Charles University in Prague

Faculty of Mathematics and Physics

**Master Thesis**

Mário Mikula

# Inference of an XML Schema with the Knowledge of XML Operations

Department of Software Engineering

| | |
|---|---|
| Supervisor of the master thesis: | RNDr. Irena Mlýnková, Ph.D. |
| Study programme: | Informatika |
| Specialization: | ISS |

Praha, 2011

I would like to thank my supervisor RNDr. Irena Mlýnková, Ph.D., for her helpful advices, corrections and suggestions.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ........ date ............

Název práce: Odvozování XML schémy se znalostí XML operací

Autor: Mário Mikula

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.

Abstrakt: V poslednej dobe bolo vyvinutých množstvo metód zaoberajúcich sa automatickým odvodzovaním XML schémy. Väčšina z nich ale používa XML dokumenty ako jediný vstup. My sa v tejto práci zameriavame na rozšírenie odvodzovania zahrnutím XML operácií, konkrétne XQuery dotazov. Diskutujeme ako je možné využiť XQuery dotazy na vylepšenie procesu odvodzovania a navrhujeme algoritmus založený na vybraných vylepšeniach, ktorý rozširuje existujúcu metódu hľadania kľúčov a ktorý môže byť začlenený do metód odvodzujúcich tzv. *počiatočnú gramatiku*. Implementovaním algoritmu sme vytvorili prvé kompletné riešenie odvodzovania XML schémy, ktoré využíva XML dokumenty spolu s XML operáciami.

Klíčová slova: XML, XML schéma, odvozování, XQuery, dotazy, XML operace
Title: Inference of an XML Schema with the Knowledge of XML Operations

Author: Mário Mikula

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, Ph.D.

Abstract: Recently, plenty of methods dealing with automatic inference of XML schema have been developed, however, most of them utilize XML documents as their only input. In this thesis we focus on extending inference by incorporating XML operations, in particular XQuery queries. We discuss how can XQuery queries help in improving the inference process and we propose an algorithm based on chosen improvements, extending an existing method of a key discovery, that can be integrated to methods inferring so-called *initial grammar*. By implementing it, we created the first solution of XML schema inference using XML documents along with XML operations.

Keywords: XML, XML schema, inference, XQuery, queries, XML operations

# Contents

# Chapter 1

# Introduction

The Extensible Markup Language (XML) [6] has become a popular standard for data representation and exchange. Its prevalence is based on its simplicity and, at the same time, its flexibility and expressive power.

With XML used to represent data themselves, it is often needed or convenient to somehow specify the structure of the represented data, their format, internal relations or restrictions, etc. In order to this need, several so-called XML schema languages (XML schemata) have been created. The most used of them are Document Type Definition (DTD) [6] and XML Schema (or XSD as XML Schema Definition) [34, 30, 20] proposed by the World Wide Web Consortium (W3C).

Despite a conveyed encouragement to use an XML schema along with an XML data representation, in practise it is done sparsely. Commonly, XML documents are not assigned with their respective XML schema at all or the schema is outdated due to modifications done to a structure of data without updating the schema [23].

Recently in reaction to this situation, a significant number of approaches dealing with automatic construction of an XML schema has been proposed. The aim is to exploit a provided set of XML documents and infer an XML schema, so that the XML documents are valid against it. In addition, the inferred schema should be reasonable in views of human-readability, preciseness and conciseness. Most of published approaches to XML schema inference are of this type - the input is a set of XML documents. They are based on various ideas and can be classified by several aspects as discussed in Chapter 3.

Besides the mentioned type, some approaches that utilize other or additional sources have been developed, for example [24] and [17]. If there are available sources like an out-dated XML schema, operations upon the XML data such as a set of XQuery [11] queries, or other, these sources can be exploited to refine the process of inference. The refinement can be achieved in various aspects such as increasing the speed of the process, getting a more precise, more concise or more readable result or inference of statements about the data which cannot be (easily) extracted from the data themselves.

Recently, the main effort has been focused on a research of the approaches that utilize XML documents, and thus, there are only few approaches of the latter type (as also discussed in Chapter 3), leaving a wide space for a possible future research and improvements.

## 1.1 Aim of this Work

Our aim is to perform a research on the problem of inference of an XML schema for the given set of XML data in a situation when we are provided also with a set of related operations (XML queries, XSLT scripts [8] etc.).

Firstly, we analyze existing inference solutions in general and discuss their advantages and disadvantages. Then, we identify and discuss information that can be extracted from a given set of XML operations and how they can be exploited to achieve more precise and realistic XML schema.

The work results in a proposal of own approach involving the improvements, its implementation and suitable experiments that show its advantages.

## 1.2 Structure of the Thesis

The thesis is structured as follows.

In Chapter 2, we introduce XML technologies used in this thesis and we briefly explain their basic principles.

Chapter 3 contains summaries of several existing approaches of XML schema inference. One of them proposes an utilisation of XQuery queries, which interests us the most.

We made an analysis of XQuery technology from a view of XML schema inference, where we suggest several ideas on how XQuery queries can be exploited in the inference process. This is a content of Chapter 4.

In Chapter 5, we discuss some essential questions that we needed to decide before a development of an algorithm.

The algorithm itself is proposed in Chapter 6. And Chapter 7 describes how to combine results of the algorithm with existing grammar inferring methods of XML schema inference.

An implementation of the proposed algorithm using the jInfer framework is described in Chapter 8. Chapter 9 then discusses performed experiments and their outcomes. And, finally, Chapter 10 concludes.

# Chapter 2

# Used Technologies and Definitions

## 2.1 XML Schema

An XML schema refers to a description of an XML document in terms of its structure and various constraints. Commonly, the XML schema describes element and attribute names, their parent-child relations, their order and type of their content. Other constraints often expressed in the XML schema are restrictions on numbers of occurrences of elements, specification of (non-)obligatory attributes, uniqueness and specification of keys.

Various languages have been proposed to express XML schemata. The most known are Document Type Definition (DTD) [6] and XML Schema Definition (XML Schema, XSD) [34, 30, 20] which are briefly described in the following sections. Another examples of the XML schema languages are RELAX NG [10] and Schematron [16].

Validity of an XML document against its XML schema expresses whether the document is well-formed [6] and, at the same time, whether it conforms to the XML schema.

### 2.1.1 An XML Example

To demonstrate the described technologies, we introduce a part of a simple XML document (see Figure 2.1). It represents books in a bookstore. Each book has a mandatory id, a title, a list of authors and an optional ISBN.

### 2.1.2 DTD

Document Type Definition (DTD) expresses the structure of XML documents by declarations of elements. An element has its name and a content declared using

```
<bookstore>
  <book id="b1">
    <title>
      Compilers: Principles, Techniques, and Tools
      (2nd Edition)
    </title>
    <authors>
      <author>Alfred V. Aho</author>
      <author>Monica S. Lam</author>
      <author>Ravi Sethi</author>
      <author>Jeffrey D. Ullman</author>
    </authors>
  </book>
  <book id="b2">
    <title>XQuery</title>
    <authors>
      <author>Priscilla Walmsley</author>
    </authors>
    <isbn>0596006349</isbn>
  </book>
</bookstore>
```

Figure 2.1: A simple XML example

syntax `<!ELEMENT` *name content*`>`.

The content of an element can be denoted by `EMPTY` for an empty element, `ANY` for any content, `(#PCDATA)` allowing only textual content (without any other subelements), or specified using regular expressions. Names of subelements are combined using operators (`|`, `+`, `*`, `?` and `,`(comma)). To express the mixed content `#PCDATA` can be used in an alternation list with the subelement names and this alternation has to be enclosed in `*` operator.

Attributes of an element are specified in an attribute list `<!ATTLIST` *element_name attribute_name type default_value*`>`. Each attribute has its name, its type and its default value or definition of obligation of occurrence. The type can be an enumeration of values `(`$value_1$ `|` $value_2$ `|` $\ldots$ `|` $value_n$`)` or one of the following values.

- `CDATA` Character data - any string.
- `ID` A unique identifier.
- `IDREF` An ID reference - a value of the ID.
- `IDREFS` A space-separated list of ID references.
- `NMTOKEN` A valid XML name.
- `NMTOKENS` A space-separated list of valid XML names.
- `ENTITY` An entity.
- `ENTITIES` A space-separated list of entities.
- `NOTATION` A name of a notation.

The default value is either a literal value or one of the following specifiers.

- `#REQUIRED` The attribute is mandatory.
- `#IMPLIED` The attribute is optional.
- `#FIXED` *value* The attribute value is constant *value*.

The DTD also provides other constructs such as declaration of entities and notations not mentioned in this work.

An example of DTD describing the book element from the XML example in Figure 2.1 is shown in Figure 2.2.

```
<!ELEMENT book (title, authors, isbn?)>
<!ATTRLIST book id ID #REQUIRED>

<!ELEMENT title #PCDATA>

<!ELEMENT authors (author+)>

<!ELEMENT author #PCDATA>

<!ELEMENT isbn #PCDATA>
```

Figure 2.2: A simple DTD example

### 2.1.3 XSD

Since the XSD language, containing many constructs and features, is quite comprehensive, we will describe just its basic principles. An important fact is that each XSD instance is a valid XML document. Its root element is always `<schema>` and the XSD instance begins with the following two lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Definition of an element in the XSD is

`<xs:element name="`*name*`" type="`*type*`"/>`

where *type* is either one of the built-in types or a user-defined type. Definition of an attribute is similar.

`<xs:attribute name="`*name*`" type="`*type*`"/>`

Attributes are optional by default. If an attribute is mandatory, it is expressed by adding `use` attribute to its definition.

`<xs:attribute name="`*name*`" type="`*type*`" use="required"/>`

The user-defined type in the XSD language is either a simple type, if it does not contain other elements and attributes, or a complex type otherwise. Attributes are only allowed to be of the simple types. Definition of a simple type is

```
<xs:simpleType name="name">
   type details
</xs:simpleType>
```

The user-defined simple types often serve to restrict the built-in types in various ways such as limiting lengths of strings, allowing only certain values and thus creating an enumeration type, and other.

A complex type is defined in the same way.

```
<xs:complexType name="name">
  type details
</xs:complexType>
```

The complex types can contain many constructs. Subelements are declared using `<xs:sequence>`, `<xs:choice>` and `<xs:all>` schema elements. If the order of the subelements is significant, we use `<xs:sequence>`, where the subelements in an XML instance must occur in the same order as they are defined in the sequence. If the order is not significant, we use `<xs:all>`. Construct `<xs:choice>` is equivalent to the alternation of several elements in the DTD.

Moreover, these three constructs can be nested and combined, assuming the combination is not ambiguous.

Many schema elements (including `<xs:element>`, `<xs:sequence>`, `<xs:choice>`, `<xs:all>`) can be assigned with an occurrence interval. The occurrence interval is expressed using `minOccurs` and `maxOccurs` attributes. For instance, if an element is optional, it definition can be

```
<xs:element name="name" type="type" minOccurs="0" maxOccurs="1"/>
```

An element can have a mixed content (can contain text and other elements at the same time). Such element has to be of a complex type and definition of the type has to contain attribute `mixed` with value `true`. The following example demonstrates the mixed content along with the definition of the element type inside `<xs:element>` schema element.

```
<xs:element name="name">
  <xs:complexType mixed="true">
    complex type details
  </xs:complexType>
</xs:element>
```

The XSD language consists of many more constructs we do not mention such as substitution groups, type extensions, integrity constraints and other.

The book element from the sample XML document in Figure 2.1 can be described using the XSD as shown in Figure 2.3.

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="authors">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="author" type="xs:string"
             maxOccurs="unbounded"/>
          </xs:sequnce>
        </xs:complexType>
      </xs:element>
      <xs:element name="isbn" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

Figure 2.3: A simple XSD example

## 2.2 XPath

XML Path Language (XPath, version 1.0) [9] is a language to select fractions of XML documents using so-called path expressions. XPath considers an XML document as a tree of nodes. It recognizes seven types of nodes: document node, element node, attribute node, text node, namespace node, processing instruction node, and comment node. The root of the tree is the document node representing the entire XML document.

### 2.2.1 Path Expressions

A path expression is composed of individual path steps separated by `/` and can be absolute or relative. The absolute path begins with `/` representing the document node. The relative path needs a non-empty starting node-set to be evaluated. A path step is
$axis$`::`$node\_test$`[`$predicate_1$`]`...`[`$predicate_n$`]`
where all predicates are optional.

### 2.2.2 Axes

An axis specifies a node-set relative to the current node. The default axis is `child` selecting all child nodes of the current node. Another important axes are `attribute`, selecting attributes of the current node, and `descendant-or-self` selecting the current node and all its descendants in the tree. The remaining axes are: `ancestor`, `ancestor-or-self`, `descendant`, `following`, `following-sibling`, `namespace`, `parent`, `preceding`, `preceding-sibling`, and `self`.

### 2.2.3 Node Tests

A node test identifies node(s) within all nodes selected by an axis. It can be a node type and/or node name. Examples of the node tests follow.

- `node()` All nodes selected by an axis.
- `text()` Text nodes.
- `*` All nodes assigned with their name (elements and attributes).

### 2.2.4 Abbreviations

Abbreviations for several most widely used constructs are defined as follows.

- *P/identifier* stands for *P*/`child::`*identifier* (child is a default axis)
- *P*/`@`*identifier* stands for *P*/`attribute::`*identifier*
- *P*/`..`/*identifier* stands for *P*/`parent::*`/*identifier*
- *P//identifier* stands for *P*/`descendant-or-self::node()`/*identifier*

### 2.2.5 Predicates

Predicates place additional conditions upon nodes that passed the node test. They can be either relative XPath paths or comparison expressions. The path predicates evaluate to true if they select a not empty set of nodes. Operators in the comparison expressions are `=`, `!=`, `<`, `>`, `<=`, `>=`, `&eq;`, `&ne;`, `&lt;`, `&gt;`, `&le;`, and `&ge;`. Operands are any XPath expressions (paths, literal values, etc).

### 2.2.6 Built-in Functions

XPath also provides a set of built-in functions. Few examples are `count(`*path*`)`, returning a number of nodes selected by *path*, `position()`, returning the position of the current node in the current node-set, and `sum(`*path*`)`, returning the sum of all nodes selected by *path*.

### 2.2.7 Usage Examples

Finally, we introduce several examples of XPath expressions, with their description, using the sample XML document in Figure 2.1.

A path that selects `author` elements of the book with id *b*1:

```
/bookstore/book[@id = "b1"]/authors/author
```

An expression returning the number of all books:

```
count(//book)
```

A path selecting the ISBN element of the book entitled *XQuery*:

```
/bookstore/book[title = "XQuery"]/isbn
```

A path returning the titles of the books written by more than one author:

```
/bookstore/book[count(authors/author) > 1]/title/text()
```

## 2.3 XQuery

An XML Query Language (XQuery, version 1.0) [11] is a language designed to query XML data. It is based on XPath 2.0. XPath 2.0 is an extension of XPath 1.0 (but not entirely compatible), adding ordered sequences, their iterations, set operations, conditional expressions, quantified expressions, and XML Schema types.

In the remainder of this section, basic features of the XQuery languages are briefly described. Many query examples can be found in Chapter 4.

### 2.3.1 Sequences

A sequence is an ordered set of items. The result of each XPath 2.0 (and also XQuery 1.0) path is a sequence. An item is either an atomic value or a node. An atomic value is a value of an XML Schema simple type. A node is an instance of one of the node types.

Each node has its identity, data type (simple or complex, according to the XML Schema types), typed value (which can be retrieved by function `fn:data()`), and string value (which can be retrieved by function `fn:string()`).

### 2.3.2 FLWOR Expressions

A basic construct of the XQuery language is $FLWOR$. It is an abbreviation of its five clauses: for, let, where, order by, return. For clause is

`for` *var* `in` *expr*

Expression *expr* is evaluated and its result is a sequence. Items of the sequence are iteratively assigned to *var* variable, which is valid also in the following clauses.

Let clause is

`let` *var* `:=` *expr*

which evaluates *expr* expression and its result is assigned to *var* variable.

Where clause is

`where` *expr*

Expression *expr* can (and usually should) contain the variables from the for clause(s) and the remaining clauses are executed for only those tuples of values of the for variables, that the *expr* evaluates to true.

Order by clause is

`order by` *expr*

and it orders the tuples of values that passed the where clause by the specified criterion.

Return clause is

`return` *expr*

The result of the whole *FLWOR* expression is *expr*. It is constructed using the tuples of the for variable values and the let variables.

### 2.3.3 Conditional Expressions

XQuery provides common `if-then-else` conditional expressions with common syntax.

```
if (condition)
then expr1
else expr2
```

If *condition* expression evaluates to true, *expr*1 is evaluated, else *expr*2 is evaluated. The else branch is optional.

### 2.3.4 Quantified Expressions

XQuery provides two types of quantified expressions, `every` and `some`. Their syntax follows.

```
every var in expr1
satisfies expr2

some var in expr1
satisfies expr2
```

Firstly, *expr*1 is evaluated, and then the result of the quantified expression is true if *expr*2 is true for every (some) item, represented by *var*, of the result of *expr*1 (which is a sequence).

### 2.3.5 Functions

XQuery provides a wide set of built-in functions.

And also new functions can be defined using the following syntax.

```
define function name(parameters) as type
{
  expr
}
```

where *name* is the function's name, *parameters* is a list of parameters (with or without specification of their types), and *type* is a type of a return value, which is the result of *expr*.

# Chapter 3

# Analysis of Recent Approaches

Existing approaches to XML schema inference can be classified using several criteria. A basic classification is based on the language the resulting schema is written in. Commonly used languages are DTD and XML Schema.

According to [23], the type of the inference method can be divided into *heuristic* and *grammar-inferring*. Heuristic methods [7, 13, 25, 35, 31] are based on experience with manual construction of schemas, motivated by real-world usages of XML schema, and their result commonly does not belong to any class of grammar. Two of these approaches are described in Chapter 3.2 and Chapter 3.3. The former one incorporates so-called MDL principle to create the (sub)optimal result. The latter one combines several verified methods together to improve the quality of the result.

On the contrary, the grammar-inferring methods [3, 4, 5, 15, 22, 12] are based on theoretical knowledge of automata and their results belong to a particular class of languages. Thus, these methods guarantee specific characteristics of the results.

Another important criterion is the type of input data. Most of the approaches process XML documents as the input of the inference process and the documents are supposed to be valid against the resulting schema. Besides approaches exploiting XML data, approaches that utilize other or additional sources may be developed. An approach utilizing XML data along with an obsolete XML schema is described in [24]. However, the most significant approaches in terms of this work are those utilizing operations over XML data. According to our best knowledge at the time of writing, there is just one approach of this category, described in [26]. It utilizes a set of XQuery queries to discover keys and foreign keys.

## 3.1 Common Caracteristics

The process of XML schema inference commonly used by a significant number of approaches is summarized in [23] as the following one: For each occurrence of element $e$ from the input XML documents and its subelements $e_1, e_2, ..., e_k$ a production $e \rightarrow e_1 e_2 ... e_k$ is constructed. The productions form so-called *initial grammar* (IG). For each element type the productions are then merged, simplified and generalized using various methods and criteria. A common approach is so-called *merging state algorithm*, where a *prefix tree automaton* (PTA) is built from the productions of the same element type and the automaton is generalized via merging of its states. Finally, the generalized automaton/grammar is expressed in syntax of the respective XML schema language.

## 3.2 XTRACT

The XTRACT [13] system is an example of a *heuristic merging state algorithm* creating the result in DTD. Its process of inference consists of three steps:

1. Generalization - Generates a set of DTD candidates by searching the input for certain patterns and generalising corresponding fragments using regular expressions.

2. Factoring - Groups of generalized candidate DTDs are factorized to a new ones by finding common sub-expressions to make them more concise.

3. Minimum Description Length (MDL) Principle - Composing a near-optimal DTD schema from the set of all generalized candidate DTDs.

### 3.2.1 Generalization

The purpose of generalization is to create a set of DTD candidates - schemata that cover fractions of the input XML data. In the last step, this set will be used to compose a (sub)optimal result with respect to a trade-off between its preciseness and conciseness. Therefore, it is desirable to create DTD candidates with various degrees of these two characteristics.

Generalization is based on replacing fragments (sequences of subelements of a given element) from the input XML data by regular expressions, thus, using metacharacters like `*`, `+`, `?`. To provide a wide set of DTD candidates, each sequence is processed several times using various values of input parameters.

Due to the very large number of possible DTD candidates, the authors employ certain real-life motivated heuristics.

For instance, paper [13] introduces the following example: Sequences `abab` and `bbbe` are generalized to `(ab)*`, `(a|b)*`, `b*e`.

### 3.2.2 Factoring

Factoring is a process of creating a new DTD candidate from two or more DTD candidates, decreasing their summed size without modifications in their semantics. The aim of this step is to decrease the MDL cost of DTD candidates calculated in the MDL step and thus refine the process of construction of the resulting DTD.

An example is introduced it paper [13]: DTD candidates `ac`, `ad`, `bc` and `bd` are factored into `(a|b)(c|d)`.

Alike the generalization step, also in this step the set of possible factored DTDs is huge and the authors propose certain heuristics to make the factorization effective.

### 3.2.3 Minimum Description Length (MDL) Principle

This is an important step trying to create the resulting DTD with the best trade-off between its preciseness and conciseness.

Paper [23] summarizes this step as follows: It expresses the quality of a DTD candidate using two aspects – conciseness and preciseness. Conciseness of a DTD is expressed using the number of bits required to describe the DTD (the smaller, the better). Preciseness of a DTD is expressed using the number of bits required for description of the input data using the DTD. In other words, the more accurately the structure is described, the fewer bits are required. Since the two conditions are contradictory, their balancing brings reasonable and realistic results.

## 3.3 Even an Ant Can Create an XSD

This work, described in [31], combines several previously proposed approaches including the XTRACT system discussed in the previous section. Its improvements of the process of XML schema inference include:

- Distinguishing elements with the same name but different context.

- Improvements of algorithms adopted from the previous works.
- Incorporating inference of an unordered sequence.
- Creating a result in the XML Schema language.

### 3.3.1 Clustering of Elements

This phase clusters elements on the basis of their context and structure. It is done by creating tree structures for each input XML document where vertices represent elements and attributes and an edge between two vertices expresses their parent-child relationship. These trees and their subtrees are then compared using an imposed tree similarity measure to find elements with the same semantics.

### 3.3.2 Schema Generalization

For each cluster, a trivial schema is created, which is then generalized to achieve a reasonable result. In search for the optimal schema, *Ant Colony Optimization (ACO)* heuristic is incorporated. The idea behind the ACO heuristic is that a set of artificial ants is searching a space of possible solutions, each ant given a subspace of the space to find a local suboptimum. An ant is performing steps (schema modifications), dying after a predefined number of steps and providing an information - positive feedback - on the quality of a solution found. The search is performed in a defined number of iterations (or it stops if a good enough solution is found) and the positive feedback from one iteration is used to find better results in the following iterations. Every step of an ant represents a modification of a schema, in particular, a merge of states in a corresponding PTA.

One of the improvements of this heuristic is an inclusion of a negative feedback after each step of an ant, visible only in the current iteration. Due to this improvement, a larger subspace of solutions is searched.

Another improvement lies in a way how an ant decides for a particular step to perform. To achieve better results, the authors propose a combination of several verified approaches. A set of all possible steps is created using *k,h-context* [3] and *s,k-string* [28, 35] methods. The optimal step is then selected employing the MDL principle [14, 13].

### 3.3.3 Result in XSD

Unlike the majority of recent approaches, this methods creates its result in XML Schema. The authors of this method focused on inferring elements with the same name but different context and the unordered sequences which can be in XSD expressed by `xs:all` construct. Elements with the same name but different context cannot be expressed in DTD and, although, the unordered sequence can be also expressed in DTD as alternations of ordered sequences, such expression in not practical nor well human-readable.

## 3.4 On Inference of XML Schema with the Knowledge of an Obsolete One

The aim of approach described in [24] is to exploit an obsolete XML schema as an additional input information to infer a new schema more efficiently. An XML schema can become obsolete due to changes in a set of XML documents, without capturing these changes in the schema. Thus, the schema becomes outdated and according to the paper, this case is quite common.

