NTNU
Norwegian University of Science and Technology

Translating XQuery to Relational Algebra

Mads Nyborg
Andreas Ravnestad

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Svein Erik Bratsberg, IDI
Co-supervisor: Øystein Torbjørnsen, Fast Search & Transfer

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

XQuery is a flexible language for querying XML data. This language may be a suitable interface for performing queries on MARS, a search engine framework under development at Fast Search & Transfer.

The task is to find or develop a method for translation of XQuery queries into MQL, a relational algebra language for MARS, and to implement a proof of concept which demonstrates some of the capabilities of this method.

Assignment given: 15. January 2008
Supervisor: Svein Erik Bratsberg, IDI

Abstract

XQuery is a flexible language for querying XML data across a variety of storage methods. This thesis is a part of iAD, an ongoing research effort in next generation information access solutions. iAD is hosted by Fast Search & Transfer, a company developing their next search engine platform MARS. This project seeks to investigate the utilisation of XQuery as a query language for MARS.

The result of this project is a novel method of translation, dubbed “Tainting Dependencies” (TD), which seeks to avoid unecessary denormalisation of intermediate results, and is designed specifically for translation to MARS’ relational algebra. This method supports a large subset of XQuery features.

Furthermore, we have developed a prototype implementation which supports basic constructs such as FLWOR and sequence construction. TD is then compared head-on to a similar method dubbed “Loop Lifting”, and the results of this comparison is evaluated through discussion.

The outcome of this project is a novel and well-documented method for translation of XQuery to MQL – a method which is designed to perform equally or better than existing implementations.

Preface

This thesis was written at the Department of Computer and Informations Science (IDI) at the Norwegian University of Science and Techology (NTNU) during the spring semester of 2008. The assignment was given by and written for the Information Access Distruption centre (iAD). The supervisors of this project has been Svein Erik Bratsberg at NTNU and Øystein Torbjørnsen at Fast Search & Transfer.

We would like to thank Svein Erik Bratsberg for feedback and proof reading of this report. Additionally we would also like to thank Øystein Torbjørnsen for taking time from his busy schedule giving guidance, feedback and teaching us about the workings of MARS.

Trondheim, June 10, 2008

Andreas Ravnestad　　　　Mads Nyborg

Contents

Glossary

ANTLR **AN**other **T**ool for **L**anguage **R**ecognition. A predicated-LL(k) parser generator that handles lexers, parsers, and tree parsers.

AST **A**bstract **S**yntax **T**ree. A two-dimensional tree which encode the structure of the the input symbols.

DAG **D**irected **A**syclic **G**raph.

DOM **D**ocument **O**bject **M**odel. A platform- and language-independent standard object model for representing XML and related formats.

EBNF **E**xtended **B**ackus **N**aur **F**orm. A metasyntax notation used to express context-free grammars

FLWOR **f**or, **l**et, **w**here, **o**rder by, **r**eturn. A XQuery expression with loop semantics. See Section 2.1.4.

Iterator dependency A concept of Tainting Dependencies. See Section 4.3.1

Loop Lifting A XQuery to relational algebra translation method developed by Torsten Grunst and Jens Teubner. See Section 2.5.

MarkXRemove A predecessor of Tainting Dependencies. See Section 4.1.

MARS A search engine platform under development at Fast Search & Transfer

MQL MARS **Q**uery **L**anguage. The relational algebra language of MARS

Normalised, relation A relation without redundancies.

Normalised, XQuery See XQuery Core, Section 2.1.6.

Production A grammar spesification rule, either terminal or non-terminal.

Tainting Dependencies A novel XQuery to MQL translation method developed as a part of this Master's thesis. See Chapter 4

Tainting, iterator dependency A concept of Tainting Dependencies. See Section 4.3.2

TD See **Tainting Dependencies**

W3C **W**orld **W**ide **W**eb **C**onsortium. An international standards organisation for the World Wide Web.

Chapter 1

Introduction

The search engines of today are capable of finding relevant documents based on simple search terms as well as weighting and ranking schemes of varying complexity. However, few are capable of joining several query results, performing structural queries, and filtering by complex full-text expressions in a single unified query operation.

XQuery is an XML query language capable of performing complex nested queries, extendable with full-text searching, including linguistics such as stemming and thesaurus. In theory, XQuery queries may be translated into relational algebra for execution on a suitable algebra processor engine.

iAd [24] (Information Access Disruptions) is an ongoing research effort in “next generation information access solutions”, in which this project partake. Our specific goal is to develop a method of translating XQuery queries into MARS relational algebra, and compare this to existing technology. Furthermore, a prototype will be implemented as a proof of concept.

This report is structured as follows: In Chapter 2 we will examine XQuery itself in further detail and investigate the current state of XQuery translator implementations and research. In Chapter 3 we will detail the tools and methods used. In Chapter 4 we will describe our novel translation method dubbed “Tainting Dependencies”. Continuing to chapter 5 we expound on the implementation of a prototype which serves as a proof of concept. Eventually the results will be presented and discussed in chapters 6 and 7. Our work is concluded in chapter 8, and we propose future work and improvements in Chapter 9.

Chapter 2

Theory

XML is designed for encapsulation of structured and semistructured data, relational data, and object repositories. The XQuery query language is designed to perform flexible queries in such data. A translation of the language into relational algebra requires detailed knowledge about XQuery for a proper translation to be made. In this chapter, section 2.1 describes important details about the XQuery language that are essential to this task. Further, existing implementations of such translators are documented and compared.

Additionally, the concept of classic relational algebra itself is outlined in Section 2.3, as the target algebra will be based on much of its semantics.

Then in Section 2.4, some common strategies for parsing and construction of parsers are outlined, as well as techniques for parsing syntax trees.

Finally, a thoroughly researched method for translating XQuery to relational algebra dubbed "Loop Lifting" is described in Section 2.5.

2.1 XQuery

XQuery is a query language developed by the XML Query working group of W3C. Version 1.0[36] became a W3C Recommendation January 2007. It was designed as a response to an emerging task: to intelligently express queries in the increasing amounts of information stored, exchanged and presented using XML. The language is derived from Quilt[10]. Development of XQuery 1.0 was coordinated with the development of XSLT 2.0, and the two teams cooperated on development of XPath 2.0.

XQuery can be used to query any kind of data structure that can be represented as an XML document. This includes text documents, relational databases and XML-compliant HTML markup.

2.1.1 Basic language features

XQuery is a functional language with a comparatively small syntax. It lacks some features known from many functional languages, such as support for higher order function declaratil ons. However, it has some of the most important benefits, such as a lack of side-effects. XQuery is a *declarative* language (as opposed to *imperative* languages), and is strongly typed. Static typing is optional, and may vary between various implementations.

XQuery is an orthogonal language, meaning that most expressions can be arbitrarily nested. For example, a path expression predicate can be another path expression:

```
/a/b[/c/d[e]]
```

Or, the return-clause in a loop construct can be another loop construct:

```
for $i in (1,2,3)
  return for $j in (4,5,6)
    return $i + $j
```

These features are important to consider for later translation, as truth values in predicates and return values may need to be coerced and/or inferred into their proper types and values.

The XQuery type system is rather complex, and we refer to some of the introductory articles[23] by Michael Rys, as well as the XQuery formal semantics specification[33] for more information about this. However, we will emphasize some important traits about the type system:

All sequences are one-dimensional. Any given sequence that is not one-dimensional, will be flattened. For example, the two-dimensional sequence `((1,2),3)` is to be flatted into `(1,2,3)`.

Sequences can evaluate to an effective boolean value. Informally, this value is defined as follows: **Anything that is not 0, empty, or *false*, evaluates to *true***. In a boolean context (such as an predicate or an `if..then..else`), this means that anything that is "something" will evaluate to *true*.

2.1.2 Path expressions

XPath (XML Path Language) is a small language for traversing and selecting nodes (both element nodes and text nodes) from XML data. XPath is a subset of XQuery, and is also available in XSLT, XML Schemas, XForms, and several other technologies related to XML.

In its abbreviated form, XPath bears a strong resemblance to file path syntax known from many modern operating systems. This implies that the XPath syntax may be familiar and intuitive for new users.

For example, consider the following XML source:

```
<a>
  <b><c>Hello World</c></b>
</a>
```

If we execute the XPath expression `/a/b/c`, we will receive the `c`-node which is a child of the `b`-node which is a child of the `a`-node which is the document root node. Note that we will *not* receive the text `Hello World`, which is a **text node**, but rather its parent node, which is the `c`-node. To retrieve the text, we would rather use the path expression `/a/b/c/text()`. The `text()` expression is known as a **kind test**. The following kind tests are available:

- `text()` - as described above, returns a text node

- `comment()` - returns a comment node, for example `<!- Hello world ->`
- `processing-instruction()` - returns processing instructions, which means constructs such as `<?xml version="1.0"?>`
- `node()` - returns any type of node

In its unabbreviated syntax (or, verbose syntax), the semantics of XPath become more clear. For the XML source above, the full syntax for the path expression to match the `c`-node would be `/child::a/child::b/child::c`. Here we see a new addition to our path expression, the `child::` axis specifier. An **axis specifier** helps navigation within the XML document, by allowing the user to specify further traits about the nodes to be matched. For example, attribute nodes can be matched using `attribute::` (or `@`, with abbreviated syntax). For a complete reference to axis specifiers, we refer to [34].

2.1.3 Predicates

Predicates are used in path expressions to filter nodes. Predicates are appended to step expressions (and filter expressions, see [34]), and multiple predicates are applied from left to right. Predicates never add to the node sets returned from the path expressions, they only restrict by filtering. Predicate expressions are appended to step expressions within square brackets, like this:

```
/a/b[@id > 1]
```

This expression will return all `b`-nodes within a `a`-node and with an attribute `id` whose value is larger than one.

Consider the following XML source:

```
<a>
  <b id="1">
    <c />
  </b>
  <b id="2">
    <c />
  </b>
</a>
```

If we apply the path expression mentioned above, we will thus receive the second `b`-node.

There are a few important things to note about predicates. Firstly, the predicate expression can be any expression, and as such its return value is coerced into a effective boolean value (either *true* or *false*), as described previously in Section 2.1.1.

However, there is one important exception – if the return value for the predicate expression evaluates to a numerical value, then the predicate becomes a **numeric predicate**, and its value is used to identify the nth node in the step expression. For example, the following path expression returns the first `b`-node: `/a/b[1]`, as it is the first `b`-node within the only `a`-node. `/a/b/c[1]` will select both `c`-nodes of the document as they both are the first `c`-node within their respected `b`-nodes.

2.1.4 FLWOR

Definition 1. *An **iteration expression** or **iterator** is an XQuery expression consisting of an **iterator variable** declaration and an **iterator body**. The **iterator body** is executed multiple times, and for each time the **iterator variable** is bound to the next item in the **iterator sequence**.*

XQuery is centered around a loop construct known as FLWOR, which is an acronym:

- **F**or - iteration over tuples
- **L**et - assignment of tuples
- **W**here - conditional expression
- **O**rder by - sorting
- **R**eturn - return expressions (similar to yielding in coroutines known from functional languages, not to be confused with a return statement in languages such as Java)

The FLWOR construct is thought to be roughly equivalent to a `SELECT`-statement in SQL. For example, consider the following SQL statement:

```
SELECT v.title FROM video v WHERE v.year = 1959
```

And then compare it to the following XQuery counterpart:

```
for $v in doc("videos.xml")//video
where $v/year = 1959
return $v/title
```

Then construct a file `videos.xml` with the following contents:

```
<videos>
  <video>
    <title>Plan 9 from outer space</title>
    <year>1959</year>
  </video>
  <video>
    <title>Earth vs. the Flying Saucers</title>
    <year>1956</year>
  </video>
</videos>
```

And finally execute the above query on this file to receive the following result:

```
<title>Plan 9 from outer space</title>
```

It is important to note the distinction of bound and free variables in FLWOR constructs – or, in other words, the scope boundaries. Consider the following example:

```
for $a in (1,2,3)
  return for $b in (4,5,$a)
    return $a + $b
```

When evaluating the `for`-clauses in this nested FLWOR expression, the *iterator sequence* is evaluated in the parent scope and not the new scope for the current FLWOR expression. We can illustrate this point by separating the scopes graphically:

```
for $a in (1,2,3)
  return for $b in (4,5,$a)
    return $a + $b
```

As can be seen, the iterator sequence for the inner loop is evaluated in the scope of the outer loop, and a new scope is not started until this iterator sequence has been evaluated. Otherwise one could risk overwriting variables in the iterator sequence when binding variables in the new scope.

Furthermore, a FLWOR construct may consist of several `for`- and `let`-clauses in any order – and each of these clauses may contain several variable bindings. For example, the following is a valid XQuery FLWOR expression:

```
for $a in (1,2), $b in (3,4)
let $c := 5, $d := 6
  return $a + $b + $c + $d
```

However note that semantically this expression is equivalent to:

```
for $a in (1,2) return
  for $b in (3,4) return
    let $c := 5 return
      let $d := 6 return
        $a + $b + $c + $d
```

The latter seems significantly less complex to parse, since this query embodies no less than *four* individual FLWOR expressions, each with one and only one `for`- or `let`-clause. This raises the question, could it be benefitial to rewrite complex FLWOR expressions into a simpler form? This question is addressed in Section 3.3 on page 30.

2.1.5 Full text extensions

XQuery is by nature a structural query language – that is, queries are based on document/data structure and not on content. The full-text extensions to XQuery reduces the smallest unit of an XML document to single words instead of nodes. Additionally, they add sophisticated tools such as stemming, thesaurus, and scoring variables.

Technically, the `ftcontains` operator applies tokenisation of the first operand, and searches for a match with the second operand among the tokens. It allows specifying match options like stemming, thesaurus, etc to a second operand modifying the criteria for finding a match. A full list of match options are described in [34].

For example, consider the following example:

```
for $b in /books/book
where $b/title ftcontains ("dog" with stemming case sensitive)
      ftand "cat"
return $b/author
```

This will match any `book`-node where the `title`-node contains a word with the stem "dog". Further, the word must be in lower case, and the word "cat" must reside inside the same node. The query will return the `author`-node of these `book`-nodes.

2.1.6 XQuery Core

XQuery Core is a less powerful but semantically equivalent language for expressing XQuery queries. XQuery Core as well as the process of normalising regular XQuery to XQuery Core is described in the document "XQuery 1.0 and XPath 2.0 Formal Semantics"[33].

The goal of this subset language is to simplify queries and remove syntactic sugar, leaving only the essential semantics without loss of expressiveness. This is useful for optimization routines and translations into new types of queries, for example relational algebra or SQL.

The process of normalisation is described through a rich set of mapping rules. These are documented in detail in [33] and will not be reiterated here. However we will examine some important examples.

First, however, it is important to take note of the syntax of the mapping rules, as described in [33], section 3.2.2.

$$[Object]_{Subscript}, premises == MappedObject$$

Figure 2.1: Mapping rules syntax

Consider Figure 2.1. Here, the left-hand side of the equality symbol (==) denotes the original object to be rewritten. The subscript indicates the type or kind of the object to be mapped, and/or additional information to be passed between mapping rules. The right-hand side denotes the rewritten object.

Rewriting FLWOR expressions

$$[\texttt{for }\$VarName_1\ \textit{OptTypeDeclaration}_1\ \textit{OptPositionalVar}_1\ \texttt{in}\ \textit{Expr}_1\texttt{,}\ \ldots\texttt{,}\ \$VarName_n\ \textit{OptTypeDeclaration}_n\ \textit{OptPositionalVar}_n\ \texttt{in}\ \textit{Expr}_n\ FormalReturnClause]_{\textit{Expr}}$$

$$==$$

$$\texttt{for }\$VarName_1\ \textit{OptTypeDeclaration}_1\ \textit{OptPositionalVar}_1\ \texttt{in}\ [\textit{Expr}_1]_{\textit{Expr}}\ \texttt{return}$$
$$\ldots\texttt{for }\$VarName_n\ \textit{OptTypeDeclaration}_n\ \textit{OptPositionalVar}_n\ \texttt{in}\ [\textit{Expr}_n]_{\textit{Expr}}\ \texttt{return}$$
$$[FormalReturnClause]_{\textit{Expr}}$$

Figure 2.2: XQuery FLWOR expression to XQuery Core mapping rule

The mapping rule for FLWOR `for`-clause expressions can be seen in figure 2.2. The mapping rule for `let`-expressions is similar and omitted for brevity, however they are also normalised into several nested bindings.

Similarly, the mapping rules for `where`-clauses, `order by`-clauses and `return`-clauses can be seen in figures 2.3, 2.4, and 2.5.

$$\begin{gathered}[\texttt{where}\ \mathit{Expr}_1 \mathit{FormalReturnClause}]_{\mathit{Expr}} \\ == \\ \texttt{if(}[\mathit{Expr}_1]_{\mathit{Expr}}\texttt{)}\ \texttt{then}\ [\mathit{FormalReturnClause}]_{\mathit{Expr}}\ \texttt{else ()}\end{gathered}$$

Figure 2.3: XQuery `Where`-clause to XQuery Core mapping rule

$$\begin{gathered}[\texttt{stable}?\ \texttt{order by}\ \mathit{OrderSpecListFormalReturnClause}]_{\mathit{Expr}} \\ == \\ [\mathit{OrderSpecList}]_{\mathit{OrderSpecList}}\ \texttt{return}\ [\mathit{FormalReturnClause}]_{\mathit{Expr}}\end{gathered}$$

Figure 2.4: XQuery `order by`-clause to XQuery Core mapping rule

$$\begin{gathered}[\texttt{return}\ \mathit{Expr}]_{\mathit{Expr}} \\ == \\ [\mathit{Expr}]_{\mathit{Expr}}\end{gathered}$$

Figure 2.5: XQuery `return`-clause to XQuery Core mapping rule

For an example of how these rules are applied, consider the following FLWOR expression:

```
for $i in (1, 2), $j in (3, 4)
let $k := $i + $j
where $k >= 5
return ($i, $j)
```

By applying the mapping rules described, this expression is typically rewritten to:

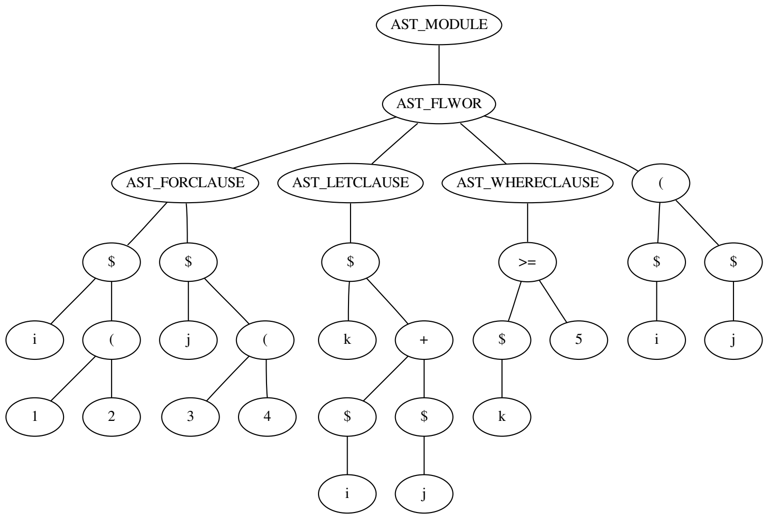
```
for $i in (1, 2) return
  for $j in (3, 4) return
  let $k := $i + $j return
    if ($k >= 5) then ($i, $j)
    else ()
```

The corresponding AST graphs can be seen in figures 2.6(a) and 2.6(b). In particular, note that multiple `for`-clauses in a FLWOR expression is rewritten into several nested FLWOR expressions, and that the `where`-clause is rewritten into an `if..then..else` expression.

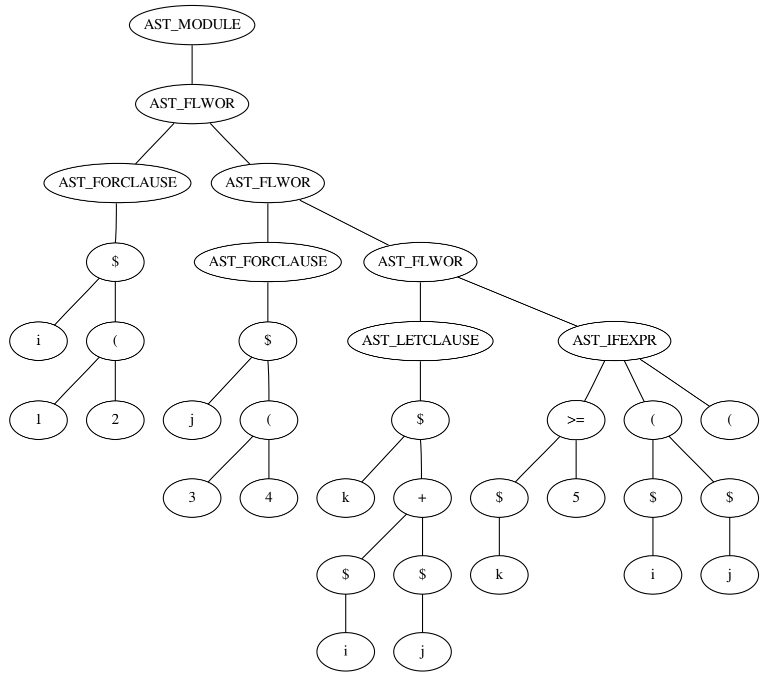
Rewriting composite relative path expressions

A composite relative path expression (for example, `a/b`), can be rewritten into a `for`-loop using the mapping rule in figure 2.7. Given the trivial example `a/b`, this translates into the following block of normalised code:

```
fs:apply-ordering-mode (
fs:distinct-doc-order-or-atomic-sequence (
  let $fs:sequence as node()* := a return
  let $fs:last := fn:count($fs:sequence) return
  for $fs:dot at $fs:position in $fs:sequence return
    b))
```



(a) FLWOR AST tree before normalisation



(b) Normalised FLWOR AST tree

Figure 2.6: A FLWOR expression before and after normalisation.

Which may seem like a rather verbose representation of such a simple path expression. In particular, for complex path expressions this may escalate into rather large rewritten expressions. However, this is a trade-off to be made for normalisation of such path expressions.

```
             [RelativePathExpr/StepExpr]_Expr
                          ==
fs:apply-ordering-mode(
  fs:distinct-doc-order-or-atomic-sequence(
    let $fs:sequence as node()* := [RelativePathExpr]_Expr return
    let $fs:last := fn:count($fs:sequence) return
    for $fs:dot at $fs:position in $fs:sequence return
          [StepExpr]_Expr ))
```

Figure 2.7: Composite relative path expression mapping rule

2.2 Existing implementations

This section describes some of the most interesting existing implementations of XQuery parsers and translators. Note that some of these are fundamentally different in some aspects (for example, eXist operates on DOM trees), but may implement certain features of interest, and are therefore included here.

2.2.1 eXist

eXist[30] is an open source native XML database with an XQuery query processor. The eXist system is written in Java. This system stores native XML data in B-trees and paged files, and document nodes in persistent DOM trees[29]. Document collections are stored in a hierarchical manner similar to a regular file system.

eXist has a numerical indexing scheme for identification of relationships between nodes (parent/child, ancestor/descendant, previous/next sibling). This provides a structural index for element attribute nodes. In addition, eXist has a fulltext index for text and attribute values, and range indexes for typed values.

Based on these provided index types, the eXist XQuery engine relies on path join algorithms[31] for efficient computation of node relationships instead of traditional tree traversals.

2.2.2 Pathfinder

Pathfinder[25] claims to be a “purely relational XQuery processor”, which theoretically can utilise any off-the-shelf RBDMS as a backend for XQuery execution in a relational context.

The technique used for transforming XQuery into relational algebra is called “Loop Lifting”[28]. Loop Lifting is a FLWOR-centric approach which essentially transforms iterations into joins. The technique is described in detail in Section 2.5.

As a relational backend, the XQuery processor uses MonetDB, an integrated component in the Pathfinder project. However, recent versions of Pathfinder is also capable of producing SQL

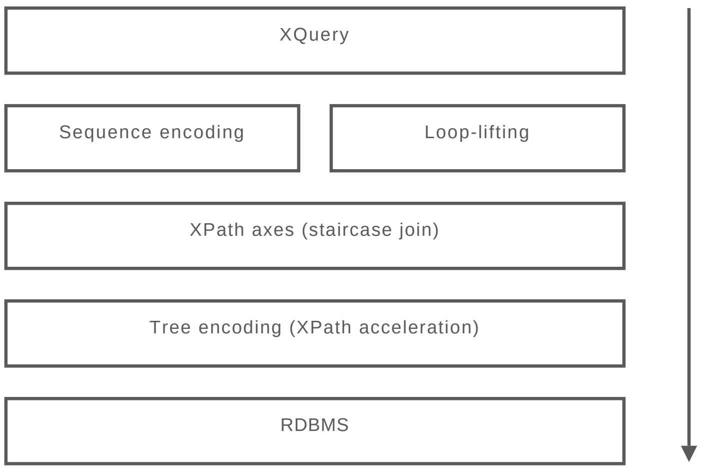

XQuery
Sequence encoding
Loop-lifting
XPath axes (staircase join)
Tree encoding (XPath acceleration)
RDBMS

Figure 2.8: Pathfinder architecture / development stack

code for execution on conventional database systems. As a proof of concept, they performed the XMark test suite on top of the IBM DB2 system[27].

XML-documents are stored the database where each XML node are stored as one tuple. Each tuple contains *pre* and *post* fields, with values corresponding to the order of visiting the node during a preorder and postorder traversal of the XML-tree, respectively. Axis steps of path expressions are evaluated utilising "staircase join", a custom made join operator utilising the *pre* and *post* encoding of the XML-documents. Figure 2.8 shows a conceptual illustration of XQuery processing in Pathfinder.

2.2.3 Galatex

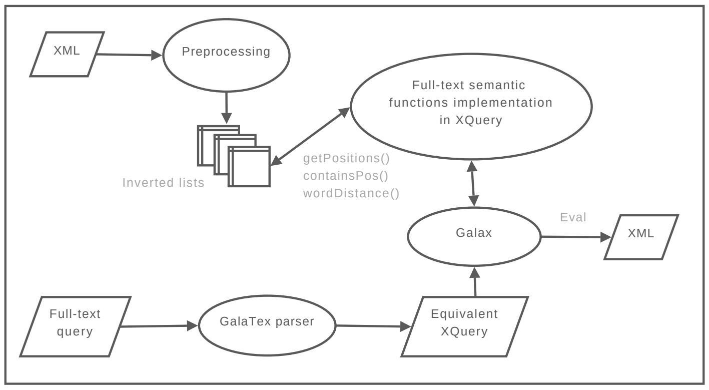


Figure 2.9: Galatex architecture, based on architecture described on the GalaTex website[12]

Galatex is claimed to be the first full implementation of the W3C XQuery 1.0 and XPath 2.0 Full-Text 1.0 specification[34]. As can be seen in figure 2.9, Galatex translates the full-text parts of the query into XQuery Core[33] in the Galatex parser. The equivalent XQuery query is then executed on the Galax query processor.

The most interesting aspect of Galatex is its full-text capabilities. This is realised first and foremost by an implementation of the AllMatches data model proposed by W3C[34].

Additionally, as mentioned above, the full-text fragments of the queries are translanted into corresponding functions in regular XQuery.

2.2.4 Trait comparison matrix

This section will compare some important traits of the described implementations, and will outline their implications. The traits chosen for the comparison matrix in table 2.1 are chosen

	eXist	Pathfinder	GalaTex
Normalisation to XQuery Core	Yes	Yes	Yes
Relational back-end	No	Yes	No
Full-text extensions	No	No	Yes
Test suite coverage	99.4%	99.4%	n/a
Free source code	Yes	Yes	Yes

Table 2.1: Comparison of implementation traits

due to their relevance to this project. As can be seen, there is a spread in diversity across these implementations. In particular, Pathfinder is the only implementation with a relational backend. This implies that Pathfinder will be a natural focal point of interest for studying existing methods of translation. As will be shown in chapters 3 and 4, Pathfinder and its Loop Lifting technique will serve as a basis for development of an improved and novel translation method.

2.3 Relational algebra

The relational model for database management was introduced for the first time by Edgar Frank Coddin 1974[5]. It was based on relational algebra which is an offshoot of first-order logic. Several terms are used when talking about relational algebra [11][8][7]:

- **Set**: A mathematical definition for a collection of objects which contains no duplicates
- **Domain**: A *set* of atomic values
- **Attribute**: A real world role played by a named *domain*
- **Tuple**: A collection of *attributes* which describe some real world entity
- **Relation**: A *set* of *tuples*
- **Degree**: The number of *attributes* of a *relation.* Sometimes called arity.
- **Cardinality**: The number of *tuples* in a *relation*
- **Union compatible**: Two relations R and S are union compatible if and only if they have the same *degree* and the *domains* of the corresponding *attributes* are the same.

It should be noted that often relational algebra is based on “bag” semantics rather than set semantics. A “bag” may contain duplicates unlike sets. Removal of duplicates can be a very costly operation in terms of computer resources.

2.3.1 Primary operators

The primary operators is a set of operators which constitutes the base of an algebra. Other operators can be defined in terms of the primary ones. If one of the primary operators is excluded, the algebra will loose some of its expressiveness. The primitive operators of Codd’s algebra are: selection, projection, union, difference, cross product and rename (later added for the sake of the named relational algebra).

Selection

Selection is a unary operator, and is used to obtain a subset of the tuples of a relation that satisfy a select condition. The resulting relation may have fewer tuples but it will have the same degree as the original relation. It is sometimes called restriction to avoid confusion with SELECT in SQL. The operator is often symbolised by $s$igma:

$$\sigma_C(R)$$

Where R is a relation and C is the select condition: a truth value or an expression yielding a truth value. The expression can be made up of any combination of the logical operators $\{\wedge, \vee, \neg\}$. Figure 2.10 shows an example of a select operation.

R	
Letter	**Number**
A	1
A	3
A	6
B	7

$$\sigma_{Letter='A' \wedge Number>2}(R) =$$

Letter	**Number**
A	3
A	6

Figure 2.10: Example showing the selection operator

Projection

Projection is also a unary operator, and is used to obtain a subset of the attributes of a relation. The resulting relation will have an equal or lower degree than the original relation. In the case of duplicates being produced as a result of omitting some attributes, the resulting relation will have fewer tuples than the original. Pi is often used to symbolise the operation:

$$\pi_{attr}(R)$$

Where R is a relation and $attr$ is the set of attributes to be returned from R. Figure 2.11 shows an example of a projection.

R		
Let	**Num**	**Sym**
A	1	%
B	1	%
C	3	#

$\pi_{Num,Sym}(R) =$

Num	**Sym**
1	%
3	#

Figure 2.11: Exaple showing the projection operator

Union and difference

Union and difference are two binary operators analogous with union and difference operators in set theory. The relational algebra version of the operators requires that the relations involved are union compatible.

The union of two relations returns a relation which includes all the tuples that are in either or both of the original relations. As the result is also a relation, any potential duplicates will be removed. The operation is commutative, and the returned relation will have the same degree as the two relations involved. A union between two relations are often symbolised as:

$$R \cup S$$

Where R and S are relations.

The difference of two relations R and S is a relation that contains all the tuples that are in R but not in S. The returned relation will, as is the case with union, have the same degree as the two relations involved. A difference between relations R and S is written like this:

$$R - S$$

Cross product

The cross product operator is sometimes referred to as the cartesian product operator. As with union and difference, this operator also stems from set theory. The operator is used to combine all tuples in one relation with all the tuples from another. The returned relation will have a degree equal to the sum of the degrees of each of the original relations, and a cardinality equal to the product of the cardinalities. The operator is commutative and written as a cross:

$$R \times S = S \times R$$

Where R and S are relations. Figure 2.12 shows an example of a cross product.

R
Let
A
B

$\times$

S	
Num	**Let**
1	C
2	A

$=$

R.Let	**S.Num**	**S.Let**
A	1	C
A	2	C
B	1	A
B	2	A

Figure 2.12: An example of cross product.

Rename

Rename is a unary operator used to rename a relation and/or a subset of its attributes. The resulting relation will be equal to the original one in all aspects except maybe some of its name properties. The Greek letter $r$ho is often used to mark the presence of the rename operator:

$$\rho_S(R)$$

Where R is the relation being renamed, and S is a relational scheme. S is on the form $T_{(a_1,...a_n)}$ for a n-degree relation, where T is the new relation name and $a_1, ...a_n$ is the new names for relation R's attributes from 1 to n. The degree of the scheme must be the same as the degree of the relation being operated on.

2.3.2 Derived operators

None of the six primary operators can be expressed as a combination of any of the others. In contrast, some useful operators can be derived using one or more of the primary ones. Most notably among these are intersection and join.

Intersection

Intersection is the fourth mentioned operator that stems from set theory. It is a binary operator, and the resulting relation will contain the set of tuples that are in both of the relations operated on. It can be expressed with the help of the difference operator, and hence require that the input relations are union compatible:

$$R \cap S = R - (R - S)$$

Joins

Joins are a group of operators that all are derived from the primary operators with the cross product as a base. Among the operators in this group is natural join, theta-join, equi-join, anti-join, semi-join, outer joins and division. Some of these will be presented in this section.

Natural join. Natural join is a binary operator that returns a relation consisting of all combinations of tuples in input relations that are equal on their common attribute names. The result relation will have a degree equal to the sum of the degrees of the two original relations subtracted the number of common attributes. Natural join can be expressed as a combination of cross product, projection and selection:

$$R \bowtie S = \pi_{a_1,...,a_n,R.b_1,...,R.b_n,c_1,...,c_n}(\sigma_{R.b_1=S.b_1 \wedge ... \wedge R.b_n=S.b_n}(R \times S))$$

Where R and S are relations, $b_1, .., b_n$ are the common attributes, $a_1, .., a_m$ are the attributes unique to R and $c_1, ..., c_k$ are the attributes unique to S. A rename operator can lastly be used to remove the prefix of the common attributes.

Equi-join and theta-join. Theta-join returns a relation which is a combination of all the tuples in the two input relations that satisfy a condition C. C is in the form $a\theta b$, where a is a attribute name from one relation, b is an attribute from the other and θ is a binary operator in the set $\{<, \leq, =, \geq, >\}$. An equi-join is a theta-join where the binary operator in the condition is the equality operator. Theta-join can be expressed as a combination of selection and a cross product:

$$R \bowtie_{a\theta b} S = \sigma_{a\theta b}(R \times S)$$

Division. Division in relational algebra can be described as the inverse operator of cross product, in the same way division and multiplication are inverse in natural numbers calculus – i.e. they are not inverse if the division gives a residue:

$$(R \times S) \div R = S \;\; and \;\; (R \times S) \div S = R$$

The resulting relation after a division contains the attribute values of the divisor relation that are associated with every member of the dividend relation[16]. The operation may be better explained as a combination of cross product, projection and difference:

$$R \div S = \pi_{a_1,...,a_n}(R) - \pi_{a_1,...,a_n}((\pi_{a_1,...,a_n}(R) \times S) - R)$$

Where R and S are relations and $a_1, ..., a_n$ are the attributes unique to R.

Semi-join. Semi-join is a binary operation which returns a relation with the attributes of the first relation, and all the tuples in this same relation for which there is a tuple in the second relation that is equal on their common attributes. Semi-join can be described with the project and natural-join operators:

$$R \ltimes S = \pi_{a_1,...,a_n}(R \bowtie S)$$

Where R and S are relations, and $a_1, ..., a_n$ are the attributes unique to R.

Anti-join. The anti-join operator is very similar to the semi-join operator (and is also sometimes referred to as the anti-semi-join), except that it returns all the tuples in the first relation for which there is no tuple in the other relation on their common attributes. It can be described with help of semi-join and difference:

$$R \rhd S = R - R \ltimes S$$

Where R and S are relations.

Outer joins. The outer joins is in many ways as the natural join, except the resulting relation will include some extra tuples based on one or both of the input relations. The right outer join ($\times =$) will return a relation with all the tuples from a natural join between the first (left) and the second (right) relation, as well as the tuples from the right relation that did not match any tuples from the left one on their common attributes. These extra tuples will have the value NULL in the result relation for all attributes that were unique to the left relation. The left outer join ($= \times$) is analogous with the right version, the only difference is that the extra tuples will be based on the left input relation. The result relation of a full outer join ($= \times =$) will have extra tuples based the ones that did not find a match in both input relations.

2.4 Parsing and syntax trees

The act of parsing is the process of analysing a series of tokens and construct a grammatical structure (syntax tree) based on a formally specified grammar. In Figure 2.13, the parser component of a generic compiler/interpreter is shown in the context of a generic compiler architecture.

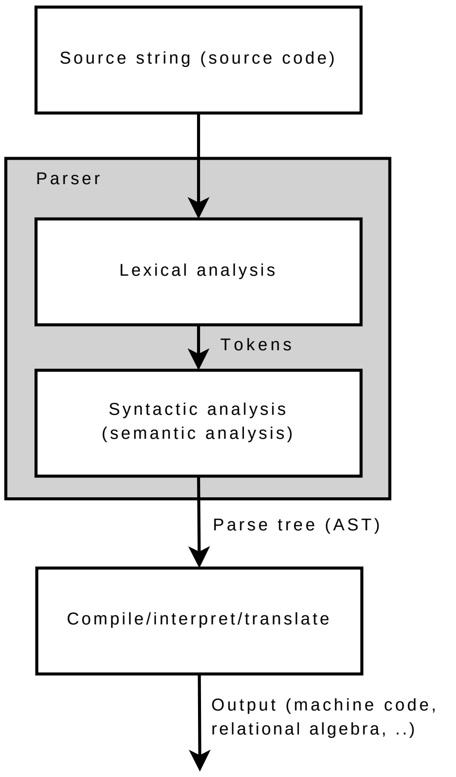


Figure 2.13: Typical compiler/interpreter data flow

2.4.1 Common parser technologies

There are two common types of parser technologies, *top-down* and *bottom-up* parsers. As their names imply, these technologies differ in the sense that top-down parsers will attempt to match production rules with the input top-down, while bottom-up parsers will start with the terminal symbols and combine them into production rules. Often this is implemented as a process of shift and reduce operations, where a symbol is shifted onto the stack after consulting a parse table, and a reduction is made when a sequence of symbols are recognised as a non-terminal production rule.

We refer to [3] for further in-depth information about parser technologies. However, it is important to note that typically a top-down parser based on an LL[1]-grammar with low token lookahead (typically one token lookahead, aka LL(1)) will perform better than a *bottom-up* parser and be subject to a higher number of well-researched optimisations, out of which a few are described in [3] and [4].

2.4.2 Parser generators

For large grammars, writing a parser by hand from the ground up may turn out to be a substantial amount of work. As a natural consequence, a large number of parser generators exist. Most of these parser generators have a very similar set of functionality. From a formal grammar, often in a notation similar to BNF, the parser generator will output the source code for a complete parser for the given grammar – this process is shown in figure 2.14. Ideally, the maintainability of this generated parser could be reduced to simply changing the grammar specification, without actually modifying the generated source code.

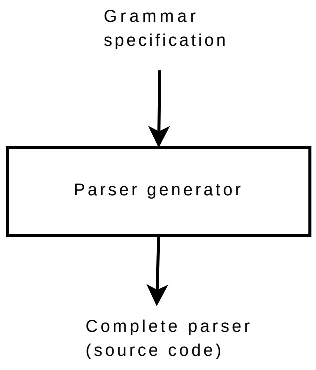


Figure 2.14: Automatic parser generation workflow

Several parser generators exists, covering several target languages and overlapping in terms of functionality and features. As expected, the most common parser generators will generate either top-down or bottom-up parsers. Typical examples of bottom-up parser generators are yacc (and derivatives), CUP, GOLD, and SableCC. Some popular top-down parser generators include JavaCC, ANTLR, Spirit, and Coco/R.

A comparison of some of the most popular parser generators were made in [19] for the development of the XQFT Parser (see Section 2.4.3), and out of these the ANTLR parser generator was chosen. We will not reiterate the features of ANTLR in this document, however it is important to note that ANTLR can generate predicated LL(*)-parsers[20] based on LL-compliant grammars. This choice was made due to the XQuery BNF specification which

[1]LL is a Left-to-right, Leftmost derivation parser, using a top-down approach

is claimed to be LL(1) (one token lookahead) – however, as discussed in [19], it may not be suitable to define the required lookahead for this grammar, as it contains ambiguities.

2.4.3 The XQFT Parser project

The XQFT Parser[19] was developed as a part of an academic project at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) throughout the autumn of 2007.

In this project the ANTLR parser generator was utilised to generate a predicated LL-parser based on the W3C specification[34] for XQuery with full-text extensions.

The output from this parser are carefully crafted abstract syntax trees which are well suited for translation into other representations.

The dataflow for actual use of the XQFT parser is shown in figure 5.2 on page 79.

AST construction

The abstract syntax tree is constructed by specifying tree rewrite rules in the grammar file (the tree rewrite rules are extensively covered in [19], section 4.5). The tree consists of nodes instantiated from the `no.ntnu.xqft.parse.XQFTTree` class. To compensate for missing tokens to determine tree context, the XQFT Parser employs the use of imaginary tokens. These are simply tokens that do not exist in the input stream, they only have an associated name and no specific token. These imaginary tokens are typically used where there are no "real" tokens available to represent the proper semantic or contextual meaning.

One example can be seen in Figure 2.15, where imaginary tokens, identified by a AST_-prefix, have been injected into the tree.

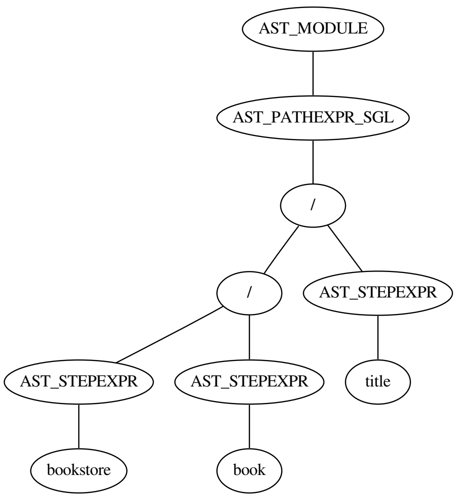


Figure 2.15: Example of injected imaginary tokens

2.4.4 Tree parsing

Tree parsing (or tree walking) is an integral part of translation from AST to new structures and representations. Several methods exist to accomplish this in various manners. Some are restricted by language limitations, while others are mostly transparent to programming paradigms and syntax. In this section we will present a small selection of common patterns, and describe their various traits.

Manual tree walker (non-visitor)

One of the simplest forms of tree parsing is the "manual non-visitor" methodology. This way of parsing a tree structure is particularly suited for trivial trees where context is not important. However, for operations that are contextually sensitive – such as context-sensitive code generation, as is the case for this project – this methodology can be difficult to maintain as the problem domain expands. This particular problem is rooted in that any contextual information must be extracted from parent nodes, and is thus not available implicitly.

Another problem with this technique is that the data structure (the tree) and the logic to parse it will often be tied together quite closely, creating another maintenance problem when/if the tree structure changes to accommodate new specifications.

Visitor pattern

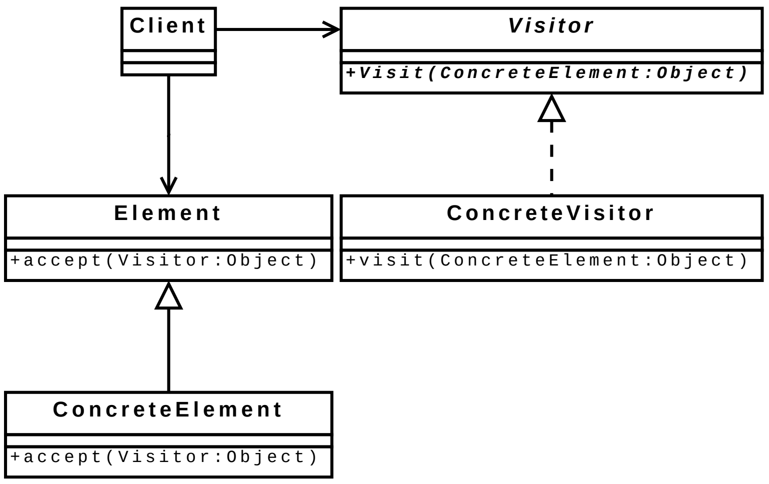


Figure 2.16: Visitor pattern

The visitor pattern[15] (Figure 2.16) is a common design pattern for parsing tree structures. The visitor pattern gives a number of ways to modify the behaviour of a hierarchy of classes without having to change those classes. Additionally, this allows the benefit of a clean separation of logic/algorithms and data structures.

