Charles University

Faculty of Mathematics and Physics

Doctoral Thesis

XPath, XSLT, XQuery: Formal Approach

Pavel Hloušek

Department of Software Engineering
Charles University
Malostranské náměstí 25
Praha, Czech Republic

2005

Declaration

This is to certify, that I wrote this Thesis on my own and that the references include all the source of information I have exploited. I authorize Charles University to lend this document to other institutions or individuals for the purpose of scholarly research.

Praha, November 17, 2005

Pavel Hloušek

Acknowledgements

I'd like to thank my supervisor Jaroslav Pokorný who offered the topic to me for an interesting topic, advice, fruitful discussions and also for spotting some errors.

A huge thanks I owe to my family. My wife Kristyna and my son Kristian both prooved their endless patience. I'd also like to thank my parents who have been of great support to me through my under- and postgraduate studies.

The last but not least I'd like to thank the company I work for, EXCON, for providing me with a quiet office, a computer and an internet connection and for allowing me so to work on my thesis.

Contents

1 Introduction **1**
- 1.1 Contributions 2
- 1.2 Structure 3

2 Background **5**
- 2.1 Historical background 5
- 2.2 W3C Standards 7
- 2.3 Related Work 9

3 Notation **12**
- 3.1 Notation Rules 12
- 3.2 Basic Types 12
- 3.3 Sequences 12
- 3.4 Tuples and Tuple Streams 14
- 3.5 Named Functions 15
- 3.6 Context 15
- 3.7 Abbreviations 17
- 3.8 Running Example 17

4 XPath **19**
- 4.1 Introduction 19
  - 4.1.1 XPath 1.0 19
  - 4.1.2 XPath 2.0 23
- 4.2 XPath Syntax 26
- 4.3 XPath Formal Semantics 29
  - 4.3.1 Expression to Sequence 30
  - 4.3.2 Expression to Boolean Value 31
  - 4.3.3 Function Call Arguments 32
- 4.4 Sorting in XPath 32
  - 4.4.1 By Example 33
  - 4.4.2 Formally 35
  - 4.4.3 Conclusion 39
- 4.5 Quantified Expressions 39
- 4.6 Conclusion 44

5 XQuery **45**
- 5.1 Introduction 45
  - 5.1.1 Constructors 46
  - 5.1.2 FLWOR Expressions 46
  - 5.1.3 Functions 49
- 5.2 XQuery Syntax 49
- 5.3 XQuery Formal Semantics 50
  - 5.3.1 FLWOR Expression 52
  - 5.3.2 Prolog 56
  - 5.3.3 Semantic Functions Relationships 57
- 5.4 Sorting in XQuery 57
  - 5.4.1 XQuery Core Inconsistency 59
  - 5.4.2 The Idea 60

5.4.3 Formally . . . 60
5.4.4 By Example . . . 64
5.4.5 Conclusion . . . 67
5.5 Conclusion . . . 67

6 XSLT **68**
6.1 Introduction . . . 68
6.1.1 Stylesheet and Template . . . 68
6.1.2 Applying Templates . . . 69
6.1.3 Context . . . 70
6.1.4 Built-in Template Rules . . . 71
6.1.5 Conflict Resolution . . . 71
6.1.6 Modes . . . 71
6.1.7 Parameters . . . 72
6.1.8 Variables . . . 74
6.1.9 Named Templates . . . 75
6.1.10 Functions . . . 75
6.1.11 Sequence Iteration . . . 76
6.1.12 Conditional Execution . . . 76
6.1.13 Constructing New Content . . . 77
6.1.14 Copying . . . 78
6.1.15 Sorting . . . 79
6.1.16 Grouping . . . 79
6.1.17 Keys . . . 80
6.1.18 Stylesheet Import and Inclusion . . . 81
6.1.19 Initiating a Transformation . . . 81
6.2 XSLT Syntax . . . 81
6.2.1 XPath Expression vs. Sequence Constructor . . . 83
6.2.2 Excluded XSLT Elements . . . 85
6.3 XSLT Core . . . 86
6.3.1 Notation . . . 87
6.3.2 Flow Control . . . 89
6.3.3 Parameters . . . 89
6.3.4 Template Rules . . . 96
6.3.5 Functions and Named Templates . . . 105
6.3.6 Sorting . . . 107
6.3.7 Grouping . . . 109
6.3.8 Keys . . . 116
6.3.9 Copying . . . 117
6.3.10 Sequences . . . 120
6.3.11 The Core Syntax . . . 120
6.4 XSLT Formal Semantics . . . 121
6.5 Conclusion . . . 122

7 Conclusions **124**
7.1 Future Work . . . 124

List of Figures

1 Relationships among selected W3C specifications. A higher drawn specification builds on a lower drawn one. 9
2 Running example DTD 17
3 Running example XML 18
4 XPath axes 20
5 XPath axes sample data 21
6 XPath Syntax, Part 1 27
7 XPath Syntax, Part 2 28
8 XPath Semantics 30
9 XPath Boolean Semantics 32
10 XPath Argument List Semantics 32
11 Proof of Lemma 4.2. 39
12 Proof of Theorem 4.1: $S' = S$. 40
13 Proof of Theorem 4.1: Derivation of $\prec_{S'}$. 41
14 Proof of Lemma 4.5. 42
15 Proof of Lemma 4.6. 43
16 XQuery Example: Tuple stream 49
17 XQuery Syntax 51
18 XQuery Semantics 52
19 XQuery Tuple Semantics 53
20 XQuery Ordering Semantics 56
21 XQuery Prolog Semantics 57
22 XQuery Semantic Functions Cycle 58
23 XSLT Compact Syntax 82
24 XSLT Syntax Mapping 84
25 XSLT, Procedure *DistinctParams* 91
26 XSLT, Procedure *PassAllParams* 92
27 XSLT, Procedure *RemoveTunnelParams* 95
28 XSLT, Procedure *RemoveTemplateRules* 99
29 XSLT, Procedure *RemoveTemplateRulesWithModes* 103
30 XSLT Core Syntax 122
31 XSLT Core Semantics 123

"In the beginning there was nothing.
God said, 'Let there be light!' And there was light.
There was still nothing, but you could see it a whole lot better."
— Ellen De Generes

1 Introduction

This thesis formally studies properties of standard query languages for XML: XPath, XQuery and XSLT.

XML – a buzz word, data format, technology, empty bubble, syntax, tree or graph data representation, etc. These things pop up in minds of people when they think about XML. It is no surprise that there are people who hate it as well as people who love it. But personal attitudes are not important. The important thing is whether someone finds XML useful. We can see now that XML gained its audience and that this audience is big – ranging from industries to database researchers – definitely big enough to take XML seriously.

Once a lot of data is stored in a particular data format, people need tools to extract information from it, change it, restructure it. At such a moment query languages come to play their part, as they provide basic data manipulation tools. As XML has become popular, a great deal of XML data has started to be stored in separate documents, native XML databases, relational or object databases, or in various hybrid systems. So, naturally, the need for XML query languages came out, which resulted in a number of proposals.

It is good to have standards, no doubt. Usage of many proprietary query languages can make data maintenance tough and integration of data from different IT vendors even infeasible. So, large industries like IBM, Microsoft or Oracle push hard to have XML query languages standardized. This standardization process is sheltered by W3C – the World Wide Web Consortium.

Looking for a standard query language that perfectly fits the needs of everyone is similar to looking for the Holy Grail. Some people want to hit the road in one direction, some the other, and some even think that it does not exist.

The XML audience can be roughly divided into two groups of the Holy Grail seekers, in terms of their anticipation. We call them a database and a document community. Whereas the database community thinks of an XML document as of a database from which they want to extract information, the document community, on the other hand, thinks of an XML document as of a text document with some structural markup that makes it easy to transform a source XML document to a specific format.

The dichotomy in understanding of what is the purpose of an XML document is reflected in the different expectations from the one and perfect XML query language. Therefore, instead of having the one and only, we have three standard XML query languages: XSLT, XQuery and XPath.

Each of these languages is supposed to be the Grail for a different community. While XSLT is handy for transformation writers and meets the needs of the document community, XQuery, on the other hand, is trying to be perfect for the database community as it is suitable for data extraction or complex restructuring tasks. Since there is a subset of tasks common for both of these

languages, the XPath language has been designed to be a subset of both XSLT and XQuery. Its purpose is to identify parts of an XML document.

First, it was XPath 1.0 and XSLT 1.0 that have been recommended in late nineties by W3C as the first versions of standards. Soon after that, W3C began its work on successor languages that incorporate strong typing based on XML Schema introducing XQuery as a new language: XPath 2.0, XSLT 2.0 and XQuery 1.0.

The database researches have studied the properties of the early versions of XPath and to some extent of XSLT recently. However, not much work has been done on the new upcoming standards that work with a different data model. Therefore, I turned my attention to them.

The initial question was: Having two Holy Grails – XSLT and XQuery – coming from the document and database community, respectively, which one is stronger? In other words, the topic I chose was a comparison between XSLT and XQuery in terms of their expressive power.

But I left this question soon as digging myself into the details of the languages slightly turned my attention to several difficulties on a lower level. After I studied the insides of XSLT, XQuery and XPath for some time, I focused on the following problems.

- XQuery Core is broken. XQuery Core is a subset of XQuery that is supposed to be equally expressive. The formal specification of XQuery Core claims that sorting is not expressible in XQuery Core unless the data model is expanded.

- XSLT has no formal semantics. There is no formal semantics for the whole language and so it is hard to reason about the properties of the language.

- XSLT has no core language. However, there have been some theoretical proposals for XSLT core language, but the relation between the proposal and XSLT stays unclear, i.e. we cannot tell whether the core language is semantically a strict subset of XSLT or not.

The problem of broken XQuery Core disattracted W3C's formal semantics of XQuery for my purposes. Therefore, I decided to build my own formal semantics that is expressive enough to capture the semantics of both XQuery and XSLT.

The problem of missing XSLT Core with clear relation to XSLT motivated me to look for mappings from various XSLT constructs to simpler ones.

My dissertation thesis addresses these two problems and I hope has it brought in some light.

1.1 Contributions

The following is a list of contributions of my thesis.

- XPath 2.0 is capable of sorting sequences. This is one of the main results of my thesis, which has an impact on the following point.

- Formal semantics of XQuery sorting can be expressed in the data model of the XQuery specification. Due to that, we know now that XQuery Core

is not broken in terms of its ability to sort. This is a contradiction to what the authors of the formal semantics specification thought.

- XSLT Core language. I identified a pretty small set of XSLT instructions that can be used to simulate the others. We show that even template rules can be removed from the language without weakening its expressive power.
- XSLT formal semantics. I provided the formal semantics for XSLT Core and thus for the whole language.
- Unified formal semantics framework for XPath, XQuery and XSLT. The formal semantics provided in this thesis for all three languages uses the same framework.

1.2 Structure

This thesis is structured as follows.

Chapter 2 Background introduces the background of this theses. This includes an overview of research on semistructured data and early query languages for XML. Description of the family of XML-related specifications published by W3C is also provided to make the relationships among not a few W3C specifications clear. Finally, an overview of related work is given with a focus on the expressive power of XPath, XQuery and XSLT or their formal semantics.

Chapter 3 Notation introduces notation used throughout the thesis. We define notation for basic types of the XML data model items, nodes and atomic values. Further, we define there notation for sequences, tuples, tuple streams as they are essential for the definition of formal semantics of the languages. Finally, we define there evaluation context and some operations that handle it. Also the running example XML data is appended to the end of this chapter, since it does not fit anywhere else.

Chapter 4 XPath is dedicated to the XPath language. Basically it comprises three parts: introduction, syntax and semantics definition, and results.

We start this chapter with an informal introduction to the syntax and semantics of both versions of XPath in section 4.1. Whereas we explain all the features of XPath 1.0 like location steps, axes, node tests, predicates, etc. we focus only on the main differences that distinguish XPath 2.0 from its 1.0 predecessor like sequences, strong typing, new if-then-else, for-in-return and quantified expressions, etc. The reader that is familiar with XPath 1.0 but not with XPath 2.0 can skip the introduction to XPath 1.0 and inspect only the introduction to XPath 2.0.

We follow with the syntax of XPath 2.0 in section 4.2. We chose to define only the syntax of Core XPath. Core XPath uses a reduced syntax of full XPath language but every expression expressible in full XPath can be normalized to an equivalent Core XPath expression.

The formal semantics of XPath 2.0 is defined in section 4.3. The formal semantics is defined for a fragment of XPath 2.0 that we need for proofs in the following sections and chapters, which includes semantics of the for-in-return, if-then-else, quantified expressions and function calls.

The results are presented in sections 4.4 and 4.5. The first presents one of the main results of our work that we published in [33]. We formally prove that XPath 2.0 is capable of sorting arbitrary sequences of items in this section. In the second one, we formally prove that quantified expressions are only syntactic sugaring of the language.

Finally, the conclusions are provided in section 4.6.

Chapter 5 XQuery is focused on the XQuery 1.0 language. This chapter comprises again of three parts: an introduction to the language, syntax and semantics, and results.

Since XQuery 1.0 is a superset of XPath 2.0 language, we reduce the introductory section 5.1 to an explanation of constructs that extend XPath: constructor expressions, FLWOR expressions and user-defined functions. All of them are demonstrated on examples.

The syntax of XQuery is provided in section 5.2. Several grammar productions that are not important to us are omitted like productions for constructor expressions and all type dependent grammar productions.

Next, we define the formal semantics including sorting semantics of the ORDER BY clause in a FLWOR expression in section 5.3. The sorting semantics is omitted in the official formal semantics [22]. The authors of this specification say that it is impossible to define it, because tuples and tuple streams exceed the limits of the data model and so it is not possible to define the semantics formally while staying in the data model. Since we do not limit ourselves to stay within the bounds of the standard XML data model, we define the formal semantics even for sorting.

The main result of this chapter is introduced in section 5.4. We formally prove that every XQuery expression can be rewritten to an equal XQuery expression without an ORDER BY clause. The proof is a non-trivial extension of the fact that XPath is capable to sort sequences that we proved in the previous chapter on XPath. This has a surprising consequence: the semantics of sorting is expressible within the bounds of the standard XML data model.

Finally, the conclusions are provided in section 5.5.

Chapter 6 XSLT is dedicated to the XSLT 2.0 language.

It starts with an lengthy introduction to the language in section 6.1. The length of this introduction is caused by the huge number of instructions in XSLT, all of which have to be described.

The syntax of XSLT is given in section 6.2. Instead of using the XML syntax of XSLT, we define our own non-XML syntax to make the following proofs clear and concise. We also give a list of omitted instructions with an explanation why an instruction is not considered.

The main result of this chapter is concentrated to section 6.3, where the Core XSLT language is identified for the selected XSLT syntax. We formally prove this language being the core of XSLT in the sense that all language constructs can be rewritten using this core language.

The semantics of Core XSLT is provided in section 6.4 using the same semantics framework as for XPath and XQuery.

Finally, conclusions are given in section 6.5 that summarize the results.

Chapter 7 Conclusions briefly summarizes the main contributions of this thesis and makes some hints for the future work.

2 Background

This chapter is organized as follows. First, we give a brief overview of research activities that precede nowadays standards in section 2.1. Then, we explain the relationships among W3C standards for XML and XML query languages in section 2.2. Finally, we provide references to related work in section 2.3.

2.1 Historical background

This section gives a brief overview of research on semistructured data and non-standard or pre-standard query languages for XML.

Semistructured data. Semistructured data is a precursor to the XML research in the database community. It started with a need to query data that was incomplete or irregular in its structure [1, 13]. The main motivation for this research field has been the idea of querying data on the web.

The works in this field include project Tsimmis [17], system Lore together with its query language Lorel [50, 2], UnQL with its calculus UnCal [14, 27], and system Strudel with its StruQL [24] query language. There are also domain specific applications that demonstrate the semistructured approach to data like MailQL [32, 34].

Project Tsimmis was targeted to data integration and is one of the first works in the field of semistructured data. It tried to solve the problem of representing and querying heterogeneous data sources with a single query. They presented a flexible data model that lacks a schema: Object Exchange Model (OEM), which is a simple graph-oriented model with labeled edges and with data stored at leaf nodes.

System Lore has been developed at Standford University since 1995. In its beginnings it was based on OEM. The OEM data model is noticeably similar to usual data models for XML, but in comparison to XML data models it lacks any notion of schema or DTD. Since semistructured data do not gain as much attention as XML today, Lore has been migrated to fully support the XML data recently. Currently, Lore is a complete prototype DBMS that uses various indexing techniques and that has a query optimizer for its own query language Lorel. Lorel is strongly based on OQL [15], which is a query language for object databases with a navigational syntax that is typical for languages that query a graph-oriented data model.

The UnQL language has been designed for semistructured data having in mind besides the Tsimmis project also system AceDB [53] for biologists. Its calculus called UnCal that is based on lambda calculus distinguishes UnQL from other languages for semistructured data, as it enables to derive optimization rules.

Strudel has been developed at AT&T Research Labs as a web-site management system. It uses the model of wrappers that transform externally stored data into a graph model, which in turn can be queried with a site-definition query language StruQL. The result of imposing a StruQL query is a site model that can be converted to browsable HTML pages or can be set as an input for another StruQL query. Since StruQL is designed to query irregular graph-oriented data, it belongs to query languages for semistructured data, though its application is not as general as Lorel's or UnQL's.

The MailQL query language demonstrates even more domain specific language. It has been developed as a language that is suitable to query a database of such irregular data structures as email messages.

XML. XML [11, 12] appeared as a successor to SGML [36] and has been designed specifically to meet the needs of publishers. Since that time, XML has been widely adopted and nowadays it is used for many different purposes such as publication, data exchange, database management.

The XML data is usually modeled as an oriented tree or graph with data stored in leaf nodes, which makes the XML data models very similar to data models used for semistructured data. Though the semistructured approach to data is more general, the huge attention paid to XML moved the attention of database researches to the field of XML. Besides development of query languages for XML, the current interest of database researchers in this area ranges over problems like storing XML data in databases, indexing XML data, defining types for XML documents (schemas), etc. Since this work is focused on query languages for XML, we briefly mention some of them.

As XML itself originated in the document community, it was the document community that gave birth to the first languages that operate on XML data. The two languages that appeared together were XPath [19], XSL [3] and XSLT [18, 40].

XPath has been proposed as a language suitable to identify parts of an XML document. It follows a navigational syntax enriched with predicates, whose purpose is to limit the result set, and built-in functions. The intent was to have a simple language with non-XML syntax that can be used on its own or which can be used as a part of a higher language like XSLT.

XSLT has been designed as a language suitable to transform an XML document into another (potentially non-XML) document. It is based on template rules that are applied to a source XML document, where each template rule defines a simple transformation and where each transformation defined in a template rule can explicitly invoke application of template rules on a selected part of the source XML document, recursively. XSLT can be viewed as a kind of a rewriting system, but as a language it is not of much interest in the database community, perhaps because it lacks formal basis and because it defines a lot of constructs.

The languages proposed for XML by the database community are especially XQL [51], XML-QL [21], Quilt [16], and recently XQuery [8].

XQL is a simple language that is very similar to XPath in its syntax. The main distinction between these two languages is that XQL is able to return newly created structures whereas XPath can return atomic values or nodesets of the original document at most. XQL's syntax is primitive, however, it offers some powerful expressions that include grouping, renaming and join.

XML-QL is one of the very first efforts to design a query language for XML. A query is composed of two parts: a pattern matching part and a construction part. The pattern matching part is used to bind values to variables that are used in the construction part of a query. It is possible to use regular path expressions for recursive structures and joins in the pattern matching part. The language is equipped with skolem functions to ease data transformation. It is also possible to express grouping with aggregate functions. So far, we mentioned the powers of the language, but there are also drawbacks. The main of them is that both pattern matching and construction parts use XML syntax, so larger queries

tend to be long and hardly readable.

Quilt evolved directly from XML-QL. The biggest change happened to the pattern matching part of a query that left the XML syntax of XML-QL and that moved towards XPath expressions. Further, the pattern matching part was divided to two types of clauses: FOR and LET, where each has a different way of binding the results of an XPath expression to a variable. While the FOR clause is used to iterate over the items in the result, binding items one by one to an associated variable, the LET clauses binds all the items in the result to a variable at once. Quilt formed a basis of nowadays W3C standard query language for XML: XQuery. In fact, Quilt has been renamed to XQuery and further developed under its new name.

2.2 W3C Standards

The World-Wide Web Consortium (W3C) is an organization that is creating standards for the web technologies. W3C creates a lot of XML-related standards. In this section, we provide a brief introduction to a family of standards that are important to our work. We describe the standards that have been or are being prepared and the relationships among them.

W3C defines several states of a document that is or is being prepared to be a standard. For simplicity, we need to distinguish only two of them: a draft and a recommendation. Whereas *draft* is not a finished standard and so it is a subject to a change, a *recommendation* is a finished standard that does not change. This implies that talking about drafts can be dangerous, since they can change. However, all the drafts that we work with are drafts that have already passed the last call, so major changes are not probable.

The following is a list of the core XML-related recommendations and drafts published by W3C.

- Extensible Markup Language (XML) 1.0 – recommendation
- Namespaces in XML 1.0 – recommendation
- XML Information Set – recommendation
- XQuery 1.0 and XPath 2.0 Data Model – draft
- XML Path Language (XPath) 2.0 – draft
- XQuery 1.0: An XML Query Language – draft
- XQuery 1.0 and XPath 2.0 Functions and Operators – draft
- XQuery 1.0 and XPath 2.0 Formal Semantics – draft
- XSL Transformations (XSLT) 2.0 – draft
- XML Schema family

XML. XML as a language is a subset of Standard Generalized Markup Language (SGML) [36], which is a markup language that is used for text processing. Syntax of XML is described in XML 1.0 [11] and Namespaces in XML 1.0 [9] specifications. We assume that the reader is familiar with XML enough

and do not dig into details of writing elements, attributes and other syntactic constructs and terms like well-formedness of an XML document. Both W3C recommendations are followed by their 1.1 versions [12, 10] that addresses some technical issues like allowing arbitrary names in XML tags, new-line conventions, etc.

XML Information Set. XML Information Set [20], shortly Infoset, defines what information can be retrieved from an well-formed XML document. Information set of a well-formed XML document consists of *information items* like document information item, where each information item has a set of associated properties like children or attributes. This document primarily serves the purpose of providing a consistent set of definitions of information in an XML document that can be referred to by other specifications.

XML Data Model. Once the information stored in an XML document is described, a data model for XML data can be defined that can be used for query languages, which is done in XQuery 1.0 and XPath 2.0 Data Model specification [25]. Notice, that however the data model has in its name XPath and XQuery, the same data model is used even for XSLT.

The data model says that XML document consists of *items*, which are either *nodes* or *atomic values*. The nodes are of seven kinds: document, element, attribute, text, comment, processing instruction, and namespace. The specification defines that each node has a *unique identity*, but no identity is assigned to atomic values. Nodes are organized in an oriented tree with a *children* property and with a reverse *parent* property.

Further, the specification defines *document order*, which is a total order on nodes of an XML document. Informally, one node precedes another node in the document order if both nodes are in the same XML document and the first node's XML representation starts closer to the beginning of an XML document then the second node's.

As the order is important, the essential data structure heavily used throughout the data model is a *sequence* of items – not a set. Sequences are not allowed to be nested, so a sequence cannot contain a sequence. However, it is possible to syntactically nest sequences, but the result is always flattened. The data model makes no distinction between an item and a singleton sequence containing that item.

The data model defines *types* of values. The typing system is that defined by the family of XML Schema specifications [23, 54, 7]. As we are not interested in typing, we do not go in more detail here.

XML Query Languages. We just described the specifications that define the syntax of an XML document and the information contained in it. Next specifications cover the query languages XPath, XQuery and XSLT.

The simplest of the languages is XML Path Language (XPath) 2.0 [5], which evolved from much simpler 1.0 version [19]. The authors of XPath declare that its main purpose is to identify parts of an XML document and to serve as a basis for XQuery and XSLT. The language has a navigational non-XML syntax, with functions and constructs for variable binding. There is a such a lot of built-in functions that a separate specification XQuery 1.0 and XPath 2.0 Functions and Operators [43] has been started.

XQuery 1.0: An XML Query Language specification [8], defines a superset of XPath 2.0, so a query in XPath 2.0 is also a query in XQuery 1.0 that returns the same result. Since the interconnection between these two languages is so

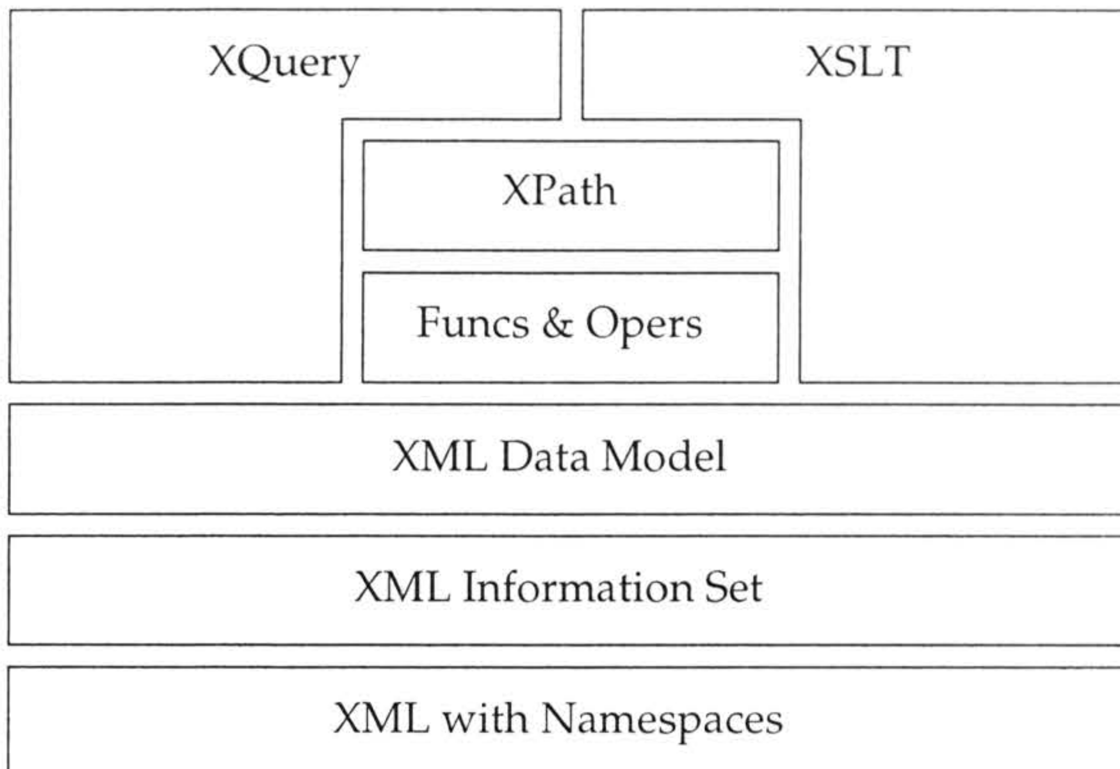

Figure 1: Relationships among selected W3C specifications. A higher drawn specification builds on a lower drawn one.

strong, they share the same specification that formally defines the semantics of these two languages: XQuery 1.0 and XPath 2.0 Formal Semantics [22].

XSL Transformations (XSLT) 2.0 specification [40] defines a language that is supposed to meet the needs of the document community. It is a direct successor to the successful 1.0 version [18]. It uses an XML syntax to define *transformation rules* that are applied to an input XML document to result in a text document that has not to be an XML document. The transformation rules are described in terms of templates. Subset of XPath can be used to define *patterns* in template rules that are matched against nodes in an input XML document and full XPath 2.0 expressions can be used inside template's bodies.

Figure 2.2 gives an overview of relationships among its selected specifications. Each specification builds on specifications that are drawn below it.

2.3 Related Work

Since there is a lot of activity in the field of XML, we tried to choose only works that study either the expressive power of standard XML query languages XPath, XSLT and XQuery in all its versions or their formal semantics. We divided the references to related work according to the language(s) that are in focus.

XPath 1.0 Both XPath 1.0 and XSLT 1.0 have not been studied much by the computational language theorists when published, but now there is a lot of research conducted in this area, specifically on XPath. Wadler [55] has been the first to provide formal semantics for XPath and his work has been of great influence to us.

Properties of various dialects of XPath have been studied by Benedikt et al. in [4] with focus on the expressive power of the dialects.

Gottlob et al. proposed Core XPath in [30, 31] as a powerful language with

linear combined complexity. This language is both syntactically and semantically a subset of XPath. Their concept has been extended to Conditional XPath by Marx in [44, 45, 46], which is a first-order complete language over trees still with linear combined complexity. Conditional XPath exceeds the expressive power of XPath.

XSLT 1.0 On the other hand, as XSLT is not much in the focus of the database community, not much formal work has been done here.

Since a formal definition of XSLT semantics is missing, there are a few papers that take a fragment of XSLT, for which the semantics is defined formally. Probably the first of them is Xemantics [42] – formal semantics for a fragment of XSLT defined using a term rewriting approach and Wadler's semantics of XPath.

Bex, Maneth and Neven developed a formal model for a fragment of XSLT in [6]. This model has been used to examine properties of XSLT like its expressive power compared to languages like XML-QL, k-pebble transducers [47], or monadic second order logic. It has been proved that a pattern is not necessary in template rules, since it can be pushed inside the body of a template rule. However, it still is an open question whether their XSLT fragment is or is not a strict subset of XSLT.

XQuery 1.0 A lot of attention is paid to XQuery by database researchers. But as the formal semantics of XQuery is a W3C standard, the most of them focus on an optimal evaluation of queries. The official formal semantics of XQuery [22] is based on the former work [26] by Fernandez et al. Its main drawback is that it is not complete – the formal semantics of ordering is not defined.

Another approach to defining the formal semantics of XQuery was taken by Yang et al in [57]. They defined it in an object-oriented semantic model using ObjectZ [52] language. The work was focused only on the main semantic constructs, but the ordering semantics is defined, though in a procedural manner rather then declaratively.

In [28] an algebra is developed that can be used for query optimization. The paper defines three λ-calculus based algebras: semantics, language and physical algebras, so optimization is available at three different levels. This provides more fine-grained control compared to the formal semantics of XQuery by W3C, whose main purpose is to formally define only the high-level semantics of the language.

Jagadish et al. presented a tree algebra for XML called TAX in [38]. This algebra has been designed for bulk operations on tree-modeled data such that database-style optimizations are possible. TAX is an extension of relational algebra and it is complete for relational algebra with aggregation. Though this algebra has not been designed specifically for XQuery, it is able to express any non-recursive XQuery query. TAX is used for optimization in Timber database [49].

XSLT 2.0 There has not been much research on XSLT 2.0 yet. The formal semantics of XSLT is not covered by a W3C standard. In [56], Yang et al. defined the formal semantics in ObjectZ [52]. It is nothing else but plain formalization of the informal XSLT 2.0 specification.

XQuery 1.0 and XSLT 2.0 An interesting paper [41] by Stephan Kepser proves that both XQuery 1.0 and XSLT 2.0 are Turing-complete languages.

Fokoue et al. provide compilation from a fragment of XSLT 2.0 to XQuery

1.0 in [29]. This work is a contribution to an effort to compare the expressive powers of XSLT and XQuery, because it shows that the template choosing mechanism of XSLT is expressible in XQuery. A comparison of the expressive powers of XSLT and XQuery has been the main motivation to us too, but we pushed hardly to simplify XSLT first. One of our results shows that the template choosing mechanism is a syntactic sugar in XSLT and so there is no need to prove its expressibility in XQuery.

XML Schema XML Schema is a standard whose main purpose is to define types for XML documents. Novak and Zamulin define formal semantics for XML Schema in [48]. Though we do not consider typing, there is a connection to our work. The connection lies in the way sequences are modeled: as ordered sets. The reader may wonder, whether this representation allows for multiple occurrences of atomic values in a sequence. The paper does not mention this problem, but we explain why this is possible.

3 Notation

In this chapter, we define basic notation that is used throughout this thesis. At the end of this section, a running example is provided.

3.1 Notation Rules

The definitions of notation in this chapter and throughout the thesis use the following rules.

The syntactic types are written in italics font and with the first letter capitalized, *Thus*.

With $o : T$ we denote that an object o is of type T.

With $\langle o_1, \ldots, o_n \rangle$ we denote a tuple with n objects.

With $T \rightarrow T'$ we denote a function type with domain T and range T'. The symbol $\rightarrow$ is left-associative, so $T \rightarrow T' \rightarrow T''$ is understood as $(T \rightarrow T') \rightarrow T''$. We use this notation for functions with multiple parameters.

3.2 Basic Types

We stick to the standard model for XML data [25], as it was introduced in 2.2. It defines that the basic data types are nodes and atomic values, which we denote by *Node* and *Atomic*. Further it defines a derived data type of an item, which is either a node or an atomic value. We denote the type of an item with *Item*.

3.3 Sequences

With $Set(T)$ we denote a type "set with objects of type T". For example, $Set(Item)$ denotes a type "set of *Item* objects". With $|S|$ we denote the cardinality of S.

With $Seq(T)$ we denote a type "sequence of objects of type T". A sequence S is a tuple $\langle S, \prec_S \rangle$, where S is a set of objects and $\prec_S \subseteq S \times S$ is a total order on them. We define following dereference operators for a set of items of a sequence, and for an ordering of a sequence, respectively:

$$\begin{aligned} \mathcal{I}_{\langle S, \prec_S \rangle} &= S \\ \sqsubset_{\langle S, \prec_S \rangle} &= \prec_S \end{aligned}$$

The most often used sequence type is $Seq(Item)$. Here we have to note that representing a sequence as an ordered set has a little technical difficulty, but which can be handled easily. The difficulty comes from the fact that atomic values have no identity in the standard model for XML data. So, in our representation it is not possible to have a sequence that contains two equal atomic values. To fix this problem, we assign a *sequence identity* to all atomic values in every sequence. We call this sort of identity a sequence identity to emphasize that it is bound to a sequence and not to the universe of all atomic values. A sequence identity of an atomic value can be naturally interpreted as its position within a modeled sequence. Two atomic values from different sequences have always a different sequence identity.

As we need to distinguish, whether we care about identity of atomic values in a sequence or not, we write all set operators and the equality operator (such as $\cup$, $\cap$, $\setminus$, $=$, etc.) either without a subscript or with a *val* subscript. For

example, let $\langle S_1, \prec_{S_1} \rangle$ and $\langle S_2, \prec_{S_2} \rangle$ be two sequences, then with $S_1 \cup S_2$ we denote a union of items in S_1 and S_2 that is based on sequence identity for atomic values, while with $S_1 \cup_{val} S_2$ we denote a union of items in S_1 and S_2 that is based on values for atomic values. In both cases, the union is based on node identity for nodes. Analogically $=$ and $=_{val}$ compare two atomic values according to their identities and according to their values, respectively.

Now, we can identify items that precede a given item in a sequence in a simple way. It is handy sometimes to identify not only all preceding items but also the given item itself. Therefore we define a partial ordering $\preceq_S$ for a sequence $\langle S, \prec_S \rangle$, which is in fact a reflexive closure of $\prec_S$.

$$\begin{aligned} S &= \{i_1, \ldots, i_n\} \\ \preceq_S &= \prec_S \cup \{\langle i_j, i_j \rangle | j \in \{1, \ldots, n\}\} \end{aligned}$$

In accordance with this definition we provide a new dereference operator that given a sequence returns the partial order just defined.

$$\sqsubseteq_{\langle S, \prec_S \rangle} = \preceq_S$$

According to XQuery 1.0 and XPath 2.0 Data Model, we make no difference between an object of type *Item* and a singleton sequence with that object. Formal equivalence follows.

$$\begin{gathered} i : \mathit{Item}, \mathrm{S} : \mathit{Seq}(\mathit{Item}) \\ i \equiv \mathrm{S} \text{ iff } \mathrm{S} = \langle \{i\}, \emptyset \rangle \end{gathered}$$

Not only logical description of sequences as sets with an order is sufficient. To describe sequences in examples, we use XPath syntax to literally denote a sequence with items. For example, the following represents a sequence with items 1, 2, and 3.

$$(1,2,3)$$

Position of an item x in a sequence S is defined naturally as cardinality of a set consisting of preceding items and item x itself.

$$pos(x, \mathrm{S}) = |\{ y \mid y \sqsubseteq_{\mathrm{S}} x \}|$$

To finish with sequences, we define a *concatenation operator* $\circ$ that given two sequences $\langle S_1, \prec_{S_1} \rangle$ and $\langle S_2, \prec_{S_2} \rangle$ returns a single sequence $\langle S, \prec_S \rangle$ with items from the first sequence followed by items from the second sequence preserving the order of items in both initial sequences.

$$\langle S_1, \prec_{S_1} \rangle \circ \langle S_2, \prec_{S_2} \rangle =_{\text{def}} \langle S, \prec_S \rangle$$

$$\begin{aligned} S &= S_1 \cup S_2 \\ \prec_S &= \{ \langle x_1, x_2 \rangle \mid \begin{array}{l} x_1, x_2 \in S_1 \text{ implies } x_1 \prec_{S_1} x_2 \\ x_1, x_2 \in S_2 \text{ implies } x_1 \prec_{S_2} x_2 \\ x_1, x_2 \notin S_i \text{ implies } x_1 \in S_1, x_2 \in S_2 \end{array} \} \end{aligned}$$

3.4 Tuples and Tuple Streams

The XQuery 1.0 specification explains the semantics of the language in terms of *tuples* forming a *tuple stream*. However, the formal semantics specification does not formally define these terms and thus the authors think that it is not possible to expressing ordering semantics formally. As we need the semantics for the whole language defined formally, we define the *Tuple* and *Tuples* types to represent a tuple and a tuple stream, respectively.

