Swiss Federal Institute of Technology

Master Project

Efficient Semi-structured Queries in Scala using XQuery Shipping

Fatemeh Borran-Dejnabadi

Computer Science

Responsible: Gilles Dubochet
Supervisor: Prof. Martin Odersky
LAMP-I&C-EPFL

February 2006

Abstract

This project proposes a new approach to interact with database systems through programming languages. A formal query language can be integrated within modern programming languages and the semi-structured queries can be evaluated using automatic transformation and query shipping. The focus of this project is on XML queries and Scala programming language. Particularly, this project optimizes the XML-based expressions of Scala using XQuery transformation and Shipping. In this work, Scala sequence comprehensions are extended to cover appropriately the whole functionalities of XQuery FLWOR expressions and XQuery sequence comparisons are introduced in Scala to facilitate query generation.

This report presents a formalization of transformation rules between Scala and XQuery languages and describes an Scala implementation. Various use cases are provided to facilitate understanding and employing this newest Scala library.

Table of Contents

Chapter 1

Introduction

This report represents the results of four months Master's Project research completed at the Programming Methods Laboratory (LAMP) of Swiss Federal Institute of Technology (EPFL).

The report is composed of six chapters:

Chapter 1 presents the latest research papers and advanced technical topics to explain the interest and motivation for realizing this project.

Chapter 2 defines the project and its goals, and then formulates the questions and problems that should be answered and resolved during this project. It explains why the other solutions are not properly capable to resolve the problem. In other words, why such a project is needed?

Chapter 3 compares Scala programming language with XQuery language which is a native XML query processing language. The aim of this comparison is not only introducing the functionalities of XML query languages, but also is discussing about the common and missing features in Scala and XQuery languages. This chapter offers an excellent reflection to the Scala developer both to extend the XML library, and to generalize the famous sequence comprehensions of Scala.

Chapter 4 represents the most important results of this project which are the transformation rules from Scala programming language to XQuery language. However this project is implemented based on the abstract syntax transformation, the achieved rules are illustrated based on the core syntax transformation, only for simplicity reasons.

Chapter 5 explains more implementation techniques to apply on the obtained results form chapter 4. It describes the structure of the implemented project and represents the modified and newly added libraries to the Scala programming language.

Chapter 6 illustrates some future works and concludes.

In this report, I assume that the reader is familiar with Scala Programming Language [1, 2] developed by LAMP laboratory at the EPFL, and understands the fundamentals of XML [4] and relevant technologies such as XPath [5] and XQuery [3, 6].

Query Language Integration

One of the most recent challenges in evolution of object-oriented (OO) programming technologies is to reduce the complexity of accessing and integrating information that is not natively defined using OO technology. The two most common sources of non-OO information are relational databases and XML. Adding general purpose query facilities into the modern programming languages are the next generation of technologies.

Integrating queries within programming languages (in a similar way as LINQ project [7, 8] for .Net Frameworks) is an advanced topic in which the query is an incorporated feature of the developer's primary programming language (e.g., Java, C#, Scala). This approach allows query expressions to benefit from a rich class library, compile-time syntax checking, static typing that was previously available only to imperative code.

Scala programming language currently contains a set of most common query operators that allows selection, projection, traversal and filter operations. The extensibility of the query architecture may provide implementations that work over both XML and SQL data. More importantly, Scala can use additional services such as remote evaluation, query translation, and optimization provided by the database systems to process the queries.

Query Shipping vs. Data Shipping

Query processing in a client-server database system raises the question of where to execute queries to minimize the communication costs and response time of a query, and to load-balance the system. The two common query execution strategies are: data shipping and query shipping. Data shipping means that queries be executed at clients; query shipping means that queries be executed at servers. The experiments with a client-server model confirm that the query execution policy is critical for the performance of a system. Neither data nor query shipping are optimal in all situations, and the performance penalties can be substantial. The best solution is a combination of both strategies which matches the best performance of data and query shipping [11].

Relational database systems are typically based on query shipping, in which the majority of query processing is performed at servers. The benefits of query shipping include: the reduction of communication costs for high selectivity queries since only query results will be sent back to client, the ability to exploit server resources when they are plentiful, and the ability to tolerate resource-poor client machines.

Object-Oriented database systems, on the other hand, are typically based on data shipping, in which required data is transmitted into the client and query is processed there. Data shipping has the advantages of exploiting the resources, including CPU, memory and disk, of powerful client machines (imagine there are a large number of client machines) and reducing communication in the presence of locality or large query results. Data shipping can make use of more resources at client side, however, it suffers when server resources are plentiful, or locality of data access at clients is poor (suppose server send a vast amount of data but less than 1% of them be used by clients).

Scala Query Integration and Shipping

Although the idea of shipping the nested comprehensions into a real SQL query already exists in some special functional languages such as Kleisli [12] and Slinks [13], this project proposes the same approach for processing the XML queries in Scala programming language using XQuery shipping. Afterward, Scala interpreter isn't aware of query optimization techniques, and XQuery processor evaluates them efficiently.

This project introduces the XML query characteristics of Scala by which developers can implement and process their own queries on XML sources. As it will be explained later in this project, evaluating queries using Scala interpreter involves XML data transferring from host to the Scala application machine. Because data shipping is not suitable due to the huge quantity of information and the reasons explained before, query shipping is applied in this project.

Hence, the aim of this project is; first, implementing a simple and efficient translator that transforms a specified Scala code to an equivalent XQuery code. Then, evaluating the transformed code using an XQuery processor (such as Galax Implementation [14]) instead of Scala interpreter. Finally, extracting the query processing results in a readable format for Scala. Although XQuery is a new language, it benefits from many optimization techniques that exist for functional programming languages and for other database query languages. This project uses Scala programming language native syntax for XML queries and extends its capabilities and performance by shipping queries toward an XQuery processor.

Chapter 2

Scala Query Shipping Definition

The basic idea in this project is that: currently Scala has a rich XML library by which one can create, parse, and process the XML documents. This library also supports some XML path expressions proposed by W3C recommendation. On the other hand, the “Sequence Comprehensions” of Scala are the powerful tools for querying, and XML library of Scala supports these comprehensions. Therefore, one may conclude that an XML query can be implemented and evaluated using Scala programming language.

The xml library of Scala supports the principal aspects of XML Query Language, such as: projection, selection, join and construction. As illustrated in the following example, a “for comprehension” can prepare a query in which the projection is supported by path expressions, the selection is implemented using yield statement, the join is guaranteed by a set of generators (`val x <- e; val y <- e'`), and finally results can be constructed using XML elements. Consequently, we own an integrated query language in Scala, by which, we can write XML queries in Scala syntax and process them using Scala interpreter.

```
for (val b <- load(“bib.xml”) \ “book”;
     val a <- b \ “author”;
     b \ “year” > 2000)
yield
     <result>
       {b \ “title”} {a}
     </result>;
```

Although xml library of Scala provides adequate techniques to query an XML data, but processing the whole query with Scala interpreter is not very efficient, because:

- Shipping the entire unprocessed data requires more transfer-time than shipping query and processed data. For instance, if we consider a client-server architecture in which XML data is not located on the same machine as Scala application, query evaluation needs data shipping. The size of data after query evaluation reduces importantly. And query size comparing to data size is approximately negligible. So query size plus processed data size is always less than unprocessed data. And transferring the first one is preferable.
- Unprocessed data requires more memory than processed data. This is a direct result from the last point.

- Query optimization and performance criteria can not be totally supported by a programming language like Scala. Because, evidently a programming language is not a database system.

Query Processing using XQuery

Our objectives in this project are to reduce the transfer-time and memory-consume of Scala application using “Query Shipping” instead of “Data Shipping”. In fact, this project proposes a new technique that accumulates the advantages of “Query Languages Integration” by “Query Shipping” strategy.

This method is described here using a scenario:

- The queries are implemented in Scala application using the same syntax.
- They are parsed and type checked using Scala parser and Scala type checker.
- But, they are not processed by Scala compiler during compile-time.
- During execution-time, the queries are transformed into XQuery language.
- The transformed query is shipping toward an XQuery processor.
- XQuery processor evaluates and returns back the results.
- The results are loaded as a well-known format in Scala application.

The principal advantages of using this approach are listed below:

- The queries have the same syntax as programming language.
- The programming language (Scala in our case) performs query syntax checking during compile-time.
- The complex queries could be implemented very easily using the programming language facilities.
- The developer is not forced to manage several query languages for realizing its own application. In fact, it is enough to be a Scala programmer and implement the queries on Scala syntax which will be interpreted by SQL database system or XQuery processor.
- The query can be interpreted partially using database system and partially using programming language interpreter.
- Since the queries are automatically transformed from a programming language, they would be meaningful.

A global view of this project is represented in Figure 1. Suppose Scala application is far from data center. The Scala code may contain a query part. This query part is separated from other parts by assigning a special type, called `TypedCode`, and transformed to another `TypedCode` that contains equivalent XQuery code. Then, XQuery code is shipped toward an XQuery processor and interpreted. Finally, the results of XQuery code is loaded in Scala application and Scala application interprets the final results. As you see, we don’t need to keep any

information about the XML data on the Scala application. As you see, the surrounded code by a `TypedCode`, from reflection library of Scala, is processed differently even during compile-time as well as execution-time.

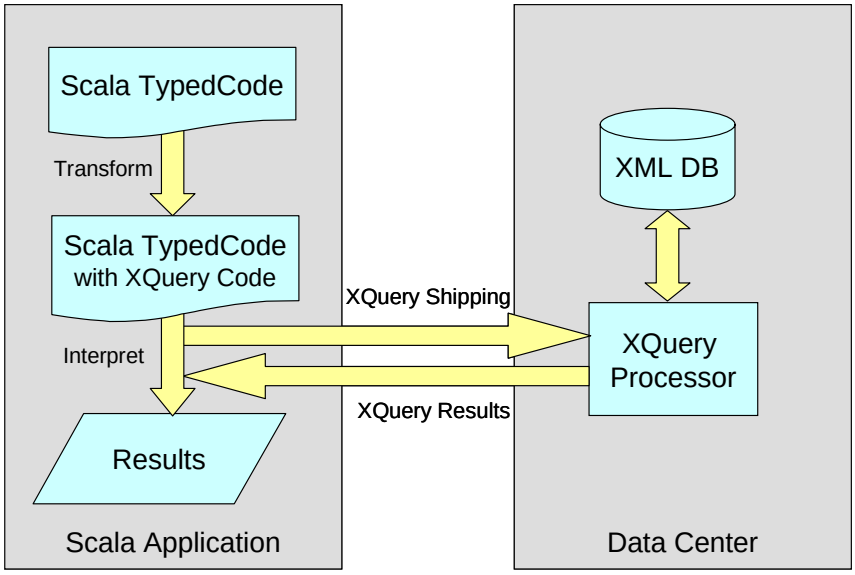

Figure 1: Global View

More detailed scenario is represented in Figure 2. Scala code is parsed by Scala parser to generate Scala Abstract Syntax Tree (AST). Then, the generated AST is type checked by Scala type system and a typed AST is created. Scala code generator generates the byte-codes in compile-time. During execution, another AST is generated for the query part (`TypedCode`) of Scala code. The constructed tree here, is used to perform the transformation through execution-time. In fact, the transformer generates another Scala AST that contains the transformed XQuery code. This newly Scala AST, invokes an XQuery processor to interpret the XQuery Code. Finally Scala interpreter evaluates the final results.

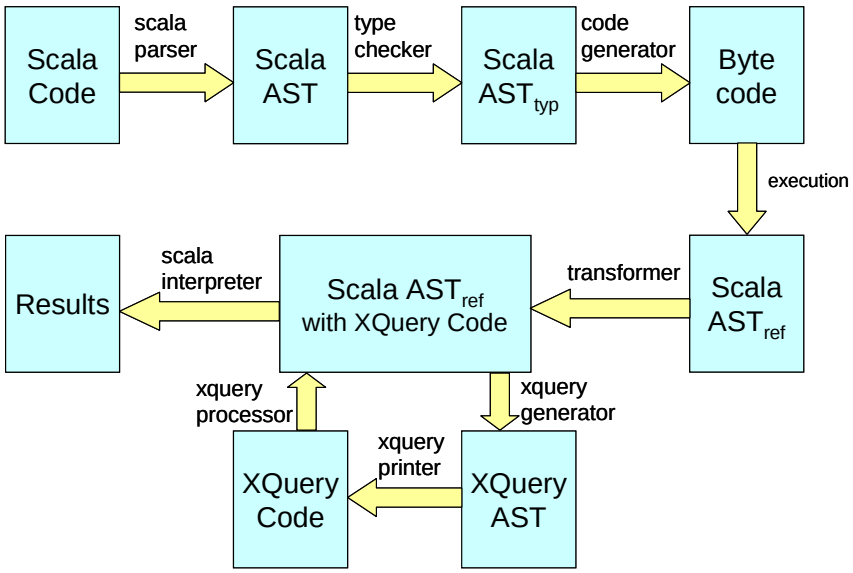

Figure 2: Detailed Scenario

The transformation is a tree to tree transformation. It means that, we have an abstract syntax tree and a pretty printer for XQuery and once we obtain the XQuery source code we can interpret it using the XQuery processor.