However, the visitor pattern does not scale well with complexity of the problem domain. In particular, for context-sensitive parsing of tree structures, the visitor pattern will quickly lead to complex visitors, cluttered with additional logic for state preservation and determining types.

Context-sensitive visitor pattern

Here we present a novel pattern based on the visitor pattern which seeks to avoid complex state preservation mechanisms and logic for type switching. This idea is loosely based on the "island grammar" concept[18][2].

The semantics of the context-sensitive visitor pattern can be captured in the following two rules:

- For a change of state which affects how a visitor should behave when encountering some node in the subject tree, switch to another visitor which incorporates this logic (use inheritance as necessary), and execute this visitor on the subject tree
- For type switching logic which depends on context in the subject tree, switch to another visitor which incorporates this logic (use inheritance as necessary), and execute this visitor on the subject tree

Simplifying this idea, it means that visitors are specialised and used interchangeably on the subject tree as needed. This leads to the benefit of encapsulating state mechanisms and type switching logic in specialised visitors, thus avoiding a single large and complex visitor.

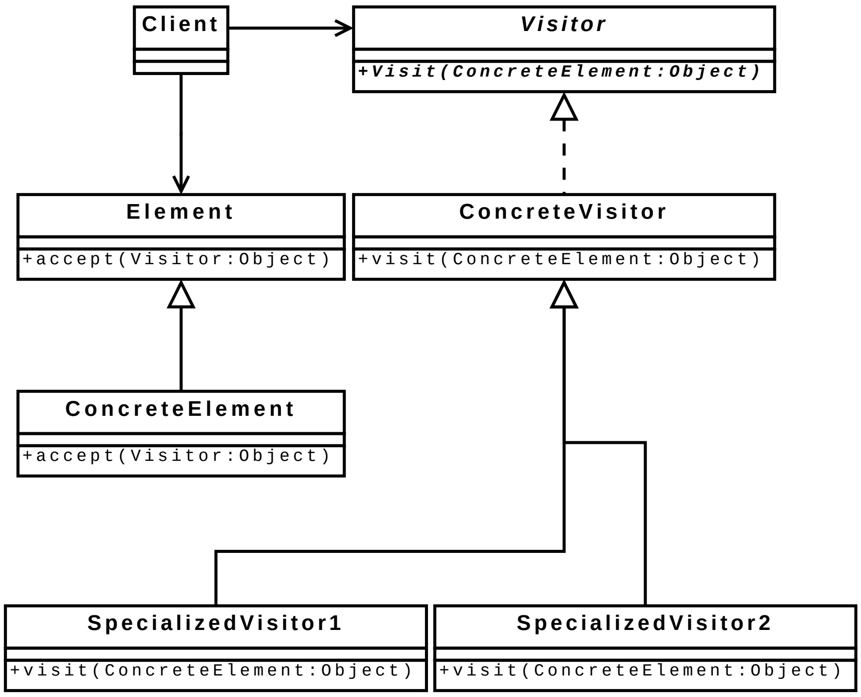


Figure 2.17: Context-sensititive visitor pattern

As can be seen in Figure 2.17, the class structure is similar to a common visitor pattern (figure 2.16). The important distinction thus lies in the logic and how the visitors are executed and how they interact. The two classes `SpecializedVisitor1` and `SpecializedVisitor2` are special-purpose visitors that can be executed from any of the others. Inheritance is implemented in this example, but is purely optional, as previously mentioned.

[2]See http://www.program-transformation.org/Transform/IslandGrammars for a brief introduction

2.5 Loop Lifting

Loop lifting is a method of translating XQuery iteration expressions into relational algebra. The method was developed by Torsten Grust and Jens Teubner and originally presented in [28]. It is a part of the Pathfinder project[25] (see Section 2.2.2).

In this section we will present Loop Lifting mainly based on the two articles [26] and [28]. The articles present the method for a subset of XQuery Core (Pathfinder rewrites queries to Core, see Section 2.2.2), of which we will only present the elements relevant in a comparison between Loop Lifting and the Tainting Dependencies method. Thus, the translation of path expressions and XML-element construction will not be handled, as pathfinder's XML-tree representation(section 2.2.2) is incompatible with MARS.

Pathfinder generates relational operator directed asyclic graphs (DAGs) rather than operator trees. The Loop Lifting method does not require such a structure, but as we will see, it will gain advantage by it, as much evaluation relies on earlier evaluations.

Accompanying the translation method is also methods for analysis, simplification and optimisation of the generated relational algebra, such as the Peep-Hole plan simplification[26].

2.5.1 Operators

Loop-lifting utilises a set of relational algebra operators, out of which the ones used in this chapter is presented in Table 2.2.

$\pi_{a_1:b_1,\ldots,a_n:b_n}$	projection and renaming
σ_a	selection
$\dot{\cup}$	disjoint union
$\times$	cartesian product
$\bowtie_{a=b}$	equi-join
$\varrho_{b:(a_1,\ldots,a_n)/p}$	numbering operator
$\circledcirc_{b:(a_1,\ldots,a_n)}$	n-ary arithmetic/comparison operator $\circ$
$\begin{array}{c\|c} a & b \\ \hline \end{array}$	literal table

Table 2.2: Operators of the Pathfinder relational algebra. a, b and p represents attributes

Most of the operators are quite standard, and can easily be understood by comparing with the operators from general relational algebra (Section 2.3) and MQL (Section 3.4.3).

Only a very restricted selection is utilised, written σ_a, which only returns tuples satisfying $a \neq 0$. Considering the numbering operator, p denotes the partitioning attribute, $a_1, \ldots, a_n$ the attributes to be sorted on and b is an added attribute holding the result of the numbering (equal to the proposed numberate-operator of MQL, Section 3.4.4). $\begin{array}{c|c} a & b \\ \hline \end{array}$ represents the creation of a relation with attributes a and b.

Operator $\circledcirc_{b:(a_1,\ldots,a_n)}$ will evaluate the arithmetic/comparison expression $a_1 \circ \ldots \circ a_n$ and place the result in b. Where $\circ \in \{+, -, <, =, \ldots\}$.

2.5.2 Basics

XQuery expressions evaluate to finite, ordered sequences of items. As a sequences are one-dimensional, it can be represented by a single relation where each tuple encodes a sequence item. The order of the sequence is maintained by an attribute *pos*. The value of the item is held in an attribute *item*.

During this section concerning Loop Lifing, variables, expressions and scopes is denoted like this (ref. section 2.1.4):

$$s \left\{ \begin{array}{l} \qquad\quad \vdots \\ \texttt{for } \$v_0 \texttt{ in } e_0 \texttt{ return} \\ \quad s_0 \, \{ e'_0 \\ \qquad\quad \vdots \end{array} \right.$$

Generally, a scope $s_{x \cdot y}$ identifies the yth child scope of scope x, $x \in \{\mathbb{N}\}$, $y \in \{\mathbb{N}\}$. Expression $e_{x \cdot y}$ evaluates to an iterator sequence and is bound to the variable $v_{x \cdot y}$. $e'_{x \cdot y}$ constitutes the coresponding iterator body, and $I_{x \cdot y}$ the whole iterator expression.

$q_x(e)$ is used to denote the relational representation of expression e in scope s_x.

2.5.3 Constant subexpressions

For a iterator expression i_x with n iterations there exists a relation $loop_x$, consisting of a single column, *iter*, with values 1,2,...,n. In the outermost scope, *loop* has a single tuple with the value 1.

A constant value c in scope s_x is *lifted* like this:

$$q_x(c) = loop_x \times \begin{array}{c|c} pos & item \\ \hline 1 & c \end{array} \tag{2.1}$$

A tuple (*iter*, *pos*, *item*) in a loop lifted relation for subexpression e'_x can be understood as during the *iter*th iteration, the item in position *pos* in e'_x has the value *item*.

2.5.4 Bound variables

An iterator sequence expression $e_{x \cdot y}$ is evaulated in scope s_x. This sequence is then iterated over and each item is successively bound to the iterator variable $v_{x \cdot y}$. The evaluation of $e'_{x \cdot y}$ is in scope $s_{x \cdot y}$ and may utilise these bindings.

Considering this, a representation of $v_{x \cdot y}$ in scope $s_{x \cdot y}$ may therefore be calculated by retaining the values of $q_x(e_{x \cdot y})$, introducing a *iter* attribute with consecutive numbers and holding the *pos* attribute to the constant value 1. In terms of algebra, the representation of $v_{x \cdot y}$ is computed like this:

$$q_{x \cdot y}(\$v_{x \cdot y}) = \frac{pos}{1} \times \pi_{iter:inner,item}(\varrho_{inner:(iter,pos)}(q_x(e_{x \cdot y}))) \tag{2.2}$$

The introduction of the *inner* attribute is used to denote evaluation of the loop in scope $s_{x \cdot y}$. The *iter* attribute of $q_x(e_{x \cdot y})$ can be viewed as an atttribute *outer*, as it denotes the iterations in the outer loop of scope s_x.

Loop lifting requires maintenance of a *loop* relation to ensure independent iterations. The iterator body in scope $s_{x\cdot y}$ needs to be evaluated once for each binding of the iterator variable $v_{x\cdot y}$. Thus, the *loop* relation needs to be redifined based on $q_{x\cdot y}(v_{x\cdot y})$:

$$loop_{x\cdot y} = \pi_{iter}(q_{x\cdot y}(v_{x\cdot y})) \tag{2.3}$$

2.5.5 Free variables

XQuery expressions may use any iterator variable bound in enclosing scopes. That is, v_x bound in scope s_x may also be referred to within any of its child scopes. When looking at one of these child scopes, $s_{x\cdot y}$, by itself, the variable $v_{x\cdot y}$ appears to be a free variable.

Consider a iterator expresion $I_{x\cdot y}$ within another iterator expression I_x, both with iterator sequences of length two. If v_x is referred to within scope $s_{x\cdot y}$, from $s_{x\cdot y}$'s point of view, v_x is free. For each binding of v_x in the *outer* iteration expression, two evaluations of the *inner* iteration expresion occur. A relation capturing the relationship between number of iterations of these two iterator expressions can be defined like this:

outer	*inner*
1	1
1	2
2	3
2	4

Where a tuple $(outer, inner)$ is read as for the *inner*th iteration of the inner iterator expression, the outer iterator expression is in its *outer*th iteration. This relation is called $map_{x,x\cdot y}$ as it maps representations between scopes s_x and $s_{x\cdot y}$. It can be calculated like this:

$$map_{x,x\cdot y} = \pi_{outer:iter,inner}(\varrho_{inner:(iter,pos)}(q_x(e_{x\cdot y}))) \tag{2.4}$$

With this relationship defined it is now possible to represent the free variable v_x in the scope $s_{x\cdot y}$ with the help of an equi-join:

$$q_{x\cdot y}(\$v_x) = \pi_{iter:inner,pos,item}(q_x\ (\$v_x) \bowtie_{iter=outer} map_{x,x\cdot y}) \tag{2.5}$$

2.5.6 Mapping back

All steps and equations presented so far have been helpful to represent sequences and variables in a lower scope level. But the result of a query will have to be in form of its representation in the outermost scope s. So a way to represent an expression $e'_{x,y}$ in its scope's parent scope s_x is needed. Once again the *map* relation may be of use, combined with an equi-join:

$$q_x(e'_{x\cdot y}) = \begin{array}{l}\pi_{iter:outer,pos:pos1,item}(\\ \quad \varrho_{pos1:(iter,pos)/outer}(q_{x\cdot y}(e'_{x\cdot y}) \bowtie_{iter=inner} map_{x,x\cdot y}))\end{array} \tag{2.6}$$

2.5.7 Other expression types

The sequence construction e_1, e_2 is essentially a disjoint union of the relational representations of the expressions, that is, $q_x(e_1)$ and $q_x(e_2)$. By temporarily adding an attribute *ord* to these relations before a renumbering of the result with ϱ, the proper ordering of the sequence is acquired. Construction of sequences can therefore be expressed like this:

$$q_x(e_1\texttt{,}\ e_2) = \begin{array}{l} \pi_{iter,pos:pos1,item}\,(\\ \quad \varrho_{pos1:(ord,pos)/iter}\,(\\ \qquad \left(\tfrac{ord}{1} \times q_x(e_1)\right) \dot\cup \left(\tfrac{ord}{1} \times q_x(e_2)\right)\!)) \end{array} \tag{2.7}$$

The $\circledcirc$ operator meets the requirement of evaluating comparison and arithmetic operations on atomic values. Given two XQuery values e_1 and e_2 in multiple iterations, with relational representations as before, the expression e_1 `+` e_2 can be translated by first joining $q_x(e_1)$ and $q_x(e_2)$ on their iteration number, i.e. *iter*. Then, for each tuple, store the sum of the values of both of the *item* attributes, before cleaning up the resulting relation with a project. Expressed as an equation, the translation of sum expressions looks like this:

$$q_x(e_1\ \texttt{+}\ e_2) = \begin{array}{l} \pi_{iter,pos,item:res}\,(\\ \quad \oplus_{res:(item,item')}\,(\\ \qquad q_x(e_1) \bowtie_{iter=iter'} \\ \qquad\quad (\pi_{iter':iter,item':item}(q_x(e_2))))) \end{array} \tag{2.8}$$

The `if(`e_1`) then` e_2 `else` e_3, is one of the more complex translations of Loop Lifting. First the boolean expression e_1 is compiled. The result is split into two new loop relations, *loop*1 and *loop*2, which uses selection on all *true* and *false* values respectively. *loop*2 is used as current *loop* relation for the compilation of e_2 and *loop*3 as *loop* relation for the mapping of e_3. A equi-join with their corresponding *loop* relation on *iter* will filter out all unnnecesary iterations. The result is the union of both branches.

$$\begin{array}{l} q_x(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) = \\ \quad \pi_{iter,pos,item}(q_x(e_2) \bowtie_{iter=iter'} (\pi_i ter' : iter(loop2))\,\dot\cup \\ \qquad \pi_{iter,pos,item}(q_x(e_3) \bowtie_{iter=iter'} (\pi_i ter' : iter(loop3)) \\ \quad loop2 = \pi_{iter}(\sigma_{item=TRUE}(q_x(e_1))) \\ \quad loop3 = \pi_{iter}(\sigma_{item=FALSE}(q_x(e_1))) \end{array} \tag{2.9}$$

2.5.8 Example

Only looking at equations may be a bit too abstract to fully understand Loop Lifting. To concretise we will present a simple example of evaluating a query with the method and show intermediate results. The naming of expressions, scopes and variables will, where possible, be the same as earlier in this section. This query is the basis of this evaluation:

$$s\left\{\begin{array}{l} \texttt{for \$v0 in (10,20) return} \\ s_0\left\{\begin{array}{l} \texttt{(\$v0, for \$v00 in (7,8,9) return} \\ s_{0,0}\,\{\texttt{\$v0 + \$v00)} \end{array}\right. \end{array}\right.$$

The goal of the evaluation is, after all other calculations, to have a representation of e_0' in scope s, that is, $q(e_0')$. This is done by nesting inwards until the deepest scope, while calculating needed helping relations on the way, before evaluating the subexpressions one by one as one nests outwards until the outermost scope.

Firstly a representation of the outermost loop is needed. With the help of equation 2.1 we find a representation of `(10, 20)` in scope s, $s(e_0)$. Then, employing equation 2.2 yields `$v0` in scope s_0, the result of which is shown in Figure 2.18(a). $loop_0$ and $map_{,0}$ can now be calculated by using equations 2.3 and 2.4 and are shown in Figure 2.18(b) and 2.18(c) respectevely (remember $loop$ consists of a single tuple with value 1).

pos	*iter*	*item*
1	1	10
1	2	20

(a) q_0(`$v0`)

iter
1
2

(b) $loop_0$

outer	*inner*
1	1
1	2

(c) $map_{,0}$

Figure 2.18: Outer loop intermediate results

Before we evaluate the sequence expression in scope s_0, we need to evaluate the inner `for` loop. By the same measure as with the outer loop we first calculate $q_{0\cdot 0}$(`$v00`), $loop_{0\cdot 0}$ and $map_{0,0\cdot 0}$. The results are shown in Figure 2.19.

pos	*iter*	*item*
1	1	7
1	2	8
1	3	9
1	4	7
1	5	8
1	6	9

(a) $q_{0\cdot 0}$(`$v00`)

iter
1
2
3
4
5
6

(b) $loop_{0\cdot 0}$

outer	*inner*
1	1
1	2
1	3
2	4
2	5
2	6

(c) $map_{0,0\cdot 0}$

Figure 2.19: Inner loop intermediate results

To be able to calculate the sum-expression, $e'_{0\cdot 0}$, we first need a representation of the variable `$v0` in scope $s_{0\cdot 0}$. As this variable is a free variable in this scope, this is done by applying equation 2.5 on q_0(`$v0`) (Figure 2.18(a)). This result is shown in Figure 2.20(a). Now that we have both `$v0` and `$v00` expressed in scope $s_{0\cdot 0}$, we can employ equation 2.8 to sum the two variables together. The resulting relation can be seen in Figure 2.20(b).

pos	*iter*	*item*
1	1	10
1	2	10
1	3	10
1	4	20
1	5	20
1	6	20

(a) $q_{0\cdot 0}$(`$v0`)

pos	*iter*	*item*
1	1	17
1	2	18
1	3	19
1	4	27
1	5	28
1	6	29

(b) $q_{0\cdot 0}(e'_{0\cdot 0}) = q_{0\cdot 0}$(`$v0 + $v00`)

Figure 2.20: Innermost expression intermediate results

The result of the summation, $q_{0\cdot 0}(e'_{0\cdot 0})$ is expressed in scope $s_{0\cdot 0}$ and will have to be mapped up to scope s_0. This is done with help from equation 2.6 and $map_{0,0\cdot 0}$ which we calculated earlier, and the result can be seen in Figure 2.21(a). With $q_0(e'_{0\cdot 0})$ evaluated, and with q_0(`$v0`) from earlier, the sequence building can be completed. This operation requires equation 2.7, and yields the relation shown in Figure 2.21(b).

pos	*iter*	*item*
1	1	17
2	1	18
3	1	19
1	2	27
2	2	28
3	2	29

(a) $q_0(e'_{0\cdot 0})$

pos	*iter*	*item*
1	1	10
2	1	17
3	1	18
4	1	19
1	2	20
2	2	27
3	2	28
4	2	29

(b) $q_0(e'_0)$

pos	*iter*	*item*
1	1	10
2	1	17
3	1	18
4	1	19
5	1	20
6	1	27
7	1	28
8	1	29

(c) $q(e'_0)$

Figure 2.21: Intermediate and final result

Finally the built sequence will have to be expressed in terms of scope s. Achieving this only requires the use of equation 2.6 one last time in combination with $map_{,0}$. The complete result of the query is shown in Figure 2.21(c).

2.6 Summary

This chapter has described XQuery and its semantics, including FLWOR expressions, paths and predicates, and the XQuery Core subset language and its normalisation rules. Further, existing implementations have been examined and compared, and we have shown that Pathfinder is one particularly interesting implementation because of its dependence on a relational back end and thus relational algebra.

Relational algebra and its semantics were described to set the stage for a description of the target language which will be presented in the next chapter. Parser generators and other relevant parser technology has been presented, as well as some strategies for parsing syntax trees. Finally, the Loop Lifting method employed by the Pathfinder project to translate XQuery to relational algebra was described.

In the next chapter, important architectural decisions and methods are presented.

Chapter 3

Method

This chapter describes the most important decisions and constraints for this project. Essentially, this chapter describes *what will be done* along with rationales for decisions that have been made. Implementation-specific details are left to Chapter 5. This chapter is initiated by Section 3.2, where a method for tree parsing is chosen. Further the method for rewriting the syntax tree is explained in Section 3.3. Finally we present the target algebra language (MQL), and the chapter ends with a description of a method for calculating algebra complexity which will be used in chapter 6 for comparisons.

3.1 Development of a novel translation method

In Section 2.5, "Loop Lifting" was presented. This is a translation method used by Pathfinder (section 2.2.4) to translate XQuery to relational algebra for execution on MonetDB. As the goal of this thesis is to find a way to translate queries in this language to MQL, the MARS Query Language, one possibility would be to modify the Loop Lifting methodology to produce MQL trees instead of algebra for MonetDB. However, this modified method would most likely not utilise the full expressiveness and power of MQL. This lead to the development of a MARS specific method of translation, "Tainting Dependencies", which will be reviewed in full in chapter 4. Another incentive for not modifying Loop Lifting is that it produces large and quite complex operator trees, which lead to unnecessarily denormalised intermediate results. This is discussed further in Section 7.8.3 on page 127.

3.2 Tree parsing

In Section 2.4.4 some common design patterns for tree parsing were described. Based on some important traits of XQuery, such as being an *orthogonal language* (see section 2.1.1), it seems natural to employ the *context-sensitive visitor pattern*, as described in section 2.4.4. The benefits of this decision is the possibility to create specialised visitors for certain tasks, such as rewrite visitors for preprocessing the syntax tree (hinted upon in section 2), or predicate visitors, or any situation where propagation of context/state information is required.

The implementation of the context-sensitive visitor pattern is described in Section 5.6.

3.3 AST rewriting

Definition 2. *The process of* ***normalisation*** *of XQuery is defined as the process of translating the whole or parts of an XQuery abstract syntax tree into an XQuery Core abstract syntax tree, as described in Section 2.1.6.*
Definition 3. *An abstract syntax tree is said to be* ***denormalised*** *if it has not yet been normalised through a process of normalisation.*

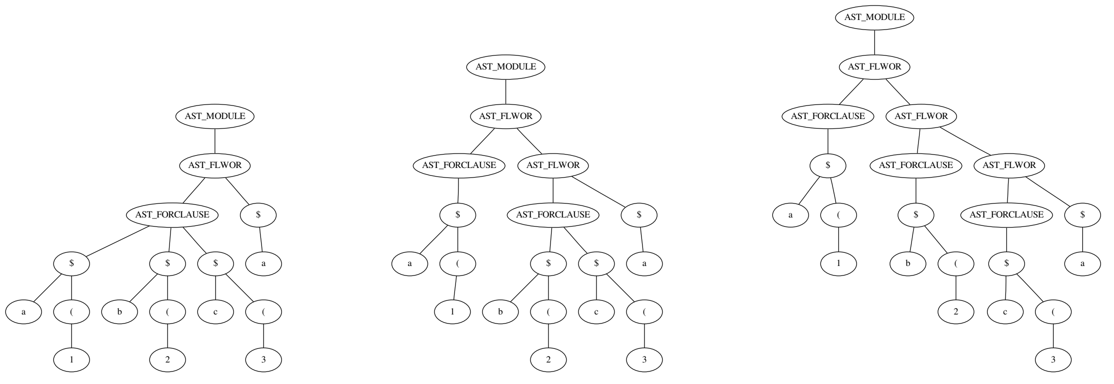
The basic abstract syntax tree produced by the XQFT Parser (see section 2.4.3) is in a denormalised form. As explained in Section 2.1.6 on page 8, normalisation has the benefit of simplification without loss of expressiveness or semantics, and thus simplifying the task of translating the syntax tree.

Based on findings in Section 2.1.4 regarding the structure of FLWOR expressions, it was briefly mentioned that complex FLWOR expressions could be transformed into semantically equivalent and simpler representations. A question was also posed – *could it be benefitial to rewrite complex FLWOR expressions into a simpler form*? We will now attempt to answer this question.

3.3.1 Normalising FLWOR expressions

A denormalised XQuery syntax tree may contain nodes such `for`-clause expressions with several variable declarations and assignments as can be seen in Figure 2.6(a). Such an expression could be normalised to several nested FLWOR expressions, as shown in Figure 2.6(b), as defined by the mapping rules in figures 2.2, 2.3, 2.4, and 2.5.

Here, the method in Section 2.1.6 has been applied to the example tree in Figure 3.1(a), and the result can be seen in Figure 3.1(c). The corresponding hypothetical source code for the denormalised and normalised trees can be seen in Figure 3.2.



(a) Step 1: original denormalised syntax tree (b) Step 2: partly normalised tree (c) Step 3: fully normalised tree

Figure 3.1: Normalisation steps of FLWOR expression with multiple `for`*-clauses*

```
for $a in (1),
    $b in (2),
    $c in (3)
      return $a
```
(a)

```
for $a in (1) return
  for $b in (2) return
    for $c in (3) return $a
```
(b)

Figure 3.2: Hypothetical source code corresponding to the trees in figures 3.1(a) and 3.1(c)

Considering these trees, we can assert the following:

1. Multiple nested FLWOR constructs with a single `for/let`-clause and a single variable binding maintains the semantics of a single FLWOR with multiple `for/let`-clauses and variable bindings
2. A FLWOR construct with a single `for/let`-clause and a single variable binding is easier to parse than a FLWOR construct with multiple `for/let`-clauses and bindings because it is known that there is only one `for/let`-clause with a variable binding, and there is no need to look for others within the same FLWOR construct

Based on these findings, it is natural to conclude that normalisation of FLWOR expressions is benefitial for the sake of simplicity in the tree parsing process. Normalisation may be attained by executing a specialised *rewrite visitor* on the syntax tree, which will be described in Chapter 5.

3.4 Target relational algebra language

Given as premise for this project was the target algebra language. This algebra is a dialect of classic relational algebra (see section 2.3) developed by Fast Search & Transfer. The language is called MQL (MARS Query Language). This section will briefly describe some of the operators of this language as well as the indexes they operate on. Further, it will demonstrate the usage as well as some important traits of this language with a few examples.

Note that this section is largely based on documentation provided by FAST which has been deemed company confidential.

3.4.1 General concepts

All queries for MQL are written as strings with a syntax reminiscient of Lisp dialects. An example query can be seen in Figure 3.4.

Data is stored in *indexes* rather than in DOM trees. Naturally, this is a prerequisite for efficient execution of relational algebra on the data. There are two basic index types: *occurence* indexes and *value* indexes. In the case of an occurence index, a lookup for a term will map the term to the occurences of this term. In a *value* index, a key (for example document id) is associated to one or more *values* in the document collection. Lookups in *value* indexes can be combined with lookups in *occurence* indexes to add data to the result set for the user. Throughout this chapter and later chapters, any reference to a value occurence index (*valocc*) implies exactly such a hypothetical combination of occurence and value indexes.

The schema of the indexes determins the fields/columns in the result sets from a lookup, but typically these three fields will be included on a lookup of a term in the *occurence* index:

- Document ID: Internal identifier for the document in which the term occurs
- Position: The term position in the document (counted by terms, not characters or nodes)
- Scope: the context of the occurence, e.g `/a/b`. Note that the scope also contains metadata about the *instance* of the scope (this is visualised like this: `a[1].b[2]`, which reads as "the first instance of `a` and the second instance of `b` within this `a`")

Additionally, a *Value* field can be added by combining the result set from an *occurence* index with the equivalent result set from a *value* index. Figure 3.3 shows an example result set from a lookup in a hypothetical *valocc* index.

DocumentID	*Position*	*Scope*	*Value*
1	7	a[1].b[2]	*c*
1	8	a[1].b[3]	*c*
1	11	a[1].b[4]	*c*

Figure 3.3: Example result set from a lookup in valocc

The input for operators are result sets from child operators. The output from operators is a single result set. Note however that some operators, such as `index()` will modify the context for child operators. This feature is not documented, and the exact mechanics are not known. However their respective behaviours are described here where applicable.

3.4.2 Language syntax

The syntax of MQL (MARS Query Language) is, as mentioned above, reminiscent of Lisp dialects. Consider the example in Figure 3.4. This example will lookup the term "c" in the scope `/a/b` in the index *valocc*.

```
index(valocc;
  scope(/a/b;
    lookup(c)));
```

Figure 3.4: Simple MQL example

The syntax for MQL can be described in a condensed form using EBNF notation as can be seen in Figure 3.5

```
OPERATORNAME  ::= IDENTIFIER
OPERATOR      ::= OPERATORNAME "(" PARAMETERLIST?
                  (";" OPERATORLIST)? ")"
OPERATORLIST  ::= OPERATOR ( "," OPERATOR )*
```

Figure 3.5: Simplified MQL EBNF

Note that `PARAMETERLIST` has no definition. This production will be described for each operator, if applicable.

Also note that parameters and child operators are separated by a semicolon. For example, in the MQL expression `index(valocc;lookup(hairdresser))` the `index()` operator is given one parameter (`valocc`) and one child operator (`lookup(hairdresser)`).

3.4.3 Operators

Lookup

The `lookup()` operator performs a lookup in the default index (if none other defined by an `index()` operator, see 3.4.3). The result set returned from this operator contains all occurences of the given term, if any. This operator will use the last index specified by the `index()` operator, otherwise the default index. See Figure 3.4 for an example of usage.

Input: none (parameters only)

Output: result set with occurences and/or values, depending on index (see Section 3.4.1)

Scope

The `scope()` operator accepts one parameter and one child operator. Informally, the result set from the child operator is filtered to match the given scope. See Figure 3.4 for an example of usage.

Input: a scope string (i.e `/a/b`), and a result set from the child operator which will be filtered for the given scope string

Output: a filtered result set with matches only within the given scope

Index

This operator modifies the behaviour of the child operator such that any lookup will use the specified index. See Figure 3.4 for an example of usage where the child operators will operate on the index *valocc*.

Input: result set from the child operator

Output: result set from the child operator

Project

The project() operator lends its name and semantics from the project and the rename operator known from basic relational algebra (Section 2.3.1). Note that it is also possible to execute functions and apply constant values to the projection. The example in Figure 3.6 may produce an output similar to that of Figure 3.6.

Input: projection parameters, and a result set from the child operator

Output: projection of result set on the given parameters

```
project(id=DocumentID, cid=max(100,DocumentID), one="1");
  index(valocc;
    scope(/a/b;
      lookup(c))));
```

Figure 3.6: Simple MQL project() example with inline function call and an applied constant field

id	*cid*	*one*
45	100	1
103	103	1
90	100	1
33	100	1
289	289	1

Figure 3.7: Hypothetical result of query in figure 3.6

Select

The `select()` operator filters tuples from the child operator based on boolean function predicates. Consider the example in figure 3.8, where `eq()` compares the two parameters and returns *true* if the parameters are equal. A hypothetical result (based on the example for project()) can be seen in figure 3.9.

```
select(eq(100, cid);
  project(id=DocumentID, cid=max(100,DocumentID), one="1");
    index(valocc;
      scope(/a/b;
        lookup(c)))));
```

Figure 3.8: Simple MQL `select()` example

id	*cid*	*one*
45	100	1
90	100	1
33	100	1

Figure 3.9: Hypothetical result of query in figure 3.8

Input: predicate expression parameter, and a result set from a child operator

Output: filtered result set from the child operator

Join

A join operator (one of hhjoin(), hljoin(), or `mergejoin()`) performs an equi-join operation as described in section 2.3.2. In the case of a `mergejoin()`, the input result sets must be sorted.

The complete syntax for any of the join operations can be described with EBNF notation as can be seen in Figure 3.10. A trivial usage example of the `mergejoin()` operator can be

```
JOINFIELD    ::= ("left." | "right.")? FIELDNAME
PROJECTFIELD ::= (FIELDNAME "=")? JOINFIELD
PROJECTLIST  ::= PROJECTFIELD ("," PROJECTFIELD)*
JOINNAME     ::= "hhjoin" | "hljoin" | "mergejoin"
OPERATOR     ::= JOINNAME "(" "[" FIELDLIST "]" ","
                 "[" FIELDLIST "]" "," "[" PROJECTLIST "]"
                  ("," "left" | "right" | "full")? ";"
                  OPERATOR "," OPERATOR ")"
```

Figure 3.10: Join operator EBNF

seen in Figure 3.11, where the result sets from two hypothetical child operators `Query1()` and `Query2()` are joined on their document ids, and the result is projected on the fields *DocumentID* and *Position*.

```
mergejoin([DocumentID], [DocumentID], [DocumentID, Position];
  Query1(..),
  Query2(..))
```

Figure 3.11: Simple MQL `mergejoin()` example

Input: join specification parameters, and result sets from child operators

Output: joined result set based on join specifications

Make

The `make()` operator is used to synthesise result sets from the given (constant) arguments. Field names can be specified, but are not required. The default field names are *field0*, *field1*, . . . , *fieldN*, where N is the number of fields. The example in Figure 3.12 will produce the output seen in figure 3.13.

```
make(1,2,3)
```

Figure 3.12: Trivial MQL `make()` example

$field0$	$field1$	$field2$
1	2	3

Figure 3.13: Hypothetical result of query in figure 3.12

A more complex example can be seen in 3.14, where field names are specified, and several tuples are synthesised. The corresponding result is shown in Figure 3.15. Notice how values are specified column-wise.

Input: tuple specification parameters

Output: result set consisting of one or more tuples according to tuple specification

```
make(name:=["A","B","C"], [1,4], [2,5] [3,6])
```

Figure 3.14: MQL `make()` example

A	*B*	*C*
1	2	3
4	5	6

Figure 3.15: Hypothetical result of query in figure 3.14

Group

```
OPERATOR ::= "group" "(" GROUPINGFIELDS ["," AGGREGATORS ]
                 ";" OPERATOR ")"
```

Figure 3.16: Group operator EBNF

The notation of the `group()` operator can be seen in the EBNF sepecification of figure 3.4.3. The operator will group tuples with the same value for the fields specified in `GROUPINGFIELDS`, and create one tuple per group. Fields not specified in the operator will not be a part of the result relation. `AGGREGATORS` may be used to specify aggregator functions to be run within each group. The result of an aggregator will be added as a field for the result tuple corresponding to the group it operated on.

```
group((cid), max(id), count(); Query1)
```

Figure 3.17: Simple MQL `group()` example

If `Query1` evaluates to the relation shown in Figure 3.7, the query of Figure 3.17 will result in the relation of Figure 3.18.

cid	*max*	*count*
100	90	3
103	103	1
289	289	1

Figure 3.18: Result of the query in Figure 3.17

Input: grouping specification, and a result set from the child operator

Output: grouped result set based on grouping specification

Cross

This operator computes a cross product between its two child operators. This operator does not take any parameters.

Input: result sets from two child operators

Output: the cross product between the two input result sets

3.4.4 Assumed functionality

Some extra functionality is required to achieve the goals of this project. This section describes new operators and functions, and their respective behaviour. Additionally, the rationales for assuming these new functionalities are explained.

Numberation/sequence generator

Order is an inherent concept of the XQuery data model, to accomodate for this a numbering operator such as `numberate` is needed. The syntax for this operator can be defined in EBNF notation as seen in figure 3.19. Given a sort order defined by the fields in `SORTLIST`, the operator numbers consequtive tuples of the relation returned from `OPERATOR`, recording the row number in the new field `FIELD`. Row numbers start from 1 in each partition defined by the fields in `PARTITIONLIST`. If no `SORTLIST` is defined, the original sort of the input relation is used. The fields specified in `SORTLIST` is not a part of the result relation. The operator is comparable to the `DENSE_RANK` operator defined by SQL:1999 [17]. No particular order is required in the result relation.

```
FIELD         ::= IDENTIFIER
SORTLIST      ::= "[" FIELD ("," FIELD)* "]"
PARTITIONLIST ::= "[" FIELD ("," FIELD)* "]"
OPERATOR      ::= "numberate" "(" FIELD ("," SORTLIST)?
                  ("," PARTITIONLIST)?  ";" OPERATOR ")"
```

Figure 3.19: EBNF definition for the `numberate()` operator

```
numberate(Seq, [C], [A];
  make(name:=["A","B","C"]; [1,1,1,2,2,2], [a,b,c,a,b,c],[6,5,4,6,5,4]))
```

Figure 3.20: Trivial example of `numberate()` usage

A	*B*	*C*
1	a	6
1	b	5
1	c	4
2	a	6
2	b	5
2	c	4

(a) Input:

Seq	*A*	*B*
1	1	c
2	1	b
3	1	a
1	2	c
2	2	b
3	2	a

(b) Output:

Figure 3.21: Input/output result sets

The example in Figure 3.20 illustrates the behaviour of this operator. In the output result set seen in figure 3.21(b), a column `Seq` has been added, with a number sequence which follows a sort order of C and restarted for each new value of A. Also note the C attribute is not part of the result relation.

Scope comparison functions

These boolean functions (not operators) accepts two arguments which must be *scope* fields from index lookups (as described in Section 3.4.1). They each correspond to one of the axes of XQuery/XPath. Informally, for one tuple, the *scope* field has a value on this format: a_1[m].a_2[n]. . . a_j[o]. In this format a_x are scope names and m, n and o are integers. Further, a_2 is the nth scope with the name a_2 which resides within the a_1 scope. a_1 is the mth scope with that name of the document. In the following description of the functions a_1[m] is called a *step* and [m] is called the *instance*. $scope_n$ is to be read as the value in the *scope* attribute for one tuple. The length of $scope_n$ is equal to the number of steps it contains.

isChild($scope_1$, $scope_2$)	Returns *true* if $scope_2$ is exactly on step longer than and a prefix of $scope_1$. Else $false$.
isDescendant($scope_1$, $scope_2$)	Returns *true* if $scope_2$ is longer than and a prefix of $scope_1$. Else $false$.
isSelf($scope_1$, $scope_2$)	Returns *true* if $scope_2$ is equal to $scope_1$. Else $false$.
isDescendantOrSelf($scope_1$, $scope_2$)	Returns *true* if $scope_2$ is equal to or a prefix of $scope_1$. Else $false$.
isFollowing($scope_1$, $scope_2$)	Returns *true* if $scope_2$ has a higher instance for a step corresponding to a step in $scope_1$. Else $false$.
isFollowingSibling($scope_1$, $scope_2$)	Returns *true* if $scope_2$ is equal to $scope_1$ except having a higher instance of the last step. Else $false$.
isParent($scope_1$, $scope_2$)	Returns *true* if $scope_1$ is exaclty one step shorter than and a prefix of $scope_2$. Else $false$.
isAncestor($scope_1$, $scope_2$)	Returns *true* if $scope_1$ is shorter than and a prefix of $scope_2$. Else $false$.
isAncestorOrSelf($scope_1$, $scope_2$)	Returns *true* if $scope_1$ is equal to or a prefix of $scope_2$. Else $false$.
isPreceding($scope_1$, $scope_2$)	Returns *true* if $scope_2$ has a lower instance for a step corresponding to a step in $scope_1$. Else $false$.
isPrecedingSibling($scope_1$, $scope_2$)	Returns *true* if $scope_2$ is equal to $scope_1$ except having a lower instance of the last step. Else $false$.

As can be seen, many of these functions can be defined by some of the other functions. But for readability we assume one function per axis.

xqBoolean/boolean truth value coercion

This function (not operator) accepts one argument and determines its *truth value*. XQuery truth values (effective boolean values) were described in Section 2.1.1.

The semantics of this function is captured in the example in figure 3.22. For the input result set produced by the `make()` operator in Figure 3.23(a), the output result in Figure 3.23(b) is produced. The example converts the integer 5 to *true* and the integer 0 to *false*.

```
project(truthVal=xqBoolean(B);
  make(name:=["A","B","C"], [1,4], [0,5] [3,6]))
```

Figure 3.22: Example of xqBoolean() usage

A	B	C
1	0	3
4	5	6

(a) Input for example in figure 3.22

$truthVal$
false
true

(b) Output from example in figure 3.22

Figure 3.23: Input/output result sets

Union/disjoint union

The `union()` operator accepts multiple operators, and will behave as the standard relational algebra operator (Section 2.3.1), except it does not remove duplicates (as disjoint union). The function of the operator is to concatenate the relations returned from its child-operators. No particular ordering of tuples in the result relation is required.

3.5 Calculating complexity in relational algebra

A method for indicating complexity of relational algebra trees was suggested by Øystein Torbjørnsen at Fast Search & Transfer. This method is based on the assumption that the algebra will be executed on an implementation written in Java or a similar object oriented language. Not withstanding the benefits of compile-level optimisations and other ways to increase performance such as caching, this method of complexity indication defines complexity as creation of new objects in run-time, and the cost of sort and join operations.

This definition of complexity does *not* account for disk I/O, nor is it a direct measurement of performance. However, given an algebra tree which is to be executed on some known host implementation, it may give an indication of spending of time and computational resources.

3.5.1 Tuple and field creation

Definition 4. *A **field** is an in-memory object which contains a value and a mapping to an attribute name in a relation*
Definition 5. *A **tuple** is defined as an in-memory object which contains a set of fields (Definition 4), where each field contains a value for some given attribute in the relation*

Semantics

A new tuple is assumed to be created every time an old tuple needs to add or remove one or more fields. A new field is assumed to be created for every new value for any attribute for any tuple.

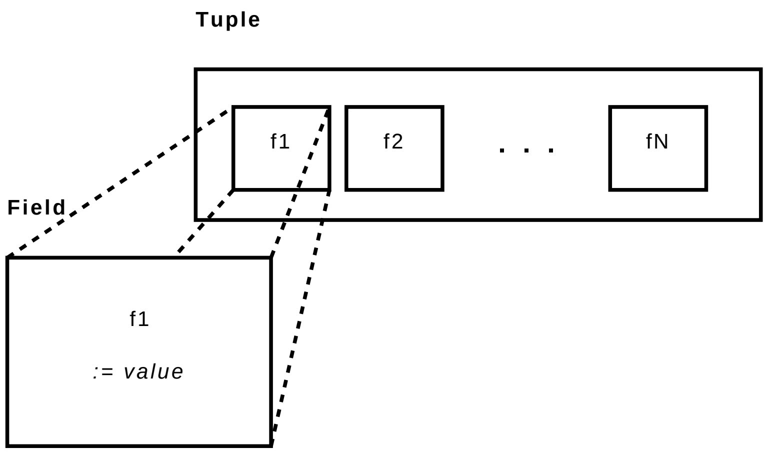


Figure 3.24: The structure of a tuple with fields

Implications of semantics

Any kind of projection of x tuples over n "new" attributes and o "old" attributes generates x new tuples and n new fields (note that duplicates are not removed). Old fields are reused, and attribute renaming is naturally capable of reusing old fields.

Any kind of tuple construction (e.g using the make() operator) for n tuples and m fields generates n new tuples and $n \times m$ new fields.

Assumptions

Pathfinder generates relational algebra which contains some operators for which the behaviour is unknown. Some assumptions have made about these operators:

- The `Join` operator is assumed to produce $n \times m$ fields for an input of n tuples and m fields
- The behaviour of the `Diff` operator is unknown, it is assumed to produce 0 tuples and 0 fields
- The behaviour of the `Distinct` operator is unknown, in favour of Pathfinder it is then assumed to produce 0 tuples and 0 fields
- The `Not` operator is assumed to perform an inversion of boolean values and thus produces n tuples and m fields for an input of n tuples and m fields (where m is always 1 in the comparisons made in this dissertation)
- The `Cast` operator is assumed to produce n tuples and 1 field for an input of n tuples and m fields
- The `Attach` operator is assumed to produce n tuples and m fields, where m is the number of fields constructed by the `Attach` operator and n is the number of input tuples

3.5.2 Join and sort tuple I/O

Definition 6. *The **input** of an operator is defined as the sum of tuples* entering *the operator*

Definition 7. *The **output** of an operator is defined as the sum of the tuples* produced *by the operator*

Semantics

For every operator that either performs a equi-join or sort operation, the number of *input* and *output* tuples are counted. The minimum, maximum, and averages are calculated for both *input* and *output.*

3.5.3 Total complexity

Definition 8. *Measurement of **complexity** is defined as that for some given operator α, counting the following:*

- *Creation of new **tuples** (definition 5)*
- *Creation of new **fields** (definition 4)*
- *If α performs an equi-join or a sort operation, all **input**(Definition 6) and **output** (Definition 7) tuples*

*The integer sum of counting field and tuple creations (from and including 1 and up) defines the **complexity** for α. Further, the sum of all the complexity sums in some given relational algebra tree defines the **complexity sum** for that given tree. Additionally, the minimum, maximum and average input and output tuples defines the **tuple I/O complexity** for that given tree.*

3.6 Summary

In this chapter the most significant decisions for this project have been presented and explained in detail. In the next chapter, we will present our novel methodology for translating XQuery queries to relational algebra, dubbed "Tainting Dependencies".

