i_{I_2} = \max(d^i_I, \log(d^s_I)) \leq p((\log(d^s_I), d^i_I))$ for some polynomial p. From the induction hypothesis then follows that for each iteration of e_2 it holds that

$d^s_{O_2} \leq p_1(d^s_I)$ and $d^i_{O_2} \leq p_2(\log(d^s_{I_2}), d^i_{I_2})$ for some polynomials p_1 and p_2 (we omit the details of this computation). Since the number of iterations is bounded by the result sequence of e_1, we know that at most $d^s_{O_1} \leq p_3(d^s_I)$ iterations can occur, where p_3 is a polynomial. The result sequences of all iterations are concatenated in order to compute the end result and hence the output sizes are bounded as follows: $d^s_O \leq p_3(d^s_I).p_1(d^s_I) \leq p_4(d^s_I)$ and $d^i_O \leq p_3(d^s_I).p_2(\log(d^s_I), d^i_I) \leq p_5(\log(d^s_I), d^i_I)$ for some polynomials p_4 and p_5.

- Path expressions [17] 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. □

Lemma 3.14. *It is impossible to simulate the "`to`" expression in $XQ^{ctr}_{at,S}$.*

Proof. Clearly the "`to`" expression has not the property of Lemma 3.13. Indeed, if we consider the empty store St_0, the environment $En = (\{\}, \{\}, \{(\text{"}\$input\text{"}, \langle k \rangle)\}, \perp)$, 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 3.15. *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. Similar to the proof of Lemma 3.13 we prove this lemma by induction on the AST. However, in this proof we will omit some details that were discussed earlier. In the proof of Lemma 3.13 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 the following hold:

- $d^s_O \leq p_1(d^s_{I_1}, 2^{d^i_{I_1}})$,
- $d^i_O \leq p_2(\log(d^s_{I_1}), d^i_{I_1})$,
- $d^s_{I_1} \leq p_3(d^s_I, 2^{d^i_I})$ 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

- $d^i_O \leq p_2(\log(p_3(d^s_I, 2^{d^i_I})), p_4(\log(d^s_I), d^i_I)) \leq p_2(p_7(\log(d^s_I), \log(2^{d^i_I})), p_4(\log(d^s_I), d^i_I))$ for some polynomial p_7 and hence $d^i_O \leq p_8(\log(d^s_I), d^i_I)$ for some polynomial p_8.

Hence we can use induction in order to prove this lemma. We know that for all XQ^{ctr}_{at} expressions there was 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 $XQ^{ctr,to}_{at}$ expressions have output sizes within the bounds specified by this lemma when evaluated against an $XQ^{ctr,to}_{at}$ environment.

□

Lemma 3.16. *It is impossible to simulate recursive function definitions in* $XQ^{ctr,to}_{at}$.

Proof. Clearly there are expressions in XQ^R that do not have the property of Lemma 3.15. Indeed, if we consider the empty store St_0, the environment $En = (\{\}, \{\}, \{(\text{"\$}input'', k)\}, \perp)$, 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 $d^s_I = 1$ 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. We will first define the set of possible outputs for an expression e when the input values are restricted to a certain alphabet of atomic values and the largest input sequence size is smaller than a given number S.

Definition 3.6 (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 $d^s_I \leq S$ 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 3.17. *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 AST where each expression corresponds to the $\langle Expr \rangle$ non-terminal of the XQ_{at}^{ctr} grammar and as a consequence each node corresponds to a construct of rules $[3-18, 26]$ of Figure 2.1.

First, consider the leafs of the query AST. Literals [4,5] return for all evaluations the same atomic value, steps [16] do not return atomic values and variables [3] only return atomic values originated from the input. All these expressions do not change the input store. Hence it holds that the number atomic values in the possible results is bounded by $N + 1$.

All other expressions have subexpressions. Note that many expressions [6, 7, 10-15, 18, 26] 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, follows that the number of atomic values in the possible results is bounded by $p(N,S)$ for some polynomial p.

We now discuss the remaining expresions.

- The let expression [9] binds a variable to the result of its first subexpression, adds this variable to the environment and passes the new environment of the first subexpression as input to the second subexpression. This in fact means that the second subexpression is evaluated against an alphabet of size $N' < p_N(N,S)$ and a store and environment with a maximal sequence size of $S' < p_S(S)$ (Lemma 3.13) for some polynomials p_N, p_S. From the induction hypothesis then follows that the number of atomic values in the possible results is bounded by $p'(N', S') < p'(p_N(N,S), p_S(S)) < p(N,S)$ for some polynomials p and p'.