One advantages of constructing an XQuery abstract syntax tree instead of generating source code is that, one may transform the constructed tree to an appropriate tree in one XQuery implementation (for example, eXist AST). In this case, instead of transferring the source code inside a string, one could transfer the root of abstract syntax tree and process the AST directly by XQuery processor. Using this approach we can improve query processing even more.

As it was explained, the transformation is a Scala `TypedCode` transformation. It means that, Scala AST (`TypedCode`) is transformed to another Scala AST (`TypedCode`) that contains XQuery Code. This method allows successive transformations. For instance, Scala code can be transformed and processed once by an XQuery processor and then by another database processor like JDBC.

The principal part of this project is extracting the transformation rules from Scala code to XQuery code, and implementing an efficient transformer from Scala AST to XQuery AST. This requires specifying a compact XQuery abstract syntax tree from XQuery EBNF proposed by W3C recommendation. And, implementing an XQuery pretty printer for its abstract tree.

The only inconvenient that we can imagine for this project is that, the transformation and transferring source code may consume more Scala processing time than interpreting. And, if no transformation rule is found, the transformer will be interrupted on execution-time and not compile-time. In this case, Scala interpreter can continue the query evaluation instead of XQuery processor.

Chapter 3

Scala and XQuery Comparison

As it was mentioned before, Scala programming language has a rich API for XML data processing, and the “Sequence Comprehensions” are powerful features for formulating XML queries. Therefore, XML queries in Scala produces the same results as an XML query implementation.

In this chapter, I compare Scala programming language with a native XML query language: XQuery. The aim of this comparison is representing some common and missing features in Scala for querying an XML data. Consequently, extracting the correspondence between Scala code and XQuery code, which results the transformation rules given in the next chapter. Also, illustrating how an Scala developer can construct its proper queries on XML sources using Scala syntax. Only the xml library of Scala is considered in this comparison.

XQuery and Scala Backgrounds

XQuery like Scala operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure, known as the “data model”. In XQuery data model, a value is always a “sequence”. A sequence is an ordered collection of zero or more “items”. An item is either an “atomic value” or a “node”. An atomic value is a value in the value space of an atomic type. Each node has a unique “node identity”, a “typed value”, and a “string value”. The typed value of a node is a sequence of zero or more atomic values. The string value of a node is a value of type string (see Figure 3) [3, 14].

An XQuery ordering called “document order” is defined among all the nodes accessible during processing of a given query, which may consist of one or more trees. Document order is the order in which nodes appear in the XML serialization of a document and it is stable, which means that the relative order of two nodes will not change during the processing of a given query.

XQuery “atomization” is applied to a value when the value is used in a context in which a sequence of atomic values is required. For example, atomization is used in processing of the arithmetic expressions, comparisons, and function calls. The result of atomization is either a sequence of atomic values or a type error.

An XQuery “type error” may be raised during the static analysis phase or the dynamic evaluation phase. During the static analysis phase, a type error occurs

when the static type of an expression does not match the expected type of the context in which the expression occurs. During the dynamic evaluation phase, a type error occurs when the dynamic type of a value does not match the expected type of the context in which the value occurs.

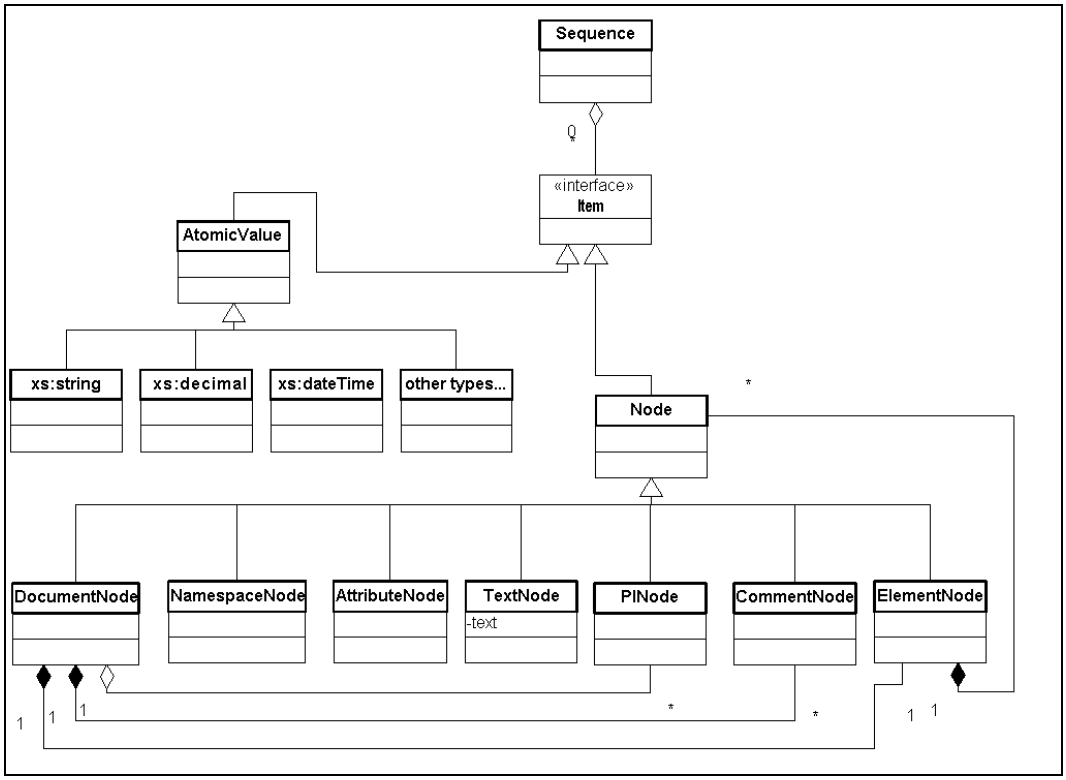

Figure 3: XQuery Data Model

The xml library in Scala has a similar architecture. `scala.xml.NodeSeq` class extends from `scala.seq[Node]` and represents any sequence of nodes or documents (see Figure 4) [2]. There are two principal kinds of XML nodes in Scala: An element (`scala.xml.Elem` class) represents an XML element including its prefix, name, scope, attributes and children. The special nodes (`scala.xml.SpecialNode` class) such as comments, processing instructions and texts are the other kinds of XML nodes. There is not a node identity with the same specification in XQuery; there is only a `hashCode()` method for each node. Each node has a string value (can obtain by `text` method) and may contain an `xmlType()` if there is an XML Schema for it. An XML node is automatically a `NodeSeq` because it is a sequence of only one node. An empty sequence is represented by `scala.xml. NodeSeq.Empty`. XML documents and XML attributes are defined differently in Scala. XQuery attributes are some special nodes while, in Scala there is another class for them (`scala.xml.MetaData`).

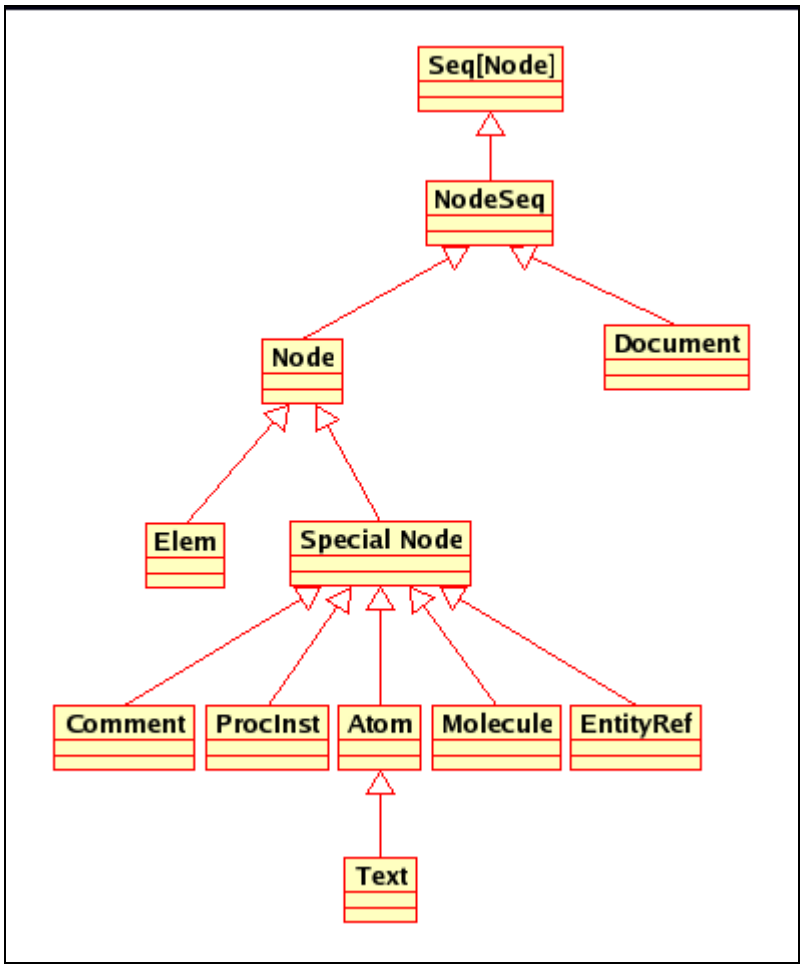

Figure 4: Scala XML Data Model

XQuery Expressions

The basic building blocks of XQuery is the expression, which is a string of characters. The principal expressions of XQuery are: primary expressions, FLWOR expressions, path expressions, comparison expressions, and so on.

This report uses bibliography data to illustrate the basic features of XQuery. It is taken from the XML Query Use Cases [3] and appears in the following table:

Table 1: bib.xml

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>V.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and Content for Digital
TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
            <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

Primary Expressions

Primary expressions are the basic primitives of XQuery language. They include literals, variable references, constructors, and function calls.

Literals

A literal is a direct syntax representation of an atomic value. XQuery supports two kinds of literals: numeric literals and string literals. The principal XQuery numeric literals are: integers (like `12`) of type `xs:integer`, decimals (like `12.5`) of type `xs:decimal`, and doubles (`125e2`) of type `xs:double`. Some other popular atomic types are `xs:date`, `xdt:dayTimeDuration`.

All values in Scala are the objects. Scala `Any` class has two subclasses: `AnyVal` and `AnyRef` representing value classes and reference classes. All value classes are predefined and they correspond to the primitive types of Java-like languages (`Double, Float, Long, Int, Short, Byte, Boolean, String`). All other classes define reference types.

Variables

A variable reference in XQuery is a QName preceded by a $-sign. QName stands for XML element names, attribute names and so on. Two variable references are equivalent if their local names are the same and their namespace prefixes are bound to the same address. An unprefixed variable reference is in no namespace. XQuery uses a $-sign for the variables to distinguish between a QName and a variable. For example in the expression `$b/title`, `b` is a variable which is defined before (binds to some sequences), but `title` is a QName defined in XML document.

Scala uses another technique to make difference between XML QNames and variables. XML names in Scala are the strings enclosed by double quotes. So the previous expression in Scala would be: `b \ "title"`. This notation may be is not the best solution, but for the moment, it resolves many problems related to the XML syntax. Other solution is importing a new type QName in Scala and defining `title` as a QName before using.

Function Calls

A function call in XQuery consist of a QName followed by a parenthesized list of zero or more expressions, called arguments. If the QName has no namespace prefix, it is considered to be in the default function namespace.

In Scala, a function can be called by its name and its arguments enclosed by parentheses if any. A function without any arguments has no parentheses. If a function is defined outside of the current scope, a full name is required. The full name of a function contains the packages names followed by the class name separated by dot notation.