Chapter 4

Tainting Dependencies

One of the greatest challenges in translating XQuery to relational algebra, is to translate the semantics of iterators. The iterative nature of expressions such as `for` expressions and the bulk oriented relational processing may seem contradictory. But because XQuery is a purely functional language, and thus free of side effects, it is semantically sound to evaluate all iterations of the iterator body in parallel.

In this chapter we will first present MarkXRemove. This was the first iterator translation proposal of this project. As it had some flaws, the method was reinvented and evolved into Tainting Dependencies (TD), accomodating for the deficiencies of MarkXRemove. The rest of the chapter is dedicated to this method. First some concepts will be discussed, followed by methods for translating various features of XQuery into MQL relational algebra trees, including the translation of iterators. Finally, the chapter will discuss some possible simplifications of trees generated with TD.

4.1 MarkXRemove

Our original proposition to a method for translating XQuery ASTs into relational algebra was named MarkXRemove. Even though it has many shortcomings and flaws, we will in this section give a quick overview of the method. This is because the Tainting Dependencies method is an evolution and a refinement of MarkXRemove, and may be easier understood when seen in the perspective of its origins. Another reason is that in case of further development of TD, it may be of help to also know what will *not* work, what will work partially and *why* it is flawed.

4.1.1 Basics

The foundation of the method is that an iterator expression is always translated by calculating the cartesian product of the iterator sequence and the iterator body, hence the "X" in the name. "Remove" stems from the removing of tuples who ends up in the wrong iteration in the cross product result. The cartesian product and the selection of tuples afterwards actually constitutes a kind of natural join (section 2.3.2) as we will see later.

As the translator comes across an iterator variable declaration with the variable name χ, it will augment the relational representation of the iterator sequence belonging to this variable with an attribute $\chi numb$. This new attribute will hold consecutive values from 1 to n for a n item long sequence. These values symbolise the iteration number of the iterator expression seen isolated from possible other surrounding iterator expressions. A function counter() returning the row number of a relation and a project operator will handle the augmentation. The corresponding algebra tree is stored in the symbol table. The "mark" of the name of the method is because of this augmentation.

4.1.2 FLWOR

If the translator later comes across a reference to an iterator variable χ, it will get the tree from the symbol table and return it to the referring AST node without any further ado. The translator is also required to know which subtrees have a child that has referred to which iterator variable. This is because the $\chi numb$ attribute could be lost in a project operation without this knowledge.

When the translator returns to the iterator expression node for the variable χ after traversing the iterator body, it will, as mentioned before, make sure that the cartesian product between the iterator sequence and body is calculated. From the result of this, the tuples where the $\chi numb$ stemming from the iterator sequence does not line up with the $\chi numb$ stemming from the body are removed. Any tuple with a $\chi numb$ value $NULL$ will be kept.

A $NULL$ value in the $\chi numb$ field of a tuple means that this tuple is not marked, i.e. it is not dependent on which iteration the χ iterator expression is in. Unmarked tuples can e.g. stem from the creation of a sequence. MarkXRemove translates sequence construction expressions, e.g. (e_1, e_1), to a simple disjoint union, $\mathbf{r}(e_1) \dot{\cup} \mathbf{r}(e_2)$. Where $\mathbf{r}()$ symbolises a function translating XQuery expressions into relational algebra.

The method creates quite simple algebra, as exemplified by the following query:

```
for $i in (1,2,3) return
    ($i, 'yes')
```

which is translated into this algebra:

```
select(ifThenElse(isNull(value), true, eq(r.inumb,l.inumb))
  cross(
    project(inumb = counter(), value;
      make(name:=["value"], [1,2,3]))
    union(
      project(inumb = counter(), value;
        make(name:=["value"], [1,2,3]))
      make(name:=["value"], ['yes']))))
```

4.1.3 Flaws

The main problem with MarkXRemove is that it requires a particular ordering of tuples in a relation which is a result of a cartesian product. Not only can the MQL cross operator not promise the particular ordering of its result, it can not promise any particular ordering at all. The ordering the method requires is that for each tuple in the left relation, the tuple is repeated for all tuples in the right relation. If this requirement is not met any item may appear anywhere in the resulting sequence, which is not acceptable for evaluation of XQuery expressions where all sequences are ordered.

Another problem with this method is that any sequence built which includes a reference to an iterator variable is not fully calculated until the cartesian product between the corresponding relation and the variables iterator sequence is evaluated. This makes it hard to evaluate expressions where such a sequence is a subexpression. The *iterator body* of this query:

```
for $i in (5,10) return
    ($i, 8) > (6,12)
```

would be translated into this relational algebra:

```
project(value = gt(l.value, r.value), inumb
  cross(
    union(
      project(inumb = counter(), value;
        make(name:=["value"], [5,10]))
      make(name:=["value"], [8]))
    make(name:=["value"], [3,12])))
```

which again would produce this relation:

inumb	*value*
1	*false*
1	*false*
2	*true*
2	*false*
NULL	*true*
NULL	*false*

The query should of course be evaluated to $(true, true)$, as the > operator in XQuery yields *true* if *one* item in the left operand is larger than *one* item in the right. This means that the relation would have to be pruned by a select or group, which can not be done generally in the relations current state. A possibility would be to postpone the pruning until after the relation is crossed with the iterator sequence, but there would still not be any trivial solution. This problem leads to the introduction of iterator dependency tainting, as we will see in Section 4.3.2.

4.2 Inference rule language explanation

During this chapter we will present some inference rules. Table 4.1 explains some of the the various typographical representations.

Inference rules are generally in this format:

$$\frac{condition}{e} \longmapsto \mathbf{r}(e)$$

$\longmapsto$	Translates into
ϑ	A set of iterator dependencies
sans serif	MQL expressions
`monospaced`	XQuery expressions
$e, \ldots, e_n$	Generic expressions
$\chi, \ldots, \chi_n$	Generic variable names
I_χ	The iterator expression which declares χ
bold	Operations done during the generation of the algebra
$\mathbf{r}(e)$	Returns the relational algebraic representation of e
$\mathbf{t}(\mathbf{r}(e),\vartheta')$	Returns $\mathbf{r}(e)$ tainted with the dependencies ϑ'
$\mathbf{B}(\mathbf{r}(e))$	Returns the effective boolean value of $\mathbf{r}(e)$.

Table 4.1: Explanation of inference rule symbols

This should be read as: if condition *condition* is satisfied, the XQuery expression e will be translated into $\mathbf{r}(e)$.

Often MQL operator trees are depicted like this:

operator1(. . . ,l.attr, r.attr. . . ;
  operator2(. . .);
  operator3(. . .))

This is to be interpreted as "operator2 is the left child of operator1 and operator3 is the right child". MARS does not allow attribute names to contain punctuation marks or allow two attributes with the same name within one relation. An operator combining two relations will therefore have renaming functionality. A typical projection list of such an operator combining two relations which both contain the attribute *attr* would look something like: . . . rattr=right.attr, lattr=left.attr. . . . To make the inference rules easier to read, this step has been dropped. The rules assume that the equal named fields will automatically be given a prefix l. (left) or r. (right) depending on which child the attribute stems from.

We assume the union-operator accepts relations with different schemas. The schema for the result relation can be described as:

$$schema(\mathsf{union}(\alpha, \beta)) = schema(\alpha) \cup schema(\beta).$$

The tuples which have more fields in the result relation than they did in the relation they stem from will have a *NULL* value for the introduced attributes. It may be not be desirable to implement the operator in such a way. In that case each child-relation will have to be augmented with the attributes needed with an project operator.

The effective boolean value function, $\mathbf{B}()$ is described in Section 4.3.3.

4.3 Basics

The method assumes left-to-right traversal of the assymetric syntax tree. The traversal is postorder, meaning a subtree can be evaluated independently from its ancestors. The relational algebra will thus be generated bottom-up. In addition to the evaluated subtree, a node must be able to inform its parent node about its iterator dependencies (ϑ), which we will discuss later.

One XQuery sequence is represented as one relation and one XQuery item is represented as one tuple. This is sound, as all XQuery items are sequences, and all sequences are one-dimensional (section 2.1.1). As we mentioned in Section 4.1, the MarkXRemove method did not actually consider the ordering of items in sequences at all. In Tainting Dependencies, however, we have introduced an attribute *index* holding the intra sequence number of the item. Consider the XQuery sequence `('a','b',...,'z')`. With this attribute, the relational representation will be as follwing:

index	*value*
1	`'a'`
2	`'b'`
...	...
n	`'z'`

As can be seen, the item value is stored in the *value* attribute. For the course of this chapter we will, for the sake of simplicity, treat *value* as a polymorphic type attribute. This simplification has minimal consequences for the method and the way XQuery expressions are translated. XQuery types and relational representation of such will is discussed in Section 7.4.

Also for simplicity, the *index*, *documentId*, *pos* and *scope* attributes have sometimes been left out of the attributes specified in project operators. If the project operator is applied to the result of a join or cartesian product, these fields will follow the *value* attribute if nothing else is specified. That is, if *r.value* is projected, then so is *r.documentId*, etc... if applicable.

Tainting Dependencies utilises a symbol table for storing of variables declared. The table has two functions:

- **put**(χ, **r**(e)) – will store the algebraic version of the expression bound to the variable \$$\chi$ with χ as the key.
- **get**(χ) – will do a lookup in the table based on the name of the variable \$$\chi$ and return the algebraic version of the expression linked to it.

The symbol table handles scoping according to XQuery semantics (Section 2.1.4), meaning the translator will always be able to find the right declared variable based on which node in the AST the translator is visiting.

4.3.1 Iterator Dependency Inheritance

The concept of iterator dependency form the basis of the Tainting Dependency method. Such dependency is defined as follows:

Definition 9. *An XQuery expression e is* ***dependent*** *on an iterator I_χ if e occurs within the iterator body of I_χ and if the evaluation of e depends on the iteration number of I_χ.*

An variable reference to an iterator variable \$$\chi$ is by this definition dependent on the iterator I_χ. Intuitively, an expression which subexpression is dependent on an iterator I_χ is also dependent on this iterator – we say the dependency is inherited. Consider the example subexpression of figure 4.1, where `$x` and `$y` both are iterator variables. Here, the expression e_1 is dependent on the two iterators $I_{\mathtt{x}}$ and $I_{\mathtt{y}}$, while expression e_2 is only dependent on $I_{\mathtt{x}}$.

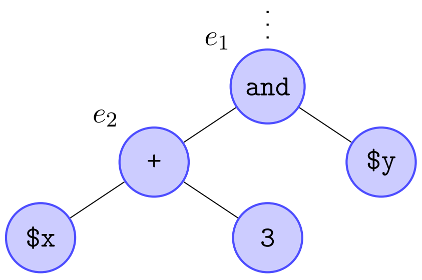


Figure 4.1: Iterator variable dependency

The iterator dependencies of an expression e are part of the set $e.\vartheta$. As mentioned earlier, an AST node must be able to inform its parent about the node's dependencies as well as the algebra generated. For an expression e this can be done by letting $e.\vartheta$ piggyback the $\mathbf{r}(e)$ returned. The variable dependencies for an expression e with the subexpressions $e_1, \ldots, e_n$ can be described as following:

$$e.\vartheta = e_1.\vartheta \cup \ldots \cup e_n.\vartheta \tag{4.1}$$

The dependency on the iterator I_χ manifest itself relationally by the attribute $\chi numb$. The value of $\chi numb$ is the iteration number of I_χ, that is, for a tuple ($\chi numb$, *value*) the value *value* will appear in the $\chi numb$th iteration of I_χ.

When an iterator variable $\$\chi$ is declared it is assigned a $\chi numb$ by renaming the *index* attribute of the corresponding iterator sequence. Which leads us to the inference rule for translating the (optional) `for`-clause of a FLWOR expression:

$$\frac{}{\texttt{for } \$\chi \texttt{ in } e \ldots} \longmapsto \begin{array}{l}\mathbf{put}(\chi,\\ \quad \mathsf{project}(\chi\mathsf{numb} = \mathsf{index},\ \mathsf{index{=}1},\ \mathsf{value};\\ \quad\quad \mathbf{r}(e)))\end{array} \tag{4.2}$$

Where the dependencies piggybacking the `project` operator can be expressed as: $\vartheta = e.\vartheta \cup I_\chi$. For a `for`-clause with multiple variable bindings the rule must be applied once per binding.

From Definition 9 it can be seen that χ is not part of the set of dependencies the iterator I_χ returns its parent. This is in fact the only case where a iterator is removed from a dependency set. Because of this, we must be careful not to incidentally remove an $\chi numb$ attribute from a relation by means of a `project` operator.

When we in this chapter write ϑ enclosed in MQL syntax it is to be interpreted as a comma separated list of all the attributes linked to the dependencies in the set ϑ. As an example, the dependency set $\vartheta = \{I_{\mathtt{x}}, I_{\mathtt{y}}, I_{\mathtt{z}}\}$, is read as xnumb, ynumb, znumb in an MQL environment.

XQuery variable reference expressions, be it iterator, `let` or `declare` variables, are translated to relational algebra quite simply by fetching the tree linked to the variable name in the symbol table:

$$\frac{}{\$\chi} \longmapsto \mathbf{get}(\chi) \tag{4.3}$$

4.3.2 Iterator dependency tainting

The iterator body of an iterator with a iterator sequence with length n will have to be executed n times. This can be done by e.g. evaluating the cartesian product between the body and the sequence, as with the MarkXRemove method. To avoid any denormalised intermediate results, an ideal solution would be to always calculate such products after all other evaluations of the query is done. Consider the following simple example of the query e:

```
for $a in (1, 2) return
    for $b in (3, 4) return
        5 + 6
```

Conceptually, the result of this query can be calculated like this:

$$\mathbf{r}(e)=\mathbf{r}(\texttt{(1, 2)})\times\mathbf{r}(\texttt{(3, 4)})\times\mathbf{r}(\texttt{5 + 6}).$$

But such a simple solution is not adequate if there is a reference to an iterator variable somewhere within the iterator body. This was managed by MarkXRemove by implementing inheritance of iterator dependencies, similar to the concept discussed in Section 4.3.1, and replacing the cartesian product operator with something like a natural join operator (Section 4.1.1).

MarkXRemove has shortcomings when it comes to evaluating expressions where a sequence constructed with at least one iterator dependent expression is a subexpression. Tainting Dependencies mend for this by requiring that all items involved in the evaluation of the result of an iterator dependent expression are iterator dependent. To meet this requirement, dependency tainting is introduced.

Definition 10. *Iterator dependency* ***Tainting*** *is to impose a representation of one expression for each iteration of the iterators another expression is dependent on.*

During sequence construction, expressions explicitly taint all other expressions part of the construction with their dependencies. Consider this subexpression:

$$\vdots$$
$$(e_1\texttt{, }e_2)$$
$$\vdots$$

Where $e_1.\vartheta = \{I_{\chi_1}\}$ and $e_2.\vartheta = \emptyset$. Here e_2 will be tainted by e_1's dependency on χ_1, but as e_2 have no dependencies, e_1 will not be tainted. The tainting process is carried out by calculating the cartesian product of e_2 and the $\chi_1 numb$ column of $\mathbf{r}(\$\chi_1)$ stored in the symbol table.

The tainting of the relational representation of expression e with the depdendencies ϑ is expressed like this:

$$\begin{aligned}&\vartheta = \{I_{\chi_1}, \dots I_{\chi_m}\}\\ &\mathbf{t}(\mathbf{r}(e),\vartheta) = \mathbf{r}(e) \times \prod_{I_{\chi_j} \in (e.\vartheta - e_i.\vartheta)} \mathsf{project}(\chi_j numb;\ \mathbf{get}(\chi_j))\end{aligned} \tag{4.4}$$

4.3.3 Unique iterations

Consider an XQuery expression consisting of nested iterators $I_{\chi_1}, \ldots, I_{\chi_n}$, where I_{χ_j} ($1 < j < n$) occurs within the iterator body of $I_{\chi_{j-1}}$. As per XQuery semantics, the iterator body of a iterator I_{χ_j} is evaluated once for each of the items in the iterator sequence of I_{χ_j}. And because of the nesting, I_{χ_j} will have to be evaluated once per item in the iterator sequence of $I_{\chi_{j-1}}$. The consequence of this is that the number of unique iterations the body of I_{χ_j} is actually evaluated can be expressed like this:

$$\textit{unique iterations evaluated for body of} I_{\chi_j} = \prod_{i=1}^{j} card(I_{\chi_i})$$

Where $card(I_\chi)$ is a function returning the cardinality of the iterator sequence of I_χ.

Of these nested iterators, let a subexpression e be dependent on the subset $\{I_{\chi_k}, I_{\chi_l}\}$. Because of dependency tainting and inheritance, the relational depiction of e will have a representation in all possible iteration combinations of I_{χ_k} and I_{χ_l}. A tuple in e, $(index, \chi_k numb, \chi_l numb, value)$, represents one of these unique iterations. When I_{χ_k} is in its $\chi_k numb$th iteration and I_{χ_l} is in its $\chi_l numb$th iteration, the item in position *index* of the sequence returned from e will be *value*. Let I_{χ_m} also be one of the nested iterators, but one which e is not dependent on. e will evaluate to the same result regardless of which iteration I_{χ_m} is in given the iteration number of I_{χ_k} and I_{χ_l} is constant.

When an subexpression such as e is used in further evaluation, it is important to seperate these iterations from each other. This is done by partitioning the relation on all unique combinations of its iterator dependency attributes. Partitioning can be done either by the group operator or by specifying the attributes to partition on in the partition list of the numberate operator.

Often the evaluation of an expression use the *value* fields of each of its subexpressions. E.g. an addition expression is evaluated by adding the value of its first subexpression with the value of the second. To be able to calculate such expressions, the values of the subexpressions will have to be in the same relation. This can be achieved by evaluating the cartesian product of the subexpressions. Assumed that the subexpressions are independent of iterators or are not dependent on the same iterators this is sufficient. But if they are depentent on one or more iterators in common, the result of the cartesian product will have to be synchronised on the common iterators iterations. This allows evaluation in each unique iteration, and is solved by turning the cartesian product into an equi-join.

Generally, for such an expression e, with the subexpressions e_1 and e_2 this can be written like this:

$$\mathbf{r}(e) = \begin{array}{l} \ldots \\ \mathsf{hhjoin}([(e_1.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_1.\vartheta)]\ldots \\ \quad \mathbf{r}(e_1) \\ \quad \mathbf{r}(e_2)) \end{array}$$

The dependencies $e.\vartheta$ is described by equation 4.1. If $e_1.\vartheta \neq e_2.\vartheta$ each subexpression will be implicitly tainted by the other's unique dependencies. If the hhjoin operator is used without specifying the join attributes, we assume it will behave as a cross operator.

How the effective boolean function $\mathbf{B}(\mathbf{r}(e))$ works will be discussed in section 7.2.1. In this chapter is sufficient to consider it as a grouping operator, grouping on the attributes specified by $e.\vartheta$ (i.e. all known unique iterations). For each group it will produce a field *pred* representing the effective boolean value of e in that unique iteration. If e holds a singleton numeric value in one group *pred* will hold this value, in all other cases it will hold a boolean value. The result relation of the function will in addition to the *pred*-attribute contain all the attributes implied by $e.\vartheta$. The main reason this function is introduced at all is that it ensures that a incoming relation will have *exactly one* tuple per unique iteration.

4.3.4 Literals

The XQuery Full Text specification[34] defines a number of literals as seen in figure 4.2. A `StringLiteral` is a text string enclosed in apostrophes or quotation marks, and the numeric literals are similar to numeric types from other programming languages.

```
[85] Literal        ::= NumericLiteral | StringLiteral
[86] NumericLiteral ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
```

Figure 4.2: Definition of literals in XQuery Full Text

To be able to include such expressions in evaluation of relational algebra, they need a relational representation. As we in this chapter assume *value* is a polymorphic type attribute, with the help of the make operator translation of literals is done in the following way:

$$\frac{e \in \{Literals\}}{e} \longmapsto \mathsf{make(name{:=}["index","value"],\ [1,}\ e\mathsf{])} \tag{4.5}$$

This is a general way to translate literals, but there exists quite a few simplifications. Most importantly when constructing sequences entirely composed of literals, as we will discuss in section 4.10.

4.4 Sequence construction

```
[33] FLWORExpr         ::= (ForClause | LetClause)+ WhereClause?
                                OrderByClause? "return" ExprSingle
[45] IfExpr            ::= "if" "(" Expr ")" "then" ExprSingle
                                "else" ExprSingle
[31] Expr              ::= ExprSingle ("," ExprSingle)*
[89] ParenthesizedExpr ::= "(" Expr? ")"
```

Figure 4.3: Excerpt from W3C XQuery EBNF showing sequence construction

A sequence in XQuery can be built with the comma operator(`,`). But this operator is the XQuery operator with the lowest precedence, therefore, in most cases a sequence construction expression will be enclosed in paratheses. This is to solve parser ambiguities, which can be seen in the excerpt of the W3C XQuery EBNF specification[36] in Figure 4.3. An `ExprSingle` can solely consist of a `ParenthesizedExpr` via a series of productions omitted from the figure. Also note a `ExprSingle` can be a `FLWORExpr`.

As also can be seen from the figure the `return`-clause of a FLWOR expression, as many other expressions, accepts an `ExprSingle`. If a sequence is to be constructed in the `return`-clause, it will have to be parenthesised.

With the concept of tainting and iterator dependencies explained, we are now ready to introduce the translation of an XQuery sequence construction expression:

$$\frac{}{e_1\texttt{,}\ \ldots\texttt{,}\ e_n} \longmapsto \begin{array}{l} \mathsf{numberate}(\mathsf{index},\ [\mathsf{sprIdx},\ \mathsf{index}],\ [\vartheta]; \\ \quad \mathsf{union}(\\ \quad\quad \mathsf{project}(\mathsf{sprIdx}{=}1,\ \mathsf{index},\ \mathsf{value}; \\ \quad\quad\quad \mathbf{t}(\mathbf{r}(e_1),\ \vartheta)); \\ \quad\quad\quad \vdots \\ \quad\quad \mathsf{project}(\mathsf{sprIdx}{=}n,\ \mathsf{index},\ \mathsf{value}; \\ \quad\quad\quad \mathbf{t}(\mathbf{r}(e_n),\ \vartheta)))) \end{array} \tag{4.6}$$

Where $\vartheta = e_1.\vartheta \cup \ldots \cup e_n.\vartheta$. Notice how all subexpressions are tainted with the iterator dependencies of all the others.

The basis of a sequence construction is the union operator – as with MarkXRemove. But because we in Tainting Dependencies have introduced the explicit ordering of items with the *index* attribute, additional operators have been added. Each item in the sequences returned from the subexpressions is equipped with a temporary field *sprIdx* (superindex) holding the relative position of each subexpression. Based on the positioning defined by *sprIdx* and *index* the numberate operator can renumberate the resulting sequence. The numbering must partition on the fields corresponding to the dependencies in ϑ, to separate the different sequences constructed for all known unique iterations.

$$\begin{array}{l} \texttt{for \$a in (10,20) return} \\ \quad \underbrace{\texttt{(\$a, "no")}}_{e_1} \end{array}$$

Figure 4.4: Simple XQuery query

Example 1: Consider the simple XQuery query of Figure 4.4. Here **r**(`$a`) will taint **r**(`"no"`) with its dependency on the iterator I_{a}, the result of which is shown in Figure 4.5(b). Further, we can see that for each iteration of I_{a} the `return`-clause will return a sequence of two items. Having in mind that *anumb* (*anb* in figure) holds the iteration number of I_{a}, this can be seen in figure 4.5(c).

The sequence construction rule also holds even if the subexpressions of the soon-to-be sequence are within the body of an iterator the sequence is not dependent on. Expanding the query of Figure 4.4 we get the query of Figure 4.6.

In this query, notice the innermost `return`-clause expression, e_1, is identical to e_1 in the previous query. Here, the result of the sequence construction will still be the relation shown in figure 4.5(c), because $e_1.\vartheta = \{I_{\mathsf{a}}\}$ – also as before. e_1 is not aware of the iterator I_{b} – and does not need to be either, as the result of e_1 is independent of which iteration number I_b is in.

anb	*idx*	*val*
1	1	10
2	1	20

(a) **r**(`$a`)

anb	*idx*	*val*
1	1	`"no"`
2	1	`"no"`

(b) **t**(**r**(`"no"`),**r**(`$a`).ϑ)

anb	*idx*	*val*
1	1	10
2	1	20
1	2	`"no"`
2	2	`"no"`

(c) **r**(`($a, "no")`)

Figure 4.5: Applying translation Rule 4.6 on (a) and (b) yields (c). Attribute names are shortened

$$\begin{array}{l}\texttt{for \$a in (10,20) return}\\ \quad\texttt{for \$b in (50,75) return}\\ \quad\underbrace{\texttt{(\$a, "no")}}_{e_1}\end{array}$$

Figure 4.6: Query of Figure 4.4 expanded

4.5 FLWOR Expressions

The `let`-clause does not explicitly cause any dependencies, only variable binding, and can therefore be translated into storing the algebraic version of the expression to be bound in the symbol table (see Section 4.3):

$$\frac{}{\texttt{let \$}\chi\texttt{ := }e\texttt{ ...}} \longmapsto \mathbf{put}(\chi,\ \mathbf{r}(e)) \tag{4.7}$$

The iterator dependences $e.\vartheta$ is stored along with $\mathbf{r}(e)$ and will piggyback this algebra tree if it later fetched from the symbol table. If there is more than one variable binding in the `let`-clause the rule must be applied once per binding as if one binding were one `let`-clause.

As seen by the excerpt of the W3C XQuery EBNF specification in figure 4.3, a FLWOR expression may be structured in many different ways. For simplicity and readablilty the translation of such expressions will be split up in more managable pieces.

A FLWOR expression may have multiple `for`-clauses, and a `for`-clause may have multiple iterator variable bindings. This means that one FLWOR may consist of many iterators, the semantics of which is described in Section 3.3. We assume all possible iterator dependencies generated from the FLWOR, that is, all iterator variables bound, is stored in a ordered set β. The dependencies are ordered in the order of which the corresponding iterator variables are bound, i.e. top down, left to right while parsing the query. When enclosed in MQL syntax β is, as ϑ, to be read as a comma separated list of the attributes corresponding to the dependencies.

$$\begin{array}{l}[\texttt{for/let ...}]+\\ \quad[\texttt{where } e_2]?\\ \quad[\texttt{order by } e_3]?\\ \texttt{return } e_4\end{array}$$

Figure 4.7: Illustration of a FLWOR expression

The translation of a single FLWOR like the one illustrated in Figure 4.7 will be executed as shown in Figure 4.8. Firstly, all `for` and `let` variables will be bound as described by the rules 4.2 and 4.7, respectively. Further, the `return`-clause will be evaluated. If there is a `where`-clause present, it is evaluated next, based on the `where`-expression and the result of the `return`-clause, referred to as $\mathbf{r}(e_C)$. The order of the items returned from the FLWOR is conditioned by the presence of an `order by`-clause in the expression. If there is a `order by`-clause, it will order the intermediate result from the `return` or `while`-clause, and finalise the FLWOR. If there is no `order by`-clause, the final evaluation of the FLWOR will be to order the intermediate result according to the iterators.

$$\begin{tikzcd}
 & & & \textbf{iterator order} \\
{[\textbf{var binding}]+} \arrow[r, dotted] & \texttt{return } e_4 \arrow[r, "\mathbf{r}(e_C)"'] & {[\texttt{where } e_2]?} \arrow[ur, "\mathbf{r}(e_C)"] \arrow[dr, "\mathbf{r}(e_C)"'] & \\
 & & & {[\texttt{order by } e_3]?}
\end{tikzcd}$$

Figure 4.8: Illustration of step-by-step translation of FLWOR.

Any possible iteration, ordering og filtering will be handled by other clauses than the `return`-clause. The `return`-clause will just ship the evaluated version of the `return`-expression forward:

$$\frac{}{\texttt{return } e_4} \longmapsto \mathbf{r}(e_4) \tag{4.8}$$

4.5.1 Iterator ordering

Acording to XQuery semantics, the `return`-clause is evaluated once for iteration for each of the iterators of the FLWOR expression. The results of these evaluations are concatenated to form the result of the FLWOR expression. As TD evaluate iterations in parallel, if the `return`-expression is not dependent on a iterator, its relational form will not have a representation for each of the iterators iterations and will have to be tainted.

Because each FLWOR iterator creates a new sequence, renumbering is needed. No expression in a sibling or parent scope of the FLWOR may be dependent on the dependencies in β. Thus, β is not part of the dependencies returned from the FLWOR, and the attributes corresponding to β must be removed.

If the FLWOR contains no `order by`-clause, the ordering of the resulting sequence is determined by the iterators of the expression. Remember, the set of iterators β is ordered according to the order the variables were bound.

$$\frac{}{\textbf{iterator order}} \longmapsto \begin{array}{c}\mathsf{numberate}(\mathsf{index},\ [\beta,\ \mathsf{index}],\ [\vartheta];\\ \mathbf{t}(\mathbf{r}(e_C),\ \beta)\end{array} \tag{4.9}$$

Where $\vartheta = e_C.\vartheta - \beta$, and e_C is the result returned from the `where`-clause, if present, otherwise it is the result of the `return`-clause.

From equation 4.4 it is clear that an expression already dependent on any iterator in β will not be tainted by these iterators.

The numberate operator will have to partition on the remaining dependencies in ϑ to seperate the sequences returned from the iterator for all iterators the result is dependent on, as described in section 4.3.3.

Example 2: Consider the query of Figure 4.9. Notice the expression in the `return`-clause (e_1) is the same as e_1 in Example 1.

$$\begin{array}{l}\texttt{for \$a in (10,20),}\\ \texttt{\ \ \ \ \$b in (50,75)}\\ \texttt{return }\underbrace{\texttt{(\$a, "no")}}_{e_1}\end{array}$$

Figure 4.9: Example query with two iterators.

Expression e_1 is not depentant on $I_{\mathtt{b}}$, and by Rule 4.9 will be tainted by this iterator. The result of the tainting is shown in Figure 4.10(a). The expression will however not be tainted by $I_{\mathtt{a}}$, as it is already dependent on this iterator (ref. equation 4.4).

bnb	*anb*	*idx*	*val*
1	1	1	10
1	2	1	20
1	1	2	"no"
1	2	2	"no"
2	1	1	10
2	2	1	20
2	1	2	"no"
2	2	2	"no"

(a) t(r(e_1), {b})

idx	*val*
1	10
2	"no"
3	10
4	"no"
5	20
6	"no"
7	20
8	"no"

(b) r($I_{\mathtt{a}}$)

Figure 4.10: Evaluating the query in Figure 4.9 yields the intermediate result (a) and the final result (b). Attribute names are shortened.

With the expression tainted the dependencies of all the iterators of the FLWOR, renumbering is the last operation required to translate the query. The numberate operator will sort the tuples first on *anumb*, then on *bnumb* and finally *index* before numbering. As the FLWOR itself does not have any dependencies the numeration will not be partitioned over any attributes. The result of the query is shown in figure 4.10(b).

4.5.2 Where-clause

The W3C describes the FLWOR expression as generating a tuple stream which contains one tuple for each combination of values bound in the expression[36]. In this view, the optional `where`-clause serves a filter for these tuples. The expression in the `where`-clause is evaluated once for each tuple. If the boolean value of this expression is *true*, the tuple is retained, if the boolean value is $false$ the tuple is discarded.

The `where`-clause can be evaluated by only selecting the iteration combinations where the `where`-expression is *true*. The result of this will be joined with the result of the `return` clause. If the result from the `return`-clause and the `where`-expression has no common iterator dependencies, the equi-join will have to be replaced by a cartesian product, as discussed in 4.3.3.

$$\frac{}{\texttt{where } e_2} \longmapsto \begin{array}{l}\mathsf{hhjoin}([(e_C.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_C.\vartheta)],\ [\mathsf{l.value},\ \vartheta];\\ \quad \mathbf{r}(e_C);\\ \quad \mathsf{select}(\mathsf{xqBoolean}(\mathsf{pred});\\ \quad\quad \mathbf{B}(\mathbf{r}(e_2))))\end{array} \tag{4.10}$$

Where $\vartheta = e_C.\vartheta \cup e_2.\vartheta$, and $\mathbf{r}(e_C)$ is the result of the translation done in the `return`-clause.

By employing the select operator before the joining of the two relations the number of tuples needed involved in the join will be minimised. Further, as the hhjoin operator allows for declaring which attributes to be projected, there is no need for a seperate project operator.

Example 3: Consider the XQuery query of Figure 4.11. The evaluation of e_1 is very similar to the evaluation of e_1 of Example 1, and the result is shown in Figure 4.12(a). The `where`-clause contains a comparison expression, which will be discussed later. Figure 4.12(b) shows the result relation of the `where`-expression.

```
for $a in (10,20)
where $a > 15
return ($a, 14)
```
$\underbrace{\texttt{(\$a, 14)}}_{e_1}$

Figure 4.11: A FLWOR expression with a `where`-clause

anb	*idx*	*val*
1	1	10
1	2	14
2	1	20
2	2	14

(a) $\mathbf{r}(e_1)$

anb	*idx*	*val*
1	1	*false*
2	1	*true*

(b) r($a > 15)

anb	*idx*	*val*
2	1	20
2	2	14

(c)

Figure 4.12: Applying Rule 4.10 on (a) and (b) yields (c). Attribute names are shortened.

During the evaluation of the FLWOR, the tuples in the `where`-expression relation where *value* (*val* in figure) is not *true* will be removed. The result is a relation with a single tuple where *anumb* holds the value 2. This result is then joined with the e_1 relation on *anumb*, as shown in figure 4.12(c). Rule 4.9 will have to be applied to complete the evaluation of the query.

4.5.3 Order by-clause ordering

As can be seen in Figure 4.5.3, an `Order by`-clause may contain several specifications on how to order and what to order by.

At this time, Taiting Dependencies only support a simple form of the clause. Only one `OrderSpec` is allowed, where the expression to be sorted on will be referred to as e_4 – confor-

```
[38] OrderByClause ::= (("order" "by") | ("stable" "order" "by"))
                            OrderSpecList
[39] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[40] OrderSpec     ::= ExprSingle OrderModifier
[41] OrderModifier ::= ("ascending" | "descending")?
                            ("empty" ("greatest" | "least"))?
                            ("collation" URILiteral)?
```

Figure 4.13: W3C EBNF order by clause specification[36].

mant to figure 4.8. Nor is any `OrderModifier`s allowed. A complete translation of the `order by`-clause is discussed in Section 7.1.4.

As with the `where`-clause W3C specify the `order by`-clause as operating on a tuple stream created by the variable bindings in the FLWOR[36]. The clause causes the stream to be reordered into a new, value-based order. This means that the iterator dependency attributes ($\chi numb$) will no longer decide the composition of the sequence returned from the FLWOR.

Firstly, to ensure that the result will have a representation for all the iterations (still valid after possible filtering by `where`-clause) of all the expressions iterators, e_C is tainted with β. The `order by`-expression is evaluated and joined with the result of the tainting on their common iterator dependencies. Lastly, this relation is renumbered according to the order of the values stemming from the `order by`-expression:

$$\frac{}{\texttt{order by } e_3} \longmapsto \begin{array}{l}\mathsf{project}(\mathsf{value} = \mathsf{l.value},\ \vartheta;\\ \quad \mathsf{numberate}(\mathsf{index},\ [\mathsf{r.value}],\ [\vartheta];\\ \quad\quad \mathsf{hhjoin}([(e_C.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_C.\vartheta)],[\mathsf{l.value},\ \mathsf{r.value},\ \vartheta];\\ \quad\quad\quad \mathbf{t}(\mathbf{r}(e_C),\ \beta);\\ \quad\quad\quad \mathbf{r}(e_3))))\end{array} \tag{4.11}$$

Where $\vartheta = (e_C.\vartheta \cup e_3.\vartheta) - \beta$, and $\mathbf{r}(e_C)$ is the result of the translation done in the `where`-clause if present, else the `return`-clause. Fields such as *index* and if applicable *documentId*, *pos* and *scope* will follow *l.value*. *r.value* is only the *value* column from the right relation. To accommodate for the `stable` keyword, the numberate operator will have to sort on *l.index* aswell, after sorting on *r.value*.

As desired, all iterator dependency attributes ($\chi numb, \chi \in \beta$) will be projected away by the hhjoin operator. An additional project operator is used to ensure that the *value* returned from the expression is the *value* attribute of the `return`-expression.

Example 4: Figure 4.14 shows a query consisting of a FLWOR expression containing a `order by` clause.

```
for $a in (4, 6, 8)
order by $a mod 3
return $a
```

Figure 4.14: Example query with order by clause

The evaluated `order by`-expression is shown in Figure 4.15(a). How modulus expressions are translated will be discussed later. After their evaluation, the relational representation of `order by`-expression relation and the `return`-clause is joined on their common iterator dependencies, $I_{\texttt{a}}$ (*anumb*). The result of the join is shown in Figure 4.15(b). Lastly, this relation is renumbered and sorted on the *r.value* (*r.val* in figure) attribute stemming from the `order by`-expression, as shown in Figure 4.15(c). An additional projection to rename *l.value* to *value* is needed to finalise the query.

idx	*anb*	*val*
1	1	1
1	2	0
1	3	2

(a) `r($a mod 3)`

l.idx	*anb*	*l.val*	*r.val*
1	1	4	1
1	2	6	0
1	3	8	2

(b)

idx	*l.val*
1	6
2	4
3	8

(c)

Figure 4.15: Intermediate results evaluating the query of figure 4.14. (a) shows the evaluated `order by` expression. (b) shows the `order by`-expression joined with the `return`-expression. (c) shows the relation in (b) renumbered. Attribute names are shortened.

4.6 Simple binary operator expressions

In this section we will present methods for translating simple XQuery binary operator expressions, namely arithmetic, logic and comparison operator expressions. XQuery binary operators not covered here will be discussed in Section 7.1.3 on page 111.

4.6.1 Arithmetic Expressions

W3C defines the XQuery arithmetic expressions as shown in figure 4.16[36]. Notice how the specified grammar handles operator precedence. `UnaryExpr` is a decendant production of `UnionExpr`.

```
[50] AdditiveExpr       ::= MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
[51] MultiplicativeExpr ::= UnionExpr ( ("*" | "div" | "idiv" | "mod") UnionExpr )*
[58] UnaryExpr          ::= ("-" | "+")* ValueExpr
```

Figure 4.16: The arithmetic expressions of XQuery

The translation of such expressions will have to be separated in binary and unary operators. For the binary operators the two values to be operated on will have to be in the same relation. To ensure the values to be operated on are from the same unique iteration (if there is any iterations at all), the relations corresponding to the two expressions will have to be joined on their common iterator dependencies, as described in section 4.3.3. Both the unary and the binary XQuery arithmetic operators accept only singleton sequences. This is discussed in Section 7.2.

$$\frac{}{e_1 \texttt{ OP } e_2} \longmapsto \begin{array}{l}\textsf{project(index=1,value=FUNC(l.value,r.value),}\vartheta\textsf{;}\\ \quad\textsf{hhjoin([}(e_1.\vartheta \cap e_2.\vartheta)\textsf{], [}(e_2.\vartheta \cap e_1.\vartheta)\textsf{],[r.value, l.value, }\vartheta\textsf{];}\\ \quad\quad \mathbf{r}(e_1)\\ \quad\quad \mathbf{r}(e_2)\textsf{)))}\end{array} \tag{4.12}$$

Where $\vartheta = e_1.\vartheta \cup e_2.\vartheta$, `OP` will map to a MQL function replacing FUNC as shown in Table 4.2. The projecting functionality of the hhjoin operator will in this case remove the *index* attributes and any possible *documentId*, *scope* and *pos* attributes.

OP	FUNC
+	sum
-	subtract
*	prod
div	div
idiv	idiv
mod	mod

s

Table 4.2: Mapping XQuery arithmetic operators to MQL functions.

Considering the unary operators, the + operator will never have any effect, and can therefore be dropped. The unary - operator will change the sign of the value it is assigned to. This is equal to multiplying the value with -1:

$$\frac{}{\texttt{-}e_1} \longmapsto \begin{array}{l}\textsf{project(value = prod(-1, value);}\\ \quad \mathbf{r}(e_1)\textsf{)}\end{array} \tag{4.13}$$

Example 5: Consider the XQuery query of Figure 4.17. Here e_1 is an arithmetic expression.

```
for $a in (1,2) return
  for $b in (3,4) return
    $a + $b
```

$\underbrace{\texttt{\$a + \$b}}_{e_1}$

Figure 4.17: Example query containing a arithmetic expression.

Both `$a` and `$b` is translated to simple two-tuple relations by rules 4.2 and 4.3. As the two expressions do not have any iterator dependencies the hhjoin operator of Rule 4.12 will be treated as a cartesian product (ref. 4.3.3). The cross product of the two relations are shown in figure 4.18(a). Lastly, the project operator is employed to calculate the sum for each iteration, the result of which is shown in Figure 4.18(b).

4.6.2 Logical Expressions

An XQuery logical expression is either an `and` expression or an `or` expression. If a logical expression does not raise an error(see Section 7.2.1), its value is always one of the boolean values *true* or *false*.

anb	*bnb*	*l.val*	*r.val*
1	1	1	3
2	1	2	3
1	2	1	4
2	2	2	4

(a)

idx	*anb*	*bnb*	*val*
1	1	1	4
1	2	1	5
1	1	2	5
1	2	2	6

(b) $\mathbf{r}(e_1)$

Figure 4.18: Results of evaluating expression e_1 of Figure 4.17. (a) shows the result of the cross product. (b) is the fully evaluated e_1. Attribute names are shortened.

A logical expression is translated in a matter very similar to the arithmetic expressions. The XQuery logical operators does however operate on the effective boolean value (see Section 2.1.1) rather than the direct value. As the operators require boolean values, and the effective boolean function $\mathbf{B}()$ may return a number, the *pred* fields will have to be run through the xqBoolean() function.

$$\frac{}{e_1 \ \texttt{OP} \ e_2} \longmapsto \begin{array}{l} \mathsf{project(index{=}1,value{=}FUNC(xqBoolean(l.pred),xqBoolean(r.pred)),}\vartheta; \\ \quad \mathsf{hhjoin}([(e_1.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_1.\vartheta)], \\ \qquad\quad [\mathsf{r.pred,\ l.pred,\ }\vartheta]; \\ \quad\ \mathbf{B}(\mathbf{r}(e_1)) \\ \quad\ \mathbf{B}(\mathbf{r}(e_2)))) \end{array} \tag{4.14}$$

Where $\vartheta = e_1.\vartheta \cup e_2.\vartheta$, `OP` will map to a MQL function replacing FUNC as shown in Table 4.3.