On input, the method is given:

- An original XML schema.
- A set of XML documents. Not all have to be valid against the original schema.

The algorithm consists of two independent steps:

1. Correction of the input schema.
2. Specialization of the input schema.

In the first step the input schema is corrected to conform to the whole set of input XML documents. This is done by creating a PTA for each production extracted from the input schema and merging it with a respective production from the initial grammar, involving ACO and MDL heuristics.

In the second step, regular expressions from the corrected schema are specialized with regard to the XML documents, resulting in a more precise and readable schema. Optional substeps are pruning of unused schema fragments, correction of lower and upper bounds of occurrences, transcription of operators to a more restrictive but simpler form, if this transcription preserves the validity, and refactorization to improve readability.

```
for $e_1$ in $P_1$
return
  for $e_2$ in
    $P_2[L_2 = \$e_1/L_1]$
  return $C_R$
```

Listing 3.1: For join pattern.

## 3.5 Discovering XML Keys and Foreign Keys in Queries

The method described in paper [26] improves automatic XML schema inference by discovering keys and foreign keys from a set of XQuery queries. Just the queries are utilized in inference, no XML data are used. The output of this method is a set of keys and foreign keys that can be captured using XML Schema **key**, **keyref** and **unique** constructs.

### 3.5.1 Assumptions and Observations

To discover keys and foreign keys, the method utilizes element/element joins. Assume a query $Q$ that joins a sequence of elements $S_1$ targeted by a path $P_1$ with a sequence of elements $S_2$ targeted by a path $P_2$ on a condition $L_1 = L_2$. For instance, see Listing 3.1 and Listing 3.2.

The method is based on an assumption that each join is done via key/foreign key pair. It means it is supposed that $L_1$ is a key of the elements in $S_1$ and $L_2$ is its respective foreign key or vice versa.

The authors describe two possible cases:

(O1) $L_1$ is a key of the elements in $S_1$, $L_2$ is a respective foreign key and it itself is not a key of the elements in $S_2$.

(O2) $L_2$ is a key of the elements in $S_2$, $L_1$ is a respective foreign key and it cannot be decided whether $L_1$ is a key of the elements in $S_1$ or not.

### 3.5.2 Join Patterns and Key Inference

For a certain join, the decision for one of the cases (O1) and (O2) is made by the form of the join. The query is searched for so-called *join patters.* These are *for join pattern* and *let join pattern* and they are proclaimed in Listing 3.1 and Listing 3.2.

```
for $\$e_1$ in $P_1$
return
  let $\$e_2$ :=
    $P_2[L_2 = \$e_1/L_1]$
  return $C_R$
```

Listing 3.2: Let join pattern.

Each occurrence of a join pattern is classified by application of the following rules R1 - R5 in this specific order. The first satisfied rule is applied. The occurrence is also assigned with a weight determining how sure the method is about the inferred statement.

The pattern occurrence is considered of case (O1) if it is the for join pattern (R1, weight: 1), if aggregation function `avg`, `min`, `max` or `sum` is applied on a target return path (R2, weight: 1) or if aggregation function `count` is applied on a target return path (R3, weight: 0.75), where target return paths are paths in $C_R$ starting with $\$e_2$ (see Listing 3.1).

Otherwise, the pattern occurrence is considered of case (O2) and the assigned weight depends on the number of target return paths. If the number is greater than one, the weight is one (R4, weight: 1), else (the number equals zero or one) the weight is one half (R5, weight: 0.5).

### 3.5.3 Summarization of the Results

The assumption the method is based on may not be fulfilled for every join in a supplied set of queries. A key $K$ may be inferred from some query and processing of another query may result to denial of $K$ as a key.

Therefore, the authors introduce a scoring function to summarize the positive and the negative statements about keys using the assigned weights. The value of the score expresses the probability that a respective key statement is satisfied. Finally, the scores of the inferred keys are normalized to be comparable with each other.

### 3.5.4 Conclusion

The output of the method is a list of scored keys and for each key a list foreign keys referencing the key. Since the method deals only with the inference of keys, it is not a complete method of the XML schema inference. It is meant to be used in collaboration with other schema-inferring methods to refine their results.

| Method name and/or paper | Input | Output | Year |
|---|---|---|---|
| [3] | XML documents | DTD | 1996 |
| XTRACT [13] | XML documents | DTD | 2000 |
| DTD-miner [25] | XML documents | DTD | 2000 |
| [12] | XML documents | DTD | 2001 |
| ECFG [7] | XML documents | XSD | 2002 |
| sk-ANT [35] | XML documents | DTD | 2003 |
| [22] | XML documents | DTD | 2003 |
| [4] | XML documents | DTD | 2006 |
| XStruct [15] | XML documents | XSD | 2006 |
| [5] | XML documents | XSD | 2007 |
| SchemaMiner [31] | XML documents | XSD | 2008 |
| [24] | XML documents, XML schema | DTD, XSD | 2009 |
| [26] | XQuery queries | –[1] | 2009 |
| [17] | XML documents, XML schema | DTD, XSD | 2011 |

[1] The result of this method is a list of discovered keys and foreign keys.

Table 3.1: Summary of Recent Approaches

Since the method is based on intuition of how XQuery constructs are commonly applied in practice, it can be imprecise in certain cases.

## 3.6 Summary

As shown in Table 3.1, most of the recent approaches of XML schema inference are based on utilization of XML documents. These incorporate various verified methods and the newer approaches often improve the older ones and/or combine them together to achieve better results.

Lately, several approaches that utilize other input sources have been proposed. Paper [24] introduces a method that utilizes an XML schema besides XML documents. The method described in [26] utilizes XQuery queries; however, the result of this method is a list of discovered keys and foreign keys, not an XML schema. Also, to our best knowledge, implementation that combines this method with another approach to get the XML schema has not been proposed yet.

There are plenty of additional sources that can be exploited in the process of inference such as other XML schema languages, queries, XSLT scripts, negative examples (XML documents that should not conform to the resulting schema). In this work we focus on utilizing XQuery queries.

# Chapter 4

# Analysis of XQuery

This chapter discusses selected constructs of XQuery language and denotes how they could be exploited in the XML schema inference process. It is divided into sections by particular domains of the inference.

Most of sample queries in this chapter are taken from [33] and [21], with or without modifications, as these were the main sources for this XQuery analysis.

## 4.1 Structure of XML documents

Most of queries can be exploited to obtain some information about the structure of respective XML documents. The structure of XML documents captures elements from these documents along with their names, attributes and their organization. What the root elements in these documents are, which elements can be contained within a certain element, whether they are optional or mandatory, etc.

Path expressions without predicates which use only child axis are the simplest example of such queries. For example, path expression `/bib/book/author` indicates that element `bib` is the root element and it contains one or more elements `book` and these contain one or more elements `author`.

Additional path expressions `/bib/book/title`, `//author/name` indicate that element `book` also contains element `title` and element `author` contains element `name`. The latter one uses also descendant axis. That means the query considers all elements `author` in the document, thus it is a hint that maybe there are several elements named `author` but with different absolute paths and some of them contain an element `name`.

Besides elements, attributes can be processed exactly in the same way. Path expression `/bib/book/@ISBN` indicates that element `book` can have attribute `ISBN`.

Statements of this kind are necessary for the XML schema inference, however their obtaining from queries is not significant due to the following reason. These statements could be determined directly from the XML documents and it could be easily done in a more convenient way. In addition, queries may not cover the whole relevant content of the documents. Let $E$ be an element occurring in an XML document and let $Q$ be an XQuery query. Consider the following cases:

1) $E$ is directly mentioned in $Q$. For example elements `bib`, `book` and `author` are mentioned in path expression `/bib/book/author`.

2) $E$ is not mentioned in $Q$ but it does occur in a result of the evaluation of $Q$. For example path expression `/bib/book/author` returns elements `author` along with their subelements `name`, `birthdate` and `nationality`.

3) $E$ is neither mentioned in $Q$, nor it occurs in the result, but it is processed by the evaluation of $Q$. For example the evaluation of path expression `//author` processes amongst others elements `bib` and `book`, in a search for elements `author`.

4) $E$ occurs in a part of the XML document not related to $Q$ at all.

How we can obtain any information about element $E$ from query $Q$? In case 1), $Q$ can be directly exploited in the inference process. Case 2) requires the result of $Q$ and additional processing of the result. Case 3) requires a step-by-step evaluation of $Q$, hence, the XML document is also required. And in case 4), it is not possible to obtain any information.

In addition, these statements do not express any obligation of occurrence of elements and attributes nor clearly determine multiple occurrence of elements. We also cannot be sure that queries target nodes actually presented in the XML documents. Although query `/bib/book/author` indicates that element `author` is contained in element `book`, the query is valid whether this is true or not. In contrast, even basic methods of XML schema inference that utilize XML documents do not have these inadequacies.

On the other hand, the structural inference could be useful when the entire set of all XML documents is not available and the provided XML documents do not cover the structure completely.

## 4.2 Number of Occurrences of Elements

Some XQuery constructs indicate multiplicity of a particular element or limit the element to occur at most once. Consider the following query assuming variable `$book1` is bound to a certain `book` element.

```
for $a in $book1/author
order by $a/last, $a/first
return $a
```

Apparently, this query expects more than one `author` element to be a child of the element the variable `$book1` is bound to. Otherwise, any sorting would lack a reason. Although we cannot be absolutely sure about it again, assuming common-sense usage of XQuery, it is very likely that `author` element can occur multiple times as a subelement of element from variable `$book1`.

### 4.2.1 Multiple Occurrence

A similar approach could be applied in many other situations. Another examples are particular usages of function `count()`, indexation, usage of set operators (`union`, `intersect`, `except`) and usage of function `one-or-more()`. Sample queries with a respective description follow.

```
<section_count>{ count(/book/section) }</section_count>
```

Function `count()` returns the number of items in a provided sequence. If the sequence is a sequence of elements, the number of these elements will probably not be limited to one. The sample query indicates that the root element `book` can contain more than one element `section`. An exception is usage of function `count()` in a predicate in expressions where it is used to determine if the number of some nodes is greater than zero. Often, this form is used to test presence of a certain node instead of actual counting of its occurrences. In this case, the node could be still limited to occur at most once.

```
($s/incision)[2]/instrument
($s/instrument)[position()>=2]
```

Indexation of nodes and common usage of function `position()` suggest that the author of such query assumes a sequence of respective nodes.

```
one-or-more(/catalog/product[@id = 5]/color)
```

In this query, the number of elements `color` in element `product` with attribute `id` equal to 5 has to be at least one, otherwise an error is raised upon execution. If we assume that this query is written correctly, with common sense and it should not raise the error, we can infer that element `product` has to contain one or more elements `color`.

### 4.2.2 Occurrence Limited to One

Contrary to the multiple occurrence, numerous XQuery constructs limit number of occurrences of an element to at most one or exactly one. Sample queries with description follow.

```
/catalog/product[1]/number lt 10
```

`lt` is a representant of so called *value comparison operators* (`eq`, `ne`, `lt`, `le`, `gt`, `ge`, see [11]) which operate two sequences of zero or one item. If an operand of a value comparison operator is a sequence of more than one item, then a *type error* (see [11]) is raised.

```
for $item in //item
order by $item/num
return $item
```

Alike the previous example, an expression in `order by` clause can be evaluated to at most one item or the *type error* is raised. Therefore, every element `item` contains zero or one element `num` but not more.

Other similar examples are arithmetic expressions and functions accepting a sequence of at most one item. Function `zero-or-one()` will raise the *type error* when supplied with a sequence of more than one item.

Those are constructs indicating limitation to zero or one occurrence. Function `exactly-one()` works similarly to the function `zero-or-one()` but accepts only sequences of exactly one item (which are in XQuery equal to this item itself).

## 4.3 Element and Attribute Types

### 4.3.1 XML Schema Built-in Types

XML Schema built-in types of elements and attributes can be inferred from the XML documents by analysing their content. Since the number of built-in types is 44 and inheritance is involved, such analysis may be imprecise, especially when a large enough set of XML documents is not available. This is the case when a set of XQuery queries may bring optimizations. For example, consider the following occurrences of element `a`.

```
<a>1</a>
<a>6</a>
<a>18</a>
```

These three occurrences are not sufficient enough to determine the accurate type of element `a`, because values `1`, `6`, `18` are valid values of several types: `decimal`, `integer`, `byte`, `short`, `unsignedInt`, `positiveInteger` and many others.

If a value of an element or attribute is used in an expression, then this expression could be often exploited to determine a type of the value. Comparing the value to another value of a known type and supplying the value to a function call as an argument of a particular type are examples of these expressions as shown below.

```
//event/date = current-date()

/catalog/product/price < 24.5

declare function local:byteFunction($arg as xs:byte) as xs:byte
{...};
/catalog/product/local:byteFunction(@id)
```

The determined types are `xs:date` for element `date`, `xs:decimal` for element `price` and `xs:byte` for attribute `id`.

Alongside common expressions, there are other XQuery constructs indicating types, such as *type casting*, *type constructors* and so called *type declarations* (see [11] for specification), demonstrated in the following examples.

Assuming variable `$var` bound to a value of some element or attribute, the following two fractions of queries indicate its type to be *xs:integer*.

```
$var cast as xs:integer
xs:integer($var)
```

The latter one is usage of *type casting*, the former one is *type constructor*.

Similarly, the following two queries are examples of *type declarations*. Value of element `number` is declared to be of type `xs:integer`.

```
every $number as element(*,xs:integer) in //number
  satisfies ($number > 0)

declare variable $firstNumber as xs:integer
:= data(//product/number[1]);
```

### 4.3.2 Enumeration

In several cases, XQuery *if-then-else* construct may be used to branch the execution of query by all possible values of a certain variable. This is equivalent to *switch* construct from other programming languages. When the control variable is bound to some element or attribute, type of this node can be inferred as enumeration and its individual values can be determined as well.

```
Query
let $cat := doc("catalog.xml")/catalog
for $dept in distinct-values($cat/product/@dept)
return <li>Department: {if ($dept = "ACC")
                        then "Accessories"
                        else if ($dept = "MEN")
                             then "Menswear"
                             else if ($dept = "WMN")
                                  then "Womens"
                                  else ()
  } ({$dept})</li>

Results
<li>Department: Womens (WMN)</li>
<li>Department: Accessories (ACC)</li>
<li>Department: Menswear (MEN)</li>
```

Finding such patterns in queries could be useful in combination with analysis of respective XML data. XML data could help to confirm or disprove this assumption based on the query analysis or vice-versa.

## 4.4 Keys

### 4.4.1 Approach from [26]

Paper [26] introduces a method of discovering keys and foreign keys by investigation of joins in queries. The basis of this discovery is a search for particular forms of joins, so called *join patterns*, but the joins are processed only if they are found in a particular syntactic form. Therefore joins with the same semantics written in different syntax are not taken into consideration.

The following examples are queries that could be processed by a similar method, but the actual method will not use them.

```
<result>
  {
    for $u in doc("users.xml")//user_tuple
    for $i in doc("items.xml")//item_tuple
    where $u/rating > "C"
       and $i/reserve_price > 1000
       and $i/offered_by = $u/userid
    return
        <warning>
            { $u/name }
            { $u/rating }
            { $i/description }
            { $i/reserve_price }
        </warning>
  }
</result>
```

Clause `where` is used for the join condition `$i/offered_by = $u/userid` `instead` of the join condition in predicate of the second `for` expression.

```
<result>
  {
    for $i in doc("items.xml")//item_tuple
    where empty(doc("bids.xml")//bid_tuple[itemno = $i/itemno])
    return
        <no_bid_item>
            { $i/itemno }
            { $i/description }
        </no_bid_item>
  }
</result>
```

This query illustrates a join where the value of one of the joined elements is not required, thus the knowledge of its existence is sufficient. Therefore the query does not have to contain the second `for` or `let` keyword and its expression can be moved to `where` clause.

```
for $item in doc("order.xml")//item,
    $product in doc("catalog.xml")//product,
    $price in doc("prices.xml")//prices/priceList/prod
where $item/@num = $product/number and $product/number = $price/@num
return <item num="{$item/@num}"
         name="{$product/name}"
         price="{$price/price}"/>
```

A common three-way join can be also utilized to infer keys and foreign keys, however, the respective XML data and their analysis would be needed to tell what is a key and what is a foreign key.

### 4.4.2 Join of Self-referencing Data

By the term "self-referencing data" are meant XML elements that somehow reference items from the same set. Example of a query that operates upon such data follows.

```
declare function local:one_level($p as element()) as element()
{
  <part partid="{ $p/@partid }" name="{ $p/@name }" >
    {
      for $s in doc("partlist.xml")//part
      where $s/@partof = $p/@partid
      return local:one_level($s)
    }
  </part>
};
```

```
<parttree>
  {
    for $p in doc("partlist.xml")//part[empty(@partof)]
    return local:one_level($p)
  }
</parttree>
```

Apparently, element `part` can contain attributes `partid` and `partof`. Elements with unspecified attribute `partof` are at the top of the recursive hierarchy, while each element with this attribute specified references an element with attribute `partid` of the same value.

By using a similar approach as described in [26] attribute `partid` can be marked as a key and attribute `partof` as its foreign key.

### 4.4.3 Negative Statements about Uniqueness

In many cases a statement refusing uniqueness of element or attribute values can be inferred. Such statements may be helpful in combination with other methods in its process of making a decision whether a particular element or attribute is unique or not.

Basic representant is a common FLWOR query.

```
<bib>
  {
    for $b in doc("http://bstore.example.com/bib.xml")//book
    where $b/publisher = "abcde" and $b/@year > 2000
    order by $b/title
    return
        <book>
            { $b/@year }
            { $b/title }
        </book>
  }
</bib>
```

Usage of `for` construct indicates that a sequence of `book` elements which satisfy the condition in `where` clause is expected. It is a condition composed of two single conditions joined by `and` logical operator. Therefore, in order to satisfy the whole condition, both single conditions must be satisfied as well. Thus, it is expected that several elements satisfy each of the single conditions.

The first of them is a test of equality of book's subelement `publisher` to a string literal. Based on the expectation, element `publisher` cannot be unique. However, the second condition is greater-than comparison of `year` attribute to an integer literal, it cannot be inferred whether this attribute is unique or not. The reason is that even if it was unique, there still might be more than one `book` element meeting this condition. Also, any statement, positive nor negative, cannot be inferred about `title` subelement.

Other simple examples are passing a result of basic path expression to a call of `distinct-values()` function and usage of *aggregation* functions.

```
<results>
  {
    let $doc := doc("prices.xml")
    for $t in distinct-values($doc//book/title)
    let $p := $doc//book[title = $t]/price
    return
      <minprice title="{ $t }">
        <price>{ min($p) }</price>
      </minprice>
  }
</results>
```

According to the use of `distinct-values()` function, it can be easily seen that the author of this query assumes possible occurrence of more than one `book` element with the same value of their `title` subelement. Thus, `title` element is not unique.

Alike, variable `$p` is bound to a price of each title and then passed to `min()` function call. That indicates that one book title is supposed to have several prices, however, it cannot be said if a certain occurrence of element `book` can contain more than one `price` subelement.

Also, many other types of queries could be exploited to obtain negative uniqueness statements like occurrence of element or attribute in `stable order by` clause or selection of an element set based on a value of particular attribute and consecutive treatment of this set as a sequence of elements.

```
let $prods := doc("catalog.xml")//product
for $prod in $prods
```

```
where $prod << $prods[@dept = $prod/@dept][last()]
return $prod
```

Usage of predicate `[last()]` indicates that `$prod` is a sequence of elements.

### 4.4.4 Uniqueness

Contrary to non-uniqueness discovery, some XQuery constructs can indicate uniqueness of elements or attributes; however, it seems to be more difficult. One of the approaches of uniqueness discovery could be an investigation of what the query does return. Consider the following query.

```
for $product in /catalog/product
let $number := $product/number
return <prod xml:id="{concat('prod', number)}"/>
```

For each `product` element, new element `prod` with attribute `xml:id` is created. Since attribute `xml:id` is supposed to be unique and there is a direct transformation of the values of `number` elements to the values of attribute `xml:id`, it is very likely that element `number` is unique in the source data.

## 4.5 Other Constructs

### 4.5.1 XML Schema `xs:sequence` and `xs:all` constructs

If an order of appearance of some element set, for example subelements of a certain element, is important, this can be expressed by XML Schema construct `xs:sequence`. On the other hand, `xs:all` is involved when the elements may occur in any order.

If there is a large enough set of XML documents available, it should be possible to correctly detect where to use `xs:sequence` and `xs:all` constructs (paper [31] deals with this problem). However, if the inference is made using a smaller set of XML documents, it may happen that every occurrence of some element set is in the same order but the input data are not representative enough to be sure. Comparison of element order in queries can help to decide for the use of `xs:sequence`.

Every two elements in an XML document that are not siblings with the same name and none of them is the root element have the nearest common ancestor determining their relative order. If there is a lack of evidence to choose between the two constructs in the ancestor, then available queries can be searched for an order comparison of these two elements like in the following query.

```
let $i := //incision[2]
for $a in //action[. >> $i]
return $a//instrument
```

Apparently, the relative order of elements `incision` and `action` is important, because the query composes the result using only those `action` elements that succeed the second `incision` element. Therefore, in a respective type definition part of their nearest common ancestor `xs:sequence` should be used in favour of `xs:all`.

### 4.5.2 Intermediate XML structure

Intermediate XML structure represents XML data that are neither read from input XML documents nor created as an output of queries but they are somehow used by the queries. Objective of this section is intermediate XML structure created directly in the queries. It may serve to various purposes and it can be created in various ways, hard-coded in queries, computed from the input XML documents to simplify their structure. The former is demonstrated in the following example.

```
let $deptNames := <deptNames>
                    <dept code="ACC" name="Accessories"/>
                    <dept code="MEN" name="Menswear"/>
                    <dept code="WMN" name="Womens"/>
                  </deptNames>
let $catalog := doc("catalog.xml")/catalog
for $dept in distinct-values($catalog/product/@dept)
return <li>Department:
         {data($deptNames/dept[@code = $dept]/@name)} ({$dept})
       </li>
```

Intermediate XML structure in this query can be used to identify possible values of attribute `dept` in `product` element and therefore determine its type as enumeration.

This is a very simple example of intermediate structure. Since there are admittedly many means of utilization of intermediate structure, further research would be needed.

# Chapter 5

# Pre-creation of Algorithm

According to the analysis in the previous chapter, there is quite a wide range of possible utilizations of XQuery queries. Besides analysis of what information can be extracted from queries, it is needed to devise how the queries will be processed. This chapter discusses some questions and issues that emerged in an early phase of the algorithm fabrication.

## 5.1 Input Data

The first important question is what is the input of the algorithm. A basic query utilization can be achieved by analysis of queries without any other input data. The analysis of XQuery in the previous chapter discusses mostly XQuery constructs which can be utilized without respective XML data, for example the inference of built-in types. This independence is also the main advantage of this approach, if there are no XML data available, this approach can be still used.

A more complex method can utilize queries along with the respective XML data. As discussed in the previous chapter, an element and attribute structure can be inferred from the XML data in a more convenient way. Also, the XML data can be used to verify information inferred from the queries or vice-versa. For example, utilizing the queries, some attribute is considered a key of its element. But in the data there are elements with the same value of this attribute, and thus, it cannot be the key. Vice-versa, we have a notion that the attribute might be the key but we are not sure about that. Analysing of the data and finding that values of the attribute are unique can increase our confidence.

Another step can be evaluation of the queries using the XML data and consecutive analysis of the results. And even the process of evaluation itself can be analysed to obtain some useful information. For instance, these are partial results of evaluating of expressions (elements selected by each path expression, real arguments in function calls, etc).

## 5.2 Forms of Query Precessing

Another important question is how the queries can be processed. Will they be just searched for certain patterns like it is performed in method [26] or will they be processed in a more sophisticated way? That can mean incorporating lexical and syntax analyses, known from creation of compilers, or even a form of an analysis of semantics [2].