Let's start informally. XQuery uses FOR and LET clauses to define a tuple stream that is composed of tuples. Each tuple represents values bound to all variables from the FOR and LET clauses. This tuple stream is reduced by the WHERE clause, ordered by the ORDER BY clause and finally iterated over to evaluate the RETURN clause of an XQuery query. Each value bound to a variable is generally a sequence of items.

With Tuple_n we denote the type of a tuple of cardinality n. When the cardinality is not important, we simply write *Tuple* without a suffix. A value in a tuple is always a sequence of items, $\mathit{Seq}(\mathit{Item})$, since it represents a value bound to a variable. Notice, that a tuple itself keeps no information about variable names it stores values for.

$$\mathit{Tuple}_n \quad =_{\text{def}} \quad \langle \underbrace{\mathit{Seq}(\mathit{Item}), \ldots, \mathit{Seq}(\mathit{Item})}_{n} \rangle$$

With Tuples_n we denote the type of a tuple stream. It is a sequence of tuples with cardinality n and a sequence of distinct variable names common for all tuples in a tuple stream. When the cardinality is not important, we simply write *Tuples* without a suffix. By *VarName* we denote a type for variable names.

$$\mathit{Tuples}_n \quad =_{\text{def}} \quad \langle \mathit{Seq}(\mathit{Tuple}_n), \mathit{Seq}(\mathit{VarName}) \rangle$$

Notice, that both *Tuple* and *Tuples* types exceed the limits of the standard XML data model that does not allow tuples and sequences of anything else but items.

We write an instance of type *Tuples*, i.e. a tuple stream, as a triplet $\mathbb{T} = \langle T, \prec_T, \theta_T \rangle$, where T is a set of tuples, $\prec_T$ is a total order on T, and θ_T is a sequence of variable names. We define the following dereference operators for it.

$$\begin{array}{rcl} \mathcal{I}_{\langle T, \prec_T, \theta_T \rangle} & = & T \\ \sqsubset_{\langle T, \prec_T, \theta_T \rangle} & = & \prec_T \\ \Theta_{\langle T, \prec_T, \theta_T \rangle} & = & \theta_T \end{array}$$

Variable names in a sequence θ_T correspond to "columns" in a tuple sequence $\langle T, \prec_T \rangle$. Thus, the first variable name corresponds to the first member of each tuple, the second variable name corresponds to the second member of each tuple, etc. For a tuple stream $\mathbb{T} : \mathit{Tuples}_n$ it is always true that $n = |\Theta_{\mathbb{T}}|$.

3.5 Named Functions

Next, we define named functions. Each user-defined function is defined by a name and a body expression. Usually, functions have named parameters. The *Functions* type describes a function that given a function name returns a sequence of parameter names, and a body expression.

$$\mathit{Functions} \quad =_{\text{def}} \quad \mathit{FnName} \rightarrow \langle \mathit{Seq}(\mathit{VarName}), \mathit{Expression} \rangle$$

If we have a function $F : \mathit{Functions}$, then $F(\mathit{fn}) = \langle \theta_{\mathit{fn}}, e \rangle$ means that function named *fn* is defined by a body expression e and that this function takes parameters with names in θ_{fn}.

3.6 Context

To evaluate expressions correctly, we need a notion of evaluation context. The *Context* type consists of a variable binding, function definitions, context item, context position, and context size. The set of a context item, position and size is usually referred to as to a *focus*.

We start with a definition of a variable binding. We define type *VarBinding* to be a function that takes a variable name and returns a sequence of items. Notice that a variable is always evaluated to a sequence of items. Due to the equivalence of an item and a singleton sequence of that item, it is possible to bind variables to atomic values.

$$\mathit{VarBinding} \quad =_{\text{def}} \quad \mathit{VarName} \rightarrow \mathit{Seq}(\mathit{Item})$$

We define type *Context* to represent an evaluation context. It is a quintuple of a variable binding *VarBinding*, a set of definitions of named functions *Functions*, a context item *Item*, and two integer numbers representing context position and context size.

$$\mathit{Context} \quad =_{\text{def}} \quad \langle \mathit{VarBinding}, \mathit{Functions}, \mathit{Item}, \mathit{Integer}, \mathit{Integer} \rangle$$

The *extend* function extends the context with a new variable binding. It takes a variable name *VarName*, a value *Seq*(*Item*) as a sequence of items that should be assigned to it and a variable binding *VarBinding*. It returns a new variable binding. The definition of its type follows.

$$\mathit{extend} : \mathit{VarName} \rightarrow \mathit{Seq}(\mathit{Item}) \rightarrow \mathit{VarBinding} \rightarrow \mathit{VarBinding}$$

Now, we define the semantics of the *extend* function. With v we denote a variable name, with Swe denote a sequence of items to be assigned to v. With ρ_1 we denote the original variable binding and with ρ_2 we denote the resulting variable binding. Recall that we represent a variable binding as a function.

$$\begin{aligned} \mathit{extend}(v, \mathrm{S}, \rho_1) &= \rho_2 \\ \rho_2(w) &= \text{if } (w = v) \text{ then S else } \rho_1 \\ \mathit{lookup}(v, \rho) &= \rho(v) \end{aligned}$$

The *lookup* function returns a value assigned to a variable. It takes a variable name *VarName* and a variable binding *VarBinding* and returns a sequence of items. The definition of its type follows.

$$lookup : VarName \longrightarrow VarBinding \longrightarrow Seq(Item)$$

Now, we define its simple semantics. To find a value of a variable with name v, we call the given variable binding function ρ.

$$lookup(v, \rho) \quad = \quad \rho(v)$$

To simplify notation, we will use the following to express a context $C = \langle \rho, F, item, pos, last \rangle$ extended by a variable with name v bound to value S.

$$C(v \rightarrow \mathrm{S}) \quad \equiv \quad \langle extend(v, \mathrm{S}, \rho), F, item, pos, last \rangle$$

It is often needed to bind a set of variables instead of a single variable. Therefore for a sequence of variable names $\mathbb{V}$, a tuple t of values to be bound, and context C we define $C(\mathbb{V} \rightarrow t)$ to be a context with variable names from $\mathbb{V}$ bound to their respective values in t. Formal definition follows. Let $\mathbb{V}$ be a sequence of distinct variable names with items $\{v_1, \ldots, v_n\}$ and with order defined by indexes of v_i. Let $t = \langle \mathrm{S}_1, \ldots, \mathrm{S}_n \rangle$ be a tuple.

$$C(\mathbb{V} \rightarrow t) \quad \equiv \quad C(v_1 \rightarrow \mathrm{S}_1) \cdots (v_n \rightarrow \mathrm{S}_n)$$

We can notice that the order in which the variables are bound is not important when variable names are distinct.

Usually, a sequence of variable names comes from a tuple stream, so for a tuple stream $\mathbb{T}$ we write $C(\Theta_{\mathbb{T}} \rightarrow t)$.

Since sometimes only focus changes in a context, we define a function to handle focus change. Function $focus(C, i, \mathrm{S})$ changes current focus to an item $i \in \mathcal{I}_{\mathrm{S}}$. Focus is not always defined, e.g. in function calls, therefore we define function *nofocus* to set focus to undefined values, which we denote by an empty set symbol.

$$\begin{array}{lcl} focus & : & Context \longrightarrow Item \longrightarrow Seq(Item) \longrightarrow Context \\ nofocus & : & Context \longrightarrow Context \end{array}$$

$$\begin{array}{rcl} focus(\langle \rho, F, item, pos, last \rangle, i, \mathrm{S}) & = & \langle \rho, F, i, pos(i, \mathrm{S}), |\mathcal{I}_{\mathrm{S}}| \rangle \\ nofocus(\langle \rho, F, item, pos, last \rangle) & = & \langle \rho, F, \emptyset, \emptyset, \emptyset \rangle \end{array}$$

```
<!DOCTYPE archive[
  <!ELEMENT archive      (cd*)>
  <!ATTLIST archive
             type   CDATA #IMPLIED>
  <!ELEMENT cd           (title, author*, artist*, year, genre?)>
  <!ELEMENT author       (first,last)>
  <!ELEMENT artist       (first,last)>
]>
```

Figure 2: Running example DTD.

3.7 Abbreviations

To simplify formulas, we write $\vec{x}$ instead of $x_1, \ldots, x_n$. Thus $\vec{x}$ does *not* denote a tuple. If we want to denote a tuple in a simplified form, we write $\langle\vec{x}\rangle$. So, for example the following holds.

$$\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle \quad \equiv \quad \langle \vec{x}, \vec{y} \rangle$$

3.8 Running Example

Here we define sample DTD and sample XML data for a running example of the thesis, given in Figure 2 and Figure 3. The XML document cd.xml describes a CD archive with compact discs with either music or speech. Each CD is described by a title and a year it was published. Any CD can have an optional list of authors and an optional list of artists performing. Also information about genre can be attached to a CD.

Knihovna Mat.-fyz. fakulty
Informatické oddělení
Malostranské náměstí 25
118 00 Praha 1

```
<archive>
    <cd type="music">
        <title>Tubular Bells</title>
        <artist><first>Mike</first><last>Oldfield</last></artist>
        <year>1992</year>
        <genre>rock</genre>
    </cd>
    <cd type="speech">
        <title>Dasenka</title>
        <author><first>Karel</first><last>Capek</last></author>
        <artist><first>Karel</first><last>Hoeger</last></artist>
        <year>1984</year>
    </cd>
    <cd type="music">
        <title>Hejira</title>
        <author><first>Joni</first><last>Mitchel</last></author>
        <artist><first>Joni</first><last>Mitchel</last></artist>
        <artist><first>Jaco</first><last>Pastorius</last></artist>
    </cd>
    <cd type="music">
        <title>Tubular Bells II</title>
        <artist><first>Mike</first><last>Oldfield</last></artist>
        <year>1992</year>
        <genre>rock</genre>
    </cd>
</archive>
```

Figure 3: Running example XML document cd.xml.

4 XPath

XPath is the base stone of the XML querying. As such, it is used in both XSLT and XQuery. In this thesis, we consider XPath in version 2.0, which was created by extending XPath 1.0 with some powerful features.

This chapter is organized as follows. We first explain what XPath is intended to be used for and we demonstrate its features by many examples in section 4.1. Then, we precisely define the syntax of XPath in section 4.2 and the formal semantics of XPath in section 4.3. Then, we show a surprising fact that XPath 2.0 is capable of sorting sequences. This is proved in section 4.4. Finally, quantified expressions are proved to be syntactic sugaring of XPath in section 4.5.

4.1 Introduction

In this section, we first introduce all the main features of the language XPath in its 1.0 version like node sets, location paths, axes, node tests, predicates, typing, and functions. All of these are explained using a lot of examples.

Then, we examine what has been changed and/or added to the language in its 2.0 version, namely adoption of the XML Schema typing model, replacement of unordered nodesets with ordered sequences, and new expressions like if-then-else conditional expression, for-in-return iterative and variable binding expression, and quantified expressions some- and every-in-satisfies.

4.1.1 XPath 1.0

XPath 1.0 has been designed to easily identify or match nodes in an XML document with the intention to be used either standalone or in XSLT 1.0 and XPointer. It is really simple to express queries like "return titles of all CDs", "return titles of all music CDs", or "select CDs with more then two artist performing" in XPath.

```
/archive/cd/title
/archive/cd[@type = "music"]/title
/archive/cd[count(artist) > 2]
```

The result of evaluating an XPath 1.0 expression is either an atomic value or a *set of nodes* from the source XML document usually referred to as node-set. As the most complicated structure of a result is a set, there is no ordering information about the items in the result set. The absence of order information in a result has been understood as a particular disadvantage. Specifically, information about document order of nodes is lost in a result.

Location path, step. As one can see, XPath 1.0 uses compact, navigational, non-XML syntax. The expressions are based on location paths. A location path describes a traversal of an XML document that consists of steps. Each step consists of an *axis*, *node test*, and zero or more *predicates*.

```
axis::node test[predicate]*
```

Axes. An axis specifies a relation between the current node and a node specified by the step. For example, child axis refers to nodes that are children

Axis	Nodes	Meaning
self	4	current item
child	8	children of the current item
descendant	8, 11, 12	descendants of the current item, i.e. children and children of children, etc.
parent	2	parent of the current item
ancestor	1, 2	ancestors of the current item, i.e. parent and parent of parent, etc.
following-sibling	5	siblings that follow the current item
preceding-sibling	3	siblings that precede the current item
following	5, 8, 9, 10, 11, 12	all nodes from the document that are after the current item in the document order
preceding	1, 2, 3, 6, 7	all nodes from the document that are before the current item in the document
attribute		attribute nodes of the current item
descendant-or-self	4, 8, 11, 12	union of descendant and self axes
ancestor-or-self	1, 2, 4	union of ancestor and self axes

Figure 4: XPath axes and their meaning. For each axis a list of node numbers is provided, where number represents a node that satisfies the axis. The sample XML tree with numbered nodes is given in Figure 5 with current node number 4.

of the current node, or parent axis refers to a node that is a parent of the current node.

Full list of axes is provided in Figure 4. For each axis its meaning is explained and demonstrated with a nodeset for sample XML data from Figure 5.

Node test. A node test filters the set of nodes given by the axis relation according to the node name or node kind. The simplest node test is a name test that is specified by a tested name, e.g. child::cd returns those child nodes of the current node that have their name equal to cd. It is also possible to use wildcard * to skip the name test, thus parent::* returns the parent node of the current node regardless of its name.

The second type of a node test is a test of a node kind. For each of the seven node kinds there is a function that is true only if the tested node is of that specific kind, e.g. node(), text(), attribute(). For the full list of these functions see section 4.2. For example, child::text() returns only text children of the current node.

Predicates. Finally, predicates are used to filter the set of nodes that resulted from applying an axis relation and a node test to the current node. For example, [child::year > 1990] leaves in the result set only nodes that have a child node with name year with value greater than 1990. Usual comparison operators like =, !=, <, >, etc. can be used.

A boolean expression is allowed in a predicate with usual operators not, and, or. Complicated boolean expression can be expressed using parentheses. For example, the following expression selects CDs with Mike Oldfield that

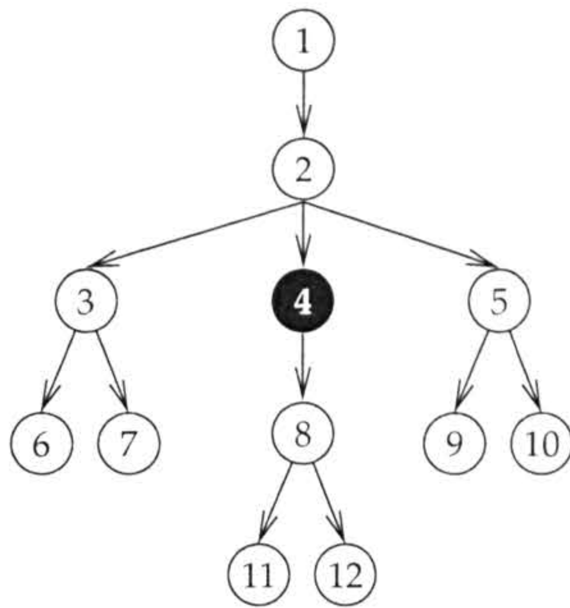

Figure 5: XPath axes sample XML data.
Current node is the black node with number 4.

were published after Tubular Bells.

```
/child::archive/child::cd
  [ child::author
      [child::last = "Oldfield" and child::first = "Mike"]
    and child::year > /descendant-or-self::cd
      [child::title = "Tubular Bells"]/year ]
```

Since such an expression like the one from the example above are rather unreadable, the following abbreviations are defined.

Abbreviation	Meaning
no axis	child::
@	attribute::
.	self::node()
..	parent::node()
//	/descendant-or-self::node()/

Using these abbreviations we can rewrite the example above to much readable form.

```
/archive/cd
    [ author[last = "Oldfield" and first = "Mike"]
      and year > //cd[title = "Tubular Bells"]/year ]
```

We already know that we can use comparisons in a predicate. Let's look closer on their semantics depending on the types of the operands. If we compare two atomic values like in 2 < 3, then the atomic values are compared. If we compare an atomic value and a nodeset like in author/last = "Oldfield" the situation is different. In such a case, the condition is true if there is at least one node in a nodeset that satisfies the condition. The third case is a comparison of two expressions both resulting in a nodeset like year > //cd[title="A"]/year.

Such condition is true if there exist two nodes, one from the nodeset returned by an expression year and the other from the nodeset returned by an expression //cd[title="A"]/year, that satisfy the condition. So, exact semantics of this expression is "there exists *at least one* node with name year in the current item with value greater than a value of *some* node year in a CD with title A".

It is possible to have zero or more predicates in a single step. If more predicates are specified after a node test then nodes in the resulting nodeset have to satisfy all of them. This means that more predicates in a single step have a meaning of a conjunction of their logical conditions.

Absolute and relative paths. So far, we explained location path subexpression. We also said that a location path traverses an XML document. What we did not mention is where the document traversal starts.

XPath identifies two types of location paths: absolute, and relative. Absolute location paths start with a slash /, which means that an XML document is to be traversed starting from its root node. On the other hand, relative paths do not start with a slash, which means they start from the currently evaluated item.

Variables. Though XPath 1.0 is not capable of defining variables, it is variables aware. This means that though variables may not be defined in XPath 1.0 expressions, they may come from outside and be referred to inside XPath 1.0 expressions. Variables can be used only in relative location paths and only in the first step. For example, suppose that $x is a variable defined outside to be a nodeset of CDs, then the following expression selects titles of CDs from this nodeset that contain some music.

```
$x[@type = "music"]/title
```

Types. XPath 1.0 uses simple typing. It distinguishes only three atomic types: boolean, number, and string. The only complex type is a nodeset. Since from XPath 1.0 perspective no type information is attached to an XML document, the type has to be declared explicitly in an XPath expression or is guessed. The guessing comes into place in an expression like year < 1990. Since year is compared to a number, the XPath processor first tries to convert the value of year to a number. If it succeeds it compares numbers otherwise both values are compared to strings. For this system to work correctly, XPath 1.0 defines a way to compute a string value for each node kind.

Arithmetics. It is possible to use all usual arithmetic operators on numbers: +, -, mod, div. The following will return an empty set, yet it is a correct XPath expression.

```
//cd[10 mod 2 > 2 + 3]
```

Functions. XPath 1.0 defines some functions that can be used in expressions. Functions can be divided into several groups depending on the type of the argument.

First group of functions represents functions that work with a nodeset like `count(`*ns*`)` that returns a number of nodes in the argument node set *ns*, or `position()` that returns the current node position in the current context with respect to the document order – so called context position, or `last()` that returns a number of nodes in the currently evaluated nodeset – so called context size, or `id(`*id*`)` that given an id *id* returns a node with that id. Functions from

this group are the most often used XPath functions. For example, the following expressions select "odd CDs", and "the last CD" from the archive, respectively.

```
//cd[position() mod 2 = 1]
//cd[position() = last()]
```

Second group of functions represents functions that operate on strings, like `concat(`s_1`,`s_2`)`, `starts-with(`s_1`,`s_2`)`, `contains(`s_1`,`s_2`)`. The most interesting function is `string(`o`)` that converts the given object o to a value of a string type.

Third group of functions represents functions that operate on numbers, like `ceiling(`n`)`, `floor(`n`)`, and `round(`n`)`. There is also a handy function `sum(`ns`)` that converts each node to a number and returns their sum. Finally, there is a function `number(`o`)` that tries to convert the given object o to a value of a number type.

Forth group of functions represents function that operate on booleans, like `not(`b`)`, `true()`, and `false()`. Of course, there is a function `boolean(`o`)` that tries to convert the given object o to a value of a boolean type.

Further, the expression evaluation context can know more functions then those defined in the specifications of the language and even such functions can be called. This is in particular a way to allow XPath 1.0 to be a building block for higher languages that are capable of defining user functions like XSLT and XQuery. Notice, that it is not possible to define a user function in XPath itself.

Alternatives operator. Several XPath 1.0 expressions can be joined together with an alternatives operator `|`. In such a case, the result is a union of nodesets returned by these expressions. For example, the following expression returns "all authors and artists" from the CD archive.

```
//author | //artist
```

4.1.2 XPath 2.0

The main purpose of the language is to address nodes in an XML document, which is the same as for the 1.0 version. XPath 2.0 has been designed as a basic language to be used in XSLT 2.0 and XQuery 1.0. It is a result of joint work of XML Query Working Group and XSL Working Group as a part of XML Activity of W3C.

Sequences. The primary purpose of XPath 2.0 is to address nodes in XML trees. The main difference between XPath 1.0 and XPath 2.0 is that an XPath 2.0 expression returns a sequence of items instead of a nodeset. Thus, the items in the returned sequence have now their order defined.

Further, its possible to define a new sequence in XPath 2.0. For example, both following expressions evaluate to sequence of 1, 2, 3 in that order.

```
(1, 2, 3)
(1 to 3)
```

XPath 2.0 also defines new operators on sequences: `union`, `intersect`, and `except` with a straightforward meaning. Notice that these operators are allowed only on sequences of items, so it is not allowed to compute an intersection of sequences of atomic values.

Types. XPath 2.0 operates on a logical structure of an XML document that is defined in XQuery 1.0 and XPath 2.0 Data Model. This model is strongly typed and is based on XML Schema type system.

XPath 2.0 is capable of inspecting types, and type casting. Available node tests have been greatly extended to support type information, e.g. the following expression selects "elements of type xs:date".

```
//element(xs:date)
```

It is also possible to test and cast to sequence types. For example, the following predicate is true if "all date elements in the inspected document are of type xs:date and there is at least one date element in the inspected document". The expression `element(xs:date)+` represents a sequence type of one or more elements of type xs:date.

```
[ //date instance of element(xs:date)+ ]
```

A node is of a specific type if its type is the same as the specific type or its type is derived by restriction or extension from the specific type.

Since we do not consider type information when comparing XQuery 1.0 and XSLT 2.0, all XPath 2.0 expressions that operate on types are omitted from the language syntax provided later.

Node tests. XPath 2.0 provides new node tests in accordance with the new data model. The `item()` node test matches all nodes or atomic values, the `document-node()` node test matches any document node, the `element()` node test matches any element, which is equal to using a wildcard in XPath 1.0.

If-then-else expression. Providing XPath 2.0 with the if-then-else expression is a minor enhancement of the language in terms of its expressive power, because the if-then-else expression can be often rewritten using alternatives operator and predicates. However, we admit that the if-then-else expression can improve readability. Both following expressions return "authors, if CD contains speech, otherwise it returns artists", thus the result is a mixture of authors and artists. The first XPath expression uses the if-then-else expression, the second uses alternative operator and predicates.

```
//cd/if (@type = "speech")
       then author
       else artist
//cd[@type = "speech"]/author | //cd[@type != "speech"]/artist
```

For-in-return expression. Unlike the if-then-else expression, the for-in-return expression brings more power to the language. This expression allows to define variables in XPath 2.0, which was not possible in the 1.0 version of the language. The for-in-return expression has the following syntax.

```
for variable in expression return expression
```

The for-in part defines the variable bindings, a variable is one by one bound to items from the sequence returned by the in expression. The result of the for-in-return expression is then the union of sequences that we get by evaluating the return expression for each variable binding. Let's make this explanation clearer with an example. The following expression returns "titles of CDs where 1, 2, or 3 artists are performing, ordered by the number of performing artists".

```
for $x in (1 to 3)
  return //cd[count(artist) = $x]/title
```

Notice, that the sequence returned by the in expression dictates the order of variable bindings and that the return expression is evaluated for each variable binding in that order. The result sequence preserves that order as well, because the items from the first evaluation of the return expression are put to the result sequence first, then items from the second evaluation of the return expression are appended to the result sequence, etc. This demonstrates the order preserving principle of all XPath 2.0 expressions.

Quantified expressions. The boolean XPath 1.0 expression have been extended with quantified expressions. In XPath 2.0, its possible to use both quantifiers: existential, and general. The following examples return "CD titles, where all artists and authors have first name Karel", and "CD titles, where at least one artist or author has first name Karel".

```
//cd[every $a in author|artist satisfies $a/first="Karel"]/title
//cd[some $a in author|artist satisfies $a/first="Karel"]/title
```

Later, in section 4.5, we prove that both types of quantified expressions do not extended the expressive power of XPath 2.0.

Comparisons. The XPath 1.0 comparison operators =, !=, <, ... and their semantics are retained in XPath 2.0, but few more operators have been added: value comparison and node comparison operators.

Newly defined value comparison operators are eq, ne, lt, le, gt, ge, which stand for equal to, not equal to, less than, less than or equal to, greater than, and greater than or equal to, respectively. These operators are applicable only to atomic values on both sides of the comparison, otherwise an error is raised. Thus, the first from the following expression asking for "titles of CDs where first name of some artist is Mike" is correct as there is always exactly one first name for each artist. However, the second expression trying to retrieve "CD titles with Mike Oldfield" is not, since there are CDs with multiple artists, for which an error will be raised.

```
//cd[artist/first eq "Mike"]/title
//cd[artist eq "Mike Oldfield"]/title
```

The node comparison operators are is, <<, and >>. The is operator tests whether two given nodes represent the same node. Such a test was missing in XPath 1.0. Operators << and >> compare the document order of two nodes. The following expression returns "titles of CDs that precede all speech CDs".

```
//cd[ every $cd in //cd[@type = "speech"]
        satisfies . << $cd ]/title
```

Functions and operators. The set of functions in XPath 2.0 has been extended tremendously. XPath now contains over a hundred functions and almost seventy operators. Sure, we do not introduce all of them, but only mention areas covered by the new function set.

There are functions for strings that now support regular expression matching, replacement, and tokenization. Functions on numbers are now defined on more fine-grained types making a difference between a floating point numbers

and integers. There are also many new functions that operate on durations and time and operators to compare values of these types.

Also functions that operate on sequences instead of nodesets have been redefined and new have been added. They now include functions to modify sequences by insertion, concatenation, or selecting a subsequence, functions to test the cardinality of a sequence, and aggregate functions to compute a sum, maximum, minimum, or an average value from a sequence of values.

As there is such a large number of them and since they are shared by XPath 2.0 and XQuery 1.0, functions and operators were dedicated a separate W3C specification called XQuery 1.0 and XPath 2.0 Functions and Operators. To distinguish functions from operators, functions use namespace prefix fn:, while operators use namespace prefix op:. Thus, to express "number of CDs" we write the following.

```
fn:count(//cd)
```

Specification status. As of writing this thesis, XPath 2.0 is still a W3C working draft. This means that changes may appear later.

4.2 XPath Syntax

In this section, we provide the syntax of XPath 2.0. We have chosen to provide only the core of the language as described in XQuery 1.0 and XPath 2.0 Formal Semantics specification, as it has the same expressive power as the full language.

The syntax is divided into two figures, Figure 6 and 7, where both are presented in the Extended Backus-Naur Form. Nonterminals start with a capital letter instead of enclosing them with angles <>. Keywords are enclosed in quotes "". Part of a grammar production can be enclosed in simple parenthesis (), usually to express zero or one appearance with a question mark ?, or zero or more appearances with an asterisk *.

Figure 6 is the main syntax of the language. Figure 7 depicts axes and kind tests.

Core vs. Full. Each expression in full XPath can be rewritten using the core XPath. The formal semantics specification defines how an expression in full XPath is converted to an expression in core XPath. This process is called *normalization*.

The major distinction between the XPath 2.0 Core and full XPath 2.0 is in for-in-return and quantified expressions. While the full version of the language allows a list of variable bindings in a for-in-return expression, the core language allows only a single variable binding. Such an expression is normalized by producing nested for-in-return expression for each variable binding. For example, the following expressions are equivalent. The former is in full XPath and the latter in core XPath.

```
for $x in e1, $y in e2 return e3
```

```
for $x in e1 return
  for $y in e2 return
    e3
```

```
XPath              ::= Expr
Expr               ::= ExprSingle ( "," Expr )?
ExprSingle         ::= FLWORExpr
                       | QuantifiedExpr
                       | IfExpr
                       | OrExpr
FLWORExpr          ::= ForExpr "return" ExprSingle
ForExpr            ::= "for $"VarName "in" ExprSingle
QuantifiedExpr     ::= ( "some" | "every" ) "$"VarName
                       "in" Expr "satisfies" ExprSingle
IfExpr             ::= "if" "(" Expr ")" "then" ExprSingle
                                     "else" ExprSingle
OrExpr             ::= AndExpr ( "or" AndExpr )?
AndExpr            ::= ComparisonExpr ( "and" ComparisonExpr )?
ComparisonExpr     ::= RangeExpr ( ComparisonOper RangeExpr )?
ComparisonOper     ::= ValueComp | GeneralComp | NodeComp
RangeExpr          ::= ArithmeticExpr ( "to" ArithmeticExpr )?
ArithmeticExpr     ::= SequenceExpr ( ArithmeticOper SequenceExpr )*
ArithmeticOper     ::= "+" | "-" | "*" | "div" | "idiv" | "mod"
SequenceExpr       ::= UnaryExpr ( SequenceOper UnaryExpr )*
SequenceOper       ::= "union" | "intersect" | "except"
UnaryExpr          ::= ("-" | "+")* PathExpr
GeneralComp        ::= "=" | "!=" | "<" | "<=" | ">" | ">="
ValueComp          ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
NodeComp           ::= "is" | "<<" | ">>"
PathExpr           ::= "/" ( RelativePathExpr )?
                       | "//" RelativePathExpr
                       | RelativePathExpr
RelativePathExpr   ::= StepExpr
                       | StepExpr ( "/" | "//" ) RelativePathExpr
StepExpr           ::= ForwardAxis NodeTest ( Predicates )?
                       | ReverseAxis NodeTest ( Predicates )?
                       | PrimaryExpr ( Predicates )?
PrimaryExpr[1]     ::= Literal | "$" VarName
                       | ParenthesizedExpr | FunctionCall
Predicates         ::= "[" Expr "]" ( Predicates )?
NodeTest           ::= KindTest | NameTest
NameTest           ::= QName | Wildcard
Wildcard           ::= "*" | NCName ":" "*" | "*" ":" NCName
Literal            ::= NumericLiteral | StringLiteral
NumericLiteral     ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
ParenthesizedExpr  ::= "()" | "(" Expr ")"
FunctionCall       ::= QName "(" ( ExprSingle ("," ExprSingle)* )? ")"
```

Figure 6: XPath 2.0 Syntax. Part 1.

```
ForwardAxis       ::= "child" "::"
                    | "self" "::"
                    | "attribute" "::"
                    | "descendant" "::"
                    | "descendant-or-self" "::"
                    | "following-sibling" "::"
                    | "following" "::"
ReverseAxis       ::= "parent" "::"
                    | "ancestor" "::"
                    | "ancestor-or-self" "::"
                    | "preceding-sibling" "::"
                    | "preceding" "::"
KindTest          ::= DocumentTest
                    | ElementTest
                    | AttributeTest
                    | SchemaElementTest
                    | SchemaAttributeTest
                    | PITest
                    | CommentTest
                    | TextTest
                    | AnyKindTest
AnyKindTest       ::= "node" "(" ")"
DocumentTest      ::= "document-node" "(" ( ElementTest )? ")"
TextTest          ::= "text" "(" ")"
CommentTest       ::= "comment" "(" ")"
PITest            ::= "processing-instruction" "(" ( NCName )? ")"
AttributeTest     ::= "attribute" "(" ( AttributeName | "*" )? ")"
ElementTest       ::= "element" "(" ( ElementName | "*" )? ")"
AttributeName     ::= QName
ElementName       ::= QName
```

Figure 7: XPath 2.0 Syntax. Part 2.

Similarly, we can use multiple variable bindings in quantified expressions, which is again normalized using nested quantified expressions as in the following example.

```
some $x in e1, $y in e2 satisfies e3
```

```
some $x in e1 satisfies
  some $y in e2 satisfies e3
```

Changes. The provided syntax rules are almost exact copy of the W3C's specification grammar. Here we identify the minor differences we did.

First, we striped out everything type-related, because we do not consider the type information in the data model. Thus, we removed expressions that inspect or cast types. Also sequence types constructors were removed.

Second, we removed all abbreviations. These include alternatives operator |, and abbreviations mentioned in a table on page 21.

Third, the order in which the grammar productions appear imposes operator precedence in W3C's specification. As we do not bother so much, we joined some rules together, like ArithmeticExpr, and SequenceExpr.

Further, we do not provide grammar productions for all kinds of literals like NumericLiteral, QName, NCName, since this is not important to evaluate the expressive power of the language. It is sufficient to know that all unspecified nonterminals represent literals. Among the others, we should emphasize QName and NCName. QName represents a qualified name that comprises a namespace prefix, double-dot, and name, e.g. xsl:value-of. NCName on the other hand represents something like an identifier. It is a sequence of letters and/or numbers, dots, and underscores, with a letter at the first position.

Finally, we replaced several productions that use an asterisk * to equivalent ones with question mark only. This is to simplify specification and improve readability of the formal semantics of the language.

4.3 XPath Formal Semantics

In this section, we provide formal semantics of XPath 2.0. The formal semantics used throughout this thesis is denotational semantics. It was especially work of Wadler on semantics of XPath [55] that we were inspired by.

We do *not* provide semantics for all syntactic constructs that are listed in XPath syntax. Rather, the semantics is defined only for such constructs that we use later in section on sorting in XPath. The formal semantics is used there to prove that presented XPath expressions do exactly what is desired. To provide full semantics of XPath, it would be necessary to define semantics of path expressions with steps, axes, and predicates, further, arithmetic expressions, and operations on sequences.

The denotational semantics of XPath that we present here, is defined with three semantic functions: $\mathcal{E}$, $\mathcal{W}$, and $\mathcal{A}$. The first function $\mathcal{E}$ evaluates an expression in a given context, the second function $\mathcal{W}$ returns a boolean value of an expression, and the third function $\mathcal{A}$ is a helper function that evaluates a list of argument expressions in a function call.

$\mathcal{E}\llbracket\text{ExprSingle, Expr}\rrbracket C$	$\mathcal{E}\llbracket\text{ExprSingle}\rrbracket C \circ \mathcal{E}\llbracket\text{Expr}\rrbracket C$
$\mathcal{E}\llbracket\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2\rrbracket C$	$\mathrm{S} = \mathcal{E}\llbracket e_1\rrbracket C$ $S = \{\ x_2 \mid\ x_1 \in \mathcal{I}_\mathrm{S},\ C_1 = C(v \rightarrow x_1),\ x_2 \in \mathcal{I}_{\mathcal{E}\llbracket e_2\rrbracket C_1}\ \}$ $\prec_S = \{\ \langle x, y\rangle \mid\ x_1, y_1 \in \mathcal{I}_\mathrm{S},\ \mathrm{S}_1 = \mathcal{E}\llbracket e_2\rrbracket C(v \rightarrow x_1),\ \mathrm{S}_2 = \mathcal{E}\llbracket e_2\rrbracket C(v \rightarrow y_1),\ x \in \mathcal{I}_{\mathrm{S}_1}, y \in \mathcal{I}_{\mathrm{S}_2},\ x_1 \sqsubset_\mathrm{S} y_1 \text{ or } x \sqsubset_{\mathrm{S}_1} y\ \}$
$\mathcal{E}\llbracket\texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f\rrbracket C$	$\begin{cases} \mathcal{E}\llbracket e_t\rrbracket C & \text{if } \mathcal{W}\llbracket e\rrbracket C \\ \mathcal{E}\llbracket e_f\rrbracket C & \text{otherwise} \end{cases}$
$\mathcal{E}\llbracket\texttt{some}\ \$v\ \texttt{in}\ e_1\ \texttt{satisfies}\ e_2\rrbracket C$	$S = \mathcal{W}\llbracket\texttt{some}\ \$v\ \texttt{in}\ e_1\ \texttt{satisfies}\ e_2\rrbracket C$ $\prec_S = \emptyset$
$\mathcal{E}\llbracket\texttt{every}\ \$v\ \texttt{in}\ e_1\ \texttt{satisfies}\ e_2\rrbracket C$	$S = \mathcal{W}\llbracket\texttt{every}\ \$v\ \texttt{in}\ e_1\ \texttt{satisfies}\ e_2\rrbracket C$ $\prec_S = \emptyset$
$\mathcal{E}\llbracket fn(\text{ArgumentList})\rrbracket C$	$\mathcal{E}\llbracket e\rrbracket nofocus(C(\theta_{fn} \rightarrow t))$ for $F(fn) = \langle\theta_{fn}, e\rangle,\ t = \mathcal{A}\llbracket\text{ArgumentList}\rrbracket C$
$\mathcal{E}\llbracket\$v\rrbracket C$	$\rho(v)$

Figure 8: XPath Semantics for context $C = \langle\rho, F, item, pos, last\rangle$.

4.3.1 Expression to Sequence

We start with the main semantic function $\mathcal{E}$. It takes an XPath expression and evaluates it in the given context. The result is a sequence of items $\langle S, \prec_S\rangle$.

$$\mathcal{E} \ :\ Expression \rightarrow Context \rightarrow Seq(Item)$$

Function $\mathcal{E}$ is defined in Figure 8. As it can be hard to read, we describe each definition in words.

Expression, expression. The first line in the figure defines the semantics for a comma-separated list of expressions. It says that each expression in the list is evaluated separately and the resulting sequences are concatenated to form the result.