Input Functions

The only input function supported by Galax implementation is `doc()` which returns an entire document, identifying by a Universal Resource Identifier (URI).

```
doc("bib.xml")
or
doc("http://www.w3schools.com/xml/note.xml")
```

There are several methods in xml library of Scala to load an XML file. The most common is `load(fileName:String)` from `XML` object which loads XML

file from given file. The `fileName` can contain an address or URL to the specific file.

```
load(“bib.xml”)
or
load(“http://www.w3schools.com/xml/note.xml”)
```

`doc()` function returns the root element inside a document node, while, `load()` method returns the root element itself.

In both cases, a dynamic error is raised if the specified document or file is not found or is not accessible.

Built-in Functions

XQuery has a set of built-in functions and operators, including aggregation function such as `min()`, `max()`, `count()`, `sum()` and `avg()`; numeric functions like `round()` and `floor()`; string functions like `concat()`, `string-length()`, `starts-with()`, `ends-with()` and etc. The input functions and `distinct-values()` are also some special built-in functions. Two other frequently used functions are `not()` and `empty()`. The `not()` function is used to inverse the boolean conditions and `empty()` function which is the opposite of `exists()` reports whether a sequence is empty.

In this project, some useful built-in functions of XQuery (such as aggregations and `distinct-values()`) that are not present in Scala are added in an auxiliary library (inside xquery library). Afterward, Scala developers have more alternatives to implement their applications using the functions from this library.

FLWOR Expressions

FLWOR expressions, pronounced “flower expressions”, are the most powerful and common expressions in XQuery. They are similar to the `SELECT-FROM-WHERE` statements in SQL. However, a FLWOR expression is not defined in terms of tables, rows, and columns, instead, it binds variables to values in for, let clauses, and uses these variable bindings to create new results. A combination of variable bindings created by the for and let clauses of a FLWOR expression is called a tuple.

For instance, here is a simple FLWOR expression that returns the title of each book that was published in the year 2000.

```
for $b in doc(“bib.xml”)//book
where $b/@year = “2000”
return $b/title
```

This query binds the variable `$b` to each book, one at a time, to create a series of tuples. Each tuple contains one variable binding in which `$b` is bound to a single book. The where clause tests each tuple to see if `$b/@year` is equal to `"2000"`, and the return clause is evaluated for each tuple that satisfies the condition expressed in the where clause.

The name FLWOR is an acronym, standing for the first letter of clauses that may occur in a FLWOR expression:

- **for clauses:** associate one or more variables to expressions, creating a tuple stream in which each tuple binds a given variable to one of the items to which its associated expression evaluated.
- **let clauses:** bind variables to the entire result of an expression, adding these bindings to the tuples generated by a for clause, or creating a single tuple to contain these bindings if there is no for clause.
- **where clauses:** filter tuples, retaining only those tuples that satisfy a condition.
- **order by clauses:** sort the tuples in a tuple stream before the return clause is evaluated in order to change the order of results.
- **return clauses:** build the result of the FLWOR expression for a given tuple.

The acronym FLWOR roughly follows the order in which the clauses occur. A FLWOR expression starts with one or more for or let clauses in any order, followed by an optional where clause, an optional order by clause, and a required return clause. The result of the FLWOR expression is an ordered sequence containing the results of these evaluation, concatenated as if by the comma operator.

Sequence comprehensions in Scala programming language are very similar to FLWOR expressions. They are also implemented for `NodeSeq` class in xml library of Scala. In fact, every data type that supports `filter`, `map`, and `flatMap` methods can be used in sequence comprehensions. For example the above query could be written in Scala as:

```
for (val b <- load(“bib.xml”) \\ “book”;
       b.attribute(“year”) == “2000”;
       val t <- b \ “title”)
yield t;
```

Scala equivalence core syntax is:

```
(load(“bib.xml”) \\ “book”).
filter(b => b.attribute(“year”) == “2000”).
flatMap(b => (b \ “title”).map(t => t))
```

Some differences between FLWOR expressions and Sequence comprehensions are listed here:

- Sequence comprehensions don't support `let` and `order by` clauses.
- There are some differences between `yield` statement and `return` clause that will be illustrated later at the end of this section.

Although sequence comprehension does not support `let` and `order by` clauses in its sweet syntax, they can be implemented as some additional methods in required classes. For instance, we can add `let()`, and `orderBy()` methods implementation in `NodeSeq` class, which have the same effect as in FLWOR expressions. The `let()` method binds this sequence to a given variable and returns another sequence. The `orderBy()` method changes the order of this sequence by a given sequence and returns the ordered sequence.

Here is the definition for these two lambda functions:

```
NodeSeq.let(f: NodeSeq => NodeSeq): NodeSeq

NodeSeq.orderBy(f: Node => NodeSeq, order: String): NodeSeq
```

For example, consider the following `let` clause in XQuery which returns first name of all authors in our bibliography:

```
let $a := doc("bib.xml")//author
return $a/first
```

Using our recent functions, equivalent Scala code is:

```
(load("bib.xml") \ "author").let(a => a \ "first")
```

And suppose, a simple `order by` clause as:

```
for $b in doc("bib.xml")//book
order by $b/title ascending
return $b
```

which returns all books sorted by their titles. Similar code in Scala is:

```
(load("bib.xml") \ "book").
orderBy(b => b \ "title", "ascending")
```

Note that, both `order by` clause in XQuery and `orderBy()` function in Scala sort a sequence of nodes by comparing the text value of each node, and not numeric value for instance. For example, if we want to sort the books by their prices (price is a double value), the result will not be what we expected! (because `125 < 35 < 65` if we consider text values)

Now if we consider more complicated query like:

```
let $b := doc("bib.xml")//book
for $a in $b/author
order by $a/last descending
return $a/first
```

Scala corresponding code will be:

```
(load(“bib.xml”) \ “book”).let(b => (b \ “author”).
orderBy(a => a \ “last”, “descending”).
flatMap(a => a \ “first”))
```

As it was mentioned before, the sequences in XQuery can not be nested, while in Scala the nested sequences are allowed. For instance, `List(1, List(2, 3))` is not equivalent to `List(1, 2, 3)`, while in XQuery they are equal. This difference becomes more serious when we compare for-comprehensions with FLWOR expression. In fact return clause in XQuery can return different values with different types in a same FLWOR expression. For instance, `1` and `List(2, 3)` results `List(1, 2, 3)`. While in Scala a for-comprehension can not yield different types.

On the other hand, a Scala node in xml library is a sequence of one node. So `Node` and `NodeSeq` can merged to generate a new `NodeSeq` within a for-comprehension. For the same reason, there is no difference between `flatMap` and `map` implementation in XQuery. In other words, `map` method becomes a `flatMap` in XQuery.

For example, following for-comprehension:

```
for (val x <- e; val y <- e’) yield e’’
```

is equivalent to the following FLWOR expression:

```
for $x in e
for $y in e’
return e’’
```

also, it is equivalent to:

```
for $x in e
return
  for $y in e’
    return e’’
```

Since the first XQuery code will be transformed to the second one for processing, it would be very simple and efficient if our Scala to XQuery transformer generates directly the second syntax. On the other hand, XQuery syntax will be simplified if we consider only the second syntax. In summery, according our implementation, a FLWOR expression is one for or let clause, followed by an optional where clause, and followed by a required return clause.

Note that, these two Scala codes are not equivalent:

```
for (val x <- e; val y <- e’) yield e’’
and
```

```
for (val x <- e)  yield
  for (val y <- e’) yield
    return e’’
```

because the first one is a `flatMap` method, while the second one is a `map` method in Scala core syntax:

```
e.flatMap(x => e’.map(y => e’’))
and
e.map(x => e’.map(y => e’’))
```

Moreover, the second code is not valid if `e` and `e’` belong to the `NodeSeq` class. But in XQuery these two codes are equal; the first one is just a sweet syntax for the second one:

```
for $x in e
for $y in e’
return e’’

for $x in e
return
  for $y in e’
    return e’’
```

So, there is an obvious difference between Scala `yield` statement and XQuery `return` clause. In fact, XQuery’s results are always “flatted”, it is exactly what we try to do by `yield` in Scala.

The Positional Variable ‘at’ in FLWOR Expressions

The for clause supports positional variables, which identify the position of a given item in the expression that generated it. For instance, the following query returns the title of second book in the bibliography:

```
for $b at $i in doc(“bib.xml”)//book
where $i = 2
return $b/title
```

This notion in FLWOR expression is similar to the numeric predicates in the XPath expressions. Above query can be written as using only the XPath expressions:

```
doc(“bib.xml”)//book[2]/title
```

However the later query is also supported by an XQuery implementation and has a compact syntax, but the XQuery processor only recognizes the first query. In fact, the second query is just a sweet syntax of the first one, for programmer usage. And, XQuery supports the positional variable of FLWOR expressions in its core syntax. In this project, in order to simplify our XQuery syntax, we generate and support only the positional variable in FLWOR expressions and not predicates in XPath expressions.

The `NodeSeq` class in xml library of Scala has an `apply()` method to make its elements accessible. So the exceeding query in Scala is:

```
(load(“bib.xml”) \\ book).apply(1) \ “title”
```

Only difference is that the sequence counter in XQuery starts from 1, though Scala like the other programming languages counts form 0.

Eliminating Duplicates with ‘distinct-values()’ in FLWOR Expressions

Data often contains duplicate values, and FLWOR expressions are often combined with the `distinct-values()` function to remove duplicates form sub-trees. The following query returns the last name of each author:

```
doc(“bib.xml”)//author/last
```

Since one of the authors wrote two books in the bibliography, the result of this query contains a duplicate: `<last>Stevens</last>`.

The `distinct-values()` function extracts the values of a sequence of nodes and creates a sequence of unique values, eliminating duplicates.

```
distinct-values(doc(“bib.xml”)//author/last)
```

As you can see, the output of the above query is:

```
Stevens Abiteboul Buneman Suciu
```

The `distinct-value()` function eliminates duplicates, but in order to do so, it extracts values form nodes. FLWOR expressions are often used together with `distinct-values()` to create subtrees that correspond to sets of one or more unique values. For the preceding query, we can use an element constructor to create a last element containing each value:

```
for $l in distinct-values(doc(“bib.xml”)//author/last)
return <last>{ $l }</last>
```

In Scala a sequence can contain duplicate values but an Scala `Set` can not. In order to obtain a sequence without the duplicates, we can put its elements in a set collection. In this project, I implemented a `distinct-values()` function which takes a sequence of nodes and returns a non-duplicated sequence. This function is located in the xquery library of Scala and it can be used by Scala programmers like other built-in functions. The definition of this function is given here:

```
distinct_values(nodes: NodeSeq): NodeSeq
```

Using this recent method, the previous query in Scala language will be:

```
distinct_values(load(“bib.xml”) \\ “author” \ “last”)
```

This implementation seems more general than the XQuery's one, because it eliminates the duplicates and simultaneously maintains the tree structure. During transformation, the required structure is constructed and added to the XQuery built-in function.

Path Expressions

In XQuery, path expressions are used to locate nodes in XML data. XQuery's path expressions are derived from XPath 1.0. A path expression consists of a series of one or more steps, separated by a slash, `/`, or double slash, `//`. Every step evaluates to a sequence of nodes.

```
doc("bib.xml")/bib/book/author
or
doc("bib.xml")//author
```

The projection functions, `\` and `\\`, in xml library of Scala are similar to path expressions in XQuery.

```
load("bib.xml") \ "book" \ "author"
or
load("bib.xml") \\ "author"
```

In both cases, the steps are evaluated from left to right. The first step identifies a sequence of nodes using an input function, a variable that has been bound to a sequence of nodes, or a function that returns a sequence of nodes. The expression on the left-hand side is evaluated firstly and returns the resulting nodes in document order, then the right-hand side expression is evaluated once for each left-hand side node, merging the results to produce a sequence of nodes in document order. If the result contains anything that is not a node, a type error is raised. When the right-hand expression is evaluated, the left-hand node for which it is being evaluated is known as the context node.

The only difference between two languages is that: path expressions in XQuery which start with an input function should include the document node while this should not be appear in projection functions of Scala. So, we should be aware of this difference to transform Scala code to XQuery code properly.

The step expressions that may occur on the right hand side of a `/` in XQuery are the following:

- A NameTest, which selects element or attribute nodes based on their name. A simple string is interpreted as an element name; for instance author in the previous examples. If the name is prefixed by the `@` character, then the NameTest evaluates to the attributes of the context node that have the specified name. For instance,

```
doc("bib.xml")/bib/book/@year
```

returns the year attribute of each book.