The result of lexical and syntax analyses can be a kind of so-called syntax tree [2]. It is a structure representing a word according to a formal grammar of a language. In our case, the language is XQuery, its grammar is defined in [11] and every query is a word of the XQuery language. Leaves of the tree represent terminals of the grammar while inner nodes represent non-terminals. From the point of view of this work, the syntax tree can be perceived as a preprocessed form of a query keeping its complete meaning and making its further processing more convenient. For instance, the tree can simplify a search for FLWOR statements. It is transitioned and nodes representing FLWORs are found. Then each subtree determined by one of the found nodes represents a FLWOR statement and it can be analysed further.

The syntax tree can be also extended by additional information. An example is a static analysis of expression types. Types of literal expressions are defined, functions have return types, path expressions can return nodes, etc. Types of complex expressions can be determined applying the rules defined in [11]. The inferred expression types can be helpful for example in the analysis of built-in types of nodes as discussed in the previous chapter.

The following text is an example of inference of a more complex query processing. Consider the following part of a query.

```
declare function local:getB($id as xs:string) as element() {
  //A[@id = $id]/B
}
... local:getBs("id") > 10 ...
```

The query consists of a function declaration and an arithmetic comparison. The comparison compares the result of the function call and literal value `10`. Since the type of `10` is `xs:integer`, the type of the function call has to be convertible to `xs:double`. Thus, it has to be a numeric type. The function returns a path expression typed as `element()`. That means the function returns one element. In the path expression, the argument `$id` can be substituted by the real value specified in the call. Thus, the return expression is `//A[@id = "id"]/B`. Therefore, we can infer that element `B` in element `A` with attribute `id` equalling string value `"id"` is of some numeric type. And, since the function is parametrized, there is a notion that this statement may be correct also for other elements `A` and `B`.

While simpler approaches of the query processing such as the pattern finding limit possibilities of the query utilization, a more complex processing of queries provides a better starting point for consecutive analyses and also for further refinements and additions. Therefore, we decided to incorporate query processing using the lexical and syntax analyses.

## 5.3 Inference of XML Structure

The question of inference of XML structure from queries is partially discussed in the previous chapter. We are able to infer XML structure from queries without their evaluation, but in a limited way. This inference is based on an analysis of path expressions. Its limitation involves the following issues. When we infer some subelements of a certain element we often cannot be sure about their number of occurrences. Also, we cannot be sure if every occurrence of a certain element contains the subelements and we even do not know if at least one occurrence of the element contains them. Thus, the inferred statement is more likely an indication than a fact about the structure.

Since the inference of XML structure utilizing only queries is not clear, we need the XML data, if we want to infer the structure more precisely. And, if we have the XML data, we can infer the structure directly from them using an existing approach and utilize queries to refine its result.

## 5.4 Extension of an Existing Approach

The existing approaches of XML schema inference deal mainly with inference of XML structure. Hence, the extension of an existing approach will be a kind of an independent addition instead of modification and refinement of its core algorithm.

The existing approaches take the XML data on their input. Therefore, the basic idea is that the input will be extended also for XQuery queries and the algorithm will consist of three phases. The first phase will be taken from an existing approach and it will process the XML data to infer the XML structure. The second phase will process the XQuery queries and it will infer statements that can be inferred independently of the XML data. The third phase will merge the statements inferred in the second phase into the resulting schema. This phase may also infer additional statements from both the XML data and the queries or it may try to verify the statements from the second phase with respect to the XML data.

A more advanced method can exploit queries to refine a core algorithm (e.g. merging of PTA) of an existing approach. The approach described in [31] distinguishes elements with the same name, but a different content model and context. Some information from queries may help to improve this algorithm. For example, consider an XML representation of company data containing names of employees, costumers, and products represented by element `name`. The names of employees and costumers consists of two subelements for first name and surname. The names of products are atomic strings. During analysis of available queries, we may find that elements representing the names of employees and elements representing the names of costumers are processed in the same way (e.g. they are mixed in one sequence), while elements representing the names of products are processed separately. This suggests that the elements representing the names of employees and costumers have the same semantic and the same content model which is different from the content model of the elements representing the names of products.

# Chapter 6

# Proposed Algorithm

In this thesis we are developing an approach dealing with inference of XML schema, whose input consists of two components; a set of XML documents and a set of XQuery queries related to the documents. The proposed algorithm introduced in this chapter describes the processing of XQuery queries only and a combination of its output with an existing method of XML schema inference (to produce a complete XML schema) is discussed in the next chapter.

There is an important assumption placed upon the input XQuery queries saying that each query must be syntactically and semantically correct. That means that if a query was evaluated (using a XQuery processor meeting the XQuery specification), the evaluation would not raise any error, neither static nor dynamic. In other words, each query is syntactically correct and it correctly queries the data in the XML documents.

## 6.1 Motivation

There is a large space of possible XQuery utilizations, but obviously, they all cannot be covered by one thesis. Hence, we had to choose only some of them. We decided to focus on inference of XSD built-in atomic types of elements and attributes, and key discovery.

The inference of types is included in several recent works ([7, 15]), but generally, it is considered a side problem and it is omitted by most of the works. Since a presence of XSD simple types in an inferred schema may be often desired or demanded, and an information on types of elements and attributes can be conveniently extracted from several XQuery constructs, we decided to incorporate the inference of simple types in our proposed algorithm and to implement it.

As a second part of the algorithm, we decided to extend the only method [26]

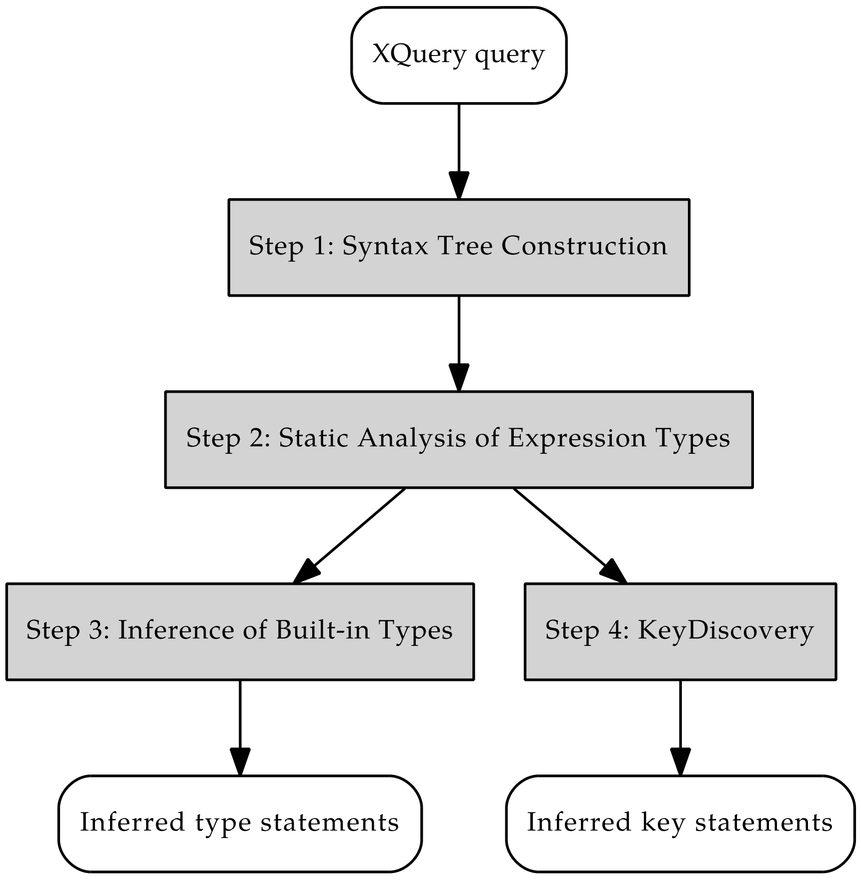

XQuery query
Step 1: Syntax Tree Construction
Step 2: Static Analysis of Expression Types
Step 3: Inference of Built-in Types
Step 4: KeyDiscovery
Inferred type statements
Inferred key statements

Figure 6.1: Steps of the proposed algorithm

which utilizes XQuery queries and infers keys and foreign keys. The extension is based on a more general and precise processing of queries, addition of a new case of query patterns that are used in the process, and a more accurate summary of key statements by incorporating searching for statements that reject uniqueness of certain elements and attributes.

Moreover, the original method has not been implemented and experimentally proven yet. We will do that as well.

## 6.2 Overview

The proposed algorithm consists of the following four main steps, show in Figure 6.1 and described in detail in the rest of the chapter.

1. **Construction of a syntax tree**. We use lexical and syntax analyses proposed in [29] and for each XQuery on input, we construct a data structure defined in Definition 6.1.

2. **Static Analysis of expression types**. The algorithm searches for ex-

pressions in the syntax trees and statically (without evaluation) determines their types. See Section 6.5.1.

3. **Inference of built-in types**. When the types of expressions are determined, selected forms of expression are utilized to infer types of elements and attributes.

4. **Key discovery**. The final step is an extension of approach [26] inferring keys and foreign keys.

As can be seen in Figure 6.1, step 4 is independent on step 3. They both depend on step 2.

## 6.3 Step 1: Construction of a Syntax Tree

The first step of the algorithm involves lexical and syntax analyses known from the construction of compilers and produces a so-called syntax tree. The analyses are taken from Jiří Schejbal's master thesis [29]. Since they are not directly related to the inference, and thus, they are not directly related to the topic of this thesis, we will not describe them. Nevertheless, they provide us with a helpful processing of XQuery queries and we can focus on the inference.

The syntactic analysis needs to be slightly modified to suit our case. It writes its result into a file in an XML representation. Instead, we need to keep the result in the main memory and pass it to consecutive steps of our algorithm. Though this requirement concerns modifications of implementation, the core of the processing remains untouched. Therefore, we also do not describe these modifications.

## 6.4 Definition of the Syntax Tree

Firstly, we formally define the syntax tree.

**Definition 6.1** (Syntax tree). Syntax tree of XQuery query $Q$ is tuple $T = (V, E, c, \mathcal{P}, o)$ where

- $V \subset \mathbb{N}$ is a set of nodes, each node representing a particular XQuery construct in query $Q$,

- $E$ is a set of pairs $(v, w)$ where $v, w \in V$ and for every $a, b \in V, a \neq b$ : $(a, b) \in E$ if and only if a construct represented by $b$ is a direct component of a construct represented by $a$ ($b$ is a child of $a$) in query $Q$,

- $c : V \to C$ is function assigning each node with its class from set $C$ of all XQuery language constructs listed in Tables 6.1, 6.2,
- $\mathcal{P}$ is a set of functions specifying additional properties of the nodes and distinguishing the nodes of the same classes,
- and $o : V \to \mathcal{O}$ is a function specifying an order of children of the nodes, where $\mathcal{O} = \{o_v : E_v \to \mathbb{N} | v \in V, E_v = \{(v, w) | w \in V, (v, w) \in E\}\}$ is set of functions specifying the children order for each node. For every $v \in V, o(v) = o_v$ so that $o_v(v, w)$ is a sequential number of a construct represented by $w$ amongst constructs represented by children of $v$ in query $Q$.

Regarding the additional properties, two constructs in $Q$ represented by two nodes of the same class from $C$ may differ in certain ways, and, therefore, it is needed to distinguish them. For instance, two different literal values in $Q$ are represented by nodes $l_1, l_2 \in V$ and $c(l_1) = c(l_2) =$ `LiteralNode` but each has a different value and type. Therefore $\mathcal{P}$ contains functions
$type_{LiteralNode} : V_{LiteralNode} \to Types_{literal}$
$value_{LiteralNode} : V_{LiteralNode} \to Values_{literal}$
where $V_{LiteralNode}$ is set $\{v | v \in V, c(v) = \texttt{LiteralNode}\}$, $Types_{literal}$ is set of all types of literal values {`DECIMAL`, `INTEGER`, `DOUBLE`, `STRING`}, and $Values_{literal}$ is a set of all literal values (all valid XQuery decimal numbers, integers, double numbers and strings).

Set $\mathcal{P}$ contains other similar functions but due to their large number, we do not define them formally. Functions $varName_{VarRefNode}$, $axisKind_{AxisNode}$, $operator_{OperatorNode}$ are examples of commonly used functions from $\mathcal{P}$. Their meaning will be explained in a place of their usage. For details, see [29].

### 6.4.1 Syntactic Abbreviations

Since we need to use the syntax tree in pseudo-algorithms, we define the following abbreviations to make its usage more simple.

For every $v \in V$, abbreviation

- $v.p$ stands for $p_{c(v)}(v)$ if $p_{c(v)} \in \mathcal{P}$. This is a shortened syntax for functions from $\mathcal{P}$. Assuming $v \in V$, $c(v) =$ `VarRefNode`, and $varName_{VarRefNode} \in \mathcal{P}$; expression $varName_{VarRefNode}(v)$ can be abbreviated to $v.varName$.
- $v.getChildren()$ stands for $\{w | w \in V, (v, w) \in E\}$ which is a set of all children of $v$.

- $v.getChild(i)$, $i \in \mathbb{N} \cup \{0\}$, stands for $w \in V$ so that $o(v)(v, w) = i + 1$. In other words, it is a child of $v$ with sequential number $i$ in order specified by $o$, starting with 0.
- $v.getChild(class), class \in C$, stands for $w \in V$ so that $(v, w) \in E$ and $c(w) = class$, if $\Big|\{u|u \in V, (v, u) \in E, c(u) = class\}\Big| = 1$. It is a node satisfying two conditions; it is a child of $v$ and its class is *class*. The function is defined when there is exactly one child of $v$ of *class* class.
- If exists $u \in V$ so that $(u, v) \in E$, $v.getParent()$ stands for that $u$, which is a parent of $v$.

## 6.4.2 Closer Look at the Nodes of the Syntax Tree

The node classes of the syntax are organized in an is-a hierarchical structure, commonly used in the object oriented programming languages, where an object can be of several types. This hierarchy is shown in tables in Tables 6.1, 6.2. The tables are composed of classes in italic and their subclasses. The classes with names in bold represent a common class of a group of subclasses, and these classes cannot be directly used in the syntax tree (abstract classes). The remaining non-bold nodes represent particular constructs of the XQuery language and nodes of these classes can be used in the syntax tree.

For example, an instance of the syntax tree cannot directly contain nodes of `Node` and `ExprNode` classes (for every $v \in V, c(v) \neq \texttt{Node}, c(v) \neq \texttt{ExprNode}$), but it can contain nodes of `AttributeNode` and `LiteralNode` classes. Regarding the multiplicity of types, a node of `LiteralNode` class is also considered to be of two indirect types: `ExprNode` and `Node`.

The node classes can be classified into three groups: inner node classes, leaf node classes and node classes that can be both inner and leaf. Inner node classes (marked `I`) stand for XQuery constructs that are composed of other constructs and can be further divided. An example of such class is `FLWORExprNode` which is composed of `TupleStreamNode`, `WhereClauseNode`, `OrderByClauseNode` and `ReturnClauseNode` classes. Leaf node classes (marked `L`) represent elements of XQuery language that cannot be further divided. For example, `LiteralNode`. Node `FunctionCallNode` is an example of the third group (marked `IL`). Function call of a function without arguments is represented by a leaf node while function call with arguments is represented by an inner node and its subnodes are nodes representing those arguments.

| ***Node*** | | ***ExprNode*** |
|---|---|---|
| `AttributeNode (I)` | `ModuleNode (I)` | `CommaOperatorNode (I)` |
| `AttrListNode (IL)` | `NameNode (IL)` | `ConstructorNode (I)` |
| `AxisNode (I)` | `OrderByClauseNode (I)` | `ContextItemExprNode (L)` |
| `CaseClauseNode (I)` | `OrderSpecNode (I)` | `EmptySequenceNode (L)` |
| `CaseClausesNode (I)` | `ParamListNode (I)` | `ExtensionExprNode (I)` |
| `CDataSectionNode (L)` | `ParamNode (I)` | `FLWORExprNode (I)` |
| `ContentNode (IL)` | `PITargetNode (IL)` | `FunctionCallNode (IL)` |
| `DefaultCaseNode (I)` | `PragmaListNode (I)` | `IfExprNode (I)` |
| `EntityRefNode (L)` | `PragmaNode (L)` | `LiteralNode (L)` |
| **ExprHolderNode** | `PredicateListNode (I)` | `OperatorNode (I)` |
| **ExprNode** | **PrologChildNode** | `OrderedExprNode (I)` |
| `FunctionBodyNode (IL)` | `StepExprNode (I)` | `PathExprNode (I)` |
| `CharRefNode (L)` | `StringNode (L)` | `QuantifiedExprNode (I)` |
| `InClausesNode (I)` | `TupleStreamNode (I)` | `TypeswitchExprNode (I)` |
| **ItemTypeNode** | `TypeNode (L)` | `UnorderedExprNode (I)` |
| `LocationHintNode (L)` | **VariableBindingNode** | `ValidateExprNode (I)` |
| `LocationHintsNode (IL)` | `VarValueNode (IL)` | `VarRefNode (L)` |
| **ModuleChildNode** | | |

Table 6.1: Syntax tree node types part 1. For details, see [29].

| ***PrologChildNode*** | ***ExprHolderNode*** |
|---|---|
| `BaseURIDeclNode (L)` | `BindingSequenceNode (I)` |
| `BoundarySpaceDeclNode (L)` | `DefaultValueNode (I)` |
| `ConstructionDeclNode (L)` | `ElseExpressionNode (I)` |
| `CopyNamespacesDeclNode (L)` | `OperandExpressionNode (I)` |
| `DefaultCollationDeclNode (L)` | `ReturnClauseNode (I)` |
| `DefaultNamespaceDeclNode (L)` | `TestExpressionNode (I)` |
| `EmptyOrderDeclNode (L)` | `ThenExpressionNode (I)` |
| `FunctionDeclNode (I)` | `WhereClauseNode (I)` |
| **ImportNode** | ***ItemTypeNode*** |
| `NamespaceDeclNode (L)` | `AnyItemNode (L)` |
| `OptionDeclNode (L)` | `AtomicTypeNode (L)` |
| `OrderingModeDeclNode (L)` | `KindTestNode (IL)` |
| `VarDeclNode (I)` | `NameTestNode (L)` |
| ***ImportNode*** | ***VariableBindingNode*** |
| `ModuleImportNode (I)` | `ForClauseNode (I)` |
| `SchemaImportNode (I)` | `InClauseNode (I)` |
| ***ModuleChildNode*** | `LetClauseNode (I)` |
| `ModuleDeclNode (L)` | *`StepExprNode`* `(I)` |
| `PrologNode (IL)` | `SelfOrDescendantStepNode (L)` |
| `QueryBodyNode (I)` | |

Table 6.2: Syntax tree node types part 2. For details, see [29].

Some pseudo-code algorithms in the following sections need to determine a class of a node. The node's direct class can be determined by function $c$ from the definition of the syntax tree in Definition 6.1, and hence, we also can determine its indirect classes. For the purpose of pseudo code, we define the following function.

**Definition 6.2** (Function $is(v, class)$)**.** For every $v \in V$ and every $class \in C$, function $is(v, class)$ returns boolean value `true` if $c(v) = class$ or $c(v)$ is a (direct or indirect) subclass of $class$, according to the described principle. Otherwise, it returns `false`.

For instance, assuming $v \in V, c(v) =$ `LiteralNode`, calls $is(v,$ `LiteralNode`$)$ and $is(v,$ `ExprNode`$)$ return `true`, while call $is(v,$ `ContentNode`$)$ returns `false`.

### 6.4.3 Characteristics of the Syntax Tree

An important characteristic of the syntax tree is related to a definition of local variables and their scope in the XQuery language. The representation of a definition of a local variable in the syntax tree is a node of `VariableBindingNode` class. Nodes of that class have only two children; a node representing the type of the variable and a node representing the binding expression (expression defining the value of the variable, and thus, it cannot use the variable). Hence, the entire subtree does not contain any expressions that use the variable. Therefore the scope of the new variable is not the subtree of the `VariableBindingNode` class node. It depends on the type of XQuery construct that the variable binding is an (indirect) component of.

For example, a node of `FLWORExprNode` class contains four subnodes of `TupleStreamNode`, `WhereClauseNode`, `OrderByClauseNode` and `ReturnClauseNode` classes. The `TupleStreamNode` class node contains a list of nodes of `VariableBindingNode` class which define variables valid in all other three subnodes of the FLWOR node.

This characteristic is explicitly described, because several algorithms later in this chapter rely on it.

### 6.4.4 Syntax Tree Example

A syntax tree constructed from sample query in Listing B.19 is shown in Figure 6.2.

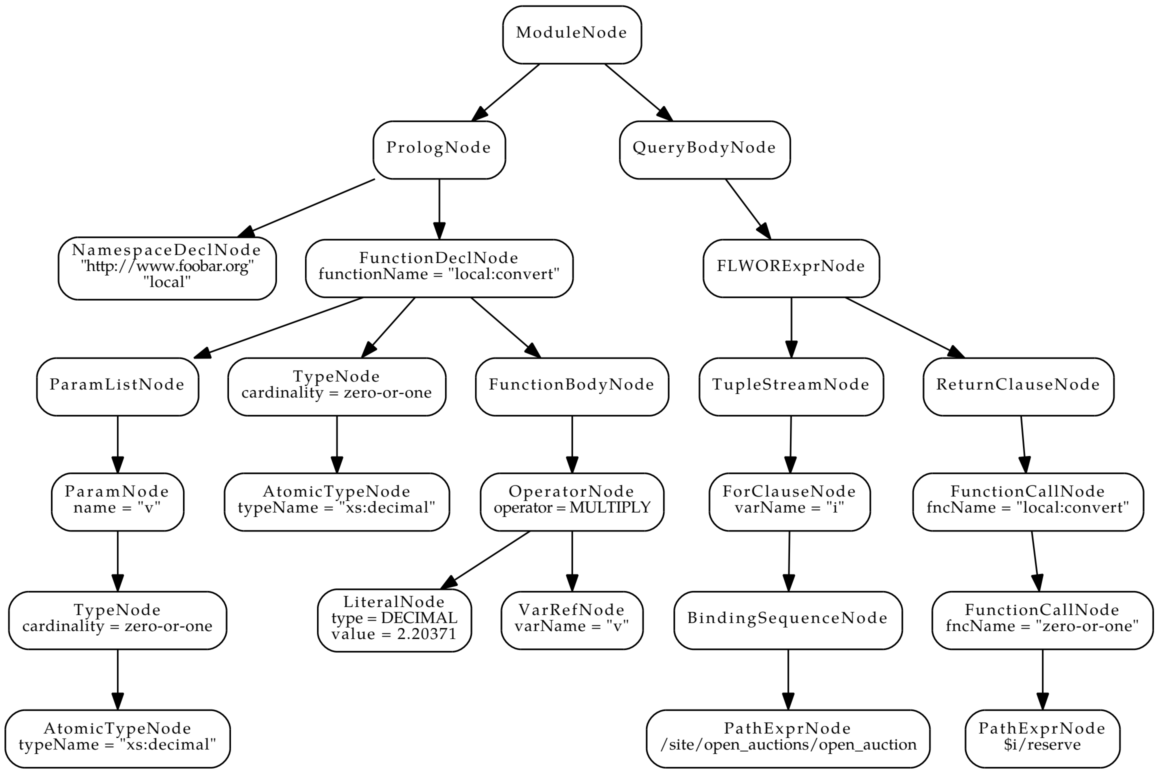

ModuleNode
PrologNode
QueryBodyNode
NamespaceDeclNode "http://www.foobar.org" "local"
FunctionDeclNode functionName = "local:convert"
FLWORExprNode
ParamListNode
TypeNode cardinality = zero-or-one
FunctionBodyNode
TupleStreamNode
ReturnClauseNode
ParamNode name = "v"
AtomicTypeNode typeName = "xs:decimal"
OperatorNode operator = MULTIPLY
ForClauseNode varName = "i"
FunctionCallNode fncName = "local:convert"
TypeNode cardinality = zero-or-one
LiteralNode type = DECIMAL value = 2.20371
VarRefNode varName = "v"
BindingSequenceNode
FunctionCallNode fncName = "zero-or-one"
AtomicTypeNode typeName = "xs:decimal"
PathExprNode /site/open_auctions/open_auction
PathExprNode $i/reserve

Figure 6.2: Sample syntax tree 1

## 6.5 Step 2: Static Analysis of Expression Types

In the second step, we statically (i.e. without evaluation of the queries) determine types of expressions in the syntax tree. Information on the types of expressions can be used by consecutive steps of the algorithm. The consecutive steps will not use the determined types of all expressions, however this step may be useful in a future extending.