For-in-return. The second line defines semantics of a for-in-return expression `for` $\$v$ `in` e_1 `return` e_2. With S we denote the sequence that results from evaluating expression e_1. The set of items S in the result of the for-in-return expression are such items that are in the result of evaluating expression e_2 in a context with variable v bound to some item in S. The order of items in S preserves the order of items bound to v and within the same variable binding the order of items that results from evaluation of e_2 is preserved. Simply said, result of a for-in-return expression is a concatenation of sequences that come from evaluation of the return expression with variable bound to items in Sin their order.

Thus, e.g. expression `for $i in (1,3) return ($i,$i+1)` returns a

sequence (1, 2, 3, 4), because first 1 is bound to variable $i, and the return expression results in a sequence (1, 2). Then, 3 is bound to $i to return a sequence (3, 4). Thus, the set of items is {1, 2, 3, 4}. The order tells that {1, 2} have to precede {3, 4} in the result and within these 1 has to precede 2 and 3 has to precede 4, because that is the order in which they were returned by the return expression.

If-then-else. The third line defines semantics of if-then-else expression `if` e `then` e_t `else` e_f. It evaluates to e_t if the boolean value of e is true, otherwise it evaluates to e_f.

Quantified expressions. The next two lines define semantics of generally and existentially quantified expressions. Both return the boolean value of the same expression. The order is empty, because nothing else then an atomic value true or false can appear in the result.

Function call. The last but one line defines semantics of a function call. The result of a function call is a result of an expression that defines the body of a function in a context with an empty focus and variables in an argument list bound to values that we get by evaluating argument expressions.

Notice, that it is not possible to define functions in XPath, which is possible only in XQuery and XSLT. Though, the semantics of a function call is defined here, as it is the same for XQuery, XSLT, and also for the built-in functions of XPath.

Variable reference. Finally, we define that a reference to a variable is evaluated to a value that is stored for that variable name in the current context.

4.3.2 Expression to Boolean Value

In Figure 9 we define semantic function $\mathcal{W}$ that evaluates an expression in a given context either to true or false, computing so an effective boolean value as defined in specification.

$$\mathcal{W} \; : \; Expression \rightarrow Context \rightarrow Boolean$$

The first line defines the effective boolean value of an expression in a given context. We can see, that the effective boolean value of an expression is false if an expression evaluates to an empty sequence, empty string, zero, or false. Otherwise the effective boolean value of an expression is true.

The second line defines semantics of a built-in boolean function fn:not() that realizes negation.

The next two lines define semantics of boolean operators and, and or.

The last but one line define semantics for existentially quantified expression `some` $\$v$ `in` e_1 `satisfies` e_2. With S we denote a sequence of items that results from evaluation of e_1. The expression is true if there exists an item x from sequence S, such that expression e_2 is true in a context with variable v bound to x.

The last line defines semantics of a generally quantified expression. The semantics distinguishes from semantics of an existentially quantified expression only in quantifier. Thus, the expression is true if for all items x from sequence Sexpression e_2 is true in a context with variable v bound to x.

$\mathcal{W}[\![e]\!]C$	$\begin{cases} \textit{false} & \text{if } \mathcal{E}[\![e]\!]C \text{ is an empty sequence, empty string, 0, or } \textit{false} \\ \textit{true} & \text{otherwise} \end{cases}$
$\mathcal{W}[\![\texttt{fn:not}(e)]\!]C$	$\neg\mathcal{W}[\![e]\!]C$
$\mathcal{W}[\![e_1 \texttt{ or } e_2]\!]C$	$\begin{cases} \textit{true} & \text{if } \mathcal{W}[\![e_1]\!]C \text{ or } \mathcal{W}[\![e_2]\!]C \\ \textit{false} & \text{otherwise} \end{cases}$
$\mathcal{W}[\![e_1 \texttt{ and } e_2]\!]C$	$\begin{cases} \textit{true} & \text{if } \mathcal{W}[\![e_1]\!]C \text{ and } \mathcal{W}[\![e_2]\!]C \\ \textit{false} & \text{otherwise} \end{cases}$
$\mathcal{W}[\![\texttt{some } \$v \texttt{ in } e_1 \texttt{ satisfies } e_2]\!]C$	$\begin{cases} \textit{true} & \text{if for S} = \mathcal{E}[\![e_1]\!]C \\ & \exists x \in \mathcal{I}_S : \mathcal{W}[\![e_2]\!]C(v \to x) \\ \textit{false} & \text{otherwise} \end{cases}$
$\mathcal{W}[\![\texttt{every } \$v \texttt{ in } e_1 \texttt{ satisfies } e_2]\!]C$	$\begin{cases} \textit{true} & \text{if for S} = \mathcal{E}[\![e_1]\!]C \\ & \forall x \in \mathcal{I}_S : \mathcal{W}[\![e_2]\!]C(v \to x) \\ \textit{false} & \text{otherwise} \end{cases}$

Figure 9: Boolean value of an XPath expression.

$\mathcal{A}[\![\text{ExprSingle, ArgumentList}]\!]C$	$\{ \langle x, \vec{y} \rangle \mid x = \mathcal{E}[\![\text{ExprSingle}]\!]C,\ \langle \vec{y} \rangle = \mathcal{A}[\![\text{ArgumentList}]\!]C \}$

Figure 10: Semantics of an argument list of an XPath function.

4.3.3 Function Call Arguments

A helper semantic function $\mathcal{A}$ is used to evaluate a list of argument expressions to a tuple of their respective values. This function is defined in Figure 10.

$$\mathcal{A} \;:\; \mathit{ArgumentList} \to \mathit{Tuple}$$

An argument list is a comma-separated list of expression, or to be precise of expression single. The only line we provide in Figure 10 explains the inductive step. It takes a head expression from an argument list that is separated by a comma from the rest of an argument list. It is evaluated to a tuple, where the first position contains a sequence returned by evaluating the first argument. The rest is built recursively, thus there is a sequence returned by evaluating the second argument at the second position in the resulting tuple.

Notice that the order in which the arguments are evaluated is not specified and that all arguments are evaluated in the same context in which the function is called.

4.4 Sorting in XPath

In this section, we introduce a surprising fact that XPath 2.0 is capable of sorting sequences. This has several important consequences, among others a consequence on the ability to formally express ordering semantics of XQuery,

which was supposed unfeasible without introducing tuples in XQuery 1.0 and XPath 2.0 Formal Semantics specification of W3C. More on this topic in 5.4.

These are the main features of our approach to sorting in XPath.

- **Generality.** Any sequence of items can be sorted.
- **Complex ordering.** The ordering expression can be any XPath 2.0 expression, thus orderings like "order by last name and within equal last names order by first name" are possible.
- **Inefficiency.** By extending XPath with for-return expressions, and range expressions of form (i to j), where i and j are numbers expressible with an expression, the language became capable of sorting. But an extra cost has to be paid: it is awfully inefficient, $O(n^3)$ where n is a number of items in the sorted sequence. A question arises, if some form of an order by expression should be added to the language to achieve better performance of sorting.

4.4.1 By Example

Here, we present a general XPath expression that sorts items in a sequence S. The whole expression is for clarity divided into three separate functions my:precedes(), my:count-less-than(), and my:sort() that can be assembled together to form a single XPath expression. The three functions have the following meaning.

First, function my:precedes($x, $y) represents a less-than relation on items from S. It returns true iff $x is less than $y, whatever to be less than means. In the following, we simply use the usual comparison operator <, but more complex expressions can be provided as shown later.

```
my:precedes($x, $y) {
  return ($x < $y)
}
```

Second, function my:count-less-than($x, S) returns a number of nodes in a sequence S that are less than $x by means of the less-than relation defined by the my:precedes() function. The for loop iterates over S creating so a sequence of items from S that are less than $x. Function fn:count() is applied to this sequence to count a number of items that are less than $x in S.

```
my:count-less-than($x,S) {
  fn:count(
    for $y in S
    return
      if (my:precedes($y, $x))
      then $y
      else ()
  )
}
```

Finally, function my:sort(S) returns the sorted sequence.

```
my:sort(S) {
  for $i in (0 to fn:count(S) - 1)
  return
    for $x in S
    return
      if ($i = my:count-less-than($x, S))
      then $x
      else ()
}
```

Let's look how it works. We examine the outer loop first. In its first iteration, it returns each item x in S for which there exists *no* item y in S such that y is less than x – the minimum of S. In its second iteration, it returns each item x in S for which there is *exactly one* node y in S that is less than x. In its third iteration, exactly two, etc. In its final iteration, it returns each item x in S for which all other items in S are less than x – the maximum of S.

If there is a subset of nodes that are equal in S by means of the less-than relation then they will all be returned in a single iteration of the outer loop. In such a case some iterations return nothing. The inner loop guarantees that the initial sequence order of the equal nodes is preserved in the resulting sequence.

The following XPath 2.0 expression is an example of an assembled expression that returns "sorted CD titles".

```
for $i in (0 to count(//cd/title) - 1)
return
  for $x in //cd/title
  return
    if ($i = count(
          for $y in //cd/title
          return
            if ($y < $x)
            then $y
            else ()
        )
    )
    then $x
    else ()
```

Earlier we mentioned that it is possible to express more complex sort order by modifying the my:precedes() function. Since this is the only place where the semantics of the less-than relation is defined, it is also the only place were it needs to be changed.

For example, we want to order a sequence by authors first by their last and second by their first name. We change the my:precedes() function to the following.

```
my:precedes($x, $y) {
  ($x/last < $y/last) or
  ($x/last = $y/last and $x/first < $y/first)
}
```

One more example. The following XPath 2.0 expression returns "CD titles ordered by CD authors first by their last name and second by their first name".

Notice, that this is a special case of the above that sorts according to information that lies outside the subtree of the sorted nodes.

```
for $i in (0 to count(//cd/title) - 1)
return
  for $x in //cd/title
  return
    if ($i = fn:count(
          for $y in //cd/title
          return
            if ( ( $y/parent::cd/author/last <
                   $x/parent::cd/author/last )
                 or
                 ( ( $y/parent::cd/author/last =
                     $x/parent::cd/author/last )
                   and
                   ( $y/parent::cd/author/first <
                     $x/parent::cd/author/first ) )
            then $y
            else ()
        )
    )
    then $x
    else ()
```

And one more notice. Usually, the if-then-else expression in my:count-less-than() function can be replaced by a step expression with a predicate. For example, for the simple case:

```
$y[self::* < $x]
```

And for the not-so-simple case:

```
$y[(last = $x/last) or
  (last = $x/last and $y/first < $x/first)]
```

4.4.2 Formally

In this section, we formally define expression *Sort* and prove that it sorts each given sequence. Notice, that we define sorting with respect to a given partial order, which is handy later as XQuery defines its ordering semantics on the partial order basis.

Throughout this section, we consider only such partial order relations whose characteristic functions are expressible with some XPath expression.

First, we define equivalence eq_R on items in partial order R.

Definition 4.1 *Let R be a partial order. With eq_R we denote a set of all R-equal items.*

$$x \ eq_R \ y \text{ iff } R(x,y)\&R(y,x)$$

Next, we define the less-than relation with respect to a given partial order R by removing equivalence from R. This is needed, since we do not want to count items that are less than or equal to the current item, which is the meaning of R, but rather we want to count items that are sharply less than the current item.

Definition 4.2 *Let R be a partial order. With lt_R we denote a total order $R \setminus eq_R$.*

We should note, that if a characteristic function of R is expressible with an XPath expression then even eq_R and lt_R are expressible with an XPath expression using boolean operators. We should also note, that the my:precedes() function defined in the previous section is an example of a characteristic function of lt_R relation.

Lemma 4.1 *Let S be a set, and $R \subseteq S \times S$ be a partial order on S. Then for each $x, y \in S$ either $x \; eq_R \; y$, or $x \; lt_R \; y$, or $y \; lt_R \; x$.*

Proof. This comes from the relations between partial order R, equivalence eq_R, and total order lt_R, namely $R = eq_R \cup lt_R$ and $eq_R \cap lt_R = \emptyset$. □

The following formally defines the natural notion of number of items in a sequence that are less than the given item. The *CLT* stands for count less than.

Definition 4.3 *For S a set, and $R \subseteq S \times S$ a partial order on S we define function $CLT_R(x, S)$.*

$$CLT_R(x, S) =_{\text{def}} |\{ \, y \in S \mid y \; lt_R \; x \, \}|$$

Next, we define the *CountLessThan* XPath expression with respect to the given partial order.

Definition 4.4 *Let S be a set, and $R \subseteq S \times S$ be a partial order on S. With expression $CountLessThan_R(x, S)$ we denote the following XPath expression.*

```
fn:count(
  for $y in S
  return
    if ($y ltR x)
    then $y
    else ()
)
```

The following lemma says that the $CountLessThan_R(x, S)$ expression really counts items from S that are sharply less than x in terms of R. We provide the proof in Figure 11 on page 39, where the semantics of the $CountLessThan_R$ expression is derived with our semantics rules.

Lemma 4.2 *Let S be a set, and $R \subseteq S \times S$ be a partial order on S. Then the result of $CountLessThan_R(x, S)$ is equal to $CLT_R(x, S)$, number of items in S that are less than x in terms of lt_R.*

The following lemma provides bounds to $CLT_R(x, S)$ and so also to results of $CountLessThan_R(x, S)$.

Lemma 4.3 *Let S be a nonempty set, and $R \subseteq S \times S$ be a partial order on S. Then for each $x \in S$ the following holds.*

$$0 \leq CLT_R(x, S) \leq |S| - 1$$

Proof. The lower bound is equal to zero, as the cardinality of a set cannot be less than zero.

The upper bound cannot be more than $|S|$, which is guaranteed by the first condition $y \in S$ in the definition of CLT_R. Moreover, it cannot be more than $|S| - 1$, which is guaranteed by the second condition $x\ lt_R\ y$, as lt_R is irreflexive, thus for each $x \in S$ at least x is not present in $\{\ y \in S \mid y\ lt_R\ x\ \}$. □

Notice, that the lower bound is reached for *all* minimal items in S, i.e. items for which there is no less-than item in S, and that there can be more of them equal to each other in terms of eq_R. Conversely, the upper bound is reached only for the maximum item in S, which has not to exist if there are multiple maximal items. If there are multiple maximal items then they are again equal to each other in terms of eq_R.

The following lemma claims that equal items have equal counts of less-than items.

Lemma 4.4 *Let S be a set, and $R \subseteq S \times S$ be a partial order on S. Then for each $x, y \in S$ such that $x\ eq_R\ y$ the following holds.*

$$CLT_R(x, S) = CLT_R(y, S)$$

Proof. This lemma can be rewritten as follows.

$$x\ eq_R\ y \text{ implies } \forall z \in S : z\ lt_R\ x \text{ implies } z\ lt_R\ y$$

For contradiction, suppose that the above is not true, so suppose that such $z \in S$ exists, for which $z\ lt_R\ x$ and not $z\ lt_R\ y$. Combining this fact and Lemma 4.1, we get that either i) $y\ lt_R\ z$, or ii) $z\ eq_R\ y$ holds.

That the following inequalities are contradictions comes from the fact that lt_R is a total order, and eq_R is an equivalence.

Ad i) $z\ lt_R\ x\ eq_R\ y\ lt_R\ z$.

Ad ii) $z\ eq_R\ x\ eq_R\ y\ lt_R\ z$. □

Next, let's define for a sequence the property of being sorted.

Definition 4.5 *Let $\langle S, \prec_S \rangle$ be a sequence, and $R \subseteq S \times S$ be a partial order on S. We say that a sequence $\langle S', \prec_{S'} \rangle$* is a sequence $\langle S, \prec_S \rangle$ sorted with R *iff for each $x, y \in S$ the following conditions hold.*

i) $S' = S$

ii) $x\ lt_R\ y$ implies $x \prec_{S'} y$

iii) $x\ eq_R\ y$ implies $x \prec_S y$ implies $x \prec_{S'} y$

We refer to a sequence $\langle S, \prec_S \rangle$ as the initial sequence and a sequence $\langle S', \prec_{S'} \rangle$ as the sorted sequence.

The first condition tells that if x is less than y in terms of partial order R, then x has to precede y also in the sorted sequence. The second condition tells that if two items are equal in R then the order of items x, and y in the initial sequence has to be preserved in the sorted sequence.

Notice, that we define a sequence to be sorted with respect to an initial sequence. Thus, a sequence has to be created prior to have been sorted. This

definition of a sequence being sorted is based on XQuery's *stable sorting* that we explain later in the chapter about XQuery.

Next, we define the XPath expression that we claim to sort a given sequence.

Definition 4.6 *Let $\langle S, \prec_S \rangle$ be a sequence, and $R \subseteq S \times S$ be a partial order on S. With expression $Sort_R(\langle S, \prec_S \rangle)$ we denote the following XPath expression.*

```
if (fn:empty(S))
then ()
else
  for $i in (0 to fn:count(S) - 1)
  return
      for $x in ⟨S,≺S⟩
      return
          if ($i = CountLessThanR($x, S))
          then $x
          else ()
```

Finally, we prove that the just defined XPath expression *Sort*() really sorts a given sequence.

Theorem 4.1 *Let $\langle S, \prec_S \rangle$ be a sequence, and $R \subseteq S \times S$ be a partial order on S. Let $\langle S', \prec_{S'} \rangle = Sort_R(\langle S, \prec_S \rangle)$ be a sequence. Then $\langle S', \prec_{S'} \rangle$ is a sequence $\langle S, \prec_S \rangle$ sorted with R.*

Proof. If the S is empty then the result of the above expression is an empty sequence, as guaranteed by the outermost if expression. In such a case, the resulting sequence is trivially sorted. In the following, we assume that S is non-empty.

First, we prove that $S' = S$, which is the first condition of the Definition 4.5 of a sorted sequence. Lemma 4.3 provides bounds to $CountLessThan_R$ expression. Since the outer loop of $Sort_R$ iterates over all values within these bounds, then for each $x \in S$ there exists such i that $i = CLT_R(x, S)$. For such i, item x is returned, thus $S \subseteq S'$. Since CLT_R is a function, there exists exactly one such i for each $x \in S$, therefore $S' = S$. Figure 12 on page 40 provides evidence using formal semantics derivation.

Second, we prove that $x\ lt_R\ y$ implies $x \prec_{S'} y$. If $x\ lt_R\ y$ holds then $CLT_R(x, S) < CLT_R(y, S)$, which is obvious from Definition 4.3 of CLT_R. Since the outer loop of $Sort_R()$ iterates over i in a growing manner, x is returned sooner than y, which defines the order of x, and y in the resulting sequence. Thus we have $x \prec_{S'} y$.

Similarly, we prove the last condition of the definition of a sorted sequence that $x\ eq_R\ y$ implies $x \prec_S y$ implies $x \prec_{S'} y$. This comes directly from Lemma 4.4, which guarantees that $CLT_R(x, S) = CLT_R(y, S)$ for $x\ eq_R\ y$. Thus x, and y are returned in the same iteration of the outer loop of the $Sort_R$ expression. Since the inner loop iterates over S in the order defined with $\prec_S$, x is bound to `$x` prior to y. Therefore x is returned sooner than y. Thus we have $x \prec_{S'} y$.

In Figure 13 on page 41, the resulting order $\prec_{S'}$ is derived from the $Sort_R$ expression with our semantics rules. We can use the result to check that the properties of $\prec_{S'}$ required for the sequence to be sorted with R are satisfied.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{I}_{\mathcal{E}}[\![\texttt{fn:count(for \$y in } S \texttt{ return if (\$y } lt_R\, x \texttt{ then \$y else ()))}]\!]C =} \\
= & \left|\mathcal{I}_{\mathcal{E}}[\![\texttt{for \$y in } S \texttt{ return if (\$y } lt_R\, x \texttt{ then \$y else ()}]\!]C\right| & (1) \\
= & |\{\, x_2 \mid \; x_1 \in S & (2) \\
 & \qquad C_1 = C(\$\mathtt{y} \to x_1) & \\
 & \qquad x_2 \in \mathcal{I}_{\mathcal{E}}[\![\texttt{if (\$y } lt_R\, x \texttt{ then \$y else ()}]\!]C_1 \,\}| & \\
= & |\{\, x_2 \mid \; x_1 \in S & (3) \\
 & \qquad C_1 = C(\$\mathtt{y} \to x_1) & \\
 & \qquad x_2 \in \left\{ \begin{array}{ll} \mathcal{I}_{\mathcal{E}}[\![\$\mathtt{y}]\!]C_1 & \text{if } \mathcal{W}[\![\$\mathtt{y}\ lt_R\ x]\!]C_1 \\ \emptyset & \text{otherwise} \end{array} \right. & \\
 & \qquad \}| & \\
= & |\{\, x_2 \mid \; x_1 \in S & (4) \\
 & \qquad C_1 = C(\$\mathtt{y} \to x_1) & \\
 & \qquad x_2 \in \mathcal{I}_{\mathcal{E}}[\![\$\mathtt{y}]\!]C_1 & \\
 & \qquad \mathcal{W}[\![\$\mathtt{y}\ lt_R\ x]\!]C_1 \,\}| & \\
= & |\{\, x_1 \mid \; x_1 \in S & (5) \\
 & \qquad C_1 = C(\$\mathtt{y} \to x_1) & \\
 & \qquad \mathcal{W}[\![\$\mathtt{y}\ lt_R\ x]\!]C_1 \,\}| & \\
= & |\{\, x_1 \in S \mid x_1\ lt_R\ x \,\}| & (6) \\
= & CLT_R(x,S) & (7)
\end{array}
$$

Figure 11: Proof of Lemma 4.2.

This is in perfect concert with the two paragraphs above. Notice, that an order $\sqsubset_{\mathcal{E}[\![\texttt{if...then \$x else ()}]\!]C(\$\mathtt{i}\to x_1)(\$\mathtt{x}\to x_2)}$ in equation (4) evaluates to an empty set. This is because the if expression returns an atomic value or an empty sequence, for which no order is defined.

$x\ lt_R\ y \to CLT_R(x,S) < CLT_R(y,S) \to x_1 < y_1 \to x \prec_{S'} y$

$x\ eq_R\ y \to CLT_R(x,S) = CLT_R(y,S) \to x \in \mathcal{I}_{S'_1},\ y \in \mathcal{I}_{S'_2},\ x_2 \prec_S y_2 \to x \prec_{S'} y$

□

4.4.3 Conclusion

In this section, we formally proved that XPath is capable of sorting arbitrary sequences. The only constraint we raise is that the sorting relation is a partial order expressible in XPath.

The fact, that XPath 2.0 can sort has consequences on XQuery formal semantics specification, which is studied in detail in section 5.4.

4.5 Quantified Expressions

In this section, we show that quantified expressions in XPath are only a syntactic sugaring. First, we prove that the generally quantified expression can be rewritten to an existentially quantified one, next we prove that an existentially quantified expression can be rewritten with an equal expression that is not quantified.

As noted in the XPath introduction, we can use some-in-satisfies and every-in-satisfies quantified expressions in XPath, where the former represents an existential quantifier, and the latter an universal quantifier.

$$
\begin{aligned}
S' &= \\
&= \mathcal{I}_{\mathcal{E}}[\![\texttt{for \$i in (0 to fn:count(}S\texttt{) - 1) return} \\
&\qquad \texttt{for \$x in } \langle S, \prec_S \rangle \texttt{ return} \\
&\qquad \texttt{if (\$i = } \mathit{CountLessThan}_R(\texttt{\$x}, S)\texttt{) then \$x else ()}]\!]C && (1) \\
&= \{\, x_2 \mid \; x_1 \in \{\, 0, \ldots, |S| - 1 \,\} \\
&\qquad C_1 = C(\texttt{\$i} \rightarrow x_1) \\
&\qquad x_2 \in \mathcal{I}_{\mathcal{E}}[\![\texttt{for \$x in } \langle S, \prec_S \rangle \texttt{ return } \ldots]\!]C_1 \,\} && (2) \\
&= \{\, x_2 \mid \; x_1 \in \{\, 0, \ldots, |S| - 1 \,\} \\
&\qquad C_1 = C(\texttt{\$i} \rightarrow x_1) \\
&\qquad x_2 \in \{\, x_4 \mid \; x_3 \in S \\
&\qquad\qquad C_2 = C_1(\texttt{\$x} \rightarrow x_3) \\
&\qquad\qquad x_4 \in \begin{cases} \mathcal{I}_{\mathcal{E}}[\![\texttt{\$x}]\!]C_1 & \text{if } \mathcal{W}[\![\texttt{\$i = } \mathit{CountLessThan}_R(\texttt{\$x}, S)]\!]C_2 \\ \emptyset & \text{otherwise} \end{cases} \\
&\qquad\qquad \} \\
&\qquad \} && (3) \\
&= \{\, x_2 \mid \; x_1 \in \{\, 0, \ldots, |S| - 1 \,\} \\
&\qquad C_1 = C(\texttt{\$i} \rightarrow x_1) \\
&\qquad x_2 \in \{\, x_4 \mid \; x_3 \in S \\
&\qquad\qquad C_2 = C_1(\texttt{\$x} \rightarrow x_3) \\
&\qquad\qquad x_4 \in \mathcal{I}_{\mathcal{E}}[\![\texttt{\$x}]\!]C_2 \\
&\qquad\qquad \mathcal{W}[\![\texttt{\$i = } \mathit{CountLessThan}_R(\texttt{\$x}, S)]\!]C_2 \,\} \\
&\qquad \} && (4) \\
&= \{\, x_2 \mid \; x_1 \in \{\, 0, \ldots, |S| - 1 \,\} \\
&\qquad x_2 \in \{\, x_3 \mid x_3 \in S,\; x_1 = \mathit{CLT}_R(x_3, S) \,\} \,\} && (5) \\
&= S && (6)
\end{aligned}
$$

Figure 12: Proof of Theorem 4.1: $S' = S$.

$$
\begin{array}{lll}
\prec_{S'} = & & \\
= & \sqsubset_{\mathcal{E}[\![\texttt{for \$i in (0 to fn:count(}S\texttt{) - 1) return}} & (1)\\
& \quad \texttt{for \$x in } \langle S, \prec_S \rangle \texttt{ return} & \\
& \quad\quad \texttt{if (\$i = } \mathit{CountLessThan}_R\texttt{(\$x,}S\texttt{)) then \$x else ()}]\!]C & \\
= & \{ \langle x,y \rangle \mid \; x_1, y_1 \in \{ 0, \ldots, |S| - 1 \} & (2)\\
& \quad \mathrm{S}_1 = \mathcal{E}[\![\texttt{for \$x in } \langle S, \prec_S \rangle \texttt{ return } \ldots]\!]C(\texttt{\$i} \to x_1) & \\
& \quad \mathrm{S}_2 = \mathcal{E}[\![\texttt{for \$x in } \langle S, \prec_S \rangle \texttt{ return } \ldots]\!]C(\texttt{\$i} \to y_1) & \\
& \quad x_1 \in \mathcal{I}_{\mathrm{S}_1}, y_1 \in \mathcal{I}_{\mathrm{S}_2} \text{ implies } x_1 < y_1 & \\
& \quad x_1, y_1 \in \mathcal{I}_{\mathrm{S}_1} \text{ implies } x \sqsubset_{\mathrm{S}_1} y \; \} & \\
= & \{ \langle x,y \rangle \mid \; x_1, y_1 \in \{ 0, \ldots, |S| - 1 \} & (3)\\
& \quad \mathcal{I}_{\mathrm{S}_1} = \{ x_3 \mid x_3 \in S, \; x_1 = CLT_R(x_3, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}_2} = \{ x_4 \mid x_4 \in S, \; y_1 = CLT_R(x_4, S) \} & \\
& \quad x_1 < y_1 \text{ or } \langle x,y \rangle \in \sqsubset_{\mathrm{S}_1} \} & \\
= & \{ \langle x,y \rangle \mid \; x_1, y_1 \in \{ 0, \ldots, |S| - 1 \} & (4)\\
& \quad \mathcal{I}_{\mathrm{S}_1} = \{ x_3 \mid x_3 \in S, \; x_1 = CLT_R(x_3, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}_2} = \{ x_4 \mid x_4 \in S, \; y_1 = CLT_R(x_4, S) \} & \\
& \quad \text{either } x_1 < y_1 & \\
& \quad \text{or } \langle x,y \rangle \in \{ \langle x',y' \rangle \mid \; x_2, y_2 \in S & \\
& \qquad\qquad x' \in \mathcal{I}_{\mathrm{S}'_1} = \{ x_5 \mid x_5 \in S, \; x_1 = CLT_R(x_2, S) \} & \\
& \qquad\qquad y' \in \mathcal{I}_{\mathrm{S}'_2} = \{ x_6 \mid x_6 \in S, \; x_1 = CLT_R(y_2, S) \} & \\
& \qquad\qquad x_2 \prec_S y_2 \text{ or } \langle x,y \rangle \in \sqsubset_{\mathcal{E}[\![\texttt{if}\ldots]\!]C(\texttt{\$i} \to x_1)(\texttt{\$x} \to x_2)} \} & \\
& \quad \} & \\
= & \{ \langle x,y \rangle \mid \; x_1, y_1 \in \{ 0, \ldots, |S| - 1 \} & (5)\\
& \quad \mathcal{I}_{\mathrm{S}_1} = \{ x_3 \mid x_3 \in S, \; x_1 = CLT_R(x_3, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}_2} = \{ x_4 \mid x_4 \in S, \; y_1 = CLT_R(x_4, S) \} & \\
& \quad \text{either } x_1 < y_1 & \\
& \quad \text{or } \langle x,y \rangle \in \{ \langle x',y' \rangle \mid \; x_2, y_2 \in S & \\
& \qquad\qquad x' \in \mathcal{I}_{\mathrm{S}'_1} = \{ x_5 \mid x_5 \in S, \; x_1 = CLT_R(x_2, S) \} & \\
& \qquad\qquad y' \in \mathcal{I}_{\mathrm{S}'_2} = \{ x_6 \mid x_6 \in S, \; x_1 = CLT_R(y_2, S) \} & \\
& \qquad\qquad x_2 \prec_S y_2 \} & \\
& \quad \} & \\
= & \{ \langle x,y \rangle \mid \; x_1, y_1 \in \{ 0, \ldots, |S| - 1 \} & (5)\\
& \quad \mathcal{I}_{\mathrm{S}_1} = \{ x_3 \mid x_3 \in S, \; x_1 = CLT_R(x_3, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}_2} = \{ x_4 \mid x_4 \in S, \; y_1 = CLT_R(x_4, S) \} & \\
& \quad x_2, y_2 \in S & \\
& \quad \mathcal{I}_{\mathrm{S}'_1} = \{ x_5 \mid x_5 \in S, \; x_1 = CLT_R(x_2, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}'_2} = \{ x_6 \mid x_6 \in S, \; x_1 = CLT_R(y_2, S) \} & \\
& \quad x_1 < y_1 \text{ or } \langle x,y \rangle \in \{ \langle x',y' \rangle \mid x' \in \mathcal{I}_{\mathrm{S}'_1}, \; y' \in \mathcal{I}_{\mathrm{S}'_2}, \; x_2 \prec_S y_2 \} \} & \\
= & \{ \langle x,y \rangle \mid \; x_1, y_1 \in \{ 0, \ldots, |S| - 1 \} & (6)\\
& \quad \mathcal{I}_{\mathrm{S}_1} = \{ x_3 \mid x_3 \in S, \; x_1 = CLT_R(x_3, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}_2} = \{ x_4 \mid x_4 \in S, \; y_1 = CLT_R(x_4, S) \} & \\
& \quad x_2, y_2 \in S & \\
& \quad \mathcal{I}_{\mathrm{S}'_1} = \{ x_5 \mid x_5 \in S, \; x_1 = CLT_R(x_2, S) \} & \\
& \quad \mathcal{I}_{\mathrm{S}'_2} = \{ x_6 \mid x_6 \in S, \; x_1 = CLT_R(y_2, S) \} & \\
& \quad x_1 < y_1 \text{ or } (x \in \mathcal{I}_{\mathrm{S}'_1}, \; y \in \mathcal{I}_{\mathrm{S}'_2}, \; x_2 \prec_S y_2) \} &
\end{array}
$$

Figure 13: Proof of Theorem 4.1: Derivation of $\prec_{S'}$.

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathcal{I}_{\mathcal{E}[\![\texttt{every \$}v\texttt{ in }e_1\texttt{ satisfies }e_2]\!]C}} & \\
= & \mathcal{W}[\![\texttt{every \$}v\texttt{ in }e_1\texttt{ satisfies }e_2]\!]C & (1)\\
= & \textit{true}\text{ iff S} = \mathcal{E}[\![e_1]\!]C, \forall x \in \mathcal{I}_S : \mathcal{W}[\![e_2]\!]C(v \rightarrow x) & (2)\\
= & \textit{true}\text{ iff S} = \mathcal{E}[\![e_1]\!]C, \neg\exists x \in \mathcal{I}_S : \neg\mathcal{W}[\![e_2]\!]C(v \rightarrow x) & (3)\\
= & \textit{true}\text{ iff S} = \mathcal{E}[\![e_1]\!]C, \neg\exists x \in \mathcal{I}_S : \mathcal{W}[\![\texttt{fn:not(}e_2\texttt{)}]\!]C(v \rightarrow x) & (4)\\
= & \neg\textit{true}\text{ iff S} = \mathcal{E}[\![e_1]\!]C, \exists x \in \mathcal{I}_S : \mathcal{W}[\![\texttt{fn:not(}e_2\texttt{)}]\!]C(v \rightarrow x) & (5)\\
= & \neg\mathcal{W}[\![\texttt{some \$}v\texttt{ in }e_1\texttt{ satisfies fn:not(}e_2\texttt{)}]\!]C & (6)\\
= & \mathcal{W}[\![\texttt{fn:not(some \$}v\texttt{ in }e_1\texttt{ satisfies fn:not(}e_2\texttt{))}]\!]C & (7)\\
= & \mathcal{I}_{\mathcal{E}[\![\texttt{fn:not(some \$}v\texttt{ in }e_1\texttt{ satisfies fn:not(}e_2\texttt{))}]\!]C} & (8)
\end{array}
$$

Figure 14: Proof of Lemma 4.5.

```
some $cd in //cd satisfies $cd/year = 1992
//cd[every $a in author|artist satisfies $a/first = "Karel"]
```

We use the Core XPath syntax of quantified expressions here. This means that only a single variable binding can appear in front of the satisfies keyword, while the full XPath syntax allows a list of variable bindings there. W3C's formal semantics explains that a quantified expression with a list of variable bindings can be rewritten to an expression with nested quantified expressions each with a single variable binding.

First, we show how an every-in-satisfies XPath expression can be rewritten with some-in-satisfies expression in Lemma 4.5.

Lemma 4.5 *Let* `every` $\$v$ `in` e_1 `satisfies` e_2 *be a generally quantified XPath expression. Then the following XPath expression has equal semantics.*

$$\texttt{fn:not(some \$}v\texttt{ in }e_1\texttt{ satisfies fn:not(}e_2\texttt{))}$$

Proof. We prove that the expression evaluates to the same. Obviously, the result is an atomic expression, so we need to check only if it has the same value.

The equations of the prove are presented in Figure 14. Equation (1) comes directly from $\mathcal{E}[\![\texttt{every}\ldots]\!]C$, equation (2) is the result of $\mathcal{W}[\![\texttt{every}\ldots]\!]C$, general quantifier is replaced with existential quantifier in (3), negation is replaced with fn:not() in (4), both sides of iff are negated in (5), the result of which is equal to $\mathcal{W}[\![\texttt{some}\ldots]\!]C$, finally negation is replaced with fn:not() in (7) to reach the evidence in (8). □

Next, we show how an some-in-satisfies expression can be rewritten using the for-in-return expression in combination with fn:boolean() function, which given a sequence returns false only if the sequence is empty. This is formalized in Lemma 4.6. We use the for-in-return expression to iterate over possible bindings and to return true for each binding that satisfies the condition from some-in-satisfies expression, and nothing for each binding that does not satisfy the condition. Thus, we get a nonempty sequence only if there is at least one variable binding that satisfies the condition, otherwise an empty sequence is the result of the for-in-return expression. Finally, we apply the fn:boolean() function to the sequence returned by the for-in-return expression, which guarantees that true is returned only if there is a variable binding that satisfies the condition, which is the semantics of the some-in-satisfies expression.

$$
\begin{array}{lll}
& \mathcal{I}_{\mathcal{E}[\![\texttt{fn:boolean(for \$}v\texttt{ in }e_1\texttt{ return if(}e_2\texttt{) then fn:true() else ())}]\!]C} & \\
= & \textit{true} \text{ iff } \left|\mathcal{I}_{\mathcal{E}[\![\texttt{for \$}v\texttt{ in }e_1\texttt{ return if(}e_2\texttt{) then fn:true() else ()}]\!]C}\right| > 0 & (1)\\
= & \textit{true} \text{ iff } |\{\, x_2 \mid x_1 \in \mathcal{I}_{\mathcal{E}[\![e_1]\!]C} & (2)\\
& \quad C_1 = C(v \rightarrow x_1) & \\
& \quad x_2 \in \mathcal{I}_{\mathcal{E}[\![\texttt{if(}e_2\texttt{) then fn:true() else ()}]\!]C_1} \,\}| > 0 & \\
= & \textit{true} \text{ iff } |\{\, x_2 \mid x_1 \in \mathcal{I}_{\mathcal{E}[\![e_1]\!]C} & (3)\\
& \quad C_1 = C(v \rightarrow x_1) & \\
& \quad \mathcal{W}[\![e_2]\!]C_1 & \\
& \quad x_2 \in \mathcal{I}_{\mathcal{E}[\![\texttt{fn:true()}]\!]C_1} \,\}| > 0 & \\
= & \textit{true} \text{ iff } |\{\, x_2 \mid x_1 \in \mathcal{I}_{\mathcal{E}[\![e_1]\!]C} & (4)\\
& \quad C_1 = C(v \rightarrow x_1) & \\
& \quad \mathcal{W}[\![e_2]\!]C_1 & \\
& \quad x_2 \in \{\, \textit{true} \,\} \,\}| > 0 & \\
= & \textit{true} \text{ iff } |\{\, \textit{true} \mid x_1 \in \mathcal{I}_{\mathcal{E}[\![e_1]\!]C} & (5)\\
& \quad C_1 = C(v \rightarrow x_1) & \\
& \quad \mathcal{W}[\![e_2]\!]C_1 \,\}| > 0 & \\
= & \textit{true} \text{ iff } \exists x_1 \in \mathcal{I}_{\mathcal{E}[\![e_1]\!]C} : \mathcal{W}[\![e_2]\!]C(v \rightarrow x_1) & (6)\\
= & \mathcal{W}[\![\texttt{some \$}v\texttt{ in }e_1\texttt{ satisfies }e_2]\!]C & (7)\\
= & \mathcal{I}_{\mathcal{E}[\![\texttt{some \$}v\texttt{ in }e_1\texttt{ satisfies }e_2]\!]C} & (8)
\end{array}
$$

Figure 15: Proof of Lemma 4.6.