OP	FUNC
or	or
and	and

Table 4.3: Mapping XQuery boolean operators to MQL functions

4.6.3 Comparative Expressions

Comparison expressions allow two values to be compared. XQuery provides three kinds of comparison expressions, called value comparisons, general comparisons, and node comparisons. The comparison operators as specified by W3C are shown in Figure 4.19. The Tainting Dependency method does at this time not support node comparisons, but possible solutions are discussed in Section 7.1.3.

```
[61] ValueComp   ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[60] GeneralComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[62] NodeComp    ::= "is" | "<<" | ">>"
```

Figure 4.19: The comparison operators of XQuery [36]

Value comparisons are used for comparing two single values. With the same premises and restrictions, the rule for translating arithmetic expressions (Rule 4.12) can be used to translate such comparison expressions. The mapping between the XQuery value comparison operators and MQL functions can be seen in Table 4.4.

OP	FUNC
eq	eq
ne	neq
lt	lt
le	leq
gt	gt
ge	geq

Table 4.4: Mapping XQuery value comparison operators to MQL functions

General comparisons are existentially quantified comparisons that may be applied to sequences of any length. If employing a general comparison operator on any pair of items consisting of one from each of the sequences yields *true*, the comparison expression yields *true*. As an example, all the comparison expressions of figure 4.20 evaluates to *true*.

```
(1, 2) = (2, 3)
(1, 2) != (2, 3)
(1, 200) < (10, 20)
(1, 200) > (10, 20)
```

Figure 4.20: Example general comparisons: all expressions evaluate to true

The big difference between general and value comparisons is that the first must accomodate for sequences. This is solved by grouping expressions' iterator dependencies, meaning that each group will contain the sequence of one unique iteration. By defining *true* as having a larger value than $false$, the max() aggregator will identify the groups with *at least* one *true* value.

As with the arithmetic expressions, the relational representation of the two operands is joined on their common dependencies to ensure that both values are from the same iteration.

$$\overline{e_1\ \texttt{OP}\ e_2} \longmapsto \begin{array}{l} \mathsf{project(index = 1, value{=}max}, \vartheta; \\ \quad \mathsf{group((}\vartheta\mathsf{), max(value)}; \\ \quad\quad \mathsf{project(value{=}FUNC(l.value,r.value)},\vartheta; \\ \quad\quad\quad \mathsf{hhjoin}([(e_1.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_1.\vartheta)],[\mathsf{l.value,\ r.value},\ \vartheta]; \\ \quad\quad\quad\quad \mathbf{r}(e_1) \\ \quad\quad\quad\quad \mathbf{r}(e_2))))) \end{array} \tag{4.15}$$

Where $\vartheta = e_1.\vartheta \cup e_2.\vartheta$. `OP` wil map to a MQL function replacing FUNC as shown in Table 4.5.

The result of a general comparison is always a singleton sequence, thus it is safe to project in an *index* attribute with the value 1. The *index* and possible *documentId*, *scope* and *pos* attributes are left out of the project list of the hhjoin operator.

Example 6: Figure 4.21 shows a simple XQuery query with a general comparison expression, e_1.

OP	FUNC
=	eq
!=	neq
<	lt
<=	leq
>	gt
>=	geq

Table 4.5: Mapping XQuery general comparison operators to MQL functions.

$$\begin{array}{l}\texttt{for \$a in (10, 20) return}\\ \quad\underbrace{\texttt{\$a > (5, 15)}}_{e_1}\end{array}$$

Figure 4.21: Example query with a general comparison expression

Because the operands of the > operator have no common iterator dependencies, the cartesian product of the two relations is calculated, as seen in Figure 4.22(a). After the inner project operator is employed, the result will be as in Figure 4.22(b). The double line illustrates the grouping on *anumb* (*anb* in the figure). The maximum value of *value* for each group is calculated and the attributes are renamed, which gives the relation of Figure 4.22(c).

anb	*l.val*	*r.val*
1	10	5
1	10	15
2	20	5
2	20	15

(a)

anb	*val*
1	*true*
1	*false*
2	*true*
2	*true*

(b)

idx	*anb*	*val*
1	1	*true*
1	2	*true*

(c) $\mathbf{r}(e_1)$

Figure 4.22: Results of evaluating expression e_1 in Figure 4.21. (a) The relations of the operands joined. (b) Each combination of the sequences evaluated. Double line illustrates grouping. (c) The final result. Attribute names are shortened.

4.7 Conditional Expressions

XQuery supports a conditional expression based on the keywords `if`, `then`, and `else`. The expression is specified by W3C as seen in Figure 4.23.

The expression following the `if` keyword is called the test expression, and the expressions following the `then` and `else` keywords are called the `then`-expression and `else`-expression, respectively. If the effective boolean value of the test expression evaluates to *true*, the `then`-expression is returned, if it evaluates to *false* the `else`-expression is returned.

Conditional expressions are translated by adding an attribute *alt* with the value 1 to the `then`-expression relation and the relational representation of the `else`-expression with the same attribute but with the value 2. These two relations are then spliced together with a

```
[45] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle
                  "else" ExprSingle
```

Figure 4.23: W3C EBNF conditional expression specification[36].

union operator. If the relations have disjoint dependencies, they will have to taint each other first.

The result of the union operation is then joined with the relational representation of the test expression on their common dependencies. Lastly, a select operator is employed on this relation to select the tuples where *alt* is 1 if the *value* field from the test expression evaluates to *true*, or *alt* is 2 if it does not:

$$\begin{array}{l}
\overline{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3} \\
\longmapsto \\
\mathsf{project(value = l.value,}\ \vartheta; \\
\quad \mathsf{select(ifthenelse(xqBoolean(pred), eq(alt,1), eq(alt,2));} \\
\quad\quad \mathsf{hhjoin([}((e_2.\vartheta \cup e_3.\vartheta) \cap e_1.\vartheta)\mathsf{],\ [}(e_1.\vartheta \cap (e_2.\vartheta \cup e_3.\vartheta))\mathsf{],[l.value,\ pred,}\ \vartheta\mathsf{,\ alt];} \\
\quad\quad\quad \mathsf{union(} \\
\quad\quad\quad\quad \mathsf{project(alt{=}1,\ value,}\ (e_2.\vartheta \cup e_3.\vartheta), \\
\quad\quad\quad\quad\quad \mathbf{t}(\mathbf{r}(e_2),\ e_3.\vartheta)); \\
\quad\quad\quad\quad \mathsf{project(alt{=}2,\ value,}\ (e_3.\vartheta \cup e_2.\vartheta), \\
\quad\quad\quad\quad\quad \mathbf{t}(\mathbf{r}(e_3),\ e_2.\vartheta))); \\
\quad\quad\quad \mathbf{B}(\mathbf{r}(e_1)))))
\end{array} \tag{4.16}$$

Where $\vartheta = e_1.\vartheta \cup e_2.\vartheta \cup e_3.\vartheta$. *index* and possible *documentId*, *pos* and *scope* attributes will follow *value* as described in Section 4.3.

$$\begin{array}{l}
\texttt{for \$a in (10, 20) return} \\
\quad \texttt{for \$b in (5, 15) return} \\
e_1 \left\{ \begin{array}{l}
\texttt{if(\$b > \$a) then} \\
\quad \texttt{\$a} \\
\texttt{else} \\
\quad \texttt{\$b}
\end{array} \right.
\end{array}$$

Figure 4.24: Example query containing a conditional expression

Example 7: The query of Figure 4.24 contains a conditional expression e_1. First the then-expression and the else-expression are tainted with each others iterator dependencies. The result of the tainting is shown in Figure 4.25(a). Then the two relations are augmented with the *alt* attribute and spliced together with an union operator to make the relation depicted in Figure 4.25(b).

The result of the union operation is then joined with the test expression (figure 4.25(c)) on their common dependencies (*anumb* and *bnumb*) to form the relation in figure 4.26(a). From

bnb	*anb*	*val*
1	1	10
1	2	20
2	1	10
2	2	20

(a)

alt	*bnb*	*anb*	*val*
1	1	1	10
1	1	2	20
1	2	1	10
1	2	2	20
2	1	1	5
2	2	1	15
2	1	2	5
2	2	2	15

(b)

anb	*bnb*	*pred*
1	1	*false*
2	1	*false*
1	2	*true*
2	2	*false*

(c)

Figure 4.25: Intermediate results of evaluating e_1 *in Figure 4.24. (a) The* `then`*-expression tainted with* I_b *(b) The* `then`*-expression and the* `else`*-expression augmented with an alt attribute and spliced together. (c) The test expression. Attribute names are shortened. index attribute is left out.*

this relation, only the tuples where *alt* has the value 1 and *r.val* is *true* or *alt* has the value 2 and *r.val* is *false* are selected. The result of the selection after the final renaming is shown in Figure 4.26(b).

alt	*bnb*	*anb*	*val*	*pred*
1	1	1	10	*false*
2	1	1	5	*false*
1	2	1	10	*true*
2	2	1	15	*true*
1	1	2	20	*false*
2	1	2	5	*false*
1	2	2	20	*false*
2	2	2	15	*false*

(a)

bnb	*anb*	*val*
1	1	5
2	1	10
1	2	5
2	2	15

(b)

Figure 4.26: Evaluating e_1 *in figure 4.24. (a) The test expression joined with the union of the* `then`*- and* `else`*-expression. (b)* e_1 *fully evaluated. Attribute names are shortened. index attribute is omitted.*

4.8 Quantified Expressions

Figure 4.27 shows the EBNF specification of the XQuery quantified expression. These expressions support existential and universal quantification, and will always result in a single *true* or *false* value.

```
[42] QuantifiedExpr ::= ("some" | "every") "\$" VarName "in" ExprSingle
                        ("," "\$" VarName "in" ExprSingle)* "satisfies" ExprSingle
```

Figure 4.27: W3C specification of quantified expressions [36]

If the quantifier is `some`, the expression only returns *true* if at least one evaluation of the `satisfies`-expression yields *true*. For the `every` quantifier, the expression will only return *true* if every evaluation of the `satisfies`-expression yields *true*.

It can be suitable to treat such expressions as iterators with the `satisfies`-expression as the iterator body. The result of the iterator is then checked if every or some of its items have the effective boolean value *true*. This can be done with a group operator similarly to the general comparison operator expressions. If we assume *true* has a bigger value than $false$, a max aggregator will reveal if there exists a tuple with the value *true* and a min operator will reveal if there exist a tuple with the value $false$. All variables bound in quantified expressions will be treated by the iterator binding translation of rule 4.2. β will refer to the set of all variables bound in one such expression

$$\frac{}{\texttt{QUANT \$...satisfies } e_1} \longmapsto \begin{array}{l}\mathsf{project(index = 1, value, }\vartheta\mathsf{;}\\ \quad\mathsf{group((}\vartheta\mathsf{), AGG(value);}\\ \quad\quad\mathsf{project(xqBoolean(value), }\vartheta\mathsf{;}\\ \quad\quad\quad\mathbf{B}(\mathbf{r}(e_1)))))\end{array} \tag{4.17}$$

Where $\vartheta = e_1.\vartheta - \beta$, and a quantifier specification `QUANT` maps to an aggregator function as seen in Table 4.6. The xqBoolean() funcion will have to be run on the *value* fields of the `satisfies` expression, as there is no requirement that it is a boolean expression.

QUANT	AGG
some	max
every	min

Table 4.6: Mapping XQuery quantifiers to MQL aggregators

Example 8: Consider the XQuery query of Figure 4.28. In this query the iterator body of the FLWOR is a quantified expression.

```
for $a in ("a","b") return
  every $e in ($a, "b") satisfies $e eq "b"
```

Figure 4.28: Example query with quantified expression

The sequence sequentially bound to the quantifier variable is treated as explained in section 4.4: where `$a` taints `"b"` and the relations are spliced together. As quantifier variables are to be handled as iterator variables, the *index* attribute of this relation will be renamed *enumb*, as illustrated in Figure 4.29(a). This relation (or rather the algebra evaluating to it) is stored in the symbol table, and is fetched during the evaluation of the `satisfies`-expression. The result of this comparison expression is shown in figure 4.29(b). Here, the double line illustrates grouping.

Except for the quantifier variable, this expression is dependent on I_{a}, which the result of the `satisfies`-expression is grouped on. Each group will be run through the `min` aggregator, because the quantifier is `every`. As $false$ has a lower value than *true*, the aggregator will reveal if the group contains a $false$. The result of the grouping is shown in Figure 4.29(c). This result will have to be renumbered to finalise the evaluation of the query.

idx	*anb*	*enb*	*val*
1	1	1	"a"
1	1	2	"b"
1	2	1	"b"
1	2	2	"b"

(a)

idx	*anb*	*enb*	*val*
1	1	1	*false*
1	1	2	*true*
1	2	1	*true*
1	2	2	*true*

(b)

idx	*anb*	*val*
1	1	*false*
1	2	*true*

(c)

Figure 4.29: Evaluating the quantified expression of figure 4.28. (a) The quantifier variable fetched from the symbol table. (b) The result of the `satisfies`-expression. The double line illustrates the groups. (c) The result of the quantified expression. Attribute names are shortened.

4.9 Path expressions and predicates

XQuery implement XPath 2.0 path expressions and predicates as described in section 2.1.2 and 2.1.3. In this section we will present a method for translating some of these expressions into MQL relational algebra.

For the translations in this section to be correct we assume the tuples returned from a scope lookup in the hypothetical valocc-index (see Section 3.4 on page 31) will have information of the scope it self in the *scope* attribute, and the contents of the scope in an *value* attribute. E.g. a lookup of $c (the $-sign indicates it is a scope) in this index may return a tuple with a[1].b[2].c[1] as its *scope* value.

The tuples returned from a lookup in the value occurence index will have to be ordered according to document order. This order will have to hold even after the result is run through a scope

The concept of the context item is important in this section, and is by the W3C defined as follows[36]:

> [Definition: The **context item** is the item currently being processed. An item is either an atomic value or a node.][...] The context item is returned by an expression consisting of a single dot (`.`). When an expression e_1`/`e_2 or e_1`[`e_2`]` is evaluated, each item in the sequence obtained by evaluating e_1 becomes the context item in the inner focus for an evaluation of e_2.

4.9.1 Path expressions

A path expression consists of a series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//". Each step is either a axis step or a filter expression, and a axis step consist of a axis and a node test. This can be seen from the excerpt of the W3C XQuery specification in Figure 4.30. A node test can be either a kind test or a name test, we will focus on the latter.

The semantics of such expressions reading from left to right, is that the result of one step expression is used as input for the next. Within one step expression, the result from the preceding step will first be used as input for the axis expression. The result of this will be filtered by a name or kind test, before this again is filtered by possible predicates.

```
[68] PathExpr         ::= ("/" RelativePathExpr?)
                        | ("//" RelativePathExpr)
                        | RelativePathExpr
[69] RelativePathExpr::= StepExpr (("/" | "//") StepExpr)*
[70] StepExpr         ::= FilterExpr | AxisStep
[71] AxisStep         ::= (ReverseStep | ForwardStep) PredicateList
[72] ForwardStep      ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[75] ReverseStep      ::= (ReverseAxis NodeTest) | AbbrevReverseStep
```

Figure 4.30: Path expressions as specified by W3C[36]

Step expressions can be abbreviated. If the axis name is omitted from an axis step, the default axis is `child`. `decendant-or-self` can be replaced by using "//" istead of "/" between the steps. `@` is an abbreviation of `attribute::`. We will only present translation of unabbreviated syntax.

To accomodate for the context item, the result of each step of a path expression will be stored on the symbol table as **dot** – each time replacing the the last entry. The context item, and references to it will be treated as iterator dependencies. The attribute corresponding to a dependency on **dot** is *dotNumb*. No expression not within the path expression can be dependent on **dot**, which is why **dot** will not be part of the dependencies ϑ returned. A general path expression is translated in the following manner:

$$\overline{[/]e_1/\ldots/e_n} \longmapsto \begin{array}{l} \mathbf{put(dot,\ t(r(}e_1\mathbf{),\ \{dot\}))} \\ \quad\vdots \\ \mathbf{put(dot,\ t(r(}e_n\mathbf{),\ \{dot\}))} \\ \mathsf{numberate(index,\ [dotNumb,\ index],\ [}\vartheta\mathsf{];} \\ \quad \mathbf{get(dot)}) \end{array} \tag{4.18}$$

Where $\vartheta = (e_n.\vartheta - \mathbf{dot})$. Axis step expressions are all dependent on the context item and will take no effect of the tainting. Tainting is used in this context to ensure filter expressions to be evaluated correctly. The ϑ piggybacking the tree in the symbol table will contain **dot**, as with the iterator variables.

A general step expression, axis + name test, can be translated like this:

$$\overline{\texttt{AXIS::}QName_1}$$

$$\longmapsto$$

```
project(docId, index, value, pos, scope, $\vartheta$;
  numberate(dotNumb, [dotNumb, subIdx], [($\vartheta - \mathbf{dot}$)];
    numberate(index, [index], [$\vartheta$];
      select(isFUNC(scope, lsc);
        hhjoin([docId],[docId],[lsc=l.scope,subIdx=r.index,r.value,$\vartheta$];
          $\mathbf{get(dot)}$;
          numberate(index, [], [];
            index(valocc;
              lookup($\$QName_1$))))))))                                  (4.19)
```

Where ϑ is the ϑ returned from **get(dot)**, $QName_1$ is any XML-qualified name. r.value is short for value = right.value, index = right.index, ...etc, and docId is short for documentId. `AXIS` will map to an MQL funciton isFUNK as described in table 4.7. To ensure correct ordering, the order of the tuples returned from the lookup operator must be the same as the order of the tuples received by the numberate operator.

`AXIS`	isFUNC
`child`	isChild
`descendant`	isDescendant
`attribute`	isChild*
`self`	isSelf
`descendant-or-self`	isDescendantOrSelf
`following`	isFollowing
`following-sibling`	isFollowingSibling
`parent`	isParent
`ancestor`	isAncestor
`ancestor-or-self`	isAncestorOrSelf
`preceding`	isPreceding
`preceding-sibling`	isPrecedingSibling

Table 4.7: Mapping between XQuery axes and MQL functions.

**For the `attribute` axis the parameter to lookup will have a $@-prefix instead of the $-prefix described in the rule.*

The relation after the join will contain to copies of the *index* attribute stemming from the lookup, *index* and *subIdx*. This is because the numberate operator will remove the attributes specified in the sort list. The next to last numberate operator will number the tuples according to the document order, but among others partition on its dependency of the context item. This numbering is necessary to solve e.g. numeric predicates. The last numberate operator will update the *dotNumb* field as it is possible that the axis step do not match the items from the last step in a 1:1 ratio.

Example 9: Consider the excerpt of a XML-document of Figure 4.32. The subscript numbers are used to differentiate the different elements with same names, and are not a part of the names. Further, let a non-iterator variable `$a` be bound to a sequence of the `A`-nodes of the figure, but *not* in document order. An illustration of the relational representation of `$a` as it is stored as the context item in the symbol table is shown in Figure 4.33(a). The *val* attribute indicates which XML-node is represented and the *scope*-attribute indicates the scope of this element. Figure 4.31 shows an excerpt of a query referring to the variable `$a`.

```
...$a/child::B/...
```

Figure 4.31: Example XQuery path expression

First, a lookup of `$B` (where `$` indicates to find a scope/node, not a word) is done in the value occurence index. The result of this is numerated by a numberate-operator, as illustrated in figure 4.33(b). The *index*-attribute (*idx* in the figure) now holds the document-order of the `B`-nodes. As there may be more `B`-nodes in the document, the *index* attribute may not start at the value 1 for the tuples relevant to the query.

```
          ⋮
<A1>
  <B1/><B2/>
</A1>
<A2>
</A2>
<A3>
  <B3/><B4/><B5/>
</A3>
          ⋮
```

Figure 4.32: Excerpt of example XML-document. The subscript numbers indicate the instance of the elements, and are not part of the QName

dNb	*val*	*scope*
1	A_2	..A[2]
2	A_3	..A[3]
3	A_1	..A[1]

(a) `r($a)`

idx	*val*	*scope*
...	...	...
5	B_1	..A[1].B[1]
6	B_2	..A[1].B[2]
7	B_3	..A[3].B[1]
8	B_4	..A[3].B[2]
9	B_5	..A[3].B[3]
...	...	...

(b)

Figure 4.33: Illustration of results evaluating the expression in Figure 4.31. (a) The variable `$a`. (b) Excerpt of a lookup on the term $B, and the following numbering. Some attribute names are shortened. val attribute indicates which XML-element is represented in the tuple.

The result of the lookup will be joined with the context item relation on their *documentId*-attribute. As we assume only one XML-document, this attribute is omitted from our example. A select operator is applied to the result of the join to prune the relation. Only the tuples where the *scope* attribute stemming from the lookup defines a scope which is the child scope (as defined by the MQL function isChild() in Section 3.4.4) of the scope defined by the *scope*-attribute stemming from the `$a` relation (called *lsc*). After the selection the result will be as illustrated in figure 4.34(a). The copy of the *index* column is not shown.

Finally, numbering is employed two times, followed by a projection removing the last attributes stemming from the context item relation. The result of the expression is illustrated in Figure 4.34(b).

4.9.2 Predicates

A predicate consists of an expression, called a predicate expression, which evaluates to a boolean value. The expression assigned the predicate is called a predicated expression. A predicate serves to filter a sequence, retaining the items where the expression evaluates to

dNb	*idx*	*val*	*scope*	*lsc*
2	9	B_5	..A[3].B[3]	..A[3]
2	8	B_4	..A[3].B[2]	..A[3]
2	7	B_3	..A[3].B[1]	..A[3]
3	5	B_1	..A[1].B[1]	..A[1]
3	6	B_2	..A[1].B[2]	..A[1]

(a)

dNb	*idx*	*val*	*scope*
1	1	B_3	..A[3].B[1]
2	2	B_4	..A[3].B[2]
3	3	B_5	..A[3].B[3]
4	1	B_1	..A[1].B[1]
5	2	B_2	..A[1].B[2]

(b)

Figure 4.34: Further evaluation of the path expression in figure 4.31. (a) The result of the selection of the joining of **r($a)** *and the numerated result of the lookup of $B. (b) Renumbering an projection on the relation of (a). Some attribute names are shortened.*

true and discarding all other. In the case of multiple adjacent predicates, the predicates are applied from left to right, and the result of applying each predicate serves as the input sequence for the following predicate. If the value of the predicate expression is a singleton atomic value of a numeric type, the predicate truth value is *true* if the value of the predicate expression is equal to the position of the context item within the input sequence.

When entering a predicate the relational algebra tree corresponding to the expression assigned to the predicate is stored as **dot**, containing a copy of *dotNumb* called *sprDotNumb*. The reason for the copy is to ensure that the predicate is applied to the right context item of the predicated expression. If the predicate expression contains a path expression, it may update the *dotNumb* fields, but *sprDotNumb* will always correspond to the right context item outside the predicate. The predicate expression is then evaluated, and joined with the relational representation of predicated expression on their common dependencies. If the context item is referred to within the predicate, the relations will have to be joined on *sprDotNumb* from the predicate expression and *index* from the predicated expression aswell. The context item may be referred to explicitly with the dot-operator (.), or implicitly through a relative path expression. When evaluating path expressions it is important to note that predicates have higher precedence than the / (step expression separator). As the predicate may remove items, renumbering is needed:

$$\overline{e_1\,[e_2]} \longmapsto \begin{array}{l}\mathsf{project(index,\ value = l.value,}\ \vartheta;\\ \quad\mathsf{numberate(index,\ [index],\ [}\vartheta\mathsf{];}\\ \quad\quad\mathsf{select(ifthenelse(isNumber(pred),eq(index,pred),\ xqBoolean(pred));}\\ \quad\quad\quad\mathsf{hhjoin([(}e_1.\vartheta \cap e_2.\vartheta\mathsf{)],\ [(}e_2.\vartheta \cap e_1.\vartheta\mathsf{)],[l.value,pred,}\vartheta\mathsf{];}\\ \quad\quad\quad\quad\mathbf{r}(e_1);\\ \quad\quad\quad\quad\mathbf{B}(\mathbf{r}(e_2))))))\end{array} \tag{4.20}$$

Where $\vartheta = (e_1.\vartheta \cup e_2.\vartheta) - \mathbf{sprDot}$. The fields of the left relation in the join will follow l.value as described in Section 4.3. The final project operator will restore the names of these attributes.

As described earlier, before translation of the expression the context item will be store in the symbol table:

$$\begin{array}{l}\mathbf{put(dot,}\\ \quad\mathsf{project(sprDotNumb{=}index,\ dotNumb{=}index,\ index{=}1,\ value,}\ \vartheta;\\ \quad\quad\mathbf{r}(e_1))).\end{array}$$

However, if **dot** $\in e_1.\vartheta$, it will have to be stored in the following way:

$$\begin{array}{l}\textbf{put(dot,}\\ \quad \mathsf{project(sprDotNumb{=}dotNumb,\ dotNumb,\ index{=}1,\ value,}\ \vartheta;\\ \qquad \mathbf{r}(e_1))).\end{array}$$

In both cases, attributes not explicitly mentioned will follow *value* and the ϑ piggybacking the tree stored in the symbol table will be (**dot** $\cup$ **sprDot** $\cup$ $e_1.\vartheta$). The reason why there is both a *sprDotNumb* and *dotNumb* in the same relation is because *dotNumb* may be updated if the predicate is a path expression (ref Section 4.9.1).

If e_2 is dependent on **sprDot** – meaning the predicate utilises the context item – the relations will be joined on $\mathbf{r}(e_1).dotNumb = \mathbf{r}(e_2).sprDotNumb$ (as opposed to $\mathbf{r}(e_1).index = \mathbf{r}(e_2).sprDotNumb$ if it is not), as well as their common dependencies.

Example 10: Consider the XQuery query of Figure 4.35. In this query the predicate expression is dependent on the context item.

```
(2, 3, 4, 5)[. mod 2 = 0]
```

Figure 4.35: Example query with a predicate.

The predicated sequence is illustrated in 4.36(a). As it is not dependent on **dot** its *index* attribute will be renamed to *sprDotNumb* (*sdNb* in the figure) before it is stored in the symbol table. A reference to the context item is like a reference to a variable, and the predicate expression evaluates to the relation illustrated in Figure 4.36(b). The two relations only joined together on *index* from the predicated expression and *sprDotNumb* from the predicate expression, as they do not have any other dependencies. Further, as the predicate expression does not evaluate to a numeric type, only the tuples where *pred* is *true* are selected. This is shown in Figure 4.36(c).

idx	*val*
1	2
2	3
3	4
4	5

(a)

sdNb	*idx*	*val*
1	1	*true*
2	1	*false*
3	1	*true*
4	1	*false*

(b)

idx	*val*	*pred*
1	2	*true*
3	4	*true*

(c)

Figure 4.36: Evaluating the query in figure 4.35. (a) The sequence assigned the predicate. (b) The predicate expression. (c) The relations (a) and (b) joined together and pruned with a selection. dotNumb is omitted for simplicity. Attribute names are shortened.

To finish the evaluation of the query, after the selection, the relation will have to be renumbered and the *pred* attribute will have to be removed with a project operator.

4.10 Simplifications

In this section we will present some possible simplifications discovered during the development of the Tainting Dependencies method. The $\Rightarrow$ sign is to be read as “can be written as”.

4.10.1 Literals

Rule 4.5 of Section 4.3.4 shows a very general way to translate XQuery literals to a relational format. But creating one relation for each literal is very often unnecessary, and often quite a bit more resource consuming than alternative solutions. The parent expression of a litteral should in most cases be informed that its subexpression is a literal instead of being handed a one-tuple relation.

One such case is if the literal will be used in a join (predicate-less) or cartesian product. A better solution will then be to project the literal into the other relation. Following is an example of how such an expression should be written:

$$\begin{array}{l} \quad\vdots \\ \mathsf{hhjoin([],[],}\ldots \\ \quad \ldots\mathbf{r}(e)\ldots \\ \quad \mathbf{r}(\mathit{Literal}) \\ \quad\vdots \end{array} \Rightarrow \begin{array}{l} \quad\vdots \\ \mathsf{project(}\ldots\mathsf{,\ rvalue=}\mathit{Literal}\ldots \\ \quad \ldots\mathbf{r}(e)\ldots \\ \quad\vdots \end{array} \tag{4.21}$$

If the reason for the join with the relation was because the literal was part of a arithmetic, comparison or logical expression, the literal may be moved inside the project operator responsible for executing the binary operation, shortening the relational algebra even more:

$$\begin{array}{l} \mathsf{project(val=OP(l.val,\ r.val)}\ldots \\ \quad \mathsf{hhjoin([],[],}\ldots \\ \qquad \ldots\mathbf{r}(e)\ldots \\ \qquad \mathbf{r}(\mathit{Literal}) \\ \qquad\vdots \end{array} \Rightarrow \begin{array}{l} \mathsf{project(val=OP(val,\ }\mathit{Literal}\mathsf{)}\ldots \\ \quad \ldots\mathbf{r}(e)\ldots \\ \quad\vdots \end{array} \tag{4.22}$$

If the cases where a literal will have to be translated to a single-tuple relation, in most cases the *index* attribute will not be needed. But this will probably be detected by the optimiser, and may be left out anyway.

4.10.2 Sequence construction

Informing a parent expression about whether or not its subexpressions will evaluate to singleton sequences can have some advantages. One is the possibility of detecting certain type errors, as will be discussed in Section 7.2. Another advantage is gained when it comes to sequence construction:

Rule 4.6 in Section 4.4 describes a general way to build sequences. But if all the expressions to build a sequence from will evaluate to singleton sequences, there is no need for the numberate operator. Further, instead of adding *sprIdx* fields to specify the order, this can be done directly on the *index* fields. A version of Rule 4.6 can therefore be put like this:

$$\frac{\{e_1, \ldots, e_n\} \subset \mathit{Singletons}}{e_1\texttt{,}\ \ldots\texttt{,}\ e_n} \longmapsto \begin{array}{l}\mathsf{union(}\\ \quad\mathsf{project(index{=}1,\ value;}\\ \qquad\mathtt{t}(\mathbf{r}(e_1),\ \vartheta));\\ \qquad\quad\vdots\\ \quad\mathsf{project(index{=}}n\mathsf{,\ value;}\\ \qquad\mathtt{t}(\mathbf{r}(e_n),\ \vartheta)))\end{array} \tag{4.23}$$

Where $\vartheta = e_1.\vartheta \cup \ldots \cup e_n.\vartheta$.

If a sequence construction expression have only literal subexpressions, the translation may be even more simplified. As the rule stands now, the sequence will be built by splicing together single-tuple relations with an union operator. The make operator does however support multiple items, so a better solution would be to collect all items in one MQL operator:

$$\frac{\{e_1, \ldots, e_n\} \subset \mathit{Literals}}{e_1\texttt{,}\ \ldots\texttt{,}\ e_n} \longmapsto \begin{array}{l}\mathsf{make(name{:=}["index",\ "value"],}\\ \qquad\quad \mathsf{[1,\ \ldots,\ n],\ [}e_1\mathsf{,\ \ldots,\ }e_n\mathsf{])}\end{array} \tag{4.24}$$

These rules may even be combined, as literals are also singleton sequences. If all the items in the soon to be sequence are singletons, all singletons which are literals as well can be inserted into a relation with the same make operator. The *index* value of the items will have to be according to their relative position within the sequence construction expression. Following is an example of a sequence construction with only singleton items where not all of them are literals:

```
                              union(
                                project(index=3, value, bnumb;
('a', 'b', $b, 'c') ⟼             r($b));
                                t(make(name:=["index", "value"],
                                        [1, 2, 4],['a', 'b', 'c']), {b}))
```

4.10.3 Path expressions

The scope operator of MQL can be used to filter tuples based on the value of their *scope* field. The operator allows only complete scope descriptions, that is, no wildcards are allowed. / separates the scopes, and can be read as "encompasses" or "is the parent scope of". E.g. a/b is read as the scope where a is the parent scope of b. This can be exploited when translating path expressions with subsequent `child` axis or `parent` axis steps. Multiple subsequent `child` axis + name test steps can be translated like this:

$$\frac{}{\texttt{child::}QName_1\texttt{/child::}\ldots\texttt{/child::}QName_n}$$

$$\longmapsto$$

```
project(docId, index, value, pos, scope, $\vartheta$;
  numberate(dotNumb, [dotNumb, subIdx], [($\vartheta$ − dot)];
    numberate(index, [index], [$\vartheta$];
      select(isChild(scope, lsc);
        hhjoin([docId],[docId],
                [lsc=l.scope,subIdx=r.index,r.value,$\vartheta$];                  (4.25)
          get(dot);
          numberate(index, [], [];
            index(valocc;
              scope($QName_1/\ldots/QName_n$;
                lookup($\$QName_n$)))))))))
```

Where ϑ is the ϑ returned from **get(dot)**, $QName_x$ is any XML-qualified name. r.value is short for value = right.value, index = right.index, . . . etc, and docId is short for documentId.

Multiple `parent` axis steps can receive corresponding treatment, except the path defined to the scope operator will have to be reversed. Only clean axis + name test steps can be translated like this, without any interruption by e.g. a predicate or kind test.

Rule 4.18 shows how whole path expressions are translated. Here, the last step is renumbered by a numberate operator. If the last step does not include some kind of filtering, e.g. in form of a predicate, the operator can be exchanged with a project operator like this: project(index = dotNumb, value, ϑ;. . . .

The next to last numberate operator in the rules for translating axis + nametest expressions (rules 4.25 and 4.19) is employed to ensure that numeric predicates always can be evaluated. If the translator with the help of a type system be make sure that the predicate expression does not evaluate to a numeric type, or if the step expression does not contain a predicate, this operator may be dropped.

4.10.4 Arithmetic expressions

Rule 4.13 describes a translation of the unary - operator. If there is multiple consecutive unary operators, there is no need to apply the translation rule the same amount of times. The translator can count the number of unary - operators assigned to one expression. If the number of operators is odd, the rule is applied, if it is even, no translation is needed.

4.11 Summary

In this chapter we have presented Tainting Dependencies – a method for translating XQuery expressions into MQL relational algebra trees. Some of the base concepts behind the method is iterator dependencies and interator dependency tainting. An expression dependent on an iterator will have a relational representation for each iteration of that iterator. This dependency can taint another expression, if that expression is a subexpression of an expression whose evaluation requires representation for all iterations. We have presented methods for translating features of XQuery complying to the implications of tainting and dependencies. Finally, we presented some possible simplifications of trees generated with TD. In section 7.1 we will discuss translation of XQuery features not presented in this chapter.

Chapter 5

Implementation

This chapter describes the steps made to implement a proof of concept (dubbed "the prototype") which utilises some of the most important translation rules developed in Chapter 4. This includes an overall system description, as well as details about usage of the XQFT Parser. Furthermore, we describe the process of building a MQL algebra tree, and how the context sensitive visitor pattern is used. The scoping and symbol table implementation is covered, as well as how metadata is passed between nodes while parsing the syntax tree during the construction of the MQL tree. Finally, this chapter describes in detail how some of the rules from Tainting Dependencies are implemented and how they can be made to work in a real-life situation.

5.1 Prerequisites

As requested specifically by Fast Search & Transfer, this proof of concept was implemented in Java 5.0, using regular object oriented techniques, and is licensed under a liberal BSD license. Instructions for compilation and installation can be found in Appendix C.

5.2 List Of Supported Features

This implementation supports the translation of the following XQuery features, here annotated with references to the descriptions of their respective translation:

- FLWOR constructs (Section 4.5, page 53)
- Sequence construction (Section 4.4, page 51)
- Integer literals (Section 4.3.4, page 51)
- Conditional expressions (Section 4.7, page 62)
- Binary comparison (Section 4.6.2, page 59)

5.3 Overall system description

A simplified class diagram describing the essentials of the system is shown in Figure 5.1. The individual parts are described in detail in the following sections of this chapter. In this section, the overall system, flow of data, and external API is described.

It is important to note that, as mentioned, this implementation is a "proof of concept" and only implements a subset of Tainting Dependencies described in chapter 4.

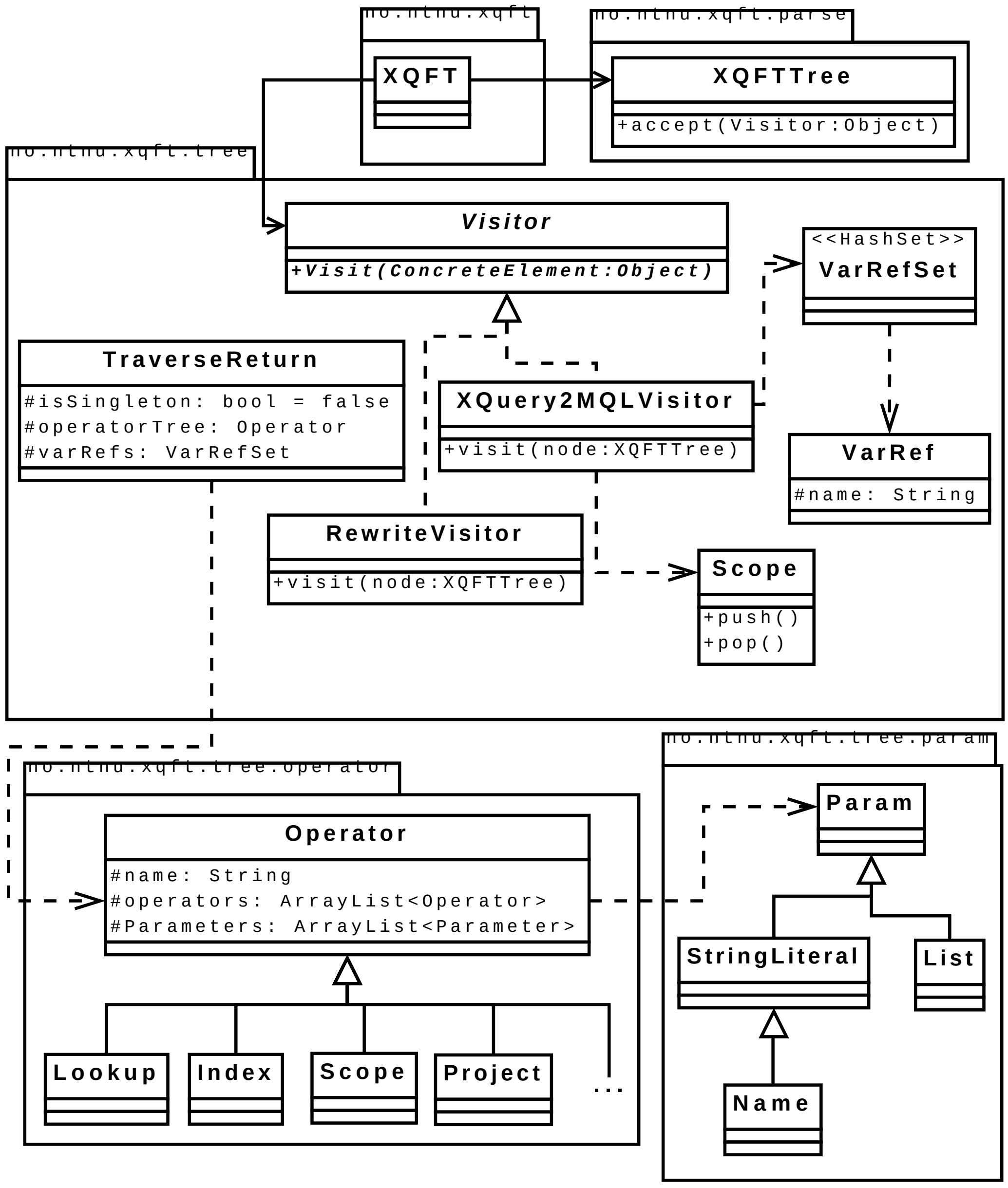


Figure 5.1: Simplified UML for complete implementation

5.3.1 Data flow

Figure 5.2 illustrates the flow of data when translating a XQuery query into a MQL query (see Section 3.4 on page 31 for a description of MQL).

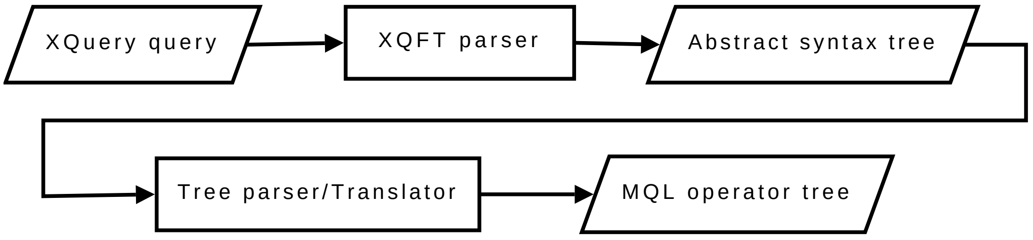
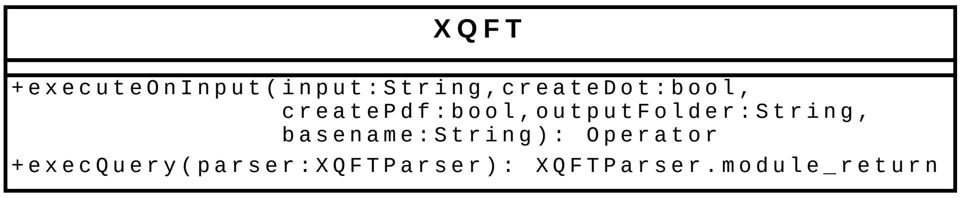


Figure 5.2: Data flow for XQuery parsing and translation to MQL

5.3.2 Visible external API

The API available to programmers is defined in a trivial manner in the `no.ntnu.xqft.XQFT` class. This class can also be executed as a standalone application (see next subsection). Figure 5.3 describes the `no.ntnu.xqft.XQFT` class.



```
XQFT
+executeOnInput(input:String,createDot:bool,
                createPdf:bool,outputFolder:String,
                basename:String): Operator
+execQuery(parser:XQFTParser): XQFTParser.module_return
```

Figure 5.3: External API for the XQuery to MQL translator

As can be seen, two methods are primarily available. Out of these two, `executeOnInput()` is the most complex, but also the most flexible. A typical usage scenario for an external user could be as follows:

```
XQFT xqft = new XQFT();
Operator mqlTree = xqft.executeOnInput(
                        "for $i in (1,2,3) return $i",
                        false, false, null, null
                   );
```

The `mqlTree` would now be a reference to a complete MQL operator tree, provided that no errors occured during the parse process or the translation process.

5.3.3 Command line interface

The command line interface is available by executing the `no.ntnu.xqft.XQFT` class as a main class, as mentioned in the previous section. The command line interface uses the Args Engine[1] for the sake of simplicity to parse options/switches on the command line.

[1]http://www.adarshr.com/papers/args

The command line usage is as follows:

```
java no.ntnu.xqft.XQFT [-p] [-t] [-o <path>] file1 file2 ... fileN
```

It is also possible to specify queries in the form of strings enclosed in double quotes, or any mix of strings and filenames. The switches are:

- `-t` : output a DOT tree (requires graphviz)
- `-p` : output a PDF syntax tree (requires graphviz)
- `-o <path>` : stores generated PDF/DOT files in the given folder, otherwise in the current folder (simply `./`)

See Appendix C for more information about installation and dependencies.

5.4 Using the XQFT Parser

The *XQFT Parser*[19] (described in section 2.4.3) is a prerequisite for providing the abstract syntax tree for this XQuery translator. This section will outline how this parser was used and interfaced with the implementation.

The *XQFT Parser* is a parser generated by the ANTLR parser generator. Thus, there is a loosely standardised API available for any implementor utilising a parser generated by ANTLR. In the case of *XQFT Parser*, two classes are generated: `XQFTParser` and `XQFTLexer`. These classes are used in conjunction on an input string to produce an abstract syntax tree (see next subsection, and also section 2.4.3).

A typical use case to achieve this is shown in figure 5.4, which is copied almost verbatim from the implementation.

```
CharStream cs
    = new ANTLRStringStream(
        "for $i in (1,2,3) return $i");

XQFTLexer lexer = new XQFTLexer(cs);

UnbufferedCommonTokenStream tokens
    = new UnbufferedCommonTokenStream();
    tokens.setTokenSource(lexer);

XQFTParser parser = new XQFTParser(tokens);
parser.setTreeAdaptor(new XQFTTreeAdaptor());
parser.setLexer(lexer);

XQFTTree ast = parser.module().getTree();
```

Figure 5.4: Using the XQFTParser and XQFTLexer classes

Note the use of `ANTLRStringStream`, `UnbufferedCommonTokenStream`, and `XQFTTreeAdaptor`. The latter, `XQFTTreeAdaptor`, is a specialised class required to create instances of the `XQFTTree` class to represent nodes in the abstract syntax tree.