- The for expression [8] first evaluates the subexpression in the "`in`" clause. From Lemma 3.13 we know that the number of items in the result sequence of this subexpression is bounded by $p_S(S)$ for some polynomial p_S and the number of different atomic values in the possible results is bounded by $p_N(N,S)$ for some polynomial p_N. The expression in the return clause is evaluated at most $p_S(S)$ 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 $N' < p_N(N,S)$ and a store and environment with a maximal sequence size of $S' < p_S(S)$. Hence, from the induction hypothesis follows that the number of atomic values in the possible results for each evaluation is bounded by $p'(N', S') < p'(p_N(N,S), p_S(S)) < p''(N,S)$ 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 $p_S(S).p''(N, S) < p(N, S)$ for some polynomial p.

- Path expressions [17] are 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.

$\square$

Lemma 3.18. *It is impossible to simulate the* `sum` *operator in* XQ_{at}^{ctr}

Proof. The XQ_S expression "`sum($x)`" does not have the property of Lemma 3.17. 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}. $\square$

3.3 Equivalence classes of XQuery fragments

As we have shown in the two previous subsections, 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 3.7 (Equivalent Fragments). *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 simulates 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 is as expressive as 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 3.1 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. 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., $(\forall XF_1 \in C_1, XF_2 \in C_2).(XF_1 \succ XF_2)$). The intuitive meaning of the dotted borders between equivalence classes in Figure 3.1 is that they divide the set of fragments in two parts: one in which we can express the construct that is used as a label of the border and one in which we know that we cannot express it. We will now prove that Figure 3.1 correctly shows the relationships between all 64 XQuery fragments. In order to prove this correctness of the figure, we first prove three simple lemmas.

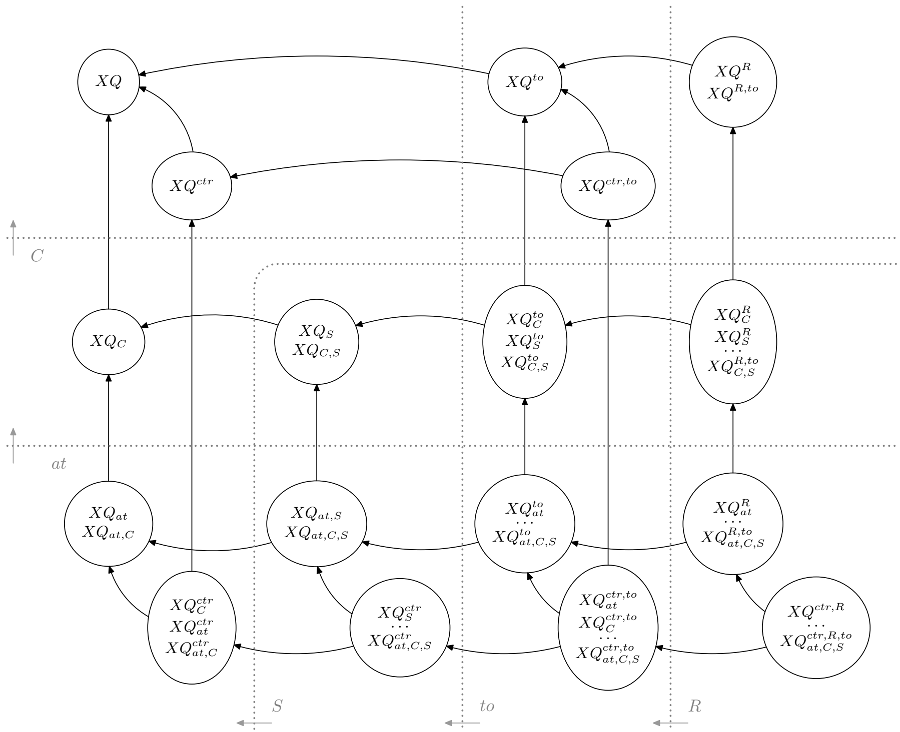

Figure 3.1: Equivalence classes of XQuery fragments

Lemma 3.19. *All fragments that appear in the same node in Figure 3.1 are within the same equivalence class.*

Proof. We show for all nodes containing more than one fragment that all of the fragments within the same node are equivalent:

- $XQ_{at} \equiv XQ_{at,C}$: this follows from Lemma 3.1
- $XQ_{C}^{ctr} \equiv XQ_{at}^{ctr} \equiv XQ_{at,C}^{ctr}$: this follows from Lemmas 3.1, 3.6
- $XQ_{S} \equiv XQ_{C,S}$: this follows from Lemma 3.2
- $XQ_{at,S} \equiv XQ_{at,C,S}$: this follows from Lemma 3.2
- $XQ_{S}^{ctr} \equiv XQ_{C,S}^{ctr} \equiv XQ_{at,S}^{ctr} \equiv XQ_{at,C,S}^{ctr}$: this follows from Lemma 3.2, 3.6
- $XQ_{C}^{to} \equiv XQ_{S}^{to} \equiv XQ_{C,S}^{to}$: this follows from Lemmas 3.2, 3.4
- $XQ_{at}^{to} \equiv XQ_{at,C}^{to} \equiv XQ_{at,S}^{to} \equiv XQ_{at,C,S}^{to}$: this follows from Lemmas 3.1, 3.2, 3.4
- $XQ_{C}^{ctr,to} \equiv XQ_{S}^{ctr,to} \equiv XQ_{C,S}^{ctr,to} \equiv XQ_{at}^{ctr,to} \equiv XQ_{at,C}^{ctr,to} \equiv XQ_{at,S}^{ctr,to} \equiv XQ_{at,C,S}^{ctr,to}$: this follows from Lemmas 3.1, 3.2, 3.4, 3.6
- $XQ^{R} \equiv XQ^{R,to}$: this follows from Lemma 3.3
- $XQ_{C}^{R} \equiv XQ_{S}^{R} \equiv XQ_{C,S}^{R} \equiv XQ_{C}^{R,to} \equiv XQ_{S}^{R,to} \equiv XQ_{C,S}^{R,to}$: this follows from Lemmas 3.2, 3.3, 3.4
- $XQ_{at}^{R} \equiv XQ_{at,C}^{R} \equiv XQ_{at,S}^{R} \equiv XQ_{at,C,S}^{R} \equiv XQ_{at}^{R,to} \equiv XQ_{at,C}^{R,to} \equiv XQ_{at,S}^{R,to} \equiv XQ_{at,C,S}^{R,to}$: this follows from Lemmas 3.1, 3.2, 3.3, 3.4
- $XQ^{ctr,R} \equiv XQ_{C}^{ctr,R} \equiv XQ_{S}^{ctr,R} \equiv XQ_{C,S}^{ctr,R} \equiv XQ^{ctr,R,to} \equiv XQ_{C}^{ctr,R,to} \equiv XQ_{S}^{ctr,R,to} \equiv XQ_{C,S}^{ctr,R,to} \equiv XQ_{at}^{ctr,R} \equiv XQ_{at,C}^{ctr,R} \equiv XQ_{at,S}^{ctr,R} \equiv XQ_{at,C,S}^{ctr,R} \equiv XQ_{at}^{ctr,R,to} \equiv XQ_{at,C}^{ctr,R,to} \equiv XQ_{at,S}^{ctr,R,to} \equiv XQ_{at,C,S}^{ctr,R,to}$: this follows from Lemmas 3.1, 3.2, 3.3, 3.4, 3.5, 3.6

□

Lemma 3.20. *Let n_1 and n_2 be two nodes in the graph of Figure 3.1 such that there is a directed path from n_1 to n_2. If XF_1 is a fragment in node n_1 and XF_2 is a fragment in node n_2 then $XF_1 \succeq XF_2$.*

Proof. From the figure we know that in both n_1 and n_2 there are equivalent fragments $XF_3 \equiv XF_1$ and $XF_4 \equiv XF_2$ such that $\mathbf{L}(XF_3)$ is a superset of $\mathbf{L}(XF_4)$ so we know for sure that all expressions that can be expressed in XF_4 and hence in XF_2, can be expressed in XF_3 and XF_1.

□

Lemma 3.21. *The dotted borders in Figure 3.1 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 labels 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.*

Proof. We prove the correctness of the dotted borders 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:

- *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 3.14 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 3.16 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 3.8 follows that "`count()`" cannot be expressed in XQ^{R} and from Lemma 3.12 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 3.10 follows that "`at`" cannot be expressed in XQ^{R}_{C}. From Lemma 3.12 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 3.1.
- *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 3.12 and Lemma 3.8 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 3.18 follows that "`sum()`" cannot be expressed in XQ^{ctr}_{C}.