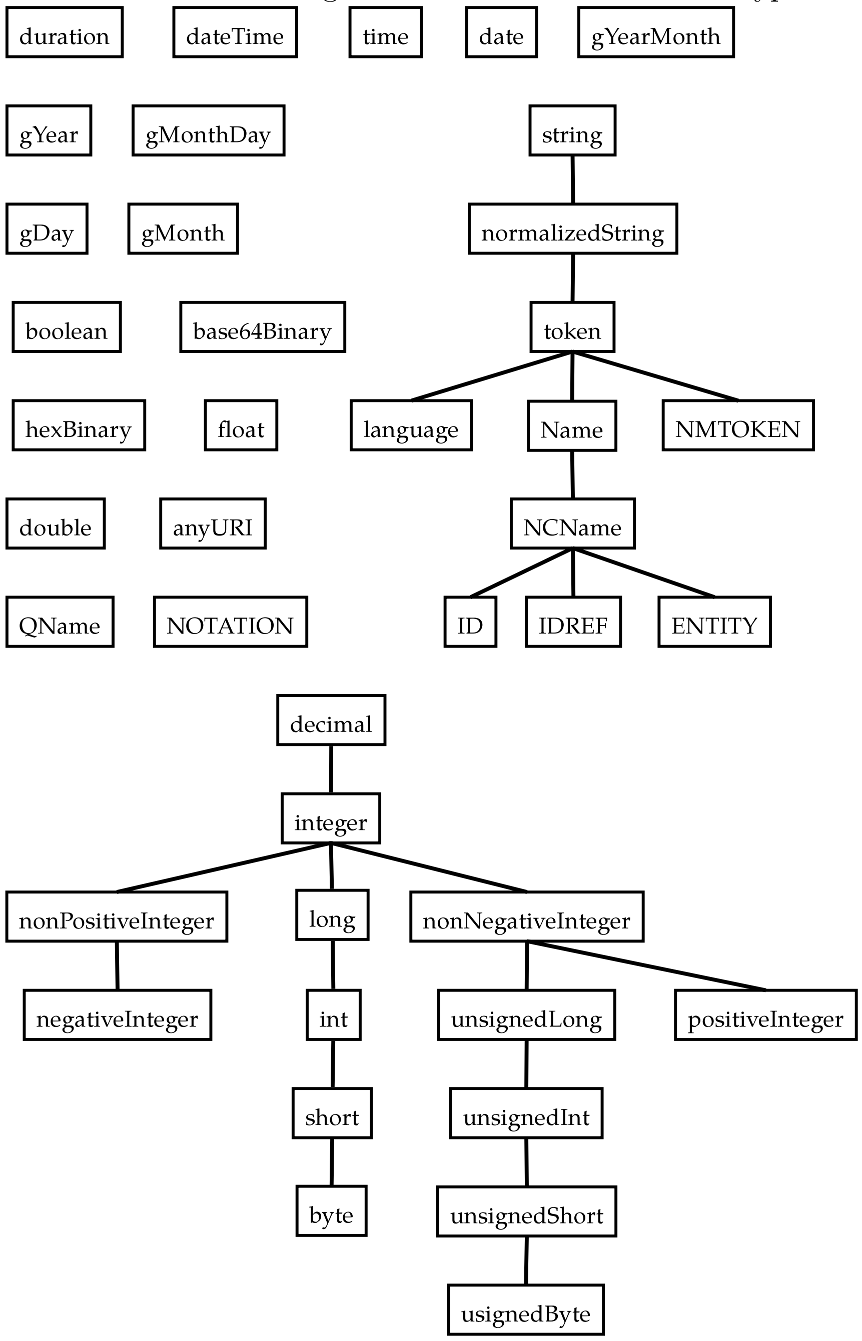
The analysis of expression types can be divided into three substeps. Determination of return types of functions, determination of types of global variables, and finally determination of types of expressions.

But firstly, we describe types of expressions we want to capture and their features.

### 6.5.1 Expression Types

- XML Schema built-in atomic types. See Figure 6.3.
- Types *ElementType*, *AttributeType*, *NodeType*, *TextNodeType*, *CommentType*, *ProcessingInstructionType*, *DocumentType* representing an element, attribute, node, text node, comment, processing instruction, doc-

Figure 6.3: XSD built-in atomic types



ument node.

- Type representing a node or a set of nodes selected by a certain path expression. The path expression is included in this type. Let this type be identified as *PathType*.

- *UnknownType* representing a type without known details, which does not suit one of the three previous types. An example is XSD type `anyType`.

### 6.5.2 PathType

PathType contains additional information. The represented path is contained by a list of its steps, in particular instances of `StepExprNode`. If a step is a reference to a variable whose type is PathType, we also want to include this information. Therefore, PathType contains association between the steps and other PathTypes and this association is defined for the PathType variable steps.

To distinguish between a common PathType selecting a set of nodes and a PathType bound to a for variable in a FLWOR expression, PathType structure contains a flag `isForBound`.

Additionally, PathType contains a list of special functions that were called with an argument of PathType type. The motivation is that in some cases of function calls, we want to know that the function call is performed with an instance of PathType because then, we can determine a type of the function call more precisely. Those special functions are built-in functions `data`, `min`, `max`, `avg`, `sum`, `distinct-values`, `zero-or-one`, `exactly-one`. And other may be added, when needed.

In summary, we represent PathType as a structure with the following member variables.

- `steps` - A list of `PathExprNode` instances.

- `substeps` - An association between variable-referencing steps and instances of PathType type.

- `isForBound` - Boolean flag determining if the type was bound to a for variable in a for clause.

- `specialFunctionCalls` - List of special functions called with this instance as an argument.

### 6.5.3 Cardinality of Types

To capture sequences, we assign the first two categories of types (all types except for PathType and UnknownType) with its cardinality as proposed in [29]. Each of those types can be perceived as a sequence. A type representing one value or one item can be perceived as a sequence of exactly one item. The cardinality expresses one of the following five sequence types.

- An empty sequence.
- A sequence of exactly one item.
- A sequence containing zero or one item (modifier `?`).
- A sequence containing zero or more items (modifier `*`).
- A sequence containing one or more items (modifier `+`).

PathType is not assigned with the cardinality since we do not evaluate the queries, and therefore, we cannot determine if a certain XQuery path targets zero, one or more nodes. Alike, UnknownType is neither assigned with the cardinality. Expressions of UnknownType are not utilized it the inference, therefore, their cardinality is not needed.

### 6.5.4 Determination of Function Return Types

Determination of return types of functions is needed because function calls can appear in expressions. A return type of a function can be determined at the moment the analysis of expressions encounters a call of the function; however, it involves multiple transitions of the syntax tree in a search for a definition of a particular function.

Instead, the syntax tree can be searched just once, before the analysis of expressions, and return types of all functions found are stored.

A simple algorithm is presented in Algorithm 6.2. It uses `getFunctionDeclarationNodes` function, defined in Algorithm 6.1, which returns a list of all function declaration nodes in the syntax three. Actually, it does not have to search the whole syntax tree as the function declaration nodes can be present only in the query prolog section.

Without loss of generality, we will focus on locally defined functions. We will not determine types of functions defined in other modules since the principle is similar but it is needed to look up the definitions in syntax trees of other queries.

**Algorithm 6.1** Function `getFunctionDeclarationNodes`: Retrieval of Function Declaration Nodes

**Input:** *syntaxTree*: A reference to the root node of a syntax tree.

**Output:** A list of syntax tree nodes representing function declarations.

```
1: $prologNode$ := null
2: for each $moduleChildNode \in syntaxTree.getChildren()$ do
3:    if $is(moduleChildNode$, PrologNode) then
4:       $prologNode := moduleChildNode$
5:    end if
6: end for
7: $functionDeclarationNodes$ := an empty list
8: if $prologNode \neq$ null then
9:    for each $prologChildNode \in prologNode.getChildren()$ do
10:      if $is(prologChildNode$, FunctionDeclNode) then
11:         add $prologChildNode$ to $functionDeclarationNodes$
12:      end if
13:   end for
14: end if
15: return functionDeclarationNodes
```

Return types of built-in functions are fixed and, thus, there is no need to analyze them. Also, since a determination of a prefix for built-in functions is a technical issue, we assume that built-in functions are either prefixed by `fn` or not prefixed.

For the rest of the thesis, we assume function `getFunctionReturnType` which takes a function name and returns the return type of the function if the function is either built-in or it is recorded in the associative array. Otherwise, `null` is returned.

Later phases of the algorithm need to process the declarations of certain functions. Therefore, we also store references to the entire function declaration nodes, accessible using function `getFunctionDeclNode`.

### 6.5.5 Auxiliary Functions

Before proceeding to the next phase of the algorithm, we introduce auxiliary functions used in pseudo algorithms.

Function `memorizeType` takes two arguments `node` and `type` and it memorizes the given type of the specified expression node. This information can be then retrieved by function `getType`, specifying the particular node as its argument.

**Algorithm 6.2** Processing of Functions

**Input:** *syntaxTree*: A reference to the root node of a syntax tree.

**Output:** An associative array of function names with their types and references to the their declaration nodes.

```
1: $functionArray$ := an empty associative array
2: for each $functionDeclarationNode \in$
   $getFunctionDeclarationNodes(syntaxTree)$ do
3:    $functionName := functionDeclarationNode.funcName$
4:    $typeNode := functionDeclarationNode.getChild(TypeNode)$
5:    $functionArray[functionName] :=$
      $(getTypeTN(typeNode), functionDeclarationNode)$
6: end for
7: return $functionArray$
```

**Algorithm 6.3** Function `getTypeTN`: Extraction of a Type from TypeNode

**Input:** *typeNode*: Syntax tree node of TypeNode class.

**Output:** Type extracted from *typeNode*.

```
1: $type$ := UnknownType
2: $cardinality := typeNode.cardinality$
3: $itemTypeNode := typeNode.getChild(ItemTypeNode)$
4: if $is(itemTypeNode$, AtomicTypeNode) then
5:    $type$ := XSD atomic built-in type $itemTypeNode.typeName$, $cardinality$
6: else if $is(itemTypeNode$, KindTestNode) then
7:    $type := itemTypeNode.nodeKind$, $cardinality$
8: end if
9: return $type$
```

Another group consists of functions `set`, `add`, and `get`. Function `set` memorizes a given value of a specified property of a specified node. It is used to assign a node with a named value, for example, `set(someNode, color, "red")` will assign `someNode` with string value `"red"` which can be then retrieved by function `get`, specifying the node and the property name. For example, `get(someNode, color)` returns `"red"`. A subsequent call of `set` assigning a node with a value of already assigned property will overwrite it, leaving the property with the newer value. However, a property can have several values and this can be achieved using function `add` with the same syntax as function `set` but instead of overwriting the existing property, `add` will append the new value to the existing ones. Then, a call of `get` on this node and property returns a list of all its values, preserving the order of their addition.

### 6.5.6 Determination of Global Variable Types

A similar approach as in the case of functions can be applied to determine types of global variables. Alike the functions, the global variables are defined in the prolog section. A type of a variable can be explicitly specified in its definition, for instance `declare variable $x as xs:byte := 12;`. If it is not, it may often be deducible from the binding expression. Again, we do not analyze external variables for the same reason as in case of external functions.

The algorithm iterates through the variable declaration nodes from the prolog and if the variable is explicitly assigned with its type, the type is noted. Otherwise, an attempt of the type deduction of the binding expression is made. The deduction of the type from the expression is presented in Algorithm 6.4. On input, it takes an expression node and types of local variables that are valid in the expression (also called context of variables or variable context). The presented algorithm is just shortened illustration since the complete version is too long to be presented in this text. Nevertheless, the principle is straightforward. Depending on the class of the expression node (show in Table 6.1), its type can be either determined directly or it depends on its subexpressions.

The meaning of the code at lines 6-14 is the following. If a variable is bound to an expression whose type is PathType, we want to assign the variable reference expression with PathType which represents path containing exactly one step, the variable reference. Some algorithm in later sections searches for paths starting with a certain variable and we want them to include also that types of expressions.

A parts of the function's semantic are separated in other functions, like the one in Algorithm 6.5, which is not presented completely because of the same

**Algorithm 6.4** Function `determineExpressionType`

**Input:**

*exprNode*: An expression node in the syntax tree.

*contextVarTypes*: Types of local variables valid in the current subtree.

**Output:** The expression type.

```
1: $type$ := UnknownType
2: if $is(exprNode$, LiteralNode$)$ then
3:     $type$ := XSD atomic type $exprNode.type$, exactly-once cardinality
4: else if $is(exprNode$, FunctionCallNode$)$ then
5:     $type := getFunctionReturnType(exprNode.fncName)$
6: else if $is(exprNode$, VarRefNode$)$ then
7:     $type := getVariableType(contextVarTypes, exprNode.varName)$
8:     if $type$ is PathType then
9:         $step$ := a path step (new instance of StepExprNode) representing the
           variable reference
10:        $substeps$ := an empty associative array
11:        $substeps[step] := type$
12:        $steps$ := an array containing one item which is $step$
13:        $type$ := PathType containing $steps$ and $substeps$
14:    end if
15: else if $is(exprNode$, PathExprNode$)$ then
16:    $type := createPathType(exprNode)$
17: else if $is(exprNode$, OperatorNode$)$ then
18:    $type := determineOperatorType(exprNode, contextVarTypes)$
19: else if $is(exprNode$, FLWORExprNode$)$ then
20:    $returnClauseNode := exprNode.getChild(ReturnClauseNode)$
21:    $type := createForUnboundType(getType(returnClauseNode))$
22: end if
23: return type
```

**Algorithm 6.5** Function `determineOperatorType`

**Input:**

*operatorNode*: An operator expression node in the syntax tree.

*contextVarTypes*: Types of local variables valid in the current subtree.

**Output:** The operator expression type.

```
1: $type$ := UnknownType
2: $operator$ := $operatorNode.operator$
3: if $isOperatorClassComparison(operator)$ then
4:   $type$ := boolean, exactly-one cardinality
5: else if $isOperatorClassAddition(operator)$ then
6:   $leftOperandType := getType(exprNode.leftSide)$
7:   $rightOperandType := getType(exprNode.rightSide)$
8:   if $isNumericType(leftOperandType)$
     $\wedge$ $isNumericType(rightOperandType)$ then
9:     $type := selectMoreGeneralNumericType(leftOperandType,$
       $rightOperandType)$
10:  else if $isNumericType(leftOperandType)$ then
11:    $type := leftOperandType$
12:  else if $isNumericType(rightOperandType)$ then
13:    $type := rightOperandType$
14:  end if
15: end if
16: return type
```

**Algorithm 6.6** Function `isOperatorClassComparison`

**Input:** *operator*: A representation of a XQuery operator.

**Output:** true if *operator* is a comparison operator, false otherwise.

```
1: if $operator$ equals one of GEN_EQUALS, GEN_GREATER_THAN,
   GEN_GREATER_THAN_EQUALS, GEN_LESS_THAN,
   GEN_LESS_THAN_EQUALS, GEN_NOT_EQUALS, VAL_EQUALS,
   VAL_GREATER_THAN, VAL_GREATER_THAN_EQUALS,
   VAL_LESS_THAN, VAL_LESS_THAN_EQUALS,
   VAL_NOT_EQUALS then
2:   return true
3: else
4:   return false
5: end if
```

**Algorithm 6.7** Function `isOperatorClassAddition`

**Input:** *operator*: A representation of a XQuery operator.

**Output:** true if *operator* is an addition operator, false otherwise.

```
1: if operator equals one of PLUS, MINUS, UNARY_PLUS, UNARY_MINUS
   then
2:     return true
3: else
4:     return false
5: end if
```

**Algorithm 6.8** Function `isNumericType`

**Input:** *type*: A representation of a type.

**Output:** true if *type* represents one of the XSD atomic built-in numeric types, false otherwise.

```
1: if type represents one of float, double, decimal, integer, long, int, short, byte,
   nonPositiveInteger, negativeInteger, nonNegativeInteger, positiveInteger, un-
   signedLong, unsignedInt, unsignedShort, unsignedByte then
2:     return true
3: else
4:     return false
5: end if
```

reason.

Function `determineExpressionType` is called within function `analysisOf-ExpressionTypes` presented in Algorithm 6.9, which recursively determines types of all subexpressions. Two important questions about this function emerge. How (in which order) are the tree nodes recursively processed and how does the function handle definitions (bindings) of new variables which may also appear in the expressions?

The order of the recursion is firstly to process children of a node and then the node itself. The order of child processing is specified by function *o* from the definition of the syntax tree in Definition 6.1. The reason is that an expression node may need to know types of its subexpressions to determine its own type and we want to determine types of all expressions. Therefore, using the described order, subexpressions of an expression are processed first, and then the expression itself without any need for further recursion, because the types of the subexpressions are already determined.

The handling of new variable definitions relies on the fact that the definitions make the new variables valid only in nodes with a higher sequence number when numbered in the order described in the previous paragraph. Therefore, the left-most (the first amongst the ordered subnodes) subnode of a certain node can be processed without an extension of the variable context. And, if the left-most subnode extends the variable context for the following nodes in the numbering, it can be easily handled, because the types of the new variables can be directly determined since the binding expressions are already processed (thus, their types are known). Every node "knows" whether or not it may define new variables for nodes with higher sequence numbers. This is expressed in the algorithm by the condition stating at line 10. If it does define new variables, they are memorized. Later, upon processing of its parent, they are retrieved and the variable context is extended. This is done at lines 4 and 5. Function `mergeContextVarTypes` presented in Algorithm 6.10 writes every record from its second argument (the new variables) into its first argument (the variable context). If there are records for variables with the same names in the context, they are overwritten to correspond to the variable overlapping. It is important to note that we assume the variable context is not passed by reference but by value, and hence, the variable context of a certain node is not affected by the recursive processing of its subnodes.

Function `analysisOfExpressionTypes` called upon a binding expression of a global variable determines the type of the binding expression, and hence, the type of the variable. In case of global variables, the argument `contextVarTypes`

**Algorithm 6.9** Function `analysisOfExpressionTypes`

**Input:**

*startingNode*: A node determining a subtree to perform the analysis on.

*contextVarTypes*: Types of local variables valid in the current subtree.

```
1: for each $i \in \{1, \ldots, |startingNode.getChildren()|\}$ ordered from the lowest
   to the highest do
2:   $subnode := startingNode.getChild(i)$
3:   $analysisOfExpressionTypes(subnode, contextVarTypes)$ // recursion
4:   $newVars := get(subnode, newVars)$
5:   $contextVarTypes := mergeContextVarTypes(contextVarTypes, newVars)$
6: end for
7: if $is(startingNode$, ExprNode$)$ then
8:   $memorizeType(startingNode, determineExpressionType(startingNode))$
9: end if
10: if $is(startingNode$, VariableBindingNode$)$ then
11:   $type :=$ null
12:   $typeNode := startingNode.getChild(TypeNode)$
13:   if $typeNode \neq null$ then
14:     $type := getTypeTN(typeNode)$
15:   else
16:     $type := determineExpressionType(startingNode.$
        $getChild(BindingSequenceNode).getChild(ExprNode),$
        $contextVarTypes)$
17:     if $is(startingNode$, ForClauseNode$)$ then
18:       $type := createForBoundType(type)$
19:     end if
20:   end if
21:   $set(startingNode, newVars, (startingNode.varName, type))$
22: else if $is(startingNode$, TupleStreamNode$) \vee is(startingNode$,
    InClausesNode$)$ then
23:   for each $varBindingNode \in startingNode.getChildren()$ do
24:     $add(startingNode, newVars, get(varBindingNode, newVars))$
25:   end for
26: end if
```

**Algorithm 6.10** Function `mergeContextVarTypes`

```
Input:
    contextVarTypes: Types of local variables valid in current context.
    extendingVarTypes: New variables in the same structure as the first argument.
Output: The context variable types extended with the variable types from the
    second argument.
1: for each $varName \in keys(extendingVarTypes)$ do
2:     $contextVarTypes[varName] := extendingVarTypes[varName]$
3: end for
4: return $contextVarTypes$
```

is empty, because there are no local variables valid in the prolog section.

**Algorithm 6.11** Function `getVariableType`

```
Input:
    contextVarTypes: Types of local variables valid in current context.
    var: A variable to determine the type of.
Output: The type of variable var.
1: if $contextVarTypes[var]$ is defined then
2:     return $contextVarTypes[var]$
3: else
4:     return $getGlobalVariableType(var)$
5: end if
```

The types of processed global variables are available trough function `getGlobalVariableType`. If the function gets a variable that is not a global one, it returns `null`. Function `getVariableType` called in Algorithm 6.4 is defined in Algorithm 6.11. It checks if the specified variable is amongst the given local variables. If so, it returns its type, else it handles the variable as a global one and if such global variable does not exist the result is `null`.

### 6.5.7 Determination of Expression Types

To determine types of expressions, we use already introduced function `analysisOfExpressionTypes`. The starting node (its first argument) is the node representing the query body and the variable context is empty as there cannot be any local variable valid in the body node.

We can also determine expression types in functions. To do this for a certain

**Algorithm 6.12** Function `createPathType`

**Input:** *pathExprNode*: A reference to a PathExprNode to create the PathType from.

**Output:** The PathType of *pathExprNode* expression.

```
1: $steps$ := an empty array
2: $substeps$ := an empty associative array
3: for each $step \in pathExprNode.getChildren()$ do
4:    $detailNode := step.getChild(ExprNode)$
5:    if $detailNode \neq$ null then
6:       if $is(detailNode,$ VarRefNode$)$ then
7:          $type := getType(detailNode)$ // PathType
8:          if $type.isForBound$ then
9:             $substeps[stepNode] := type$
10:            add $step$ to $steps$
11:         else
12:            add all $type.steps$ to $steps$
13:         end if
14:      else
15:         add $step$ to $steps$
16:      end if
17:   else
18:      add $step$ to $steps$
19:   end if
20: end for
21: return $PathType$ with $steps$, $substeps$, and $isForBound$ set to false
```

**Algorithm 6.13** Function `createForBoundType`

**Input:** *type*: A type to create the for bound type from.

**Output:** The for bound type from *type*.

```
1: if $type$ is UnknownType then
2:    return UnknownType
3: else if $type$ is NodeType or XSD atomic type then
4:    return $type$ with cardinality set to exactly-one
5: else
6:    // It is PathType
7:    return $type$ with $isForBound$ flag set to true
8: end if
```

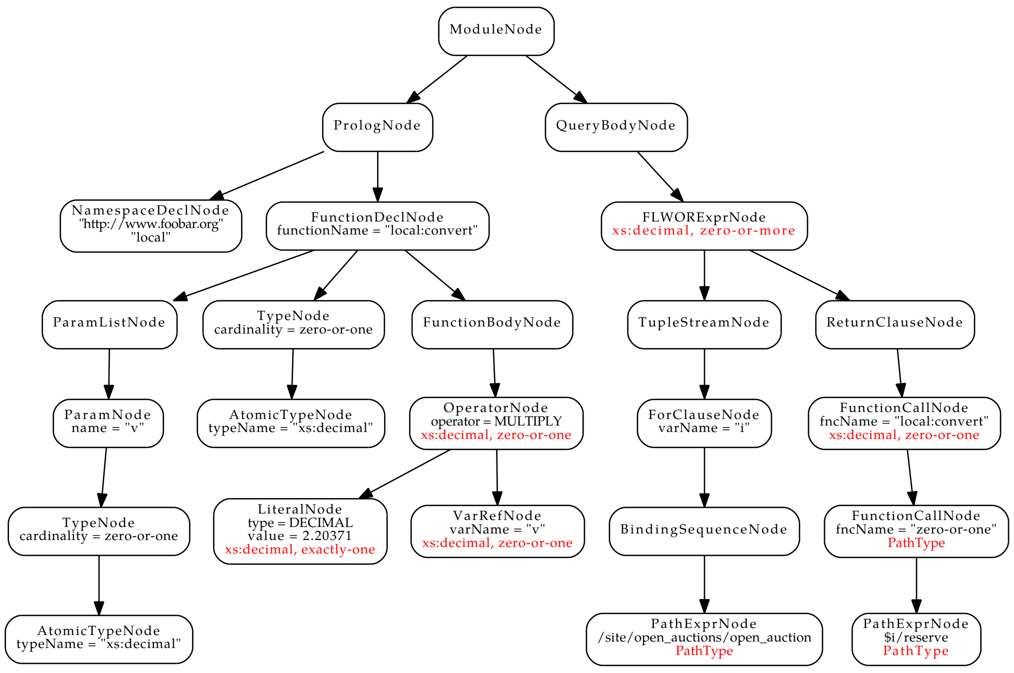

ModuleNode
PrologNode
QueryBodyNode
NamespaceDeclNode "http://www.foobar.org" "local"
FunctionDeclNode functionName = "local:convert"
FLWORExprNode xs:decimal, zero-or-more
ParamListNode
TypeNode cardinality = zero-or-one
FunctionBodyNode
TupleStreamNode
ReturnClauseNode
ParamNode name = "v"
AtomicTypeNode typeName = "xs:decimal"
OperatorNode operator = MULTIPLY xs:decimal, zero-or-one
ForClauseNode varName = "i"
FunctionCallNode fncName = "local:convert" xs:decimal, zero-or-one
TypeNode cardinality = zero-or-one
LiteralNode type = DECIMAL value = 2.20371 xs:decimal, exactly-one
VarRefNode varName = "v" xs:decimal, zero-or-one
BindingSequenceNode
FunctionCallNode fncName = "zero-or-one" PathType
AtomicTypeNode typeName = "xs:decimal"
PathExprNode /site/open_auctions/open_auction PathType
PathExprNode $i/reserve PathType

Figure 6.4: Sample syntax tree 1 after the static analysis of types

function, `analysisOfExpressionTypes` function has to be called with a function declaration node as the starting node. In this case, the function declaration node contains a subnode specifying function's formal arguments. These arguments are set as the variable context for the function body represented by another subnode.