In Scala an attribute can be extracted in the same manner as XQuery or using attribute method of `Node` class.

```
load("bib.xml") \ "book" \ "@year"
or
for (val b <- load("bib.xml") \ "book") yield
  Text(b.attribute("year"))
```

The namespaces and wildcards are supported in both languages. The namespaces comes before the node name separated by `':'` in both languages and wildcards are `'*'` and `'_'` in XQuery and Scala.

- A KindTest, which selects processing instructions, comments, texts and nodes: `processing-instruction(), comment(), text(), node()`. Scala does not support this kind of tests in its path expressions.

- An expression that uses an explicit "axis" together with a NameTest or KindTest to choose nodes with a specific structural relationship to the context node. If the NameTest book selects book elements, then `child::book` selects book elements that are children of the context node. Principal axis supported by XQuery are:

  - Forward axis: `self, child, attribute, descendant, descendant-or-self, following and following-sibling`.
  - Revese axis: `parent, ancestor, ancestor-or-self, preceding and preceding-sibling`.

  There are some corresponding methods for these axis in Scala:
  `child, attribute, descendant, descendant_or_self`.

  In general, some forward axis can be replaced by equivalent path expressions, for instance, `books/child::book` is equivalent to `books/book` and so on. So, it is not really necessary to have an implementation for each of them in Scala.

- A PrimaryExpression, which may be a literal, a function call, a variable name, or a parenthetical expression.

  The predicates follow the PrimaryExpressions and there are two different types in XQuery:

  - Boolean Predicates, are boolean conditions, between square brackets, `[]`, that select a subset of the nodes computed by a step expression.

    ```
    doc("bib.xml")//author[last="Stevens"]
    ```

But, this query is equivalent to the following FLWOR expression:

```
for $author in doc(“bib.xml”)//author
where $author/last = “Stevens”
return $author
```

So we don't need to support the boolean predicates in path expressions of Scala. We can simply write a for-comprehension that filters the nodes which satisfy the predicate:

```
for (val author <- load(“bib.xml”) \\ “author”;
author \ “last” == “Stevens”)
yield author
```

- Single numeric value Predicate

  The predicate like:

  ```
  doc(“bib.xml”)//book/author[1]
  ```

  is equivalent to the following FLWOR expression:

  ```
  for $b in doc(“bib.xml”)//book
  for $a at $i in $b/author
  where $i = 1
  return $a
  ```

  So, once again, it is not necessary to support the numeric value predicates in Scala and the equivalent code in Scala is:

  ```
  for (val b <- load(“bib.xml”) \ “book”)
  yield (b \ “author”)(0)
  ```

  The only problem is that; Scala raises an exception when we try to extract the first author of a book that doesn’t have any author.

Sequence Expressions

An XQuery sequence is an ordered collection of zero or more “items”. A sequence containing exactly one item is a “singleton”. Sequences are never nested in XQuery; for example, combining the values 1, and (2, 3) into a single sequence results in the sequence (1, 2, 3). As you see, one way to construct a sequence in XQuery is by using the “comma operator”, which evaluates each of its operands and concatenates the resulting sequences, in order, into a single result sequence. A sequence containing zero items is called an “empty sequence”.

In Scala, `NodeSeq` class represents a sequence of nodes and contains project functions and comprehension methods. Sequence comparison are added to this class during this project.

Sequence Operators

XQuery provides the `union`, `intersect`, and `except` operators for combining sequences of nodes. Each of these operators combines two sequences, returning a result sequence in document order. A sequence of nodes that is in document order, never contains the same node twice. If an operand contains an item that is not a node, an error is raised.

The `union` operator takes two node sequences and returns a sequence with all nodes found in the two input sequences. This operator has two lexical forms: `|` and `union`. The `intersect` operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands. The `except` operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second one.

All these operators eliminate duplicate nodes from their result sequences based on node identity and the resulting sequence is returned in document order.

The `union`, `intersect`, and `except` methods from `List` class of Scala correspond to these XQuery sequence operators. Since a sequence of nodes may be treated as a list of nodes using `asList` method from `NodeSeq` class in Scala, so the sequence operators are already integrated in xml library of Scala.

Arithmetic Expressions

XQuery supports the arithmetic operators for addition (`+`), subtraction (`-`), multiplication (`*`), division (`div` and `idiv`), and modulus (`mod`), in their usual binary and unary forms. Unary operators have higher precedence than binary operators and parentheses should be used when is needed. A subtraction operator must be preceded by whitespace if it could otherwise be interpreted as part of the previous token. For example `a-b` will be interpreted as a name, but `a - b` and `a -b` will be interpreted as arithmetic expressions.

The `div` operator performs division on any numeric type. The `idiv` operator requires integer arguments, and returns an integer as a result rounding toward 0. All other arithmetic operators have their conventional meanings. If an operand of an arithmetic operator is a node, atomization is applied. For instance, the following query returns 4:

```
2 + <int>{ 2 }</int>
```

The order of operand evaluation is implementation-dependent. If an operand is an empty sequence, the result of an arithmetic operator is an empty sequence. Empty sequences in XQuery frequently operate like `nulls` in SQL or `NodeSeq.Empty` in Scala. The result of the following query is an empty sequence:

```
2 + ()
```

If the atomized operand is a sequence of length greater than one, a type error is raised. If an operand is untyped data, it is cast to a double, raising a dynamic error if the cast fails. This implicit cast is important, because a great deal of XML data is found in documents that do not use W3C XML Schema, and therefore do not have simple or complex types. Many of these documents however contain data that is to be interpreted as numeric. The prices in our bibliography are one example of this. The following query adds the first and second prices, returning the result as a double:

```
let $p := doc(“bib.xml”)//price
return $p[1] + $p[2]
```

In xml library of Scala, each node has an implicit converter from its string value to a double value, using `parseDouble()` method of `java.io.Double` class, so the arithmetic operators of `Double` class will be used, when there is an arithmetic operator between two node operands. A dynamic error will be raised if the double casting fails. The previous query in Scala language can be written:

```
val p = load(“bib.xml”) \\ “price”;
p(0) + p(1)
```

Scala’s sweet syntax allows the apply function be demonstrated in a simplify manner as represented above. As you see, this query returns nothing, and don’t have the same effect as XQuery one; because let expressions are not present in Scala.

Logical Expressions

The only logical operators in XQuery are `‘and’` and `‘or’`. The first step in evaluating a logical expression is find the “effective boolean value” of each of its operands. Once again, the order is implementation-dependent. If an operand is an empty sequence, its value is false. If an operand is a sequence whose first item is a node, its value is true. If an operand has a string type or it is untyped data, its value is false if the operand has zero length; otherwise it is true. If an operand has any numeric type, its value is false if it is numerically equal to zero; otherwise it is true. In all other cases the logical operation raises a type error.

In addition to `‘and’` and `‘or’` operators, XQuery provides a `not()` function that takes a general sequence as parameter and returns a boolean value.

In Scala, the logical operators (`&`, `|`, and `!`) have the same results for the boolean values, but there is no signification of logical operators neither for the sequences nor other atomic values.

Comparison Expressions

XQuery has several sets of comparison operators, including value comparisons (`eq, ne, lt, le, gt, ge`), general comparisons (`=, !=, <, <=, >, >=`), node comparisons (`is, <<, >>`). Value comparisons and general comparisons are closely related; in fact, each general comparison operator combines an existential quantifier with a corresponding value comparison operator.

Value Comparisons

The value comparisons compare two atomic values. If either operand is a node, atomization is used to convert it to an atomic value. If either operand is untyped, it is treated as a string. Using value comparisons, strings can only be compared to the other strings, which means that value comparisons are fairly strict about typing. Therefore, an explicit cast is needed to cast price to a decimal in the following query:

```
for $b in doc(“bib.xml”)//book
where xs:decimal($b/price) gt 100.0
return $b/title
```

In general, if the data you are querying is meant to be interpreted as typed data, but there are no types in the XML, value comparisons force your query to cast when doing comparisons; general comparisons are more loosely typed and do not require such casts. This problem does not arise if the data is meant to be interpreted as string data, or if it contains the appropriate types.

If either operand is an empty sequence, a value comparison evaluates to the empty sequence. If an operand contains more than one item, then a value comparison raises an error. For example, the following query raises an error:

```
for $b in doc(“bib.xml”)//book
where $b/author/last eq “Stevens”
return $b/title
```

The reason for the error is that many books have multiple authors, so the expression `$b/author/last` returns multiple nodes.

Although there is not any equivalent value comparison in Scala, we can use general comparison instead. The reason is explained in the following subsection.

General Comparisons

There are two significant differences between value comparisons and general comparisons: The first is that, general comparisons apply atomization to both operands, but the result of this atomization may be a sequence of atomic values. The general comparison returns true if any value on the left matches any value on the right, using the appropriate comparison. The second difference involves the treatment of untyped data; general comparison try to cast to an appropriate “required type” to make the comparison work. When a general comparison tests a

pair of atomic values and one of these values is untyped, it examines the other atomic value to determine the required type to which it casts the untyped operand:

- If the other atomic value has a numeric type, the required type is double.
- If the other atomic value is also untyped, the required type is string.
- Otherwise, the required type is the dynamic type of the other atomic value. If the cast to the required type fails, a dynamic error is raised.

These conversation rules mean that the comparisons done with general comparisons rarely need to cast when working with data that does not contain W3C XML Schema simple types. On the other hand, when working with strongly typed data, value comparisons offer greater type safety.

In the preceding query, we can use the general comparison `'='` instead of the value comparison `'eq'`, and obtain suitable result that is title of books whose are written by Stevens:

```
for $b in doc("bib.xml")//book
where $b/author/last = "Stevens"
return $b/title
```

But sometimes when an operand has more than one step, the general comparison can lead to confusing results.

The following example contains three general comparisons. The value of the first two comparisons is true, and the value of the third comparison is false. This example illustrates the fact that general comparisons are not transitive!

```
(1, 2) = (2, 3)
(2, 3) = (3, 4)
(1, 2) = (3, 4)
```

The following example contains two general comparisons, both of which are true. This example illustrates the fact that `'='` and `'!='` operators are not inverse of each other!

```
(1, 2) = (2, 3)
(1, 2) != (2, 3)
```

Usual comparisons in Scala `(==, !=, <, <=, >, >=)`are implemented only for operands with atomic values. In fact, a typical version of Scala does not support "sequence comparisons". To implement the previous query in Scala language, one should use a complicated code like:

```
for (val b <- load("bib.xml") \\ "book";
     val l <- b \ "author" \ "last";
     l.text == "Stevens";
     val t <- b \ "title")
yield t;
```

In this project, the concept of general comparisons is introduced for the `NodeSeq` class of xml library by adding some new methods (`==, !=, <, <=, >, >=`). As you know, each operator in Scala is a method of the left hand side object. So a sequence of nodes (such as `b \ “author” \ “last”`) can be compared to `Any` value (like string `“Stevens”`) in Scala using the usual operators. For instance, the previous query using our methods can be simplified to:

```
for (val b <- load(“bib.xml”) \\ “book”;
     b \ “author” \ “last” == “Stevens”;
     val t <- b \ “title”)
yield t;
```

The only problem is that comparison can not be done in the reverse way; it means we cannot compare a string in left hand side with a `NodeSeq` in right hand side, because in this case the comparison operator from String class will be used by Scala and not the general comparison from `NodeSeq` class. But, since the comparison in the reverse order is not frequently used in the programming languages, for example one used to write `x == 2` and not `2 == x`, this problem can be disregarded in this project. One other alternative is using `==(2, x)` instead of `2 == x`, for the moment.

Since value comparisons can be replaced by general comparisons, if the type error is ignored, they are not explicitly implemented in xml library of Scala. For example, the result of these two queries are equivalent, if we consider there is only one title per book; otherwise the first query results a type error while the second one results true:

```
for $b in doc(“bib.xml”)//book
where $b/title eq “Data on the Web”
return $b

for $b in doc(“bib.xml”)//book
where $b/title = “Data on the Web”
return $b
```

Node Comparisons

Node comparisons in XQuery are used to compare two nodes, by their identity or by their document order. Each operand must be either a single node or an empty sequence; otherwise a type error is raised. If either operand is an empty sequence, the result of the comparison is an empty sequence. A comparison with the `‘is’` operator is true if the two operand nodes have the same identity, and are thus the same node; otherwise it is false. For example the following comparison is false because each constructed node has its own identity:

```
<a>5</a> is <a>5</a>
```

A comparison with the `‘<<’` operator returns true if the left operand node precedes the right operand node in document order; otherwise it returns false. A

comparison with the '>>' operator returns true if the left operand node follows the right operand node in document order; otherwise it returns false.