The actual parsing is performed by calling the method `module()`, which is the top-level production rule in the grammar for the XQFT parser (see Appendix D).

The `XQFTTree` class represents a node in the produced syntax tree. When a syntax tree is returned from the parser, the root node is an instance of this class, as well as all children (see Figure 5.4)

To make practical use of the XQFT Parser, what remains is nothing more than to translate the abstract syntax tree acquired from the call to `getTree()`, which is the object `ast` on the last line of code in Figure 5.4.

5.5 Constructing the MQL algebra tree

MQL queries are constructed as trees, where each node represents an operator. Each node is an instance of an operator class, and may contain a list of child operators and a list of parameters. The trees are constructed bottom-up while parsing the abstract syntax tree corresponding to a XQuery query.

5.5.1 Operators and parameters

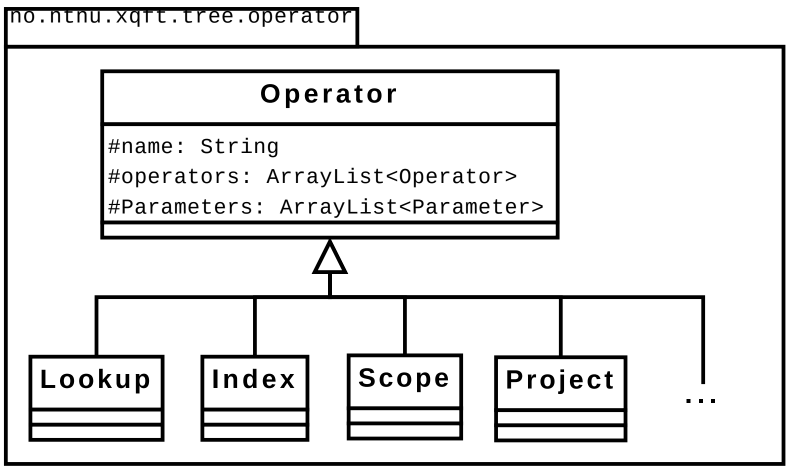

no.ntnu.xqft.tree.operator
Operator
#name: String
#operators: ArrayList<Operator>
#Parameters: ArrayList<Parameter>
Lookup
Index
Scope
Project
...

Figure 5.5: Simplified class diagram of MQL operators

The operators modeled in the implementation correspond to the operators described in Section 3.4.3. A simplified class diagram is shown in Figure 5.5. Converting an operator to a string is in most cases handled by the default fallback in the `Operator` class, where the string generated will be on the form:

```
operator_name(param1, param2, ..., paramN;
    operator1;
    operator2;
    ...;
    operatorM)
```

Constructing such a string for the complete query tree is achieved by calling `toPrettyString(0)` on the root node. The parameter to the method specifies the initial indentation.

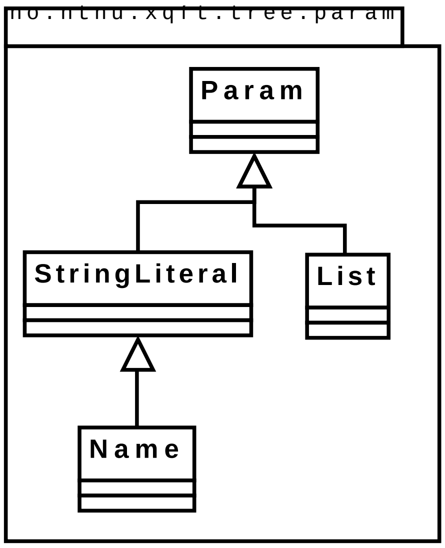


Figure 5.6: Class diagram of MQL parameters

MQL parameters (as described in 3.4.1) are modeled as seen in Figure 5.6. Parameters require no complex structure, and are only created and added to operators as needed.

5.5.2 Usage

The operator classes are designed to be intuitive and simple to use. Figure 5.7 shows one example where a simple operator tree is built and converted to an MQL query string. The result can be seen in Figure 5.8).

```
Lookup lookup = new Lookup("Death in the clouds");
Scope scope = new Scope("/books/book/title", lookup);
Project project = new Project("author", scope);
System.out.println(project.toPrettyString(0));
```

Figure 5.7: Example java code to construct a MQL operator tree

```
project(author;
  scope(/books/book/title;
    lookup("Death in the clouds")))
```

Figure 5.8: Resulting MQL query string from example in figure 5.7

5.6 Context-sensitive Visitor

In Section 2.4.4 a number of techniques for tree parsing were presented. In Section 3.2 the *context-sensitive visitor pattern* was chosen as the technique for this implementation.

The context-sensitive pattern is designed to be flexible and to generate code with a higher level of maintainability, for which the rationale was presented in Section 2.4.4.

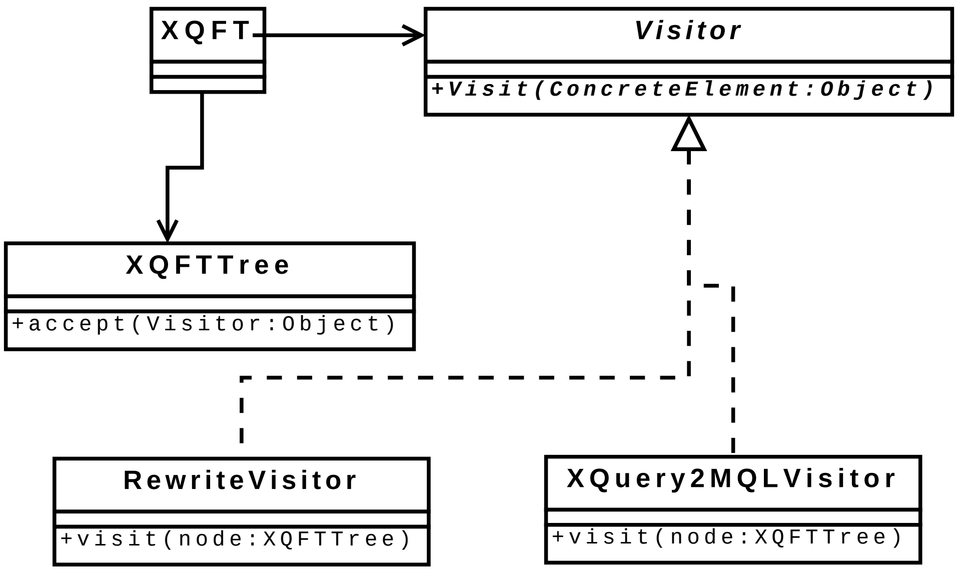


Figure 5.9: Context sensitive visitor implementation

The class diagram for the actual implementation of the context-sensitive visitor pattern can be seen in Figure 5.9. Compare this to the generalised class diagram in figure 2.17 on page 22.

Note that the use of `XQFTTree` as the element class implies that the `XQFTTree` be supplemented with an `accept()` method to accommodate this pattern. This method is essentially a static dispatcher which will call the appropriate method on the visitor based on the token type of the node currently being visited. Figure 5.10 shows an excerpt of this method and how it acts on the visitor class.

```
public TraverseReturn accept(Visitor visitor) {
    switch(this.getType()) {
        case XQFTParser.AST_MODULE:
            return visitor.visitAST_MODULE(this);
        case XQFTParser.AST_FLWOR:
            return visitor.visitAST_FLWOR(this);
        case XQFTParser.AST_FORCLAUSE:
            return visitor.visitAST_FORCLAUSE(this);
        case XQFTParser.AST_LETCLAUSE:
            return visitor.visitAST_LETCLAUSE(this);
        case XQFTParser.AST_ORDERBYCLAUSE:
            return visitor.visitAST_ORDERBYCLAUSE(this);
        case XQFTParser.AST_WHERECLAUSE:
            return visitor.visitAST_WHERECLAUSE(this);
                            ...
```

Figure 5.10: Excerpt from the accept() method in the XQFT class

The prototype utilises two different visitors, namely the *Rewrite visitor* and the *XQuery2MQL* *visitor*. The *Rewrite visitor* is used to perform rewrite operations on the abstract syntax tree before performing the actual translation. In particular, these rewrite operations consists of normalising the required subtrees of the syntax tree to a subset of XQuery Core (as described in sections 2.1.6 and 3.3).

The *XQuery2MQL visitor* performs the bulk of the work related to performing the translation of XQuery to MQL. This visitor is capable of re-instantiating itself (or other visitors) when entering new contexts, such as path predicates.

5.7 Scoping and Symbol Tables

Crucial to the implementation of the Tainting Dependencies methodology described in Chapter 4 is the ability to maintain a contextual environment with scoping and symbol tables. This section details the implementation of this, and how it is used to meet the requirements of TD.

5.7.1 Concepts

The scoping system in the implementation is based on building a scope tree. The previous scope, if any, is set as parent of the new scope, and the previous scope maintains a list of child scopes – this is referred to as *pushing a scope*. When exiting a scoped subexpression in the AST, the previous scope is again set as the current scope. This is referred to as *popping a scope*. A reference to the root scope node is always maintained. Considering the example XQuery query in figure 5.11, the scope tree in figure 5.12 is generated. The scope itself contains *one* symbol table for the current scope.

```
for $i in (1,2,3) return
  for $a in (4,5,for $b in (6,7,8) return $b)
    return ($i,$a)
```

Figure 5.11: Scope tree example code

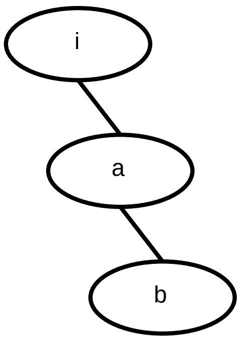


Figure 5.12: Scope tree for source code in figure 5.11

Entries in the symbol table are represented through an instance of the `SymTabEntry` class which maintains metadata about symbols (such as symbol name, a flag indicating whether it is an iterator variable, and an evaluated expression). The symbol table is realised as a subclass of the `HashMap` class in the `java.util` package, and is constrained to storing instances of `SymTabEntry`, with the symbol name as key.

5.7.2 Semantics

The scoping semantics are encapsulated in a singleton manner in the class `Scope`, with static methods available for pushing and popping scopes, and storing and retrieving symbols. The external (static) API as available to a user of the scope system is shown in Figure 5.13.

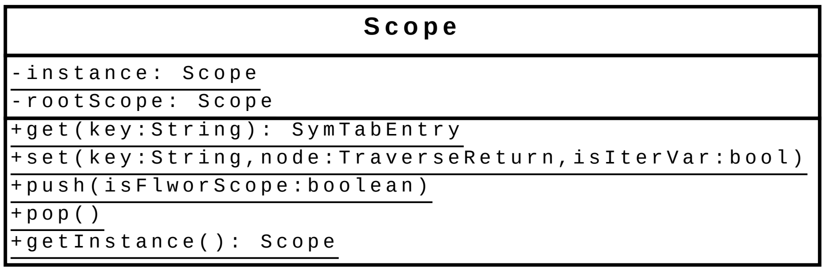

Scope
-instance: Scope
-rootScope: Scope
+get(key:String): SymTabEntry
+set(key:String,node:TraverseReturn,isIterVar:bool)
+push(isFlworScope:boolean)
+pop()
+getInstance(): Scope

Figure 5.13: Scope API

A new scope is *pushed* whenever a `for`-clause is encountered while parsing the abstract syntax tree, and the current scope is *popped* after evaluating a `return`-clause – both of which occur within a FLWOR expression.

The scoping system also tracks iteration variables. That is, for any scope, there is *one and only one* iteration variable, except in the top scope where there is no iteration variable. The concept of an iteration variable is explained in Definition 9. Tracking of these variables are reviewed in Section 5.9.

5.8 Passing Metadata Between Nodes

To implement the Tainting Dependencies method it is necessary to pass metadata upwards when parsing the syntax tree, such as iterator dependencies and flags to indicate singleton nodes (for simplifications). Additionally, the operator tree which is being built bottom-up (as described earlier in section 5.5) is also required to be passed upwards.

This is achieved through the `TaverseReturn`, which models a return type when visiting nodes in the syntax tree. That is, the visitor methods are responsible of 1) visiting any child nodes, and 2) returning an instance of the `TraverseReturn` class based on what was returned from the child nodes, if anything.

5.8.1 The TraverseReturn Class

The class diagram for the `TraverseReturn` class is shown in figure 5.14. Note the flag to indicate if the current context is a singleton, the reference to an MQL operator tree (which is being built bottom-up), and a reference to a set of iterator dependencies (in the implementation called `varRefs`).

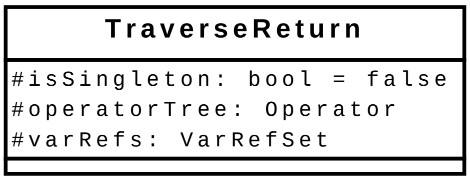


Figure 5.14: TraverseReturn class diagram

The `TraverseReturn` class is, as mentioned, used in the visitor when visiting nodes in the abstract syntax tree (see section 5.6). A typical use case is shown in figure 5.19, which is an excerpt from the implementation.

5.8.2 Iterator Dependencies

Iterator dependencies, described in Section 4.3, are passed upwards together with the MQL operator tree being built during the syntax tree parsing process. These sets of dependencies are handled by th `VarRef` and `VarRefSet` classes. A class diagram for these classes is shown in Figure 5.15.

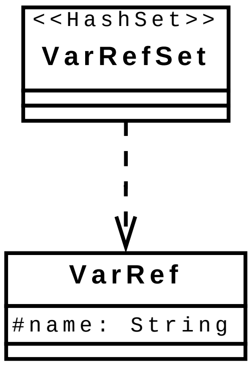


Figure 5.15: `VarRefSet` and `VarRef` class diagram

As described in Section 4.3.1, an iterator variable reference is always dependent on its corresponding iterator. Thus, when a iterator variable is encountered during the parse process, and the variable is being "read" and not assigned or declared, the corresponding iterator is added to the current set of iterator dependencies. The example in Figure 5.16 shows the variable `$a` being read, in which case the iterator is added to the `TraverseReturn` to-be-returned's set of dependencies.

```
for $i in (1,2,3) return
    ($a,4,5)
```

Figure 5.16: Example of the variable `$a` being read. Note that the iterator variable `$i` is never read

The source code excerpt in Figure 5.17 shows how iterator dependencies are treated in the visitor implementation.

```
// Fetch entry from symtab
SymTabEntry entry = Scope.get(tree.getChild(0).getText());

// Obtain and append new var ref
TraverseReturn tr = entry.getTraverseReturn();
tr.getVarRefs().add(new VarRef(tree.getChild(0).getText()));

return tr;
```

Figure 5.17: Appending a new variable reference

5.8.3 Singleton nodes

Singleton nodes are nodes corresponding to expressions that return a sequence of exactly one item. In the cases where this is known to be true, the result from a translation can be tagged with this information and used later to simplify the translation of sequence construction (as described in Section 5.9.3).

This is the case of integer literal nodes as well as iterator variable lookups in the symbol table. The case of integer literal nodes is shown in figure 5.19 in the next section. The case of variable lookups is somewhat less intuitive, since the singleton flag is actually stored when a variable is first set. That is, the right-hand side of the assignment is translated once and annotated with the singleton flag, which is then set for all subsequent lookups in the symbol table. The excerpt in figure 5.18 shows how this is done in the implementation.

```
// Visit children on the right side of the assignment
TraverseReturn tr = acceptThis(tree.getChild(1));

// Required for tainting deps method
Project project = new Project("[" + varName + "numb, value]",
                      tr.getOperatorTree());

// Assign metadata
tr.setOperatorTree(project);
tr.setSingleton(true);

// Enter into symbol table
SymTabEntry tmp = Scope.set(tree.getChild(0).getText(),
                      tr, isIterationVar);
```

Figure 5.18: Iterator variable assignment example, annotated with the singleton flag before being entered into the symtab

5.8.4 Example of usage

In the example in Figure 5.19, an integer literal node is visited (a node that simply holds an integer). A `make()` MQL operator as well as a new `TraverseReturn` instance is created. The `make()` operator is then appended to the `TraverseReturn` instance, and the *isSingleton* flag is set to *true* since the result of this translation is a single item.

```
public TraverseReturn visitIntegerLiteral(XQFTTree tree) {

    Make make = new Make("name:=[index, value], [1, " + tree.getText());
    TraverseReturn tr = new TraverseReturn();
    tr.setSingleton(true);
    tr.setOperatorTree(make);
    return tr;
}
```

Figure 5.19: TraverseReturn usage example

5.9 Tainting dependencies

Tainting Dependencies is a method of translating XQuery queries to relational algebra. The semantics of this method is described in detail throughout Chapter 4. This section describes an implementation of a subset of the rules in this method – an implementation which is capable of translating simple FLWOR expressions, sequences, and variables.

5.9.1 Tainting

The concept of tainting one expression with the iterator dependencies of another is described in section 4.3.2 on page 49. A method `taint()` is introduced to cover the semantics of this concept. Residing in the *XQuery2MQL visitor* class it is reachable from all nodes of the tree. The method is implemented like this:

```
protected TraverseReturn taint(TraverseReturn tr, VarRefSet varRefs) {

    TraverseReturn result = new TraverseReturn();

    VarRefSet taintBy = (VarRefSet)varRefs.clone();
    taintBy.removeAll(tr.getVarRefs());

    Operator expr = tr.getOperatorTree();
    Project project;
    for (VarRef varRef : taintBy) {
        project = new Project(varRef.getName() + "numb",
                      Scope.get(varRef.getName()).getOperatorTree());
        expr = new Cross(project, expr);
    }

    tr.getVarRefs().addAll(taintBy); \\ Add gained dependencies
    result.setVarRefs(tr.getVarRefs());
    result.setOperatorTree(expr);
    result.setSingleton(tr.isSingleton());

    return result;
}
```

5.9.2 FLWOR expressions

The translation process for FLWOR expressions was outlined in section 4.5. Consider inference rule 4.2 on page 48. This inference rule states how to translate and bind an iterator variable in a `for`-clause in a FLWOR expressions. Furthermore, consider the abstract syntax tree example in Figure 5.20. First a `for`-clause is visited, and the child is flagged as a FLWOR tuplet definition.

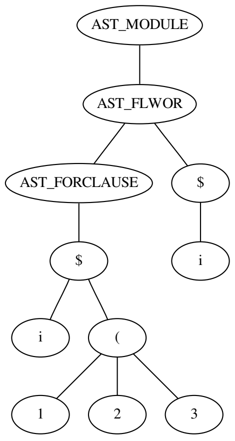


Figure 5.20: FLWOR syntax tree example

Next a dollar sign is visited (which carries the meaning of a variable in the abstract syntax tree). If the child count is more than one, it is an assignment. Note that the `isIterationVar` flag is *true* if this assignment is a tuple definition as flagged earlier. Then, if this is an assignment and a tuple definition, the right-hand side of the assignment is translated, and the symbol is entered into a symbol table. The creation of the project operator is required by inference rule 4.2:

```
// Visit children on the right side of the assignment
TraverseReturn tr = acceptThis(tree.getChild(1));

// Augment with -numb attribute
Project project = new Project(varName + "numb, index=1, value",
   tr.getOperatorTree());

// Assign metadata
tr.setOperatorTree(project);
tr.setSingleton(true);

// Enter into symbol table
SymTabEntry tmp = Scope.set(tree.getChild(0).getText(),
                      tr, isIterationVar);

if (tree.isFlworTupleDef()) {
   Scope.setCurrentIterVar(new VarRef(tmp.getName()));
}
```

Following the translation of the `for`-clause, the `return`-clause and its subexpressions are translated. When the visitor returns to the AST_FLWOR node, as there is no `where` or `order by`-clauses iterator ordering (Section 4.5.1, page 54) is applied:

```
...
// Taint if needed
returnClause = this.taint(returnClause, Scope.getCurrentIterVar());

// Remove current iterator from dependencies
VarRefSet newVarRefs
        = (VarRefSet)returnClause.getVarRefs().clone();
newVarRefs.remove(Scope.getCurrentIterVar());

// Sort and partition fields
String[] sortBy = {Scope.getCurrentIterVar().getName()
                          + "numb", "index"};
String[] partitionBy
        = new String[returnClause.getVarRefs().size() - 1];

int i = 0;
for (VarRef ref : prevVarRefs) {
    partitionBy[i] = ref.getName();
    i++;
}

// Construct MQL
Numberate numberate = new Numberate("index",
                                    sortBy,
                                    partitionBy,
                                    returnClause.getOperatorTree());

// Construct result
TraverseReturn result = new TraverseReturn();
result.setSingleton(false);
result.setVarRefs(newVarRefs);
result.setOperatorTree(numberate);

Scope.pop();
return result;
```

5.9.3 Sequences

The translation process for sequence construction is described in section 4.4. First, the iterator dependencies of the expression is calculated. This is used to taint all the subexpressions:

```
boolean allSingletons = true;

// Collect all iterator dependencies
VarRefSet allVarRefs = new VarRefSet();
for (TraverseReturn childResult : childResults) {
    allVarRefs.addAll(childResult.getVarRefs());
    if(!childResult.isSingleton())
```

```
            allSingletons = false;
    }

    Union union = new Union();

    for (TraverseReturn childResult : childResults) {
        if(allSingletons)
            projectString = "index = " + c + ", value";
        else
            projectString = "sprIdx = " + c + ", index, value";

        union.addOperator(new Project(projectString,
                this.taint(childResult, allVarRefs).getOperatorTree()))
        c++;
    }

    if(!allSingletons)
        ...
```

The tainted expressions are added to an union operator. But first they will have to be inserted into a project operator, with parameters depending on they are all singleton sequences or not. If they are, the union is wrapped in a `TraverseReturn`, completing the translation of the sequence constructor. If they are not, a numberate operator is needed.

Note that the parantheses in sequence expressions are not required, and according to specification, sequence expressions are recognised by the comma symbols and not parantheses. However, the XQFT Parser rewrites sequence expressions into including a paranthesis as start token for sequence subtrees within the AST.

5.9.4 If-then-else

The translation process for conditional expressions is explained in section 4.7. In particular, rule 4.16 describes this translation.

First the child expressions are visited, and the variables `e1`, `e2`, and `e3` correspond to the expressions e_1, e_2, and e_3 in Rule 4.16, while `r_e1`, `r_e3`, and `r_e3` correspond to $\mathbf{r}(e_1)$, $\mathbf{r}(e_2)$, and $\mathbf{r}(e_3)$.

Continuing, sets of iterator dependencies are obtained:

```
// VarRefs: e2 union e3
VarRefSet v_e2_u_e3 = (VarRefSet)r_e2.getVarRefs().clone();
v_e2_u_e3.addAll(r_e3.getVarRefs());

// VarRefs: (e2 union e3) intersect e1
VarRefSet v_e2_u_e3_i_e1 = (VarRefSet)v_e2_u_e3.clone();
v_e2_u_e3_i_e1.retainAll(r_e1.getVarRefs());

// VarRefs: e1 union e2 union e3
VarRefSet v_e1_u_e2_u_e3 = (VarRefSet)r_e1.getVarRefs().clone();
v_e1_u_e2_u_e3.addAll(r_e2.getVarRefs());
v_e1_u_e2_u_e3.addAll(r_e3.getVarRefs());
```

The variable v_e2_u_e3 corresponds to $e_2 \cup e_3$, v_e2_u_e3_i_e1 corresponds to $(e_2 \cup e_3) \cap e_1$, and v_e1_u_e2_u_e3 corresponds to $e_2 \cup e_3 \cup e_1$.

This result is used to create the to project() operators and union them together:

```
// Alternatives
Project alt1 = new Project("index, alt=1, " + v_e2_u_e3.toStringList() + ", value",
                      this.taint(r_e2, r_e3.getVarRefs()).getOperatorTree());

Project alt2 = new Project("index, alt=2, " + v_e2_u_e3.toStringList() + ", value",
                      this.taint(r_e3, r_e2.getVarRefs()).getOperatorTree());

// Union
Union union = new Union(alt1, alt2);
```

The translation is finalised by using this result to construct a join and apply the select() operator:

```
// HHjoin
HHJoin hhjoin = new HHJoin("[" + v_e2_u_e3_n_e1.toStringList() + "]," +
                           "[" + v_e2_u_e3_n_e1.toStringList() + "]," +
                           "[index = l.index, " + v_e1_u_e2_u_e3.toStringList() +
                           ", lvalue = l.value, rvalue = r.value]",
                             union, r_e1.getOperatorTree());

// Select
Select select = new Select("ifthenelse(xqBoolean(rvalue),
                    eq(alt,1), eq(alt,2))", hhjoin);

// Project
Project project = new Project("index, " +
                  v_e1_u_e2_u_e3.toStringList() +
                  ", value = lvalue" , select);
```

5.10 Summary

This chapter has described the implementation of a proof of concept for the "Tainting Dependencies" method. In the next chapter, some results are presented, such as theoretical algebra output. Additionally, algebra generated by the prototype described here will be compared to that generated by Pathfinder.

Chapter 6

Results

This chapter will demonstrate a series of relational algebra trees. First, in Section 6.1, example trees computed by hand using the rules for Tainting Dependencies are presented, utilising some major capabilities of this method which is not supported by the prototype implementation. Further, in Section 6.2, some trivial and complex queries are generated by the prototype implementation and displayed. Furthermore, in Section 6.3, comparisons are made to algebra generated by Pathfinder which uses classic Loop Lifting techniques.

6.1 Theoretical Algebra

In this section, a collection of XQuery query examples and their translation to relational algebra are presented. The translation is done manually using the "Tainting Dependencies" method described in Chapter 4, and includes the simplifications specified in section 4.10. For the sake of brevity, only the rules used throughout the translation will be noted. Intermediate translations are not shown here, however they are all shown in their entirety in appendix A.

6.1.1 Extensive FLWOR

This example will illustrate the translation of the following FLWOR expression:

```
for $a in (1,2,3) let $b := 2
  where $a gt $b
  order by $a
  return ($a, $b)
```

The translation process in its entirety is shown step by step in Appendix A.1, page 133. And the result of the translation is shown in figure 6.1. The operator tree can be converted to the DAG seen in Figure 6.2.

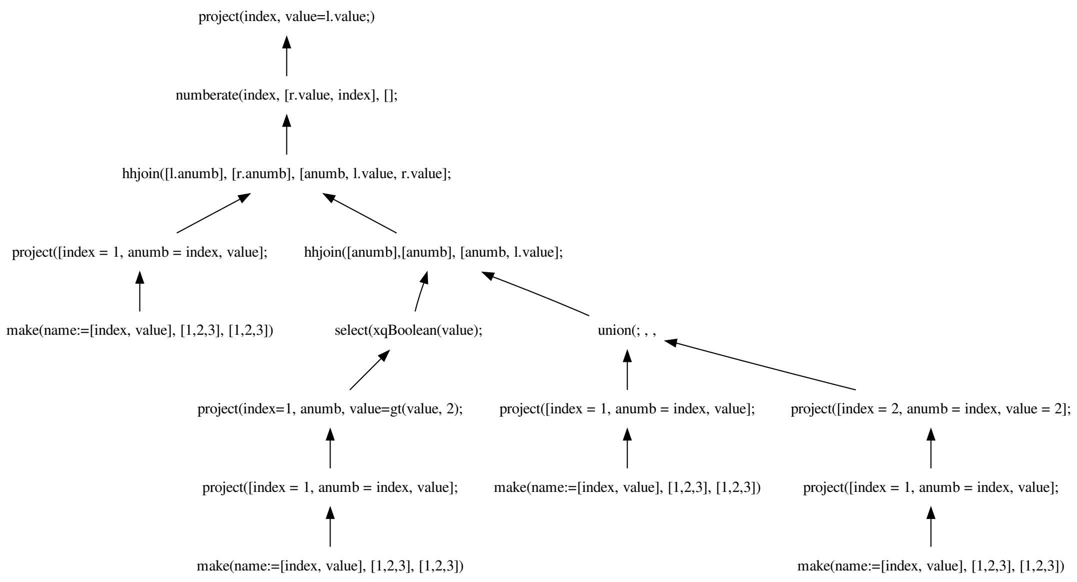

project(index, value=l.value;)
numberate(index, [r.value, index], [];
hhjoin([l.anumb], [r.anumb], [anumb, l.value, r.value];
project([index = 1, anumb = index, value];
hhjoin([anumb],[anumb], [anumb, l.value];
make(name:=[index, value], [1,2,3], [1,2,3])
select(xqBoolean(value);
union(; , ,
project(index=1, anumb, value=gt(value, 2);
project([index = 1, anumb = index, value];
project([index = 2, anumb = index, value = 2];
project([index = 1, anumb = index, value];
make(name:=[index, value], [1,2,3], [1,2,3])
project([index = 1, anumb = index, value];
make(name:=[index, value], [1,2,3], [1,2,3])
make(name:=[index, value], [1,2,3], [1,2,3])

Figure 6.1: Complete translation of expression extensive FLWOR expression

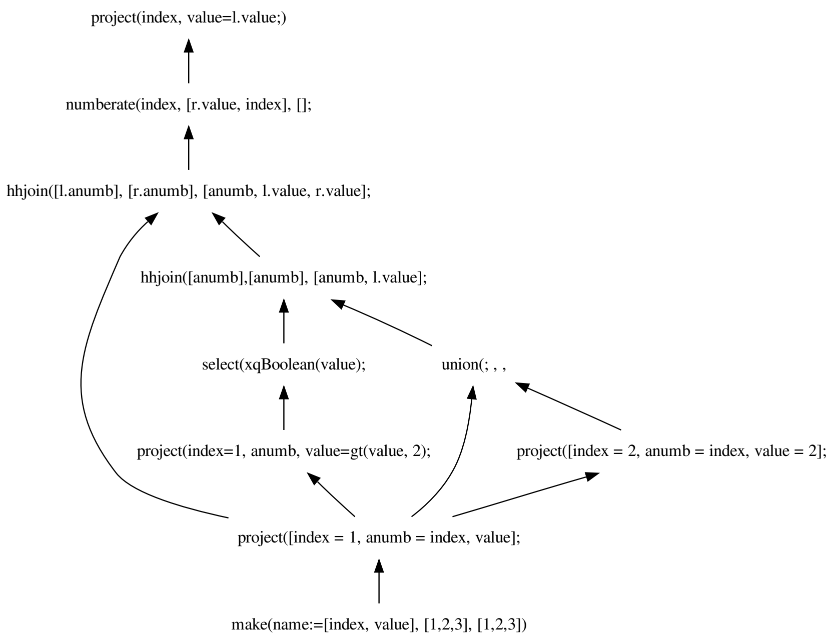

project(index, value=l.value;)
numberate(index, [r.value, index], [];
hhjoin([l.anumb], [r.anumb], [anumb, l.value, r.value];
hhjoin([anumb],[anumb], [anumb, l.value];
select(xqBoolean(value);
union(; , ,
project(index=1, anumb, value=gt(value, 2);
project([index = 2, anumb = index, value = 2];
project([index = 1, anumb = index, value];
make(name:=[index, value], [1,2,3], [1,2,3])

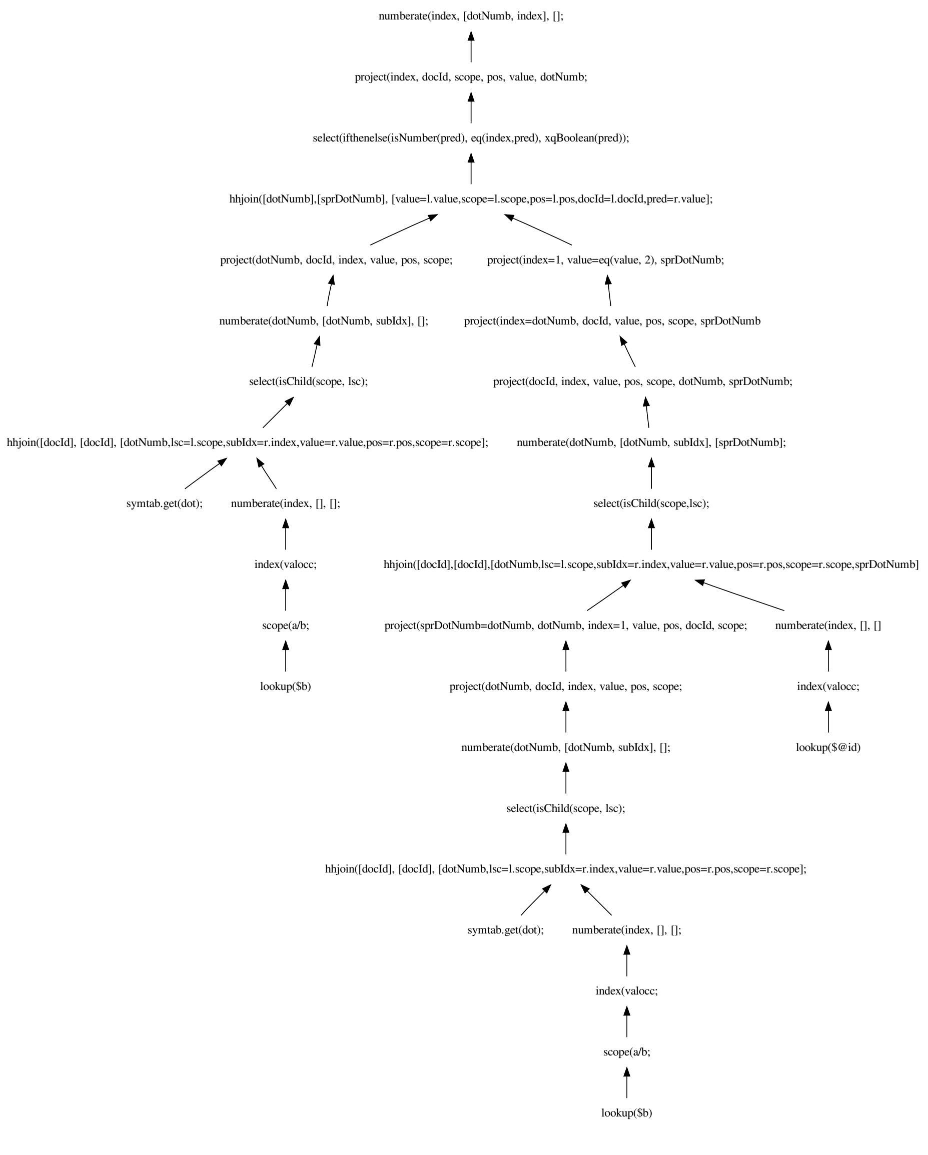
Figure 6.2: DAG representation of the translated extensive FLWOR expression

6.1.2 Path expression with predicate

This example will illustrate the translation of a path expression with a predicate:

```
/a/b[@id eq 2]
```

The translation process in its entirety is shown step by step in Appendix A.3, page 140, and the result of the translation is shown in figure 6.3. The operator can be converted to the DAG seen in Figure 6.4.



numberate(index, [dotNumb, index], [];
project(index, docId, scope, pos, value, dotNumb;
select(ifthenelse(isNumber(pred), eq(index,pred), xqBoolean(pred));
hhjoin([dotNumb],[sprDotNumb], [value=l.value,scope=l.scope,pos=l.pos,docId=l.docId,pred=r.value];
project(dotNumb, docId, index, value, pos, scope;
project(index=1, value=eq(value, 2), sprDotNumb;
numberate(dotNumb, [dotNumb, subIdx], [];
project(index=dotNumb, docId, value, pos, scope, sprDotNumb
select(isChild(scope, lsc);
project(docId, index, value, pos, scope, dotNumb, sprDotNumb;
hhjoin([docId], [docId], [dotNumb,lsc=l.scope,subIdx=r.index,value=r.value,pos=r.pos,scope=r.scope];
numberate(dotNumb, [dotNumb, subIdx], [sprDotNumb];
symtab.get(dot);
numberate(index, [], [];
select(isChild(scope,lsc);
index(valocc;
hhjoin([docId],[docId],[dotNumb,lsc=l.scope,subIdx=r.index,value=r.value,pos=r.pos,scope=r.scope,sprDotNumb]
scope(a/b;
project(sprDotNumb=dotNumb, dotNumb, index=1, value, pos, docId, scope;
numberate(index, [], []
lookup($b)
project(dotNumb, docId, index, value, pos, scope;
index(valocc;
numberate(dotNumb, [dotNumb, subIdx], [];
lookup($@id)
select(isChild(scope, lsc);
hhjoin([docId], [docId], [dotNumb,lsc=l.scope,subIdx=r.index,value=r.value,pos=r.pos,scope=r.scope];
symtab.get(dot);
numberate(index, [], [];
index(valocc;
scope(a/b;
lookup($b)

Figure 6.3: Complete translation of the path expression.

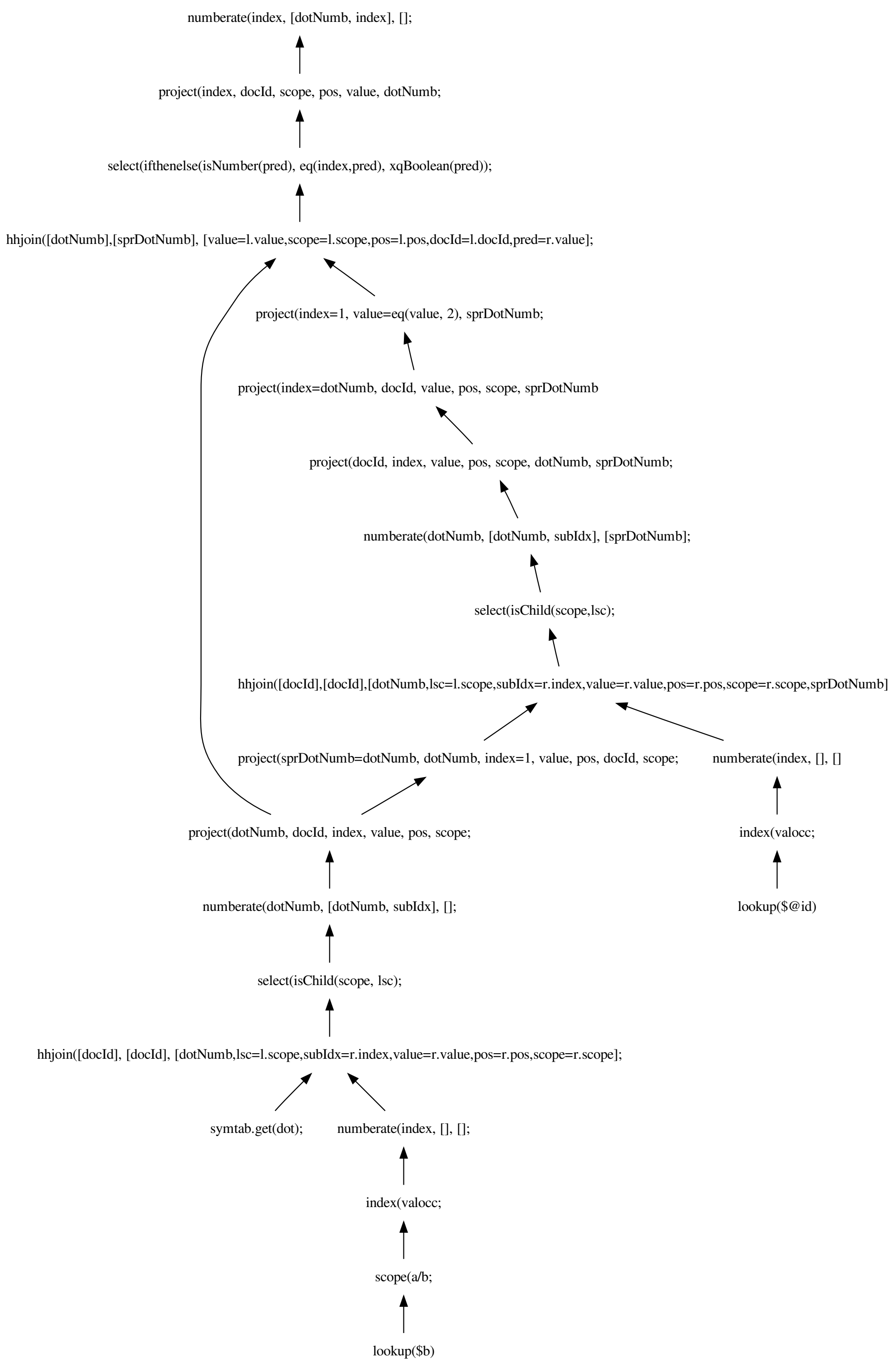

numberate(index, [dotNumb, index], [];
project(index, docId, scope, pos, value, dotNumb;
select(ifthenelse(isNumber(pred), eq(index,pred), xqBoolean(pred));
hhjoin([dotNumb],[sprDotNumb], [value=l.value,scope=l.scope,pos=l.pos,docId=l.docId,pred=r.value];
project(index=1, value=eq(value, 2), sprDotNumb;
project(index=dotNumb, docId, value, pos, scope, sprDotNumb
project(docId, index, value, pos, scope, dotNumb, sprDotNumb;
numberate(dotNumb, [dotNumb, subIdx], [sprDotNumb];
select(isChild(scope,lsc);
hhjoin([docId],[docId],[dotNumb,lsc=l.scope,subIdx=r.index,value=r.value,pos=r.pos,scope=r.scope,sprDotNumb]
project(sprDotNumb=dotNumb, dotNumb, index=1, value, pos, docId, scope;
numberate(index, [], []
project(dotNumb, docId, index, value, pos, scope;
index(valocc;
numberate(dotNumb, [dotNumb, subIdx], [];
lookup($@id)
select(isChild(scope, lsc);
hhjoin([docId], [docId], [dotNumb,lsc=l.scope,subIdx=r.index,value=r.value,pos=r.pos,scope=r.scope];
symtab.get(dot);
numberate(index, [], [];
index(valocc;
scope(a/b;
lookup($b)

Figure 6.4: DAG representation of the translated path expression.

6.1.3 If-then-else

This example will illustrate the translation of a conditional expression:

```
for $a in (1,2,3) return
  if $a gt 2 then $a else 3
```

The translation process in its entirety is shown step by step in Appendix A.2, page 137, and the result of the translation is shown in figure 6.5. The operator tree can be converted to the DAG seen in Figure 6.6.