□

Theorem 3.1. *For the graph in Figure 3.1 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. The proof consists of two parts:

- If XF_1 and XF_2 are in the same node then it follows from Lemma 3.19 that they are equivalent.
  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 by Lemma 3.21 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 by Lemma 3.20 that $XF_1 \succeq XF_2$ and since XF_1 and XF_2 appear in a different node they are not equivalent, so $XF_1 \succ XF_2$.
  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 3.1 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 $XF_1 \not\succ XF_2$.

$\square$

Chapter 4

Related Work

The problem of expressive power of languages has been widely studied in Computer Science literature. There are works in knowledge languages literature, programming languages and querying languages literature. Obviously, we are more interested into the last two topics. One of the first works, about expressive power of programming languages, is "Beating the Averages" by Paul Graham [6]. In this paper he argues that some languages are more powerful than others, and posits a hypothetical middle of the road language called Blub. He describes the paradox arising when a Blub programmer consider other languages. There are two kind of languages for the Blub programmer: languages obviously less expressive than Blub because they are missing some features the programmer is used to, and languages with a lot of useless things as Blub is enough for him - he thinks in Blub. The author starts from this paradox to prove that it is not objectively possible to say that a language has more expressive power that another (as they are usually complete), but that the preference is always subjective since based on what the programmer is used to. It treats the notion of programming language power as a continuum, with assembly language at the bottom and some other languages at the top. He claims that it correctly orders languages in terms of raw programming facility (he consider only domain-independent features): a language A is more powerful than language B if A contains features that couldn't be obtained in B without writing an interpreter for (a subset of) A in B.

Besides this general notion of expressive power of languages, there have been many works that have studied the expressive power of database query languages. One of the first works we consider, studies the expressive power of relational algebra as a core of expressive power often used to compare the power of other, more complex, languages. In [11], Paredaens offers a method to detect whether a relation is an answer to any question for a given relational database, giving a good introduction to deeper formal studies of the relational algebra.

A famous result, of the comparison between SQL and relational algebra, is that SQL cannot express recursive queries such as reachability. This problem has been widely studied by Libkin, [10], who studied the expressive power of SQL and proved that recursion (introduced in SQL3) adds expressive power to SQL2 because reachability queries cannot be expressed over unordered types over ordered domains, than the new construct is justified. Another work that studies the problem of recursion expressive power is [3]: in this work, the authors claim that the primitive operations of database query language should be organized around types. In particular, they study the property of structural recursion over bags and sets, and prove that recursive queries such as transitive closure are not definable with the help of grouping, summation, and product over columns, and standard rational arithmetic.

The idea of studying the expressive power of a language, breaking the language itself into fragments, is borrowed by [1]. In this paper, the authors study structural properties of each of the main sublanguages of XPath commonly used in practice. The paper is divided into two parts: first, they characterize the expressive power of these fragments in terms of logics and tree patterns; second, they study closure properties, focusing on the ability to perform basic Boolean operations while remaining into the fragment. To our knowledge, our work is the first one to address the problem of different fragments of XQuery, with the aim of discovering different degree of expressive power.

Chapter 5

Conclusion

This work investigates the expressive power of XQuery, trying to focus on fragments of the language itself 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. As future work, we want to compare the expressive power of our XQuery fragments with other languages such as Relational Algebra, SQL and XPath in order to better understand the expressive power of this XML query language.

Bibliography

[1] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *International Conference on Database Theory*, 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 (`http://www.w3.org/TR/xquery/`), 2003.

[3] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computes Science*, 149(1):3–48, 1995.

[4] 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 (`http://www.w3.org/TR/xquery-semantics/`), 2004.

[5] M. Felleisen. On the expressive power of programming languages. In *3rd European Symposium on Programming*, volume 432, pages 134–151. 1990.

[6] P. Graham. Beating the averages. Franz Developer Symposium, Cambridge (`http://paulgraham.com/avg.html`), 2001.

[7] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *Second International XML Database Symposium*, 2004.

[8] H. Katz, editor. *XQuery from the Experts*, chapter 5 (Introduction to the Formal Semantics), pages 299–302. Addison Wesley, 2003.

[9] M. Kay, N. Walsh, and H. Zongaro. XSLT 2.0 and XQuery 1.0 serialization. W3C Working Draft `http://www.w3.org/TR/xslt-xquery-serialization/`, 2004.

[10] L. Libkin. Expressive power of SQL. Theoretical Computer Science, 2003.

[11] J. Paredaens. On the expressive power of the relational algebra. Information Processing letters, vol. 7, number 2, 1978.