The node comparisons are not supported by Node class of Scala. Because, node identity is not really defined for Node class. One can re-implement the hashCode() method in order to guarantee the unique node identity and document order in Scala.

Constructors

XQuery provides constructors that can create XML structures within a query. Constructors are provided for element, attribute, document, text, comment, and processing instruction nodes. Two kinds of constructors are: direct constructors and computed constructors.

In a similar way, Scala programming language also provides constructors to create XML structures through Scala code. Scala code can be combined with XML expressions to generate content dynamically.

In both languages, the direct constructor, curly braces, {}, delimit enclosed expressions, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value, using XQuery or Scala interpreter, as illustrated by the following example:

```
<example>
  <p> Here is a query in XQuery. </p>
  <eg> doc(“bib.xml”)//title <eg>
  <p> Here is the result of the query. </p>
  <eg>{ doc(“bib.xml”)//title }</eg>
</example>

<example>
  <p> Here is a query in Scala. </p>
  <eg> load(“bib.xml”) \\ “title” <eg>
  <p> Here is the result of the query. </p>
  <eg>{ load(“bib.xml”) \\ “title” }</eg>
</example>
```

The above queries might generate the following results:

```
<example>
  <p> Here is a query in XQuery. </p>
  <eg> doc(“bib.xml”)//title <eg>
  <p> Here is the result of the query. </p>
  <eg><title>Data on the Web</title></eg>
</example>

<example>
  <p> Here is a query in Scala. </p>
  <eg> load(“bib.xml”) \\ “title” <eg>
  <p> Here is the result of the query. </p>
```

```
  <eg><title>Data on the Web</title></eg>
</example>
```

Since these languages use curly braces to denote enclosed expressions, some convention is needed to denote a curly brace used as an ordinary character. For this purpose, a pair of identical curly brace characters within the content of an element or attribute are interpreted as a single curly brace character (that is, the pair "{{" represents the character "{"and the pair "}}" represents the character "}").

An alternative way to create nodes is by using a computed constructor. In XQuery an element book with a publication year as its attribute and a title and list of authors as its children can be created as below:

```
element book {
  attribute year {"2003"}
  element title {"XQuery from Experts"}
  element author {"Don Chamberlin"}
  element author {"Denise Draper"}
  element author {"Mary Fernandez"}
}
```

The same element book can be constructed in Scala using `Elem` case class:

```
Elem(null,"book",new UnprefixedAttribute("year","2003",Null),
  TopScope,
  Elem(null,"title",Null,TopScope,Text("XQuery from Experts")),
  Elem(null,"author",Null,TopScope,Text("Don Chamberlin")),
  Elem(null,"author",Null,TopScope,Text("Denise Draper")),
  Elem(null,"author",Null,TopScope,Text("Mary Fernandez"))
)
```

The interface of `Elem` class is given here:

```
Elem(prefix: String, label: String, attributes: MetaData,
scope: NamespaceBinding, child: Node*)
```

An element attribute in Scala extends from `MetaData` class and may be `Null`, prefixed or un-prefixed:

```
UnprefixedAttribute(key: String, value: String,
next: MetaData)
PrefixedAttribute(prefix: String, key: String, value:
String, next: MetaData)
```

However the attributes are usually considered as strings, but in Schema-based documents they may be typed, as well as other XML elements. So, an improved technique would be considering a type for their values and conserving them for extraction usage. In Scala we can imagine an attribute value with type `Any`. In the preceding example, the attribute `year` of element `book` is perhaps an `integer`. By importing this approach in the future versions of Scala, we will be able to visualize the comparison represented below:

```
for (val b <- load("bib.xml"); b.attribute("year") > 2000)
  yield b
```

Quantified Expressions

Some queries need to determine whether at least one item in a sequence satisfies a condition (existential quantifier), or whether every item in a sequence satisfies a condition (universal quantifier). Qualifiers sometimes make complex queries much easier to write and understand.

Both languages support the quantifiers. The `some` expression in XQuery corresponds to the `exists()` method in Scala and `every` expression corresponds to the `forall()` method in Scala.

```
some $a in doc("bib.xml")//author satisfies
($a/last = "Stevens")
```

is equivalent to:

```
(load("bib.xml") \\ "author").
exists(a => a \ "last" == "Stevens")
```

The result of a quantifier expression is boolean (`true` or `false`) in Scala, and `true()` or `false()` built-in function in XQuery.

In general, if a universal quantifier is applied to an empty sequence, it always returns true, because every item in that (empty) sequence satisfies the condition, even though there are no items.

Conditional Expressions

XQuery's conditional expressions are used in the same why as conditional expressions in other languages. Following example shows a query that uses a conditional expression to list the first two authors' names for each book a dummy name containing `"et al."` to represent any remaining authors. In XQuery both the `then` and `else` clause are required. The empty sequence `()` can be used to specify that a clause should return nothing.

```
for $b in doc("bib.xml")//book
return
  <book>
    { $b/title }
    { for $a at $i in $b/author
      where $i <= 2
      return $a }
    { if (count($b/author) > 2)
      then <author>et al.</author>
      else () }
  </book>
```

The equivalence query in Scala is:

```
for (val b <- scala.xml.load("bib.xml") \ "book") yield
<book>
  { b \ "title" }
  { (b \ "author")(0) (b \ "author")(1) }
  { if ((b \ "author").length > 2) <author>et al.</author> }
</book>
```

Conditional expressions are not supported by current version of `TypedCode` class in reflection library of Scala. So in this project, corresponding transformation rules are not represented, however as you can imagine, the transformation is quite simple.

Other Expressions

The operators represented in this section are not really introduced in this project. But, here we want to present the provided facilities in XQuery and compare them with a programming language such as Scala, and express the fact that it is roughly possible to re-implement any programming code using XQuery language even if it seems strange.

Instance Of

The `instance of` operator in XQuery is very similar to the `isInstanceOf` method from Scala `Any` class and it tests an item for a given type. For instance, the expression `3.14 instance of xs:decimal` is true and is equivalent to `(3.14).isInstanceOf[Double]` in Scala.

TypeSwitch

The `typeswitch` expression chooses an expression to evaluate base on the dynamic type of an input value. It is similar to the `SWITCH-CASE` statement in usual programming languages, but it branches based on the argument's type, not on its value. It is exactly what we can do using "pattern matching" expressions in Scala.

Cast As

The `cast as` expression occasionally is necessary to convert a value to a specified type. It creates a new value of a specified type based on an existing value. It is comparable with `asInstanceOf` method form Scala `Any` class. XQuery also provides an expression that tests whether a given value is `castable` into a specified type.

Treat As

The `treat as` expression asserts that a value has a particular type, and raises an error if it does not. It is similar to a `cast`, except that it does not change the type of the its arguments. `treat as` and `instance of` could be used together to implement the same functionality as `typeswitch`, however in general, `typeswitch` that provides better type information and do static typing is preferable.

User-Defined Functions

XQuery permits its users to create the functions, when a query becomes large and complex. It allows functions to be recursive, which is often important for processing the recursive structure of XML documents.

Variable Definitions

A variable can be define in the prolog part of a query. Such a variable is available at any point after it is declared. For instance, if access to the titles of books is used several times in a query, it can be provided in a variable definition:

```
define variable $titles { doc(“bib.xml”)//title }
```

To avoid circular references, a variable definition may not call functions that are defined prior to the variable definition.

Chapter 4

Scala to XQuery Transformer

In the previous chapter, I tried to extract the transformation rules from Scala sweet syntax to XQuery sweet syntax by introducing equivalence expressions in both languages. In this chapter, the transformation rules for the core syntax are represented. Since the abstract syntax transformation may be too complicated to read and understand, the transformation rules are only represented for the core syntax. In the next chapter some more details about abstract syntax transformation are given.

Transformation Rules

Because Scala is a pure object-oriented language, each value in Scala is an object and since it is class-based, all values are instances of a class. So the transformation rules should be defined for the fields and methods of each required class. Since our transformer in this project is a XQuery transformer, we consider only xml library of Scala. The principal class in xml library is `NodeSeq` class. So we first represent the transformation rules for the fields and methods of `NodeSeq` and its subclasses. The comparison operators (general comparisons) are also added to the `NodeSeq` class. Then, the transformation rules for sequence comprehensions are illustrated. Finally, two recently added methods in `NodeSeq` class for supporting `let` and `order by` clauses are demonstrated.

The transformation rules are represented using “inference rules”. In this representation, expressions before the horizontal bar illustrate the preconditions and Scala core syntax (in sweet representation), and following expressions illustrate equivalence code in XQuery core syntax. Suppose `T` is a one-to-one transformation function which takes an Scala code and returns corresponding XQuery code. The expression $e \in E$ describes the fact that expression `e` has a type `E`. If `e` is an expression of type `NodeSeq` in Scala; then `T(e)` is an expression of type “sequence” in XQuery.

NodeSeq Class

Transformation of projection functions is almost straightforward. The only point is that, if the expression at the left hand side is an XML document load, the transformation should add the “document node”. Because XPath expressions in XQuery have an explicit “document node”. As a substitute, one can use a double

slash (//), always after a document load, to skip the “document node” automatically (see third rule below).

```
e ∈ NodeSeq   e’ ∈ String
e \ e’
_____________________________________
T(e)/T(e’)


e ∈ NodeSeq   e’ ∈ String
e \\ e’
_____________________________________
T(e)//T(e’)


e = load(s’)   s’, e’ ∈ String
e \ e’
_____________________________________
T(e)//T(e’)
```

Scala provides an `apply` function to access the elements of a sequence. XPath expressions offer a syntax which is similar to access the elements of an array in typical programming languages (position of element in the sequence enclosed by square braces, []). But, as you remember from previous chapter, I proposed to employ the positional function `at` within for clauses, to simplify our XQuery abstract syntax.

```
e ∈ NodeSeq   i ∈ int
e (i)
_____________________________________
for $x at $j in T(e)
where $j = i+1
return $x
```

The `filter` method, and `apply` method with an anonymous function as argument `(e(x => p))`, in `NodeSeq` class have the same signification so the transformation rule is identical:

```
e ∈ NodeSeq   x ∈ Node   p ∈ Boolean
e filter (x => p)
_____________________________________
for $x in T(e)
where T(p)
return $x
```

`find` method in Scala returns the first element of an `Iterable` object that satisfies a condition, if any; otherwise it returns `None`.

```
e ∈ Iterable   x ∈ Node  p ∈ Boolean
e find (x => p)
_____________________________________
for $y at $i in
  for $x in T(e)
  where T(p)
  return $x
where $i = 1
return $y
```

As you can see, `map` and `flatMap` functions have the same transformation rules:

```
e ∈ NodeSeq    x, x’ ∈ Node
e map (x => x’)
_____________________________________
for $x in T(e)
return T(x’)
```

```
e, e’ ∈ NodeSeq    x ∈ Node
e flatMap (x => e’)
_____________________________________
for $x in T(e)
return T(e’)
```

Quantified function from Scala language can be transformed to the similar expressions in XQuery:

```
e ∈ Iterable   x ∈ Node  p ∈ Boolean
e exists (x => p)
_____________________________________
some $x in T(e) satisfies (T(p))
```

```
e ∈ Iterable   x ∈ Node  p ∈ Boolean
e forall (x => p)
_____________________________________
every $x in T(e) satisfies (T(p))
```

Concatenation of two sequences in Scala can be done using `concat()` method. In XQuery, concatenation is done using “comma separator”:

```
e, e’ ∈ Iterable
e concat e’
_____________________________________
(T(e), T(e’))
```

One possible transformation rule is represented here to transform `sameElements` method using quantified expressions:

```
e, e' ∈ Iterable
e sameElements e'
_____________________________________
every $x in T(e) satisfies
(some $y in T(e') satisfies ($x = $y))
```

There exists some XQuery built-in functions for methods `length` and `text`:

```
e ∈ NodeSeq
e length
_____________________________________
count(T(e))
```

```
e ∈ NodeSeq
e text
_____________________________________
string(T(e))
```

In this part, some special transformations are represented (that are specified by *). In fact, there is not any correspondence for these Scala functions in XQuery and I transformed them differently as explained for each case.