Lemma 4.6 *Let* `some $v in` e_1 `satisfies` e_2 *be an existentially quantified XPath expression. Then the following XPath expression has equal semantics.*

```
fn:boolean(
  for $v in e1 return
    if (e2)
    then fn:true()
    else ()
)
```

Proof. Since the result of both expressions is clearly a single atomic value, we consider only items and not the order of the result sequence.

The equations of the proof are presented in Figure 15. Equation (1) represents the semantics of fn:boolean() functions on sequences, which returns true only if the sequence is nonempty. Equation (2) unfolds the for-in-return expression. Equation (3) unfolds the if-then-else expression. Notice, that this expression returns a value only in the then branch, therefor we can rewrite in this simple way. Equation (4) unfolds the fn:true() function, which results in a single atomic value. Equation (5) simplifies the set expression emphasizing that only *true* values can be returned. Equation (6) comes from the fact that the set in (5) is nonempty if at least one item in $\mathcal{I}_{\mathcal{E}[\![e_1]\!]C}$ satisfies the condition $\mathcal{W}[\![e_2]\!]C(v \rightarrow x_1)$. We can notice, that (6) is exactly the semantics of some-in-satisfies expression, as expressed in equation (7), and (8). □

Finally, we can conclude that the quantified expressions some-in-satisfies and every-in-satisfies of XPath 2.0 are only syntactic sugaring.

Theorem 4.2 *XPath 2.0 without quantified expressions some-in-satisfies, and every-*

in-satisfies has the same expressive power as XPath 2.0 with these quantified expressions.

Proof. As Lemma 4.5 proves, each every-in-satisfies expression can be replaced with an equal some-in-satisfies expression. Thus every-in-satisfies can be removed from XPath 2.0 without loss of expressive power of the language.

Further, Lemma 4.6 proves that some-in-satisfies expression can be replaced with an equal expression that consists from fn:boolean() function, a for-in-return expression, and if-then-else expression. Therefore even some-in-satisfies expression can be removed from XPath 2.0 without losing the expressive power. □

To sum up, we proved in this section that quantified expressions do not extend the expressive power of XPath 2.0.

4.6 Conclusion

We studied a query language XPath 2.0 for XML, a successor to the most recognized query language for XML – XPath 1.0.

In section 4.1, we provided introduction to both versions of the XPath language. We put an emphasis on distinguishing new features of XPath 2.0 from features of its predecessor.

Then, we provided formal syntax for an untyped fragment of XPath in section 4.2. For an important part of this fragment we defined formal semantics in section 4.3 that is used in formal proofs.

A surprising fact that XPath 2.0 is so strong that it can even sort arbitrary sequences is formally proved in section 4.4. Consequences of XPath's sorting ability are used later in section 5.4 on sorting in XQuery.

Finally, we proved in section 4.5 that quantified expressions do not extend expressive power of XPath 2.0.

5 XQuery

This chapter focuses on XQuery 1.0 – a query language for XML developed by W3C to become a standard query language for XML.

XQuery is strongly connected to XPath 2.0. Not only they share a data model, a set of functions, and a formal semantics specification, but XQuery is in fact a superset of XPath, so every XPath expression is also an XQuery expression.

Historically, XQuery developed from Quilt [16], which has its roots in XML-QL [21], XQL [51], SQL [37], and OQL [15]. Influence certainly comes besides XML pioneer query languages also from query languages for semistructured data, relational, and object databases.

XQuery language is distributed in several W3C's specifications. They are XQuery 1.0 [8], which defines syntax and semi-formal semantics, XQuery 1.0 and XPath 2.0 Formal Semantics [22], which defines formal semantics, and XQuery 1.0 and XPath 2.0 Functions and Operators [43].

As XQuery is a language with a lot of syntactic structures, a core language called XQuery Core is introduced in the formal semantics specification, for which the semantics is defined.

The contribution of this chapter is twofold.

- We introduce formal semantics including sorting, which is missing in the W3C's specification. This semantics is based on tuples and nicely demonstrates XQuery sorting semantics that some people find hard to understand.
- We explain that, unlike the authors of XQuery formal specification thought, sorting semantics is expressible in the used data model, which we prove. This leads to cleaner XQuery Core, which has a strange property nowadays: the specification provides no transformation from a general FLWOR expressions with an order by clause to XQuery Core, as it is considered infeasible. We show that such a transformation exists.

This chapter is organized as follows. First, we introduce XQuery expressions that are not present in XPath in section 5.1 and illustrate them with examples. Then, syntax and formal semantics is defined in sections 5.2 and 5.3, respectively. Finally, we explain insufficiency of formal semantics of sorting and provide a transformation of a general FLWOR expression to an equal query without an order by clause in section 5.4.

5.1 Introduction

XQuery 1.0 is a superset of XPath 2.0 both in syntax and semantics. As such, every XPath expression is an XQuery expression. In this section, we focus on those parts of XQuery that distinguish it from XPath. These are constructor expressions that create new nodes, FLWOR expressions, and last but not least the ability to define functions.

5.1.1 Constructors

Unlike XPath, XQuery provides expressions to construct new nodes. Constructor expressions use XML syntax. For example, the following is an XQuery constructor that returns a singleton sequence comprising one element node with name artist and with an id attribute with value 1. This element has two subelements with names first and last where each contains a text node with content Jaco, and Pastorius respectively.

```
<artist id="1">
  <first>Jaco</first>
  <last>Pastorius</last>
</artist>
```

All newly constructed nodes get a new node identity. Document order is also defined for newly constructed nodes, but one has to be careful. For some expression XQuery defines the document order for newly constructed nodes, while for some expression the document order depends on implementation. For example, in the example above, the document order is defined for elements first and last, since they have a common parent.

We showed how new nodes can be constructed, but we provided only a constant expression that is independent from the queried data. However, we can compute the contents of new nodes in XQuery. If we want to use an XQuery expression inside a constructor, we have to encapsulate it into curly braces. For example, the following expression returns a singleton sequence comprising a new element node named all-authors that contains all authors from a queried document.

```
<all-authors>{ //author }</all-authors>
```

It is not only element nodes with attributes XQuery has constructors for, even processing instructions and comments can be constructed similarly using XML syntax.

5.1.2 FLWOR Expressions

FLWOR expression is the most characteristic language construct of XQuery. In fact, it is an extension of XPath's for-in-return expression, where a FLWOR expression can in addition to for and return clauses contain a let, where and order by clauses. All of them are optional.

The idea is that the for and let clauses define an ordered stream of variable bindings. This stream is then filtered using an expression in the where clause to be sorted according to an expression in the order by clause. The resulting stream consists of satisfying variable bindings in the desired order. Finally, for each tuple from this stream the return expression is evaluated.

We explain the semantics of a FLWOR expression on the following example.

```
for $cd in //cd
let $numAuthors := count($cd/author),
    $numArtists := count($cd/artist)
where $cd/year > 1990
order by $numAuthors, $numArtists
```

```
return
  <cd>{
    $cd/title,
    <num-authors>{ $numAuthors }</num-authors>,
    <num-artists>{ $numArtists }</num-artists>
  }</cd>
```

This query iterates over a sequence of CDs that are one by one bound to a $cd variable. For each CD a number of authors and a number of artists performing on that CD are counted in the let clause and bound to $numAuthors and $numArtists variables, respectively. The stream of variable bindings is in this case a stream of 3-tuples, where the first value is bound to variable $cd, the second value is bound to variable $numAuthors, and the third value is bound to variable $numArtists.

The number of tuples in the initial stream is equal to a number of CDs in a queried document.

Now, the where clause is applied. It filters this stream to contain only such tuples that refer to a CD that originated after 1990.

Then, the order by clause is applied. It sorts the filtered stream according to a number of authors and a number of artists.

Finally, the sorted stream is iterated. For each tuple the values are bound to their respective variables and with this binding the return expression is evaluated. In our example, each tuple refers to one CD. So for each satisfying CD a return expression is evaluated, returning a new cd element with a CD's title element and with number of authors and artists.

For and Let Clause. Notice, that both for and let clauses define a variable that is initialized with an expression. Since every expression evaluates to a sequence, in both cases a variable is initialized with a sequence. The difference is that while the for clause iterates over this sequence and binds a variable to one item after another, the let clause binds a variable to the whole sequence at once.

Multiple Variables in a For Clause. There can be more than one variable defined in a for clause. In such a case, the query is understood as if it contained nested for expressions, where each for expression binds only one variable. Let's examine the following example.

```
for $cd in //cd,
    $art in $cd/artist
return ...
```

The for clause defines two variables: $cd that binds one by one to all cd elements in a document, and $artist that binds one by one to all artist within an $cd. This query is understood as follows.

```
for $cd in //cd
return
    for $art in $cd/artist
    return ...
```

Thus, if multiple variables are defined in a for clause, the query is rewritten with nested for expressions, where each for expression is handled separately.

Multiple Variables in a Let Clause. Even a let clause can define multiple variables. In such a case, a let clause is understood as if there was a single let clause for each variable binding. But unlike for clauses, let clauses are not nested. Let's check the example below.

```
let $cd := //cd,
    $art := $cd/art
return ...
```

The let clause defines two variables: $cd that binds to a sequence of all cd elements in a document, and $artist that binds to a sequence of all artists of all CDs. This query is understood as follows.

```
let $cd := //cd
let $artist := //artist
return ...
```

Thus, if multiple variables are defined in a let clause, the query is rewritten to a query with multiple let clauses at the same level and no nesting is done.

Sorting. Here, we want to make the semantics of the order by clause of a FLWOR expression clear.

We already explained that it is the stream of tuples that is sorted according to the contents of an order by clause.

The example query, that we present at the beginning of this section, reveals that an order by clause can contain several expressions separated by a comma. These expressions are called *order specifications*, or orderspecs for short. The order in which orderspecs appear in the order by clause is important, as the significance of orderspecs for the resulting order decreases from left to right.

The resulting order of a tuple stream is defined as a condition for two tuples that tells whether one precedes the other in the sorted stream. We start with the first orderspec. We compute its value twice: first, with variables bound to values from the first tuple and second, with variables bound to values from the second tuple. Now, we have two computed values of the first orderspec. If the first is less then the second, then the first tuple has to precede the second tuple in the sorted stream, and vice versa. If the two values are equal, we take the second orderspecs and continue likewise.

In case all computed values of all orderspecs are equal, then the tuples appear in the sorted stream in the same order as in the initial stream.

Let's check it on an example. Figure 16a) illustrates a tuple stream for our example query. Each CD is represented by a number that corresponds to an order in which that CD appears in cd.xml file. For cd.xml see Figure 3 on page 18.

The sorted stream is depicted in Figure 16b). Since the orderspecs are direct references to values of variables $numAuthors and $numArtists, the stream is sorted first according to $numAuthors column and second to $numArtists column.

Notice, that equal values for both orderspecs are computed for tuples with CD number 1 and 4. Since the tuple with CD number 1 precedes the tuple with CD number 4 in the initial stream, the tuple with CD number 1 precedes the tuple with CD number 4 in the sorted stream as well.

$cd	$numAuthors	$numArtists
1	0	1
2	1	1
3	1	2
4	0	1

a) initial tuple stream

$cd	$numAuthors	$numArtists
1	0	1
4	0	1
2	1	1
3	1	2

b) sorted tuple stream

Figure 16: Tuple stream for the example query.

5.1.3 Functions

Extending XPath with syntax for function definition really extends the power of XQuery greatly. Functions can have arguments. A body of a function is defined with an XQuery expression.

```
declare function local:CDsAuthoredBy($author) {
    for $cd in //cd
    let $a := $cd/author
    where $a/first = $author/first and $a/last = $author/last
    return $cd
};
```

The above defines a function that gets an author element and returns a sequence of CDs authored by this author. Once defined, a function can be called at any XQuery expression where a function call is allowed. For example, the following query returns CDs authored by Karel Capek.

```
let $capek :=
  <author>
    <first>Karel</first>
    <last>Capek</last>
  </author>
return local:CDsAuthoredBy($capek)
```

As we know, function calls can appear inside a path expression. One only has to be careful. The focus, in which a function is called, is *not* transferred inside the function's body. Rather, a function is evaluated with an empty focus. See semantics of a function call in XPath semantics in section 4.3.

Functions can be called recursively in XQuery.

5.2 XQuery Syntax

The syntax for an XQuery fragment is defined in Figure 17.

An XQuery 1.0 query consists of a prolog and a query body. We separate the syntax for query prolog from the syntax for query body with a horizontal line.

According to the specification, the prolog is used to define namespaces, import schemas, declare variables and functions, and import other XQuery modules. We can see that in our XQuery fragment we allow only definition of functions and variables. The removal of the other syntax elements is explained later.

The query body is an expression that is represented by Expr nonterminal in the grammar. An expression Expr is a possibly singleton comma-separated list of simple expressions, these are referred to as ExprSingle in the grammar. A grammar production for ExprSingle is exactly the same as in XPath, we provide it here only because it references the FLWOR expression, which is richer in XQuery.

In concert with the tuple semantics of a FLWOR expression, the FLWOR expression comprises a tuple stream generating expression TupleExpr and a return expression. TupleExpr consists of a block of mixed for and let expressions, followed by an optional where clause. The tuple stream can be sorted according to an optional order by clause that comes right after the where clause.

What is removed. It copies the XQuery 1.0 syntax but XPath syntax, constructors, and all type dependent expressions are removed.

As we provide the syntax with XPath extracted, we can see that there is not much added in XQuery compared to XPath. It is only constructors, function definitions and FLWOR expression that makes XQuery syntax more powerful.

We do not provide XQuery constructors, as this would enlarge the listing only and there is nothing so much interesting about it. The XQuery constructors are introduced informally in section 5.1.1.

As we do not bother with types, we do not bother with importing schemas in the prolog.

Further more, we do not even consider importing modules, as the module importing mechanism in query's prolog is designed only to develop libraries of functions and so ease the work of a query writer. From our point of view, there is no difference between importing a module and pasting the contents of the imported module directly into the importing module.

Moreover, we limit the syntax of prolog to a block of function declarations followed by a block of variable declarations, though the specifications allows function and variable declarations to be mixed up. The reason is that an expression from a variable declaration can refer to *any* function in the prolog, even to one that is declared after the variable declaration. By forcing the function declarations block precede the variable declarations block, we get a syntax that is easier to define semantics for, while not changing the expressive power of the language.

Not XQuery Core. We do not use the syntax of XQuery Core as the basis. We take the broader syntax definition instead, because we want to define tuple semantics of the language formally. This is good to understand the sorting semantics of XQuery as there is no formal definition of sorting in W3C's specifications.

Though, as we prove in section 5.4, XQuery Core can be used and is sufficient to express the sorting semantics of XQuery.

5.3 XQuery Formal Semantics

In this section, we provide formal semantics for an XQuery syntax fragment that is described in the preceding section. The formal semantics we use is based on tuple semantics that is defined in two W3C specifications: XQuery 1.0 [8] and XQuery 1.0 and XPath 2.0 Formal Semantics [22].

As we mentioned earlier, the W3C formal semantics is not able to express semantics of sorting. In fact, the whole document appears as if it was firstly

```
XQuery        ::=  Prolog Expr
Prolog        ::=  FunctionDefs? VariableDefs?
FunctionDefs  ::=  SimpleFnDef ( FunctionDefs )?
SimpleFnDef   ::=  "declare function" QName "(" ( ParamList )? ")"
                   "{" Expr "}"
ParamList     ::=  "$"VarName ( ParamList )?
VariableDefs  ::=  SimpleVarDef ( VariableDefs )?
SimpleVarDef  ::=  "declare variable" "$"VarName "{" Expr "}"
```

```
Expr          ::=  ExprSingle ( "," Expr )?
ExprSingle    ::=  FLWORExpr
                   | QuantifiedExpr
                   | IfExpr
                   | OrExpr
FLWORExpr     ::=  TupleExpr "return" ExprSingle
TupleExpr     ::=  WhereExpr ( "order by" OList )?
WhereExpr     ::=  FLExpr ( "where" ExprSingle )?
FLExpr        ::=  ("for" FList | "let" LList ) ( FLExpr )?
FList         ::=  "$"VarName ( "at" "$"VarName )? "in" ExprSingle
                   ("," FList )?
LList         ::=  "$"VarName ":=" ExprSingle ( "," LList )?
OList         ::=  ExprSingle ( "," OList )?
```

Figure 17: XQuery syntax – with XPath extracted.

$\mathcal{E}[\![\text{Prolog Expr}]\!]C$	$\mathcal{E}[\![\text{Expr}]\!]\mathcal{P}[\![\text{Prolog}]\!]C$
$\mathcal{E}[\![\text{FLWOExpr}\ \texttt{return}\ e]\!]C$	$\mathbb{T} = \mathcal{T}[\![\text{FLWOExpr}]\!]C$ $S = \{\ x \mid t \in \mathbb{T},\ x \in \mathcal{I}_{\mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \to t)}\ \}$ $\prec_S = \{\ \langle x, y \rangle \mid t_1, t_2 \in \mathbb{T},\ x \in \mathcal{I}_{\mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \to t_1)},\ y \in \mathcal{I}_{\mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \to t_2)},\ t_1 \sqsubset_{\mathbb{T}} t_2\ \}$

Figure 18: XQuery expression evaluation semantics for context $C = \langle \rho, F, item, pos, last \rangle$.

designed as formal semantics of XPath and then extended to express the semantics of XQuery, which did not fit much nicely. However, the authors of the formal specification concluded that sorting is not expressible in the data model of XPath and XQuery, which we later show not to be true.

As there is no formal specification of tuple semantics of FLWOR expressions, we decided to provide one.

We define XQuery formal semantics using semantic function as in the section on XPath. The main semantic function is function $\mathcal{E}$ that evaluates an expression in a given context. The result of this evaluation is a sequence of items.

$$\mathcal{E}\ :\ Expression \to Context \to Seq(Item)$$

Function $\mathcal{E}$ is defined in Figure 18. As we omit XPath language constructs, we omit the definition of $\mathcal{E}$ on XPath expressions, as well. In fact, function $\mathcal{E}$ returns the same as function $\mathcal{E}$ that we defined in section 4.3 on XPath formal semantics.

Function $\mathcal{E}$ is defined for a FLWOR expression and for a whole query. We are interested mainly in the tuple semantics of a FLWOR expression, which we inspect in the following section. To be complete, we provide the semantics also for the XQuery prolog in section 5.3.2.

5.3.1 FLWOR Expression

We focus on the semantics of a FLWOR expression. With $\mathbb{T}$we denote a tuple stream that is generated by the FLWO part of the FLWOR expression, with S we denote items S of the resulting sequence and with $\prec_S$ we denote the order of items in S.

We can see, that S is constructed as follows. For each tuple $t = \langle x_1, \ldots, x_n \rangle$ from $\mathbb{T}$, which represents values of variables $v_1, \ldots, v_n$ that are defined in the FL part of the FLWOR expression, the return expression e is evaluated in a context with variables $v_1, \ldots, v_n$ bound to their respective values $x_1, \ldots, x_n$.

The order of items in S is defined according to the order of tuples in the tuple stream. One item from S precedes another item from S if the tuple of the former item precedes the tuple of the latter item in the tuple stream.

$\mathcal{T}[\![v := e]\!]C$	$T = \{ \langle x \rangle \mid x = \mathcal{E}[\![e]\!]C \}$ $\prec_T = \emptyset$ $\theta_T = \langle \{v\}, \emptyset \rangle$
$\mathcal{T}[\![v \texttt{ in } e]\!]C$	$T = \{ \langle x \rangle \mid x \in \mathcal{I}_{\mathcal{E}[\![e]\!]C} \}$ $\prec_T = \{ \langle t_1, t_2 \rangle \mid t_1 = \langle x_1 \rangle \in T,\ t_2 = \langle x_2 \rangle \in T,\ x_1 \sqsubset_{\mathcal{E}[\![e]\!]C} x_2 \}$ $\theta_T = \langle \{v\}, \emptyset \rangle$
$\mathcal{T}[\![v_1 \texttt{ at } v_2 \texttt{ in } e]\!]C$	$T = \{ \langle x, p \rangle \mid x \in \mathcal{I}_{\mathcal{E}[\![e]\!]C},\ p = pos(x, \mathcal{E}[\![e]\!]C) \}$ $\prec_T = \{ \langle t_1, t_2 \rangle \mid t_1 = \langle x_1, p_1 \rangle \in T,\ t_2 = \langle x_2, p_2 \rangle \in T,\ x_1 \sqsubset_{\mathcal{E}[\![e]\!]C} x_2 \}$ $\theta_T = \langle \{v_1, v_2\}, \{\langle v_1, v_2 \rangle\} \rangle$
$\mathcal{T}[\![\text{LetSimple, LList}]\!]C$	$[\![\text{LetSimple}]\!] \otimes [\![\text{LList}]\!]$
$\mathcal{T}[\![\text{ForSimple, FList}]\!]C$	$[\![\text{ForSimple}]\!] \otimes [\![\text{FList}]\!]$
$\mathcal{T}[\![\text{LetExpr FLExpr}]\!]C$	$[\![\text{LetExpr}]\!] \otimes [\![\text{FLExpr}]\!]$
$\mathcal{T}[\![\text{ForExpr FLExpr}]\!]C$	$[\![\text{ForExpr}]\!] \otimes [\![\text{FLExpr}]\!]$
$\mathcal{T}[\![\text{FLExpr} \texttt{ where } e]\!]C$	$\mathbb{T} = \mathcal{T}[\![\text{FLExpr}]\!]C$ $T = \{ x \mid x \in \mathcal{I}_{\mathbb{T}},\ \mathcal{W}[\![e]\!]C(\Theta_{\mathbb{T}} \to \langle \vec{x} \rangle) \}$ $\prec_T = \sqsubset_{\mathbb{T}}$ $\theta_T = \Theta_{\mathbb{T}}$
$\mathcal{T}[\![\text{WhereExpr} \texttt{ order by } \text{OList}]\!]C$	$\mathbb{T} = \mathcal{T}[\![\text{WhereExpr}]\!]C$ $T = \mathcal{I}_{\mathbb{T}}$ $\prec_T = \mathcal{O}_{\mathbb{T}}[\![\text{OList}]\!]C$ $\theta_T = \Theta_{\mathbb{T}}$

Figure 19: XQuery tuple semantics. Each expression evaluates to tuple stream $\langle T, \prec_T, \theta_T \rangle$, which provides variable bindings in a specified order.

Function $\mathcal{T}$ describes how a tuple stream is constructed. It evaluates a tuple expression, which is the FLWO part of a FLWOR expression, within a context and returns an ordered sequence of tuples – a tuple stream.

$$\mathcal{T} \quad : \quad TupleExpr \to Context \to Tuples$$

Function $\mathcal{T}$ is defined in Figure 19. Each expression evaluates to a tuple stream that comprises a set of tuples T, an order on tuples $\prec_T$, and a variable index θ_T. Recall that a variable index is a sequence of variable names where the first variable name corresponds with the first values of tuples in T, the second variable name corresponds with the second values of tuples in T, etc.

Function $\mathcal{T}$ is defined as follows. Firstly, it defines how 1-tuples are generated from let and for clauses with a single variable. It shows how these 1-tuples are joined using a tuple glue operator defined below. So far, the initial tuple

stream is constructed from a block of for and let clauses. Then, an expression from the where clause is applied to filter the initial tuple stream. And finally, a list of expressions from the order by clause is used to sort the tuple stream.

1-tuples. Let's examine the creation of 1-tuples. The first line in Figure 19 represents a variable assignment $v := e$ from a let clause. We can see, that the result is one 1-tuple that contains a value of expression e. The order $\prec_T$ is empty, and the variable index θ_T contains only variable name v.

For example, expression `$a := (10, 20)` generates the following tuple stream with one 1-tuple that contains sequence (10, 20). The variable index contains only a name of variable `a`.

$$\begin{aligned} \mathbb{T} &= (\langle(10,20)\rangle) \\ \Theta_{\mathbb{T}} &= (\texttt{a}) \end{aligned}$$

The second line describes variable iteration v `in` e from a for clause. Now, the tuple stream contains a 1-tuple for each item from a sequence that resulted from evaluation of e, with a value of that item. The order $\prec_T$ of tuples is derived from the order of items they contain. A tuple precedes another tuple if an item it contains precedes the item from the other tuple in the result of e evaluation. The variable index contains only variable v.

For example, expression `$a in (10, 20)` generates the following tuple stream with two tuples with values 10 and 20, respectively. The variable index contains only a name of variable `a`.

$$\begin{aligned} \mathbb{T} &= (\langle 10\rangle, \langle 20\rangle) \\ \Theta_{\mathbb{T}} &= (\texttt{a}) \end{aligned}$$

The third line is a special case of a variable iteration from a for clause. Expression v_1 `at` v_2 `in` e has the same meaning as v_1 `in` e with a positional variable v_2 added. Positional variable v_2 is bound to a position of item that is bound to v_1 in the result of e. The order $\prec_T$ is the same as in the previous case, and the variable index contains variable name v_1 at the first place and variable name v_2 in the second place.

For example, expression `$a at $p in (10, 20)` generates the following tuple stream with two tuples with values 10 and 20, respectively. The variable index contains names of variables `a` and `p` in that order.

$$\begin{aligned} \mathbb{T} &= (\langle 10,1\rangle, \langle 20,2\rangle) \\ \Theta_{\mathbb{T}} &= (\texttt{a,p}) \end{aligned}$$

n-tuples. So far, we are done with the building blocks of a tuple stream that generate streams of 1-tuples (or 2-tuples in case of positional variables). These are generated by a single variable assignment in a let or for clause. If there is more than one assignment, let's say there are two assignments, we first evaluate the first assignment and get an initial stream of 1-tuples. Then we iterate this stream and for each tuple we bind its value to the first variable and in this context we evaluate the expression from the second assignment getting again a stream of 1-tuples for this binding. The result is a stream of 2-tuples, where each tuple has at the first position a value from the initial stream, and at the second position it has a value from the stream that we get by evaluating

the second assignment with the first variable bound to the value at the first position. The resulting variable index is concatenation of the variable indexes of the two subexpressions.

This mechanism is formally described with a *tuple glue operator* denoted by $\otimes$. The glue operator takes two variable binding subexpressions e_1, and e_2 where e_2 follows e_1 in a query and returns a tuple stream. We use it to glue together tuples generated by simpler variable bindings to produce a variable binding that holds bindings for variables both from e_1 and e_2. We should recall that e_2 has to be evaluated in a context that binds variables defined in e_1 to specific values.

The glue operator is similar to join. The result of $[\![e_1]\!] \otimes [\![e_2]\!] = \langle T, \prec_T, \theta_T \rangle$ is defined as follows. $\mathbb{T}_1 = \mathcal{T}[\![e_1]\!]C$ represents variable bindings defined by e_1, $\mathbb{T}_2$ represents variable bindings defined by e_2. As e_2 is evaluated in a context with variables of e_1 already bound, $\mathbb{T}_2\langle\vec{x}\rangle = \mathcal{T}[\![e_2]\!]C(\Theta_{\mathbb{T}_1} \rightarrow \langle\vec{x}\rangle)$ is functionally dependent on a specific variable binding. Resulting tuples preserve the order of the tuples in $\mathbb{T}_1$ and $\mathbb{T}_2$ in that order.

$$
\begin{array}{rcl}
T & = & \{ \langle\vec{x},\vec{y}\rangle \mid \langle\vec{x}\rangle \in \mathcal{I}_{\mathbb{T}_1} \\
 & & \qquad\qquad\quad \langle\vec{y}\rangle \in \mathcal{I}_{\mathbb{T}_2\langle\vec{x}\rangle} \} \\
\prec_T & = & \{ \langle t_1, t_2\rangle \mid t_1 = \langle\vec{x_1},\vec{y_1}\rangle, t_2 = \langle\vec{x_2},\vec{y_2}\rangle \in T \\
 & & \qquad\qquad\quad \langle\vec{x_1}\rangle \sqsubseteq_{\mathbb{T}_1} \langle\vec{x_2}\rangle \\
 & & \qquad\qquad\quad \langle\vec{x_1}\rangle = \langle\vec{x_2}\rangle \text{ implies } \langle\vec{y_1}\rangle \sqsubset_{\mathbb{T}_2\langle\vec{x_1}\rangle} \langle\vec{y_2}\rangle \} \\
\theta_T & = & \Theta_{\mathbb{T}_1} \circ \Theta_{\mathbb{T}_2\langle\vec{x}\rangle} \text{ for any } \langle\vec{x}\rangle
\end{array}
$$

The tuple glue operator is used only to simplify definition of $\mathcal{T}$ in Figure 19, and appears at no other place in this thesis.

For example, an expression `$a in (10, 20), $b in (1, 2)` that is a subexpression of some for expression. It is evaluated as follows. Tuple stream of this expression is defined with a glue operator to be $[\![$`$a in (10, 20)`$]\!] \otimes [\![$`$b in (1, 2)`$]\!]$, which evaluates to the following.

$$
\begin{array}{rcl}
\mathbb{T} & = & (\langle 10,1\rangle, \langle 10,2\rangle, \langle 20,1\rangle, \langle 20,2\rangle) \\
\Theta_{\mathbb{T}} & = & (\mathtt{a},\mathtt{b})
\end{array}
$$

In this example, the result is the cartesian product, as the expression of variable $b is not functionally dependent on a value of variable $a. Another example with functionally dependent is subexpression `for $cd in cd, $a in $cd/author`, which creates 2-tuples, where the first position in a tuple refers to a CD, and the second position in a tuple refers to an author of that CD.

Where Clause. When the block of initial for and let clauses is processed, we get a stream that is filtered by an expression from a where clause as depicted on the last but one line in Figure 19. We take each tuple and bind its values to their respective variables. In such a context, we evaluate a boolean value of the expression from the where clause. This is realized with semantic function $\mathcal{W}$ that we defined in section 4.3 on XPath formal semantics. If it evaluates to true, the tuple is left in the stream, if it is false, the tuple is removed.

Order By Clause. The last line in Figure 19 defines the semantics of an order by clause. We can see that we take the tuple stream filtered by the where clause and that the result contains the same tuples and the same variable index.

$\mathcal{O}_{\mathbb{T}}[\![e]\!]C$	$\{ \langle t_1, t_2 \rangle \mid t_1, t_2 \in \mathcal{I}_{\mathbb{T}}$ $y_1 = \mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \rightarrow t_1)$ $y_2 = \mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \rightarrow t_2)$ $y_1 \leq y_2$ $y_1 = y_2$ implies $t_1 \sqsubset_{\mathbb{T}} t_2 \}$
$\mathcal{O}_{\mathbb{T}}[\![e, \text{OList}]\!]C$	$\{ \langle t_1, t_2 \rangle \mid t_1, t_2 \in \mathcal{I}_{\mathbb{T}}$ $y_1 = \mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \rightarrow t_1)$ $y_2 = \mathcal{E}[\![e]\!]C(\Theta_{\mathbb{T}} \rightarrow t_2)$ $y_1 \leq y_2$ $y_1 = y_2$ implies $\langle t_1, t_2 \rangle \in \mathcal{O}_{\mathbb{T}}[\![\text{OList}]\!]C \}$

Figure 20: XQuery ordering semantics.

The order of tuples is defined with semantic function $\mathcal{O}$ that is defined in Figure 20. It takes a tuple stream $\mathbb{T}$ and a comma-separated list of order expressions and evaluates it in the given context. The result is an order of tuples in the given tuple stream $\mathbb{T}$, which we represent as a binary relation on tuples.

$$\mathcal{O} \; : \; OrderExpr \rightarrow Context \rightarrow Tuples_n \rightarrow Set(\langle Tuple_n, Tuple_n \rangle)$$

The first line defines an order for two tuples t_1, and t_2 based on a single order expression e. It says that first value y_1 of e is computed in a context with variables bound to values from t_1, and second value y_2 of e is computed in a context with variables bound to values from t_2. If y_1 is less then y_2 then t_1 precedes t_2 in the ordered stream. If the two values are equal, then the order of t_1 and t_2 in the initial stream is preserved.

The second line inductively defines an order of t_1, and t_2 in case of multiple order expressions. First, we compute values y_1, and y_2 as in the previous case. If y_1 is less then y_2 then t_1 precedes t_2 in the ordered stream, so far the same. But if the two values are equal, then the order is defined by the rest of the order expressions.

This definition ensures that if for two tuples all computed values of all order expressions are equal, they preserve their order from the initial stream in the ordered stream.

5.3.2 Prolog

In this section, we provide the semantics of an XQuery query prolog. A prolog is used to define so called static context for query evaluation. We restricted it to contain definitions of functions and initialization of variables. See section 5.2 on XQuery syntax on what and why is removed from a general prolog.

We define the following function $\mathcal{P}$ that according to a prolog modifies a given context. It uses a helper function $\mathcal{L}$ that transforms a comma-separated list of argument names to a sequence of names.

$$\begin{aligned} \mathcal{P} \; &: \; Prolog \rightarrow Context \rightarrow Context \\ \mathcal{L} \; &: \; ArgList \rightarrow Seq(VarName) \end{aligned}$$

$\mathcal{P}[\![\texttt{declare variable}\ v := e]\!]C$	$\rho' = extend(v, \mathcal{E}[\![e]\!]C, \rho)$	
$\mathcal{P}[\![\text{SimpleVarDef VariableDefs}]\!]C$	$C' = \mathcal{P}[\![\text{VariableDefs}]\!]\mathcal{P}[\![\text{SimpleVarDef}]\!]C$	
$\mathcal{P}[\![\texttt{declare fn}\ name(\text{ArgList})\{e\}]\!]C$	$F'(n) = \begin{cases} \langle \mathcal{L}[\![\text{ArgList}]\!], e\rangle & \text{if } n = name \\ F(n) & \text{otherwise} \end{cases}$	
$\mathcal{P}[\![\text{SimpleFnDef FunctionDefs}]\!]C$	$F' = \mathcal{P}[\![\text{FunctionDefs}]\!]\mathcal{P}[\![\text{SimpleFnDef}]\!]C$	
$\mathcal{P}[\![\text{FunctionDefs VariableDefs}]\!]C$	$C' = \mathcal{P}[\![\text{VariableDefs}]\!]\mathcal{P}[\![\text{FunctionDefs}]\!]C$	
$\mathcal{L}[\![p]\!]$	$\langle\{p\}, \emptyset\rangle$	
$\mathcal{L}[\![p, \text{ArgList}]\!]$	$\mathcal{L}[\![p]\!] \circ \mathcal{L}[\![\text{ArgList}]\!]$	

Figure 21: XQuery prolog semantics for context $C = \langle \rho, F, item, pos, last\rangle$. Each transformation results in context $C' = \langle \rho', F', item, pos, last\rangle$. We provide only the affected parts of C'.

Function $\mathcal{P}$ is defined in Figure 21. Notice, that an expression from a variable declaration is evaluated in the context of all previously defined functions and variables. Also notice, that only the variable binding function ρ and the function definitions F are changed in the context.

5.3.3 Semantic Functions Relationships

It is quite interesting to inspect the relations among the semantics functions that we provide. Figure 22 depicts what semantic functions can be called by which semantic function. A possible call is depicted by an arrow from the caller to the callee. We can notice that each function that is calling the main semantic function $\mathcal{E}$ is also called by $\mathcal{E}$, except the ordering semantic function $\mathcal{O}$. This figure clearly depicts that ordering semantics is defined on tuples.

5.4 Sorting in XQuery

In this section, we inspect sorting semantics of XQuery, explaining inconsistency of its definition in W3C's XQuery formal semantics [22]. We explain that consistency can be gained again if we remove the order by clause from XQuery Core, not affecting the expressive power of the language. This is a consequence of the ability of XPath 2.0 to sort sequences.