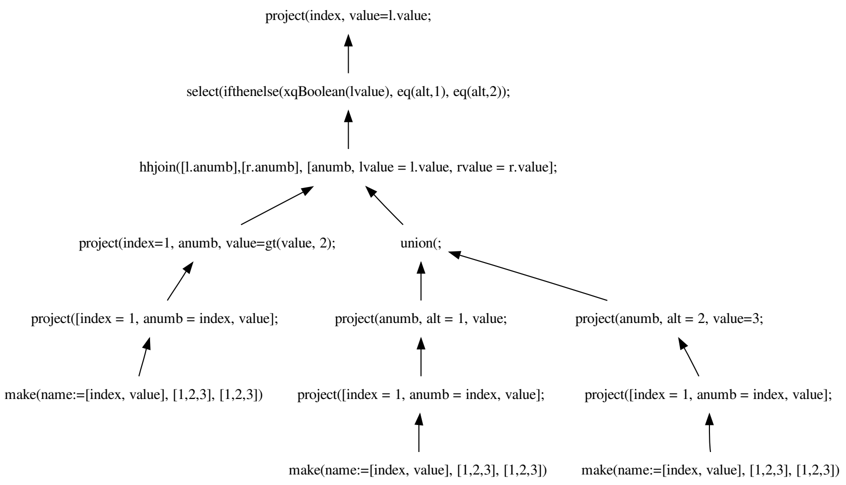

project(index, value=l.value;
select(ifthenelse(xqBoolean(lvalue), eq(alt,1), eq(alt,2));
hhjoin([l.anumb],[r.anumb], [anumb, lvalue = l.value, rvalue = r.value];
project(index=1, anumb, value=gt(value, 2);
union(;
project([index = 1, anumb = index, value];
project(anumb, alt = 1, value;
project(anumb, alt = 2, value=3;
make(name:=[index, value], [1,2,3], [1,2,3])
project([index = 1, anumb = index, value];
project([index = 1, anumb = index, value];
make(name:=[index, value], [1,2,3], [1,2,3])
make(name:=[index, value], [1,2,3], [1,2,3])

Figure 6.5: Complete translation of the conditional expression

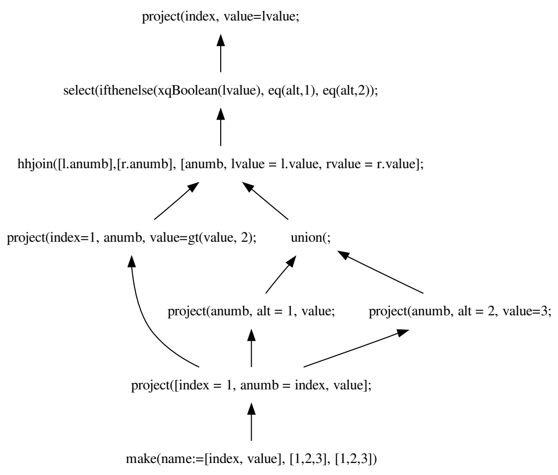

project(index, value=lvalue;
select(ifthenelse(xqBoolean(lvalue), eq(alt,1), eq(alt,2));
hhjoin([l.anumb],[r.anumb], [anumb, lvalue = l.value, rvalue = r.value];
project(index=1, anumb, value=gt(value, 2);
union(;
project(anumb, alt = 1, value;
project(anumb, alt = 2, value=3;
project([index = 1, anumb = index, value];
make(name:=[index, value], [1,2,3], [1,2,3])

Figure 6.6: DAG representation of the translated conditional expression

6.2 Algebra Generated By Implementation

In this section, a collection of queries are translated to relational algebra using the implemented proof of concept described in chapter 5. Naturally, this implementation also uses the "Tainting Dependencies" method, however the results from these translations will be used in a comparison with Pathfinder in the next section.

6.2.1 Trivial FLWOR

The following query:

```
for $a in (1,2,3) return $a
```

gives the operator tree in figure 6.7.

Result

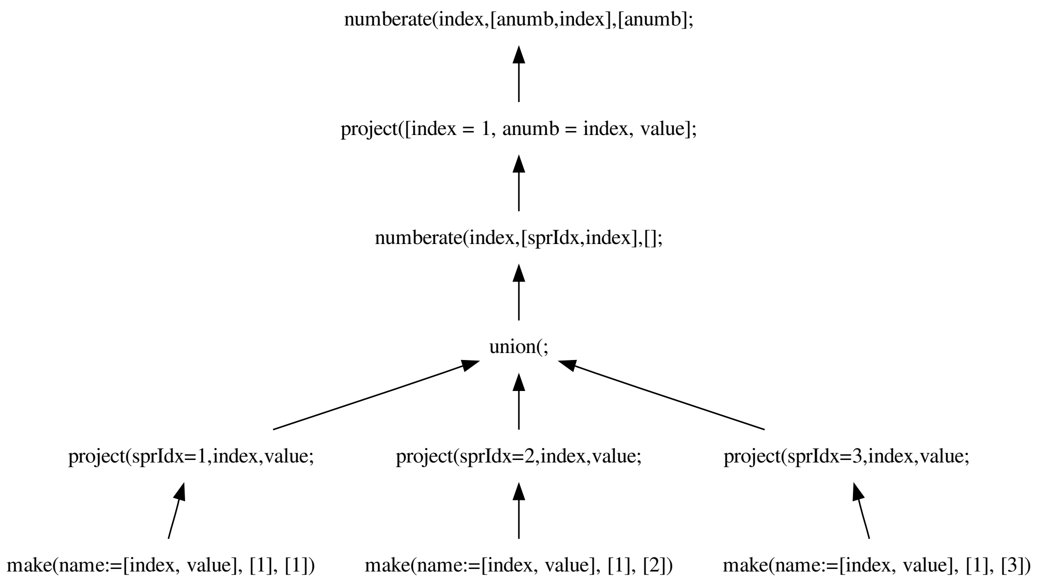

numberate(index,[anumb,index],[anumb];
project([index = 1, anumb = index, value];
numberate(index,[sprIdx,index],[];
union(;
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
project(sprIdx=3,index,value;
make(name:=[index, value], [1], [1])
make(name:=[index, value], [1], [2])
make(name:=[index, value], [1], [3])

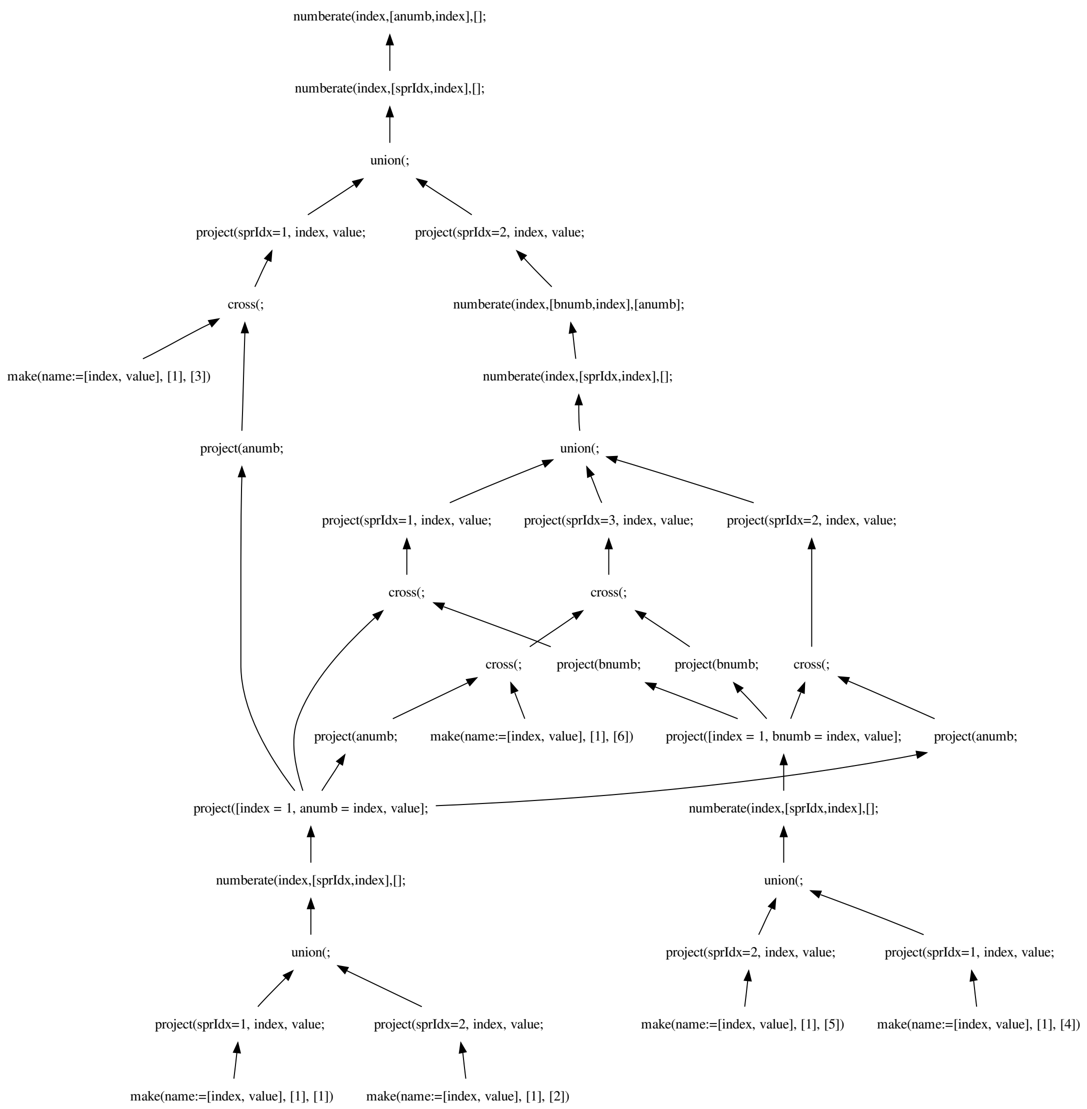
Figure 6.7: Complete translation of trivial FLWOR

6.2.2 Complex FLWOR

This query:

```
for $a in (1,2) return (3, for $b in (4,5) return ($a, $b, 6))
```

gives the operator tree of Figure 6.9, which can be converted into the corresponding DAG of figure 6.8.



numberate(index,[anumb,index],[];
numberate(index,[sprIdx,index],[];
union(;
project(sprIdx=1, index, value;
project(sprIdx=2, index, value;
cross(;
numberate(index,[bnumb,index],[anumb];
make(name:=[index, value], [1], [3])
numberate(index,[sprIdx,index],[];
project(anumb;
union(;
project(sprIdx=1, index, value;
project(sprIdx=3, index, value;
project(sprIdx=2, index, value;
cross(;
cross(;
cross(;
project(bnumb;
project(bnumb;
cross(;
project(anumb;
make(name:=[index, value], [1], [6])
project([index = 1, bnumb = index, value];
project(anumb;
project([index = 1, anumb = index, value];
numberate(index,[sprIdx,index],[];
numberate(index,[sprIdx,index],[];
union(;
union(;
project(sprIdx=2, index, value;
project(sprIdx=1, index, value;
project(sprIdx=1, index, value;
project(sprIdx=2, index, value;
make(name:=[index, value], [1], [5])
make(name:=[index, value], [1], [4])
make(name:=[index, value], [1], [1])
make(name:=[index, value], [1], [2])

Figure 6.8: The translated complex FLWOR query converted to a DAG

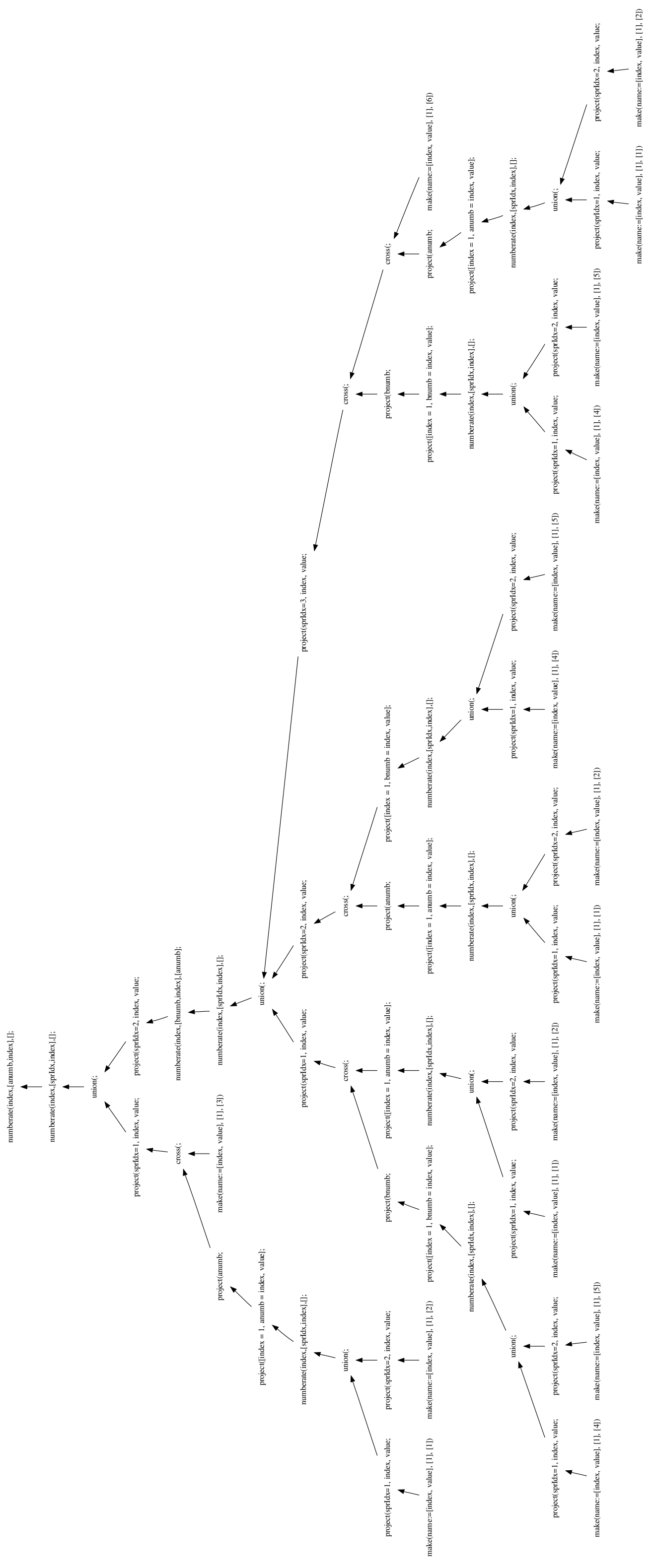


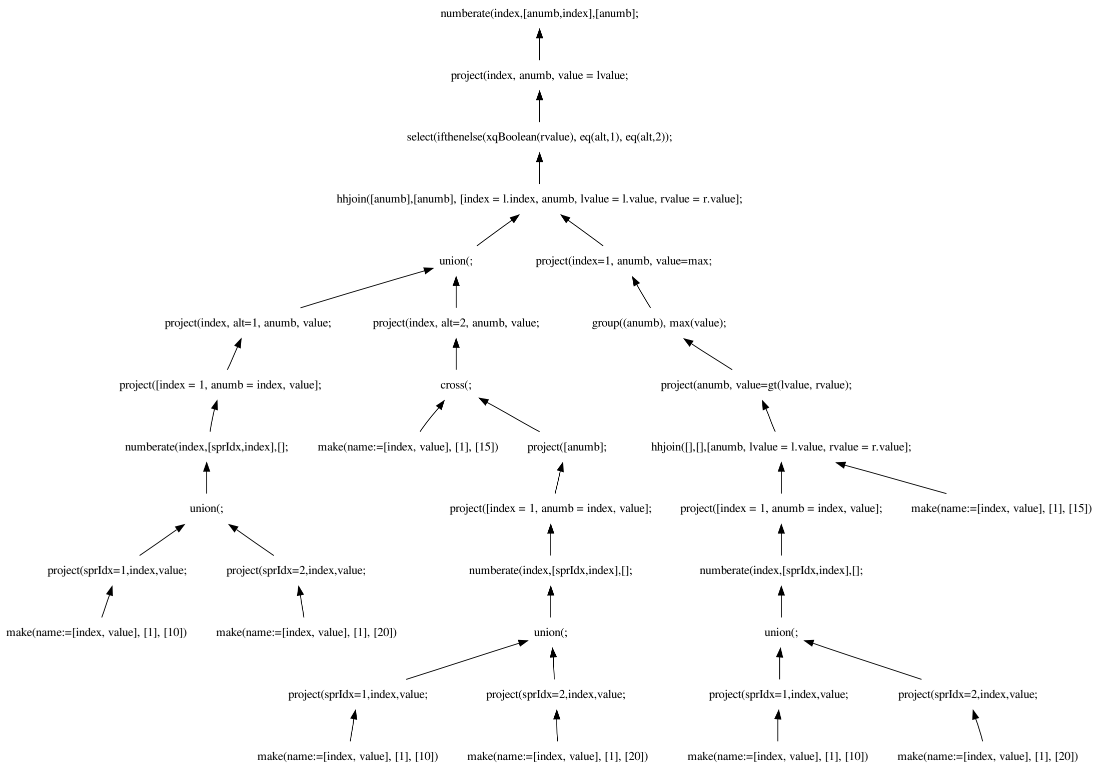
Figure 6.9: Complete translation of the complex FLWOR query

6.2.3 FLWOR with conditional

The following query:

```
for $a in (10,20) return if ($a > 15) then $a else 15
```

is translated into the operator tree of Figure 6.10. This can be converted to the corresponding DAG of Figure 6.11.



numberate(index,[anumb,index],[anumb];
project(index, anumb, value = lvalue;
select(ifthenelse(xqBoolean(rvalue), eq(alt,1), eq(alt,2));
hhjoin([anumb],[anumb], [index = l.index, anumb, lvalue = l.value, rvalue = r.value];
union(;
project(index=1, anumb, value=max;
project(index, alt=1, anumb, value;
project(index, alt=2, anumb, value;
group((anumb), max(value);
project([index = 1, anumb = index, value];
cross(;
project(anumb, value=gt(lvalue, rvalue);
numberate(index,[sprIdx,index],[];
make(name:=[index, value], [1], [15])
project([anumb];
hhjoin([],[],[anumb, lvalue = l.value, rvalue = r.value];
union(;
project([index = 1, anumb = index, value];
project([index = 1, anumb = index, value];
make(name:=[index, value], [1], [15])
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
numberate(index,[sprIdx,index],[];
numberate(index,[sprIdx,index],[];
make(name:=[index, value], [1], [10])
make(name:=[index, value], [1], [20])
union(;
union(;
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
make(name:=[index, value], [1], [10])
make(name:=[index, value], [1], [20])
make(name:=[index, value], [1], [10])
make(name:=[index, value], [1], [20])

Figure 6.10: Complete translation of FLWOR with conditional expression

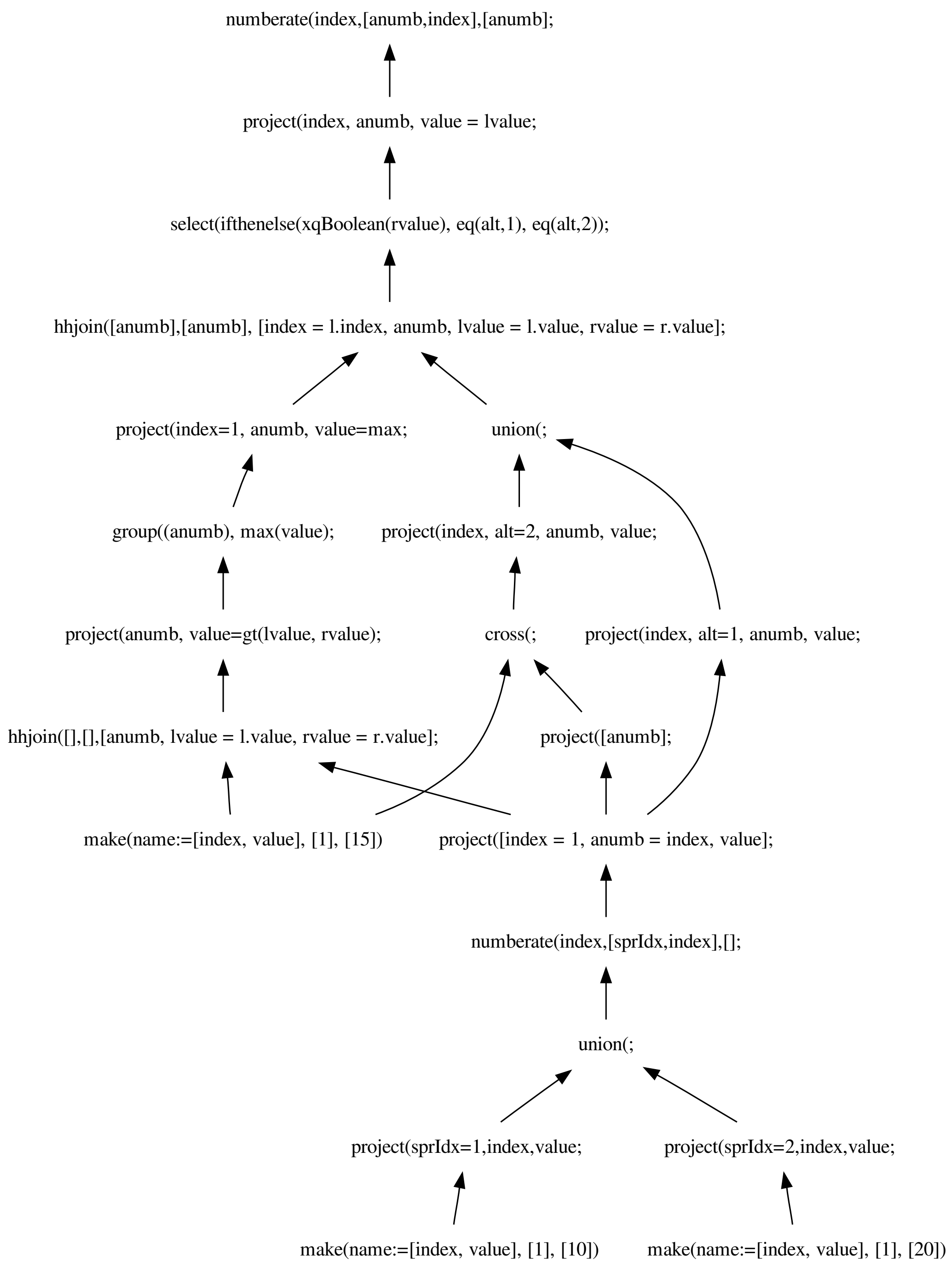

numberate(index,[anumb,index],[anumb];
project(index, anumb, value = lvalue;
select(ifthenelse(xqBoolean(rvalue), eq(alt,1), eq(alt,2));
hhjoin([anumb],[anumb], [index = l.index, anumb, lvalue = l.value, rvalue = r.value];
project(index=1, anumb, value=max;
union(;
group((anumb), max(value);
project(index, alt=2, anumb, value;
project(anumb, value=gt(lvalue, rvalue);
cross(;
project(index, alt=1, anumb, value;
hhjoin([],[],[anumb, lvalue = l.value, rvalue = r.value];
project([anumb];
make(name:=[index, value], [1], [15])
project([index = 1, anumb = index, value];
numberate(index,[sprIdx,index],[];
union(;
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
make(name:=[index, value], [1], [10])
make(name:=[index, value], [1], [20])

Figure 6.11: Translated FLWOR with condition converted to a DAG

6.3 Comparison

6.3.1 Assumptions

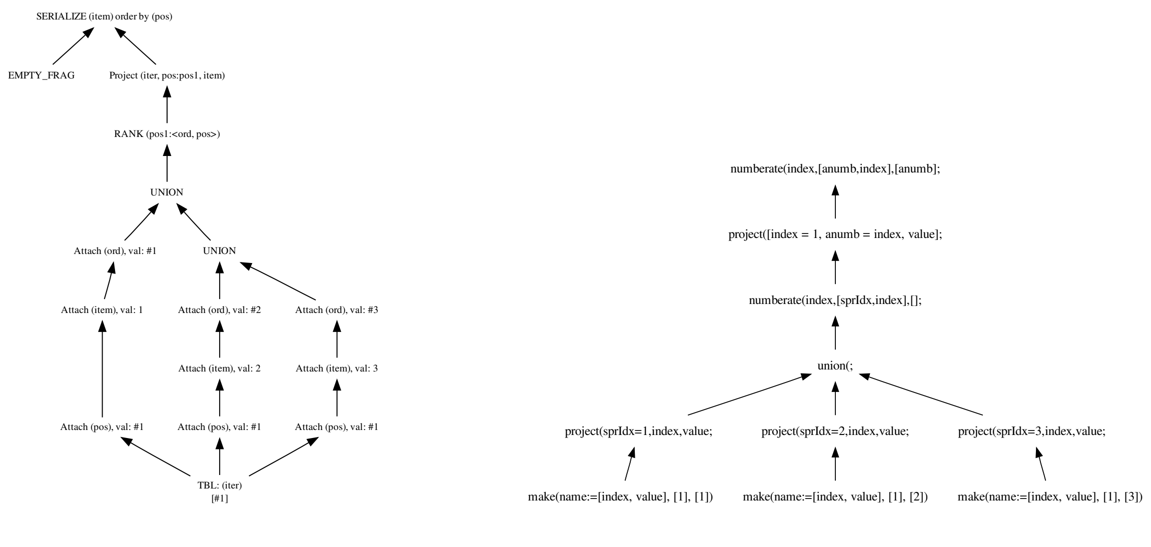
This comparison must be seen in the context of a number of assumptions about the systems being compared. With regards to fairness, it is important to note that the algebra trees generated by Pathfinder may have been simplified (to which the exact extent is not known), while the algebra trees generated by the prototype developed throughout this project *does not apply any simplifications or optimalisations* at all. The simplifications possibly applied by Pathfinder are noted in [26].

Some important effects on the Pathfinder algebra tree from these optimalisations are typically:

- The cartesian products between a loop relation and a constant subexpression are transformed into projections
- The custom operator attach is roughly a simpler equivalent to a project() where a new field with a constant field is added

6.3.2 DAG comparison

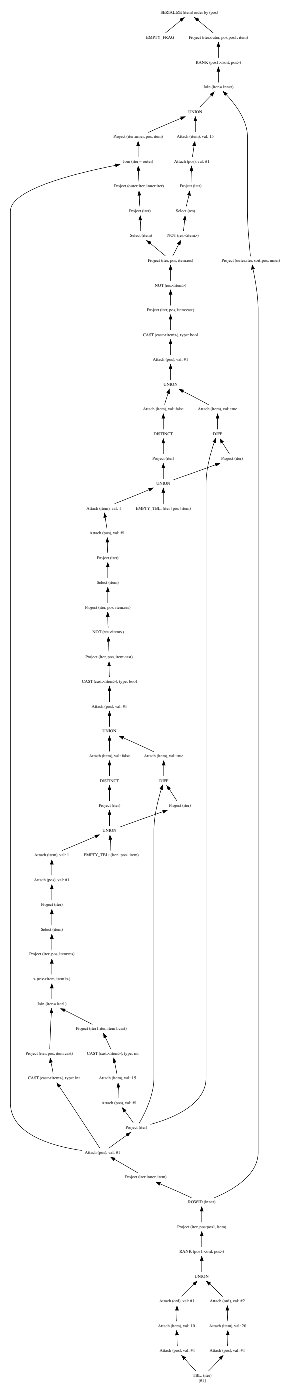
Note that the readability for these DAG comparisons are not essential – however, links to large-scale versions of these diagrams are noted in appendix B.

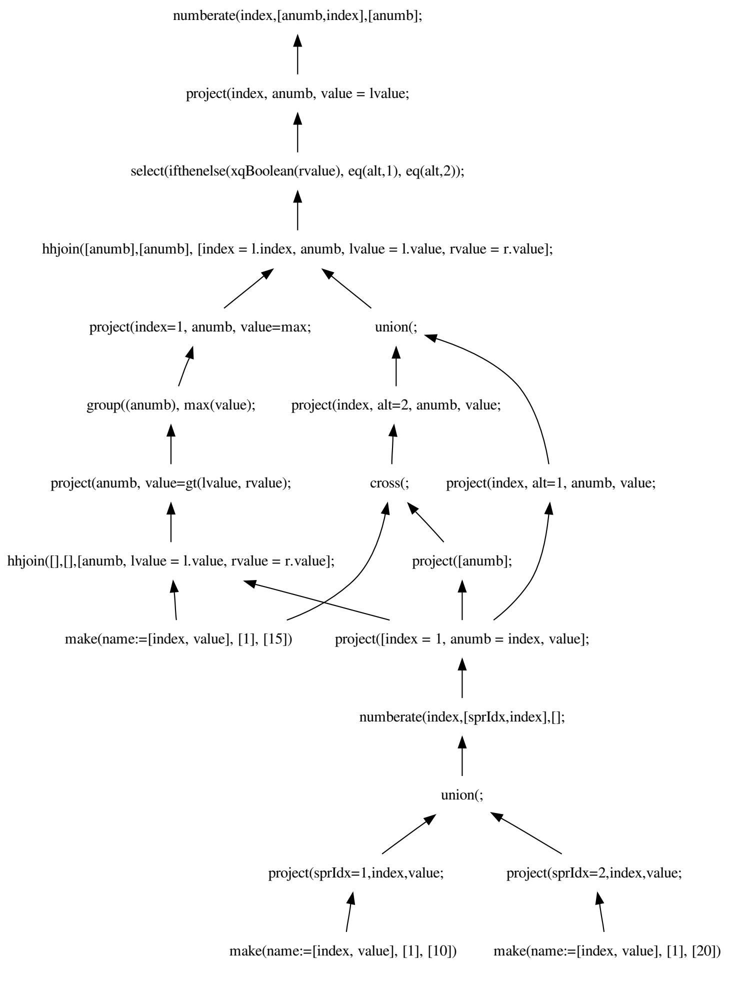

SERIALIZE (item) order by (pos)
EMPTY_FRAG
Project (iter, pos:pos1, item)
RANK (pos1:<ord, pos>)
UNION
Attach (ord), val: #1
UNION
Attach (item), val: 1
Attach (ord), val: #2
Attach (ord), val: #3
Attach (item), val: 2
Attach (item), val: 3
Attach (pos), val: #1
Attach (pos), val: #1
Attach (pos), val: #1
TBL: (iter)
[#1]
numberate(index,[anumb,index],[anumb];
project([index = 1, anumb = index, value];
numberate(index,[sprIdx,index],[];
union(;
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
project(sprIdx=3,index,value;
make(name:=[index, value], [1], [1])
make(name:=[index, value], [1], [2])
make(name:=[index, value], [1], [3])

(a) Pathfinder/MonetDB (b) Prototype implementation

Figure 6.12: Comparison of DAGs for the trivial expression in section 6.2.1

Figure 6.12 compares the DAGs generated for the trivial expression in section 6.2.1. As expected, both implementations produces relatively small algebra for this example.

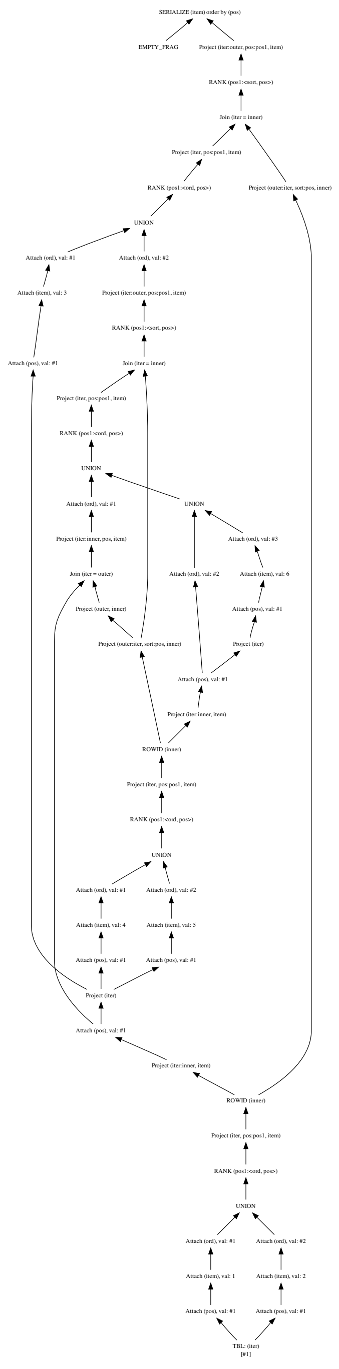

(a) Pathfinder/MonetDB


numberate(index,[anumb,index],[anumb];
project(index, anumb, value = lvalue;
select(ifthenelse(xqBoolean(rvalue), eq(alt,1), eq(alt,2));
hhjoin([anumb],[anumb], [index = l.index, anumb, lvalue = l.value, rvalue = r.value];
project(index=1, anumb, value=max;
union(;
group((anumb), max(value);
project(index, alt=2, anumb, value;
project(anumb, value=gt(lvalue, rvalue);
cross(;
project(index, alt=1, anumb, value;
hhjoin([],[],[anumb, lvalue = l.value, rvalue = r.value];
project([anumb];
make(name:=[index, value], [1], [15])
project([index = 1, anumb = index, value];
numberate(index,[sprIdx,index],[];
union(;
project(sprIdx=1,index,value;
project(sprIdx=2,index,value;
make(name:=[index, value], [1], [10])
make(name:=[index, value], [1], [20])

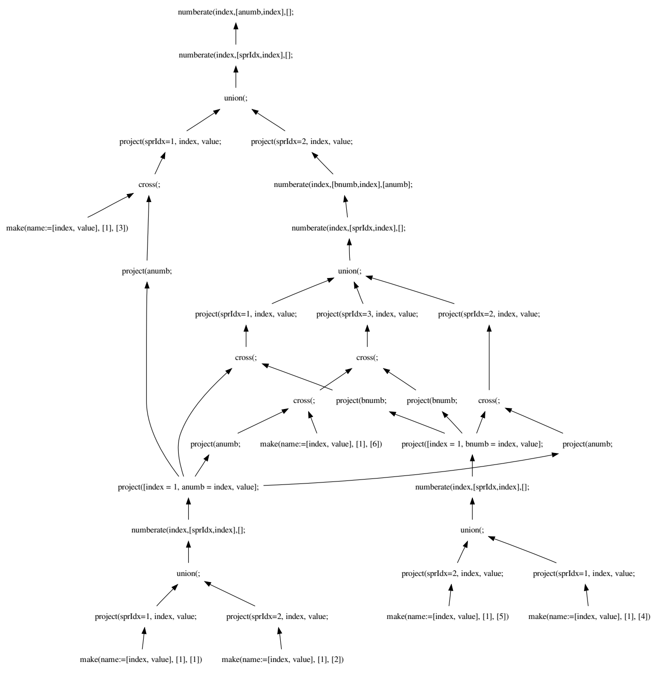
(b) Prototype implementation

Figure 6.13: Comparison of DAGs for the conditional expression in section 6.2.3

Figure 6.13 compares the DAGs generated for the conditional expression in section 6.2.3. The immediate impression may be that the algebra generated by Pathfinder is substantially more complex, however this stems partly from the numerous Attach operators. Also note that Pathfinders algebra contains three joins, whereas the algebra generated by our prototype only contains two joins.



(a) Pathfinder/MonetDB


numberate(index,[anumb,index],[];
numberate(index,[sprIdx,index],[];
union(;
project(sprIdx=1, index, value;
project(sprIdx=2, index, value;
cross(;
numberate(index,[bnumb,index],[anumb];
make(name:=[index, value], [1], [3])
numberate(index,[sprIdx,index],[];
project(anumb;
union(;
project(sprIdx=1, index, value;
project(sprIdx=3, index, value;
project(sprIdx=2, index, value;
cross(;
cross(;
cross(;
project(bnumb;
project(bnumb;
cross(;
project(anumb;
make(name:=[index, value], [1], [6])
project([index = 1, bnumb = index, value];
project(anumb;
project([index = 1, anumb = index, value];
numberate(index,[sprIdx,index],[];
numberate(index,[sprIdx,index],[];
union(;
union(;
project(sprIdx=2, index, value;
project(sprIdx=1, index, value;
project(sprIdx=1, index, value;
project(sprIdx=2, index, value;
make(name:=[index, value], [1], [5])
make(name:=[index, value], [1], [4])
make(name:=[index, value], [1], [1])
make(name:=[index, value], [1], [2])

(b) Prototype implementation

Figure 6.14: Comparison of DAGs for the complex expression in section 6.2.2

Figure 6.14 compares the DAGs generated for the complex expression in section 6.2.2. Again it may seem that Pathfinders algebra is substantially more complex, however this fact is, as in the previous comparison, partly magnified due to the numerous Attach operators.

6.3.3 Complexity estimation and comparison

Complexity estimation is performed as detailed in section 3.5. The comparison matrix shown in table 6.1 details the differences in *complexity*. This result is also charted in figures 6.15 and 6.16.

Refer to Section 3.5 on page 39 for a detailed account of how these complexity estimations are made.

	Pathfinder		**Prototype**	
	Tuples	Fields	Tuples	Fields
Trivial	16	16	15	18
Complex	215	265	136	102
Conditional	94	50	31	44

Table 6.1: Complexity comparison matrix

For the trivial example, the generated algebra seems to be close in complexity. However, for the more extensive examples, the complexity of the algebra generated by Pathfinder seems substantially larger. For example, consider the "complex" example where Pathfinder seems to produce 58% more tuples and almost 160% more fields.

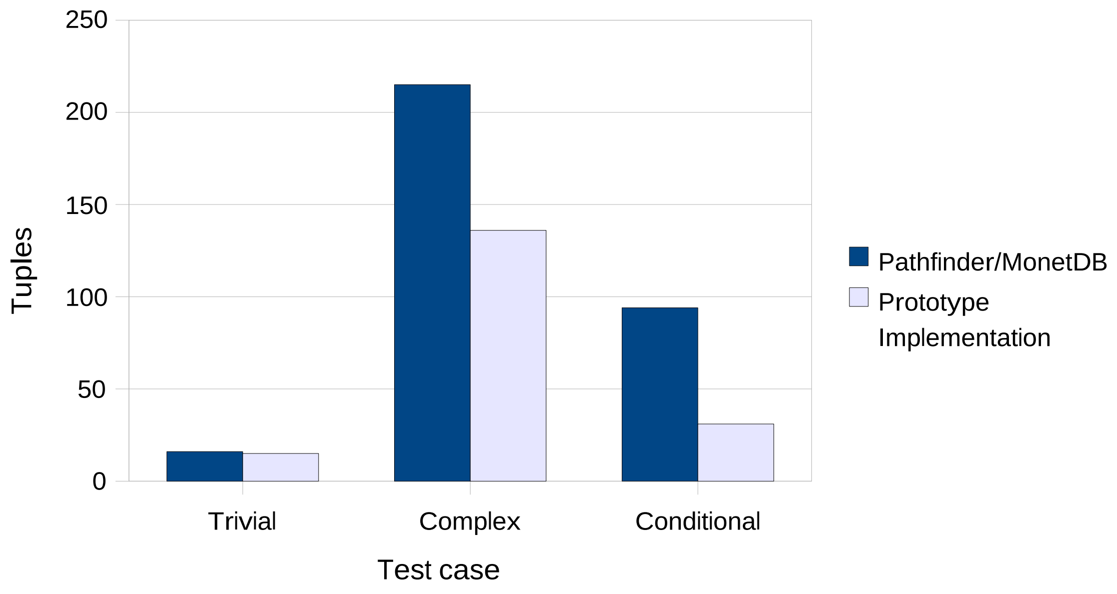


Figure 6.15: Comparison of complexity based on tuple creation

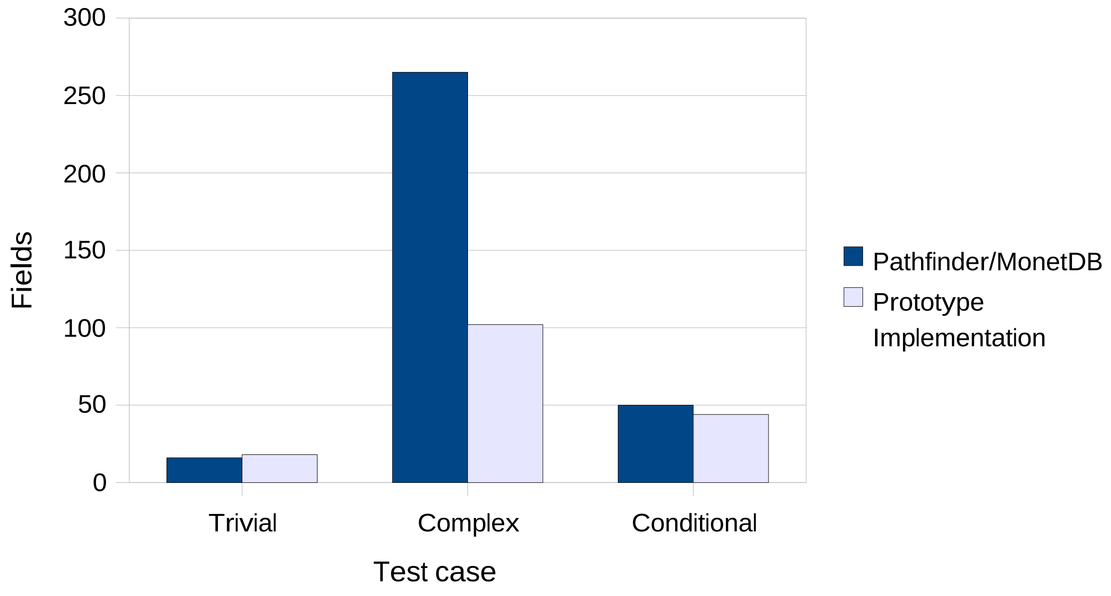


Figure 6.16: Comparison of complexity based on field creation

The *tuple I/O complexity* differences for join and sort operators are shown in tables 6.2 and 6.2.

	Pathfinder						
		In			**Out**		
	#	Min	Max	Avg	Min	Max	Avg
Trivial	0	0	0	0	0	0	0
Complex	3	6	14	12.67	4	14	10
Conditional	3	4	6	4.67	1	2	1.67
	Prototype						
		In			**Out**		
	#	Min	Max	Avg	Min	Max	Avg
Trivial	0	0	0	0	0	0	0
Complex	0	0	0	0	0	0	0
Conditional	2	3	6	4.5	3	6	4.5

Table 6.2: Tuple input/output in join operators

Considering Table 6.2, it seems that for the "complex" example, the prototype implementation does not produce any join operators at all, while Pathfinder produces three join operators. Also note that for the same example, Pathfinder has an average of 12.67 input tuples.

Table 6.3 shows that in the case of sort operators, Pathfinder and the prototype are more equal in complexity

	Pathfinder						
		In			**Out**		
	#	Min	Max	Avg	Min	Max	Avg
Trivial	1	3	3	3	3	3	3
Complex	6	2	14	8.8	2	14	8.8
Conditional	2	2	2	2	2	2	2
	Prototype						
		In			**Out**		
	#	Min	Max	Avg	Min	Max	Avg
Trivial	2	3	3	3	3	3	3
Complex	6	2	14	9.33	2	14	9.33
Conditional	2	2	2	2	2	2	2

Table 6.3: Tuple input/output in sort operators

6.4 Summary

In this chapter, a series of algebra trees have been presented. Specifically, algebra computed by hand as well as algebra generated by the prototype implementation has been presented.

In the next chapter, these results as well as challenges and problems are discussed in detail.

Chapter 7

Discussion

In this chapter, some central aspects of this project will be discussed. Some of them are concrete problems or challenges, others are of a more predictive nature and may serve as a basis for further research. Also, possible solutions or ideas are proposed where applicable.

Most importantly, XQuery features not currently supported by Tainting Dependencies (TD) are discussed. Challenges related to XQuery sequences are presented, and some possible solutions are proposed. The external environment and communication with it poses some problems which are reviewed, and the lack of support for a type system in TD is elaborated upon as well. Furthermore, we discuss some optimisations and how they may be implemented. The prototype implementation and the usage of syntax tree normalisation is discussed. Finally, we briefly analyse the results presented in the previous chapter, and insecurities as well as possible sources of errors are accounted for.

7.1 XQuery features not supported

In this section we will present some of the features of XQuery at the time not supported by Tainting Dependencies, and some ideas around how a possible solution may be realised. Some of the features are not supported because they involve types, which is discussed in Section 7.4.

7.1.1 Full-text extensions

XQuery Full Text[34] is a superset of XQuery. A quick overview of the extensions made to XQuery can be found in Section 2.1.5. Tainting Dependencies does however at this time support any of these features. One of the most important expressions of the extension is the `ftcontains` expression. An excerpt of the EBNF specification of this expression and some of its subexpressions can be seen in Figure 7.1. The `FTPrimaryWithOptions` production is a straight descendant of `FTMildNot`, and `FTWordsValue` is a straight descendant of `FTPrimary`. The complete EBNF specification can be found in Appendix D.