The method `foreach` from `Iterable` class which applies a function to all its elements and returns nothing, has not equivalent expression in XQuery. So we are not able to process a `for-do` loop using XQuery processor. However a `for-do` statement in Scala may contain the other parts transformable to XQuery code. For example, following query can be partially processed by XQuery and Scala interpreter:

```
for (val b <- load("bib.xml") \ "book";
     b \\ "last" == "Stevens")
  Console.println(b \ "title")
```

This code is equivalent to:

```
(load("bib.xml") \ "book").filter(b => b \\ "last" == "Ste-
vens").foreach(b => Console.println(b \ "title"))
```

And, `filter` can be transformed to XQuery code, as you know. So we only transform the expression behind `foreach` method to XQuery code. Then, an Scala code is constructed to apply a `foreach` method to the evaluated results by XQuery processor. Finally, Scala interpreter, evaluates the final results:

```
    e ∈ Iterable   x ∈ Node   u ∈ Unit
    e foreach (x => u)
    ──────────────────────────────────
*   T(e)
```

Since `NodeSeq` and `Seq[Node]` have the same signification, they are both a sequence of nodes in XQuery, the methods that convert these two types to each other, can not be transformed in XQuery and they should be added later by Scala interpreter:

```
    e ∈ Seq[Node]
    NodeSeq.fromSeq(e)
    ──────────────────────────────────
*   T(e)
```

```
    e ∈ Seq[Node]
    NodeSeq.view(e)
    ──────────────────────────────────
*   T(e)
```

The `conv` method is added to the `NodeSeq` class in order to expand the arithmetic operators. It takes the text format of a `NodeSq` and casts it to a double value if it is castable and it contains only one single node; otherwise an exception is raised.

```
    e ∈ NodeSeq
    NodeSeq.conv(e)
    ──────────────────────────────────
*   T(e)
```

In fact, by adding this implicit function in `NodeSeq` class, Scala can support the queries like:

```
for (val b <- load("bib.xml") \ "book";
    (b \ "price") / 2 > 50) yield b
```

In XQuery there is only one type which describes sequence of items. While in Scala there are several types which represent a collection such as: `Seq`, `List`, `Iterator`, … So the methods that convert these types to each other, can not be transformed in XQuery, and they should be processed by Scala interpreter. Obviously, the methods that are based on Scala `Lists` such as `foldLeft`, `foldRight`, `/:`, and `:\` are not transformable in XQuery.

Each method in {asList, toList, elements, theSeq, toString()} will be transformed identically and then the appropriate method will be added by Scala interpreter to obtained results:

```
    e ∈ NodeSeq
    e asList
    ──────────────────────────────────────
*   T(e)
```

General comparisons

General comparisons are implemented for NodeSeq class and its subclasses. In this project, the structural and general equalities are supposed to be the same. This suggestion is not actually exact, and once two different syntaxes are supplied for them, we can separate their definitions.

For each Scala operation in {==, !=, <, <=, >, >=} an XQuery operation exists in {=, !=, <, <=, >, >=}. The transformation rule for general equality is represented here. For the others, the transformation is similar.

```
e ∈ NodeSeq   x ∈ Any
e == x
──────────────────────────────────────
T(e) = T(x)
```

Let Method

As it was explained before, there was no equivalence for let clauses in Scala. I added a let method in NodeSeq class which has the same signification. The let clause gives a name for a NodeSeq and returns another NodeSeq.

```
x, e, e' ∈ NodeSeq
e let (x => e')
──────────────────────────────────────
let $x := T(e)
return T(e')
```

OrderBy Method

In a same way, I defined a method orderBy() similar to the XQuery order by clauses. An orderBy() method, takes a NodeSeq and changes the order of this sequence by a given key and returns another sequence.

```
e, e' ∈ NodeSeq   x ∈ Node
s ∈ {ascending, descending}
e orderBy (x => e', "s")
```

```
for $x in T(e)
order by T(e') s
return $x
```

Node Class

The transformation rules for all methods that `Node` class inherits from `NodeSeq` class or other classes are the same. Only most useful and important methods are transformed from this class:

```
n ∈ Node   key ∈ String
n attribute (key)
```

```
T(n)/@T(key)
```

```
n ∈ Node   uri, key ∈ String
n attribute (uri, key)
```

```
T(n)/@T(uri):T(key)
```

```
n ∈ Node
n attributes
```

```
T(n)/@*
```

```
n ∈ Node
n child
```

```
T(n)/*
```

```
n ∈ Node
n descendant
```

```
T(n)/descendant::*
```

```
n ∈ Node
n descendant_or_self
```

```
T(n)/descendant-or-self::*
```

As you observed, the `Node` class implements some methods to access the other nodes in document order such as `child`, `descendant`, ... But, it forgets

about some other important methods such as `parent`, `ascendant` and so on. My personal idea is that if we are interested to introduce the document order and locating nodes in Scala language, we should respect the entire standards from W3C documentation, otherwise the user will be surprised!

Elem Class

Transformation rules for constructing XML elements are given here. If an XML element doesn't contain Scala code in its attribute value nor its content the transformation is unique (`T(e) = e`). Otherwise, the transformation should be applied on attributes and children elements.

```
prefix, label ∈ String    attributes ∈ MetaData
scope ∈ NamespaceBinding    child ∈ Node*
Elem(prefix, label, attributes, scope, child)
_____________________________________________
<prefix:label T(attributes)>
  T(child)
</prefix:label>


e ∈ Elem     attrs ∈ MetaData
e % (attrs)
_____________________________________________
<e.prefix:e.label T(attrs)>
  e.child
</e.prefix:e.label>
```

MetaData Class

Transformation rules for constructing attributes are given here. An attribute is usually constructed inside an element. If there is no element an auxiliary element should be constructed to include the attribute, but it is not the case in current Scala version. An attribute in Scala can be `Null`, prefixed or unprefixed. Transformation of `Null` attribute is nothing (`T(Null) = ()`).

```
key, value ∈ String    next ∈ MetaData
UnprefixedAttribute(key, value, next)
_____________________________________
T(key) = T(value) T(next)


prefix, key, value ∈ String    next ∈ MetaData
PrefixedAttribute(prefix, key, value, next)
___________________________________________
T(prefix):T(key) = T(value) T(next)
```

Input Functions

Most usual input function in Scala is `load` from `XML` object. There are some other functions for loading an XML document with processing instructions and comments in Scala that are not transformed to XQuery and are not represented in this report.

```
s ∈ String
load(s)
──────────────────────────────────────
doc(s)
```

Examples

In this section, the sequence comprehensions of xml library are transformed to FLWOR expressions using the transformation rules given before. Since a sequence comprehension is composed of `map`, `filter`, and `flatMap` functions, and we have already transformed such functions, so the rules represented here are already well-known. In the case, when we have a sequence of generators followed by filters, each generator can be transformed separately.

In this representation the first expression is in Scala sweet syntax, the second expression is in Scala core syntax, the third expression is in XQuery core syntax, and the last expression, if any, is in XQuery sweet syntax. However the sequence comprehensions only use some special order of these functions (for example a `filter` followed by a `map` function never happens, similarly, a `flatMap` of `filter` function), but all possible orders can be transformed to XQuery language without any difficulty.

```
x, e′ ∈ Node    e ∈ NodeSeq
for (val x <- e) yield e′
(or) ──────────────────────────────────────
e map (x => e′)
──────────────────────────────────────
for $x in T(e)
return T(e′)
```

```
x, e′ ∈ Node    f ∈ Node => Boolean    e ∈ NodeSeq
for (val x <- e; f(x)) yield e′
(or) ──────────────────────────────────────
(e filter (x => f(x))) map (x => e′)
──────────────────────────────────────
for $x in
  for $x in T(e)
  where T(f(x))
  return $x
return T(e′)
(or) ──────────────────────────────────────
```

```
for $x in T(e)
where T(f(x))
return T(e’)
```

```
x, y, e’’ ∈ Node    e, e’ ∈ NodeSeq
for (val x <- e; val y <- e’) yield e’’
(or) ——————————————————————————————
e flatMap (x => e’ map (y => e’’))
——————————————————————————————————
for $x in T(e)
return
  for $y in T(e’)
  return T(e’’)
(or) ——————————————————————————————
for $x in T(e), $y in T(e’)
return T(e’’)
```

```
x, e’ ∈ Node    f ∈ Node => Boolean    e ∈ NodeSeq
s: sequence of generators
for (val x <- e; f(x); s) yield e’
(or) ——————————————————————————————
(e filter (x => f(x))) flatMap (x => … map (z => e’))
——————————————————————————————————
for $x in
  for $x in T(e)
  where T(f(x))
  return $x
return
  for T(s)
  return T(e’)
(or) ——————————————————————————————
for $x in T(e), T(s)
where T(f(x))
return T(e’)
```

Using the definition of `let()` method, a `let` of `filter` or `map`, and `flatMap` of `let` expressions are allowed.

```
y, e, e’ ∈ NodeSeq    x ∈ Node    f ∈ Node => Boolean
(e filter (x => f(x))) let (y => e’)
————————————————————————————————————
let $y :=
  for $x in T(e)
  where T(f(x))
  return $x
return T(e’)
```

```
x, e, e’ ∈ NodeSeq    y, e’’ ∈ Node
e let (x => e’ map (y => e’’))
————————————————————————————————————
let $x := T(e)
return
  for $y in T(e’)
  return T(e’’)
```

```
(or) ————————————————————————————————————
let $x := T(e)
for $y in T(e’)
return T(e’’)
```

```
x, e, e’ ∈ NodeSeq    y ∈ Node    f ∈ Node => Boolean
e let (x => e’ filter (y => f(y)))
————————————————————————————————————————————
let $x := T(e)
return
  for $y in T(e’)
  where T(f(y))
  return $y
(or) ————————————————————————————————————
let $x := T(e)
for $y in T(e’)
where T(f(y))
return $y
```

```
y, e, e’, e’’ ∈ NodeSeq    x ∈ Node
e flatMap (x => e’ let (y => e’’))
————————————————————————————————————————————
for $x in T(e)
return
  let $y := T(e’)
  return T(e’’)
(or) ————————————————————————————————————
for $x in T(e)
let $y := T(e’)
return T(e’’)
```

An `orderBy` method can be followed by `map` or `flatMap`:

```
e, e’ ∈ NodeSeq    x, y, e’’ ∈ Node
s ∈ {ascending, descending}
(e orderBy (x => e’, “s”)) map (y => e’’)
————————————————————————————————————————————
for $y in
 for $x in T(e)
 order by T(e’) s
 return $x
return T(e’’)
(or) ————————————————————————————————————
for $x in T(e)
order by T(e’) s
return T(e’’)
```

```
e, e’, e’’ ∈ NodeSeq    x, y ∈ Node
s ∈ {ascending, descending}
(e orderBy (x => e’, “s”)) flatMap (y => e’’)
————————————————————————————————————————————
```

```
for $y in
  for $x in T(e)
  order by T(e’) s
  return $x
return T(e’’)
(or) ——————————————————————————————
for $x in T(e)
order by T(e’) s
return T(e’’)
```

Finally a FWOR expression can be written as demonstrated here:

```
e, e’ ∈ NodeSeq    x, y ∈ Node    f: Node => Boolean
s ∈ {ascending, descending}
(e filter (x => f(x))) orderBy (y => e’, “s”)
——————————————————————————————————————————
for $y in
  for $x in T(e)
  where T(f(x))
  return $x
order by T(e’) s
return $y
(or) ——————————————————————————————
for $x in T(e)
where T(f(x))
order by T(e’) s
return $x
```

Chapter 5

Scala Query Shipping Implementation

In this chapter, some implementation techniques are explained and more clarified. This project includes an xquery library in Scala programming language. Also, some modifications are applied to the existing xml and reflection library.

As mentioned before, the Scala code that contains an XML query will be processed by an XQuery processor and it should be marked by a special type called `TypedCode` from reflection library of Scala. A `TypedCode` is parsed and type checked by Scala parser and type checker. But, Scala compiler does not process the code marked as a `TypedCode` so does not generate the byte codes; instead, it generates a code that will be construct an abstract tree during execution-time. This abstract tree which contains the useful information from Scala query is transformed to the XQuery code. The transformation is a tree to tree transformation. So for a given Scala AST, an equivalence XQuery AST is generated, and then the XQuery source is created using the XQuery pretty printer.