To demonstrate sorting in XQuery, we use the following query in examples. It returns pairs of author and artist who do not perform on the same cd in their respective roles. Suppose, that the dist() function gets a sequence of authors or artists and returns the same sequence with duplicates removed, so e.g. dist(//author) returns each author only once.

```
for $auth in dist(//author),
    $art in dist(//artist)
```

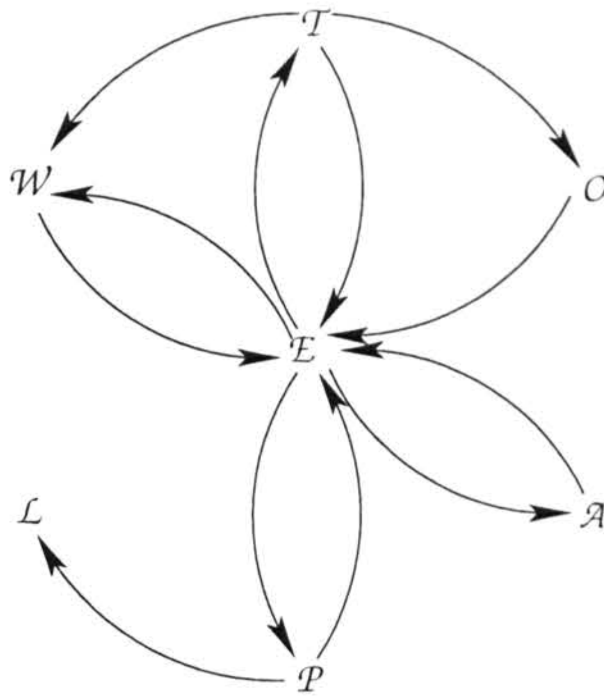

Figure 22: What XQuery semantic function can be called by which semantic function. An arrow leads from the caller to the callee.

```
where not($auth[parent::*/artist[last=$art/last and
                                 first=$art/first]])
order by $auth/last, $art/last, $auth/first
return
    <not-together>{ $auth, $art }</not-together>
```

The resulting pairs are ordered first by the last name of authors and for the same last name of authors, it is sorted by the last name of artists. If even the last name of artists are the same, then we sort according to first name of an author. This defines somewhat artificial sorting, but we need this kind of sorting to demonstrate insufficiency of XQuery Core definition.

The result to this query can look like the one in the following listing.

```
...
<not-together>
    <author>
        <first>Jaco</first>
        <last>Pastorius</last>
    </author>
    <artist>
        <first>Joni</first>
        <last>Mitchel</last>
    </artist>
</not-together>
...
```

The not-together tag encapsulates a single pair of an author and an artist. If the above pair appears in the result, it means that Joni Mitchel never performed as an artist on a cd authored by Jaco Pastorius.

5.4.1 XQuery Core Inconsistency

Since XQuery allows a lot of syntactic sugaring, a subset called XQuery Core has been extracted from it, on which formal semantics has been defined. We can transform an XQuery query to an XQuery Core query through a process called *normalization*. The Core language is supposed to be equally expressive as the full language, but one can easily find out that it is not. It is sorting, that one cannot express in XQuery Core.

While the grammar of XQuery Core defines a nonterminal for an order by clause, there is no grammar production that refers to it. Further, the normalization of an order by clause of an XQuery query is also omitted in the specification, mentioning only that a data type for a tuple would have to be introduced to define the formal semantics of sorting and since a tuple is not in the data model, the specification does not define the formal semantics of sorting at all.

Let's check it on our example. If we normalize the example FLWOR query according to the normalization rules, we end up with two nested queries, one that iterates over authors, and the other that iterates over artists. The original return clause is the return clause of the innermost FLWOR.

```
for $auth in dist(//author)
return
    for $art in dist(//artist)
    where ...
    return
        <not-together>{ $auth, $art }</not-together>
```

Notice, that we removed the order by clause. In fact, we have to, because it cannot appear neither in the outer FLWOR nor in the inner FLWOR. It cannot appear in the outer FLWOR expression, since the outer FLWOR expression has no knowledge about artists bound to $art variable. Further, it cannot appear in the inner FLWOR expression, since the inner FLWOR expression is evaluated in the context that has an author bound to a variable $auth, so it cannot sort among all authors.

Moreover, the order by clause cannot be split to be partly in the outer FLWOR and partly in the inner FLWOR. This is because the only meaningful split, which we depict in the following listing, does not define the same order of the result. It sorts according to the author's first name prior to sorting according to the artist's last name, which is not the same as we defined in the original query.

```
for $auth in dist(//author)
order by $auth/last, $auth/first
return
    for $art in dist(//artist)
    where ...
    order by $art/last
    return
        <not-together>{ $auth, $art }</not-together>
```

Summing up, we can see that the process of FLWOR normalization is not sufficient, as the order by clause is lost. Further, it can be concluded from the above demonstrated reasons that the order by clause *has to be removed* from the normalized FLWOR expression.

5.4.2 The Idea

On the other hand, we introduced an XPath 2.0 expression $Sort_R$ in section 4.4 that can sort arbitrary sequences. The only constraint is that the required resulting order is defined in terms of a partial order R with characteristic function expressible in XPath 2.0.

Since XPath 2.0 is a subset of XQuery with equal semantics in both languages, we can use the sorting expression in XQuery as well. Naturally, the constraint for a partial order can be extended to a partial order with a characteristic function that is expressible in XQuery.

Further, when examining the formal semantics of sorting within a FLWOR expression, we can see that the semantics is based on partial order, which we prove later. To remind semantics of sorting with the order by clause see section 5.1.2 for informal definition or section 5.3 for formal definition.

Putting these together, it seems now that it could be possible to sort a sequence generated by a FLWR expression without the order by clause with help of the $Sort_R$ expression. The only question remains. Can we simulate the tuples? The answer is yes. XQuery provides constructor expressions to build an XML document, so it can construct a tree. Therefore we can use it build a simpler structure – a tuple.

5.4.3 Formally

In this section, we formally show that each FLWOR XQuery expression can be rewritten to an equal one that contains no order by clause.

First, we define helpful expressions that cut a FLWOR expression to pieces.

Definition 5.1 *Let e be a FLWOR expression, and let $o_1, \ldots, o_n$ be a list of expressions from the order by clause of e, and r be an expression in the return clause of e. Then we define the following.*

i) $FLW(e)$ denotes a FLW expression that we get by removing the order by and return clauses from e.

ii) $FLWR(e)$ denotes a FLWR expression that we get by removing the order by clause from e.

iii) $OList(e)$ denotes the list $o_1, \ldots, o_n$ of expressions from the order by clause of e.

iv) $Return(e)$ denotes the expression r from the return clause of e.

Now, we handle the most important part – creation of tuples. In the following three definitions we define expression *Wrap* that rewrites a given FLWOR expression to a more decorated expression using two helper expressions: *Stream* and *Tuples*.

First, we define expression *Stream*, which strips the order by clause from the initial expression. It also extends the return clause to return a tuple that consists of order values in ordval tags and the initial return expression in a result tag. Each order value represents a value of an expression from the initial order by clause, where the order of order values in the resulting tuple corresponds to the order of expressions from the order by clause. The modified FLWOR expression is rooted with a stream tag to fix the order of tuples, otherwise there would be no guarantee about the order.

Definition 5.2 *Let e be a FLWOR expression, and let OList(e)* $= o_1, \ldots, o_n$. *Then with Stream(e) we denote the following XQuery expression.*

```
<stream>
{
    FLW(e)
    return
      <tuple>
        <ordval>{ o_1 }</ordval>
        ...
        <ordval>{ o_n }</ordval>
        <result>{ Return(e) }</result>
      </tuple>
}
</stream>
```

Next, we provide a simple expression whose only task is to strip the stream tag and retain the order of tuples.

Definition 5.3 *Let e be an XQuery expression. Then with Tuples(e) we denote the following expression.*

```
for $t in (e)/tuple
return $t
```

Finally, we define the *Wrap* expression to be the modified FLWOR expression rooted with a stream tag, as defined with *Stream* expression, but with the root stream tag immediately stripped away with *Tuples* expression. As we noted above, we have to provide the root stream tag because without it the order of tuples is not defined. But as we want the result of the *Wrap* expression to be a sequence of tuples and not a singleton sequence of the stream tag, we strip the stream tag off.

Definition 5.4 *Let e be a FLWOR expression. Then with Wrap(e) we denote an XQuery expression Tuples(Stream(e)).*

In the following definition, we define expression *Unwrap*, which extracts the result of the initial return expression that was previously wrapped with *Wrap*.

Definition 5.5 *Let e be an XQuery expression. Then with Unwrap(e) we denote the following expression.*

```
for $t in e
return $t/result/*
```

Notice, that neither *Wrap* nor *Unwrap* reorders the sequence. Thus, the order of the resulting sequence of the *Wrap(e)* expression is imposed by the order of variable bindings in *FLW(e)*. For the *Unwrap(e)* expression the order of the resulting sequence of is imposed by the order of tuples in *e*.

Further, the return expression is in both initial and wrapped expressions evaluated in the same context. Thus, the following holds.

Lemma 5.1 *Let e be a FLWOR expression. Then the following holds.*

$$\mathcal{E}[\![Unwrap(Wrap(e))]\!]C = \mathcal{E}[\![FLWR(e)]\!]C$$

Proof. This is proved informally in the text above this lemma. □

Next, we define an expression that extracts an i-th order value from a given tuple.

Definition 5.6 *Let x be an XQuery expression, and i be an integer. Then with $Oval_i(x)$ we denote the following XQuery expression.*

```
x/ordval[i]
```

In the following two definitions, we define partial order R_e that we use later to sort the wrapped sequence with the *Sort* expression. The *Sort* expression is defined in section 4.4 on sorting in XPath.

As required, the R_e relation has to be expressible in XQuery with a characteristic function. Therefore, we first define characteristic function χ_1^e and than R_e.

Function χ_1^e tells whether one tuple should precede another tuple in the resulting sequence depending on the list of expressions in the order by clause of the initial FLWOR expression. It is a straight formalization of lexicographic sorting semantics as it is defined informally in XQuery 1.0 specification [8]. First, we take the first expression in the expression list from the order by clause. If the order value of this expression for tuple x is less than the order value from tuple y, than we are finished because x precedes y in the resulting tuple stream. If the two values are equal, we have to check the next expression from the order by clause, and so on.

Definition 5.7 *Let e be a FLWOR expression with $OList(e) = o_1, \ldots, o_n$. Let x and y be XQuery expressions, and let i be an integer such that $1 \leq i \leq n-1$. Then with $\chi_i^e(x,y)$ we denote the following XQuery expression that is defined recursively.*

$$\begin{aligned}
\chi_i^e(x,y) &=_{\text{def}} Oval_i(x) \leq Oval_i(y) \text{ and } (Oval_i(x) = Oval_i(y) \rightarrow \chi_{i+1}^e(x,y)) \\
\chi_n^e(x,y) &=_{\text{def}} Oval_n(x) \leq Oval_n(y)
\end{aligned}$$

Definition 5.8 *Let e be a FLWOR expression. Then with R_e we denote a binary relation with characteristic function χ_1^e.*

Lemma 5.2 *Let e be a FLWOR expression. Then binary relation R_e is a partial order on $\mathcal{I}_{\mathcal{E}[\![Wrap(e)]\!]C}$.*

Proof. As R_e is defined to be χ_1^e, we prove this lemma by decreasing induction on i in χ_i^e.

Let's take $i = n$ for the initial step. Then $\chi_n^e(x,y)$ is defined as $Oval_n(x) \leq Oval_n(y)$, which is obviously a partial order condition.

Suppose that χ_{i+1}^e is a characteristic function of a partial order relation and let's check whether even χ_i^e defines a partial order. A relation is a partial order if it is reflexive, antisymmetric and transitive. We check each property separately.

Reflexivity. Let's check $\chi_i^e(x,x)$.

$$Oval_i(x) \leq Oval_i(x) \texttt{ and } (Oval_i(x) = Oval_i(x) \rightarrow \chi_{i+1}^e(x,x))$$

Evaluates to the following, which is obviously true.

$$true \texttt{ and } (true \rightarrow true)$$

Thus, $\chi_i^e(x,x)$ holds for each x and reflexivity is satisfied.

Antisymmetry. To prove antisymmetry, we have to show that the following holds.

$$\chi_i^e(x,y) \;\&\; \chi_i^e(y,x) \rightarrow Oval_i(x) = Oval_i(y)$$

From the construction of χ_i^e we can see that both $\chi_i^e(x,y)$ and $\chi_i^e(y,x)$ are true only if $Oval_i(x) = Oval_i(y)$. Thus, antisymmetry is satisfied.

Transitivity. To prove transitivity, we have to show that the following is satisfied.

$$\chi_i^e(x,y) \;\&\; \chi_i^e(y,z) \rightarrow \chi_i^e(x,z)$$

This comes trivially from the construction of χ_i^e.

Since χ_i^e represents a partial order, even χ_1^e and so even R_e represent a partial order. □

Theorem 5.1 *Let e be a FLWOR expression. Then the following holds.*

$$\mathcal{E}[\![Unwrap(Sort_{R_e}(Wrap(e)))]\!]C = \mathcal{E}[\![e]\!]C$$

Proof. We should recall that $Wrap(e)$ returns the same results as $FLWR(e)$ with the only distinction that there is more information attached to each result. The order is the same as well.

Further, R_e is a partial order relation by definition expressible with an XQuery expression, therefore even $Sort_{R_e}$ is an XQuery expression. Since semantics of XPath 2.0 are the same for XQuery, expression $Sort_{R_e}$ applied to a sequence $\langle S, \prec_S \rangle$ returns this sequence sorted with R_e. So, $Sort_{R_e}(Wrap(e))$ returns a sequence of tuples sorted with R_e.

Finally, since *Unwrap* expression preserves the order of results extracted from a sequence of tuples, we end up with a same sequence of results as if we evaluated expression e. □

Collorary 5.1 *Let e be a FLWOR expression. Then e can be rewritten to an equal expression with no order by clause.*

Proof. Since none of *Unwrap*, $Sort_{R_e}$, and *Wrap* XQuery expressions contains an order by clause, we can use the expression of 5.1 to get an equal expression with no order by clause. If there is a subexpression of e that is a FLWOR expression then the same transformation can be used. □

5.4.4 By Example

This section demonstrates the transformation of the original query introduced on page 57. So, we want to write an equivalent query to the following one. We label it *e*.

```
for $auth in dist(//author),
    $art in dist(//artist)
where not($auth[parent::*/artist[last=$art/last and
                                 first=$art/first]])
order by $auth/last, $art/last, $auth/first
return
    <not-together>{ $auth, $art }</not-together>
```

First, we modify the query according to the *Wrap* expression. The following is the listing of $Wrap(e) = Tuples(Stream(e))$. We bind the intermediate result to variable $stream, and the result of expression $Wrap(e)$ to variable $wrappedResults. Since the where clause is not of much importance to us, we simplify it in the listing.

```
let $stream :=
  <stream>{
    for $auth in dist(//author),
        $art in dist(//artist)
    where ...
    return
      <tuple>
        <ordval>{ $auth/last }</ordval>
        <ordval>{ $art/last }</ordval>
        <ordval>{ $auth/first }</ordval>
        <result>{
          <not-together>{ $auth, $art }</not-together>
        }</result>
      </tuple>
  }</stream>
let $wrappedResults :=
  for $t in $stream/tuple
  return $t
```

Thus, we get a sequence of tuple elements for each variable binding defined by the FLW part of the original FLWOR query. Each tuple element comprises a sequence of ordval elements and a single result element. Each ordval element contains a value for the ordering expression from the order by clause. We can see that there are three ordval elements constructed for our example query and that their order reflects the order of the ordering expressions in the order by clause. Last, the return expression is exactly the same as the return expression of the original query.

Notice, that values in the ordval elements are evaluated in the same context as the result element, as required by the specification.

So, we have tuples now that contain a result and all order values that are needed to sort the resulting sequence. Next, we define the characteristic function of the sorting relation R for $OList(e)$ comprising of the following ordering expressions in that order: $auth/last, $art/last, $auth/first. We provide the

characteristic function for $R(x,y)$ in the listing below, according to the $\chi_i^e(x,y)$ function defined in Definition 5.7. We replace the implication $a \rightarrow b$ with equal expression $\neg a \vee b$.

```
Oval1(x) <= Oval1(y) and (
Oval1(x) != Oval1(y) or (
  Oval2(x) <= Oval2(y) and (
  Oval2(x) != Oval2(y) or (
    Oval3(x) <= Oval3(y)
  ))
))
```

Next, we unfold the $Oval_i(x)$ expressions to get the following.

```
x/ordval[1] <= y/ordval[1] and (
x/ordval[1] != y/ordval[1] or (
  x/ordval[2] <= y/ordval[2] and (
  x/ordval[2] != y/ordval[2] or (
    x/ordval[3] <= y/ordval[3]
  ))
))
```

Now, we can use the $Sort_R$ expression, where R is the just defined partial order, to sort the sequence of tuples.

```
let $sortedTuples := SortR($wrappedResults)
```

Prior to unfolding this expression, we present the characteristic function of total order lt_R that is used inside the $Sort_R$ sorting expression instead of partial order R. Definition of lt_R is provided in Definition 4.2 on page 36.

```
x/ordval[1] < y/ordval[1] or (
x/ordval[1] = y/ordval[1] and (
  x/ordval[2] < y/ordval[2] or (
  x/ordval[2] = y/ordval[2] and (
    x/ordval[3] < y/ordval[3]
  ))
))
```

Next, we present the sorting expression unfolded.

```
let $sortedTuples :=
  for $i in (0 to count($wrappedResults) - 1)
  return
    for $x in $wrappedResults
    return
      if ($i = count(
            for $y in $wrappedResults
            return
              if ($y/ordval[1] < $x/ordval[1] or (
                  $y/ordval[1] = $x/ordval[1] and (
                    $y/ordval[2] < $x/ordval[2] or (
                    $y/ordval[2] = $x/ordval[2] and (
                      $y/ordval[3] < $x/ordval[3]
                    ))
```

```
                ))
            )
            then $y
            else ()
        )
    )
    then $x
    else ()
```

Thus, variable $sortedTuples is bound to a sequence of tuples sorted according to the original order by clause. We need to extract the contents of result elements from the tuples now, which is exactly what the *Unwrap* expression does.

```
for $r in $sortedTuples/tuple/result
return $r/*
```

Finally, we present the whole rewritten query in a single listing.

```
let $stream :=
  <stream>{
    for $auth in dist(//author),
        $art in dist(//artist)
        where not($auth[parent::*/artist[last=$art/last and
                                         first=$art/first]])
    return
      <tuple>
        <ordval>{ $auth/last }</ordval>
        <ordval>{ $art/last }</ordval>
        <ordval>{ $auth/first }</ordval>
        <result>{
          <not-together>{ $auth, $art }</not-together>
        }</result>
      </tuple>
  }</stream>
let $wrappedResults :=
  for $t in $stream/tuple
  return $t
let $sortedTuples :=
  for $i in (0 to count($wrappedResults) - 1)
  return
    for $x in $wrappedResults
    return
      if ($i = count(
            for $y in $wrappedResults
            return
              if ($y/ordval[1] < $x/ordval[1] or (
                  $y/ordval[1] = $x/ordval[1] and (
                    $y/ordval[2] < $x/ordval[2] or (
                    $y/ordval[2] = $x/ordval[2] and (
                      $y/ordval[3] < $x/ordval[3]
                    ))
                  ))
            )
```

```
                then $y
                else ()
          )
      )
      then $x
      else ()
for $t in $sortedTuples
return $t/*
```

5.4.5 Conclusion

First, we explained that XQuery Core is defined in a cumbersome way. It is claimed to have the ability to express all XQuery expressions and it provides normalization rules that translate XQuery expressions to XQuery Core, but there is no normalization rule to translate FLWOR expression with an order by clause. Moreover, authors of the specification thought that semantics of an order by clause is not expressible in the data model that is used.

In this section, we proved that, unlike the authors of the formal semantics specification thought, it is possible to express sorting semantics of an order by clause while not extending the data model. We made use of the fact that XPath itself can sort sequences to prove it. Sorting capability of XPath is proved in the previous chapter in section 4.4.

As a result we get a cleaner XQuery Core. Our proof is in fact a sort of normalization rule: it tells how a general FLWOR expression can be rewritten to an equal one without an order by clause. Therefore, a grammar production for an order by clause can be removed from XQuery Core syntax, which makes the resulting core consistent.

Not only the formal proof is presented in this section, it is also an example rewrite of an FLWOR expression to an equivalent expression without an order by clause.

5.5 Conclusion

In this chapter, we focused on XQuery as an XML query language that extends XPath. We introduced expressions that are not present in XPath and demonstrated them on examples.

Further, we provided formal semantics in a model that recognizes a tuple as a structure. This allowed a straightforward formalization of sorting semantics of an order by clause from a FLWOR expression, which was missing in W3C's XQuery formal specification.

Then, we inspected sorting in XQuery. We found that we can make use of XPath's sorting ability, which we proved in the previous chapter, to provide a transformation from a general FLWOR expression to an equal expression without an order by clause.

This result fills the gap in XQuery Core definition that misses such a transformation. The interesting point is that the authors of this specification thought it infeasible to define sorting semantics in the data model used. We showed that unlike the authors thought this is possible.

6 XSLT

This chapter inspects XSLT 2.0 – a language designed to transform an XML document to another XML, HTML, SVG, or any other text document.

XSLT is a declarative language that originated from an effort to develop an eXtensible Stylesheet Language for XML (XSL). The effort has been started by W3C. Its goal was to design a platform- and media-independent formatting language, based on DSSSL [35]. As the time passed, former XSL split into two parts: XSL Transformation (XSLT) [18], which is a simple language that declaratively describes a transformation of an input XML document, and XSL Formatting Objects (XSL-FO) [3], which defines an XML vocabulary for specifying formatting semantics.

XSLT 1.0 [18], since 1999 a W3C Recommendation, has been designed by James Clark and nowadays it is widely accepted by the world-wide web and documents oriented community. There is a huge number of XSLT processors that implement this specification. XSLT 1.0 heavily relies on XPath 1.0, which in fact has been developed as part of XSLT 1.0. These two languages share the same data model that is based on node sets with primitive typing.

XSLT 2.0 [40], now a W3C Last Call Working Draft, has been designed by Michael Kay and it greatly extends its predecessor. The main distinction is an improved data model that is based on sequences of nodes and that adopts the XML Schema [23, 54, 7] typing system. The improved data model, described in section 2.2 on W3C standards on page 7, is shared together with XPath 2.0 and XQuery 1.0.

The aim of this chapter is to provide XSLT Core – a language that uses as smallest set of instructions as possible and that has an expressive power of XSLT. For this language we define the formal semantics.

This chapter is organized as follows. First, we provide an introduction to XSLT in section 6.1. Our own non-XML syntax for XSLT is then defined in section 6.2. Then, in section 6.3, we look for the core language of XSLT, for which we define the formal semantics in section 6.4. Conclusions are given in section 6.5.

6.1 Introduction

Here, we provide an introduction to XSLT 2.0. We choose the most of language constructs and explain them in detail and demonstrate on examples.

6.1.1 Stylesheet and Template

An XSLT transformation is provided as an XML document called a *stylesheet*, or XSLT stylesheet. We say that a stylesheet is applied to an input XML document, which means that the input XML document is transformed according to the stylesheet into an output document. The result of a transformation is generally a text document, but usually XML documents are produced.

A stylesheet can be looked at as a set of rewriting rules. Each rewriting rule is equipped with a *patten* and a *body expression*. When a stylesheet is applied to an input XML document, a rewriting rule is found, whose pattern matches the root document node of the input XML document. This rule's body expression

defines a transformation for the root node, so the rule's body expression is evaluated to compute the output.

In XSLT, a rewriting rule is called a *template rule*. A pattern of a template rule is an XPath 2.0 expression, but not all XPath expressions are patterns. Generally a pattern is restricted to use only child and attribute axis. It is defined that a pattern matches a node, if the sequence that results from evaluating the XPath expression of the pattern contains the node.

A body expression of a template rule contains a *sequence constructor*, which is a list of XSLT instructions. The power of XSLT inheres in the set of instructions it provides. There are instructions that evaluate an XPath expression, instructions that copy nodes from an input XML document to the output, instructions that output new nodes, instructions that iterate over a sequence of nodes, grouping instructions, etc. Each instruction returns a sequence of items. A sequence constructor is evaluated so that each its instruction is evaluated and the result is a concatenation of these sequences. So, the result of a sequence constructor is again a sequence of items.

The following is an example of a simple template rule.

```
<xsl:template match="/">
    <message>This will be output for a document root.</message>
</xsl:template>
```

Its pattern is an XPath expression that matches the root document node. Its body expression contains only an element node labeled message, which will be output, when a document root node matches.

When a stylesheet consisting of only the rule above is applied to an XML document, the result will be an XML document with a root node labeled message with a text node with "This will be output for a document root." text in it.

Notice the syntax of a template rule. A template rule is defined with an xsl:template element with a pattern in its match attribute. The body expression is the contents of an xsl:template element. As XSLT defines a lot of tags, we want to distinguish XSLT elements from other XML element such as the message tag in the example above that is not an XSLT instruction but rather it constructs a new element in the output. Therefore we start every XSLT element with an *xsl* namespace prefix.

XSLT elements can be roughly divided into i) *top-level* elements, and ii) *instructions*. Top-level elements like xsl:template are allowed to be directly put into a stylesheet body, while instructions can be used only inside some of the top-level elements.

6.1.2 Applying Templates

It would not be much useful, if only a root node of an XML document could be matched against a sequence of template rules. XSLT is designed to walk through an XML document recursively. This is achieved with an instruction *xsl:apply-templates* that identifies nodes, for which template rules are applied. Let's check the following stylesheet.

```
<xsl:template match="/archive">
    <cd-list>
```

```
        <xsl:apply-templates/>
    </cd-list>
</xsl:template>

<xsl:template match="cd">
    <cd>
        <xsl:value-of select="title"/>
    </cd>
</xsl:template>
```

This stylesheet returns an XML document with a root element labeled cd-list, whose contents comprises a cd element for each CD in an archive, where each cd element contains a textual node with a title of that CD.

Notice the xsl:apply-templates instruction in the first template that selects all children of the current node to be processed. In this case, it selects all children of the root node, which are cd elements. For each cd element a matching template rule is found. Each cd element matches the second template rule, as the pattern of a template rule is evaluated in the context of the currently processed element.

The *xsl:value-of* instruction in the body of the second template returns a text node. The text it contains is determined with an XPath expression in its select attribute. In our example, it returns the textual value of an element node labeled title that is a child of the currently processed cd element.

Further, it has not to be only children of the current node that can be selected for template application. The xsl:apply-templates instruction has an optional select attribute to identify such a sequence of nodes. For example, the following rule matches every cd element, where template rules are applied only to authors of the matched cd element.

```
<xsl:template match="cd">
    <cd>
        <xsl:apply-templates select="author"/>
    </cd>
</xsl:template>
```

6.1.3 Context

The evaluation context comprises i) focus, and ii) additional variables.

The focus is the same as when XPath or XQuery query is evaluated. It consists of i) *context item*, which is an item that is currently being processed, ii) *context position*, which is a position of context item within the sequence of items currently being processed, and iii) *context size*, which is a number of items within the sequence of items currently being processed.

Focus is changed with instructions like xsl:apply-templates or xsl:for-each. The context components item, position, and size can be retrieved by XPath expression . (dot), position(), and last(), respectively.

The additional variables in the context keep additional information about a stylesheet currently being processed, like the current mode, the current group and grouping key, and some not so important other variables. Additional variables are discussed in detail later in section 6.3.5 on page 105.

6.1.4 Built-in Template Rules

It can happen that no template rule matches the current context item. In such a case, built-in template rules come to play their part. A built-in rule is defined for every node kind.

For a document or element node, a built-in template rule calls the xsl:apply-templates instruction on the children of the current context item.

For a text or attribute node, a built-in rule simply copies the the context node to the output.

Otherwise, the result of applying a built-in rule is an empty sequence.

The built-in templates simulate a depth-first traversal of the input XML document for nodes that do not match any template rule.

6.1.5 Conflict Resolution

It can also happen that there is more than one template rule in a stylesheet, whose pattern matches the context item. Then, we talk about a *conflict*. A conflict is resolved with a complex algorithm called conflict resolution. It is quite a magic algorithm that considers generality of a pattern so that the more specific pattern like `cd` is preferred to a more general one like `node()`. Further, template rules can be given a numeric priority with a priority attribute, which is considered in conflict resolution. Lastly, the order of templates is considered thus templates at the beginning of a stylesheet have precedence to templates at the end of a stylesheet. The conflict resolution is the most tricky part of XSLT and it can take a lot of time to tune a stylesheet to work correctly.

The important thing about conflict resolution is that it guarantees to select always a single template rule. In fact, it provides a total order on a sequence of template rules and this order is defined statically. We make use of this later to get rid off template rules.

There is an *xsl:next-match* instruction that applies a template rule for the context item. The rule it applies is the rule whose pattern matches the context item and that is just behind this template rule in the total order mentioned in the previous paragraph. This means, that the selected rule has either a lower import precedence and/or priority.

6.1.6 Modes

In a complex stylesheet, we often need to process a document several times but with different actions taken for the same node. Suppose, we want to transform an XML document that contains a documentation, to an HTML document with a table of contents. In such a case, we need to go through the input document once to produce the table of contents and once to produce the HTML decoration. Therefore we need a template rule that matches e.g. a chapter to produce a link in a table of contents and one more rule that matches a chapter to produce its HTML formatted title and paragraphs.

```
<xsl:template match="/">
    ...
    <xsl:apply-templates/>
    ...
    <xsl:apply-templates mode="toc"/>
```

```
    ...
</xsl:template>

<xsl:template match="chapter">
    ...
</xsl:template>

<xsl:template match="chapter" mode="toc">
    ...
</xsl:template>
```

We can see that xsl:template and xsl:apply-templates instructions can be given a mode attribute. If xsl:apply-templates instruction is called with a mode attribute, then only templates with that specific mode are considered for matching. Thus, both calls to xsl:apply-templates in the example above process children of a root node. But, only templates with no mode set are checked for match in the first call, while only templates with a mode set to "toc" are checked in the second call.

Special Modes

Besides user defined modes, there are three special modes *#default*, *#current*, and *#all*.

When mode #default is specified, then it has the same meaning as if no mode was specified. This mode can be specified in xsl:apply-templates instruction and in a template rule.

When mode #current is specified, then the mode in which the current template is evaluated is supplied. It is allowed to use this special mode only in xsl:apply-templates instruction.

When mode #all is specified, which is possible only in template rules, then a template rule is considered for every xsl:apply-templates instruction, regardless of a mode specified in that instruction.

6.1.7 Parameters

It is possible to parametrize templates in XSLT. To parametrize a template, we have to provide a list of parameters in a template rule. The parameters are declared with a *xsl:param* instruction, like in the following example.

```
<xsl:template match="*">
    <xsl:param name="param1"/>
    <xsl:param name="param2">Default value for param2</xsl:param>
    <xsl:param name="param3" select="//cd"/>
</xsl:template>
```

In this example, we declared three parameters in a template rule. The first is named param1 and has no default value, i.e. its default value is an empty sequence. The second parameter is named param2 and its literally defined default value is a string "Default value for param2". The third parameter is named param3 and its default value is computed with an XPath expression in its select attribute.

The values of parameters passed to a template can be either constant or computed with an XSLT expression. If a parameter is declared in a template, but it is not passed, its default value is taken. If a parameter is passed that is not declared, it is ignored. Parameters are passed with an *xsl:with-param* instruction.

```
<xsl:apply-templates>
    <xsl:with-param name="param1">3</xsl:with-param>
    <xsl:with-param name="param2">
        <xsl:apply-templates select="//author"/>
    </xsl:with-param>
    <xsl:with-param name="param3" select="//title"/>
</xsl:apply-templates>
```

The first parameter param1 is passed a literal value 3. The second parameter param2 is passed a value that is the result of evaluation of the contained XSLT instruction, which applies templates for all authors. The third parameter param3 is passed a value computed with an XPath expression that looks for all titles.

XSLT does not allow parameter names to be computed with an expression, their names have to be provided literally.

Parameters are referenced to as XPath variables. For example, the following returns the last names of authors in parameter param2.

```
<xsl:value-of select="$param2/last"/>
```

Tunnel Parameters

In a complex stylesheet, one may want to pass a parameter from a top-level template to some deeply called template. In such a case, a stylesheet writer would have to add a parameter declaration to every template that might be walked through and she would have to add an instruction to every xsl:apply-templates call to pass that parameter's value to the target template.

Since this would complicate the stylesheet a lot, *tunnel parameters* are introduced to ease the work of a stylesheet writer and make the resulting stylesheet more readable. A parameter declared as tunnel is implicitly passed in every template call.

```
<xsl:template match="/a">
    ...
    <xsl:apply-templates select="b">
        <xsl:with-param name="tparam" tunnel="yes">
            some value
        </xsl:with-param>
    </xsl:apply-templates>
    ...
</xsl:template>

<xsl:template match="b">
    ...
    <xsl:apply-templates select="c"/>
    ...
</xsl:template>
```

```
<xsl:template match="c">
    <xsl:param name="tparam" tunnel="yes"/>
    ...
</xsl:template>
```

In this example, the first template matches the root node and applies templates to all b children. It passes a tunnel parameter named tparam. For b elements the second template is matched that applies templates to its c children. It does not pass any parameter explicitly, but our tunnel parameter tparam is implicitly passed. The third template is matched for all c elements and a value of tparam is set to a value that was set in the first template rule.

Notice, that a template has to declare that a parameter is tunnel, otherwise, it is supposed that a parameter is a usual parameter. If the last template rule does not declare tparam to be tunnel in the example above, then tparam is treated as a usual parameter and as such, it is not assigned the value of tunnel parameter tparam. To avoid a name clash between a usual parameter and a tunnel parameter of the same name, it is required that all usual and tunnel parameters have distinct names in a template or function.

6.1.8 Variables

XSLT allows to define variables and assign them any value within the data model, which means they generally contain sequences of items. The value of a variable can be defined either with an XPath expression or with a sequence constructor.

```
<xsl:variable name="var1" select="//author/last"/>
<xsl:variable name="var2">
    <xsl:apply-templates/>
</xsl:variable>
```

This code defines two variables with names var1 and var2, respectively. The value of var1 is defined with an XPath expression and contains a sequence of last names of all authors. The value of var2 is defined with a sequence constructor.

Variables are referred to the same way as parameters: with an XPath variable reference. The following can be used to apply templates for items in a sequence of variable var1.

```
<xsl:apply-templates select="$var1"/>
```

Notice, that variables can be defined both locally inside a sequence constructor, and globally as a top-level element. The difference is a scope of a variable. A locally defined variable is valid from the point of its definition to the end of the last following sibling of its xsl:variable instruction. A locally defined variable is not defined in a called template or function. On the other hand, a globally defined variable is defined throughout the whole stylesheet. It is an error to refer to a variable that is not declared in the scope of referring instruction.

6.1.9 Named Templates

So far, we talked only about template rules, but XSLT provides also a different type of a template: a named template. A named template differs from a template rule in that it has no pattern, it has a name instead. Therefore, an XSLT processor does not look among named templates for a match, when an xsl:apply-templates instruction is called. But, unlike a template rule, a named template can be called directly with an *xsl:call-template* instruction using its name.

```
...
    <xsl:call-template name="tpl"/>
...

<xsl:template name="tpl">
    ...
</xsl:template>
```

This code calls a template named tpl, which evaluates the sequence constructor in the body of tpl, which is then returned.

Even named templates can be passed parameters. The declaration of parameters and passing values to parameters is the same as for template rules and xsl:apply-templates instruction.

In fact, named templates are something like functions. They have a name, a list of parameters, and a body that is defined with a sequence constructor.

6.1.10 Functions

XSLT 2.0 provides an *xsl:function* instruction that can be used to define a function. Such a function can than be called from within an XPath expression. A function, like a named template, has a name, a list of parameters, and a body that is defined with a sequence constructor. The way of passing a parameter follows the XPath syntax and is therefore different from calling a named template.

```
...
    <xsl:value-of select="fn(1)"/>
...

<xsl:function name="fn">
    <xsl:param name="number"/>
    ...
</xsl:template>
```

This code calls an XPath function fn with one argument, which is set to 1. This results in evaluation of a sequence constructor in the body of fn definition, where the argument is assigned to parameter labeled number.

Let's mention the main differences between functions and named templates.

First, a function can be called only from within an XPath expression, while a named template can be called from within a block of XSLT instructions – a sequence constructor.

Second, function body is evaluated with an empty focus (context item, position, and size), which is exactly the semantics of an XPath call. For more detail

see the chapter on XPath. On the other hand, the body of a named template is evaluated with the current focus.

6.1.11 Sequence Iteration

The *xsl:for-each* instruction has a mandatory select attribute that defines a sequence with an XPath expression. This sequence, which is iterated over with the for-each instruction, is called an initial sequence. The instruction evaluates its body expression once for each item from the initial sequence, each time with the focus set to that item in the initial sequence.

Notice, that xsl:for-each is one of the few instruction that changes the evaluation context.

```
<ul>
    <xsl:for-each select="//author">
        <li>
            <xsl:value-of select="first"/>
            <xsl:value-of select="last"/>
        </li>
    </xsl:for-each>
<ul>
```

The code snippet above creates a bulleted list of authors in HTML, where each author is a separate item. Notice, that the XPath expressions in xsl:value-of instructions are evaluated in the context of each author node.

Further, the sequence can be sorted in the iteration. This is achieved by putting *xsl:sort* instructions just after the xsl:for-each instruction. The following sorts the authors according to the last names. Within authors with equal last name, it sorts according to their first names.

```
<ul>
    <xsl:for-each select="//author">
        <xsl:sort select="last"/>
        <xsl:sort select="first"/>
        <li>
            <xsl:value-of select="first"/>
            <xsl:value-of select="last"/>
        </li>
    </xsl:for-each>
<ul>
```

We refer to XPath expressions in the select attribute of xsl:sort instructions as to order specifications.

6.1.12 Conditional Execution

The *xsl:choose* instruction allows to branch an XSLT transformation. It is similar to a switch statement in C language. In its body it contains a list of *xsl:when* instructions, where each has a mandatory test attribute with an XPath expression. Optionally, it can contain an *xsl:otherwise* instruction.