A simple `ftcontains` expression checking if a node contains a literal may be quite simple to translate. This can be done by looking up the literal and joining the result with the node on

```
 [51] FTContainsExpr ::= RangeExpr ( "ftcontains" FTSelection FTIgnoreOption? )?
[144] FTSelection    ::= FTOr FTPosFilter* ("weight" RangeExpr)?
[145] FTOr           ::= FTAnd ( "ftor" FTAnd )*
[146] FTAnd          ::= FTMildNot ( "ftand" FTMildNot )*
...
[149] FTPrimaryWithOptions ::= FTPrimary FTMatchOptions?
[166] FTMatchOption  ::= FTLanguageOption
                       | FTWildCardOption
                       | FTThesaurusOption
                       ...
[152] FTWordsValue   ::= Literal | ("{" Expr "}")
```

Figure 7.1: Excerpt of W3C EBNF full text specification[34]

their *scope* attributes. Something like this:

$$\frac{}{e\ \texttt{ftcontains}\ literal} \rightsquigarrow \begin{array}{l} \ldots \\ \quad \mathsf{hhjoin}([\mathsf{l.scope}],[\mathsf{r.scope}],\ldots; \\ \qquad \mathbf{r}(e); \\ \qquad \ldots \\ \qquad\quad \mathsf{lookup}(literal)) \end{array}$$

`ftand` and `ftor` expressions may extend upon this solution. These operators makes it possible to check for more than one term per node. If the translator keeps track of which scope is the current scope according to a path expression, as discussed in Section 7.5.2, this can be translated quite nicely into MQL. MQL supports two operators and and or which when surrounded by a scope operator will require the results from the two operands to stem from the same scope. A simple `ftand` expression may therefore be translated something like this:

$$\frac{}{e\ \texttt{ftcontains}\ literal_1\ \texttt{ftand}\ literal_2} \rightsquigarrow \begin{array}{l} \ldots \\ \quad \mathsf{hhjoin}([\mathsf{l.scope}],[\mathsf{r.scope}],\ldots; \\ \qquad \mathbf{r}(e); \\ \qquad \ldots \\ \qquad\quad \mathsf{scope}(e.scope; \\ \qquad\qquad \mathsf{and}(\\ \qquad\qquad\quad \mathsf{lookup}(literal_1); \\ \qquad\qquad\quad \mathsf{lookup}(literal_2) \end{array}$$

A problem do however arise when the operands of `ftcontains` are not a node and a literal. As can be seen from the specification in Figure 7.1, a general expression may also be an operand. This is no simple task to accomodate for. One possible solution would be to let the lookup operator take a relation as input. There will also be problems if the first operand is not a node. If this is the case, there is no *scope* attribute to join on. An example of such a query might be:

```
"a man and his dog" ftcontains "dog"
```

Here, the MQL processor would have to split up and search through the first operand for any matches with the second.

`FTMatchOption` contains a great deal of options which modify in the way two terms or phrases are matched. The options are specified like e.g. `with stemming` and `with thesaurus`. One possible solution for accomodating for such options would be a context operator comparable to the index operator (Section 3.4.3). The operator would take the match options as parameters, and set the context for possible lookup operators within its subtree. Another possibility would be to use the options as parameters directly to the lookup operator.

7.1.2 Ordering mode

XQuery contains `ordered` and `unordered` expressions. The purpose of these expressions is to set the ordering mode in to `ordered` or `unordered` for a certain region in a query. The expressions set up an environment enclosed by curly braces in which the specific ordering mode applies. The default ordering mode is `ordered`. A performance advantage may be realised by setting the ordering mode to `unordered` for expressions where the ordering of the result is not significant. The system will then be granted the flexibility to return the result in the order it finds most efficient.

The *index* attribute and the numberate operator are responsible for ensuring correct order in Tainting Dependencies. One of the problems with MarkXRemove was that it did not consider the ordering of items, while one of its advantages was its simplicity. By introducing the concept of tainting to MarkXRemove, this would probably be a good start for finding a method of translating in `unordered` mode. But as TD is an evolution of MarkXRemove and a more complete method, a better solution might be to simplify this method by removing numberate operators and all operators whose only intent is to manipulate *index* fields. Utilising context sensitive visitor patterns (Section 2.4.4), differentiating translation of `ordered` and `unordered` mode expressions will be made easy.

7.1.3 Binary expressions

In this section we will present some of the XQuery binary operator expressions not handled in section 4.6, and some ideas for possible translations of them.

Node comparison operators

The node comparison operators of XQuery are `is`, « and », and are currently not handled by Tainting Dependencies. A comparison with the `is` operator yields *true* if the two operand nodes have the same node identity. Where node identity is defined by W3C like this [32]:

> Each node has a unique identity. Every node in an instance of the data model is unique: identical to itself, and not identical to any other node.

One solution would be to translate such expressions into checking if the two nodes have the same value in their respective *scope* and *documentId* fields:

$$\frac{}{e_1 \texttt{ is } e_2} \rightsquigarrow \begin{array}{l} \ldots \\ \quad \mathsf{project(value = and(eq(l.docId,r.docId),\ eq(l.scope,\ r.scope))\ldots;} \\ \quad\quad \mathsf{hhjoin}([(e_1.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_1.\vartheta)], \ldots; \\ \quad\quad\quad \mathbf{r}(e_1) \\ \quad\quad\quad \mathbf{r}(e_2) \end{array}$$

This holds true as *documentId* is unique per document, and no two nodes may be in the exactly same position within one document.

A comparison with the « and » operators returns *true* if, in document order, the left operand node precedes the right operand node and if the left operand node follows the right operand node, respectively. Otherwise it returns *false*. These operators have semantics comparable with the `preceding` and `following` axes in path expressions, and may therefore be translated by utilising the isPreceding and isFollowing functions (Section 3.4.4, page 37):

$$\frac{}{e_1 \texttt{ COMP } e_2} \rightsquigarrow \begin{array}{l} \ldots \\ \quad \mathsf{project(value = isFUNK(l.scope,\ r.scope),\ldots;} \\ \quad\quad \mathsf{hhjoin}([(e_1.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_1.\vartheta)], \ldots; \\ \quad\quad\quad \mathbf{r}(e_1) \\ \quad\quad\quad \mathbf{r}(e_2) \end{array}$$

When the comparison operator `COMP` is « the MQL function isFUNK must be isPreceding, and when the operator is » the function must be isFollowing.

Combining node sequences

XQuery provides the following operators for combining sequences of nodes:

- `union` and | (read as "or") which are equivalent. They take two node sequences as operands and return a sequence containing all the nodes that occur in either of the operands.
- `intersect` takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands.
- `except` 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 operand.

As all these operators eliminate duplicate nodes from their result sequences (based on node identity), the operators are the XQuery equivalent to the relational algebra operators union, intersect and difference discussed in Section 2.3 on page 13. The result is also required to be in document order.

MQL does currently not implement a pure distinct operator, but if it did it would need partition functionality similar to that of the numberate operator, and a translation of `union` and `|` may have looked something like this:

$$\frac{}{e_1 \ \texttt{union} \ e_2} \rightsquigarrow \begin{array}{l} \mathsf{numberate(index,\ [documentId,\ scope],}\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad \mathsf{distinct([documentId,\ scope],}\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad\quad \mathsf{union(} \\ \quad\quad\quad \mathbf{r}(e_1); \\ \quad\quad\quad \mathbf{r}(e_2)))) \end{array}$$

Where the first parameter list of distinct is the field combinations which need to be distinct, and the second list is the fields to partition on. This solution also requires the possibility sort on *scope* fields.

The `intersect` operator may be implemented as a join on their node identity, that is, *documentId* and *scope*. The result will have to be run through a distinct operator like in the case of the `union` expression, in case any of the operand sequences contains duplicates. A solution might look something like:

$$\frac{}{e_1 \ \texttt{intersect} \ e_2} \rightsquigarrow \begin{array}{l} \mathsf{numberate(index,\ [documentId,\ scope],}\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad \mathsf{distinct([documentId,\ scope],}\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad\quad \mathsf{hhjoin([docId,scope,}(e_1.\vartheta \cap e_2.\vartheta)\mathsf{],\ [docId,scope,}(e_2.\vartheta \cap e_1.\vartheta)\mathsf{],}\ldots; \\ \quad\quad\quad \mathbf{r}(e_1); \\ \quad\quad\quad \mathbf{r}(e_2)))) \end{array}$$

The translation of the `except` operator may have been solved with the help of a anti-join (section 2.3.2 on page 17). But as MQL does not implement a anti-join operator, the same effect may be achieved by using a left-outer-join followed by a selection. A distinct operator is needed here aswell, as the left operand may contain duplicates:

$$\frac{}{e_1 \ \texttt{intersect} \ e_2} \rightsquigarrow \begin{array}{l} \mathsf{numberate(index,\ [documentId,\ scope],}\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad \ldots \\ \quad \mathsf{distinct([documentId,\ scope],}\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad\quad \mathsf{select(eq(r.value,\ NULL);} \\ \quad\quad\quad \mathsf{hhjoin([docId,scope,}(e_1.\vartheta \cap e_2.\vartheta)\mathsf{],\ [docId,scope,}(e_2.\vartheta \cap e_1.\vartheta)\mathsf{],} \\ \quad\quad\quad\quad\quad \ldots\mathsf{,\ left;} \\ \quad\quad\quad\quad \mathbf{r}(e_1); \\ \quad\quad\quad\quad \mathbf{r}(e_2))))) \end{array}$$

As can be seen, all these proposals involve multiple resource-expensive operators such as numberate and distinct. So a better solution would be preferred, or at least to minimise the use of the sequence combination operators when forming XQuery queries.

Range expressions

Range expressions are in the format e_1 `to` e_2 and can be used to construct a sequence of consecutive integers. Both operands must be integers or castable to `xs:integer`. If a operand is an empty sequence, or the integer value of first operand is greater than the integer value of the second, the result is an empty sequence. Otherwise, the result is a sequence containing the two integers and every integer between the two operands, in increasing order.

If the operands are literals, the translator can turn the expression into a sequence construction expression containing the needed literals. However, if the value of the operands is not known before runtime, such expressions will have to be handled by the MQL processor. The least unnatural solution to this would be to implement a MQL operator taking one relation as input, two attribute names to create a sequence from and to, and a list of attributes names for fields that must be a part of the result. The operator would then create tuples based on the value of the fields specified as the from and to attributes, and augment each produced tuple with a copy of the value of the fields specified to be part of the result. This would look something like:

$$\frac{}{e_1\ \texttt{to}\ e_2} \leadsto \begin{array}{l} \ldots \\ \mathsf{rangeExpr}([\mathsf{l.value},\ \mathsf{r.value}],\ [e_1.\vartheta \cup e_2.\vartheta]; \\ \quad \mathsf{hhjoin}([(e_1.\vartheta \cap e_2.\vartheta)],\ [(e_2.\vartheta \cap e_1.\vartheta)], \ldots; \\ \quad\quad \mathbf{r}(e_1); \\ \quad\quad \mathbf{r}(e_2)))) \end{array}$$

The rangeExpr operator would be a very specialised operator, and probably quite complex to implement. If no better translation option is found, a better solution would be to require range expressions to have literals as operands.

7.1.4 Order by

As can be seen from the `order by`-clause specification of Figure 4.5.3 on page 57, the ordering of tuples returned from a FLWOR expression is very flexible as it may be set by one or more ordering specifications. Options may also be set for each order specification, called order modifiers. Currently, Tainting Dependencies only accomodates for a single order specification and no order modifiers. Expanding the translation rule for `order by` ordering (Rule 4.11) to allow multiple ordering specification may be done by sequentially joining the specifiers on their common dependencies before joining the result of this with the relation stemming from the `where` or `return`-clause, and finally renumbering while sorting on the values from the order specifier in the correct order:

$$\frac{}{\texttt{order by}\ e_{3\cdot 1}, \ldots, e_{3\cdot n}} \leadsto \begin{array}{l} \mathsf{project}(\mathsf{value} = \mathsf{l.value},\ \vartheta; \\ \quad \mathsf{numberate}(\mathsf{index},\ [\mathsf{value1}, \ldots, \mathsf{value}n],\ [\vartheta]; \\ \quad\quad \mathsf{hhjoin}(\ldots [\mathsf{l.value}, \mathsf{r.value1}, \ldots, \mathsf{r.value}n, \ldots]; \\ \quad\quad\quad \mathbf{t}(\mathbf{r}(e_C),\ \beta); \\ \quad\quad\quad\quad \mathsf{hhjoin}(\ldots [\mathsf{value1} = \mathsf{l.value}, \ldots, \mathsf{r.value}n, \ldots]; \\ \quad\quad\quad\quad\quad \mathbf{r}(e_{3\cdot 1})))) \\ \quad\quad\quad\quad\quad \mathsf{hhjoin}(\ldots \\ \quad\quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad\quad\quad \mathsf{hhjoin}(\ldots, \mathsf{value}n = \mathsf{r.value} \ldots; \\ \quad\quad\quad\quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad\quad\quad\quad \mathbf{r}(e_{3\cdot n})))) \end{array}$$

Where $\vartheta = (e_C.\vartheta \cup e_{3\cdot 1}.\vartheta \cup \ldots \cup e_{3\cdot n}.\vartheta) - \beta$. If the `order by` clause is defined as `stable`, $l.index$ will have to be added as a last attribute to sort on in the numberate operator.

The best way to implement the order modifiers would probably be to allow some corresponding parameters to be specified in the numberate operator. It may be cumbersome to implement such an operator allowing modifiers to be specified for each of the attributes to sort on. In such a case, each of the order specifier relations may have their *value* fields sorted and updated by their own numberate operator.

7.1.5 XQuery functions

XQuery supports numerous built-in functions, such as `fn:not` and `fn:count`, all specified in [35]. These functions are identified by being a member of the `fn`-namespace. In addition, XQuery allows users to declare functions of their own. A function declaration specifies the name of the function, the names and datatypes of the parameters, and the datatype of the result, as can be seen from the specification in Figure 7.2.

```
[26] FunctionDecl ::= "declare" "function" QName "(" ParamList? ")"
                        ("as" SequenceType)? (EnclosedExpr | "external")
[27] ParamList    ::= Param ("," Param)*
[28] Param        ::= "$" QName TypeDeclaration?
```

Figure 7.2: W3C XQuery function declaration specification[36]

The `external` keyword means that the function is implemented outside the query environment. This is discussed in Section 7.3. To support built-in and query declared functions with TD, a function table may be introduced. By querying this table with the name of the function, a operator tree is returned with pointers to possible references of the parameters within the tree. When the translator comes upon a function call, it fetches this tree and follows the pointers to the parameter references and inserts the corresponding algebra tree. The built-in functions will have to be hardcoded and reside within the function table at startup.

7.2 XQuery sequences

There is no distinction between an item, that is, a node or an atomic value, and a singleton sequence containing that item in XQuery. An item is equivalent to a singleton sequence containing that item and vice versa. A sequence may contain nodes, atomic values, or any mixture of nodes and atomic values. But it may be advantageous for a translator to differentiate singleton sequences from other sequences.

As we saw in Section 4.10, by knowing that all subexpressions return singleton sequences, the translation of the sequence construction expression may be simplified. If the `return`-clause expression is a singleton sequence the translation of iterator ordered FLWOR expressions may also be simplified. If the FLWOR only contains only one iterator and no `where`-clause the renumbering can be replaced by a renaming of the $-numb$ field corresponding to the iterator to *index*. Understanding that this works can be done by considering the rule for translation of iterator ordering (rule 4.9). As the `return`-clause is a singleton, the *index* fields will have the constant value 1. β will contain only one $-numb$ attribute, holding information of which iteration the value in *value* occurs. The iteration number will then become the *index* field of the sequence created by the FLWOR.

Some expressions, such as arithmetic expressions, `order by`-expressions and value comparisons, require their subexpressions or operands to be singleton sequences. This means that a query such as `(1, 2) + 3` will raise a type error. By having the knowledge of the cardinality of the sequence returned to such an expression, the translator may raise the error, and avoid a faulty query being run on the MQL processor. Evaluating the cardinality of seqences returned from expressions is in many cases a simple task. Some expressions will always return singletons, such as logical, comparison and aritmetic expressions, iterator variable references and literals. The cardinality of sequences constructed of such expressions may also be calculated in the translator. A problem arises, however, when dynamic content (not from the query itself) is included. Consider the following query:

```
for $a in doc("people.xml")//person
order by $a/surname
return $a
```

As previously stated, the `order by`-expression only accepts singletons. If the document contains a `person` node containing two `surename` nodes the query should fail. The translator does however not have the ability to evaluate if a type error should be raised or not. The query stated will, without a check for multiple `surname` node per `person`, result in a sequence where the `person` nodes containing more than one `surname` node will occur more than one time.

The problem lies in the fact that the query is not a erroneous MQL query, but a erroneous XQuery query. One solution would be to implement a check in MQL, which would inform the MQL processor of any potential error. This can be done e.g. by a MQL function raiseError(), which would abort the evaluation of the query and e.g. throw an exception. All items in the relational representation of sequences in Tainting Dependencies are marked with their position within the sequence with the *index* field. One way to check if an expression e does not return a singleton sequence can be the following:

```
select(ifthenelse(eq(index, 1), true, raiseError());
  r(e))
```

The check may of course be omitted if the translator is sure, by the means discussed earlier, e will return a singleton.

7.2.1 Effective boolean value

Within certain circumstances it is necessary to find the effective boolean value of a sequence. The effective boolean value is by W3C defined as follows[36]:

> The effective boolean value of a value is defined as the result of applying the `fn:boolean` function to the value.

Where the function is declared as `fn:boolean($arg as item()*) as xs:boolean` and the dynamic semantics are defined as follows[35]:

1. If `$arg` is the empty sequence, `fn:boolean` returns *false*.

2. If `$arg` is a sequence whose first item is a node, `fn:boolean` returns *true*.
3. If `$arg` is a singleton value of type `xs:boolean` or a derived from `xs:boolean`, `fn:boolean` returns $arg.
4. If `$arg` is a singleton value of type `xs:string` or a type derived from `xs:string`, `xs:anyURI` or a type derived from `xs:anyURI` or `xs:untypedAtomic`, `fn:boolean` returns *false* if the operand value has zero length; otherwise it returns *true*.
5. If `$arg` is a singleton value of any numeric type or a type derived from a numeric type, `fn:boolean` returns *false* if the operand value is *NaN* or is numerically equal to zero; otherwise it returns *true*.
6. In all other cases, `fn:boolean` raises a type error.

In the chapter presenting Tainting Dependencies, Chapter 4, we solved resolving the effective boolean value of sequences with the help of an assumed MQL function xqBoolean() and an undefined abstract function $\mathbf{B}()$. In that chapter we said it was enough to consider it as a grouping function returning one tuple per unique iteration (as implied by the iterator dependencies). The points 3-5 of the definition of the effective boolean value are all handled by xqBoolean() without problems, as they all concern singleton sequences. Points 2 and 6 together, however, creates a interesting situation. They imply that when finding the effective boolean value of all possible XQuery sequences, all should cause an error, except the sequences where a node is the first element, regardless of the other items in the sequence.

This means that a simple check as proposed for evaluating the cardinality of operands for e.g. arithmetic expressions earlier in this section will not do. A possible solution for the $\mathbf{B}()$ function would be to run a group operator on the relation, grouping on the unique iterations, and run a count and some aggregator functions selecting the value and type of the first item of each group. Further, the result will then have to be checked similar to the check proposed earlier.

```
select(or(eq(type, node), ifthenelse(eq(count,1), true, raiseError());
  group(($e.\vartheta$), count(), selProj(eq(index,1), value), selProj(eq(index,1), type);
    $\mathbf{r}(e)$))
```

Where selProj($pred$, $field$) is a kind of selection and projection hybrid, selecting field $field$ if predicate $pred$ is *true*. Such a aggregator function may however seem strange, and there is nothing similar implemented in the MQL processor.

An alternative can be to utilise the groupby operator. The semantics of this operator is the same as for the group operator, except that the input relation is unchanged and returned as the output. Further, the result of the grouping is returned as a separate, named relation ("result set" in MQL lingo). The resultset operator with the name as its parameter will fetch this named result:

```
select(and(eq(index,1), or(eq(type,node), ifthenelse(eq(count,1), true, raiseError()))));
  hhjoin([$e.\vartheta$], [$e.\vartheta$], [l.value, r.count];
    groupby(countrelation, ($e.\vartheta$), count();
      $\mathbf{r}(e)$);
    resultset(countrelation)))
```

Both solutions assume a type system where a field *type* holds the type of the item represented in the tuple. The last proposal of a implementation of the $\mathbf{B}()$ function only uses operators and functions already implemented in the MQL processor. A disadvantage with this solution, however, is that it will probably be more resource-demanding as it consists of both a grouping and a join.

Point #1 in the definition of evaluation of effective boolean value is also a bit problematic. This is because Tainting Dependencies, as it is presented in Chapter 4, does not handle empty sequences explicitly. And in most cases this is sufficient, as non-empty sequences containing empty items should be normalised. In the following query it is clear that the result must be all the `maybe` nodes of the document which is a child of a `elem` node:

```
fn:doc("nodes.xml")//elem/maybe
```

The `elem` nodes which does *not* contain a `maybe` node (making `../elem/maybe` an empty sequence) are irrelevant to the evaluation of this query. The only time the need for knowledge of a possible empty sequence arises is when evaluating the effective boolean value. Consider the following XQuery expression:

```
if($a/maybe) then
  "exists"
else
  "empty"
```

Further, let `$a` be an iterator variable consecutively bound to one and one `elem` node. If some of the `elem` nodes does not contain a `maybe` node, some items of the resulting sequence should be the string literal `"empty"`.

A possible solution to this would be to differantiate between the times the empty sequences are needed, and the times they are not. This can be done by evaluating all the descendant expressions of an expression which is to be calculated into a effective boolean value, in its own context. To evaluate some expressions in another matter according to the context is made simple by implementing a context sensitive visitor pattern (section 2.4.4). The difference in translating in the logical context as opposed to the default context will only be that all joins on common dependencies will have to be made into full outer joins. In addition joins such as in the axis+name test translation (Rule 4.19) where only the left operand may have dependencies, will have to be turned into a left-outer join. Always employing outer joins will ensure that no $-numb$ field corresponding to one unique iteration will be removed from the relation. Using this solution xqBoolean() must return $false$ if it is run with $NULL$ as input.

7.3 External environment

XQuery provides some different ways to communicate with the external environment, that is, the world outside the query. One of these features is that variables and functions may be declared with the `external` keyword. A query containing such expressions will have to be translated by the translator, leaving space where the expressions are referred to. These spaces will have to be filled by another part of the system. If the contents of the variables and functions are known before runtime, a better solution would be to insert them into the symbol and function tables of the translator before translation commences.

The functions `fn:doc` and `fn:collection` also provide access to external data. The `fn:doc` function takes a string containing a URI. If that URI is associated with a document among the available documents, the function returns a document node representing the given document. Conformant to Tainting Dependencies, this function would need to map the document name to a document id, and return a relation containing one tuple with this document id in the *documentId* field and an empty *scope* field.

A collection can be any sequence of nodes. The `fn:collection` function also takes takes a string containing a URI. The URI will have to be associated with a collection, which is returned from the function. To accomodate for this function with Tainting Dependencies, the collection returned must be in form of a relation with *index*, *documentId*, *pos*, *scope* and *value* fields. The collection may of course be a collection of document nodes.

The final way for XQuery to access the external environment is through initial values (appendix C.2 of [36]). Most important of these is the context item and the default collection. The default collection is referred to with a function call with `fn:collection` without an URI as parameter. The initial context item is referred to explicitly with the `.` (dot) operator, or implicitly at the start of a path expression. These features may be supported in Tainting Dependencies by inserting the values in the function and symbol table, respectively.

7.4 Type system considerations

Currently, neither Tainting Dependencies (TD) nor the prototype implementation assumes any form of availability of a type system. However according to the formal semantics specification[33] (and also noted in [13]), XQuery Core is inherently fully statically typed. This suggests that full normalisation of queries to XQuery Core would imply the availability of this. Static typing could help solve some problems, such as distinguishing numeric predicates. However, XQuery Core has semantics for solving this exact problem – the predicate normalisation mapping applies a `typeswitch` construct which contains the necessary logic to differentiate between numeric predicates and other predicates. And regardless, numeric predicates were solved in TD as per Rule 4.20.

Another related challenge is the lack of explicit typing in the MQL language. The only concrete specification given by Fast Search & Transfer is the fact that if a column is typed, then if can only contain fields of that type. MQL has typed columns, however it is not possible to specify type. For example, the make operator has no parameters for type specification. This is complicated further by the fact that a XQuery sequence is simply just a sequence. The individual items themselves can have vastly different types.

The intricate challenges related to typing are numerous. For example, consider the sequence `(1, <a><b>2</b></a>, "3")`. This sequence can not be represented in a relation without resorting to a BLOB[1]-like data type for the value column. However, that implies that the semantics of the second item as an XML element is lost unless it is somehow serialised in a common format with which MQL is compatible. That again implies that metainformation about fields may be required to indicate the type of the contents.

Furthermore, the is*axis*() functions proposed in section 3.4.4 on page 37 requires a *scope* field in the tuple being tested. Consider this non-sensical example:

[1]Binary Large Object

```
for $i in (1,2,3) return
  $i/a
```

Somehow, this error must be discovered and prevented. Likewise, the following example must be allowed without errors:

```
for $i in (<a><b>1</b></a>, <a><b>2</b></a>, <a><b>3</b></a>) return
  $i/b
```

On a final note, non-heterogeneous sequences are seldom of practical use, and can appear irrational. Path expressions always return nodes, string operations always return strings, and so on. Non-heterogeneous sequences are, as far as known to the authors, only specifiable by an end-user of XQuery, for example by attempting to execute a query such as this:

```
for $i in (1, <a><b>2</b></a>, "3") return
  $i * 2
```

This particular example appears non-sensical, and will likely not execute. For example, the Saxon parser[6] returns this run-time error message when attempting to execute the above query:

```
XPTY0004: Unsuitable types for * operation
Query processing failed: Run-time errors were reported
```

Naturally, it appears that typing is an important but complex aspect of XQuery, and several issues such as the ones described here needs to be solved for a full implementation to take place.

7.5 Optimisation

In Section 4.10 we presented some situations where the translation from XQuery into MQL can be specialised and simplified. In this section we will outline some ideas for further simplification and optimisation.

As can be seen of the tree in Figure 6.3 on page 95, Tainting Dependencies can produce relational algebra trees with consecutive project operators. This is of course unnecessary, and such operators can be merged into one single project. Additionally, attributes which is not part of the result or part of evaluation of the result can be pruned. This may however not be anything the translator would need to consider, as the MARS optimiser already implements methods for detecting and simplifying such trees.

Rule 4.16 on page 63 describes the translation of XQuery `if..then..else` expressions. It was implemented considering the result would be a operator tree and not a DAG, and the number of operations should be minimised. In this translation both the `then`-expression and the `else`-expression will be evaluated and their results spliced together. From this relation the tuples stemming from the right expression according to the result of the test expression is selected. If the MQL processor labours DAGs it may be a better solution to evaluate the two expressions only for the cases where they are to be returned.

This may be done in a matter very similar to the Loop Lift solution (equation 2.9 on page 26). In this solution different *loop* relations are made depending on the outcome of the test-expression. One of the relations are used to evaluate the `then`-expression, another the `else`-expression. As Tainting Dependencies does not utilise *loop* relations, some changes would have to be made. Evaluation of the test-expression will reveal which unique iterations which expression is to be evaluated in. Based on this information iterations can be pruned from the two expressions before they are evaluated.

7.5.1 XQuery semantics

As XQuery is a purely functional language, an implementation is always free to evaluate the operands of an operator in any order[36]. This means that a MQL optimiser is able to rearrange operators as it sees fit, with the possible outcome of a less expensive query execution. One such case would be in a path expression such as this:

```
$i/child::elem/descendant::maybe
```

If the optimiser has some notion of the frequency of occurence of the `maybe` and `elem` nodes, as well as the cardinality of the relational representation of `$i`, it may not be evaluated from left to right. If there exists only a few `maybe` elements and the `$i` relation is relatively large, the most effective execution plan for this path expression would most likely be from right to left.

In some cases, a XQuery implementation can determine the result of an expression without accessing all the data that is implied by the formal expression semantics. A consequence of this is that some errors goes undetected. W3C specify to which extent an implementation may optimise its access to data at the cost of not detecting errors like this [36]:

> Consider an expression Q that has an operand (sub-expression) E. In general the value of E is a sequence. At an intermediate stage during evaluation of the sequence, some of its items will be known and others will be unknown. If, at such an intermediate stage of evaluation, a processor is able to establish that there are only two possible outcomes of evaluating Q, namely the value V or an error, then the processor may deliver the result V without evaluating further items in the operand E. [. . .]
>
> There is an exception to this rule: If a processor evaluates an operand E (wholly or in part), then it is required to establish that the actual value of the operand E does not violate any constraints on its cardinality. [. . .]

This feature could be utilised in situations where path expressions are to be evaluated to effective boolean values (as described in Section 7.2.1). If the path expression is to return either nodes or an empty sequence, it is sufficient to find *one* node per unique iteration. This might prove difficult to implement on an MQL processor however, as it is hard to know which unique iteration an node belongs to before the node relation and a relation with representations in all unique iterations are joined together.

Another situation where not accessing all data is required is with numeric predicates. The formal description of filter expressions[33] says that an expression such as `$s[1]` should be evaluated by consecutively examining the items of the sequence `$s`, and selecting all items where `position()=1`. A better solution would be to only pick the first item of the sequence. But as Tainting Dependencies stores sequences from many iterations in the same relation, and no ordering can be assumed, this can not be done easily. The MQL processor will have to sequentially scan the relation to find the tuples holding the items which are the first item of their respected sequences, unless there is som kind of indexing of the relation on the *index* fields. In such a case the processor could simply lookup the tuples where *index* has the value 1.

7.5.2 Path expressions

Quite a lot of research has been done on optimising XPath (Path expressions in XQuery stem from XPath) since it became a W3C Recommendation in 1999. Of the documents produced by this research we would recommend [22], [21] and [14]. As this is outside the scope of this report the contents of these will not be resited here. However, [9] presents a reverse axis removal algorithm which may be interesting in a MQL and Taiting Dependencies specific setting. The algorithm recognises path expressions containing reverse axis steps, and rewrites them into pure forward axis step expressions with possible predicates. This may be helpful as the MQL `scope` operator only accepts steps equivalent to the `child` axis.

By letting the translator keep track over valid consecutive `child` axis steps, the `scope` operator may be employed to filter the results from the lookup (ref Rule 4.19, page 67), most likely reducing the tuples involved in the subsequent join. Consider the following path expression:

```
/a/b[g]/c//d[h]/e
```

Here, the result of looking up `g` may be filtered by a `scope` operator with `/a/b/g` as its parameter. Similarly, the lookup of `c` may be filtered on `/a/b/c`. Because the name test `d` is not part of a `child` axis step (rather a `descendant-or-self` axis step), no filtering can be done. However `e` may be filtered by `scope` with the parameter `d/e`. Some consideration will still have to be done concerning how much cheaper the join will become with the filtering compared to the cost of filtering one of its operands. Additionally, if a leading slash (`/`) in the parameter to the `scope` operator would indicate an absolute path, the operator could in many cases perform a more strict and accurate filtering. Without this leading slash, the filtering would be on a relative path – as before.

One idea would be to move all predicates out of the path expression, and apply them as post filters. This would probably reduce the number of required joins – at least in some cases. In other cases it may be to costly to build a filter fitted to the whole path expression. If e.g. `h` from the last example query was a number instead, the path expression up until `d` would have to be evaluated, filtered with the `g` predicate, and finally joined with the evaluation of the path expression without the last predicate on the *scope* fields of their `d` step.

The rule for translating predicates, Rule 4.20 on page 70, is a general rule. Here, a reference to the context item from the outside of the predicate is "copied" inside the predicate (containing *sprDotNumb*). This copy is then operated on, and the evaluated predicate expression and the predicated expression is joined on the reference to the context item. An example of this process is seen in figure 6.4 on page 96. Here, the copying is seen in the form of a upwards split. However, if the context item is only referred to once within the predicate this translation may however be simplified.Consider the following XQuery query:

```
//person[name eq "Robert"]
```

This can be solved by joining the `name` relation with the `person` relation, and keeping all the attributes from both relations. The attributes stemming from the `name` relation would have to be marked in some way, as these are not a part of the result of the query. The relation will have to be filtered with a selection removing the tuples where the *scope* stemming from the `name` is not a child scope of the *scope* stemming from the `person` relation. Further, only the tuples where the *value* stemming from the `name` relation is `"Robert"` is retained. Finally the attributes stemming from the `name` relation are projected away.

A similar solution may be considered for other types of predicates as well, but if there is more than one reference to the context item within the predicate things get more cumbersome. In such a case the method would have to be sure that no reference to the context item is removed from the relation at any time before the finalisation of the evaluation.

7.6 Implementation

Chapter 5 describes how a prototype was implemented to demonstrate the “Tainting Dependencies” method. This implementation was dependent on a number of constraints:

- The availability of a free[2] pre-made XQuery parser capable of producing abstract syntax trees
- The ability to parse and manipulate syntax trees and re-write them into new structures
- The ability to translate syntax trees into MQL (MARS relational algebra)

In this section, the methods chosen to achieve the goals of the implementation are discussed and elaborated upon.

7.6.1 Manual vs. automated tree parser construction

ANTLR provides a utility for automated construction of AST parsers. This utility requires the specification of a separate tree grammar. The tree grammar is almost identical to the original parser grammar. Practically, the parser grammar can be copied verbatim, renamed, modified slightly and used as a tree grammar. This process is described in detail in [20], section 8.1.

This introduces some redundancy. If not all nodes in the AST can be matched by the tree grammar, the parser will throw an error for encountering an unknown token. This implies that an ANTLR tree grammar will need to recognise all tokens in a syntax tree, thus the tree grammar in some ways depends on being synchronised with the parser grammar to be able to function properly.

This creates a potential problem with maintainability. As the parser grammar and rewrite rules are not freezed at this point but rather highly subject to change, any changes made in the parser grammar will need to be transferred to the tree grammar, and vice versa.

In [2], Terence Parr argues that the traditional visitor pattern (Section 2.4.4) only provides a simplistic facility for triggering events on the AST, that no tree structure validation is implicitly available, and that context information has to be passed down through the tree during the parse or by setting global variables.

[2]By “free” is meant a liberal license and availability of source code

In another point of view strongly polar to that of Terence Parr, Andy Tripp argues that manual tree parsing is better[1]. He establishes the following points of argument which are of particular interest to this project:

- Duplication of code and effort – the concept of "what is a valid AST" would have to be implemented in both the parser and the AST transformer phase (as a rebuttal to validation of AST).
- With regards to contextual information, There seems to be nothing wrong with depending on the physical structure of the AST.
- Defining a traditional parser in grammar is practical because the grammar usually resembles the ouput AST. In the case of a tree parser proposed by Parr where the grammar actually resembles the input AST, this mapping may break down completely if the output is another tree structure.

In particular, the last point holds a strong indication that a tree grammar may not be suited for this project, as the goal of this tree parser would be to transform the AST into a relational algebra tree.

7.6.2 AST rewriting and the visitor pattern

In Chapter 3, methods to achieve the goals of the implementation were presented. The method chosen for parsing of the abstract syntax tree (see Section 3.2) was the *context sensitive visitor pattern*. This pattern laid the foundation for a clean and simplistic implementation. The semantics of the tree parsing process itself did not interfer unecessarily with the rest of the implementation.

The process of rewriting the abstract syntax tree was implemented as a stand-alone visitor (the `RewriteVisitor` class). This implementation exploited the visitor pattern extensively, resulting in a clear separation of concerns. In particular, it seems to hold true that the visitor pattern typically will cleanly separate a data structure from an algorithm which is operating on that structure.

7.6.3 Constructed algebra trees and performance

As explained in Section 5.5 on page 81, the MQL is constructed as an in-memory tree structure. This was done by instantiating a new `Operator` subclass (the exact class depending on context) for every node in the tree. It is important to note that even though this could become a performance bottleneck for very large and complex queries, it is still an important trade-off. In exchange for a theoretical performance bottleneck, the implementation achieves a higher level of maintainability.

7.7 Normalisation and rewriting

In Section 3.3 on page 30, some methods for rewriting (normalising) certain expressions to XQuery Core were described. In the prototype implementation, these rewrites are made using the `RewriteVisitor` class. The advantage of normalising to XQuery Core is simplification of the syntax tree while maintaining full semantics. That is – the final syntax tree may be bigger and appear more complex. However, expressions such as FLWOR and path expressions are split into smaller subexpressions that are easier to parse by themselves.

This project has taken an pragmatic approach to normalisation. The rules defined in Chapter 4 do not rely on normalisations. However, for the sake of simplicity in the prototype, a rewrite visitor was applied to simplify FLWOR expressions. This must be seen in the light of the fact that XQuery Core is a very extensive specification[33], and so strict adherence to this specification would imply a substantially larger amount of effort into normalisation.

Consider the normalisation of `RelativePathExpr/StepExpr` which is normalised into a FLWOR expression[3]. This may be counter-productive as the usage of the `scope` operator in MQL will imply that this normalisation will somehow have to be reversed.

Furthermore, the normalisation of FLWOR expressions themselves require that `where`-clauses are rewritten to `if..then..else` expressions. The rules for this normalisation process is shown in section 2.1.6 and figure 2.3 on page 2.3. However, in Tainting Dependencies (TD) the translation of a `where`-clause (Rule 4.10 on page 56) is optimised and shown to be more efficient than the translation of an equivalent `if..then..else` expression (rule 4.16 on page 63). This is a paradoxical situation, and raises the question of whether other normalisation rules may also affect the efficiency of the resulting translation.

On a final note, the TD method in its current state does not rely on denormalised XQuery – however it is compatible with XQuery Core since XQuery Core is a subset of XQuery.

7.8 Results

Chapter 6 presented a series of algebra trees – some calculated by hand using Tainting Dependencies, and others generated using the prototype implementation described in Chapter 5. Furthermore we compared algebra generated by the prototype with that generated by Pathfinder. This section will discuss these results in detail.

7.8.1 Translation output

In sections 6.1 and 6.2, a series of XQuery queries were translated using the novel Tainting Dependencies (TD) methodology developed and described in Chapter 4. In section 6.1, where hand-computed translations were presented, a series of simplifications were applied (these simplifications were described in Section 4.10). However, the prototype developed in Chapter 5 did not implement any of these simplifications. This was an important point to keep in mind when later comparing this algebra to that generated by Pathfinder, and is discussed more thoroughly in the next section.

One characterisation of the algebra generated by TD is that nodes with more than one parent node are typically located far towards the bottom of the algebra tree.

Another characteristic of the algebra is that is seems to maintain a fairly compact form. This is partly due to the fact that the tainting process does not affect constant subexpressions, and

[3]See http://www.w3.org/TR/xquery-semantics/#id-axis-steps for details

thus the potential size of the algebra is reduced significantly. Compare this to Loop Lifting used by Pathfinder, where all expressions within a loop body are loop lifted – as explained in sections 2.5.3, 2.5.6, and 2.5.7.

7.8.2 Complexity comparison

The complexity calculation method (see Section 3.5 on page 39) defined by Øystein Torbjørnsen at Fast Search & Transfer was used to compare complexity in the algebra generated by the prototype implementation to that of Pathfinder. This comparison was based on three queries (*Trivial*, *Complex*, and *Conditional*). For each of these queries, algebra was generated on both the prototype implementation as well as Pathfinder. Then the described method of calculating complexity was applied to these trees, and the result was used to compare the prototype and Pathfinder.

We found that in terms of counting tuple and field creations, TD seems to excel in large and complex queries. In the case of more trivial queries, Loop Lifting and TD seem to perform similarly. Furthermore, in the case of tuple input/output in join and sort operators, we saw that for the "complex" query example, TD did not produce any equi-joins at all, while Loop Lifting produces 3 equi-joins for the same expression.

Though an interesting comparison, this is a sparse source of data – it is difficult to be conclusive based this data alone. However, with the exception of the most trivial query, it seems that the "Tanting Dependencies" (TD) method generates substantially less complex algebra than Pathfinder. As one may consider TD more specialised method than the general Loop Lifting technique, this should emerge as a natural consequence.

However, there are some sources of uncertainty for this comparison. It is not known exactly to which degree simplifications and optimalisations have been applied to the algebra generated by Pathfinder. In any case, the algebra generated by TD is not simplified or optimised, and as such puts these implementations on equal footing.

It is also known that Pathfinder does *not*:

- generate algebra using pure Loop Lifting as would be expected from [28] and [26] – this is deduced by comparing the output from Pathfinder with the output from the rules defined in [28] and [26]
- apply all simplifications described in [26] – again, this is deduced by comparing the output from Pathfinder with the supposed output from the simplifications in [26]

Additionally, in Section 3.5.1 on page 40 it was assumed that the `Diff` and `Distinct` operators utilised by Pathfinder both creates 0 tuples and 0 fields, only relaying the input to output. It is natural to assume that both of these are in fact costly operators in some aspects. However, given that `Diff` computes a difference between result sets, this does not imply that it creates new tuples and/or fields. Furthermore, the `Distinct` operator only removes duplicates, and as such it should be safe to assume that it does not generate new tuples and/or fields. This implies that these assumptions may favour Pathfinder, but likely not in the context of the method for calculation of complexity used here.

Furthermore, it is natural to assume that Pathfinder generates algebra which is tailored for execution on the MonetDB database system (as indicated in [26]).

With regards to performance measurement, and with the lack of availability of a proper implementation of a MQL processor (as mentioned in section 3.4 on page 31), it could have been of interest to generate algebra using TD modified for MonetDB, and compare actual performance of Loop Lifting vs. TD on this database system. This notion is further detailed in Section 9.

Finally, it is important to note that the complexity comparison performed did not in any way account for disk I/O or CPU and memory usage, and the results must not be interpreted as such. Again, see Section 3.5 on page 39 for an detailed account of this method.

7.8.3 Loop Lifting vs Tainting Dependencies

This project has studied two approaches for translating XQuery to relational algebra; Loop Lifting as implemented by Pathfinder, and “Tainting Dependencies” (TD), which is a novel method developed in this project which still shares a few common traits with Loop Lifting. However, the motivation for the development of TD was the fact that the more expressive MQL algebra allowed more flexibility in the translation. Furthermore, Loop Lifting had the disadvantage of full denormalisation, as noted in [26]:

> [..] Loop Lifting consequently leads to a fully denormalised representation for e and thus to – at least potentially – significant data redundancy

And so a major motivation for the development of TD was avoiding this level of denormalisation.

Another common trait of Loop Lifting, which is also noted in [28] and [26], is that the algebra trees will quickly grow very large. Consider the example in section 2.5.8 on page 26; here, the intermediate results grow in size very quickly. In particular, they are comparatively large seen in the context of the trivial query they are produced from.

When the algebra trees generated by Loop Lifting are converted to DAGs, this trait may not seem so appareant. However, it is easily recognised by the fact that nodes with more than one parent often are located in the middle and higher parts of the tree, for example as seen in figure 6.13(b) on page 104. If this particular DAG was converted to a tree, it would indeed be substantially larger.

Further, when comparing the rules in Loop Lifting (described in [28]) and TD, it appears that TD will in most cases produce less operators and less complex trees than Loop Lifting, as well as substantially smaller intermediate results. This comparison does not consider simplifications, however. In some situations, especially for trivial queries, optimised Loop Lifting trees *may* perform better than unoptimised TD.

7.8.4 Considerations for executing MQL

The “Tainting Dependencies” method produces relatively simplistic algebra when compared to the full feature set of MQL. In Section 3.4, only the operators used throughout this project was described. However, this is only a subset of the features in MQL. This was done with the intent of creating algebra which can be optimised using common techniques already available. Introducing new and exotic operators complicates this process, and so this was avoided. However, in the light of the expressiveness available in MQL, it may be benefitial to employ a wider array of operators available when constructing MQL algebra. In any case, this requires further documentation and concrete performance measurements.

Chapter 8

Conclusion

Throughout this project, we have explored the nature of XQuery and relational algebra. We have studied one well-researched method of translation, "Loop Lifting", and found points of improvement as well as untapped potential in the fact that the target language MQL is a more expressive form of algebra which may allow more creative and efficient translations.