However, in the previous chapter, the transformation rules are represented based on the core syntax, for simplification reasons, the implemented transformer in this project, transforms an Scala abstract syntax to corresponding XQuery abstract syntax.

XQuery AST and Pretty Printer

The xquery library defines the XQuery tokens and sets up an abstract syntax tree and pretty printer for XQuery language. The `Tokens` object in xquery library, initiates all required tokens to construct XQuery source code from XQuery abstract syntax. In fact, it contains all operators and some built-in functions of XQuery language. The `Tree` object defines an abstract syntax tree and the `Printer` object, defines a pretty printer for XQuery language.

In this project, I tried to simplify the XQuery EBNF and specify a compact abstract syntax tree. Since XQuery expressions and XML elements are closely combined, XQuery grammar must support the XML expressions. On the other hand, Scala syntax supports XML expressions within Scala code. So our proposal is to reuse these XML expressions inside XQuery expressions. In order to share the XML expressions between Scala language and XQuery language, I suppose that every XQuery expression is an XML node with a name specified by the expression and with a content as expression itself. For instance, the following for

clause in XQuery is represented only using XML expressions. In this representation, only the useful information of an XQuery is stored, so we obtain a compressed syntax comparing with XQueryX syntax.

```
for $b in doc(“bib.xml”)//book
where $b/price < 100
return $b/title

<for>
  <var>b</var>
  <dslash>
    <doc><literal>bib.xml</literal>
    </doc><ident>book</ident>
  </dslash>
  <lthan>
    <slash><var>b</var>
    <ident>price<ident></slash>
    <literal>100</literal>
  </lthan>
  <slash>
    <var>b</var><ident>title</title>
  </slash>
</for>
```

So, each node in XQuery abstract syntax tree contains the useful information of its children and extends from the `Node` class of Scala:

```
case class For(v: Var, pos: Option[Var], domain: Node,
where: Option[Node], order: Option[NodeSeq], return: Node)
extends Node

case class Var(ident: String) extends Node
```

Also, each case class overrides the `label`, `child`, and `text` method of `Node` class. The `print` method from `Printer` is used to pretty printing of an XQuery expression. Using this approach, we provide a simple, compact, and exact abstract tree for XQuery language. A query `Q` has an abstract syntax as represented in table 2:

Table 2: XQuery AST

```
Q = Sequence {Q}
Q = For V [V] Q [Q] [Q O1] Q
Q = Let V Q [Q] [Q O1] Q
Q = Some V Q Q
Q = Every V Q Q
V = Var ident
Q = Op O2 Q [Q]
Q = Literal value
Q = Ident ident
Q = Attribute ident
Q = FunCall F {Q}
```

In this abstract syntax, O1 is a set of order by operations: `ascending` and `descending`. O2 is a set of unary and binary operators. And F is a set of built-in functions defined by XQuery language. Using this abstract syntax, the previous query is constructed by:

```
For(
  Var(“b”),
  None,
  Op(DSLASH, FunCall(DOC, Literal(“bib.xml”)),
     Ident(“book”)),
  Some(OP(LTHAN, Op(SLASH, Var(“b”), Ident(“price”)),
       Literal(100))),
  None,
  Op(SLASH, Var(“b”), Ident(“title”))
)
```

Some other simplifications that are considered in this project, are listed here: Although a FLWOR expression is a sequence of for or let clauses followed by some optional clauses and a required return clause, in our syntax only one for clause or let clause is allowed per return clause. In other words, the first definition is only used by XQuery user as a sweet syntax, while the second one is used by XQuery processor in the core syntax. For example, these three expressions are equivalent in XQuery language; but generating the last one is straightforward from abstract syntax:

```
for $x in e, $y in e’, $z in e’’
return e’’’

for $x in e
for $y in e’
for $z in e’’
return e’’’

for $x in e
return
  for $y in e’
  return
    for $z in e’’
    return e’’’
```

The same simplification is applied for the quantifiers. For example, the two codes represented below are equal; but the last definition is used in our abstract syntax:

```
some $x in e, $y in e’, $z in e’’ satisfies (p)

some $x in e satisfies (
  some $y in e’ satisfies (
    some $z in e’’ satisfies (p)))
```

The other simplification in FLWOR expressions, which is implied by Scala, is that considering only one `order by` specification per expression. In fact, using Scala `orderBy()` method that was defined in this project, we can not apply two

order for a given sequence. For instance, the following query can not be generated by current version of Scala:

```
for $a in doc(“bib.xml”)//author
order by $a/last descending $a/fist descending
return $a
```

This query sorts the authors first in reverse order by the last name, then if there are two authors with the same last name, sorts them in reverse order by their first name. Finally, it returns the alphabetically sorted authors.

An XML element in Scala is the same as an XML element in XQuery. It may contain an attribute or a content which is Scala code or XQuery expression itself. In this case, the non-XML expression in attribute value or element content should be enclosed by curly braces: {}. Even if transformation of an XML element in Scala code to an XML element in XQuery code is straightforward, but the Scala code in attribute value or element content should be transformed to the equivalence code in XQuery.

However, there are two different ways to create an XML element in Scala and XQuery: direct and computed construction. But, only the first one is used in XQuery abstract syntax. It means, either the `Elem` constructor or XML literals in Scala are transformed to the XML literals in XQuery. Only for some simplification and printing reasons, another case class `ElemNode` is added to the XQuery abstract tree. This case class has the same number of arguments with the same signification as the `Elem` class. Only the attributes and children are a little bit different for simplifying the transformation. Also, the text method is implemented differently to print properly the XML elements inside XQuery expressions, and vise versa.

XQuery Transformer

In this project and maybe in many other similar projects, a specified Scala code which is processed using another processor or interpreter during execution-time, is defined as a `TypedCode` variable. The `TypedCode` class of reflection library of Scala only supports a subset of Scala code. It means we can not write every Scala valid code inside a `TypedCode` variable. For the moment, we added only the necessary expressions for this project. For instance, we can not declare a new variable or function inside a `TypedCode` variable. Moreover, conditional expressions, pattern matching expressions, and try-catch expressions, which are very useful in Scala usual code, are not added to the `TypedCode` class. However, some modern query languages such as Oracle and XQuery support these types of expressions, but they are not very essential in formal query languages.

The `Transform` object from xquery library, defines the `trans()` method which takes a `TypedCode` variable with a given type argument T and returns another `TypedCode` with the same type T. The argument contains the Scala code that should be transformed and return value contains the Scala code to invoke XQuery processor on the XQuery generated code. In order to transform the following Scala code to corresponding XQuery code, the `trans()` method is represented as shown here. Variable `tc` stands for `typedcode` and `ttc` for transformed `typedcode`. Both `tc` and `ttc` have the same type.

```
val tc:TypedCode[NodeSeq] =
  for (val b <- load("bib.xml") \ "book";
       b \ "price" < 100) yield b

val ttc:TypedCode[NodeSeq] = Transform.trans(tc)
```

The generated `TypedCode` as the result of `trans()` method, after transformation, should contain an Scala code which is very similar to:

```
load(new java.io.StringReader(
GalaxTest.run("for $b in doc(\"bib.xml\")//book where
$b/price < 100 return $b")))
```

Finally, to evaluate the query and obtain the results, an Scala interpreter should be invoked using `Interpreter.interpret(ttc)` which should have the same result as `Interpreter.interpret(tc)`. For the moment, such an Scala interpreter does not exist. Once an Scala interpreter is implemented for `TypedCode`, the final results of query processing will be attained.

The scenario of `trans()` method is explained here using previous example: First of all, the code inside variable `tc` typed by `TypedCode` is not processed by Scala compiler, instead, during execution-time an abstract syntax tree from reflection library is constructed. The method `trans()` from Transform object transforms this generated Scala AST to corresponding XQuery AST. Then, XQuery pretty printer generates the source code from XQuery AST using `print` method. Afterward, an Scala code should be generated to invoke an XQuery processor (like Galax implementation). Finally, the results of XQuery processor should be loaded in a well-known type for Scala such as `NodeSeq`.

Galax implementation returns the results inside an XML document on the standard output and the method `run()` in `GalaxTest` object serializes the results to an string. This is why, a java string reader is used to load the resulted XML document. So, `GalaxTest` object in xquery library specifies a `run()` method which takes an XQuery source code as string and serializes the evaluated results to another string.

One question is that, how the external variables and methods are defined inside a `TypedCode` variable? It means, how we can transform a variable or method

inside a TypedCode which is declared before. In this project, since we will use an Scala interpreter to evaluate the results, later after the transformation, we can delegate this responsibility to the Scala interpreter. So, Scala interpreter is responsible to replace variable and method definitions using the conception of reflection.

For instance, suppose following Scala code:

```
val x:double = 100
…
val tc:TypedCode[NodeSeq] =
  for (val b <- load(“bib.xml”) \ “book”;
       b \ “price” < x) yield b
…
val ttc:TypedCode[NodeSeq] = Transform.trans(tc)
```

However, variable x is well-defined inside the TypedCode, and there is no compile error, but variable x is not defined for XQuery processor. And the transformer can not transform it uniformly. In other words, the usual transformation is not allowed in this case and transformer should not generate a code like:

```
load(new java.io.StringReader(GalaxTest.run(“for $b in
doc(\“bib.xml\”)//book where $b/price < x return $b”)))
```

Our solution is to generate following code instead of the code above:

```
load(new java.io.StringReader(
GalaxTest.run(“for $b in doc(\“bib.xml\”)//book where
$b/price < ”
+ x.toString() +
“ return $b”)))
```

When this code is evaluated by Scala interpreter, variable x will be replaced by its value and then method run will be invoked with an well-known string. In order to be able to do this transformation, an auxiliary node is added in XQuery abstract syntax tree called ScalaVar. It contains the Scala code for variable, and during printing, a new Scala code will be generated to concatenate XQuery source code with Scala code.

Using this technique, an external variable can be used wherever inside a TypedCode and transformation can continue transforming properly without any difficulty. But the external methods depend on their positions in the query and their arguments should be treated differently. Usually, the external methods that contain literal arguments can be transformed using the same approach as external variables. See following example for more details:

```
def square(x:double):double = { x * x }
…
val tc:TypedCode[NodeSeq] =
  for (val b <- load(“bib.xml”) \ “book”;
       b \ “price” < square(10)) yield b
…
```

```
val ttc:TypedCode[NodeSeq] = Transform.trans(tc)
```

This query can be transformed in this way:

```
load(new java.io.StringReader(
GalaxTest.run("for $b in doc(\"bib.xml\")//book where
$b/price < "
+ square(10).toString() +
" return $b")))
```

The problem produces when an external method contains an unevaluated statement. In this case, if the unevaluated statement doesn't have any dependant variable, it can be transformed and process by XQuery code, otherwise, the transformation would be impossible and Scala interpreter should evaluate the entire query. The following example represents two different cases which use an external sum function; the first one is transformable but the second one is not:

```
def sum(nodes:NodeSeq):double = {
  var s:double = 0
  for (val node <- nodes)
    s = s + node
  s
}
…
val tc:TypedCode[double] =
  sum(load("bib.xml") \ "book" \ "price")
…
val ttc:TypedCode[double] = Transform.trans(tc)
```

The suggested result of transformation is shown here:

```
sum(load(new java.io.StringReader(
GalaxTest.run("doc(\"bib.xml\")//book/price"))))
```

But, there is not any proposition for transforming this query:

```
val tc:TypedCode[NodeSeq] =
  for (val b <- load("bib.xml") \ "book";
  sum(b \ "price") < 200) yield b
```

Because, variable `b` is undefined inside the method `sum` and Scala interpreter is unable to evaluate the expression `b \ "price"` without any information of variable `b`. The only solution can be replacing the definition of the external function inside the `TypedCode` and then transforming the entire code to the XQuery code.