When xsl:choose is evaluated, the processor looks for the first xsl:when, whose test expression evaluates to true. If it finds one, its body is evaluated

and returned as a result. If none is found and xsl:otherwise is present, its body is evaluated and returned. Otherwise, the result of xsl:choose is an empty sequence.

```
<xsl:for-each select="//cd">
    <xsl:choose>
        <xsl:when test="artist and author">
            <both><xsl:value-of select="title"/></both>
        </xsl:when>
        <xsl:when test="artist">
            <artist><xsl:value-of select="title"/></artist>
        </xsl:when>
        <xsl:when test="author">
            <author><xsl:value-of select="title"/></author>
        </xsl:when>
        <xsl:otherwise>
            <none><xsl:value-of select="title"/></none>
        </xsl:otherwise>
    </xsl:choose>
</xsl:for-each>
```

The code above iterates over all CDs. For each CD it evaluates the xsl:choose instruction, which looks for the first xsl:when instruction that evaluates its test expression to true. One by one they test for the presence of both artists and authors, at least an artist, and at least an author, respectively. If there is no author and no artist, xsl:otherwise is evaluated. Each time the result is a CD title, but it is enclosed inside an element with a different name: both, artist, author, and none, respectively.

Further, XSLT provides one more simple instruction *xsl:if* for conditional execution. It contains a test attribute with an XPath expression. If the test expression evaluates to true, the body of xsl:if is executed. Otherwise, evaluation of xsl:if results in an empty sequence. Notice, that there is no else branch.

```
<xsl:if test="artist and author">
    <both><xsl:value-of select="title"</both>
</xsl:if>
```

This code checks whether the context item is a node that has at least one artist and at least one author as a child node. In such a case it returns a title subnode enclosed in a tag labeled both, otherwise it returns an empty sequence.

6.1.13 Constructing New Content

XSLT provides two ways new content can be constructed: i) literal result construction, ii) attribute value templates, and ii) constructor expressions.

The first, *literal result*, is used when the newly constructed nodes are constant and do not depend on the input XML document. It is written with usual XML syntax. For example, the tag labeled both in the previous section is an example of a literal result element. The same way attributes, processing instructions, text, etc. can be constructed.

The second, *attribute value templates*, is a way to compute a value of an attribute with an XPath expression. The attribute is written with an XML syntax

and if its string value contains curly braces, then the content of curly braces is evaluated as an XPath expression, the result is stringified and replaces the expression in curly braces.

```
<cd title="{./title}"/>
```

The code above constructs a cd element with a title attribute, whose contents is set to the contents of a title child of the context item.

Notice, that attribute value templates can be used not only in literal result constructors, but also in a lot of attributes of XSLT instructions. But we have to be careful, as not all attributes of XSLT instructions allow attribute value templates in them. This is not very nice feature of the language.

The last, *constructor expression*, is used when the properties of the newly constructed nodes depend on the input XML document. For example, we can set an element name, conditionally set an attribute to an element, etc. Each of the seven node kinds has its own *constructor*, which is an XSLT instruction that constructs the specific node kind.

```
<xsl:for-each select="//author|//artist">
    <xsl:element name="{name()}">
        <xsl:attribute name="first" select="./first"/>
        <xsl:attribute name="last">
            <xsl:value-of select="./last"/>
        </xsl:attribute>
    </author>
</xsl:for-each>
```

This code snippet iterates over all authors and artist. For each of them it constructs a new element node with equal name and with attributes labeled first, and last with value of author's or artist's first, and last name, respectively. Notice, that the former is computed with an XPath expression in a select attribute of xsl:attribute instruction, while the latter is computed with a sequence constructor.

6.1.14 Copying

To copy a value from an input XML document, XSLT provides three instructions: xsl:value-of, xsl:copy, and xsl:copy-of. All of them construct new nodes, but each is suitable for a different task.

We already mentioned *xsl:value-of* above. It is specific with that it always creates a new text node. The xsl:value-of instruction accepts both select attribute with an XPath expression, or a sequence constructor in its body, which in both cases evaluates to a sequence. This sequence is atomized, then each item is converted to a string and the result is concatenated and returned.

Instruction *xsl:copy* creates a copy of the context item. If the context item is an element node, neither its attributes nor children are copied. Instruction xsl:copy accepts a sequence constructor that is called to create the contents of the newly created node. This expression is evaluated only for document and element nodes.

Instruction *xsl:copy-of* copies a whole subtree. As well as xsl:value-of it accepts a select attribute, or a sequence constructor in its body, the result of which is copied. The copying is deep, so for an element node all its attributes and children are copied, recursively.

6.1.15 Sorting

We already mentioned sorting ability of the xsl:for-each instruction above, but XSLT provides one more instruction that sorts a sequence. It gets a sequence, and a list of order specifications and returns that sequence sorted. This instruction is *xsl:perform-sort* and as many others accepts a sequence defined either with an XPath expression in its select attribute or with a sequence constructor in its body. The order specifications are defined with a xsl:sort instruction explained in section 6.1.11 on sequence iteration.

```
<xsl:perform-sort select="//author">
    <xsl:sort select="last"/>
    <xsl:sort select="first"/>
</xsl:perform-sort>
```

This code results in a sequence of authors sorted according to last and first name.

Notice, that order specifications are expressions that are evaluated in the context of each item in the initial sequence. Order specifications define usual lexicographic ordering on the initial sequence based on the order values of order specifications.

6.1.16 Grouping

XSLT instruction *xsl:for-each-group*, described in this section, runs a sequence constructor for each group that is identified in a given sequence. We call the given sequence an initial sequence. The initial sequence is defined with an XPath expression in a select attribute of xsl:for-each-group instruction.

Since the sequence constructor in the body of this instruction has to have a way to reference the group it is run for, XPath function *current-group()* is introduced, which evaluates to a sequence of items of the current group.

The xsl:for-each-group instruction can group the initial sequence in four different ways, depending on which attribute group-by, group-adjacent, group-starting-with, or group-ending-with is used.

Group-By. The group-by attribute is set to an XPath expression. This expression is evaluated for each item in the initial sequence, getting so a sequence of grouping keys. Each distinct grouping key identifies a group and an item belongs to every group that is identified by any of its grouping keys. Notice, that this means that an item can belong to more than one group.

There is an XPath function *current-grouping-key()* defined in a sequence constructor of xsl:for-each-group instruction that has a group-by attribute. It returns the current grouping key.

```
<xsl:for-each-group select="//cd" group-by="author">
    <author>
        <xsl:attribute name="last"
                select="current-grouping-key()/last"/>
        <xsl:attribute name="first"
                select="current-grouping-key()/first"/>
        <xsl:for-each select="current-group()">
            <cd><xsl:value-of select="title"/></cd>
        </xsl:for-each>
```

```
    </author>
</xsl:for-each-group>
```

In this example, the CDs with the same author are grouped together. For each author, an author element is created with attributes first and last, values of which are set to author's first and last name, respectively. For each CD in a group, a cd element is created whose contents is set to a title of that CD.

We should note, that this grouping variant removes duplicates from a group. It is defined that in a case when two items are equal, the one that appears before the other in the initial sequence stays and the other is removed.

Group-Adjacent. The group-adjacent attribute is also set to an XPath expression. This grouping variant groups together adjacent items from the initial sequence that evaluate the XPath expression to an equal value.

Group-Ending-With. The group-starting-with attribute is restricted to a pattern, which is a subset of XPath as noted earlier. This grouping variant groups together adjacent items from the initial sequence, where start of a new group is identified by a pattern match against an item in the initial sequence.

Group-Starting-With. The group-ending-with attribute is also restricted to a pattern. It also groups together adjacent items from the initial sequence, but in this case, it is the end of a group that is identified by a pattern match against an item from the initial sequence.

6.1.17 Keys

Keys are designed in XSLT to help with references. They allow to define a named reference that is navigated by a key() function just like id() function is used to follow a reference to elements identified with an id attribute. Keys are used to explicitly identify hidden references in an input XML document to ease the navigation and provide a hint for XSLT processors to store these references efficiently.

A key is declared with a key-declare instruction with the following syntax.

```
key-declare(name, pattern, use)
```

The declaration defines a reference named *name* that returns nodes that match the *pattern*. Not all matching nodes are returned with a call to key(*name*, *value*), but only such nodes are returned, for which the value of expression *use* is equal to *value*.

For example, let's declare a simple key named divkey that references all div elements using their name attribute.

```
key-declare("divkey", //div, @name)
```

Now, a call to key("divkey", "tab") returns all div elements that have a name attribute equal to string tab.

We can use multiple key declarations to declare a single key reference. In such a case, the result is the same as if we evaluated the key() function for each declaration separately and then concatenated the resulting sequences.

6.1.18 Stylesheet Import and Inclusion

There are two ways one can extend her own stylesheet with contents of another stylesheet: inclusion and import, which are represented with *xsl:include* and *xsl:import* top-level elements. They both take a href attribute that points to another stylesheet with an URI reference.

An xsl:include element is interpreted the same way as if the contents of the included stylesheet was pasted into the stylesheet, in which xsl:include is called.

If a stylesheet is imported with a xsl:import element, then the difference is that the imported template rules have lower import precedence compared to template rules from the importing stylesheet. Import precedence is a property of a template rule that is considered in the conflict resolution process described above.

Simply said, whereas included template rules have the same precedence as template rules in the including stylesheet, imported template rules have lower precedence then the template rules in the importing stylesheet.

Further, XSLT provides an *xsl:apply-imports* instruction that behaves similar to xsl:next-match instruction. It chooses for the context item a template rule from a set of template rules that were imported into a stylesheet, from which xsl:apply-imports instruction was called. This provides means of calling a template rule that is overridden by a template rule from the importing stylesheet.

6.1.19 Initiating a Transformation

At the beginning of this introduction, we said that a transformation is started with a root node of an input document as the current context node, for which a template is chosen, evaluated, and the result is returned as a result of a transformation. But this is not the only way a transformation can be initiated.

We can also initiate a transformation with a name of a named template, which is the first to be called. Such a template is evaluated, the result of which is then returned as a result of the transformation. This feature is important, as it allows us to show that template rules can be removed from XSLT, while not losing the expressive power of the language.

6.2 XSLT Syntax

The syntax of XSLT is defined in W3C specification. XSLT is one of the languages from the large XML family that uses XML as its basic syntax. As such, even a simple transformation is by no means brief. Therefore we provide here a more concise syntax to make the code listings shorter and easier to read.

There exists a huge number of compact non-XML syntaxes for XML, but none of them seems suitable for our needs. The reason is that we do not have to use a general syntax that would be able to express all the XML stuff like namespaces, tag names, attribute names, etc. We provide the syntax only for the XSLT tags instead.

Figure 23 defines our syntax for the considered fragment of XSLT 2.0. Since XSLT is a functional language, we write XSLT elements as functions.

As we provide our own syntax, a need arises to show how the original XML syntax is mapped to it. This is given in Figure 24, in which there is always an

```
Stylesheet          ::=  stylesheet(Import* Include* Function* Key* ParamDecl* Template*)
Import              ::=  import(URI)
Include             ::=  include(URI)
Function            ::=  function(QName, Bool, ParamDecl*, Expression)
Key                 ::=  key-declare(QName, Pattern, Expression)
ParamDecl           ::=  param-declare(QName, Expression)
Template            ::=  Template-Named | Template-Rule
Template-Named      ::=  template-named(QName, ParamDecl*, Expression)
Template-Rule       ::=  template-rule(Pattern, ParamDecl*, QName, Expression)
Expression          ::=  Expression*
                         | Instruction
                         | Constructor
                         | XPathExpr
Instruction         ::=  apply-imports(Param*, Expression)
                         | apply-templates(Param*, Sort*, Expression)
                         | call-template(QName, Param*)
                         | choose(When*, Expression)
                         | copy(Expression)
                         | copy-of(Expression)
                         | for-each(Expression, Sort*, Expression)
                         | for-each-group-by(Expression, Expression, Sort*, Expression)
                         | for-each-group-adjacent(Expression, Expression, Sort*, Expression)
                         | for-each-group-starting-with(Expression, Pattern, Sort*, Expression)
                         | for-each-group-ending-with(Expression, Pattern, Sort*, Expression)
                         | if(Expression, Expression)
                         | if-then-else(Expression, Expression, Expression)
                         | next-match(Param*)
                         | perform-sort(Sort*, Expression)
                         | sequence(Expression)
                         | value-of(Expression)
                         | variable(QName, Variable, Expression)
Constructor         ::=  attribute(Expression, Expression, Expression)
                         | element(Expression, Expression, Expression)
                         | namespace(Expression, Expression)
                         | comment(Expression)
                         | processing-instruction(Expression, Expression)
                         | text(PCData)
                         | result-document(Expression)
Param               ::=  param(QName, Expression)
Sort                ::=  sort(Expression)
When                ::=  when(Expression, Expression)
```

Figure 23: Our compact syntax for XSLT.

XSLT element on the left side and an corresponding element in our compact non-XML syntax on the right side.

One may miss some XSLT elements in the mapping. All excluded elements can be found in section 6.2.2 with comments on why we do not consider them.

One can also notice missing attributes in some of XSLT elements. We omitted all attributes connected to schema and typing, because typing is of no interest to us. We do not provide a complete list of excluded attributes.

In XSLT, almost every element that has a select attribute allows also for a sequence constructor in its body. All elements that use this combination are restricted not to use the select attribute together with the sequence constructor. The thing is that while we can use XSLT instructions in the sequence constructor, only an XPath expression is allowed in the select attribute. An interesting question arises from this: are an XPath expression and a sequence constructor capable of expressing the same? The positive answer is proved in section 6.2.1.

6.2.1 XPath Expression vs. Sequence Constructor

The XSLT syntax we provide in Figure 23, allows us to mix XSLT instructions with XPath expressions, though originally XPath expressions are allowed only in an select attribute of some XSLT instructions. In this section, we prove that XPath expressions in XSLT are of the same expressive power as a XSLT's sequence constructor, thus we do not have to distinguish between them.

First, we should define a sequence constructor itself.

Definition 6.1 *A* sequence constructor *is a sequence of XSLT Instructions and/or Constructors as defined in Figure 23.*

Lemma 6.1 *An XPath expression and an XSLT sequence constructor have the same expressive power.*

Proof. It is trivial to show that a sequence constructor can express any XPath expression. Using the xsl:copy-of instruction serves the purpose. Thus, having an XPath expression x the following sequence constructor returns the value of x.

```
<xsl:copy-of select="x"/>
```

To prove the opposite direction, we have to use user defined functions. As function's body is defined with a sequence constructor, an XPath expression with functions has the power of a sequence constructor.

This seems to be enough to show, but it is not. In general, a sequence constructor is evaluated in terms of its context, so XSLT instructions can make use of variables and parameters defined in the current context. But a sequence constructor defined as a body of function f does not have the same context as an XSLT instruction that called the f function, because the focus is emptied for the evaluation of f. Thus, using instructions in the body of f will not get the same values of variables and parameters as if using the same instructions in the calling context.

But this is not a problem. Due to the fact that functions can have parameters in XSLT, parameters can be used to pass the necessary variables and/or parameters from the calling context to functions. Now, we can be sure that whatever

<xsl:apply-imports> *ParamList Expr*</>	apply-imports(*ParamList*, *Expr*)
<xsl:apply-templates> *ParamList SortList Expr*</>	apply-templates(*ParamList*, *SortList*, *Expr*)
<xsl:call-template name="*Name*"> *ParamList*</>	call-template(*Name*, *ParamList*)
<xsl:choose> *WhenList Otherwise*</>	choose(*WhenList*, *Otherwise*)
<xsl:copy>*Expr*</>	copy(*Expr*)
<xsl:copy-of>*Expr*</>	copy-of(*Expr*)
<xsl:for-each select="*Select*"> *SortList Expr*</>	for-each(*Select*, *SortList*, *Expr*)
<xsl:for-each-group select="*Select*" group-by="*Group*"> *SortList Expr*</>	for-each-group-by(*Select*, *Group*, *SortList*, *Expr*)
<xsl:for-each-group select="*Select*" group-adjacent="*Group*"> *SortList Expr*</>	for-each-group-adjacent(*Select*, *Group*, *SortList*, *Expr*)
<xsl:for-each-group select="*Select*" group-starting-with="*Pattern*"> *SortList Expr*</>	for-each-group-starting-with(*Select*, *Pattern*, *SortList*, *Expr*)
<xsl:for-each-group select="*Select*" group-ending-with="*Pattern*"> *SortList Expr*</>	for-each-group-ending-with(*Select*, *Pattern*, *SortList*, *Expr*)
<xsl:if test="*Expr*"> *Expr*</>	if(*Expr*, *Expr*)
<xsl:key name="*Name*" match="*Pattern*"> *Expr*</>	key-declare(*Name*, *Pattern*, *Expr*)
<xsl:next-match> *ParamList*</>	next-match(*ParamList*)
<xsl:perform-sort> *SortList Expr*</>	perform-sort(*SortList*, *Expr*)
<xsl:sequence select="*Select*"/>	sequence(*Select*)
<xsl:value-of select="*Select*"/>	value-of(*Select*)
<xsl:variable name="*Name*">$Expr_1$</>$Expr_2$	variable(*Name*, $Expr_1$, $Expr_2$)
<xsl:template name="*Name*"> *ParamDeclList Expr*</>	template-named(*Name*, *ParamDeclList*, *Expr*)
<xsl:template match="*Pattern*" mode="*Mode*">*ParamDeclList Expr*</>	template-rule(*Pattern*, *ParamDeclList*, *Mode*, *Expr*)

Figure 24: Mapping W3C's original XSLT syntax to our compact syntax.

is expressible with a sequence constructor is expressible with an XPath expression. This topic is covered more deeply in section 6.3.5. □

In this section, we proved that XPath expression and a sequence constructor have the same expressive power.

6.2.2 Excluded XSLT Elements

In this section, we provide a list of all XSLT elements that we excluded from consideration. Elements are divided into several groups by the reason that led us to omit them.

String-oriented. The following elements were excluded because we think them to be only a syntactic sugaring.

```
<xsl:analyze-string>
<xsl:matching-substring>
<xsl:non-matching-substring>
```

They allow XSLT instructions to be applied not only on sequences of nodes and other items, but allow XSLT instructions to be run on parts of a string that either match or do not match a given regular expression. Since XQuery 1.0 and XPath 2.0 Functions and Operators specification provides operations to match strings by regular expressions, and operations to split strings, we believe these excluded elements to be expressible with the standard functions and operators.

Schema-oriented. The following element was excluded because we do not bother with the type information.

```
<xsl:import-schema>
```

This element servers the only purpose: it imports type information from an XML Schema document.

Evaluation-oriented. The following elements are connected to the way an XSLT processor evaluates a stylesheet.

```
<xsl:fallback>
<xsl:message>
```

The xsl:fallback instruction is designed to recover from errors that arise from the existence of two versions of XSLT: 1.0 and 2.0, whereas the xsl:message instructions is designed to help debug XSLT stylesheets – it can print messages to console while a stylesheet is evaluated. None of them influences the expressive power of XSLT.

Output-oriented. The following elements influence only output serialization and/or encoding.

```
<xsl:character-map>
<xsl:decimal-format>
<xsl:number>
<xsl:output>
<xsl:output-character>
<xsl:preserve-space>
<xsl:strip-space>
```

Character maps, provided with xsl:character-map and xsl:output-character elements, are used to easily write stylesheets that output an XML-like but non-XML text. It is designed to map characters to XML-confusing ones. For example, if the dollar sign character $ is mapped to lower-than character <, then when the output is serialized all dollar sign characters are converted to lower-than characters, providing so the desired output while not breaking the XSLT stylesheet.

Elements xsl:preserve-space and xsl:strip-space drive the white-space manipulation in terms of its importance in the output. Since white-space is not important in our data model, we do not consider them.

The xsl:output element is used to specify the output method. It dictates whether certain characters should be escaped or not when the output is serialized. The process of output serialization is defined in XSLT 2.0 and XQuery 1.0 Serialization specification of W3C. This element can also be used to specify the desired output encoding. Since features provided by this element refer to the output serialization and not the data model, we do not consider them.

Literal result elements. XSLT specification says that it is possible to construct new elements in the resulting document in two ways: either i) by using an constructor instruction, or ii) by literally naming the element. Our XSLT excerption does not allow to use literal result elements as it is only syntactic sugar, to keep the language concise.

More sugar. The following elements provide more syntactic sugaring.

```
<xsl:attribute-set>
<xsl:namespace-alias>
```

The xsl:attribute-set element is designed to ease the manipulation of several attributes together by assigning a name to a set of attributes and its values. Then, instead of typing every single attribute from the set and its value, we can use the named set. It is clear, that this is designed to cut the text the stylesheet programmer has to write short.

The xsl:namespace-alias element converts a namespace prefix of result elements to a different prefix. This is needed especially when generating XSLT stylesheets with XSLT. Since the transformation stylesheet is written in XML, it uses the xsl namespace prefix to distinguish its own instructions from the result elements. Then, what prefix should be given to XSLT instructions that are to become a part of the resulting document? Giving them the same prefix xsl would cause the XSLT processor to evaluate them, but we want them to stay in the output text. The solution provided by the XSLT specification is to give them a different prefix (e.g. axsl) and declare that this prefix has to be mapped to XSLT namespace URI in the resulting document. To sum up, this feature is needed only due to XML syntax of the XSLT language and as such does not influence the expressive power of the language, so we do not consider it.

In this section, we listed all XSLT elements that we do not consider when evaluating expressive power of XSLT. We also mentioned with each element excluded the reasons that led to our decision.

6.3 XSLT Core

In this section we look for a simpler language that has the same expressive power as the selected XSLT fragment.

6.3.1 Notation

As we want to prove that equality of expressions within stylesheets, we start by formalizing XSLT stylesheets and items they are constructed from.

First, we define patterns, which is a subset of an XPath expression. We introduce patterns in section 6.1.1.

Definition 6.2 *With Pattern we denote a set of all patterns.*

Next, we formally define a stylesheet.

Definition 6.3 *Let $S = \langle T, N, F, P, U, M, \tau \rangle$ be an XSLT stylesheet, with the following components.*

- *T is a set of template rules including those from imported and included stylesheets,*
- *N is a set of named templates,*
- *F is a set of functions defined in S,*
- *P is a set of all non-tunnel parameter names,*
- *U is a set of all tunnel parameter names,*
- *M is a set of all mode names,*
- *$\tau \subset T \times T$ is a template choosing order – a total order on template rules.*

The template choosing order τ is used to resolve a conflict of multiple template rules matching the current context item. Notice, that this order exists for every stylesheet. More on the conflict resolution process can be found in section 6.1.5 on page 71.

As we need to identify the order of a template rule in a subset of T later, we define function τ that takes a set $T' \subseteq T$ of template rules, and a natural number n and returns a template rule that is n-th in T' according to template choosing order τ. Thus, $\sigma_S(T', 1)$ returns the first template rule in template choosing order τ, while $\sigma_S(T', |T'|)$ returns the last rule in template choosing order τ.

Definition 6.4 *Let $S = \langle T, N, F, P, U, M, \tau \rangle$ be an XSLT stylesheet. Then, $\sigma_S : 2^T \times \mathbf{N} \rightarrow T$ is a function that takes as arguments a set of template rules $T' \subseteq T$, and a natural number n such that $1 \leq n \leq |T'|$ and returns a template rule from T'. The function is defined inductively as follows.*

$$\begin{aligned} \sigma_S(T', 1) &= \min_\tau(T') \\ \sigma_S(T', i) &= \min_\tau(T' \setminus \bigcup_{j<i} \sigma_S(T', j)) \end{aligned}$$

We also define an inverse function $\sigma_S^{-1} : 2^T \times T \rightarrow \mathbf{N}$ that takes as arguments a set of template rules $T' \subseteq T$, and a template rule $t \in T'$ from this set and returns a number that represent the order of t in T' according to template choosing order τ.

Definition 6.5 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet, and* $t \in T' \subseteq T$ *be a template rule. Then, with* $\sigma_S^{-1}(T', t)$ *we denote such* i*, for which* $t = \sigma_S(T', i)$.

Since a template is a structure, we provide several functions to decompose it. First, two functions that work with a template rule only.

Definition 6.6 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet, and* $t \in T$ *be a template rule. Then, function* $patt(t)$ *returns a pattern of* t*, and function* $mode(t)$ *returns a mode of* t.

$$\begin{aligned} patt &: T \rightarrow Pattern \\ mode &: T \rightarrow M \end{aligned}$$

The following definition introduces functions that work with both template rules and named templates, and with functions as well.

Definition 6.7 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet,* $t \in T \cup N \cup F$ *be a template or function, and* $p \in P \cup U$ *be a parameter name. Then, function* $body(t)$ *returns a body expression of* t*,* $params(t)$ *returns a set of parameter names that are declared in* t*, and finally* $default(t, p)$ *returns a default value expression of parameter* p *declared in* t.

$$\begin{aligned} body &: (T \cup N \cup F) \rightarrow Expr \\ params &: (T \cup N \cup F) \rightarrow P \\ default &: (T \cup N \cup F) \times (P \cup U) \rightarrow Expr \end{aligned}$$

Next, we define what we mean with a stylesheet equivalence.

Definition 6.8 *We say that two stylesheets are* equivalent *if they return the same result for every input document they are applied to, with respect to our data model.*

We refer to a set of instructions that select a template rule for the current context item as to applying instructions. If even call-template instruction is to be considered, we refer to it as to a template call.

Definition 6.9 *An instruction is called* applying instruction *if it is either apply-templates, apply-imports, or next-match.*

Every expression that is either an applying instruction, or a call-template instruction, we call a template call.

We refer to every expression that results in an evaluation of a template's or function's body expression as to a call. We also define a function that returns all calls from a stylesheet.

Definition 6.10 *Every expression that is either an applying instruction, call-template instruction, or an XPath function call, we call a* call expression, *or simply a* call. *Let* S *be an XSLT stylesheet, then with* $calls(S)$ *we denote a set of all calls in* S.

The following function returns a set of all parameters passed by a call.

Definition 6.11 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet, and* $c \in calls(S)$ *be a call expression. Then with* $paramsPassedBy(c)$ *we denote a set of parameters passed by* c *to a template or function.*

$$paramsPassedBy : calls(S) \rightarrow (P \cup U)$$

6.3.2 Flow Control

Instead of two flow control instructions choose, and if, we introduce a single if-then-else instruction that is both capable of expressing the former two, and expressible by the former two. Thus, we simplify the language while not changing its expressive power.

Further, the if-then-else instruction can be expressed with the if-then-else XPath expression. Therefore, even if-then-else instruction is only a syntactic sugar.

Since instruction choose greatly improves readability, we still use it in the following code listings.

6.3.3 Parameters

Parameters provide way for templates to behave differently under different circumstances. Without parameters, the language would lost a lot of its expressive power. Parameters can be passed when calling named templates or when one of applying instructions is used. Even a function call that appears as a step in an XPath expression can be equipped with parameters.

In this section, we first note that we can safely add new parameters to a stylesheet, then we show that parameter declarations can be removed from templates, and finally we show that tunnel parameters can be simulated with non-tunnel parameters.

Adding New Parameters

Here, we show a fact that is trivial, but worth noting: we can safely add new parameters to a stylesheet and we do not have to care about a name conflict.

It is required that when a parameter is to be passed, its name must be set as a literal. A very important fact can be deduced from this: i) parameter names are statically known, since they neither depend on a processed document nor can be computed at run-time, and ii) number of parameters used in a stylesheet is finite.

Lemma 6.2 *Let $S = \langle T, N, F, P, U, M, \tau \rangle$ be an XSLT stylesheet. Then, there exists a new parameter name p, such that $p \notin P \cup U$.*

Proof. Since both sets of parameter names P, and U are finite, independent from the input document, and statically know, we can always create a new name that is not in $P \cup U$. □

Due to this lemma, we can always add a new parameter to a stylesheet without a fear of a name conflict with one of the stylesheet's former parameters, because we can always choose a name that never appears in the former stylesheet. From now on, when adding parameters, we suppose that the name provided differs from all statically known parameters in the stylesheet, possible conflicts can be always solved by renaming.

Parameter Declarations

In this section, we show that parameter declarations of non-tunnel parameters can be safely removed from a stylesheet, which is achieved by always passing

all parameters. The only problem is a possible name clash of parameters that define different default value expressions, which we show not to be a problem. When referring to parameters, we always talk about non-tunnel parameters in this section.

XSLT specification requires template rules and named templates to declare all parameters they want to accept. It is possible to provide a default value expression for each parameter. If a parameter is not explicitly passed to a template, then its default value expression is evaluated in the current context to get the value of the parameter.

We claim that default value expressions and even parameter declarations are only syntactic sugar.

Idea. The idea is to pass all parameters in P by every call expression and to move the default value expressions from parameter declarations to call expressions. The move can be done safely only when each parameter is declared only in one template or function, because if a parameter p is declared in two templates or functions t and t', then the default value expressions in t and t' are generally different, so we do not know, which default value expression to pass by calls. This can be solved by parameter renaming, as a scope of a parameter is a template's or function's body and as we can add new parameter names safely, which we proved in Lemma 6.2.

Then, when all parameters are passed in every call and all parameters are declared with no default value expression in every template or function, we can do without parameter declarations, as for each template or function they are the same.

Formally, we transform a stylesheet $S = \langle T, N, F, P, U, \sigma \rangle$ to an equivalent one without parameter declarations, $\forall t \in T \cup N \cup F : \mathit{params}(t) = \emptyset$, which is done in three steps: i) we ensure that each parameter is declared only in one template or function, ii) we modify all call expression to pass all parameters, and iii) we remove parameter declarations from all templates and functions in a stylesheet.

Distinct Parameter Names. First, we ensure that every template uses parameters with names different from parameter names in other templates. This is achieved with procedure *DistinctParams* depicted in Figure 25 that we describe here. For each template or function t from $T \cup N \cup F$, if there exists a template or function t' from $T \cup N \cup F$ such that they share at least one parameter declaration with the same name, let $C := \mathit{params}(t) \cap \mathit{params}(t') \neq \emptyset$ be the set of conflicting parameter names. Create C', a set of $|C|$ new parameter names, $C' \cap P = \emptyset$. Now, change t by renaming the conflicting parameter names from C with new parameter names from C' and add these new parameters to $P := P \cup C'$. Further, in each call in S that passes a parameter $p \in C$ that has been renamed to $p' \in C'$, pass even p' and set its value to the same expression as p. Follow, until there are no templates that share the same parameter name $\forall t, t' \in T \cup N \cup F : \mathit{params}(t) \cap \mathit{params}(t') = \emptyset$.

Lemma 6.3 *Let $S = \langle T, N, F, P, U, M, \tau \rangle$ be an XSLT stylesheet. Then, there exists an XSLT stylesheet $S' = \langle T', N', F', P', U, M, \tau' \rangle$ such that S' is equivalent to S and for each $p \in P'$ there is exactly one $t \in T' \cup N' \cup F'$ that declares p in its body.*

Proof. Let $S' = \mathit{DistinctParams}(S)$.

First, we check the stylesheet equivalence. Notice, that procedure *DistinctParams* affects only bodies of templates and functions in $T \cup N \cup F$. As we do

```
1: procedure DistinctParams(S = ⟨T, N, F, P, U, M, τ⟩)
2:     for all t ∈ T ∪ N ∪ F do
3:         for all t' ∈ T ∪ N ∪ F do
4:             C ← {p_1, ..., p_n} = params(t) ∩ params(t')          ▷ Name clash
5:             C' ← {p'_1, ..., p'_n} such that {p'_1, ..., p'_n} ∩ P = ∅ ▷ New names
6:             P ← P ∪ C'
7:             in t, rename declaration of p_i to p'_i, for 1 ≤ i ≤ n
8:             for all c ∈ calls(S) do
9:                 C'' ← {p_{j_1}, ..., p_{j_m}} = paramsPassedBy(c) ∩ C
10:                change c to pass p'_{j_i} with the same value as p_{j_i}, for 1 ≤ i ≤ m
11:            end for
12:        end for
13:    end for
14:    return S
15: end procedure
```

Figure 25: Procedure *DistinctParams* ensures that each template uses a unique set of parameter names.

not modify an order of template rules in T nor we create or delete template rules, it is clear that template choosing order τ' is the same as τ, only it is defined on the templates and functions with modified bodies. Therefore, we only need to check that templates and functions behave the same in the modified stylesheet S' as in the initial stylesheet S.

If there are two templates or functions $t, t' \in T \cup N \cup F$ such that they both declare the same parameter, then the conflicting parameters are renamed to new parameter names in t. The for all expression on lines 8 to 11 ensures that if t is called from whatever call expression, then it is passed in the modified stylesheet S' the same value as it was in the initial stylesheet S. Clearly, there is nothing changed in t'. Therefore, stylesheet S' has to return the same result as S for the same input document, so they are equivalent.

Next, we check that parameter p is declared only once in the modified stylesheet S'. Let's suppose for contradiction, that p is declared in two templates or functions $t, t' \in T \cup N \cup F$. It cannot happen that $p \in P' \setminus P$ is a renamed parameter, as we assign new names to renamed parameters on line 5. Thus, $p \in P$ and $params(t) \cap params(t') \neq \emptyset$ and such a parameter is renamed on line 7 to new name p', which is a contradiction. □

Pass All Parameters. Now, we continue with a stylesheet $DistinctParams(S)$, which declares each parameter only in one template or function. We take every call expression and change it to pass all parameters from P, as described by procedure *PassAllParams* depicted in Figure 26. It does the following. For each parameter name $p \in P$, if the call passes parameter p, then we leave it untouched. If the call does not pass p, we add p to parameters passed by the call with its value set to $default(t, p)$, where t is the only template or function that declares parameter p. It is the result of procedure *DistinctParams* that exactly one such t exists.

Remove Parameter Declarations. Finally, we simply remove parameter declarations from all templates and functions.

We have to ensure that a template or function that does not declare a pa-

1: **procedure** *PassAllParams*($S = \langle T, N, F, P, U, M, \tau\rangle$)
2: **for all** $c \in calls(S)$ **do**
3: **for all** $p \in P$ **do**
4: **if** $p \notin paramsPassedBy(c)$ **then**
5: $t \leftarrow t' \in T \cup N \cup F$ such that $p \in params(t')$
6: change c to pass p with its value set to $default(t, p)$
7: **end if**
8: **end for**
9: **end for**
10: **return** S
11: **end procedure**

1: **procedure** *RemoveParamDecls*($S = \langle T, N, F, P, U, M, \tau\rangle$)
2: $S \leftarrow PassAllParams(S)$
3: **for all** $t \in T \cup N \cup F$ **do**
4: $params(t) \leftarrow \emptyset$
5: **end for**
6: **end procedure**

Figure 26: Procedure *PassAllParams* ensures that all parameters are passed in every call. Procedure *RemoveParamDecls* removes parameter declarations from all templates and functions in a stylesheet.

rameter, resets its value in the beginning of a body expression, as described by procedure *RemoveParamDecls* in Figure 26. For each template or function $t \in T \cup N \cup F$, we add instructions that set every undeclared parameter $p \in P \setminus params(t)$ to an empty sequence at the beginning of $body(t)$. Then, we remove all parameter declarations $params(t)$ from t.

Lemma 6.4 *Let $S = \langle T, N, F, P, U, M, \tau\rangle$ be an XSLT stylesheet. Then, there exists an XSLT stylesheet $S' = \langle T', N', F', P', U, M, \tau'\rangle$ such that S' is equivalent to S and $\forall t \in T' \cup N' \cup F' : params(t) \cap P = \emptyset$.*

Proof. Let $S' = RemoveParamDecls(DistinctParams(S))$.

Let's check that we solved the problem. Since only parameters are changed either in calls or in declarations in templates and functions, we inspect only parameters. As we proved in Lemma 6.3, $S'' = DistinctParams(S)$ is equivalent to S and each parameter is declared in only one template or function.

Further, we inspect the change from S'' to stylesheet S' caused by procedure $RemoveParamDecls(S'')$. We check the value of a parameter in S'', when it is evaluated in a body of a template or function, and compare it to its corresponding value in S'. Notice, that this evaluation is always caused by some call expression c.

Principally, the following two situations can arise when evaluating a parameter $p \in P$ in a template or function t in stylesheet S'': i) parameter p is declared and its value is passed by c, ii) parameter p is declared but its value is *not* passed by c. Notice, that if a parameter is not declared but its value is passed by c is not interesting for us. A reference to such a parameter raises an error, so a stylesheet that contains it is not correct and so we do not consider it.

In the first case, when p is both declared in t and passed by c, nothing

changes in S', as changes are done only for parameters that are either undeclared or not passed by c.

In the second case, when p is declared but it is not passed by c, a default value for p is used in S''. The same behavior in S' is guaranteed by procedure *PassAllParams* on lines 4 to 7. □

In this section, we proved that parameter declarations with default value expressions are only syntactic sugaring and as such they can be safely removed from the language, while not weakening the expressive power of XSLT.

Tunnel Parameters

In this section, we show that tunnel parameters are only a syntactic sugar, by transforming a stylesheet with tunnel parameters to a stylesheet without them. Formally, we transform a stylesheet $S = \langle T, N, F, P, U, M, \tau \rangle$ to an equivalent stylesheet $S' = \langle T', N', F', P', U' = \emptyset, M, \tau' \rangle$.

Tunnel parameters allow passing parameter values through templates that do not care about them to templates that do. Once a parameter is passed as a tunnel parameter, it is automatically passed on by every template call, even without explicitly saying so.

When evaluating a stylesheet, there is a set of tunnel parameters that is passed on in every template call. This set can only grow, because there is no mechanism to remove a tunnel parameter from this set. The set of tunnel parameters is passed to templates, but not to functions.