Figure 6.4 shows the syntax tree from Figure 6.2 after the static analysis of expression types. The types are shown in red color. Note that the node representing `zero-or-one` function call is of a PathType type, as well as its argument. It is so, because the function returns its argument unchanged, and thus, we included the function in the special functions list in the PathType definition.

Also note, that the function call node of `local:convert` in the FLWOR return clause has `zero-or-one` cardinality and the FLWOR node has `zero-or-more` cardinality. The change of cardinality is a result of the for unbinding shown in Algorithm 6.14.

## 6.6 Step 3: Inference of Built-in Types

In this step, the algorithm goes through the syntax tree to infer types of elements and attributes from expressions using the type information from the previous step. These two steps could be merged together but for better comprehension we present it separately.

How are the types inferred from the expression types? We do not exploit all

**Algorithm 6.14** Function `createForUnboundType`

**Input:** *type*: A type to create the for unbound type from.

**Output:** The for unbound type from *type*.

```
1: if type is UnknownType then
2:     return UnknownType
3: else if type is NodeType or XSD atomic type then
4:     return type with cardinality set to zero-or-more
5: else
6:     // It is PathType
7:     return type with $isForBound$ flag set to false
8: end if
```

expressions. Only expressions of a particular type are exploited. Specifically, an expression has to contain a subexpression $E$ of PathType type (expression representing a certain element or attribute or a set of elements or attributes). In the following text, the set represented by expression $E$ is denoted $S$. Another requirement is that the expression has to be either a function call or an arithmetic operation. As discussed in Chapter 4, also other XQuery constructs can be utilized to infer built-in types, but, since the principle is similar, we focus on the two mentioned.

Likewise the previous step, the syntax tree is recursively searched for expressions meeting the conditions for the type inference. A little difference is that the recursion stops at `FunctionsBodyNode` and `PathExprNode` nodes, because the processing of these nodes requires a different approach, which we do not deal with in this thesis.

The output of this step is a set of statements of the form $P \rightarrow T$, where $P$ is an instance of PathType and $T$ is an XML Schema built-in atomic type.

### 6.6.1 Function Calls

This case is quite straightforward. The algorithm encounters a function call and one of the arguments is a set of elements of attributes (subexpression $E$ representing $S$) represented by PathType $P$. To determine the type of $S$, it is only needed to determine the type of the corresponding formal argument from the definition of the function. The function is either a built-in one so its definition is known or it is defined in the prolog section. External functions are not processed as was mentioned already.

If the type $T$ of the formal argument is a built-in type or its sequence, then

$T$ is also the inferred type of $S$. The inferred statement is $P \rightarrow T$. Otherwise, no statement is inferred.

### 6.6.2 Arithmetic Operations

If the operator in an arithmetic operation is one of `+`, `-`, `div`, `mod`, `*`, `/` (the class of the expression node is `Operator` and it represents one of `PLUS`, `MINUS`, `IDIV`, `MOD`, `MUL`, `DIV` (for all operators, see Attachment A.1)), one operand is of PathType $P$ and the type $T$ of the other operand is one of numeric built-in types, then the inferred statement is $P \rightarrow T$.

If the operator is one of `<`, `>`, `<=`, `>=`, `=`, `!=` (the class of the expression node is `Operator` and it represents one of `GEN_LESS_THAN`, `GEN_GREATER_THAN`, `GEN_LESS_THAN_EQUALS`, `GEN_GREATER_THAN_EQUALS`, `GEN_EQUALS`, `GEN_NOT_EQUALS`), one operand is of PathType $P$ and the type $T$ of the other operand is one of built-in types, then the inferred statement is $P \rightarrow T$.

### 6.6.3 Example

Figure 6.5 shows a fraction of the syntax tree from Figure 6.4. When the inference of built-in types encounters the node marked by blue color, a statement will be inferred using the principle described in Chapter 6.6.1. The node represents a call of `local:convert` function.

This function has one formal argument of type $T =$ `xs:decimal`, `zero-or-one` cardinality. The real argument is of a PathType $P$. In particular, it is a PathType representing path `$i/reserve`, where the `$i` variable refers to a PathType representing for-bound `/site/open_auctions/open_auction` path, and one special function call of `zero-or-one` function is noted.

Since the criteria for the inference from a function call are met, statement $P \rightarrow T$ is inferred.

### 6.6.4 Possible Extensions

As mentioned, the proposed algorithm utilizes only a small portion of XQuery constructs which can be be possibly utilized. Also, the algorithm does not perform the inference of types inside user-defined functions (it just uses the return types determined in earlier phases). That can be easily done as we know the definitions of functions and types of their real arguments. An algorithm performing the analysis in the user-defined functions can works as follows.

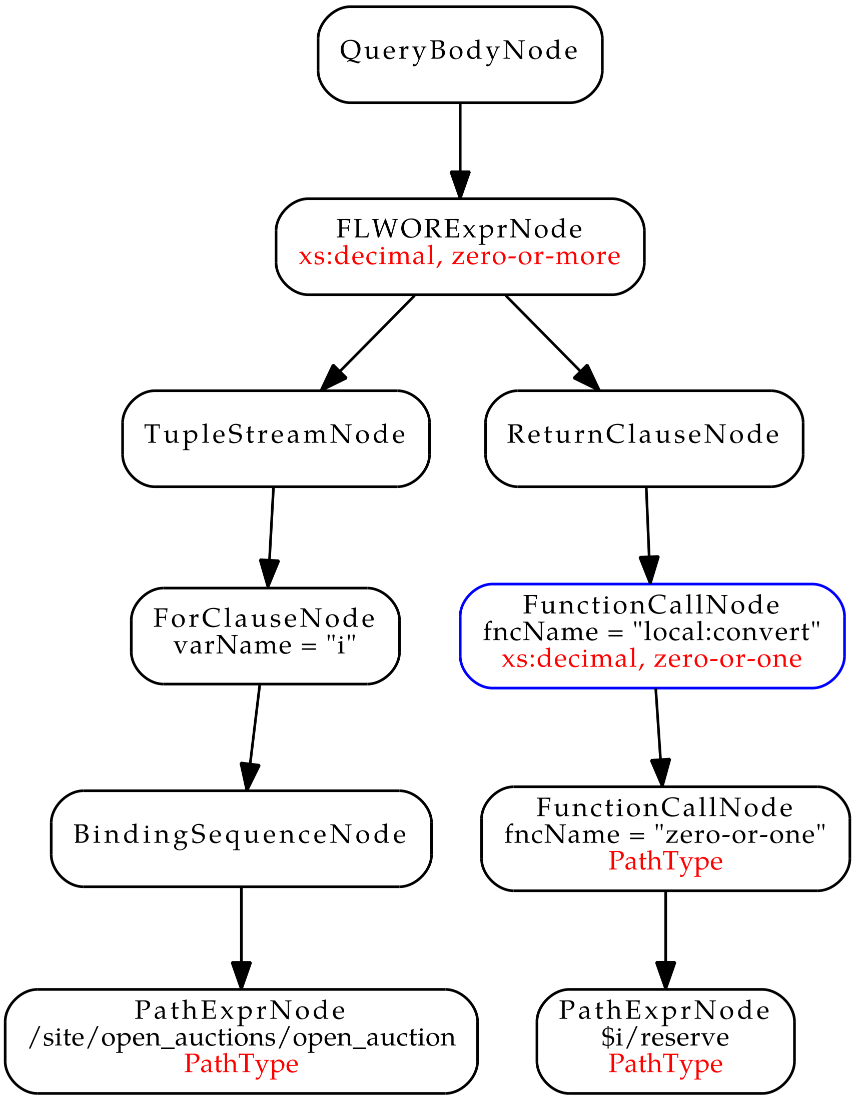


Figure 6.5: Example of an inference of a type from a function call

If a currently processed expression is a function call of an user-defined function and at least one of its real arguments is an instance of PathType, get the function declaration node. In the function body, replace the formal arguments by the real arguments and set them as the local variables, so the proposed algorithms can be applied to process the function's body. Analyze the types of expressions in the body with the new information on the types of arguments by incorporating of the proposed algorithms and replace the function's return type by the one currently determined.

A little complication is a recursion. A simple algorithm of that kind may never end because of the infinite analysis of the same recursive function(s). Therefore, we shall keep track of the currently analysed functions (e.g. in a stack of function calls) and do not process the recursive calls.

## 6.7 Step 4: Key Discovery

In this step, the algorithm discovers keys of elements, incorporating the approach from paper [26], described in Chapter 3.5, and extending it. Despite the approach was proposed, it has not been implemented yet. Therefore we will be the first to implement it and perform consecutive testing.

Firstly, the syntax tree is searched for forms of FLWOR expressions to infer the keys from. Then, as described later, certain constructs are used to support or degrade the evidence of inferred statements in a final summary.

### 6.7.1 Auxiliary Functions

Firstly, we define some auxiliary functions used by algorithms in this chapter.

Function `usesOnlyChildAndDescendantAxes` in Algorithm 6.15 takes an instance of PathType as its arguments and returns true if the path uses only child, descendant and descendant or self axes. Otherwise, it returns false. Function `usesOnlyChildAndDescendantAndAttributeAxes` is the same except it allows also attributes axis.

Function `isWithoutPredicates` in Algorithm 6.16 checks whether a path represented by a given PathType instance does not contain predicates. A Similar function is `isWithoutPredicatesExceptLastStep`. This function is not explicitly defined, because its definition is the same as the definition of function `isWithoutPredicates`, except it does not check the last step of the path. So the last step may contain predicates and the function will return true.

**Algorithm 6.15** Function `usesOnlyChildAndDescendantAxes`

**Input:** *pathType*: A PathType instance.

**Output:** A boolean result.

```
1: for each $step \in pathType.steps$ do
2:   if $step.isAxisStep$ then
3:     $axisKind := step.axisNode.getAxisKind()$
4:     if $axisKind \neq$ CHILD $\wedge$ $axisKind \neq$ DESCENDANT $\wedge$ $axisKind \neq$ DESCENDANT_OR_SELF then
5:       return false
6:     end if
7:   end if
8: end for
9: $detailNode := step.getChild(ExprNode)$
10: if $detailNode \neq$ null then
11:   if $is(detailNode,$ VarRefNode$)$ then
12:     if $usesOnlyChildAndDescendantAxes(pathType.substeps[step]) =$ false then
13:       return false
14:     end if
15:   end if
16: end if
17: return true
```

**Algorithm 6.16** Function `isWithoutPredicates`

**Input:** *pathType*: A PathType instance.

**Output:** A boolean result.

```
1: for each $step \in pathType.steps$ do
2:   if $step.hasPredicates$ then
3:     return false
4:   end if
5:   if $pathType.substeps[step] \neq$ null then
6:     if $isWithoutPredicates(pathType.substeps[step] =$ false then
7:       return false
8:     end if
9:   end if
10: end for
11: return true
```

Function `endsWithExactlyOnePredicate` in Algorithm 6.17 checks if a path represented by a given PathType instance contains exactly one predicate in its last step and if so, the predicate is returned. Otherwise, it returns false.

**Algorithm 6.17** Function `endsWithExactlyOnePredicate`

**Input:** *pathType*: A PathType instance.

**Output:** If *pathType* ends with exactly one predicate, return value is the predicate, otherwise return value is false.

```
1: lastStep := the last item from pathType.steps
2: if lastStep.hasPredicates() = false then
3:     return false
4: end if
5: if number of items in list lastStep.getChild(PredicateListNode)
   .getChildren() is higher than 1 then
6:     return false
7: end if
8: return the only item in lastStep.getChild(PredicateListNode)
   .getChild(0)
```

Function `isTargetPath` in Algorithm 6.18 checks if a path represented by a given PathType instance is so-called target path for the specified variable. A target path for a variable is a path where its first step is a reference to the variable.

Function `getTargetReturnPathTypes` in Algorithm 6.19 searches the given FLWOR for PathTypes representing so-called target return paths. A target return path is a return path in a where clause of a FLWOR. The FLWOR is processed recursively using function `getTargetReturnPathTypesRecursive` in Algorithm 6.20.

A similar couple of functions is function `getTargetReturnPaths` and function `getTargetReturnPathsRecursive`. The only difference between those two couples of functions is that the former one searches for all expressions which are of PathType representing a target return path, while the later one searches only for instances of `PathExprNode`.

To illustrate this difference, we introduce the following example. Assuming $P$ is a return target path expression for variable *var* and *flwor* is a FLWOR expression with return clause `data(`$P$`)`, function call `getTargetReturnPathTypes(`*flwor*, *var*`)` returns two PathTypes. One is the type of expression $P$ and the other is the type of the function call `data(`$P$`)` which is also PathType. On the other hand, function call `getTargetReturnPaths(`*flwor*, *var*`)` returns just one

**Algorithm 6.18** Function `isTargetPath`

**Input:**

$pathType$: A PathType instance.

$varName$: A variable name.

**Output:** Returns true, if the first step of path represented by the PathType instance is a reference to the specified variable.

```
 1: $firstStep$ := first item from $pathType.steps$
 2: $detailNode$ := $firstStep.detailNode$
 3: **if** $detailNode$ = null **then**
 4:     **return** false
 5: **end if**
 6: **if** $is(detailNode$, VarRefNode$)$ = false **then**
 7:     **return** false
 8: **end if**
 9: **if** $detailNode.varName = varName$ **then**
10:     **return** true
11: **else**
12:     **return** false
13: **end if**
```

**Algorithm 6.19** Function `getTargetReturnPathTypes`

**Input:**

$flworNode$: A FLWORExprNode instance to search for target return paths.

$varName$: A variable name.

**Output:** A list of target return paths for the specified variable, in the given FLWOR, represented by PathType instances.

```
1: $pathTypes$ := an empty array
2: $getTargetReturnPathTypesRecursive(flworNode.$
   $getChild(ReturnClauseNode), varName, pathTypes)$
3: **return** $pathTypes$
```

**Algorithm 6.20** Function `getTargetReturnPathTypesRecursive`

**Input:**

*node*: A node to search for target return paths.

*varName*: A variable name.

*pathTypes* An output list to add found target return path types to.

```
1: if $is(node, \text{ExprNode})$ then
2:     $type := getType(node)$
3:     if $type$ is $PathType \wedge isTargetPath(type, varName)$ then
4:         add $type$ to $pathTypes$
5:     end if
6: end if
7: for each $child \in node.getChildren()$ do
8:     $getTargetReturnPathTypesRecursive(child, varName, pathTypes)$
9: end for
```

PathType representing $P$.

**Algorithm 6.21** Function `getTargetReturnPaths`

**Input:**

*flworNode*: A FLWORExprNode instance to search for target return paths.

*varName*: A variable name.

**Output:** A list of target return paths for the specified variable, in the given FLWOR, represented by PathExprNode instances.

```
1: $paths :=$ an empty array
2: $getTargetReturnPathsRecursive(flworNode.$
   $getChild(ReturnClauseNode), varName, paths)$
3: return $paths$
```

## 6.7.2 Definition of Keys

To define a key and a foreign key, we adopt the definitions introduced in the original approach [26] with the following modifications.

- XPath paths are replaced by PathType instances.
- All paths can use also descendant or self axis.
- *Key paths* can use also attribute axis.

The modified definitions are as follows.

**Algorithm 6.22** Function `getTargetReturnPathsRecursive`

**Input:**

*node*: A node to search for target return paths.

*varName*: A variable name.

*paths* An output list to add found target return path types to.

```
if $is(node, \text{PathExprNode})$ then
    $type := getType(node)$
    if $type$ is PathType $\wedge$ $isTargetPath(type, varName)$ then
        add $type$ to $paths$
    end if
else if $is(node, \text{VarRefNode})$ then
    if $node.varName = varName$ then
        $type := getType(node)$
        if $type$ is PathType $\wedge$ $isTargetPath(type, varName)$ then
            add $type$ to $paths$
        end if
    end if
end if
for each $child \in node.getChildren()$ do
    $getTargetReturnPathsRecursive(child, varName, paths)$
end for
```

```
for $e_1$ in $P_1$
for $e_2$ in
    $P_2[L_2 = \$e_1/L_1]$
return $C_R$
```

Listing 6.1: Other form of the for join pattern.

```
for $e_1$ in $P_1$
let $e_2$ :=
    $P_2[L_2 = \$e_1/L_1]$
return $C_R$
```

Listing 6.2: Other form of the let join pattern.

**Definition 6.3** (Key)**.** A key is a construct $(C, P, \{L\})$, where $C$, $P$ and $L$ are PathType instances without predicates and without special function calls that use only child, descendant, and descendant or self ($L$ also attribute) axes. $C$ is called *context path*, $P$ *target path* and $L$ *key path.* $C$ can be omitted, i.e. we can write $(P, \{L\})$. This is equivalent to $(/, P, \{L\})$. If $C$ is omitted we call the key *global key.* Otherwise, it is called *relative key.*

**Definition 6.4** (Foreign key)**.** A foreign key is a construct $(C, (P_1, \{L_1\}) \rightarrow (P_2, \{L_2\}))$, where $(C, P_2, \{L_2\})$ is a key and $P_1$ and $L_1$ are PathType instances without predicates and without special function calls that use only child, descendant, and descendant or self ($L$ also attribute) axes. $C$ can be omitted as in the case of keys.

An example of a global key is `(/site/people/person, {@id})`. An example of a foreign key to that key is `((/site/closed_auctions/closed_auction, {buyer/@person})` $\rightarrow$ `(/site/people/person, {@id}))`.

```
for $e_1$ in $P_1$
for $e_2$ in $P_2$
where $e_2/L_2 = \$e_1/L_1$
return $C_R$
```

Listing 6.3: Join pattern 3.

### 6.7.3 Join Patterns

The two join patterns from the original approach are shown in Listings 3.1 and 3.2. Additional join patterns are introduced in Listings 6.1, 6.2 and 6.3. In all join patterns $P_1$, $P_2$, $L_1$, $L_2$ are XQuery paths without predicates, using only child, descendant, and descendant or self ($L_1$, $L_2$ also attribute) axes. Actually, the join patterns from Listings 6.1 and 6.2 are covered by the for and let join pattern in the original approach, but we introduce them explicitly, because their structure in the syntax tree is different.

Since in this step we already know the types of all expressions in the syntax tree, the requirement that $P_1$, $P_2$, $L_1$, $L_2$ are paths of the described form can be replaced by a more general requirement that $P_1$, $P_2$, $L_1$, $L_2$ are expressions of PathType satisfying the same requirements.

As described in the original method and summarized in Chapter 3.5 we recognize two cases (O1) and (O2) of inference of keys from occurrences of the join patterns, and rules (R1 - R5) to classify each occurrence into one of these cases. For the join pattern 3, we introduce a new rule, considering the join pattern 3 of case (O1), assigned with weight of 0.5. The lesser weight is chosen because there is a lower probability that join of the join pattern 3 type is done via a key/foreign key pair.

To find the occurrences of the join patterns, the algorithm recursively, in preorder, searches the syntax tree and every node representing a FLWOR expression is processed. The FLWOR processing is shown in Algorithm 6.23. Its input is an array of couples containing names of variables which has been bound in for clauses in FLWOR expressions represented by ancestor nodes, and references to those respective for clause nodes. The processing iterates through subnodes of a current node and the iteration consists of two logical parts.

In the first one, a subnode is checked if it is a for or let clause that forms a join pattern occurrence with one of clauses from the input array. If it does, the occurrence is noted. The responsible code is partially moved to function `determineJoinPattern` in Algorithm 6.24. If the subnode does not form an occurrence, and if it is a for clause binding a new variable satisfying the conditions to be the first for clause of a join pattern, it is added to the array along with the variable name. It can be then processed in the following clauses of the current FLWOR node or in its descendant FLWOR nodes, later in the recursion.

**Algorithm 6.23** Processing of FLWOR expressions

**Input:**

$flworNode$: A node representing a FLWOR expression.

$forVars$: An array of couples of for variables and their for clause nodes.

**Output:** Updated variable $forVars$.

```
1: $bindingNodes := bindingNode \in flworNode.getChild(TupleStreamNode).getChildren()$
2: $whereClause := flworNode.getChild(WhereClauseNode)$
3: $checkJoinPattern3 :=$ false
4: $whereExpr :=$ null
5: if $whereClause \neq$ null then
6:    $whereExpr := whereClause.getChild(ExprNode)$
7:    if $is(whereExpr,$ OperatorNode$) \wedge whereExpr.operator =$ GEN_EQUALS then
8:       $checkJoinPattern3 :=$ true
9:    end if
10: end if
11: for each $bindingNode \in bindingNodes$ do
12:    $bindingExpr := bindingNode.getChild(BindingSequenceNode).getChild(ExprNode)$
13:    $type := getType(bindingExpr)$
14:    if $type$ is PathType $\wedge$ $usesOnlyChildAndDescendantAxes(type) \wedge isWithoutPredicatesExceptLastStep(type)$ then
15:       $P := endsWithExactlyOnePredicate(type)$
16:       if $P \vee (checkJoinPattern3 \wedge isWithoutPredicates(type))$ then
17:          for each $(var, node) \in forVars$ do
18:             $forVars := determineJoinPattern(bindingNode, P, node, var, forVars, checkJoinPattern3, whereExpr)$
19:          end for
20:       end if
21:       if $isWithoutPredicates(type)$ then
22:          if $is(bindingNode,$ ForClauseNode$)$ then
23:             add $(bindingNode.varName, bindingNode)$ to $forVars$
24:          end if
25:       end if
26:    end if
27: end for
28: return $forVars$
```

**Algorithm 6.24** Function `determineJoinPattern`

**Input:**

$curBindingNode$: A variable binding node in the current FLWOR node.

$P$: A predicate from the binding expression.

$forBindingNode$: A for binding node from an ancestor FLWOR node.

$forVar$: A variable from the ancestor for binding node.

$checkJoinPattern3$: A flag determining whether to analyze the join pattern as possible join pattern 3.