Abstract Syntax Transformation

As you know, `TypedCode` class in reflection library has an abstract syntax tree which is a subset of Scala abstract syntax. On the other hand, the `trans()` method of `Transform` object in xquery library transforms such an abstract syntax

to an XQuery abstract syntax. So it is necessary to be familiar with this syntax. This section represents the reflection abstract syntax and explains the abstract syntax transformation in more details. The abstract syntax for TypedCode class is given in the following table (table 3):

Table 3: Scala TypedCode AST

```
T = TypedCode C
C = Ident S
C = Select C S
C = Literal value
C = Apply C {C}
C = TypeApply C {C}
C = Function {S} C
C = This S
C = Block {C} C
C = New C
```

As you see, a `TypedCode T` can contain only a limited code (`C`) form Scala AST. For instance, it can not hold an `IF-THEN-ELSE` statement nor `MATCH-CASE` statement. Moreover, we can not define a new variable or function inside a `TypedCode` (For example, `val x = 2` or `def f(x:int):int = { x }` are not legal inside a `Typedcode`). But, the provided code is quite enough for implementing XML queries in Scala `TypedCode`. Also, `TypedCode` has a type which represents the type of enclosed code. For the moment, `TypedCode` has a function type (anonymous function `()=>SomeType`). So every code inside a `TypedCode` is a `Function` with `empty List (Nil)` as parameter and some codes as its body. The body can be one of the codes mentioned in the above table.

The abstract syntax for `Symbol(S)` and `Type(T)` classes are represented here (see table 4 and table 5):

Table 4: Scala Symbol AST

```
S = Class name
S = Method name T
S = Field name T
S = TypeField name T
S = LocalValue S name T
S = LocalMethod S name T
S = NoSymbol
S = RootSymbol
```

In this project, since transformation doesn’t consider the types, only `NoType` is used form `Type` class.

Table 5: Scala Type AST

```
T = NoPrefix
T = NoType
T = NamedType name
T = PrefixedType T S
T = SingleType T S
T = ThisType S
T = AppliedType T {T}
T = TypeBounds T T
T = MethodType {T} T
T = PolyType {S} {T, T} T
T = ImplicitMethodType {T} T
```

However the combination of these syntaxes can produce a vast number of Scala code, only few of them are used in practice. In fact, there are limited number rules to create Scala abstract syntax from Scala core syntax. Some of these rules are explained here:

In Scala abstract syntax, the code to invoke a filed of an object or class has the following format. Similarly, invoking a method without any argument has the same syntax:

```
Select(c:Code, Method(m))
```

Using this syntax, the field or the method `m` is invoked on the object `c` and it is equivalent to the dot notation `c.m` in Scala core syntax. Corresponding abstract syntax to invoke a method `m` with arguments `args` on `c` is represented below:

```
Apply(Select(c, Method(m)), args)
```

The equal core syntax is: `c.m(args)`.

An argument of a method in Scala can be a simple code such as `Literal` or `Ident` or more complicated code such as anonymous `Function`. In the second case, some local variables are defined in its parameters and a code is defined in its body.

Using these two major rules for Scala abstract syntax, the transformation rules form Scala abstract syntax to XQuery abstract syntax can be simplified and even done automatically.

For example, suppose variable `b` is a book in our bibliography. The method `child` from `Node` class can be called on `b` like: `b.child`. The abstract syntax for this code is given here:

```
Select(Ident(LocalValue(“b”)), Method(“child”))
```

In this representation, we overlooked the type and owner of variable `b` and the type of method `child`. This information does not influence the transformation rules. As you remember, `b.child` in Scala is equivalent to `b/*` in XQuery. So the `trans()` function transforms the above Scala abstract syntax into a `BinOp` node in XQuery abstract syntax:

```
BinOp(SLASH, Var(“b”), Ident(“*”))
```

Another example represents how a `map` method is transformed to a FLWOR expression. Suppose variable `biblio` contains all books from our bibliography. Following code only yields all of the books from `biblio`:

```
biblio.map(b => b)
```

In abstract syntax, this code is equivalent to:

```
Apply(Select(Ident(LocalValue(“biblio”)), Method(“map”)),
      Function(Ident(LocalValue(“b”)),
               Ident(LocalValue(“b”))))
```

Corresponding XQuery abstract syntax after transformation is:

```
For(Var(“b”), None, Var(“biblio”), None, None, Var(“b”))
```

In other words, an Scala abstract syntax like:

```
Apply(Select(c:Code, Method(“map”)),
      Function(x:Symbol, y:Code))
```

is equivalent to a `For` node in XQuery:

```
For(x, None, T(c), None, None, T(y))
```

In this transformation both `c` and `y` (Scala Code) should be transformed and replaced by their equivalent XQuery code.

Chapter 6

Future Works

First of all, I would like to present some enrichments to the Sequence Comprehensions. In this project, I added some original lambda functions similar to the let and order by clauses from XQuery language in xml library of Scala. These functions can be imported to the Scala sweet syntax to generalize existing sequence comprehensions. In fact using these functions a for-comprehension will be capable to do exactly what we expect from FLOWR expressions. Moreover, some new concepts can be introduced in Scala sequence structure, such as sequence comparisons and sequence arithmetic and logic operations. In this project, sequence comparisons are only added into the xml library.

Some enhancements are proposed for xml library of Scala:

- **XML QNames:** In the current version of xml library in Scala, the famous XML QNames (name of XML elements, attributes,…) are considered as strings. If this type can be replaced by another literal type such as QName, then XML and XPath representation in Scala will be more elegant.
- **XML Attributes:** In Scala, XML attributes are treated differently form XML nodes, while in almost all data models proposed for XML this is not the case. An XML attribute like an XML element has a QName and a value similar to the element content that may hold any type and not only string. Moreover, an XML attribute without any XML element is not meaningful and a temporary element should be constructed to enclose the attribute.
- **Sequence Comprehensions:** The usual for-comprehensions are composed from an optional `filter` function followed by a required `map` or `flatMap` function. In the case of node sequences, `flatMap` function can cover the whole functionality of `map` function (as you know each single node is a sequence of only one node). Furthermore, by constructing a for-comprehension based on `filter` and `flatMap` functions for `NodeSeq` class, the user can expect the same result as FLWOR expressions. In fact, `yield` statement will collect the sequences in the same way as `return` clause.
- **Lambda Functions:** Other problem related to `map` and `flatMap` functions in `NodeSeq` class is that, their return values should be necessarily an Scala node or a sequence of Scala nodes. This constraint implies many limitations because we are not able to yield the strings, numeric values and so on. An ideal solution would be authorizing them to yield `Any`. In fact, `map`

yields `Any` and `flatMap` yields `Seq[Any]`. But I am not sure if this solution could be realistic.

- **Path Expressions:** The document order and XPath expressions are not entirely supported by xml library of Scala. There is only a limited access from one node to the other nodes in a document. For instance, there is not any function to access the parent of a given node.
- **Node Identity:** More important, Scala doesn't generate a unique identity for a created node. In fact, the node identity should be created either when an XML document is loaded or when a node is created directly. Once this feature is supported, the node comparisons can also be introduced in Scala.

Finally, some modifications that improve this project are represented:

- **Scala TypedCode Extension:** The Scala code that can be transformed to XQuery code should be extended; by adding the useful classes in `Code` class of reflection library. Current version supports only identifiers, literals, method selection, argument apply, functions, blocks, and new instance creation. Other expressions that can be added and are supported by XQuery language are: conditional expressions, pattern matching, new variable and method definition. The XML literals are supported partially; for instance, XML attributes can not be generated.
- **XQuery Syntax Extension:** XQuery tokens and abstract syntax tree can be completed by adding the other expressions. Once there is a correspondence between the abstract syntax of Scala and the abstract syntax of XQuery, the transformation rule can also be added.
- **TypedCode Limitations:** The parameter type of a `TypedCode` must accept any Scala legal type, instead of anonymous functions. Other limitation is that the variables with a `TypedCode` type can not be defined inside the other functions; for instance, inside the main function. Another problem of `TypedCode` class is that, we can not define two polymorphic methods with a `TypedCode` argument and different parameter types, because Scala recognizes a double function definition. This problem in this project prevents differently transforming of different `TypedCodes`. For instance, a `TypedCode` that contains a double should be transformed differently from a string one or node one.
- **XQuery library Extension:** Some aggregation functions and `distinct-values()` function are already added in an auxiliary library and are accessible for Scala programmers. Other required functions from different query languages can also be added in the same manner to improve Scala querying capabilities.
- **Scala Interpreter:** The next necessary step would be certainly implementing an Scala interpreter for evaluating `TypedCode` class.
- **Relational Databases:** Finally, the other similar project is defining and transforming Scala semi-structured queries for relational database interactions.

Conclusion

Scala programming language evolution is extremely fast. All powerful existing functionalities of Scala as well as all recently incorporated features make it easier to exploit academic, research and development areas. Comparing with other languages, (such as XML API of Java or XLinq project of .Net [9]) Scala's XML library is finely adapted to the standards proposed by W3C Consortium, and is simply flexible to provide additional norms related to this technology. In addition, this project includes some useful methods like sequence comparisons and sequence ordering into this library to increase querying capabilities.

This project combines both the provided facilities from XML library and the integrated query features of Scala to realize an XML query language, then improves the performance of the XML query processing in Scala using query shipping. One of our objectives in this project was reducing transfer-time and memory-consumption of XML data in Scala application.

This project proposed the transformation rules from Scala core syntax to XQuery core syntax. Only Scala's XML library and relevant expressions are supported in this transformation. In this project a compact and exact abstract syntax tree for XQuery language, based on XQuery EBNF, was introduced. Only essential expressions from XQuery are considered in this project. However, the proposed abstract syntax is fairly extensible and can be completed later as much as required. Due to the generality of the Scala's XML expressions, these expressions are shared between Scala and XQuery languages.

Since there is not an standard implementation for XQuery language, several existing XQuery processors such as Galax, eXist, and Saxon were analyzed in this project. We preferred Galax implementation for XML query processing because it provides useful APIs and for some simplicity reasons. However, Galax is a project under development but it benefits from the most important aspects of XQuery such as data model, static type system and optimization.

More importantly, a simple and efficient transformer from Scala abstract code to XQuery abstract code was implemented in this project. In order to perform the transformation at execution-time, another abstract syntax was proposed for Scala code from reflection library. This abstract syntax was expanded during this project to cover all necessary Scala expressions for XML query manipulation.

To conclude, Scala programming language possesses sufficient facilities to interact with native XML database systems. On the other hand, Scala transformation to XQuery expressions is absolutely achievable. However in this project we attained almost all our objectives mentioned in the second chapter, but one fun-

damental issue that remains undecided for Scala developers is that: In reality, which part of Scala code should be transformed into XQuery code and when query shipping is preferable? It means that, Scala programmer decides which part of Scala code should be evaluated by Scala interpreter and which part by XQuery processor to obtain efficient results.

Acknowledgements

I would like to thank my project supervisor, Prof. Martin Odersky, and my project assistant, Gilles Dubochet, for initiating the idea of this project and for their helps, advises and supports. I would also like to thank Burak Emir for his key advices about xml library of Scala. I feel extremely grateful for the opportunity to spend a whole semester working on a project in a field of my personal interest. And I hope, this project would be a good start point in the new generation of programming languages and XML technologies.

References

1. M. Odersky, et al., "*An Overview of the Scala Programming Language*", LAMP-EPFL, 2004.
2. M. Odersky, et al., *"The Scala Language Specification"*, LAMP-EPFL, 2006.
3. D. Chamberlin, et al., *"XQuery from the Experts"*, Addison-Wesley, 2004.
4. F. Yergeau, T. Bray, et al., *"Extensible Markup Language (XML) 1.0 (Third Edition)"*, W3C Recommendation, 2004.
5. A. Berglund, S. Boag, et al., *"XML Path Language (XPath) Version 2.0"*, W3C Candidate Recommendation, 2005.
6. S. Boag, D. Chamberlin, et al., *"XQuery 1.0: An XML Query Language"*, W3C Candidate Recommendation, 2005.
7. D. Box, A. Hejlsberg, *"The LINQ Project - .NET Language Integrated Query"*, Microsoft Corporation, 2005.
8. *"The .NET Standard Query Operators"*, Microsoft Corporation, 2005.
9. *"XLinq .NET Language Integrated Query from XML Data"*, Microsoft Corporation, 2005.
10. M. Fernandez, J. Simeon, and P. Wadler, *"A semi-monad for semi-structured data"*, 2001.
11. D. Kossmann and M. Franklin, *"A study of Query Execution Strategies for Client-Server Database Systems"*, 1996.
12. L. Wong, *"Kleisli, a Functional Query System"*, 1998.
13. G. Dubochet, *"The Slinks Language"*, Master Thesis, 2005.
14. M. Fernández, J. Siméon, *"The Galax System"*, 2005.