As with usual parameters, a declaration of a tunnel parameter in a template can specify a default value using an expression. XSLT specification says that a default value of a tunnel parameter is local to a template: thus, a default value of a tunnel parameter does not change any value in the set of tunnel parameters that is passed on in further template calls.

Unfortunately, this feature complicates the transformation to a stylesheet without tunnel parameters, because the problem now differs greatly from the problem with usual parameters that we described in the previous section. It is that with usual parameters we could solve the default value problem statically. As usual parameters have to be passed explicitly, we have a static knowledge whether a default value should be used or whether a value is passed directly in a call. Now, with tunnel parameters, this information is not statically known as it depends on the computation branch the XSLT processor runs through and so it depends on an input XML document.

In the following, we use parameter declarations again, though we proved they are only syntactic sugar. Notice, that parameter declarations we work with, are declarations of non-tunnel parameters that can be safely removed from a stylesheet according to Lemma 6.4.

Idea. The trick inheres in keeping with each tunnel parameter $u \in U$ one flag parameter u' that indicates, whether the parameter u has been set or not in one of the preceding template calls. Then, we set a local default value in a declaration of u to a value passed by a call, if its flag indicates that the parameter has not been set before.

As in the case of usual parameters, we pass all tunnel parameters with every template call, and their respective flag parameters too. Now, of course, none of the parameters is marked as tunnel.

Since functions cannot be passed tunnel parameters, we have to set a value of each tunnel parameter in a function call to an empty sequence and we have to set a value of its flag parameter to indicate that the parameter is not set.

To sum up, the transformation of a stylesheet with tunnel parameters to a stylesheet with no tunnel parameters is done in two steps: i) every tunnel parameter is passed as a non-tunnel parameter in each call, and ii) declarations of tunnel parameters are changed to declarations of non-tunnel parameters with modified default value expressions.

Formally, let's assume that all tunnel parameters U have names different from all usual parameters P, $P \cap U = \emptyset$. We can assume this without loss of generality, because as we can distinguish usual parameters from tunnel parameters in a stylesheet, we can rename the conflicting ones. For a parameter u, let u' denote its new flag parameter, $u' \notin P \cup U$. A flag parameter is always set either to *true* or *false*.

Pass No Tunnel Parameter. In the first step, we ensure that every tunnel parameter is passed in every template or function call and that it is passed as a non-tunnel parameter. This process is described by procedure *PassNoTunnelParam* in Figure 27, we describe it in more detail here.

For each tunnel parameter $u \in U$, add u to every template call's parameter list that does not pass u and add u' to every template or function call's parameter list. The values of the former tunnel parameter u and its flag parameter u' vary depending on whether we modify a template call or a function call.

For a template call, if u was already in the parameter list, set u' to *true*, otherwise set u' to $\$u'$ – an expression evaluating to the actual value of u' (a value passed to a template). Set value of u to the following code.

```
if-then-else($u', $u, ())
```

It passes on a value of u, if u has been set earlier in the computation, or an empty sequence, if it has not been set before.

For a function call, set u to an empty sequence and its flag parameter to *false* to indicate that the parameter u is unset.

Fix Default Value Expressions. In the second step, we fix the default value expressions of former tunnel parameters, so that parameter u is assigned its default value expression $default(t, u)$ in a template $t \in T \cup N$, only if its flag parameter u' is unset. Otherwise, we keep the value passed to u by the current call. This is realized with the following code that replaces the default value expression of u.

```
if-then-else($u', $u, default(t,u))
```

It returns a value of u, if u has been set earlier in the computation. Otherwise it returns the default value $default(t, u)$. The whole process is described by procedure *RemoveTunnelParams* given in Figure 27.

Lemma 6.5 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet. Then, there exists an XSLT stylesheet* $S' = \langle T', N', F', P', U', M, \tau' \rangle$ *such that* S' *is equivalent to* S *and* $U' = \emptyset$.

Proof. Let $S' = RemoveTunnelParams(S)$.

It is clear that stylesheet S' contains tunnel parameters neither in calls nor in declarations. The former is due to procedure *PassNoTunnelParam* that changes

```
1: procedure PassNoTunnelParam(S = ⟨T, N, F, P, U, M, τ⟩)
2:     for all u ∈ U do
3:         for all c ∈ calls(S) do
4:             if c is a template call then
5:                 passed ← u ∈ paramsPassedBy(c)
6:                 change c to pass u′ flag parameter
7:                 change c to pass u as a non-tunnel parameter
8:                 if passed then
9:                     set value passed by u′ to true
10:                else
11:                    set value passed by u′ to $u′
12:                    set value passed by u to if-then-else($u′, $u, ())
13:                end if
14:            else                                          ▷ c is a function call
15:                set value passed by u′ to false
16:                set value passed by u to ()
17:            end if
18:        end for
19:    end for
20:    return S
21: end procedure
```

```
1: procedure RemoveTunnelParams(S = ⟨T, N, F, P, U, M, τ⟩)
2:     S ← PassNoTunnelParam(S)
3:     for all t ∈ T ∪ N do
4:         for all u ∈ U do
5:             declare u as non-tunnel parameter in t
6:             if u ∈ params(t) then
7:                 default(t, u) ← if-then-else($u′, $u, default(t, u))
8:             end if
9:         end for
10:    end for
11:    return S
12: end procedure
```

Figure 27: Procedure *RemoveTunnelParams* transforms a stylesheet with tunnel parameters to an equivalent one without tunnel parameters.

every tunnel parameter in every call to be passed as a non-tunnel parameter, the latter is due to procedure *RemoveTunnelParams* that changes declaration of every tunnel parameter in every template to be declared as a non-tunnel parameter.

When the transformation is initialized, all former tunnel parameters are unset and their flags too.

Functions are treated to be passed all tunnel parameters with their values and flag parameters unset, which is correct.

Further we inspect only template calls. If a former tunnel parameter u is passed in some call, then u' is always set to true. Otherwise former value of u and its flag u' is passed on.

Thus, if a former tunnel parameter u is declared in a template t, then it is assigned a value passed by a call only if its flag u' is set to true, which indicates that u has been set by some previous call. Otherwise, if u has not been set by some previous call, u is assigned the default value, which is exactly the semantics of tunnel parameters. □

In this section, we proved that tunnel parameters can be simulated with non-tunnel parameters.

Conclusion

Here, we put together, what we proved in this section on parameters in XSLT. The tunnel parameters can be simulated with non-tunnel parameters, which in turn do not need to be declared. Therefore, the following lemma holds.

Lemma 6.6 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet. Then, there exists an XSLT stylesheet* $S' = \langle T', N', F', P', U', M, \tau' \rangle$ *such that* S' *is equivalent to* S, $U' = \emptyset$, *and* $\forall t \in T' \cup N' \cup F' : params(t) = \emptyset$.

Proof. Let $S' = RemoveParamDecls(DistinctParams(RemoveTunnelParams(S)))$. S' satisfies the claimed conditions due to Lemma 6.4 and Lemma 6.5. □

6.3.4 Template Rules

Template rules are the first thing one learns when studying XSLT. Surprisingly, they do not extend the expressive power of XSLT. In this section, we prove that for a stylesheet with template rules we can always find an equal stylesheet without template rules. A stylesheet without template rules cannot use the default initiation, but it has to be initialized with a named template. For details on stylesheet initialization see section 6.1.19 on page 81.

First, we show how template rules and applying instructions can be removed to get an equivalent stylesheet for stylesheets with no modes in template rules, and then even with modes in template rules.

Second, we explain why stylesheet inclusion and import are only a helpful tool for a programmer that does not improve the expressive power of the language.

Finally, we make a note that there is no difference in matching a pattern, i.e. a limited XPath expression, or an unlimited XPath expression.

Template Rule Choosing Mechanism

Here, we demonstrate that the template choosing mechanism is expressible in XSLT itself, which is rather surprising. We use only template rules without modes in this section.

To prove our claim, we need to remove template rules and replace all applying instructions apply-templates, apply-imports, and next-match with non-applying instructions, which we do one by one.

Remove Template Rules. In this step, we remove the template rules from a stylesheet, which we do together with removal of apply-templates instruction.

The idea is to replace the template choosing mechanism with a named template that tries to match the current context item against the patterns of template rules in the template choosing order. If the current context item matches the pattern, it evaluates the body expression of the corresponding template rule. Sure, apply-templates instruction has to be replaced with an instruction that calls this named template for each item selected for templates application. Further, order of selected items has to be the same as in the apply-templates instruction.

Once template rules are removed, we always have to initiate a transformation with a template call. We provide a special named template for this.

First, we define an expression that applies built-in template rules for the current context item.

Definition 6.12 *With BuiltIn(ParamList) we denote the following expression.*

```
choose(
  when( self::document() or self::element(),
    apply-templates(child::*, ∅, ParamList))
  when (self::text() or self::attribute(),
    value-of(.))
  otherwise( () )
)
```

Notice, that this is direct reproduction of informal specification of built-in rules, which we provided in section 6.1.4. Also notice, that expressions document(), attribute(), etc. are node tests that try to match the current context item.

Next, we define an expression that is a replacement for the apply-templates instruction. We do not provide a separate definition for the apply-templates replacement expression, but rather we provide an expression $BODY(t)$ that is a replacement of a body $body(t)$ of a template or function t.

Definition 6.13 *Let $S = \langle T, N, F, P, U = \emptyset, M = \emptyset, \tau \rangle$ be an XSLT stylesheet, and $t \in T \cup N \cup F$ be a template or function. With $BODY(t)$ we denote a sequence constructor $body(t)$ that has every occurrence of instruction apply-templates(ParamList, SortList, Expr) replaced with the following expression.*

```
for-each(Expr, SortList,
  call-template("apply-tpl", ParamList)
)
```

Notice, that template apply-tpl is called in the same context and in the same order as template rules are called with the replaced apply-templates instruction. This is ensured with the for-each instruction that changes the current context and sorts the processed sequence of items in the desired way.

Next, we define a named template that is a replacement for the template rule choosing mechanism. We use the fact, that we have a template choosing order τ, which is a total order, in which template rules are searched for to find the one that matches. Recall that $\sigma_S(T, i)$ returns the i-th template rule in T in template choosing order τ.

Definition 6.14 *Let $S = \langle T, N, F, P, U = \emptyset, M = \emptyset, \tau \rangle$ be an XSLT stylesheet. Then, with $ApplyTpl(S)$ we denote the following named template.*

```
template-named("apply-tpl",
  choose(
    when( some $node in patt(σS(T,1)) satisfies . is $node,
      BODY(σS(T,1)))
    ...
    when( some $node in patt(σS(T,|T|)) satisfies . is $node,
      BODY(σS(T,|T|)))
    otherwise( BuiltIn(ParamList) )
  )
)
```

In the template choosing mechanism replacement above, we use instruction choose to match the patterns of former template rules in an order defined by τ. The modified body of the former template rule, whose pattern matches first, is evaluated. If no pattern matches, then the built-in rules are applied using our *BuiltIn* expression.

Pattern matching is realized with some-in-satisfies XPath expression. Notice, that it directly follows the semantics of pattern matching described in the introduction section. It looks for the current context item among a sequence of nodes that are the result of evaluating the pattern XPath expression of the former template rule. First, sequence of nodes represented by the pattern is evaluated, then variable $node is bound one by one to the nodes from this sequence and the satisfies expression is evaluated for each such binding. The satisfies expression here is true, if the current context item and variable $node refer to the same node. The some-in-satisfies expression is then true, if the pattern matches the current context item.

Also notice, that we silently assume that apply-tpl is a name used by no named template in N. As the set of named templates is finite and statically known, we can always make up a brand new name instead of apply-tpl. Therefore, this can be assumed without loss of generality.

Next, we provide a named template that is used to start the transformation with.

Definition 6.15 *With StartTpl we denote the following named template.*

```
template-named("start-tpl",
  for-each( /, ∅,
    call-template("apply-tpl", ∅)
  )
)
```

1: **procedure** *RemoveTemplateRules*($S = \langle T, N, F, P, U = \emptyset, M = \emptyset, \tau \rangle$)
2: **for all** $t \in T \cup N \cup F$ **do**
3: $body(t) \leftarrow BODY(t)$
4: **end for**
5: $N \leftarrow N \cup \{ApplyTpl(S), StartTpl\}$
6: $T \leftarrow \emptyset$
7: $\tau \leftarrow \emptyset$
8: **return** S
9: **end procedure**

Figure 28: This procedure takes a stylesheet with template rules and no modes and converts it to an equivalent stylesheet without template rules. The initial stylesheet cannot contain other applying instruction than apply-templates for the procedure to work correctly.

It simulates the default initialization of transformation, when template rules are applied to the root node of the input document. It simply selects the root node of the input document and calls the named template, which we use as a replacement of the template choosing mechanism.

We put the above definitions together in procedure *RemoveTemplateRules* depicted in Figure 28. It replaces apply-templates instructions in bodies of templates and functions in lines 2 to 4. Then, it adds the template choosing mechanism replacement $ApplyTpl(S)$ together with the initiating named template *StartTpl* to named templates. Finally, template rules are removed and template choosing order τ as well, as it is no longer needed when there are no template rules in the resulting stylesheet.

Lemma 6.7 *Let* $S = \langle T, N, F, P, U = \emptyset, M = \emptyset, \tau \rangle$ *be an XSLT stylesheet with no applying instruction other than apply-templates. Then, there exists a stylesheet* $S' = \langle T' = \emptyset, N', F', P, U = \emptyset, M = \emptyset, \tau' = \emptyset$ *with no template rules that is equivalent to* S.

Proof. Let $S' = RemoveTemplateRules(S)$. Clearly, it contains no template rules and template choosing order is therefore empty too.

We need to check the equivalence with S, only. As we replaced apply-templates instruction with a for-each instruction that calls a new named template apply-tpl, we need to check if it behaves the same.

As we noted earlier, the replacement of apply-templates instruction works well, as it selects the right context in which apply-tpl template is called.

We also noted above, that apply-tpl works well, as it correctly finds the matching pattern and as it evaluates the body expression corresponding to the matching pattern in the right context, which is the same context, in which apply-tpl template is called. The order, in which template rules are checked, is the right one, as we choose the templates in template choosing order. Finally, if no template rule matches the current context node, then built-in rules are simulated with our *BuiltIn* expression. □

Remove apply-imports. The apply-imports instruction applies a template rule to the current context item, where it considers only template rules from the stylesheets imported to a stylesheet with the current template rule. We do

not go into technical details in this section, we rather spot the pitfalls of the removal of apply-imports instruction and outline the solution.

This is not so easy, as it might look. Consider, that the current template rule from stylesheet S calls a named template from a different stylesheet S'. If in this named template apply-imports is called, then the considered template rules are not the template rules imported to S', but it is the template rules imported to S, as the current template rule is not changed by a call to a named template.

Therefore, we need to keep an identification of the current template rule throughout the computation. Since all template rules are totally ordered with template choosing order τ, we can identify each template rule with a number that represents the order of a template rule in τ. From now on, for each template rule $t \in T$, with id of t we denote $\sigma_S^{-1}(T, t)$, which is the order of $t \in T$ in template choosing order. Thus, we have a numerical id for every template rule, and so the id of the current template rule can be passed on through templates with a parameter.

Then, for a stylesheet S, we can construct a named template apply-tpl-S that simulates apply-imports instruction called, when the current template rule is in S. It has the same structure as named template apply-tpl, but the rules it is built from come only from the stylesheets imported to stylesheet S. Further, each body expression is changed to pass the id of the current template rule.

Next, we have to assure that an apply-instruction replacement calls the proper apply-tpl-S template. It means that we have to provide a mapping that maps an id of the current template rule to the corresponding apply-tpl-S. This can be easily done with a choose instruction.

Remove next-match. The `next-match` instruction applies a template rule to the current context item, where it considers only template rules that succeed the current template rule in the template choosing order.

Here, we have to work with an id of the current template rule as when removing the apply-imports instruction. Recall that id of a template rule is its order in the template choosing order.

To replace next-match instruction, we construct a named template apply-next-tpl that has the same structure as named template apply-tpl. It is extended with a test that allows only rules that succeed the current template rule in template choosing order to match. It can look like the following.

```
template-named("apply-next-tpl",
  choose(
    when( some $node in patt(σS(T,1)) satisfies . is $node
                                      and 1 > $CurrentTplRuleID,
      BODY(σS(T,1)))
    ...
    when( some $node in patt(σS(T,|T|)) satisfies . is $node
                                      and |T| > $CurrentTplRuleID,
      BODY(σS(T,|T|)))
    otherwise( BuiltIn(ParamList) )
  )
)
```

Notice, that we extended the satisfies expression in comparison to apply-tpl named template with a condition $i > id$, where i is id of the former template rule, whose pattern is tried for a match, and id is the id of the current template

rule. As id of a template rule is the order of a template rule in template choosing order τ, the above condition ensures that only template rules that succeed the current template rule in τ can match.

Sure, each instruction next-match is replaced with a call-template instruction that calls the just defined apply-next-tpl template.

To sum up, we showed that template rules and applying instructions can be removed from a stylesheet that has no modes, such that we end up with a stylesheet that is equivalent to the initial one.

Lemma 6.8 *Let $S = \langle T, N, F, P, U = \emptyset, M = \emptyset, \tau \rangle$ be an XSLT stylesheet. Then, there is an XSLT stylesheet $S' = \langle T' = \emptyset, N', F', P', U = \emptyset, M = \emptyset, \tau' = \emptyset \rangle$ without template rules that is equivalent to S.*

Proof. We showed that we can safely remove apply-templates with template rules in Lemma 6.7, then we outlined how apply-import instructions can be replaced, and finally we showed how next-match instructions can be replaced. Thus, after applying all three transformations to S we get a stylesheet with no template rules and no applying instruction. □

Modes

In the previous section, we proved that we can transform a stylesheet with template rules and applying instructions to an equivalent one without them. The only restriction we put, is that a stylesheet cannot use modes. In this section, we extend the transformation to work even for stylesheet that use modes.

Formally, we transform a stylesheet $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ with modes to $S' = \langle T' = \emptyset, N', F', P', U' = \emptyset, M' = \emptyset, \tau = \emptyset \rangle$ that is equivalent to S and that uses no template rules and no modes.

The idea is to follow the transformation from the previous section. We slightly change it, such that apply-templates replacement now passes its mode with a new parameter $current_mode. We split the apply-tpl named template that replaces the template choosing mechanism to several parts, where each part contains only template rules with the same mode, and replace the apply-tpl named template to choose the proper template according to the value of $current_mode.

We provide the transformation for apply-templates instruction only. The other two applying instructions can be removed accordingly.

First, we define a set of template rules that are checked when mode m is specified in apply-templates instruction.

Definition 6.16 *Let $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ be an XSLT stylesheet, and $m \in M$ be a mode. Then, with $all(m)$ we denote the following set of template rules.*

$$all(m) \quad = \quad \{\, t \in T \mid mode(t) \in \{m,\ \texttt{\#all}\,\}\}$$

Next, we provide a replacement of the apply-templates instruction that considers its mode. Like in the previous section, we give it as a replacement of a body of a template or function. Notice, that special mode #current is treated so that it passes on the current value of $current_mode parameter.

Definition 6.17 *Let* $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ *be an XSLT stylesheet, and* $t \in T \cup N \cup F$ *be a template or function, and* $m = mode(t)$ *be its mode. Then, with* $BODYMode(t)$ *we denote a sequence constructor* $body(t)$ *that has every occurrence of instruction apply-templates(ParamList, SortList, Expr) replaced with one of the following expressions, depending on* m.

If mode $m \neq$ `#current` *then replace it with the following.*

```
for-each(Expr, SortList,
  call-template("apply-tpl",
      param($current_mode, m) ParamList)
)
```

If mode $m =$ `#current` *then replace it with the following.*

```
for-each(Expr, SortList,
  call-template("apply-tpl",
      param($current_mode, $current_mode) ParamList)
)
```

Next, we define an expression that chooses a template rule for the current mode. It adheres to apply-tpl structure, but only template rules with the specified mode or with special mode #all are included in this case.

Definition 6.18 *Let* $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ *be an XSLT stylesheet,* $m \in M$ *be a mode, and* $n = |all(m)|$ *be the number of template rules with mode* m *or with special mode* `#all`. *Then, with* $ApplyMode(m)$ *we denote the following XSLT expression.*

```
choose(
  when( some $node in patt(σS(all(m),1)) satisfies . is $node,
    BODYMode(σS(all(m),1)))
  ...
  when( some $node in patt(σS(all(m),n)) satisfies . is $node,
    BODYMode(σS(all(m),n)))
  otherwise( BuiltIn(ParamList) )
)
```

Expression $patt(\sigma_S(all(m), i))$ in the above definition denotes a pattern of template rule t, where $mode(t)$ is either m or special mode #all and template rule t is i-th among such template rules in template choosing order τ.

Further, we define the replacement of $ApplyTpl(S)$ for a stylesheet without modes with $ApplyTplMode(S)$ for a stylesheet with modes. It chooses the set of template rules, among which we search for a match, according to the current mode.

Definition 6.19 *Let* $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ *be an XSLT stylesheet, where* $M = \{m_1, \ldots, m_n\}$ *is a set of modes. Then, with* $ApplyTplMode(S)$ *we denote the following named template.*

```
template-named("apply-tpl",
  choose(
    when( $current_mode = #default,
      ApplyMode(#default))
```

1: **procedure** *RemoveTemplateRulesWithModes*($S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$)
2: **for all** $t \in T \cup N \cup F$ **do**
3: $body(t) \leftarrow BODYMode(t)$
4: **end for**
5: $N \leftarrow N \cup \{ApplyTplMode(S), StartTplModes\}$
6: $P \leftarrow P \cup \{\text{"current_mode"}\}$
7: $T \leftarrow \emptyset$
8: $\tau \leftarrow \emptyset$
9: $M \leftarrow \emptyset$
10: **return** S
11: **end procedure**

Figure 29: This procedure takes a stylesheet with template rules and modes and converts it to an equivalent stylesheet without template rules and modes. The initial stylesheet cannot contain other applying instruction than apply-templates for the procedure to work correctly.

```
    when( $current_mode = m1,
      ApplyMode(m1))
    ...
    when( $current_mode = mn,
      ApplyMode(mn))
) )
```

Finally, we define the starting template *StartTplModes*, which is the same as *StartTpl*, but it initializes $current_mode to special mode #default.

Definition 6.20 *With StartTplModes we denote the following named template.*

```
template-named("start-tpl",
  for-each( /, ∅,
    call-template("apply-tpl",
        param($current_mode, #default))
) )
```

We put it all together in procedure *RemoveTemplateRulesWithModes* depicted in Figure 29. We can notice that it is exactly the same as procedure *RemoveTemplateRules* depicted in Figure 28, but $BODY(t)$ is replaced with $BODYMode(t)$, and $ApplyTpl(S)$ is replaced with $ApplyTplMode(t)$, and former initiating template *StartTpl* is replaced with *ApplyTplMode*(.) Further, the set of parameter names is enlarged with current_mode and finally, the set of modes is emptied at the end of this procedure.

Lemma 6.9 *Let* $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ *be an XSLT stylesheet with no applying instruction other than apply-templates. Then, there exists a stylesheet* $S' = \langle T' = \emptyset, N', F', P, U = \emptyset, M = \emptyset, \tau' = \emptyset$ *with no template rules and no modes that is equivalent to* S.

Proof. Let $S' = RemoveTemplateRulesWithModes(S)$. We check that it satisfies the conditions of lemma.

Since the set of mode names M is finite and statically known, we can do the transformation as described by the given procedure, which means that $ApplyTplMode(S)$ and $ApplyMode(m)$ can be constructed, as they are finite.

It is clear from the definitions, which follow the informal definitions of XSLT modes, that current mode is passed with all template calls with new parameter $current_mode.

If the stylesheet S' is initialized with *StartTplModes* named template, then $current_mode is initialized to #default special mode, which is the default behavior.

If apply-templates instruction is called in S with mode m, then only template rules with mode m or with special mode *#all* are considered.

Special care is taken about special mode #current in apply-templates instruction, which means that current mode is used. This is treated by passing current value of $current_mode parameter in apply-templates replacement.

This is how semantics on template rules with modes is defined. □

It is possible to follow the same principle when modifying transformations for apply-imports and next-match instructions. We only have to ensure, that these two applying instruction pass $current_mode parameter unmodified.

In this section, we proved that template rules with modes can be removed from XSLT without loss of the expressive power of the language.

Including, Importing

A stylesheet can be included or imported from another stylesheet. This is covered by include and import instructions, respectively. For explanation of the semantics of these two instructions, see section 6.1.18.

Lemma 6.10 *Let S be a stylesheet. Then, there is an equivalent stylesheet without include and import instructions.*

Proof. If a stylesheet is included, the semantics is the same like if the text of the included stylesheet was pasted into the including stylesheet. Therefore, include instruction is replaced with the contents of an included stylesheet and we are done.

Importing a stylesheet is already covered in two previous sections on removing template rules. □

Patterns

In both XSLT 1.0 and XSLT 2.0, patterns are mainly used to choose the right template rule for the current context item. In both specifications, pattern is a subset of XPath. Whereas XSLT 1.0 limits usage of XPath to forward only axis and no predicates, XSLT 2.0 limits just to forward only axis, while predicates are allowed.

From our transformation of a general stylesheet to a stylesheet without template rules, it can be clearly seen that full XPath expressions can be used when matching the current node against the patterns.

Conclusion

In this section, we proved that a stylesheet with template rules can be replaced with an equivalent stylesheet without template rules, which is achieved with

replacing template rules with named templates. We formalize our result in the following theorem.

Theorem 6.1 *Let* $S = \langle T, N, F, P, U = \emptyset, M, \tau \rangle$ *be an XSLT stylesheet. Then, there is a stylesheet* $S' = \langle T' = \emptyset, N', F', P', U' = \emptyset, M' = \emptyset, \tau' = \emptyset \rangle$ *that is equivalent to* S.

Proof. We proved a removal of apply-templates instruction from a stylesheet with modes in Lemma 6.9, and we outlined a removal of next-match and apply-imports in a rather detailed manner in section Template Rule Choosing Mechanism. □

Therefore, we do not use template rules and instructions apply-templates, next-match, apply-imports in XSLT Core.

6.3.5 Functions and Named Templates

A body of a function, as well as a body of a named template is defined with a sequence constructor. A function, as well as a named template can be called and passed parameters to. Where is the difference? In this section, we show that they have the same expressive power.

As we explained in introduction to XSLT, the difference is twofold. First, unlike named templates, functions are not passed the current context. Thus, we do not now the current template rule, current mode, current context position, etc. in a function. Second, functions are not passed tunnel parameters.

Lemma 6.11 *Functions and named templates have the same expressive power in XSLT.*

Proof. First, let's consider calls to functions and named templates. A function can be called from within an XPath expression, while a named template can be called from a sequence constructor. As we proved in section 6.2.1, XPath expressions and sequence constructors have the same expressive power. Therefore, we can always call both a function or a named template.

Second, we study the expressiveness of functions and named templates. Surely, named templates can express at least the same as functions, because they are defined like functions with a sequence constructor and they like functions can be passed parameters to. Further, they have more information: context, and tunnel parameters.

Next, we study only if functions can express the same as named templates.

Let's start with tunnel parameters. In Lemma 6.5 we proved that tunnel parameters can be replaced with non-tunnel parameters. Therefore, we do not have to take care about them as we can transform a stylesheet to an equivalent one without tunnel parameters. In such a transformed stylesheet, we have to bother only with context, which we inspect next.

Let's examine the evaluation context now. According to the XSLT specification, the context consists of two parts: i) *focus* that keeps the currently evaluated node and its position in a sequence being currently processed, and ii) additional context variables.

The focus is the same as for XQuery, it consists of the following.

- *Context item*, which represents the current node or atomic value being processed.

- *Context position,* which represents the position of current item in the currently processed sequence.
- *Context size,* which represents the number of items in the currently processed sequence.

Current item can be found out using the current() built-in function in a function body. Other focus items cannot be addressed in a function.

Additional context variables are defined in XSLT specification to be the following. We show that every variable either does not affect the expressive power of the language, or is not needed in XSLT Core, because a mechanism it refers to is removed from the core language.

- *Current template* refers to the current template rule being processed. This context variable is needed only to define semantics of instructions apply-imports and next-match. Since we already proved that these instructions can be removed in section 6.3.4, this variable is not needed anymore.
- *Current mode* keeps the mode of the current template. This is needed to correctly evaluate special mode #current. Since template rules even with modes can be removed from the language, as we proved in section 6.3.4, the notion of current mode is not needed in the context anymore.
- *Current group* represents a sequence of items that are processed collectively in one iteration of the for-each-group instructions. Since grouping instructions are proved to be only syntactic sugaring and therefore are removed from the core language, the notion of current group is not needed in the context. For details see section 6.3.7.
- *Current grouping key* represents the key of the current group in the for-each-group-by instruction. For the same reason as current group, the notion of current grouping key is not needed in the context.
- *Current captured substrings* represents a sequence of strings that result from a match of a string against a regular expression. This context variable is needed only to define semantics of instruction matching-substrings, which we omitted from consideration. Therefore, we do not need the notion of current captured substrings in the context for XSLT Core.
- *Output state* tells whether a final output or only temporary output is constructed. This context property is not interesting from the perspective of the expressive power of the language.

The analysis provided makes it a bit clearer. No additional context variables are needed in a context for the core language. Thus, only focus has to be considered.

We show that a function can be passed the focus. As the context position and size are both integers, they can be passed explicitly to a function using parameters. The context item has not to be passed, since current() function can be used in a function to get it.

Thus, a function can express the same as a named template, because the only distinction between them is that a function is not passed the current focus

and we just showed that a function can be passed everything from the current focus using parameters.

To sum up, functions and named templates have the same expressive power. □

We keep functions in XSLT Core, since they allow to mix XPath expressions with sequence constructors, as proved in section 6.2.1.

6.3.6 Sorting

In this section, we show that we do not need special sorting instructions and language constructs in XSLT Core.

In XSLT 2.0, we have two instructions that use a list of order specifications *SortList*. They are perform-sort instruction, which takes a sequence and a list of order specifications and returns that sequence sorted, and for-each instruction, which sorts the initial sequence prior to iterating over it.

There is also apply-templates instruction that can have a list of order specifications, but we already removed apply-templates instruction from XSLT Core in section 6.3.4 and replaced it with for-each instruction. Therefore, we do not consider apply-templates.

We prove that language constructs for sorting are not needed in XSLT Core in the following way. First, we prove that for-each instruction with order specifications can be replaced with a for-each instruction without order specifications, as we can presort the sequence iterated over with perform-sort instruction. Second, we prove that perform-sort is expressible in XSLT Core. As the semantics of order specifications in XSLT is in fact the same as in XQuery, we use a similar method to express sorting semantics of perform-sort as we used to express sorting semantics of order by clause in XQuery.

Lemma 6.12 *Every instruction for-each(Expr1, SortList, Expr2) can be replaced with an equivalent for-each(Expr3, ∅, Expr2) instruction without order specifications.*

Proof. Recall that *Expr1* is the initial sequence, which is to be sorted according to order specifications *SortList*. The sorted sequence is than iterated and for each item *Expr2* is evaluated.

The lemma says that *Expr3* is the initial sequence that is already sorted according to *SortList*. Trivially, let *Expr3* be perform-sort(*Expr1*, *SortList*). □

From now on, we can assume that a list of order specifications of a for-each instruction is empty. We can assume this without loss of generality due to the above lemma.

Let's now turn our attention to perform-sort. Recall that every order specification is an expression, which is evaluated for each item in the initial sequence to get an order value. The list of order values of every item is used to lexicographically sort the initial sequence.

The idea is to sort with $Sort_R$ expression defined in Definition 4.6 on page 38. Now, as we can mix XPath expressions with sequence constructors, we can use XSLT instructions in it, of course.

We use the same trick as with XQuery sorting transformation. We first construct a stream of tuples, where each tuple contains for each item in the initial sequence a list of order values and the item itself. Then, we sort this stream of

tuples with relation R that defines the same lexicographic order as in XQuery sorting transformation. Finally, we extract only the items themselves from the stream of tuples to get the sorted sequence.

Since the transformation is so similar to XQuery's transformation, we only redefine expressions *Stream*, which constructs the tuple stream, *Tuples*, which extracts tuples from the tuple stream, and *Unwrap*, which extracts the items from the sorted stream. Everything else stays the same, or the differences are trivial.

Definition 6.21 *Let e be a perform-sort(Expr, SortList) instruction, where SortList is a sequence of* $e_1, \ldots, e_n$ *expressions. Then, with* $Stream(e)$ *we denote the following expression.*

```
<stream>
    for-each( Expr, ∅,
        <tuple>
            <ordval> e1 </ordval>
            ...
            <ordval> en </ordval>
            <result> . </result>
        </tuple>
    )
</stream>
```

Notice, that a dot expression in the result tag of the just defined expression refers to the current context item, which is always an item from the initial sequence represented by *Expr* expression, as we iterate over it with a for-each instruction.

Definition 6.22 *Let e be an XSLT expression. Then, with* $Tuples(e)$ *we denote the following XSLT expression.*

```
for-each( e, ∅, ./tuple )
```

Definition 6.23 *Let e be an XSLT expression. Then, with* $Unwrap(e)$ *we denote the following XSLT expression.*

```
for-each( e, ∅, ./result/* )
```

The following theorem justifies a removal of perform-sort instruction together with order specifications from XSLT Core.

Theorem 6.2 *Let e be an perform-sort instruction. Then, there is an XSLT expression* e' *such that e is equivalent to* e' *and* e' *contains no perform-sort instruction nor an order specification.*

Proof. Let $e' = Unwrap(Sort_{R_e}(Wrap(e)))$. The proof is the same as for Theorem 5.1 on page 63. We only have to check that tuples are constructed correctly, and that result is correctly extracted from the sorted stream, and that the characteristic function of a lexicographic order is expressible in XSLT.

First, each tuple is constructed in the right focus, as current context item is the item put into the tuple, and context size is the size of the initial sequence, and context position is the position of current item in the initial sequence. This

is guaranteed by the semantics of for-each instruction that iterates over the initial sequence. So, order values are the values of order specifications evaluated in the right context.

Second, the extraction of tuples from the initial tuple stream and extraction of results from the sorted stream is trivially correct.

Third, since lexicographic ordering represented by expression χ_1^c in Definition 5.7 on page 62 uses only XPath references to order values and boolean operators, it is definitely expressible in XSLT. □

6.3.7 Grouping

In this section, we demonstrate that the grouping XSLT instruction xsl:for-each-group is a syntactic sugar and so it does not belong to our XSLT Core language.

Recall, that xsl:for-each-group instruction has four variants that differ in the way in which items are assigned to groups, as we explained in an introduction to grouping in section 6.1.16 on page 79. These variants are: i) group-by that groups according to a value of an expression, ii) group-adjacent that groups together adjacent items, whose grouping expression evaluates to the same value, iii) group-starting-with, and iv) group-ending-with that group together adjacent items, where start or end of a new group is identified by a match of a grouping pattern, respectively.

For each of these four grouping variants we introduce a single instruction in our non-XML XSLT grammar: for-each-group-by, for-each-group-adjacent, for-each-group-starting-with, and for-each-group-ending with, respectively.

Since the first grouping variant group-by is a way different from the other variants in that it groups together potentially non-adjacent items from the initial sequence, we handle the transformation of a group-by variant separately in a later section Ad Hoc Groups. The transformations of the other three variants that group together adjacent items of the initial sequence are handled in the following section.

Continuous Subsequence Groups

To provide the transformations for adjacent items grouping, we make use of the fact that a group always consists of items that form a continuous subsequence of the initial sequence. The idea is to replace each for-each-group instruction with an expression that gets an initial sequence and a sequence of group starting positions, which is a list of positions of the first item in each group, and returns the same as the replaced instruction. Since for each variant the starting position is identified differently, we define three expressions that return a sequence of group starting positions – for each variant one.

Common Replacement Expression. Let's start with the for-each-group replacement expression, which iterates over a sequence of group starting positions and for each starting position it defines a variable that is assigned the items of the group. Then, the result expression is evaluated, where every expression that refers to current group is replaced with an expression that refers to that variable. Notice, that the current group can be referred to only using current-group() function in XSLT.

First, we define a continuous group iterator that is a formalization of the three grouping instructions that work with continuous subsequence groups.

Definition 6.24 *Let e, g, and r be XSLT expressions. Then, with* continuous group iterator $\langle e, g, r\rangle$ *we denote any of the following expressions.*

```
for-each-group-adjacent(e, g, r)
for-each-group-starting-with(e, g, r)
for-each-group-ending-with(e, g, r)
```

Let's look closer on the definition above. Expression e denotes an initial sequence, i.e. an initial sequence is the result of e evaluation. Expression g denotes so called grouping expression. Notice, that group-starting-with and group-ending-with variants are restricted to grouping patterns, which we generalize to stronger grouping expressions, which we show to be also expressible in XSLT Core. Expression r is a result expression that is evaluated for each group.

Next, we define a replacement for the result expression of a continuous group iterator.

Definition 6.25 *Let r be an XSLT expression, and v be a variable name. Then, with FixCurrentGroup(r, v) we denote an XSLT expression, where each occurrence of current-group() function in r is replaced with expression \$v, excluding occurrences in nested for-each-group instructions.*

Since current-group() function can appear only at the first position of an XPath expression, we can safely replace it in the definition above with a variable reference, which is restricted the same way. Thus, *FixCurrentGroup*(r, v) is a correct XSLT expression.

Next, we define an expression that returns a continuous subsequence from an initial sequence given two group starting positions.