$whereExpr$: An expression from a where clause of the current FLWOR node.

```
1: if $P$ is of form $forVar/L_1 = curBindingNode.varName/L_2 \wedge usesOnlyChildAndDescendantAndAttributeAxes(L_1) \wedge usesOnlyChildAndDescendantAndAttributeAxes(L_2)$ then
2:   if $is(curBindingNode$, ForClauseNode$)$ then
3:     memorize $(forBindingNode, curBindingNode)$ as an occurrence of the for join pattern
4:   else if $is(curBindingNode$, LetClauseNode$)$ then
5:     memorize $(forBindingNode, curBindingNode)$ as an occurrence of the for let pattern
6:   end if
7: end if
8: if $checkJoinPattern3 \wedge is(curBindingNode$, ForClauseNode$) \wedge usesOnlyChildAndDescendantAndAttributeAxes(L_1) \wedge usesOnlyChildAndDescendantAndAttributeAxes(L_2)$ then
9:   if $whereExpr$ is of form $forVar/L_1 = bindingNode.varName/L_2$ then
10:    memorize $(forBindingNode, curBindingNode)$ as an occurrence of the join pattern 3
11:  end if
12: end if
```

### 6.7.4 Analysis of the Join Pattern Occurrences

For each found join pattern occurrence, the algorithm displayed in Algorithm 6.25 and Algorithms 6.26, 6.27, 6.28 decides whether it is (O1) or (O2) case, using the rules from the original method and the rule for the join pattern 3 considering it of case (O1) and assigning it with weight of 0.5.

As it can be seen in Algorithm 6.25, rules R2 and R3 are applied using instances of PathType while, rules R4 and R5 use instances of `PathExprNode`. This difference is partially explained in definition of the auxiliary functions in Chapter 6.7.1. Rules R4 and R5 count target return paths. If they use instances of PathType, they can count one path more times (as is described in the mentioned chapter), and thus, give a wrong result.

### 6.7.5 Inference of Keys from Join Pattern Occurrences

Now, when the join patterns occurrences are classified into (O1) and (O2) cases, we infer the key statements according to the original approach [26].

Let $w$ be the weight assigned to a pattern occurrence $\pi$. If $\pi$ is marked as (O1), the following statements with weight $w$ are inferred:

- $(P_2, \{L_2\})$ is not satisfied
- $(P_1, \{L_1\})$ is satisfied
- $(P_2, \{L_2\}) \rightarrow (P_1, \{L_1\})$ is satisfied

If $\pi$ is marked as (O2), the following statements with weight $w$ are inferred:

- $(P_2, \{L_2\})$ is satisfied
- $(P_1, \{L_1\}) \rightarrow (P_2, \{L_2\})$ is satisfied

The original approach [26] describes also inference of relative keys. Refer to it for details.

### 6.7.6 Rejection of Uniqueness

While the previous steps of the key discovery produce mostly positive statements about the keys, in this step, the algorithm searches for evidence on non-uniqueness of elements and attributes. The aim of this step is to eliminate or decrease the number of falsely inferred keys. A key can be inferred falsely when the assumption that every join is done via a key/foreign key pair is not correct for a particular join.

**Algorithm 6.25** Classification of join pattern occurrences

```
Input: $\mathcal{O}$: A set of all found join pattern occurrences.
Output: $\mathcal{O}'$: A set of the classified join pattern occurrences from $\mathcal{O}$ assigned
    with their respective weights.
 1: for each $jpOccurrence \in \mathcal{O}$ do
 2:   if $jpOccurrence$ is the for join pattern then
 3:     mark $jpOccurrence$ as case (O1), weight 1
 4:   else if $jpOccurrence$ is the join pattern 3 then
 5:     mark $jpOccurrence$ as case (O1), weight 0.5
 6:   else
 7:     $secondBindingNode$ := the second binding node from $jpOccurrence$
 8:     $flworNode := secondBindingNode.getParent().getParent()$
 9:     $secondVarName := secondBindingNode.varName$
10:     $returnPathTypes := getReturnPathTypes(flworNode,$
        $secondVarName)$
11:     if $checkR2(returnPathTypes, jpOccurrence)$ then
12:       continue
13:     end if
14:     if $checkR3(returnPathTypes, jpOccurrence)$ then
15:       continue
16:     end if
17:     $targetReturnPaths := getTargetReturnPaths(flworNode,$
        $secontVarName)$
18:     $useR4R5(returnPaths, jpOccurrence)$
19:   end if
20:   update $jpOccurrence$ in $\mathcal{O}$
21: end for
22: return $\mathcal{O}$
```

**Algorithm 6.26** Function `checkR2`

```
Input:
    $returnPathTypes$: A list of target return path types.
    $jpOccurrence$: An occurrence of a join pattern to classify.
Output: True if the $jpOccurrence$ was classified as case (O1) using the R2 rule.
    False otherwise.
1: for each $returnPathType \in returnPathTypes$ do
2:    if $returnPathType.specialFunctionCalls$ contains one of 'min', 'max',
      'sum', 'avg' then
3:       mark $jpOccurrence$ as case (O1), weight 1 // R2
4:       return true
5:    end if
6: end for
7: return false
```

**Algorithm 6.27** Function `checkR3`

```
Input:
    $returnPathTypes$: A list of target return path types.
    $jpOccurrence$: An occurrence of a join pattern to classify.
Output: True if the $jpOccurrence$ was classified as case (O1) using the R3 rule.
    False otherwise.
1: for each $returnPathType \in returnPathTypes$ do
2:    if $returnPathType.specialFunctionCalls$ contains 'count' then
3:       mark $jpOccurrence$ as case (O1), weight 0.75 // R3
4:       return true
5:    end if
6: end for
7: return false
```

**Algorithm 6.28** Function `useR4R5`

```
Input:
    $targetReturnPaths$: A list of target return paths.
    $jpOccurrence$: An occurrence of a join pattern to classify.
1: $retPathsNumber$ := a number of paths in $targetReturnPaths$
2: if $retPathsNumber > 1$ then
3:    mark $jpOccurrence$ as case (O2), weight 1 // R4
4: else
5:    mark $jpOccurrence$ as case (O2), weight 0.5 // R5
6: end if
```

When the algorithm discovers that a certain element (or attribute) is probably not unique, it assigns the element (or attribute) with a negative weight that will decrease the sum of its weights in the summarizing step.

We can formally define negative uniqueness statements as follows:

**Definition 6.5** (Negative uniqueness Statement)**.** A negative uniqueness statement is a construct $(C, P)$, where $C$ and $P$ are PathType instances that use only child, descendant, and descendant or self ($P$ also attribute) axes. $C$ is called *context path*, $P$ *target path*. $C$ can be omitted as in the case of keys.

Note that the restrictions placed on the paths $C$ and $P$ are less restrictive than in the case of keys. If we find a negative uniqueness statement ($C$, $P$), where $P$ or $C$ contains predicates, then it implies that also $(C', P')$ is a negative uniqueness statement, where $C'$ and $P'$ are $C$ and $P$ stripped of predicates. In the case of special function calls, the explanation is the same.

The first XQuery construct utilized by this step is a function call of `distinct-values` function as discussed in Chapter 4.4.3 and shown in Algorithm 6.29. The algorithm searches for calls of the mentioned functions with a PathType on their input. Elements and attributes selected by those PathTypes are then considered to be not unique, and consequently, they cannot be keys.

Aggregation functions `min`, `max`, `sum` are not utilized by the function, because they were utilized by the search for join patterns, and in the process of classification of found join patterns, they were included in the computation of the key weights.

**Algorithm 6.29** Rejection of uniqueness - aggregation functions

**Input:** *node*: A node of the syntax tree.

```
1: if $is(node, \text{FunctionCallNode})$ then
2:    if $node.fncName$ = 'distinct-values' then
3:       $argument := node.getChild(0)$
4:       $type := getType(argument)$
5:       if $type$ is PathType
         $\wedge$ $usesOnlyChildAndDescendantAndAttributeAxes(type)$
         $\wedge$ $isWithoutPredicatesExceptLastStep(type)$ then
6:          memorize ($argument$) as the negative key statement, weight 1
7:       end if
8:    end if
9: end if
```

Besides the `distinct-values` function, a certain form of FLWOR expressions is utilized in this step. They are FLWORs that iterate through a sequence of nodes selected by a path expressions using only child, descendant, and descendant or self axes, and either that path expressions end with a predicate comparing a value of a subnode with some other value (literal constant, reference to a variable which is constant in this expressions, etc), or, the comparison is located in a where clause of a respective FLWOR.

As discussed in Chapter 4.4.3, when a for clause is used to iterate through a set of items, it is expected that the number of the items may be more than one. And, together with that, the set is restricted to contain only items which satisfy a condition that a value of one of their subnodes (or descendant nodes) equals a certain value. That implies that the value in those nodes is not unique, and, alike the utilization of the aggregation functions, non-unique nodes cannot be keys. This situation is handled in Algorithms 6.30 and 6.31.

**Algorithm 6.30** Rejection of uniqueness - comparison with a constant

**Input:** *node*: A node of the syntax tree.

```
1: if $is(node$, FLWORExprNode) then
2:    $forVars$ := an empty array
3:    for each $bindingNode \in node.getChild(TupleStreamNode).getChildren()$
      do
4:       if $is(bindingNode$, ForClauseNode) then
5:          $expr := bindingNode.getChild(BindingSequenceNode).$
            $getChild(ExprNode)$
6:          if $getType(expr)$ is PathType $\wedge$ $usesOnlyChildAndDescendantAxes($
            $getType(expr))$ then
7:             add $bindingNode.varName$ to $forVars$
8:          end if
9:       end if
10:   end for
11:   $whereClause := node.getChild(WhereClauseNode)$
12:   if $whereClause \neq$ null then
13:      $processWhere(forVars, whereClause.getChild(ExprNode))$
14:   end if
15: end if
```

In spite of the presented algorithms search for global negative uniqueness statements ($C$ path is omitted) only, we included the context path in the definition for possible future extensions.

**Algorithm 6.31** Rejection of uniqueness - function processWhere

**Input:**

$forVars$: Names of variables to look for in the where expression.

$exprNode$: The where expression node.

```
1: if $is(exprNode$, OperatorNode) then
2:     if $exprNode.operator$ = GEN_EQUALS then
3:         for each $var \in forVars$ do
4:             if $exprNode$ is of a form $var/L = C$ where $getType(var/L)$ is PathType $\wedge$ $isWithoutPredicatesExceptLastStep(getType(var/L))$ $\wedge$ $usesOnlyChildAndDescendantAndAttributeAxes(getType(var/L))$ $\wedge$ $C$ is a literal constant then
5:                 memorize $(var/L)$ as the negative key statement, weight 0.9
6:             end if
7:         end for
8:     else if $exprNode.operator$ = AND then
9:         $processWhere(forVars, exprNode.leftSide)$
10:        $processWhere(forVars, exprNode.rightSide)$
11:    end if
12: end if
```

### 6.7.7 Normalization of Key Statements

The original approach [26] employs the following normalization which we adopt. It also incorporates a heuristic for precision enhancement. For details, we refer to it.

Let $K_1, \ldots, K_n$ be the inferred keys. Let $S_i$ be the score (the sum of weights) of $K_i$ and $N_i$ be the number of the inferred statements about $K_i$.

Let $NU_1, \ldots, NU_m$ be the discovered negative uniqueness statements. Let $W_j$ be the weight of $NU_j$.

The negative uniqueness statements affect the scores as follows. For each negative uniqueness statement $NU_j = (C_j^{NU}, P_j^{NU})$, find $\mathcal{K}_j$ set of keys which are rejected by the statement.

$$\mathcal{K}_j = \left\{ K_i = (C_i^K, P_i^K, \{L_i^K\}) \middle| C_j^{NU} = C_i^K, P_j^{NU} = P_i^K / L_i^K \right\}.$$

Then, for each key $K_i \in \mathcal{K}_j$ decrement its score $S_i$ with weight of the negative uniqueness statement $W_j$.

Let $S^{max}$ be the maximum from $|S_1|, \ldots, |S_n|$ and $N^{max}$ be the maximum norm from $N_1, \ldots, N_n$. The normalized score $S_i$ of $K_i$ is computed as

$$S_i^{norm} = \frac{S_i}{S^{max}} * (1 - \frac{N^{max} - N_i}{\sum_{i=1}^{n} N_i})$$

### 6.7.8 Example

Figure 6.6 shows the syntax tree of the query in Listing B.9 (subtree of `ConstructorNode` is omitted because of lack of space). In this tree, the algorithm founds one occurrence of the join pattern 3. The two for clauses and one where clause which form the occurrence are shown in the red boxes.

- $P_1 =$ `/site/people/person`
- $P_2 =$ `/site/closed_auctions/closed_auction`
- $L_1 =$ `@id`
- $L_2 =$ `buyer/@person`

The next phase marks this occurrence as (O1) case with weight 0.5. Paths $P_1$ and $P_2$ does not have a common context, and thus, the following statements are inferred:

- Key $K = ($`/site/people/person`$, \{$`@id`$\})$ is satisfied.

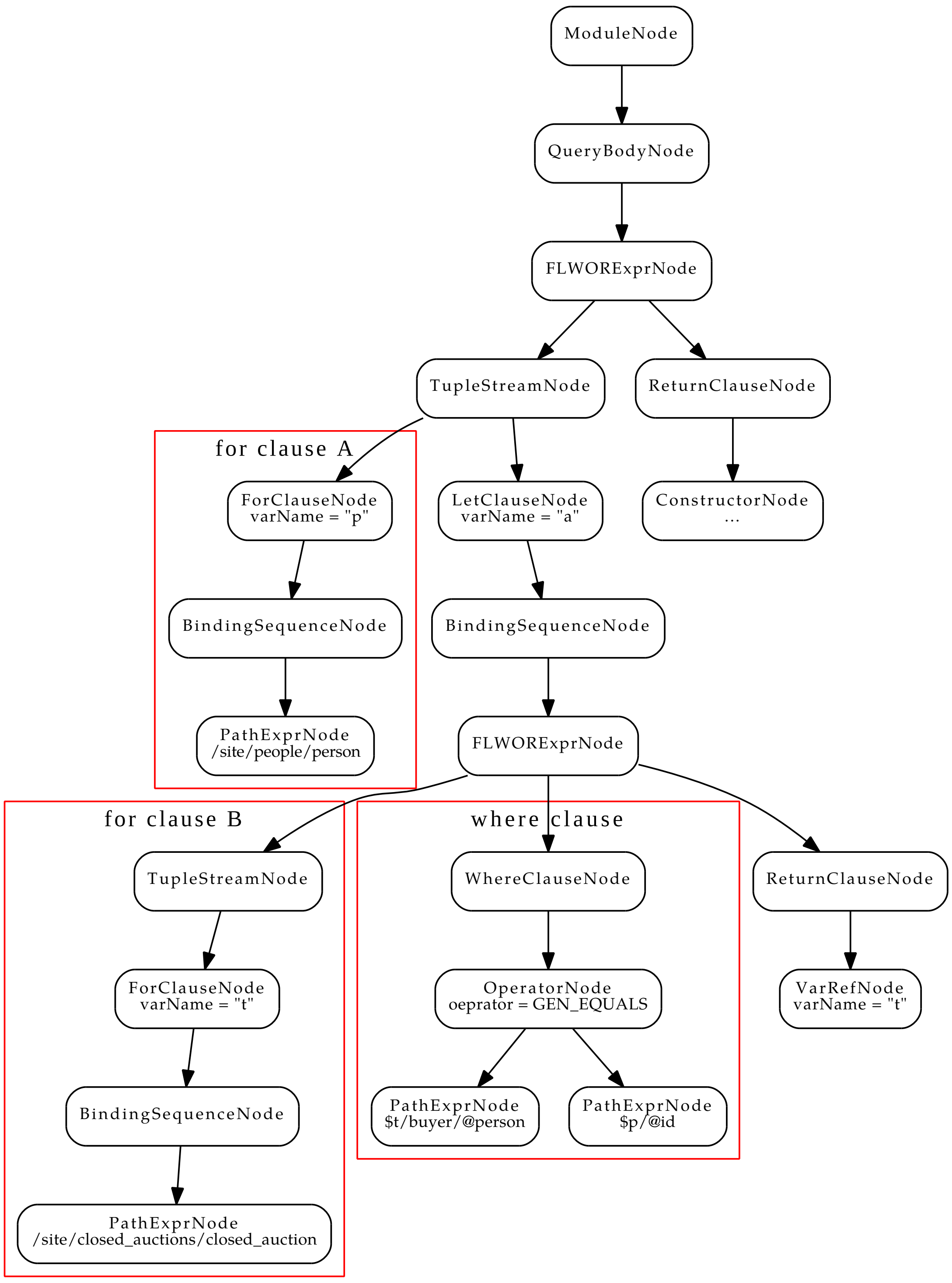


Figure 6.6: Sample syntax tree 2

- Foreign key ((`/site/closed_auctions/closed_auction`, {`buyer/@person`}) $\rightarrow$ $K$) is satisfied.
- Key (`/site/closed_auctions/closed_auction`, {`buyer/@person`}) is not satisfied.

# Chapter 7

# Combination with Existing Methods of Inference

In this chapter, we describe how to incorporate the inferred statements to existing methods of XML schema inference. We focus on a class of methods which are based on a creation and subsequent simplification of the initial grammar (as discussed in Chapter 3).

The initial grammar contains rules of form $e \to e_1 e_2 \dots e_k$, where $e$ is an element and $e_1, \dots, e_k$ are its subelements. After the simplification, the simplified grammar contain rules of form $e \to E$, where $E$ is a regular expression composed of subelements of element $e$, describing its content.

Attributes of elements are not contained in the grammar directly, but every element carries an information on its attributes.

The combination with such methods of inference is straightforward. The rules of the grammar describe the XML structure using the most general aspect of elements and their subelements. Since the statements inferred by our method do not involve the XML structure by defining a subelement structure of elements, there are no conflicts between the grammar rules and our statements that need to be resolved.

Our statements are of the three following forms. The first one is $P \to T$, where $P$ is a PathType representing an XQuery path selecting a set of elements or attributes, and $T$ is an XSD built-in atomic type.

The second and third forms are $K = (C, P, \{L\})$ and $K_f = (C(P_1, \{L_1\}) \to (P_2, \{L_2\}))$, representing a key and a foreign key.

In all three cases, we want to determine elements from the grammar (or their attributes), targeted by the respective paths ($P$ from the first form and $C$ from the second and third form). This is done in two steps. The first one is normalization of a particular PathType and the second one is selection of the targeted elements.

## 7.1 Evaluation of Paths

A path represented by PathType can contain variable references and association of these references with other paths. That is the reason why the normalization is convenient. It simply finds the variable references in the path and replaces them with steps from the associated paths, which are normalized recursively.

The selection of elements is a simplified XPath evaluation. The simplification involves two aspects; the evaluation is not performed upon XML data, but the simplified grammar containing rules with a regular expression on their right side, and, partially related, predicates in path steps are not evaluated, they are ignored.

It iterates through steps of a path, maintaining a so-called *context set*, which is a set of elements to evaluate the current step upon. The evaluation of one step is shown in Algorithm 7.1. For the retaining of readability of the code, we present an evaluation of self or descendant, child, and attribute axes, and nodes specified by name.

At first, the algorithm determines the axis of the step. If it is self or descendant axis, it returns the result of `evaluateStep_selfOrDescendant` function. This function returns the given context set extended by all descendants of the elements from the context set in the given grammar.

If the axis is child axis, for each element in the context set, the algorithm determines its subelement using the grammar, and, if those with the name equal to the name specified by to step are added to the resulting context set. Function `getTokens` at line 10 retrieves all elements from a regular expression. A specific form of the regular expression is not important, because we only need to know which elements are possible subelements of the particular element.

And, at last, if the axis is attribute axis, elements from the context set are searched to contain an attribute with the specified name and those found attributes are added to the result.

## 7.2 Saving the Inferred Statements

In case of keys, the algorithm iterates through the inferred key statements. A key's context path $C$ is evaluated and the key is assigned to the target elements selected by the context path, one element can be assigned with multiple keys. Additionally, each key is assigned with a list of foreign keys that are referencing

**Algorithm 7.1** Function evaluateStep

**Input:**

*step*: An instance of StepExprNode representing a step of the path to evaluate. *contextSet*: A set of elements and attributes to evaluate *step* upon. *grammar*: The grammar.

**Output:** A context set after the step evaluation.

```
1: if $is(step, \text{SelfOrDescendantStep})$ then
2:    return $evaluateStep\_selfOrDescendant(contextSet, grammar)$
3: end if
4: $newContextSet$ := an empty set
5: $axisKind := step.getChild(axisNode).axisKind$
6: $nodeName := step.getChild(axisNode).getChild(nameTestNode).name$
7: if $axisKind = \text{CHILD}$ then
8:    for each $node \in contextSet$ do
9:       if $node$ is element then
10:         $subelements := getTokens(grammar[node])$
11:         for each $subelement \in subelements$ do
12:            if $subelement.name = nodeName$ then
13:               add $subelement$ to $newContextSet$
14:            end if
15:         end for
16:      end if
17:   end for
18: else if $axisKind = \text{ATTRIBUTE}$ then
19:   for each $node \in contextSet$ do
20:      if $node$ is element then
21:         $attributes := node.attributes$
22:         for each $attribute \in attributes$ do
23:            if $attribute.name = nodeName$ then
24:               add $attribute$ to $newContextSet$
25:            end if
26:         end for
27:      end if
28:   end for
29: end if
30: return $newContextSet$
```

it.

In case of inferred types, the situation is slightly less simple, because there can occur conflicting statements. The examples of such conflicts for path $P$ without predicates are $P_1 \to$ `date`, $P_2 \to$ `string`, and, $P_1 \to$ `byte`, $P_2 \to$ `int`, where $P_1$, $P_2$ are paths that when stripped of predicates, they equal $P$.

Note that, in both examples, one type is castable to the other (`date` to `string`, and `byte` to `int`). Both types are inferred correctly for nodes targeted by $P$, but one of them was inferred from a more convenient expression and is more precise.

Consider these two expressions; PathType $P$ is compared to an integral literal constant, and, PathType $P$ is an argument of a function where a formal type of the argument is `byte`. The first expression is utilized to infer statement $P \to$ `integer`, and the second one to infer statement $P \to$ `byte`. Both of the types are correct, but the second one is more accurate.

A problem emerges for example if PathType $P$ is compared to an integral constant, and the real type of elements (or attributes) selected by $P$ is `double`. In that case, statement $P \to$ `integer` is inferred, but it is not correct.

### 7.2.1 Verification using XML data

To solve the problems, we propose a simple verification using XML data.

For each normalized PathType $P$ from the inferred type statements $\mathcal{S}_t$, we find set $\mathcal{T}_P$ of all inferred types. $\mathcal{T}_P = \{T | (P \to T) \in \mathcal{S}_t\}$. Then, we create sequence $\mathcal{T}'_P$ by ordering the set $\mathcal{T}_P$ from the most specific type to the most general one. For example, if $\mathcal{T}_P = \{$`double`, `byte`, `int`$\}$, $\mathcal{T}'_P = ($`byte`, `int`, `double`$)$.

Since we have the XML data and path $P$ is an XQuery path, we can use an XQuery processor (a program that evaluates XQuery paths or queries) to select nodes $N$ targeted by $P$. The verification algorithm iterates through $\mathcal{T}'_P$ and for each $T' \in \mathcal{T}'_P$ it checks if every node in $N$ conforms to $T'$. If so, $T'$ is the inferred type for nodes $N$ (and PathType $P$), else the inferred type is `string`.

# Chapter 8

# Implementation

The solution proposed in Chapters 6 and 7, except the verification of types, was implemented using the jInfer framework [19]. It is a framework for implementing methods of XML schema inference created as a software project at the Faculty of Mathematics and Physics, Charles University in Prague. It is written in Java as a plugin for NetBeans platform.

The framework consists of modules representing logical parts of a process of inference. The main idea behind the modules is that they can be replaced by other modules with the same interface but different implementation and new modules can be connected to extend functionality.

## 8.1 jInfer Process of Inference

Figure 8.1 shows the process of inference in jInfer. The grey and yellow rectangles represent steps of the inference, the white boxes represent input and output. Originally, the inference was composed of **Initial Grammar Generator**, **Simplifier**, and **Schema Generator** steps (grey). Steps (yellow) **XQAnalyzer**, **XQuery Processor**, and **Merger** are implementations of main parts of our proposed solution.