Definition 6.26 *Let e be an XSLT expression, and let i and j be two integer expressions. Then, with Subseq(e, i, j) we denote the following XSLT expression.*

```
fn:subsequence( e, i, j - i )
```

Notice, that fn:subsequence(e, *start*, *length*) is a standard function that returns a continuous subsequence of a sequence represented by expression e. The subsequence starts at position *start* and contains *length* items. On the other hand, our *Subseq* expression uses an index of an item that is right next to the last item in the initial sequence, instead of a number of items in the resulting sequence.

Next, we define a replacement expression for a continuous group iterator.

Definition 6.27 *Let $\langle e, g, r\rangle$ be a continuous group iterator, and let Indices be an XSLT expression that represents a sequence of group starting positions. Further, let currentGroup and pos be new variable names that do not appear in e, r, and Indices. Then, with ContinuousGroupIterator$(\langle e, g, r\rangle$, Indices$)$ we denote the following XSLT expression.*

```
for-each( Indices
    variable( pos, fn:position(),
        if-then-else( $pos = 1,
            (),
            variable( currentGroup, Subseq(e, Indices[$pos − 1], Indices[$pos]),
                FixCurrentGroup(r, currentGroup)
) ) ) )
```

This expression iterates over a list of group starting positions *Indices* and evaluates the result expression r of the former continuous group iterator for each index and so for each group. The result expression is evaluated in the context with variable *currentGroup* set to the current group of items and any reference to current-group() in r is changed to a reference to variable *currentGroup*. The current group is assigned to this variable as a continuous subsequence of the initial sequence e, where the starting position of the subsequence is the value of the preceding group starting position and the ending position is the value of current group starting position. Variable *pos* stores the current index of the current group starting position in *Indices*, so expression *Indices*[\$*pos*] represents the current group starting position.

Notice, that we expect the list of group starting positions to always start with value 1 and to always end with the number of items in the initial sequence plus 1.

Getting Group Starting Positions. Now, we turn our attention to a way we obtain a list of group starting positions. For each variant of continuous group iterator for-each-group-adjacent, for-each-group-starting-with, and for-each-group-ending-with, we get the list of group starting positions in a different way, therefore we introduce here three *GetIndices* expressions – one expression for each variant.

Prior to defining the *GetIndices* expressions, we define a helper expression $Eval(g, e, i)$ that evaluates an expression g in the context of sequence e, where the context item is the i-th item in e. It is used to evaluate the grouping expression of a continuous group iterator.

Definition 6.28 *Let g, e, and i be XSLT expressions. Then, with $Eval(g, e, i)$ we denote the following XSLT expression.*

```
for-each( e,
    if-then-else( fn:position() = i,
        g,
        ()
) )
```

Notice, that we have to be careful using powerful *Eval* expression, as the position argument is evaluated in a context different from the context, from which *Eval* expression is called. For example, *Eval*(g, e, `fn:position()`) does not evaluate expression g in the context of an item in e that is on the current context position. Such an expression evaluates g for *every* item in e.

Now, we define *GetIndices* expression for each variant of a continuous group iterator in the following three definitions. They all return a list of group starting positions. The first item is always 1 and the last item of the result is always a number of items in the initial sequence plus 1, as required by *ContinuousGroupIterator* expression.

Definition 6.29 *Let $\langle e, g, r\rangle$ be a* for-each-group-adjacent*(e, g, r) continuous group iterator. Further, let pos be a new variable name that appears neither in g nor in e. Then, with GetIndices(e) we denote the following XSLT expression.*

```
for-each( e,
    variable( pos, fn:position(),
        if-then-else( $pos = 1 or g != Eval(g,e,$pos - 1),
```

```
            $pos,
            ()
) ) )
fn:count(e) + 1
```

The adjacent variant of *GetIndices*(e) iterates over items of the initial sequence e. It returns the position of the current item if it is the first item in e or if the value of grouping expression g evaluated in the current context is different from the value of g evaluated in the context of the preceding item. This is exactly the semantics of adjacent grouping.

Notice, that we have to store the current context position in a variable *pos*. We cannot put it directly to *Eval* expression, as it would be evaluated in a wrong context, as explained above.

Definition 6.30 *Let* $\langle e, g, r\rangle$ *be a* for-each-group-starting-with*(e, g, r) continuous group iterator. Then, with GetIndices*(e) *we denote the following XSLT expression.*

```
for-each( e,
    if-then-else( fn:position() = 1 or g,
        fn:position(),
        ()
) )
fn:count(e) + 1
```

The starting-with variant of *GetIndices*(e) also iterates over items of the initial sequence e. It returns the position of the current item whenever grouping expression g is evaluated to true and also for the first item in e. Notice, that if g is a pattern that matches the current item in e, then g evaluates to true in the if-then-else instruction.

Notice, that since we do not use *Eval* expression here, additional variable *pos* has not to be introduced to keep the current context position, unlike in the previous definition.

Definition 6.31 *Let* $\langle e, g, r\rangle$ *be a* for-each-group-ending-with*(e, g, r) continuous group iterator. Then, with GetIndices*(e) *we denote the following XSLT expression.*

```
for-each( e,
    choose(
        when( fn:position() = 1, fn:position() )
        when( g, fn:position() + 1 )
) )
if-then-else( Eval(g,e,fn:count(e) ),
    (),
    fn:count(e) + 1
)
```

The ending-with variant of *GetIndices*(e) is the most tricky one. It iterates the initial sequence e, as well as the previous variants. It returns 1 for the first item in e and it returns the value of the current context position plus 1 for every other context item that satisfies grouping expression g. It has to return position plus one, as the ending-with variant of the group iterator uses g to identify the last item in the group, so the starting position of a new group is the current context position plus 1 for the current context item that matches g.

For this variant, we cannot simply add number of items in e plus 1 to the end of the returned sequence, like we did in the *GetIndices* definitions for the first two continuous group iterator variants, as it can happen that even the last item from e matches g and the number of items in e plus 1 is already in the resulting sequence. Therefore we check with the if-then-else instruction, whether grouping expression g matches the last item in e. If it matches, nothing is added to the resulting sequence, as the desired final number is already there, otherwise we add it to the resulting sequence.

Lemma 6.13 *Let S be an XSLT stylesheet. Then, there is an equivalent XSLT stylesheet S' without continuous group iterators.*

Proof. We get an equivalent stylesheet if we replace each occurrence of a continuous group iterator $\langle e, g, r\rangle$ with $ContinuousGroupIterator(\langle e, g, r\rangle, GetIndices(e))$.

We checked in the text above, that *GetIndices* expression returns a sequence of group starting positions that always starts with 1 and always ends with number of items in the initial sequence plus 1.

We also checked in the text above that if *ContinuousGroupIterator* expression is given such a list of group starting positions it returns the same results as the original continuous grouping iterator. □

So far, we proved that the adjacent, starting-with, and ending-with variants of the xsl:for-each-group instruction are expressible with other expressions of XSLT.

Ad Hoc Groups

Now, let's have a look at the *group-by* variant. We cannot use the same idea as before, because a group in this case is generally not a continuous subsequence of the initial sequence. Therefore we follow a different idea. First, we gather a sequence of grouping keys, then for each grouping key, we get a sequence of items in a group identified by that specific grouping key. We store items of a group in a variable, exactly as we do it in the preceding section. Finally, for each group we evaluate a result expression.

The most tricky part is surprisingly the generation of the grouping keys. XSLT specification requires that i) the order of grouping keys have to follow the order of items from the initial sequence that originated the grouping keys, and ii) no duplicates are allowed in the sequence of grouping keys so that if there are two duplicate keys then the former is preserved and the latter is removed.

The tricky part is the removal of duplicate values. Though XQuery 1.0 and XPath 2.0 Functions and Operators specification defines fn:distinct-values() function, we cannot use it, because it does not guarantee, which of the duplicate values is erased from the list. Therefore we define our expression $Uni(S)$ that takes a sequence S and returns the same sequence with all duplicates removed in the required way: from a sequence of items with duplicate values we keep the one that appears first in S.

Definition 6.32 *Let S be an XSLT expression. Further, let item and pos be new variable names that do not appear in S. Then, with $Uni(S)$ we denote the following XSLT expression.*

```
for-each( S,
    variable( item, self::*,
    variable( pos, fn:position(),
        if-then-else( S[self::* = $item and fn:position() < $pos],
            (),
            copy-of( $item )
) ) ) )
```

Expression *Uni* iterates over the initial sequence S. In each iteration, it stores current item from S in variable *item* and it stores the position of the current item within S in variable *pos*. The if-then-else condition tests, whether there is such an item in S that has a value equal to *item* and that appears prior to *item* in S. If true, it means we found a duplicate value, which does not appear first in S, so we do not want it in the result and therefore an empty sequence is returned. Otherwise, there is no other item in S with a value equal to *item* that precedes *item* in S, so we return *item*. Summing up, we check each item in S and return only items that have no preceding duplicate in S, which is exactly what we wanted.

Next, we define an expression *GroupItems* that for a grouping expression, a grouping key, and an initial sequence returns items of a group identified by the grouping key.

Definition 6.33 *Let g, key, and S be XSLT expressions. Then, with $GroupItems_g(S, key)$ we denote the following XSLT expression.*

```
for-each( S,
    if-then-else( g = key,
        copy-of( self::* ),
        ()
) )
```

Expression $GroupItems_g(S, key)$ is rather simple. It iterates over the initial sequence S and if the grouping expression g is evaluated to a sequence that contains the grouping key *key*, then the current context item is returned. So the current context item belongs to a group identified by the grouping key *key*.

Notice, that expressions g and *key* generally evaluate to sequences, though usually expression *key* evaluates to a singleton sequence. Thus, an expression $g = key$ in the definition above is true, if there is a grouping key in a sequence returned by expression g that is equal to at least one item in sequence returned by expression *key*.

Finally, we define a replacement expression *ForEachGroupBy* for the group-by variant of the xsl:for-each-group instruction.

Definition 6.34 *Let* for-each-group-by*(S, g, r) be an XSLT expression. Further, let currentGroup be a new variable name that does not appear in S, g, and r. Then, with $ForEachGroupBy(S, g, r)$ we denote the following XSLT expression.*

```
for-each( Uni(for-each(S,g)),
    variable( key, self::*,
    variable( currentGroup, GroupItems_g(S, key),
        FixCurrentGroup(r, currentGroup)
) ) )
```

The inner for-each instruction generates a grouping key for each item in the initial sequence. The result is a concatenation of grouping key sequences, as the grouping expression generally evaluates to a sequence. To get a list of distinct keys, we use expression *Uni* that removes all duplicate keys in the desired way, as checked earlier.

Thus, the outer for-each instruction then iterates a sequence of distinct grouping keys. We store each grouping key in variable *key* and for each grouping key, we get a sequence of items of its group and assign it to variable *currentGroup*. For each grouping key, and so for each group, the result expression r is evaluated, where each reference to current-group() is replaced with a reference to a variable *currentGroup*.

Lemma 6.14 *Let S be an XSLT stylesheet. Then, there exists an equivalent stylesheet S' with no for-each-group-by instruction.*

Proof. We construct S' from S by replacing each occurrence of for-each-group-by(S, g, r) instruction with $ForEachGroupBy(S, g, r)$ expression that does not contain a for-each-group-by instruction.

In the text above, we carefully inspected that the semantics of every definition provided matches the semantics defined for the for-each-group-by instruction. So, the transformation is correct. □

Conclusion

In the two previous sections, we proved that all variants of xsl:for-each-group instruction are syntactic sugaring. We formalize the result in the following theorem.

Theorem 6.3 *Let S be an XSLT stylesheet. Then, there exists an equivalent stylesheet S' that uses no variant of the xsl:for-each-group instruction.*

Proof. Instruction xsl:for-each-group has four variants, which we represent as four different instructions in our non-XML syntax. We divided these four instructions into two groups: i) continuous group iterators: for-each-group-adjacent, for-each-group-starting-with, for-each-group-ending-with, and ii) for-each-group-by instruction.

Expressibility of continuous group iterators in XSLT Core is proved with Lemma 6.13, while expressibility of for-each-group-by instruction in XSLT Core is proved with Lemma 6.14. □

Due to this theorem, we can safely remove all grouping instruction from our XSLT Core language.

Comments

We can notice that both starting-with and ending-with variants of the grouping instruction have not to be limited to a grouping pattern, but an expression can be used instead and still our transformation works. Thus, there is an unnecessary constraint in these XSLT grouping instructions definition.

We should also note that the context inside the xsl:for-each-group instruction is not well defined in the XSLT specification. Or better, it is not defined

at all and as such the context should stay unchanged from the context outside the instruction call, as the specification claims in its section on context. We find this quite cumbersome and we have two following comments on this.

First, checking the reference implementation of XSLT 2.0 – Saxon [39] – that is written by the author of the XSLT specification himself, we discovered that the grouping instructions change the context in the following way. The context item is set to the first item in the group, the context position is set to the position of a group currently being processed, and the context size is set to zero. Since we cannot manipulate the context values manually and we are limited to use only instructions like for-each, we cannot simulate such a behavior like setting context size to zero, specifically. Thus, this specific behavior is not equivalent and cannot be equivalent to the behavior of our or any other transformation.

Second, what context would be most natural? In this case, we think that the best context sequence would be a sequence of groups, where each group is a sequence of its items. So, the context item would be the current group and context position would be its position within the groups. Then, function current-group() would not be needed, as we have the current group stored in the context and we can refer to it with self::* expression. But a sequence of sequences is inexpressible in the XML Data Model.

To sum up, the for-each-group instruction does not fit nicely into the XML Data Model that XSLT sticks to.

6.3.8 Keys

In this section, we show that keys are only syntactic sugaring in XSLT and as such they do not belong to our XSLT Core.

For introduction on keys see section 6.1.17 on page 80. Here, we only remind that keys are named references that can be navigated with a key() function. The keys are defined with a pattern and a value expression. The result of key(*name*, *S*) function call is evaluated as follows. First, candidate items are found, i.e. items that match the pattern that is declared for key *name*. For each candidate item the value expression is evaluated in the singleton focus based on that candidate item to get key values. If any key value is equal to any item in *S*, then the candidate item is returned.

Notice, that singleton focus based on an item *i* is a focus with context item set to *i* and context position and size set to 1.

First, we formally define a key declaration. Recall, that there can be several key declarations for one key name, where each declaration is a pair of a pattern and a value.

Definition 6.35 *Let* $k \in QName \times (Pattern \times Expr)^n$ *be called a* n-ary key declaration.

Next, we define an expression *Key* that is a replacement for a key() function call. It takes the same arguments as key(): a name of a key and an expression.

Definition 6.36 *Let* $k = \langle name, \{\langle p_1, e_1\rangle \ldots \langle p_n, e_n\rangle\}\rangle$ *be a n-ary key declaration. Further, let values be a new variable name that does not appear in* p_i*, and* e_i *for all i. Then, with* $Key(name, S)$ *we denote the following XSLT expression.*

```
variable( values, S,
    for-each( p1,
        for-each( self::*,
            if( e1 = $values, self::* )
    ) )
    ...
    for-each( pn,
        for-each( self::*,
            if( en = $values, self::* )
) ) )
```

Now, we prove that *Key* expression is a correct replacement for a key() function call.

Lemma 6.15 *Let S be an XSLT stylesheet. Then, there exists an equivalent stylesheet S' that contains neither key-declare instructions nor key() functions.*

Proof. We get stylesheet S' by removing all key-declare instructions and by replacing every key(*name*, *S*) function call with $Key(name, S)$ expression.

Definition 6.36 of $Key(name, S)$ expression is a direct rewrite of the semantics of the key(*name*, *S*) function call. Each outer for-each instruction iterates over items that match one of the patterns p_i. These are the candidate items. Each candidate item is then iterated with the inner for-each instruction to provide singleton focus based on that item. The if instruction returns the candidate item only if any value in *S* is equal to any value in the result of e_i value expression.

Notice, that we store the value of expression *S* in variable *values*. This is to protect the evaluation context of expression *S*. Since for-each expression changes the context, we cannot use expression *S* in the sequence constructor of the for-each expression, as it would be evaluated in a wrong context. Therefore we compute expression *S* before we call for-each instruction and therefore we keep its computed value in variable *value*. □

To sum up, key-declare instruction and key() function do not belong to our XSLT Core language.

6.3.9 Copying

As explained in section 6.1.14 on page 78, XSLT defines instructions value-of, copy-of, and copy that allow copying of data from the source document to the result.

Let's briefly remind the semantics of each instruction. The value-of(*Expr*) instruction executes and XPath expression *Expr* and returns a new text node. The copy-of(*Expr*) instruction makes a deep copy of *Expr*. Finally, the copy(*Expr*) instruction makes a shallow copy of the current context item, where *Expr* is evaluated only for document and element nodes, for whom it constructs the content.

In this section, we show that instruction copy-of is expressible with instruction copy, which in turn is expressible with value-of instruction. Further, the value-of instruction is expressible with an XPath expression, but as we need a syntactical bridge between XPath and XSLT, we keep it in XSLT Core.

Knihovna Mat.-fyz. fakulty
Informatické oddělení
Malostranské náměstí 25
118 00 Praha 1

Deep Copy

Here, we show that deep copy, which is realized with copy-of expression, can be transformed to an expression that uses the copy instruction.

The idea is to split a deep copy of the copy-of instruction to i) a recursive traversal and ii) a shallow copy, where the recursive traversal is realized with a recursive call to a named template and the shallow copy is realized with the copy instruction.

First, we define a new named template that copies the current context node with all its children and attributes.

Definition 6.37 *Let* $S = \langle T, N, F, P, U, M, \tau \rangle$ *be an XSLT stylesheet, and* copy-of-tpl *be a name that is different from all names of named templates N. Then, with CopyOfTpl(S) we denote the following XSLT named template.*

```
template-named( "copy-of-tpl", (),
  copy(
      for-each( child::* | attribute::* ,
        call-template( "copy-of-tpl", () )

) ) )
```

We can see, that this expression defines a named template that makes a shallow copy of the current context item. If the current context item is a document or element node, then the contents of the copied node are the deeply copied children and attributes of the current node. So, for a document or element node its children and attributes are copied, recursively. Notice, that copying of namespace nodes is handled by instruction copy itself, which we discuss later.

Next, we define a replacement expression for a copy-of(e) instruction.

Definition 6.38 *Let e be an XSLT expression. Then, with CopyOf(e) we denote the following XSLT expression.*

```
for-each( e,
    call-template( "copy-of-tpl", () )
)
```

Finally, we state that it is possible to remove all occurrences of copy-of instruction and still to have an equivalent stylesheet.

Lemma 6.16 *Let S be an XSLT stylesheet. Then, there exists an equivalent stylesheet* S' *with no copy-of instruction.*

Proof. We get S' by modifying S. We add *CopyOfTpl*(S) named template to a set of named templates and we replace each occurrence of copy-of(e) with *CopyOf*(e).

Notice, that in *CopyOf*(e) expression e is evaluated first, which results in a sequence of nodes. Then, template copy-of-tpl is called for each node with one of the nodes set as the current context item. Template copy-of-tpl then copies the current context item and for a document or element node it recursively copies its children and attributes. The namespace nodes are silently copied with copy instruction, as well. □

Thus, instruction copy-of does not belong to our XSLT Core language.

Shallow Copy

Here, we show that the copy instruction is expressible with simpler constructs of XSLT Core.

The idea is to test a node kind of the current context item with a node test function and to use an appropriate node constructor to make a copy of the context item.

First, we define an expression *CopyNamespaces* that copies all namespace nodes of the current context item. We define it separately to simplify the replacement expression for the copy instruction.

Definition 6.39 *With CopyNamespaces we denote the following XSLT expression.*

```
variable( "currentItem", self::*,
    for-each( in-scope-prefixes(.),
        namespace( self::*,
            namespace-uri-for-prefix( self::*, $currentItem )
) ) )
```

As the namespace axis is deprecated in XPath 2.0, we cannot simply iterate over namespace nodes of the current context item. Instead, we store the current context item in a new variable currentItem and iterate over namespace prefixes. For each prefix, we construct a new namespace node with the same prefix and with a URI set to the URI associated with the prefix in an input document.

Next, we define expression *Copy* that serves as a replacement expression for an occurrence of the copy instruction.

Definition 6.40 *Let e be an XSLT expression. Then with Copy(e) we denote the following XSLT expression.*

```
choose(
    when( self::document-node(),
        result-document(
            CopyNamespaces
            e
    ) )
    when( self::element(),
        element( name(), namespace-uri(),
            CopyNamespaces
            e
    ) )
    when( self::attribute(),
        attribute( local-name(), value-of( self::* ) )
    )
    when( self::text(),
        value-of( self::* )
    )
    when( self::comment(),
        comment( value-of( self::* ) )
    )
    when( self::processing-instruction(),
        processing-instruction( name(), value-of( self::* ) )
    )
    otherwise(
```

```
        namespace( name(), value-of( self::* ) )
) )
```

Lemma 6.17 *Let S be an XSLT stylesheet. Then, there is an equivalent stylesheet S′ that uses no copy instruction.*

Proof. We get stylesheet S' by replacing each occurrence of copy(e) in stylesheet S with $Copy(e)$.

The transformation is pretty straightforward. As we said earlier, we test a node kind of the current context item. When we match a specific node kind, we use its constructor expression to create a new node of the same kind. Namespaces are copied for document and element nodes as required by the semantics of the original copy instruction. If the new node is either a document or element node, then expression e is evaluated to construct its content. □

Notice, that the otherwise branch of the choose instruction can be selected only for namespace nodes, since every other node kind is matched by some of the when branches.

Since copy instruction can be replaced with simpler constructs of XSLT Core, it does not belong to XSLT Core.

XPath Bridge

In this section, we explain, why the value-of instruction stays in XSLT Core.

In section 6.2.1 on page 83, we proved that sequence constructors of XSLT can be freely mixed with XPath expression in our non-XML syntax for XSLT. If we study our arguments thoroughly, we find that when we use an XPath expression in a place where a sequence constructor is expected, the copy-of instruction is used in XML syntax for XSLT.

Since, we proved in the previous two sections that copy-of instruction is expressible with copy instruction, and that copy instruction is in turn expressible with value-of instruction, we have to keep the value-of instruction in XSLT Core, as it provides a bridge between XPath and the XML syntax for XSLT.

The above definition of *Copy* explicitly uses instruction value-of to mark out all places, where a sequence constructor is expected, but where we need to put an XPath expression.

6.3.10 Sequences

The sequence instruction creates a sequence of nodes. It is often used to create a sequence that consists of existing nodes, or to create a sequence with an XPath expression like (1 to 4).

Since we can mix XPath expressions with XSLT sequence constructors in our non-XML syntax, we can throw the sequence instruction away, because it is not needed in XSLT Core.

6.3.11 The Core Syntax

Figure 30 provides the syntax of XSLT Core. Compare it with the syntax of the considered fragment of XSLT 2.0 in Figure 23 on page 82 to see the amount of syntactic sugaring in the full language.

In section 6.3.2, we explain the replacement of the choose and if instructions with a single if-then-else instruction.

In section 6.3.3, we justify removal of parameter declarations. We also show in this section how tunnel parameters can be simulated with usual parameters.

In section 6.3.4, we prove that the whole template choosing mechanism together with template rules can be removed from the core language. The proof includes stylesheet inclusion and import. Proving template rules are only syntactic sugar, we can remove from XSLT Core the following instructions: template-rule, import, include, apply-templates, apply-imports, and next-match.

In section 6.3.5, we further simplify the language by removing named templates. As their expressive power is the same as the expressive power of function, we can do with functions only.

In section 6.3.6, we show that order specifications are not needed in the for-each instruction, because a sequence iterated by the for-each instruction can be presorted with perform-sort instruction. Further, as XPath is capable of sorting sequences, the perform-sort instruction can be replaced with an equivalent $Sort_R$ expression.

In section 6.3.7, we prove that all variants of grouping instructions are expressible with other language constructs.

In section 6.3.8, we justify removal of keys form XSLT Core.

In section 6.3.9, we show that both copy and copy-of instructions are expressible with other language constructs and so they can be removed from XSLT Core.

In section 6.3.10, we justify removal of the sequence instruction from the core language.

Let's inspect what has left. Each instruction has its well defined meaning with the semantics completely different from the other instructions. The for-each instruction is a sequence iterator that is capable of changing the current context. The value-of instruction provides a syntactical bridge between XPath and XSLT in XML syntax. Finally, the variable instruction binds a variable name to a value.

6.4 XSLT Formal Semantics

In this section, we define formal semantics for XSLT. As we cut the number of XSLT instructions down in the previous section, we provide the formal semantics only for XSLT Core instructions.

The main semantics function is denoted by $\mathcal{E}$ and is defined in Figure 31. It takes an expression and returns a sequence of items. In the definition, we use semantic function $\mathcal{W}$ that has the same semantics as in XQuery (only instead of XQuery expressions it takes XSLT expressions).

Notice, that we do not have to define sorting semantics, as there is no sorting instruction in XSLT Core due to the huge power of XPath.

Let's describe the hard-to-read formulas of the formal semantics definition.

First, the result of for-each(e_1, e_2) in the given context C is a sequence of items defined with a set of items S and an order relation $\prec_S$ on them. Let's check the items first. S_1 denotes a sequence that is a result of evaluating the e_1 expression in the current context C, y denotes an items from S_1, C_1 is a context that is changed from the current context by setting a focus on y in S_1, finally x

Stylesheet	::=	stylesheet(Function* Expression)
Function	::=	function(QName, Bool, ParamDecl*, Expression)
Expression	::=	Expression*
		| Instruction
		| Constructor
		| XPathExpr
Instruction	::=	for-each(Expression, Expression)
		| value-of(Expression)
		| variable(QName, Expression, Expression)
Constructor	::=	attribute(Expression, Expression, Expression)
		| element(Expression, Expression, Expression)
		| namespace(Expression, Expression)
		| comment(Expression)
		| processing-instruction(Expression, Expression)
		| text(PCData)
		| result-document(Expression)
Param	::=	param(QName, Expression)

Figure 30: XSLT Core language syntax.

denotes an item of a sequence that is a result of evaluating the e_2 expression in context C_1. We say that x is generated by y.

Next, we focus on the resulting order of items in S. Symbols y_1 and y_2 denote ys, form which x_1 and x_2 are generated, respectively. The formula says that x_1 is to precede x_2 in the resulting order, whenever y_1 precedes y_2 in S_1. As it can happen that x_1 and x_2 are generated from the same item $y = y_1 = y_2$, the last line specifies that in such a case the order of x_1 and x_2 in $\mathcal{E}[\![e_2]\!]C_1$ is preserved.

The result of the value-of expression e is the result of expression e. Notice, that this is only a syntactical bridge.

The result of the variable expression variable(v, e_v, e) is the result of expression e that is evaluated in the context with a variable name v bound to a value that is the result of evaluating e_v expression.

A list of expression is evaluated so that each expression is evaluated in the same context C and the resulting sequences are concatenated together to form the result.

We do not provide formal semantics for node constructors, as this is rather straightforward so we do not see much sense in doing so. Also our formalism is not powerful enough to express them.

6.5 Conclusion

XSLT is nowadays a well accepted language that is suitable for simple transformations of an XML document. The more the structure of an input XML document resembles the structure of an output document, the easier is to write the transformation with an XSLT stylesheet.

As to be handy for stylesheet writers, XSLT offers a huge number of instructions. In this work, we proved many of them to be only a syntactic sugaring,

$\mathcal{E}[\![\texttt{for-each}(e_1, e_2)]\!]C$	$S = \{ x \mid S_1 = \mathcal{E}[\![e_1]\!]C,\ y \in \mathcal{I}_{S_1},\ C_1 = focus(C, y, S_1),\ x \in \mathcal{I}_{\mathcal{E}[\![e_2]\!]C_1} \}$ $\prec_S = \{ \langle x_1, x_2 \rangle \mid y_1, y_2 \in S_1 = \mathcal{I}_{\mathcal{E}[\![e_1]\!]C},\ C_1 = focus(C, y_1, S_1),\ C_2 = focus(C, y_2, S_1),\ x_1 \in \mathcal{I}_{\mathcal{E}[\![e_2]\!]C_1},\ x_2 \in \mathcal{I}_{\mathcal{E}[\![e_2]\!]C_2},\ y_1 \sqsubset_{\mathcal{E}[\![e_1]\!]C} y_2,\ y_1 = y_2 \text{ implies } x_1 \sqsubset_{\mathcal{E}[\![e_2]\!]C_1} x_2 \}$
$\mathcal{E}[\![\texttt{value-of}(e)]\!]C$	$\mathcal{E}[\![e]\!]C$
$\mathcal{E}[\![\texttt{variable}(v, e_v, e)]\!]C$	$\mathcal{E}[\![e]\!]C(v \rightarrow \mathcal{E}[\![e_v]\!]C)$
$\mathcal{E}[\![e \text{ ExpressionList}]\!]C$	$\mathcal{E}[\![e]\!]C \circ \mathcal{E}[\![\text{ExpressionList}]\!]C$

Figure 31: Formal Semantics of XSLT Core. Each expression evaluates to a sequence of items $\langle S, \prec_S \rangle$.

which includes e.g. instructions for sorting, copying, grouping. Further, we proved that even the most recognized items of a stylesheet – XSLT template rules – can be expressed with simpler constructs of the language.

The result of our work on XSLT is summarized in the following two achievements.

- We identified XSLT Core that has the same expressive power as XSLT 2.0. For a list of XSLT 2.0 features that we do not consider see section 6.2.2.
- For the XSLT Core language we defined formal semantics in the unified semantics framework that we use also for XPath 2.0 and XQuery 1.0 in the previous chapters.

Remarkably, we do not prove whether XSLT Core is a minimal core language for XSLT.

7 Conclusions

In this work, we formally studied properties of the upcoming standard query languages for XML: XPath 2.0, XQuery 1.0 and XSLT 2.0. We provided a formal semantics framework, which we used as a unified tool to express semantics of these three languages.

We proved a surprising fact that XPath can sort arbitrary sequences using general sorting criteria [33].

Further, we proved that the XPath's ability to sort sequences can be used to express sorting semantics of XQuery, thus, order by clause does not belong to XQuery Core and the data model has not to be expanded to express formal semantics of sorting. This is a contradiction to the expectations of the authors of the specification [22].

Finally, we provided XSLT Core as a core language for XSLT. We proved that our XSLT Core is equally expressive as XSLT 2.0 by providing a transformation of every non-core XSLT feature to XSLT Core. We left out from consideration several instructions that are identified in 6.2.2. We defined formal semantics for XSLT Core and thus for XSLT.

7.1 Future Work

Having the formal semantics of XQuery and XSLT expressed in a unified framework, it should not be hard now to show that these two languages have really the same expressive power.

References

[1] Serge Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, 1997.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[3] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible stylesheet language (XSL) version 1.0, 1999. W3C Recommendation. `http://www.w3.org/TR/xsl/`.

[4] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 79–95, London, UK, 2003. Springer-Verlag.

[5] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML path language (XPath) 2.0, 2005. W3C Working Draft. `http://www.w3.org/TR/xpath20/`.

[6] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of XSLT. *Inf. Syst.*, 27(1):21–39, 2002.

[7] Paul V. Biron and Ashok Malhotra. XML Schema part 2: Data types second edition, 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-2/`.

[8] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML query language, 2005. W3C Working Draft. `http://www.w3.org/TR/xquery/`.

[9] T. Bray, D. Hollander, and A. Layman. Namespaces in XML 1.0. W3C Recommendation. `http://www.w3.org/TR/REC-xml-names`.

[10] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.1. W3C Recommendation. `http://www.w3.org/TR/xml-names-11`.

[11] T. Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Franois Yergeau, and John Cowan. Extensible markup language (XML) 1.0, 1998. W3C Recommendation. `http://www.w3.org/TR/REC-xml-19980210/`.

[12] T. Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Franois Yergeau, and John Cowan. Extensible markup language (XML) 1.1, 2004. W3C Recommendation. `http://www.w3.org/TR/xml11/`.

[13] Peter Buneman. Semistructured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, Tucson, Arizona, 1997.

[14] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 505–516, Montreal, Quebec, Canada, 1996. ACM Press.

[15] R. G. G. Cattell. *The Object Database Standard: ODMG-93 (Release 1.1)*. Morgan Kaufmann, 1994.

[16] Donald Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In Dan Suciu and Gottfried Vossen, editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[17] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis project: Integration of heterogenous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[18] James Clark. XSL transformations (XSLT) version 1.0, 1999. W3C Recommendation. `http://www.w3.org/TR/xslt/`.

[19] James Clark and Steve DeRose. XML path language (XPath) version 1.0, 1999. W3C Recommendation. `http://www.w3.org/TR/xpath20/`.

[20] John Cowan and Richard Tobin. XML information set, 2004. W3C Recommendation. `http://www.w3.org/TR/xml-infoset/`.

[21] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, 1998.

[22] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Corporation, Kristoffer Rose, Michael Rys, Jerome Simeon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics, 2005. W3C Working Draft. `http://www.w3.org/TR/xquery-semantics/`.

[23] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer second edition, 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-0/`.

[24] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3):4–11, 1997.

[25] Mary Fernandez, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 data model, 2005. W3C Working Draft. `http://www.w3.org/TR/xpath-datamodel/`.

[26] Mary Fernandez, Jerome Simeon, and Philip Wadler. An algebra for XML Query. *Lecture Notes in Computer Science*, 1974:11, 2000.

[27] Mary F. Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Washington, DC, USA, 1998. IEEE Computer Society.

[28] Damien Fisher, Franky Lam, and Raymond K. Wong. Algebraic transformation and optimization for XQuery. *Lecture Notes in Computer Science*, 3007:201–210, 2004.

[29] Achille Fokoue, Kristoffer Rose, Jerome Simeon, and Lionel Villard. Compiling XSLT 2.0 into XQuery 1.0. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 682–691, New York, NY, USA, 2005. ACM Press.

[30] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 189–202, Washington, DC, USA, 2002. IEEE Computer Society.

[31] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[32] Pavel Hlousek. MailQL: A query language for an email message base, 2000. Master thesis. Charles University. In Czech.

[33] Pavel Hlousek. XPath 2.0: It can sort! In Daniela Florescu and Hamid Pirahesh, editors, *Proceedings of the Second International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>, in cooperation with ACM SIGMOD*, Baltimore, Maryland, USA, 2005.

[34] Pavel Hlousek and Jaroslav Pokorny. Refining OEM to improve features of query languages over semistructured data. In J. Grundspenkis et al, editor, *Information Systems Development: Advances in Methodologies, Components, and Management*. Kluwer Press, 2002.

[35] Joint Technical Committee ISO/IEC JTC1, Information technology, ISO/IEC 10179:1996 Information technology – Processing languages – Document Style Semantics and Specification Language (DSSSL), 1996.

[36] ISO 8879. Information processing – Text and office systems – Standard Generalized Markup Language (SGML), 1986.

[37] ISO/IEC: Information technology – Database languages – SQL, 1992.

[38] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A tree algebra for XML. In Giorgio Ghelli and Gösta Grahne, editors, *DBPL*, volume 2397 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2001.

[39] Michael Kay. Saxon. `http://saxon.sourceforge.net/`.

[40] Michael Kay. XSL transformations (XSLT) version 2.0, 2005. W3C Working Draft. `http://www.w3.org/TR/xslt20/`.

[41] Stephan Kepser. A simple proof for the Turing-completeness of XSLT and XQuery. In *Extreme Markup Languages*, 2004.

[42] Claude Kirchner, Zhebin QIAN, Preet Kamal SINGH, and Jurgen STUBER. Xemantics: a rewriting calculus-based semantics of XSLT, 2001. Technical Report A01-R-386, LORIA.

[43] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 functions and operators, 2005. W3C Working Draft. `http://www.w3.org/TR/xpath-functions/`.

[44] Maarten Marx. Conditional XPath, the first order complete XPath dialect. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 13–22, Paris, France, 2004. ACM Press.

[45] Maarten Marx. XPath with conditional axis relations. In *Advances in Database Technology - EDBT 2004: 9th International Conference on Extending Database Technology*, pages 477–494, Heraklion, Crete, Greece, 2004. Springer-Verlag.

[46] Maarten Marx. First order paths in ordered trees. In *Database Theory - ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings*, Edinburgh, UK, 2005. Springer-Verlag.

[47] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *PODS '00: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems*, pages 11–22, Dallas, Texas, USA, 2000. ACM Press.

[48] Leonid Novak and Alexandre Zamulin. Algebraic semantics of XML schema. *Lecture Notes in Computer Science*, 3631:209–222, 2005.

[49] Stelios Paparizos, Shurug Al-Khalifa, Adriane Chapman, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native system for querying XML. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD Conference*, page 672. ACM, 2003.

[50] Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Querying semistructured heterogeneous information. In *DOOD '95: Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 319–344, London, UK, 1995. Springer-Verlag.

[51] Jonathan Robie, J. Lapp, and D. Schach. XML query language (XQL), 1998.

[52] Graeme Smith. *Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[53] Lincoln D. Stein and Jean Thierry-Mieg. AceDB: A genome database management system. *Comput. Sci. Eng.*, 1(3):44–52, 1999.

[54] H. S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures second edition, 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

[55] Philip Wadler. A formal semantics of patterns in XSLT and XPath. *Markup Lang.*, 2(2):183–202, 2000.

[56] Hong Li Yang, Jin Song Dong, Ke Gang Hao, and Jun Gang Han. Formalizing semantics of XSLT using Object-Z. *Lecture Notes in Computer Science*, 2642:120–131, 2003.

[57] Hong Li Yang and Ke Gang Han, Jun Gang Hao. The common semantic constructs of XML family. *Lecture Notes in Computer Science*, 2885:416–431, 2003.

Knih... fakulty
Info... oddělení
Malostranské náměstí 25
118 00 Praha 1