- **XQAnalyzer** - An implementation of the lexical and syntax analyses proposed in [29] and modified to create syntax trees. It parses input XQuery files and outputs their syntax trees.

- **XQuery Processor** - An implementation of algorithms proposed in Chapter 6. In particular, construction of a syntax tree, static analysis of expression types, inference of built-in types, and key discovery. On input, it takes

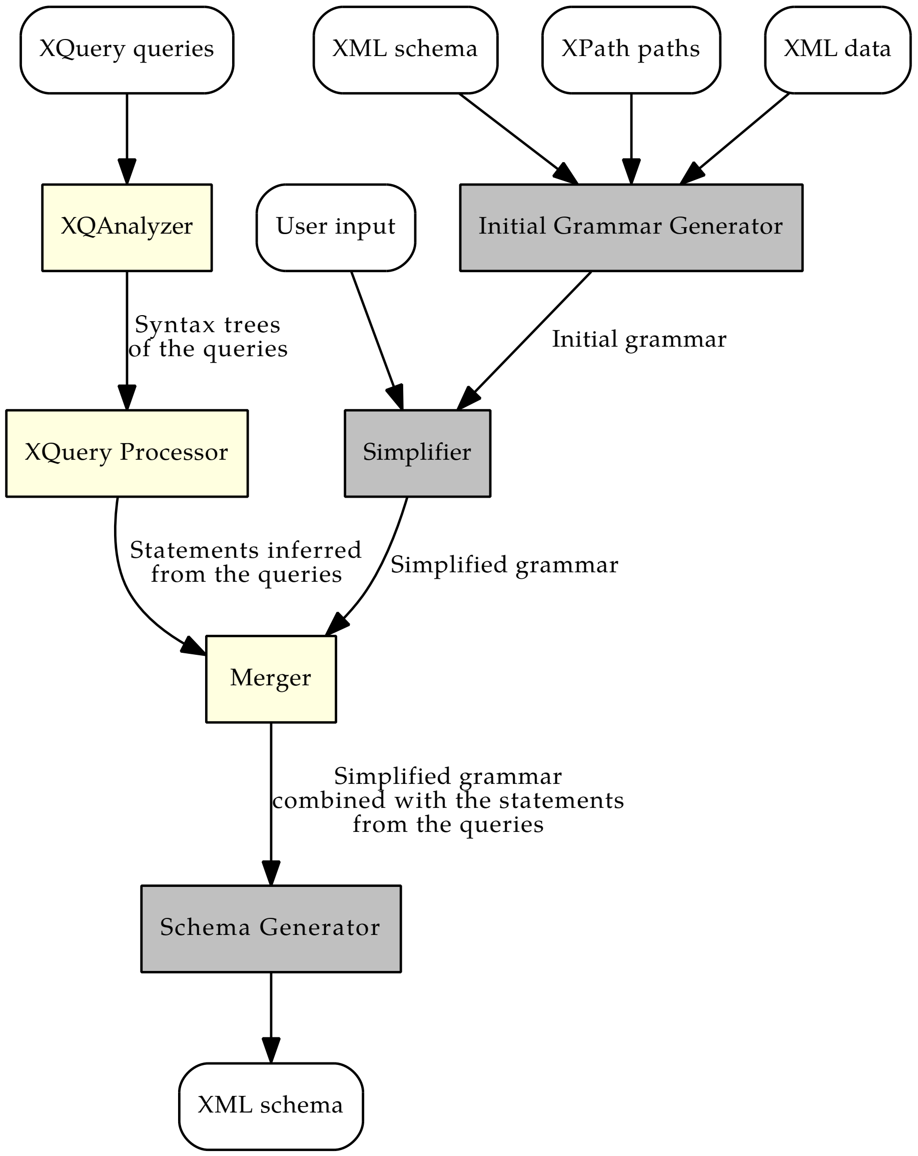


Figure 8.1: jInfer process of inference

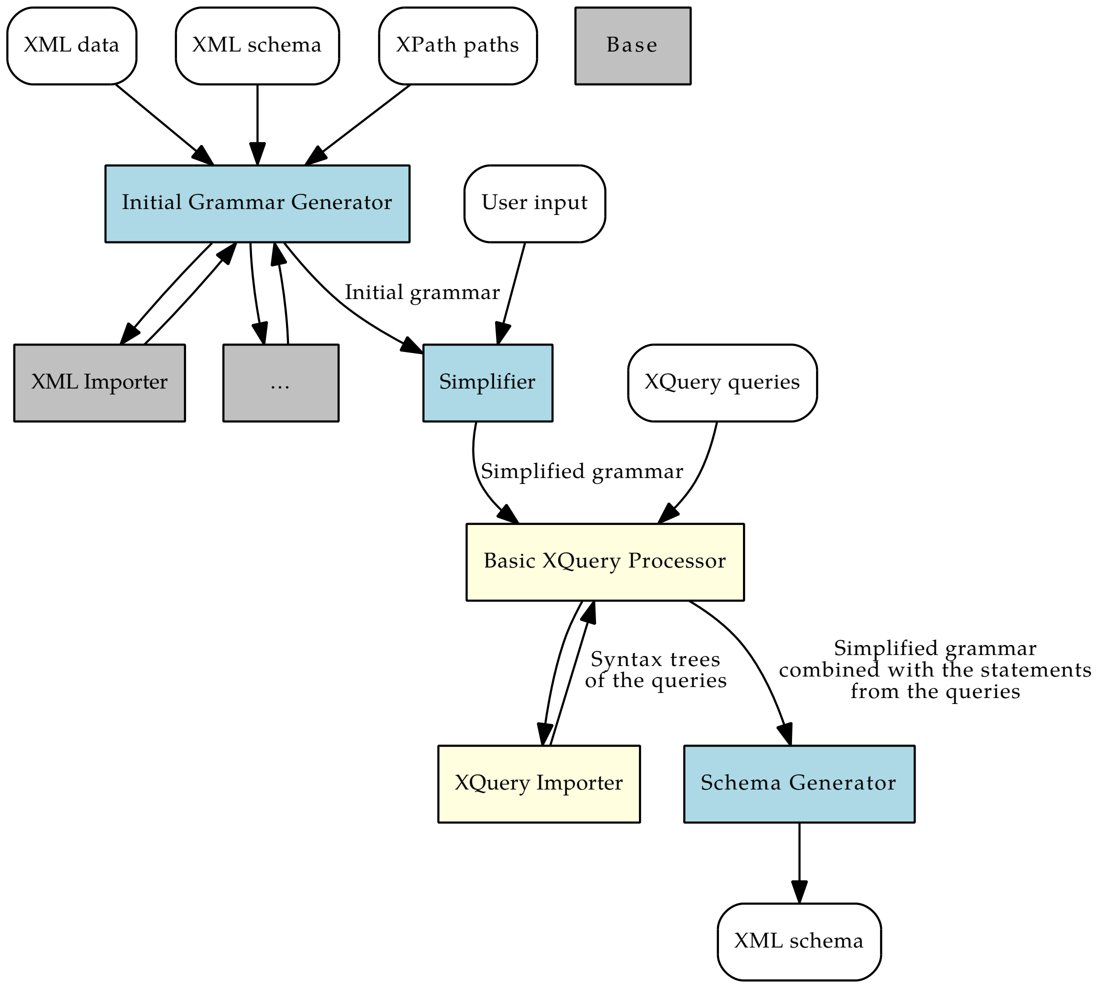


Figure 8.2: jInfer modules

the syntax trees, and its output are statements inferred from the syntax tress.

- **Merger** - A step of inference responsible for combining the simplified grammar with the statements inferred by **XQuery Processor** as described in Chapter 7, except the verification of types. Its result is the simplified grammar extended by the statements from **XQuery Processor** in a way that each statement is analysed and its information is assigned to concerned elements and attributes from the grammar.

## 8.2 jInfer Modules

The previous section describes a high level schematic view of the process of inference. But, from a more technical view, jInfer modules do not utterly correspond with the steps of the presented process of inference.

The first difference is that none of the modules runs in parallel (in several threads) with other, as it is schematically shown in Figure 8.1. The modules run in a serial order as shown in Figure 8.2. The second difference is that one step is not always represented by one module, or one module is not always representing just one step.

Actually, the blue modules shown in Figure 8.2 are module abstractions with specified interface. Actual modules are then implementations complying the interface. Since we provide only one implementation of modules for the proposed solution, this is not very important for this work and we do not describe the principle in detail. We only note that **Basic XQuery Processor** module is an implementation of **Non-grammatical Input Processor** module abstraction.

As in the previous section, newly added modules are those shown in yellow boxes. They are **Basic XQuery Processor** and **XQuery Importer**. We also modified modules **Basic XSD Exporter** (an implementation of **Schema Generator** interface) and **Base** to extend their functionality.

- **Base** - This is module containing common classes used by other modules and defining interfaces. We added five packages to this module. Package `cz.cuni.mff.ksi.jinfer.base.objects.xquery.syntaxtree.nodes` implements the structure of syntax trees. Package `cz.cuni.mff.ksi.jinfer-.base.interfaces.xquery` contains `Type` interface which is an interface for types used in the type analysis of syntax trees. Package `cz.cuni.mff.ksi-.jinfer.base.objects.xquery.types` contains implementations of `Type` interface and type utility classes. Package `cz.cuni.mff.ksi.jinfer.base-.objects.xquery.keys` provides representations of keys and foreign keys, and package `cz.cuni.mff.ksi.jinfer.base.objects.xsd` provides representation of XSD built-in atomic types.

- **XQuery Importer** - This module represents **XQAnalyzer** step in Figure 8.1. It is responsible for creating syntax trees from input XQuery files (step 1).

- **Basic XQuery Processor** - The main module consisting of steps **XQuery Processor** and **Merger** in Figure 8.1. Package `cz.cuni.mff.ksi.jinfer-.basicxqueryprocessor` contains the main module class `NonGrammatical-InputProcessorImpl`. Other packages are `cz.cuni.mff.ksi.jinfer.-basicxqueryprocessor.expressiontypesanalysis` implementing the static analysis of expression types (step 2), `cz.cuni.mff.ksi.jinfer.basic-xqueryprocessor.builtintypeinference` implementing the inference of

XSD built-in types (step 3), `cz.cuni.mff.ksi.jinfer.basicxquery-processor.keydiscovery` and their subpackages implementing the key discovery (step 4), `cz.cuni.mff.ksi.jinfer.basicxqueryprocessor.merger` implementing the merging with the grammar, and `cz.cuni.mff.ksi.jinfer-.basicxqueryprocessor.utils` containing various utilities.

- **Basic XSD Exporter** - This is the module responsible for generating the resulting schema in XSD. It was modified to process the additional information assigned to the grammar in **Basic XQuery Processor** module.

# Chapter 9

# Experiments

In this chapter we describe how we performed experiments with the implementation and what problems we faced.

We made two test scenarios; one dealing with input data that were not made for purposes of this experiments, and the other extending the first dataset by data created for better test coverage.

## 9.1 Test Scenario A

### 9.1.1 Test Data

To get meaningful results of the experiments, test data should be composed of XML documents, which are instances of a certain, possibly not known, XML schema, and a set of XQuery queries which query the XML documents. The amount of the XML data does not have to be large. On the other hand, the set of queries should be large (at least hundreds of queries) and the queries should be real, not artificially made.

In a search for such test data, we have not succeeded. Large sets of XML data are available, but large sets of XQuery queries are not or it is not a simple task to obtain them.

If we cannot obtain an ideal set of test data, we can at least try to find the most suitable one from available non-ideal sets.

Sets of XML data and XQuery queries can be found in W3C XML Query Use Cases [21]. However, those are very small sets of queries and the analysis of XQuery in Chapter 4 was worked out using those queries, and thus, the relevancy of such test data is questionable.

Another considered possibility was to obtain some set of XML data and create

| |
|---|
| `/site/open_auctions/open_auction/bidder/increase/text()` → `integer` |
| `/site/closed_auctions/closed_auction/price/text()` → `integer` |
| `/site/open_auctions/open_auction/initial/text()` → `integer` |
| `/site/open_auctions/open_auction/initial/text()` → `integer` |
| `/site/people/person/profile/@income` → `integer` |
| `/site/open_auctions/open_auction/reserve` → `decimal` |

Table 9.1: Inferred type statements A

queries to it. This notion was rejected because such set would have all of the negative characteristics; it would be small, artificially made, and it would not be independent, as well.

At last, we concluded to use data provided by the XMark project [1]. They are attached in Appendix B and they consists of automatically generated XML data and a set of twenty XQuery queries related to the data. Although, this set is also very small, it is more or less real and we did not known the set in the process of developing the algorithm.

## 9.1.2 Results

### Type Inference

Six type statements shown in Table 9.1 were inferred.

Only the last inferred statement is correct. Others are incorrect, because, comparing to the data, real type of nodes selected by the paths is `decimal`, as well.

To reveal the cause of the incorrect type inference, see, for example, query in Listing B.13. In the where clause, values of `/site/people/person/profile/-@income` are compared to the integer literal constant `50000`. From this expression, it is not possible to infer the type correctly. The problem is that this is the only inferred statement. Better results can be achieved by providing a larger set of input queries, containing also expressions that can be exploited to infer correct statements. Then the verification with data can be incorporated to choose the correct statements.

### Key Discovery

Four key statements and their normalized weights shown in Table 9.2 were inferred.

| **Key** | **Weight** |
|---|---|
| `(/site/closed_auctions/closed_auction, {buyer/@person})` | -1.0 |
| `(/site/regions/europe/item, {@id})` | -0.417 |
| `(/site/people/person, {@id})` | 1.0 |
| `(/site/closed_auctions/closed_auction, {itemref/@item})` | 0.417 |

Table 9.2: Inferred key statements A

The first one is correct, a buyer is not a key of closed auctions. The second one is not correct, because `id` attribute is a key of `item` elements. The third one is correct and the fourth one declares `itemref` element to be a key of closed auctions, which is not true, but only with weight 0.417.

Closer analysis of input queries reveals that all of the statements were inferred from occurrences of the join pattern 3. That knowledge leads to the two following observations.

- The original method of key discovery would not infer anything on this input data.
- The cause of the incorrectly inferred statements are not that they were inferred from join patterns occurrences, where a join is not done by a key/-foreign key pair. It is that the for clauses (definitions of paths $P_1$ and $P_2$) in the join pattern 3 occurrence (in query in Listing B.10) are swapped, so the real key is considered as a foreign key and vice-versa.

A partial solution is an extension of the test data by queries containing expression that can be exploited to infer negative uniqueness statements. Such expression is, for example,
`distinct-values(/site/closed_auctions/closed_auction/itemref/@item)`
to get unique ids of items that was sold in some auction. Since, one item may be sold several times in auctions organized in different time periods, `distinct-values` function is applied.

From the original data, one negative uniqueness statement was inferred:
`/site/people/person/profile/interest/@category` is not unique with weight 1, and it it correct. Since, there is not such key statement inferred, the negative uniqueness statement is not used to any correction of weight.

From data extended by the mentioned expression, another negative uniqueness was inferred:
`/site/closed_auctions/closed_auction/itemref/@item` with weight 1. This

| **Key** | **Weight** |
|---|---|
| `(/site/closed_auctions/closed_auction, {buyer/@person})` | -1.0 |
| `(/site/regions/europe/item, {@id})` | -0.417 |
| `(/site/people/person, {@id})` | 1.0 |
| `(/site/closed_auctions/closed_auction, {itemref/@item})` | -0.417 |

Table 9.3: Key statements inferred from the extended test data A

| **Key** | **Weight** |
|---|---|
| `(/site/people/person, {@id})` | 1.0 |
| `(/site/closed_auctions/closed_auction, {itemref/@item})` | -0.333 |

Table 9.4: Key statements inferred from the extended test data using the modified JP3 statements

one decreases the weight of the falsely inferred key. Key statements inferred from the extended data are shown in Table 9.3.

As was demonstrated, larger sets of input data may lead to better results. However, the problem with the falsely rejected key still remains. It may be solved by modification of the statements inferred from occurrences of the join pattern 3. If we omit the negative statement of a key from the second for clause ($(P_2, \{L_2\})$ is not satisfied), we get better results. The modified statements inferred from an occurrence of the join pattern 3 are the following (assigned weight remains 0.5):

- $(P_1, \{L_1\})$ is satisfied
- $(P_2, \{L_2\}) \rightarrow (P_1, \{L_1\})$ is satisfied

From the extended test data, using the modified join pattern 3 statements, key statements in Table 9.4 were inferred. Those are the best results from all test runs, though it is not clear that the modified statements will produce the best results also on other larger input data. Therefore, further tests with large sets of queries are required to determine the best settings for the algorithm.

## 9.2 Test Scenario B

As mentioned before, the previous test scenario involves only data with JP3 occurrences, and therefore, it only tests the extension of the original method, while the original method itself remains untested. To correct it, we add several new test queries created by ourselves.

### 9.2.1 Test Data

```
for $item in /site/regions/europe/item
let $closed_auctions := /site/closed_auctions/closed_auction
   [itemref/@item = $item/@id]
return <item><id>{$item/@id}</id><max-price>{max(
   $closed_auctions/price)}</max-price></item>
```

Listing 9.1: Test query B1 containing a for join pattern occurrence.

```
for $open_auction in /site/open_auctions/open_auction
let $item := /site/regions/europe/item[@id = $open_auction/
   itemref/@item]
return $item/personref/@person
```

Listing 9.2: Test query B2 containing a let join pattern occurrence.

```
 1 for $person in /site/people/person
 2 where $person/profile/@income < 100.00 and $person/profile/
      gender = "m"
 3 return
 4     <list>{
 5         for $auction in /site/closed_auctions/closed_auction
 6         let $item := /site//item[@id = $auction/itemref/
              @item]
 7         let $price := $auction/price
 8         where $auction/buyer/@person = $person/@id
 9         return
10             <record><person>{$person/@id}</person><item>{
                  $item/@id}</item><price>{$price}</price></
                  record>
11     }</list>
```

Listing 9.3: Test query B3 containing let join pattern and JP3 occurrences.

In this scenario, we use test data from the Test Scenario A extended by queries in Listings 9.1, 9.2, and 9.3.

The first two ones are simple queries containing one join pattern occurrence each. The third one contains occurrences of let join pattern (binding clauses at lines 5 and 6) and JP3 (binding clauses at lines 1 and 5, and where clause at line 8), two expressions exploitable by the inference of types (both at line 2) and one negative uniqueness statement of the comparison-with-a-constant form

| |
|---|
| `/site/open_auctions/open_auction/bidder/increase/text()` → `integer` |
| `/site/closed_auctions/closed_auction/price/text()` → `integer` |
| `/site/open_auctions/open_auction/initial/text()` → `integer` |
| `/site/open_auctions/open_auction/initial/text()` → `integer` |
| `/site/people/person/profile/@income` → `integer` |
| `/site/open_auctions/open_auction/reserve` → `decimal` |
| `/site/people/person/profile/@income` → `decimal` |
| `/site/people/person/profile/gender` → `string` |

Table 9.5: Inferred type statements B

| **Key** | **Weight** |
|---|---|
| `(/site//item, @id)` | 0.333 |
| `(/site/people/person, @id)` | 0.6 |
| `(/site/closed_auctions/closed_auction, buyer/@person)` | -0.6 |
| `(/site/closed_auctions/closed_auction, itemref/@item)` | -0.917 |
| `(/site/regions/europe/item, @id)` | 0.6 |

Table 9.6: Inferred key statements B

(comparison with `"m"` at line 2). Thus, the third query contains instances of all types of the constructs utilized by our method.

### Results

Results of this test scenario are shown in Tables 9.5, 9.6, and 9.7. All seem to be as we expected.

Note that all inferred type statements are correct, however, the weight of the `(/site//item, @id)` key is only 0.333. The reason is that there is only one join pattern occurrence resulting to this key, and therefore in summary, its normalized weight is low and it is correct.

We demonstrated that also the original method works and we can get better results with little larger input dataset. Though, this set is still very small and we

| **Node** | **Weight** |
|---|---|
| `(/site/people/person/profile/interest/@category)` | 1 |
| `(/site/closed_auctions/closed_auction/itemref/@item)` | 1 |
| `(/site/people/person/profile/gender)` | 0.9 |

Table 9.7: Inferred negative uniqueness statements B

do not show how the methods work with large real-world data, possibly containing FLWOR constructs not satisfying our assumption that each join is done by a key/foreign key pair.

We attach the resulting XSD in Appendix A.1. It was made using threshold 0.3. It means that all key statements with normalized weight equal or higher than 0.3 are considered correct and are included in the schema.

# Chapter 10

# Conclusion

The aim of this thesis was to employ XML operations in the XML schema inference process. We analyzed several existing methods of the XML schema inference and we searched for methods that utilize selected XML operations. We found only one method, inferring keys from XQuery queries.

Since the notion of XML operations is very general and the range of XML technologies is very large, for the purpose of this work, we decided to focus on XQuery technology. We made the overview of possible utilization of XQuery queries in the process of XML schema inference.

Before creation of the algorithm itself, we had to take several decisions in questions that emerged. Since there is a lack of the methods dealing with the utilisation of XML operation, there is also a lack of practically proven solutions that can help in such decision making.

In the proposed solution, we decided to incorporate lexical and syntax analyses of XQuery queries, because it is more general and more extensible than a pattern searching. To achieve that, we adopted the algorithm from a recent master thesis dealing with an analysis of XQuery queries.

We also implemented several ideas from the overview to infer XSD built-in types of elements and attributes. And we extended and implemented the one existing method dealing with the inference of keys. We experimentally demonstrated that on some input files, results of the extended method are better than results of the original method. However, we did not succeed in the search of an ideal, large enough set of test data. The testing was performed on a small set of input queries and further testing and algorithm tuning is required.

Finally, we proposed a simple way how to combine the inferred statements with existing methods of the XML schema inference inferring initial grammar and we implemented it using the jInfer framework. Thus, we created the first complete, ready-to-use, and extensible implementation of the XML schema inference exploiting XQuery queries besides XML data.

As the main advantages and disadvantages of the work we list the following.

+ Elaboration of the overview how XQuery queries can help in the process of XML schema inference.
+ Incorporation of the lexical and syntax analyses of input queries creating their syntax trees.
+ Development of the static type analysis including PathTypes, which can be easily extended and used in possible future extensions.
+ Extension of the existing method of key discovery, achieving better results.
+ Our implementation using the jInfer framework is the first implementation of XML schema inference method utilizing XML operations.

– In the proposed algorithm we dealt with only a small part of possible refinements discussed in the overview. The reason for this is that inclusion of more of them would exceed scope of one thesis.
– The combination with existing methods of inference is done in the simplest possible way - by not modifying the grammar rules. However, in the view of recent XML schema inference research, the modification of the grammar rules according to the statements inferred from XML operations is of considerable interest. The reason for the simplest combination is related with the previous point and it is that we did not deal with refinements affecting the grammar rules.
– Since large real-world sets of XQuery queries are not available, we did not perform tests using such sets. We only tested our method using a relatively small number of input queries, decreasing the level of reliability of the experimental results.

## 10.1 Future Work

As already mentioned, this work implements only some ideas discussed in the overview in Chapter 4, leaving most of them for future work. Also, the implemented algorithms can be further refined as was already mentioned in Chapter

6 with the presentation of the algorithms. For example thorough processing of user-defined function calls.

Chapter 5 discusses some possible future enhancements as well. A research on possibilities of modifying the grammar rules based on information extracted from XQuery queries may bring interesting results. We analyse the queries statically only, we do not evaluate them. An analysis of queries together with their results can be a topic of another future research direction.

The utilization of XQuery queries certainly provides a space for incorporating interaction with user. For example, a user may influence the scoring of inferred keys to get more precise results.

And, a very large space for a possible future research is provided by utilisation of other XML operations. The main representant is XSLT.

Besides the mentioned, our opinion is that the most urgent future work is obtaining a large enough test data set, performing proper experiments, and refining the algorithm by modification of its settings (statements inferred from join pattern occurrences, weights, etc.) according to experimental results. This process was suggested in Chapter 9.

# Bibliography

[1] Xmark - an xml benchmark project. Website of the project: http://www.xml-benchmark.org/.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2 edition, September 2006.

[3] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4, 1996.

[4] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise dtds from xml data. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 115–126. VLDB Endowment, 2006.

[5] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring xml schema definitions from xml data. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 998–1009. VLDB Endowment, 2007.

[6] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[7] Boris Chidlovskii. Schema extraction from xml collections. In *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, JCDL '02, pages 291–292, New York, NY, USA, 2002. ACM.

[8] James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[9] James Clark and Steven DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[10] James Clark and Murata Makoto. RELAX NG Specification. OASIS Committee Specification and ISO/IEC 19757-2, December 2001.

[11] Mary F. Fernández, Daniela Florescu, Scott Boag, Jonathan Robie, Don Chamberlin, and Jérôme Siméon. XQuery 1.0: An XML query language (second edition). W3C proposed edited recommendation, W3C, April 2009. http://www.w3.org/TR/2009/PER-xquery-20090421/.

[12] Henning Fernau. Learning xml grammars. In *Proceedings of the Second International Workshop on Machine Learning and Data Mining in Pattern Recognition*, MLDM '01, pages 73–87, London, UK, 2001. Springer-Verlag.

[13] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. Xtract: a system for extracting document type descriptors from xml documents. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 165–176, New York, NY, USA, 2000. ACM.

[14] Peter Grünwald. A tutorial introduction to the minimum description length principle. In *Advances in Minimum Description Length: Theory and Applications*. MIT Press, 2005.

[15] Jan Hegewald, Felix Naumann, and Melanie Weis. Xstruct: Efficient schema extraction from multiple and large xml documents. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, pages 81–, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Rick Jelliffe. Academia Sinica Computing Centre's Schematron Home Page, 2001.

[17] Michal Klempa. Optimization and refinement of xml schema inference approaches. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2011.

[18] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer tutorial. `http://jinfer.sourceforge.net/doc_tutorial.html`.

[19] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer xml schema inference framework. `http://jinfer.sourceforge.net/modules/paper.pdf`. Website of the project: `http://jinfer.sourceforge.net`.

[20] Ashok Malhotra and Paul V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[21] Massimo Marchiori, Don Chamberlin, Peter Fankhauser, Jonathan Robie, and Daniela Florescu. XML query use cases. W3C note, W3C, March 2007. http://www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323/.

[22] Jun-Ki Min, Jae-Yong Ahn, and Chin-Wan Chung. Efficient extraction of schemas for xml documents. *Inf. Process. Lett.*, 85:7–12, January 2003.

[23] Irena Mlynková. An analysis of approaches to xml schema inference. In *Proceedings of the 2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*, SITIS '08, pages 16–23, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Irena Mlýnková. On inference of xml schema with the knowledge of an obsolete one. In *Proceedings of the Twentieth Australasian Conference on Australasian Database - Volume 92*, ADC '09, pages 77–84, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[25] Chuang-Hue Moh, Ee-Peng Lim, and Wee-Keong Ng. Re-engineering structures from web documents. In *Proceedings of the fifth ACM conference on Digital libraries*, DL '00, pages 67–76, New York, NY, USA, 2000. ACM.

[26] Martin Nečaský and Irena Mlýnková. Discovering xml keys and foreign keys in queries. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 632–638, New York, NY, USA, 2009. ACM.

[27] Theodore Norvell. A short introduction to regular expressions and context free grammars. `http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf`.

[28] Anand Raman, Jon Patrick, and Palmerston North. The sk-strings method for inferring pfsa. In *In Proceedings of the*, 1997.

[29] Jiří Schejbal. A system for analysis of collections of xml queries. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2010.

[30] Henry S. Thompson, Murray Maloney, David Beech, and Noah Mendelsohn. XML schema part 1: Structures second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[31] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *Proceedings of the 13th international conference on Database systems for advanced applications*, DASFAA'08, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] Julie Vyhnanovská. Automatic construction of an xml schema for a given set of xml documents. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2009.

[33] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

[34] Priscilla Walmsley and David C. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[35] Raymond K. Wong and Jason Sankey. On structural inference for xml data. Technical report, 2003.

# Appendix A

# Attachments

Logical: AND, OR.
Comparison: GEN_EQUALS, GEN_NOT_EQUALS, GEN_LESS_THAN, GEN_LESS_THAN_EQUALS, GEN_GREATER_THAN, GEN_GREATER_THAN_EQUALS, VAL_EQUALS, VAL_NOT_EQUALS, VAL_LESS_THAN, VAL_LESS_THAN_EQUALS, VAL_GREATER_THAN, VAL_GREATER_THAN_EQUALS, NOD_IS, NOD_PRECEDES, NOD_FOLLOWS.
Range: TO.
Additive: PLUS, MINUS.
Multiplicative: MUL, DIV, IDIV, MOD.
Set: UNION, INTERSECTION, DIFFERENCE.
Type test: INSTANCE_OF, CASTABLE_AS.
Type conversion: TREAT_AS, CAST_AS.
Unary: UNARY_PLUS, UNARY_MINUS.

Figure A.1: All possible values representing an operator in an instance of OperatorNode

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- Inferred on Sat Mar 31 21:28:32 CEST 2012 by Basic IG
   generator, TwoStep(Iname (with attributes), Automaton
   Merging State(GreedyMDL(Combined(k,h-context, s,k-strings
   , Null, Null),Naive Alphabet), State Removal Ordered(
   Weighted)), Chained(Empty Children, Nested Concatenation,
    Null)), Basic XSD exporter -->

<!-- global types -->
```

```
<xs:complexType name="Temph" mixed="true">
  <xs:sequence>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:choice>
        <xs:element name="keyword" type="Tkeyword"/>
        <xs:element name="bold" type="Tbold"/>
      </xs:choice>
    </xs:sequence>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tbold" mixed="true">
  <xs:sequence>
    <xs:choice>
      <xs:choice>
        <xs:sequence>
          <xs:element name="emph" type="Temph"/>
        </xs:sequence>
      </xs:choice>
      <xs:sequence>
        <xs:element name="keyword" type="Tkeyword"/>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tkeyword" mixed="true">
  <xs:sequence>
    <xs:choice>
      <xs:sequence minOccurs="0">
        <xs:element name="bold" type="Tbold"/>
      </xs:sequence>
      <xs:sequence>
        <xs:element name="emph" type="Temph"/>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Ttext" mixed="true">
```

```
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="bold" type="Tbold"/>
    <xs:element name="keyword" type="Tkeyword"/>
    <xs:element name="emph" type="Temph"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="Tparlist">
  <xs:sequence>
    <xs:element name="listitem" type="Tlistitem" minOccurs
        ="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tlistitem">
  <xs:choice>
    <xs:element name="text" type="Ttext"/>
    <xs:element name="parlist" type="Tparlist"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="Tdescription">
  <xs:choice>
    <xs:element name="parlist" type="Tparlist"/>
    <xs:element name="text" type="Ttext"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="Tincategory">
  <xs:attribute name="category" type="xs:string" use="
     required"/>
</xs:complexType>

<xs:complexType name="Tmail">
  <xs:sequence>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="to" type="xs:string"/>
    <xs:element name="date" type="xs:string"/>
    <xs:element name="text" type="Ttext"/>
  </xs:sequence>
```

```
</xs:complexType>

<xs:complexType name="Tmailbox">
  <xs:sequence minOccurs="0">
    <xs:element name="mail" type="Tmail"/>
    <xs:element name="mail" type="Tmail" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Titem">
  <xs:sequence>
    <xs:element name="location" type="xs:string"/>
    <xs:element name="quantity" type="xs:string"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="payment" type="xs:string"/>
    <xs:element name="description" type="Tdescription"/>
    <xs:element name="shipping" type="xs:string"/>
    <xs:element name="incategory" type="Tincategory"
       minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="mailbox" type="Tmailbox"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tafrica">
  <xs:sequence>
    <xs:element name="item" type="Titem"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tasia">
  <xs:sequence>
    <xs:element name="item" type="Titem"/>
    <xs:element name="item" type="Titem"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Taustralia">
  <xs:sequence>
    <xs:element name="item" type="Titem"/>
    <xs:element name="item" type="Titem"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Teurope">
  <xs:sequence>
    <xs:element name="item" type="Titem" minOccurs="0"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tnamerica">
  <xs:sequence>
    <xs:element name="item" type="Titem" minOccurs="0"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tsamerica">
  <xs:sequence>
    <xs:element name="item" type="Titem"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tregions">
  <xs:sequence>
    <xs:element name="africa" type="Tafrica"/>
    <xs:element name="asia" type="Tasia"/>
    <xs:element name="australia" type="Taustralia"/>
    <xs:element name="europe" type="Teurope"/>
    <xs:element name="namerica" type="Tnamerica"/>
    <xs:element name="samerica" type="Tsamerica"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tcategory">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="description" type="Tdescription"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tcategories">
  <xs:sequence>
    <xs:element name="category" type="Tcategory"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tedge">
  <xs:attribute name="from" type="xs:string" use="required
     "/>
  <xs:attribute name="to" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tcatgraph">
  <xs:sequence>
    <xs:element name="edge" type="Tedge"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Taddress">
  <xs:sequence>
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
    <xs:choice>
      <xs:element name="zipcode" type="xs:string"/>
      <xs:sequence>
        <xs:element name="province" type="xs:string"/>
        <xs:element name="zipcode" type="xs:string"/>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tinterest">
  <xs:attribute name="category" type="xs:string" use="
     required"/>
</xs:complexType>
```

```
<xs:complexType name="Tprofile">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="education" type="xs:string"/>
      <xs:element name="interest" type="Tinterest"/>
    </xs:choice>
    <xs:choice>
      <xs:element name="business" type="xs:string"/>
      <xs:sequence>
        <xs:element name="gender" type="xs:string"/>
        <xs:element name="business" type="xs:string"/>
      </xs:sequence>
    </xs:choice>
    <xs:element name="age" type="xs:string" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="income" type="xs:integer" use="
     required"/>
</xs:complexType>

<xs:complexType name="Twatch">
  <xs:attribute name="open_auction" type="xs:string" use="
     required"/>
</xs:complexType>

<xs:complexType name="Twatches">
  <xs:sequence>
    <xs:element name="watch" type="Twatch" minOccurs="0"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tperson">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="emailaddress" type="xs:string"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="address" type="Taddress"/>
      <xs:element name="phone" type="xs:string"/>
      <xs:element name="homepage" type="xs:string"/>
      <xs:element name="creditcard" type="xs:string"/>
    </xs:choice>
    <xs:choice>
      <xs:sequence minOccurs="0">
        <xs:element name="profile" type="Tprofile"/>
        <xs:element name="watches" type="Twatches" minOccurs
           ="0"/>
      </xs:sequence>
      <xs:element name="watches" type="Twatches"/>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Tpeople">
  <xs:sequence>
    <xs:element name="person" type="Tperson" minOccurs="0"
       maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tpersonref">
  <xs:attribute name="person" type="xs:string" use="required
     "/>
</xs:complexType>

<xs:complexType name="Tbidder">
  <xs:sequence>
    <xs:element name="date" type="xs:string"/>
    <xs:element name="time" type="xs:string"/>
    <xs:element name="personref" type="Tpersonref"/>
    <xs:element name="increase" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Titemref">
  <xs:attribute name="item" type="xs:string" use="required
     "/>
</xs:complexType>
```

```
<xs:complexType name="Tseller">
  <xs:attribute name="person" type="xs:string" use="required
     "/>
</xs:complexType>

<xs:complexType name="Tauthor">
  <xs:attribute name="person" type="xs:string" use="required
     "/>
</xs:complexType>

<xs:complexType name="Tannotation">
  <xs:sequence>
    <xs:element name="author" type="Tauthor"/>
    <xs:element name="description" type="Tdescription"/>
    <xs:element name="happiness" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tinterval">
  <xs:sequence>
    <xs:element name="start" type="xs:string"/>
    <xs:element name="end" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Topen_auction">
  <xs:sequence>
    <xs:element name="initial" type="xs:integer"/>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="reserve" type="xs:decimal"/>
      <xs:element name="bidder" type="Tbidder"/>
    </xs:choice>
    <xs:element name="current" type="xs:string"/>
    <xs:element name="privacy" type="xs:string" minOccurs
       ="0" maxOccurs="unbounded"/>
    <xs:element name="itemref" type="Titemref"/>
    <xs:element name="seller" type="Tseller"/>
    <xs:element name="annotation" type="Tannotation"/>
    <xs:element name="quantity" type="xs:string"/>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="interval" type="Tinterval"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="Topen_auctions">
  <xs:sequence>
    <xs:element name="open_auction" type="Topen_auction"
       minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tbuyer">
  <xs:attribute name="person" type="xs:string" use="required
     "/>
</xs:complexType>

<xs:complexType name="Tclosed_auction">
  <xs:sequence>
    <xs:element name="seller" type="Tseller"/>
    <xs:element name="buyer" type="Tbuyer"/>
    <xs:element name="itemref" type="Titemref"/>
    <xs:element name="price" type="xs:integer"/>
    <xs:element name="date" type="xs:string"/>
    <xs:element name="quantity" type="xs:string"/>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="annotation" type="Tannotation"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tclosed_auctions">
  <xs:sequence>
    <xs:element name="closed_auction" type="Tclosed_auction"
        minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tsite">
  <xs:sequence>
    <xs:element name="regions" type="Tregions"/>
    <xs:element name="categories" type="Tcategories"/>
    <xs:element name="catgraph" type="Tcatgraph"/>
    <xs:element name="people" type="Tpeople"/>
    <xs:element name="open_auctions" type="Topen_auctions"/>
    <xs:element name="closed_auctions" type="
       Tclosed_auctions"/>
  </xs:sequence>
</xs:complexType>

<!-- top level element -->
<xs:element name="site" type="Tsite">
  <xs:key name="key1">
    <xs:selector xpath="people/person"/>
    <xs:field xpath="@id"/>
  </xs:key>
  <xs:key name="key2">
    <xs:selector xpath=".//item"/>
    <xs:field xpath="@id"/>
  </xs:key>
  <xs:key name="key3">
    <xs:selector xpath="regions/europe/item"/>
    <xs:field xpath="@id"/>
  </xs:key>
  <xs:keyref name="keyRef1_key1" refer="key1">
    <xs:selector xpath="closed_auctions/closed_auction"/>
    <xs:field xpath="buyer/@person"/>
  </xs:keyref>
  <xs:keyref name="keyRef2_key2" refer="key2">
    <xs:selector xpath="closed_auctions/closed_auction"/>
    <xs:field xpath="itemref/@item"/>
  </xs:keyref>
  <xs:keyref name="keyRef3_key3" refer="key3">
    <xs:selector xpath="closed_auctions/closed_auction"/>
    <xs:field xpath="itemref/@item"/>
  </xs:keyref>
  <xs:keyref name="keyRef4_key3" refer="key3">
    <xs:selector xpath="open_auctions/open_auction"/>
    <xs:field xpath="itemref/@item"/>
  </xs:keyref>
</xs:element>
```

```
</xs:schema>
```

Listing A.1: Resulting XSD of Test Scenario B

# Appendix B

# Test Data

Test data from the XMark project [1]. Using the provided XML generator, XML document of size approximately 1.5 MB was generated. Figure B.1 is its DTD and other figures in this appendix are XQuery queries that query the XML document.

The queries was slightly modified by replacing calls of doc function in paths by / (document node).

```
<!-- DTD for auction database -->
<!-- $Id: auction.dtd,v 1.15 2001/01/29 21:42:35 albrecht
   Exp $ -->

<!ELEMENT site            (regions, categories, catgraph,
   people, open_auctions, closed_auctions)>

<!ELEMENT categories      (category+)>
<!ELEMENT category        (name, description)>
<!ATTLIST category        id ID #REQUIRED>
<!ELEMENT name            (#PCDATA)>
<!ELEMENT description     (text | parlist)>
<!ELEMENT text            (#PCDATA | bold | keyword | emph)
   *>
<!ELEMENT bold            (#PCDATA | bold | keyword | emph)
   *>
<!ELEMENT keyword         (#PCDATA | bold | keyword | emph)
   *>
<!ELEMENT emph            (#PCDATA | bold | keyword | emph)
   *>
<!ELEMENT parlist         (listitem)*>
<!ELEMENT listitem        (text | parlist)*>
```

```
<!ELEMENT catgraph        (edge*)>
<!ELEMENT edge            EMPTY>
<!ATTLIST edge            from IDREF #REQUIRED to IDREF #
   REQUIRED>

<!ELEMENT regions         (africa, asia, australia, europe,
   namerica, samerica)>
<!ELEMENT africa          (item*)>
<!ELEMENT asia            (item*)>
<!ELEMENT australia       (item*)>
<!ELEMENT namerica        (item*)>
<!ELEMENT samerica        (item*)>
<!ELEMENT europe          (item*)>
<!ELEMENT item            (location, quantity, name, payment
   , description, shipping, incategory+, mailbox)>
<!ATTLIST item            id ID #REQUIRED
                          featured CDATA #IMPLIED>
<!ELEMENT location        (#PCDATA)>
<!ELEMENT quantity        (#PCDATA)>
<!ELEMENT payment         (#PCDATA)>
<!ELEMENT shipping        (#PCDATA)>
<!ELEMENT reserve         (#PCDATA)>
<!ELEMENT incategory      EMPTY>
<!ATTLIST incategory      category IDREF #REQUIRED>
<!ELEMENT mailbox         (mail*)>
<!ELEMENT mail            (from, to, date, text)>
<!ELEMENT from            (#PCDATA)>
<!ELEMENT to              (#PCDATA)>
<!ELEMENT date            (#PCDATA)>
<!ELEMENT itemref         EMPTY>
<!ATTLIST itemref         item IDREF #REQUIRED>
<!ELEMENT personref       EMPTY>
<!ATTLIST personref       person IDREF #REQUIRED>

<!ELEMENT people          (person*)>
<!ELEMENT person          (name, emailaddress, phone?,
   address?, homepage?, creditcard?, profile?, watches?)>
<!ATTLIST person          id ID #REQUIRED>
<!ELEMENT emailaddress    (#PCDATA)>
```

```
<!ELEMENT phone          (#PCDATA)>
<!ELEMENT address        (street, city, country, province?,
    zipcode)>
<!ELEMENT street         (#PCDATA)>
<!ELEMENT city           (#PCDATA)>
<!ELEMENT province       (#PCDATA)>
<!ELEMENT zipcode        (#PCDATA)>
<!ELEMENT country        (#PCDATA)>
<!ELEMENT homepage       (#PCDATA)>
<!ELEMENT creditcard     (#PCDATA)>
<!ELEMENT profile        (interest*, education?, gender?,
   business, age?)>
<!ATTLIST profile        income CDATA #IMPLIED>
<!ELEMENT interest       EMPTY>
<!ATTLIST interest       category IDREF #REQUIRED>
<!ELEMENT education      (#PCDATA)>
<!ELEMENT income         (#PCDATA)>
<!ELEMENT gender         (#PCDATA)>
<!ELEMENT business       (#PCDATA)>
<!ELEMENT age            (#PCDATA)>
<!ELEMENT watches        (watch*)>
<!ELEMENT watch          EMPTY>
<!ATTLIST watch          open_auction IDREF #REQUIRED>

<!ELEMENT open_auctions  (open_auction*)>
<!ELEMENT open_auction   (initial, reserve?, bidder*,
   current, privacy?, itemref, seller, annotation, quantity,
    type, interval)>
<!ATTLIST open_auction   id ID #REQUIRED>
<!ELEMENT privacy        (#PCDATA)>
<!ELEMENT initial        (#PCDATA)>
<!ELEMENT bidder         (date, time, personref, increase)>
<!ELEMENT seller         EMPTY>
<!ATTLIST seller         person IDREF #REQUIRED>
<!ELEMENT current        (#PCDATA)>
<!ELEMENT increase       (#PCDATA)>
<!ELEMENT type           (#PCDATA)>
<!ELEMENT interval       (start, end)>
<!ELEMENT start          (#PCDATA)>
<!ELEMENT end            (#PCDATA)>
```

```
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>

<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price,
   date, quantity, type, annotation?)>
<!ELEMENT buyer           EMPTY>
<!ATTLIST buyer           person IDREF #REQUIRED>
<!ELEMENT price           (#PCDATA)>
<!ELEMENT annotation      (author, description?, happiness)>

<!ELEMENT author          EMPTY>
<!ATTLIST author          person IDREF #REQUIRED>
<!ELEMENT happiness       (#PCDATA)>
```

Listing B.1: DTD of the test XML data

```
for $b in /site/people/person[@id = "person0"] return $b/
   name/text()
```

Listing B.2: Test query 1.

```
for $b in /site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>
```

Listing B.3: Test query 2.

```
for $b in /site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2 <= $b/
   bidder[last()]/increase/text()
return
  <increase
  first="{$b/bidder[1]/increase/text()}"
  last="{$b/bidder[last()]/increase/text()}"/>
```

Listing B.4: Test query 3.

```
for $b in /site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

Listing B.5: Test query 4.

```
count(
  for $i in /site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

Listing B.6: Test query 5.

```
for $b in //site/regions return count($b//item)
```

Listing B.7: Test query 6.

```
for $p in /site
return
  count($p//description) + count($p//annotation) + count($p
     //emailaddress)
```

Listing B.8: Test query 7.

```
for $p in /site/people/person
let $a :=
  for $t in /site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Listing B.9: Test query 8.

```
let $ca := /site/closed_auctions/closed_auction return
let
    $ei := /site/regions/europe/item
for $p in /site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
    let $n := for $t2 in $ei where $t/itemref/@item = $t2/
       @id return $t2
    return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

Listing B.10: Test query 9.

```
for $i in
  distinct-values(/site/people/person/profile/interest/
     @category)
let $p :=
  for $t in /site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

Listing B.11: Test query 10.

```
for $p in /site/people/person
let $l :=
  for $i in /site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{$p/name/text()}">{count($l)}</items>
```

Listing B.12: Test query 11.

```
for $p in /site/people/person
let $l :=
  for $i in /site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</
   items>
```

Listing B.13: Test query 12.

```
for $i in /site/regions/australia/item
return <item name="{$i/name/text()}">{$i/description}</item>
```

Listing B.14: Test query 13.

```
for $i in /site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

Listing B.15: Testing query 14.

```
for $a in
  /site/closed_auctions/closed_auction/annotation/
     description/parlist/
   listitem/
   parlist/
   listitem/
   text/
   emph/
   keyword/
   text()
return <text>{$a}</text>
```

Listing B.16: Test query 15.

```
for $a in /site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/parlist/
         listitem/text/emph/
        keyword/
        text()
    )
  )
return <person id="{$a/seller/@person}"/>
```

Listing B.17: Test query 16.

```
for $p in /site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

Listing B.18: Test query 17.

```
declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal?) as xs:
   decimal?
{
  2.20371 * $v (: convert Dfl to Euro :)
};

for $i in /site/open_auctions/open_auction
return local:convert(zero-or-one($i/reserve))
```

Listing B.19: Test query 18.

```
for $b in /site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending empty greatest
return <item name="{$k}">{$b/location/text()}</item>
```

Listing B.20: Test query 19.

```
<result>
  <preferred>
    {count(/site/people/person/profile[@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        /site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count(/site/people/person/profile[@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in /site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```

Listing B.21: Test query 20.

# Appendix C

# Content of CD

The CD attached to this thesis has the following structure.

- `content.txt` - A file with this text.
- `text/` - A PDF version of the thesis.
- `src/` - Source codes of the jInfer framework including implementation of our method. The same source codes can be also obtained from public Subversion repository by issuing command:
  ```
  svn co -r 2155
  https://jinfer.svn.sourceforge.net/svnroot/jinfer/jinfer/trunk/
  ```
- `bin/` - jInfer plugins for NetBeans 7.0.1. See jInfer Tutorial [18] for step-by-step instructions, but use NetBeans 7.0.1. Since the tutorial is for the last official jInfer release 1.0, it says the required version of NetBeans is at least 6.9. But this is not true in our case, because we use the development version of jInfer and it requires NetBeans 7.0.1.
- `testing/` - A directory containing sets of test data and test results. Again, see the Jinfer Tutorial [18] for instructions how to run the inference.

# List of Tables

# List of Figures