We have developed a novel method of translation dubbed "Tainting Dependencies" (TD) which seeks to avoid unecessary denormalisation of intermediate results, and which is also designed specifically for translation to MQL. Our method currently supports a substantially large subset of the XQuery language – however lacking functionality has been accounted for, and suggestions for solutions have been proposed.

Furthermore, **we developed a prototype as a proof of concept**, which is capable of translating XQuery queries containing basic constructs such as FLWOR expressions, sequence constructions, and conditional expressions (`if..then..else`).

Finally, based on a method of measurement for complexity defined by Øystein Torbjørnsen at Fast Search & Transfer, **we staged a comparison of our prototype TD implementation and an implementation of Loop Lifting called Pathfinder** developed by Teubner et. al at the University of Konstanz. Not withstanding the weaknesses of this method of comparison, we then empirically suggested that our method may produce less complex and more efficient relational algebra.

Further research may be required, however the outcome of this project is a fairly complete, novel and well-documented method for translation of XQuery to MQL – a method which is designed to perform equally or better than existing implementations.

Chapter 9

Future Work

The outcome of this project is a fairly well-defined method of translation, there is still headroom for improvement and continued research. In particular, this relates to performance benchmarking, simplifications and optimisations, and improvement of XQuery feature support.

We propose the following research related to performance benchmarking:

- **Run the XMark benchmark suite and compare results**. Given that Fast Search & Transfer develops a working implementation of an MQL processor some time in the future, it could be interesting to extend or rewrite the prototype developed here, and execute the XMark[1] benchmark suite and compare the result to other existing implementations
- **Execute algebra generated by Tainting Dependencies on MonetDB**. If a working MQL processor can not or will not be developed in the foreseeable future, then the prototype and the rules of TD may be modified and interfaced with MonetDB, and a comparison with Pathfinder could be performed on this combined system. However, the adaptation of TD to MonetDB may be non-trivial, as the method depends on some MQL specific features.
- **Further research on optimisation and simplification of the TD methodology**. This thesis suggests some simplifications (section 4.10 and 7.5), however we suspect there are still substantial gains to be made on this account
- **Investigate applicability of parallell execution of subtrees in the algebra tree**. MQL supports threading/branching within the language itself, and this may be exploited to parallellise the execution of algebra and boost performance

Furthermore, we propose the following improvements:

- **Improved support for interfacing with the external environment** (as described in Section 7.3)
- **Improved support of XQuery functionality**, including: full-text operations, `ordered` and `unordered` mode, binary operators not currently supported, multiple order specifi-

[1]http://www.xml-benchmark.org/

cations for the `order by`-clause, and user-defined functions as well as built-in XQuery functions (within the `fn` namespace)

- **Implementation of the full XQuery type system** into TD, which may also possibly be exploited for optimisations
- **Optimisation of `if..then..else` expressions** by assuming execution of DAGs and not trees, as described in section 7.5

Appendix A

Translation process Using TD

This appendix contains step-by-step translations of some XQuery expressions using Tainting Dependencies.

A.1 An extensive FLWOR expression

This example shows the translation process for extensive FLWOR expression of Section 6.1.

The translation process is initiated by entering the FLWOR expression in the syntax tree and visiting the for- and let-clauses. Recall from section 4.3 on page 46 that $\mathtt{r}(e)$ returns the translation of the XQuery expression e into relational algebra. This function is then defined through a set of rules described throughout Section 4.

The translation process starts with the `for`-clause which we translate using Rule 4.2. However, to produce $\mathtt{r}(e)$, we must translate `(1,2,3)` using rules 4.6 and 4.5.

By applying Rule 4.5 to each of the elements in the sequence, we obtain the following translations:

```
make(name:=[index, value], [1], [1])
```

```
make(name:=[index, value], [1], [2])
```

```
make(name:=[index, value], [3], [3])
```

By applying Rule 4.6 to this result we obtain the following translation of the sequence (1,2,3):

```
numberate(index,[sprIdx,index],[];
  union(;
    make(name:=[index, value], [1], [1])
```

```
    make(name:=[index, value], [1], [2])
```

```
    make(name:=[index, value], [3], [3])))
```

We can now continue translating the `for`-clause, as the above translation equates to $\mathbf{r}(e)$. By Rule 4.2, we obtain the following translation which is to be entered into the symbol table for this scope (for the symbol $\chi = \text{a}$):

```
project([anumb = index, index = 1, value];
  numberate(index,[sprIdx,index],[];
    union(;
      make(name:=[index, value], [1], [1])
```

```
      make(name:=[index, value], [1], [2])
```

```
      make(name:=[index, value], [3], [3]))))
```

Any later reference to `$a` is now replaced with a lookup in the symbol table which will return this algebra expression.

The let clause is translated in a similar manner, however the entry in the symbol table will not be tainted by `$b`. To translate the let-clause, we apply the Rule 4.5 on the expression and obtain:

```
make(name:=[index, value], [1],[2])
```

Which is entered into the symbol table as per Rule 4.7.

Before translating the where- and orderby-clauses, it is benefitial to translate the return-clause (since the two former requires the translation of the latter). The translation is obtained using rule 4.8 (return-clause) and rule 4.6 (sequence construction), which is trivial. Recall that references to `$a` and `$b` are replaced by their translations in the symbol table:

```
numberate(index,[sprIdx,index],[];
  union(;
    project([anumb = index, index = 1, value];
      numberate(index,[sprIdx,index],[];
        union(;
          project(sprIdx=1,index=0,value;
            make(name:=[index,value], [1], [1])),
          project(sprIdx=2,index=0,value;
            make(name:=[index, value], [2], [2])),
          project(sprIdx=3,index=0,value;
            make(name:=[index, value], [3], [3]))))),
    project([anumb = index, index = 2, value];
      make(name:=[index, value],[1], [2]))))
```

This translation is now used to replace $\mathbf{r}(e_C)$ in the translation of the where-clause, which is translated using rules 4.10 and 4.12:

The translation process is finalised by translating the where-clause using rules 4.10 and 4.15. First we obtain the following translation from `$a gt $b`:

```
    project(index=1, value=gt(r.value, l.value), anumb;
      hhjoin([], [], [anumb, lvalue=l.value, rvalue=r.value, alt];
        project([anumb = index, index = 1, value];
          numberate(index,[sprIdx,index],[];
            union(;
              project(sprIdx=1,index=0,value;
                make(name:=[index, value], [1], [1])),
              project(sprIdx=2,index=0,value;
                make(name:=[index, value], [2], [2])),
              project(sprIdx=3,index=0,value;
                make(name:=[value], [3], [3]))))),
          make(name:=[index, value],[1], [2])))
```

Which is entered together with $\mathbf{r}(e_C)$ into the translation of the where-clause:

```
hhjoin([],[], [l.value, anumb];
  numberate(index,[sprIdx,index],[];
    union(;
      project([anumb = index, index = 1, value];
        numberate(index,[sprIdx,index],[];
          union(;
            project(sprIdx=1,index=0,value;
              make(name:=[index, value], [1], [1])),
            project(sprIdx=2,index=0,value;
              make(name:=[index, value], [2], [2])),
            project(sprIdx=3,index=0,value;
              make(name:=[value], [3], [3]))))),
      project([anumb = index, index = 2, value];
        make(name:=[index, value],[1], [2]))),
  select(xqBoolean(value);
    project(index=1, value=gt(r.value, l.value), anumb;
      hhjoin([], [], [anumb, lvalue=l.value, rvalue=r.value, alt];
        project([anumb = index, index = 1, value];
          numberate(index,[sprIdx,index],[];
            union(;
              project(sprIdx=1,index=0,value;
                make(name:=[index, value], [1], [1])),
              project(sprIdx=2,index=0,value;
                make(name:=[index, value], [2], [2])),
              project(sprIdx=3,index=0,value;
                make(name:=[value], [3], [3]))))),
          make(name:=[index, value],[1], [2]))))))
```

This result is used to replace $\mathbf{r}(e_C)$ in the translation of the orderby-clause, and we obtain the complete translation:

```
project(value=l.value, index;
  numberate(index, [r.value, index], [];
    hhjoin([anumb], [anumb], [l.value, r.value];
      project([anumb = index, index = 1, value];
        numberate(index,[sprIdx,index],[];
          union(;
            project(sprIdx=1,index=0,value;
              make(name:=[index, value], [1], [1])),
            project(sprIdx=2,index=0,value;
              make(name:=[index, value], [2], [2])),
            project(sprIdx=3,index=0,value;
              make(name:=[value], [3], [3]))))),
      hhjoin([],[], [l.value, anumb];
          project(index=1, value=gt(l.value, r.value), anumb;
            hhjoin([], [], [anumb, lvalue=l.value, rvalue=r.value, alt];
              numberate(index,[sprIdx,index],[];
                  union(;
                    project([anumb = index, index = 1, value];
                      numberate(index,[sprIdx,index],[];
                        union(;
                          project(sprIdx=1,index=0,value;
                            make(name:=[index, value], [1], [1])),
                          project(sprIdx=2,index=0,value;
                            make(name:=[index, value], [2], [2])),
                          project(sprIdx=3,index=0,value;
                            make(name:=[value], [3], [3]))))),
                    project([anumb = index, index = 2, value];
                      make(name:=[index, value],[1], [2])))),
              select(xqBoolean(value);
                project(index=1, value=gt(l.value, r.value), anumb;
                  hhjoin([], [], [anumb, lvalue=l.value, rvalue=r.value, alt];
                    project([anumb = index, index = 1, value];
                      numberate(index,[sprIdx,index],[];
                        union(;
                          project(sprIdx=1,index=0,value;
                            make(name:=[index, value], [1], [1])),
                          project(sprIdx=2,index=0,value;
                            make(name:=[index, value], [2], [2])),
                          project(sprIdx=3,index=0,value;
                            make(name:=[value], [3], [3]))))),
                  project([anumb = index, index = 2, value];
                    make(name:=[index, value],[1], [2])))))))))))
```

This can be simplified by replacing the sequence construction nodes with a single operation `make(name:=[index, value], [1,2,3], [1,2,3]))`:

```
project(value=l.value, index;
  numberate(index, [r.value, index], [];
    hhjoin([anumb], [anumb], [l.value, r.value];
      hhjoin([],[], [l.value, anumb];
        numberate(index,[sprIdx,index],[];
          union(;
            project([anumb = index, index = 1, value];
              make(name:=[index, value], [1,2,3], [1,2,3])),
            project([anumb = index, index = 2, value];
              make(name:=[index, value],[1],[2])))),
        select(xqBoolean(value);
          project(index=1, value=gt(l.value, r.value), anumb;
            hhjoin([l.anumb], [r.anumb], [l.value, r.value, anumb];
              project([anumb = index, index = 1, value];
                make(name:=[index, value], [1,2,3], [1,2,3])),
              make(name:=[index, value],[1],[2]))))),
      project([anumb = index, index = 1, value];
        make(name:=[index, value], [1,2,3], [1,2,3])))
```

And further by magic:

```
project(value=l.value, index;
  numberate(index, [r.value, index], [];
    hhjoin([anumb], [anumb], [l.value, r.value];
      hhjoin([anumb],[anumb], [l.value, anumb];
          union(;
            project([anumb = index, index = 1, value];
              make(name:=[index, value], [1,2,3], [1,2,3])),
            project([anumb, index = 2, value = 2];
              make(name:=[index, value], [1,2,3], [1,2,3]))),
        select(xqBoolean(value);
          project(index=1, value=gt(value, 2), anumb;
            project([anumb = index, index = 1, value];
              make(name:=[index, value], [1,2,3], [1,2,3]))))),
      project([anumb = index, index = 1, value];
        make(name:=[index, value], [1,2,3], [1,2,3])))))
```

A.2 If-then-else

This translation is initiated precisely as the previous "Extensive FLWOR" example, so we will not reiterate the translation of the for-clause here. However, note that the following expression is obtained and entered into the symbol table to represent `$a`:

```
project([anumb = index, index = 1, value];
  numberate(index,[sprIdx,index],[];
    union(;
      project(sprIdx=1,index=0,value;
        make(name:=[index, value], [1], [1])),
      project(sprIdx=2,index=0,value;
        make(name:=[index, value], [2], [2])),
      project(sprIdx=3,index=0,value;
        make(name:=[value], [3], [3]))))
```

The translation of the if-then-else expression follows rule 4.16 on page 63. First, however, the boolean expression `$a > 2` (which corresponds to e_1 in Rule 4.16) must be translated. This can be done using Rule 4.15, as for the where-clause in the previous FLWOR example:

```
    project(index=1, value=gt(r.value, l.value), anumb;
      hhjoin([], [], [anumb, lvalue=l.value, rvalue=r.value, alt];
        project([anumb = index, index = 1, value];
          numberate(index,[sprIdx,index],[];
            union(;
              project(sprIdx=1,index=0,value;
                make(name:=[index, value], [1], [1])),
              project(sprIdx=2,index=0,value;
                make(name:=[index, value], [2], [2])),
              project(sprIdx=3,index=0,value;
                make(name:=[value], [3], [3]))))),
          make(name:=[index, value],[1], [2])))
```

Next we translate e_2 (which is simply a lookup in the symbol table for `$a`):

```
project([anumb = index, index = 1, value];
  numberate(index,[sprIdx,index],[];
    union(;
      project(sprIdx=1,index=0,value;
        make(name:=[index, value], [1], [1])),
      project(sprIdx=2,index=0,value;
        make(name:=[index, value], [2], [2])),
      project(sprIdx=3,index=0,value;
        make(name:=[value], [3], [3])))))
```

The last expression, e_3 (which is tainted by `$a`), can be translated using a simplification where instead of constructing the tuples, an additional field with the value 3 is projected onto the result from a lookup of `$a` in the symbol table:

```
project(anumb, index = 1, value = 3;
  project([anumb = index, index = 1, value];
    numberate(index,[sprIdx,index],[];
      union(;
        project(sprIdx=1,index=0,value;
          make(name:=[index, value], [1], [1])),
        project(sprIdx=2,index=0,value;
          make(name:=[index, value], [2], [2])),
        project(sprIdx=3,index=0,value;
          make(name:=[value], [3], [3]))))))
```

Now the translated representations of e_1, e_2 and e_3 (where the latter is also tainted) can be entered into Rule 4.16:

```
project(value=lvalue, anumb;
  select(ifthenelse(xqBoolean(rvalue), eq(alt,1), eq(alt,2));
    hhjoin([l.anumb],[r.anumb], [anumb, lvalue=l.value, rvalue=r.value, alt];
      union(
        project(anumb, alt = 1, value;
          project([anumb = index, index = 1, value];
            numberate(index,[sprIdx,index],[];
              union(;
                project(sprIdx=1,index=0,value;
                  make(name:=[index, value], [1], [1])),
                project(sprIdx=2,index=0,value;
                  make(name:=[index, value], [2], [2])),
                project(sprIdx=3,index=0,value;
                  make(name:=[value], [3], [3])))))),
        project(anumb, alt = 2, value;
          project(anumb, index = 1, value = 3;
            project([anumb = index, index = 1, value];
              numberate(index,[sprIdx,index],[];
                union(;
                  project(sprIdx=1,index=0,value;
                    make(name:=[index, value], [1], [1])),
                  project(sprIdx=2,index=0,value;
                    make(name:=[index, value], [2], [2])),
                  project(sprIdx=3,index=0,value;
                    make(name:=[value], [3], [3]))))))),
      project(index=1, value=gt(l.value, r.value), anumb;
        hhjoin([], [], [anumb, lvalue=l.value, rvalue=r.value, alt];
          project([anumb = index, index = 1, value];
            numberate(index,[sprIdx,index],[];
              union(;
                project(sprIdx=1,index=0,value;
                  make(name:=[index, value], [1], [1])),
                project(sprIdx=2,index=0,value;
                  make(name:=[index, value], [2], [2])),
                project(sprIdx=3,index=0,value;
                  make(name:=[value], [3], [3])))),
            make(name:=[index, value],[1], [2]))))))
```

This can be simplified further by replacing `make()` as described in Rule 4.24:

```
project(value=lvalue, anumb;
  select(ifthenelse(xqBoolean(rvalue), eq(alt,1), eq(alt,2));
    hhjoin([l.anumb],[r.anumb], [anumb, lvalue=l.value, rvalue=r.value, alt];
      union(
        project(anumb, alt = 1, value;
          project([anumb = index, index = 1, value];
            make(name:=[index, value], [1,2,3], [1,2,3]))),
        project(anumb, alt = 2, value;
          project(anumb, index = 1, value = 3;
            project([anumb = index, index = 1, value];
             make(name:=[index, value], [1,2,3], [1,2,3])))),
      project(index=1, value=gt(value, 2), anumb;
        project([anumb = index, index = 1, value];
          make(name:=[index, value], [1,2,3], [1,2,3]))))))
```

The translation is finalised by applying Rule 4.9:

```
numberate(index, [anumb, index], [];
  project(value=lvalue, anumb;
    select(ifthenelse(xqBoolean(rvalue), eq(alt,1), eq(alt,2));
      hhjoin([l.anumb],[r.anumb], [anumb, lvalue=l.value, rvalue=r.value, alt];
        union(
          project(anumb, alt = 1, value;
            project([anumb = index, index = 1, value];
              make(name:=[index, value], [1,2,3], [1,2,3]))),
          project(anumb, alt = 2, value;
            project(anumb, index = 1, value = 3;
              project([anumb = index, index = 1, value];
               make(name:=[index, value], [1,2,3], [1,2,3])))),
        project(index=1, value=gt(value, 2), anumb;
          project([anumb = index, index = 1, value];
            make(name:=[index, value], [1,2,3], [1,2,3])))))))
```

A.3 Path expression with a predicate

It should be noted that the query is not a valid query unless the context item is set before execution. We assume the context item is stored in the symbol table, and have no dependencies. First the axis steps consisting of the name tests `a` and `b` is translated. As these are consecutive child-axis steps (no axis specifier is read as `child::`), Rule 4.25 may be used once for both steps, instead of using the more general rule (Rule 4.19) twice:

```
project(dotNumb, docId, index, value, pos, scope;
  numberate(dotNumb, [dotNumb, subIdx], [];
    select(isChild(scope, lsc);
      hhjoin([docId], [docId], [dotNumb,lsc=l.scope,subIdx=r.index,
            value=r.value,pos=r.pos,scope=r.scope];
        symtab.get(dot);
        numberate(index, [], [];
          index(valocc;
            scope(a/b;
              lookup($b)))))))
```

Also note that the next to last numberate operator from the rule is dropped, as the predicate expression must evaluate to a boolean type. This is described in Section 4.10.3.

Then the predicate will have to be taken into account. The result of `/a/b` is stored in the symbol table as **sprDot**, where *dotNumb* is copied to *sprDotNumb* and the *index* fields will be set to the value 1. When this is done, the comparison expression in the predicate will be evaluated. First `@id` is evaluated. This contains a implicit reference to the context node, and in its verbose form is formed like this: `./attribute::id`. This expression is evaluated by Rule 4.19. As this path expression (within the predicate) does not contain any predicate, the next to last numberate operator will be dropped. As it is the complete path expression it will be finalised by Rule 4.18. The last step of the path expression (still the one within the predicate) does not contain some kind of filtering, thus the final numberate operator will be exchanged with a project operator:

```
project(index=dotNumb, docId, value, pos, scope, sprDotNumb
  project(docId, index, value, pos, scope, dotNumb, sprDotNumb;
    numberate(dotNumb, [dotNumb, subIdx], [sprDotNumb];
      select(isChild(scope,lsc);
        hhjoin([docId],[docId],[dotNumb,lsc=l.scope,subIdx=r.index,
              value=r.value,pos=r.pos,scope=r.scope,sprDotNumb]
          project(sprDotNumb=dotNumb, dotNumb, index=1, value, pos,
                docId, scope;
            project(dotNumb, docId, index, value, pos, scope;
              numberate(dotNumb, [dotNumb, subIdx], [];
                select(isChild(scope, lsc);
                  hhjoin([docId], [docId], [dotNumb,lsc=l.scope,
                        subIdx=r.index,value=r.value,pos=r.pos,
                        scope=r.scope];
                    symtab.get(dot);
                    numberate(index, [], [];
                      index(valocc;
                        scope(a/b;
                          lookup($b))))))))
          numberate(index, [], []
            index(valocc;
              lookup($@id)))))))
```

The comparison expression will then be evaluated. By the simplifications described in section 4.10.1, the whole expression will be evaluated by adding a single project operator:

```
project(index=1, value=eq(value, 2), sprDotNumb;
  project(index=dotNumb, docId, value, pos, scope, sprDotNumb
    project(docId, index, value, pos, scope, dotNumb, sprDotNumb;
      numberate(dotNumb, [dotNumb, subIdx], [sprDotNumb];
        select(isChild(scope,lsc);
          hhjoin([docId],[docId],[dotNumb,lsc=l.scope,
                subIdx=r.index,value=r.value,pos=r.pos,
                scope=r.scope,sprDotNumb]
            project(sprDotNumb=dotNumb, dotNumb, index=1, value,
                  pos, docId, scope;
              project(dotNumb, docId, index, value, pos, scope;
                numberate(dotNumb, [dotNumb, subIdx], [];
                  select(isChild(scope, lsc);
                    hhjoin([docId], [docId], [dotNumb,lsc=l.scope,
                          subIdx=r.index,value=r.value,pos=r.pos,
                          scope=r.scope];
                      symtab.get(dot);
                      numberate(index, [], [];
                        index(valocc;
                          scope(a/b;
                            lookup($b))))))))
            numberate(index, [], []
              index(valocc;
                lookup($@id))))))))
```

Then the rule for translating predicates (Rule 4.20) are employed. The predicate expression and the predicated expression will have to be joined on their *sprDotNumb* and *dotNumb* attributes respectively, as they have no other common dependencies. Finally the whole path expression is finalised by a renumbering, as described by Rule 4.18.

```
numberate(index, [dotNumb, index], [];
  project(index,docId,scope,pos,value,dotNumb;
    select(ifthenelse(isNumber(pred),
          eq(index,pred),xqBoolean(pred));
      hhjoin([dotNumb],[sprDotNumb], [value=l.value,scope=l.scope,
            pos=l.pos,docId=l.docId,pred=r.value];
        project(dotNumb, docId, index, value, pos, scope;
          numberate(dotNumb, [dotNumb, subIdx], [];
            select(isChild(scope, lsc);
              hhjoin([docId], [docId], [dotNumb,lsc=l.scope,
                    subIdx=r.index,value=r.value,pos=r.pos,
                    scope=r.scope];
                symtab.get(dot);
                numberate(index, [], [];
                  index(valocc;
                    scope(a/b;
                      lookup($b)))))))
        project(index=1, value=eq(value, 2), sprDotNumb;
          project(index=dotNumb,docId,value,pos,scope,sprDotNumb;
            project(docId,index,value,pos,scope,dotNumb,sprDotNumb;
              numberate(dotNumb, [dotNumb, subIdx], [sprDotNumb];
                select(isChild(scope,lsc);
                  hhjoin([docId],[docId],[dotNumb,lsc=l.scope,
                        subIdx=r.index,value=r.value,pos=r.pos,
                        scope=r.scope,sprDotNumb]
                    project(sprDotNumb=dotNumb,dotNumb,index=1,
                          value,pos,docId,scope;
                      project(dotNumb,docId,index,value,pos,scope;
                        numberate(dotNumb, [dotNumb, subIdx], [];
                          select(isChild(scope, lsc);
                            hhjoin([docId], [docId], [dotNumb,
                                  lsc=l.scope,subIdx=r.index,
                                  value=r.value,pos=r.pos,
                                  scope=r.scope];
                              symtab.get(dot);
                              numberate(index, [], [];
                                index(valocc;
                                  scope(a/b;
                                    lookup($b)))))))
                    numberate(index, [], []
                      index(valocc;
                        lookup($@id))))))))))))
```

Appendix B

Links, resources and further reading

This appendix details some additional material which may be of use for the reader. Many of these resources are informally written, and should be considered as such.

B.1 Diagrams and graphs

Large-scale versions of the DAG graphs in Section 6.3 can be found in the subversion repository for this project available at the project website located at http://code.google.com/p/xqft-parser/. In particular, all graphs are available in full-scale versions in the `doc/img/graphs` folder in the project root, and all diagrams are available in full-scale versions in the `doc/diagrams` folder. Additionally, all example queries used throughout this project are available in `doc/graph_queries` and `doc/td_src`.

B.2 Further reading regarding XQuery

Michael Rys maintains an interesting weblog which contains some writing about the *XQuery type system*. His weblog is located at http://www.sqljunkies.com/WebLog/mrys/archive/2004/05/13/2480.aspx.

Michael Kay has written a series of papers and articles regarding XQuery and practical use thereof, especially some interesting pieces on XQuery Core and compilation of queries into Java bytecode. These writings are maintained at the *Saxon Diaries* website, located at http://saxonica.blogharbor.com/.

Appendix C

Installation and usage

Visit the project website at http://code.google.com/p/xqft-parser/ for the latest updates and installation instructions.

Project Directory Contents

The project directory contains a multitude of files and folders. Here is a list of the most important ones with a short explanation:

- bin - contains the compiled binaries
- doc - documentation (this report)
- etc - contains the source grammar file, XQFT.g
- lib - necessary runtime dependencies
- Makefile - Makefile for GNU Make
- src - all source code, the generated parser/lexer is moved to this directory after generation
- test - contains files related to testing and debugging
- tmp - temporary files generated during parser generation

Prerequisites

A computer with a Unix-based operating system is recommended however not required. Note that the makefiles provided are not immediately suitable for win32-based operating systems.

Software

This software should be available for download from the internet if not already installed on the system.

- GNU Make
- Subversion
- Java JDK 1.5.0 or newer

Optional software for generating AST and algebra graphs:

- Graphviz
- GNU Sed

Getting the Source

To download the source code using Subversion, execute the following in a command line interface:

```
svn checkout https://xqft-parser.googlecode.com/svn/trunk/ xqft-parser
```

To download the source code as a tarball, please visit the project website at http://code.google.com/p/xqft-parser/.

Compiling the Source

Move to the root directory of the source code, and enter the following command: `make`. This will generate a new parser/lexer pair, compile them together with supporting classes, and generate a convenient `ntnu-xqft.jar`-file.

lol section fitte

Command line interface

Refer to Section 5.3.3 on page 79 for a description of the command line interface.

Appendix D

EBNF for XQuery 1.0 Full-text grammar

This grammar is limited in the sense that it requires productions from XML and other specifications (such as XPath). For the full grammar with contextual links, please visit http://www.w3.org/TR/xpath-full-text-10/.

```
[1] Module ::= VersionDecl? (LibraryModule | MainModule)
[2] VersionDecl ::= "xquery" "version" StringLiteral
                    ("encoding" StringLiteral)? Separator
[3] MainModule ::= Prolog QueryBody
[4] LibraryModule ::= ModuleDecl Prolog
[5] ModuleDecl ::= "module" "namespace" NCName "=" URILiteral Separator
[6] Prolog ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import)
               Separator)* ((VarDecl | FunctionDecl | OptionDecl |
               FTOptionDecl) Separator)*
[7] Setter ::= BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl |
               ConstructionDecl | OrderingModeDecl | EmptyOrderDecl |
               CopyNamespacesDecl
[8] Import ::= SchemaImport | ModuleImport
[9] Separator ::= ";"
[10] NamespaceDecl ::= "declare" "namespace" NCName "=" URILiteral
[11] BoundarySpaceDecl ::= "declare" "boundary-space" ("preserve" | "strip")
[12] DefaultNamespaceDecl ::= "declare" "default" ("element" | "function")
                              "namespace" URILiteral
[13] OptionDecl ::= "declare" "option" QName StringLiteral
[14] FTOptionDecl ::= "declare" "ft-option" FTMatchOptions
[15] OrderingModeDecl ::= "declare" "ordering" ("ordered" | "unordered")
[16] EmptyOrderDecl ::= "declare" "default" "order" "empty" ("greatest" |
                        "least")
[17] CopyNamespacesDecl ::= "declare" "copy-namespaces" PreserveMode ","
                            InheritMode
[18] PreserveMode ::= "preserve" | "no-preserve"
[19] InheritMode ::= "inherit" | "no-inherit"
[20] DefaultCollationDecl ::= "declare" "default" "collation" URILiteral
[21] BaseURIDecl ::= "declare" "base-uri" URILiteral
[22] SchemaImport ::= "import" "schema" SchemaPrefix? URILiteral ("at"
                      URILiteral ("," URILiteral)*)?
[23] SchemaPrefix ::= ("namespace" NCName "=") | ("default" "element"
                      "namespace")
[24] ModuleImport ::= "import" "module" ("namespace" NCName "=")?
```

```
                         URILiteral ("at" URILiteral ("," URILiteral)*)?
[25] VarDecl ::= "declare" "variable" "\$" QName TypeDeclaration? ((":="
                    ExprSingle) | "external")
[26] ConstructionDecl ::= "declare" "construction" ("strip" | "preserve")
[27] FunctionDecl ::= "declare" "function" QName "(" ParamList? ")" ("as"
                         SequenceType)? (EnclosedExpr | "external")
[28] ParamList ::= Param ("," Param)*
[29] Param ::= "\$" QName TypeDeclaration?
[30] EnclosedExpr ::= "{" Expr "}"
[31] QueryBody ::= Expr
[32] Expr ::= ExprSingle ("," ExprSingle)*
[33] ExprSingle ::= FLWORExpr
                      | QuantifiedExpr
                      | TypeswitchExpr
                      | IfExpr
                      | OrExpr
[34] FLWORExpr ::= (ForClause | LetClause)+ WhereClause? OrderByClause?
                      "return" ExprSingle
[35] ForClause ::= "for" "\$" VarName TypeDeclaration? PositionalVar?
                      FTScoreVar? "in" ExprSingle ("," "\$" VarName
                      TypeDeclaration? PositionalVar?
                      FTScoreVar? "in" ExprSingle)*
[36] PositionalVar ::= "at" "\$" VarName
[37] FTScoreVar ::= "score" "\$" VarName
[38] LetClause ::= (("let" "\$" VarName TypeDeclaration?) | ("let" "score"
                      "\$" VarName)) ":=" ExprSingle ("," (("\$" VarName
                      TypeDeclaration?) | FTScoreVar) ":=" ExprSingle)*
[39] WhereClause ::= "where" ExprSingle
[40] OrderByClause ::= (("order" "by") | ("stable" "order" "by"))
                         OrderSpecList
[41] OrderSpecList ::= OrderSpec ("," OrderSpec)*
[42] OrderSpec ::= ExprSingle OrderModifier
[43] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest"
                         | "least"))? ("collation" URILiteral)?
[44] QuantifiedExpr ::= ("some" | "every") "\$" VarName TypeDeclaration?
                          "in" ExprSingle ("," "\$" VarName TypeDeclaration?
                          "in" ExprSingle)* "satisfies" ExprSingle
[45] TypeswitchExpr ::= "typeswitch" "(" Expr ")" CaseClause+ "default"
                          ("\$" VarName)? "return" ExprSingle
[46] CaseClause ::= "case" ("\$" VarName "as")? SequenceType "return"
                      ExprSingle
[47] IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[48] OrExpr ::= AndExpr ( "or" AndExpr )*
[49] AndExpr ::= ComparisonExpr ( "and" ComparisonExpr )*
[50] ComparisonExpr ::= FTContainsExpr ( (ValueComp
                          | GeneralComp
                          | NodeComp) FTContainsExpr )?
[51] FTContainsExpr ::= RangeExpr ( "ftcontains" FTSelection
                          FTIgnoreOption? )?
[52] RangeExpr ::= AdditiveExpr ( "to" AdditiveExpr )?
[53] AdditiveExpr ::= MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr)*
[54] MultiplicativeExpr ::= UnionExpr ( ("*" | "div" | "idiv" | "mod")
                              UnionExpr )*
[55] UnionExpr ::= IntersectExceptExpr ( ("union" | "|")
                      IntersectExceptExpr )*
[56] IntersectExceptExpr ::= InstanceofExpr ( ("intersect" | "except")
                               InstanceofExpr )*
[57] InstanceofExpr ::= TreatExpr ( "instance" "of" SequenceType )?
[58] TreatExpr ::= CastableExpr ( "treat" "as" SequenceType )?
```

```
[59] CastableExpr ::= CastExpr ( "castable" "as" SingleType )?
[60] CastExpr ::= UnaryExpr ( "cast" "as" SingleType )?
[61] UnaryExpr ::= ("-" | "+")* ValueExpr
[62] ValueExpr ::= ValidateExpr | PathExpr | ExtensionExpr
[63] GeneralComp ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[64] ValueComp ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[65] NodeComp ::= "is" | "<<" | ">>"
[66] ValidateExpr ::= "validate" ValidationMode? "{" Expr "}"
[67] ValidationMode ::= "lax" | "strict"
[68] ExtensionExpr ::= Pragma+ "{" Expr? "}"
[69] Pragma ::= "(#" S? QName (S PragmaContents)? "#)"/* ws: explicit */
[70] PragmaContents ::= (Char* - (Char* '#)' Char*))
[71] PathExpr ::= ("/" RelativePathExpr?)
                | ("//" RelativePathExpr)
                | RelativePathExpr /* xgc: leading-lone-slashXQ */
[72] RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*
[73] StepExpr ::= FilterExpr | AxisStep
[74] AxisStep ::= (ReverseStep | ForwardStep) PredicateList
[75] ForwardStep ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[76] ForwardAxis ::= ("child" "::")
                   | ("descendant" "::")
                   | ("attribute" "::")
                   | ("self" "::")
                   | ("descendant-or-self" "::")
                   | ("following-sibling" "::")
                   | ("following" "::")
[77] AbbrevForwardStep ::= "@"? NodeTest
[78] ReverseStep ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[79] ReverseAxis ::= ("parent" "::")
                   | ("ancestor" "::")
                   | ("preceding-sibling" "::")
                   | ("preceding" "::")
                   | ("ancestor-or-self" "::")
[80] AbbrevReverseStep ::= ".."
[81] NodeTest ::= KindTest | NameTest
[82] NameTest ::= QName | Wildcard
[83] Wildcard ::= "*"
                | (NCName ":" "*")
                | ("*" ":" NCName) /* ws: explicitXQ */
[84] FilterExpr ::= PrimaryExpr PredicateList
[85] PredicateList ::= Predicate*
[86] Predicate ::= "[" Expr "]"
[87] PrimaryExpr ::= Literal | VarRef | ParenthesizedExpr |
                   ContextItemExpr | FunctionCall | OrderedExpr |
                   UnorderedExpr | Constructor
[88] Literal ::= NumericLiteral | StringLiteral
[89] NumericLiteral ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
[90] VarRef ::= "\$" VarName
[91] VarName ::= QName
[92] ParenthesizedExpr ::= "(" Expr? ")"
[93] ContextItemExpr ::= "."
[94] OrderedExpr ::= "ordered" "{" Expr "}"
[95] UnorderedExpr ::= "unordered" "{" Expr "}"
[96] FunctionCall ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")"
                    /* xgc: reserved-function-namesXQ */
                    /* gn: parensXQ */
[97] Constructor ::= DirectConstructor | ComputedConstructor
[98] DirectConstructor ::= DirElemConstructor
                         | DirCommentConstructor
```

```
                               | DirPIConstructor
[99] DirElemConstructor ::= "<" QName DirAttributeList ("/>" | (">"
                                DirElemContent* "</" QName S? ">"))
                                /* ws: explicitXQ */
[100] DirAttributeList ::= (S (QName S? "=" S? DirAttributeValue)?)*
                           /* ws: explicitXQ */
[101] DirAttributeValue ::= ('"' (EscapeQuot | QuotAttrValueContent)* '"')
                            | ("'" (EscapeApos | AposAttrValueContent)* "'")
                            /* ws: explicitXQ */
[102] QuotAttrValueContent        ::= QuotAttrContentChar | CommonContent
[103] AposAttrValueContent ::= AposAttrContentChar
                                    | CommonContent
[104] DirElemContent ::= DirectConstructor
                                | CDataSection
                                | CommonContent
                                | ElementContentChar
[105] CommonContent ::= PredefinedEntityRef | CharRef | "{{" | "}}" |
                               EnclosedExpr
[106] DirCommentConstructor ::= "<!--" DirCommentContents "-->"
                                       /* ws: explicitXQ */
[107] DirCommentContents ::= ((Char - '-') | ('-' (Char - '-')))*
                             /* ws: explicitXQ */
[108] DirPIConstructor ::= "<?" PITarget (S DirPIContents)? "?>"
                           /* ws: explicitXQ */
[109] DirPIContents ::= (Char* - (Char* '?>' Char*))
                        /* ws: explicitXQ */
[110] CDataSection ::= "<![CDATA[" CDataSectionContents "]]>"
                       /* ws: explicitXQ */
[111] CDataSectionContents ::= (Char* - (Char* ']]>' Char*))
                                /* ws: explicitXQ */
[112] ComputedConstructor ::= CompDocConstructor
                               | CompElemConstructor
                               | CompAttrConstructor
                               | CompTextConstructor
                               | CompCommentConstructor
                               | CompPIConstructor
[113] CompDocConstructor ::= "document" "{" Expr "}"
[114] CompElemConstructor ::= "element" (QName | ("{" Expr "}")) "{"
                               ContentExpr? "}"
[115] ContentExpr ::= Expr
[116] CompAttrConstructor ::= "attribute" (QName | ("{" Expr "}")) "{"
                               Expr? "}"
[117] CompTextConstructor ::= "text" "{" Expr "}"
[118] CompCommentConstructor ::= "comment" "{" Expr "}"
[119] CompPIConstructor ::= "processing-instruction" (NCName | ("{" Expr
                            "}")) "{" Expr? "}"
[120] SingleType ::= AtomicType"?"?
[121] TypeDeclaration ::= "as" SequenceType
[122] SequenceType ::= ("empty-sequence" "(" ")")
                       | (ItemType OccurrenceIndicator?)
[123] OccurrenceIndicator ::= "?" | "*" | "+"
                               /* xgc: occurrence-indicatorsXQ */
[124] ItemType ::= KindTest | ("item" "(" ")") | AtomicType
[125] AtomicType ::= QName
[126] KindTest ::= DocumentTest
                 | ElementTest
                 | AttributeTest
                 | SchemaElementTest
                 | SchemaAttributeTest
```

```
                  | PITest
                  | CommentTest
                  | TextTest
                  | AnyKindTest
[127] AnyKindTest ::= "node" "(" ")"
[128] DocumentTest ::= "document-node" "(" (ElementTest |
                       SchemaElementTest)? ")"
[129] TextTest ::= "text" "(" ")"
[130] CommentTest ::= "comment" "(" ")"
[131] PITest ::= "processing-instruction" "(" (NCName | StringLiteral)? ")"
[132] AttributeTest ::= "attribute" "(" (AttribNameOrWildcard (","
                        TypeName)?)? ")"
[133] AttribNameOrWildcard ::= AttributeName | "*"
[134] SchemaAttributeTest ::= "schema-attribute" "(" AttributeDeclaration ")"
[135] AttributeDeclaration ::= AttributeName
[136] ElementTest ::= "element" "(" (ElementNameOrWildcard ("," TypeName
                     "?"?)?)? ")"
[137] ElementNameOrWildcard ::= ElementName | "*"
[138] SchemaElementTest ::= "schema-element" "(" ElementDeclaration ")"
[139] ElementDeclaration ::= ElementName
[140] AttributeName ::= QName
[141] ElementName ::= QName
[142] TypeName ::= QName
[143] URILiteral ::= StringLiteral
[144] FTSelection ::= FTOr FTPosFilter* ("weight" RangeExpr)?
[145] FTOr ::= FTAnd ( "ftor" FTAnd )*
[146] FTAnd ::= FTMildNot ( "ftand" FTMildNot )*
[147] FTMildNot ::= FTUnaryNot ( "not" "in" FTUnaryNot )*
[148] FTUnaryNot ::= ("ftnot")? FTPrimaryWithOptions
[149] FTPrimaryWithOptions ::= FTPrimary FTMatchOptions?
[150] FTPrimary ::= (FTWords FTTimes?) | ("(" FTSelection ")") |
                    FTExtensionSelection
[151] FTWords ::= FTWordsValue FTAnyallOption?
[152] FTWordsValue ::= Literal | ("{" Expr "}")
[153] FTExtensionSelection ::= Pragma+ "{" FTSelection? "}"
[154] FTAnyallOption ::= ("any" "word"?) | ("all" "words"?) | "phrase"
[155] FTTimes ::= "occurs" FTRange "times"
[156] FTRange ::= ("exactly" AdditiveExpr)
                | ("at" "least" AdditiveExpr)
                | ("at" "most" AdditiveExpr)
                | ("from" AdditiveExpr "to" AdditiveExpr)
[157] FTPosFilter ::= FTOrder | FTWindow | FTDistance | FTScope |
                      FTContent
[158] FTOrder ::= "ordered"
[159] FTWindow ::= "window" AdditiveExpr FTUnit
[160] FTDistance ::= "distance" FTRange FTUnit
[161] FTUnit ::= "words" | "sentences" | "paragraphs"
[162] FTScope ::= ("same" | "different") FTBigUnit
[163] FTBigUnit ::= "sentence" | "paragraph"
[164] FTContent ::= ("at" "start") | ("at" "end") | ("entire" "content")
[165] FTMatchOptions ::= FTMatchOption+       /* xgc: multiple-match-options */
[166] FTMatchOption ::= FTLanguageOption
                        | FTWildCardOption
                        | FTThesaurusOption
                        | FTStemOption
                        | FTCaseOption
                        | FTDiacriticsOption
                        | FTStopwordOption
                        | FTExtensionOption
```

```
[167] FTCaseOption ::= ("case" "insensitive")
                     | ("case" "sensitive")
                     | "lowercase"
                     | "uppercase"
[168] FTDiacriticsOption ::= ("diacritics" "insensitive")
                           | ("diacritics" "sensitive")
[169] FTStemOption ::= ("with" "stemming") | ("without" "stemming")
[170] FTThesaurusOption ::= ("with" "thesaurus" (FTThesaurusID
                          | "default")) | ("with" "thesaurus" "("
                          (FTThesaurusID | "default")
                          ("," FTThesaurusID)* ")") | ("without"
                           "thesaurus")
[171] FTThesaurusID ::= "at" URILiteral ("relationship" StringLiteral)?
                        (FTRange "levels")?
[172] FTStopwordOption ::= ("with" "stop" "words" FTRefOrList
                            FTInclExclStringLiteral*) | ("without" "stop"
                           "words") | ("with" "default" "stop" "words"
                           FTInclExclStringLiteral*)
[173] FTRefOrList ::= ("at" URILiteral)
                    | ("(" StringLiteral ("," StringLiteral)* ")")
[174] FTInclExclStringLiteral ::= ("union" | "except") FTRefOrList
[175] FTLanguageOption ::= "language" StringLiteral
[176] FTWildCardOption ::= ("with" "wildcards") | ("without" "wildcards")
[177] FTExtensionOption ::= "option" QName StringLiteral
[178] FTIgnoreOption ::= "without" "content" UnionExpr
```

Terminal Symbols

```
[179] IntegerLiteral ::= Digits
[180] DecimalLiteral ::= ("." Digits) | (Digits "." [0-9]*)
                         /* ws: explicitXQ */
[181] DoubleLiteral ::= (("." Digits) | (Digits ("." [0-9]*)?)) [eE] [+-]?
                        Digits /* ws: explicitXQ */
[182] StringLiteral ::= ('"' (PredefinedEntityRef | CharRef
                        | EscapeQuot | [^"&])* '"') | ("'" (PredefinedEntityRef
                        | CharRef | EscapeApos | [^'&])* "'")
                        /* ws: explicitXQ */
[183] PredefinedEntityRef ::= "&" ("lt" | "gt" | "amp" | "quot" | "apos")
                              ";" /* ws: explicitXQ */
[184] EscapeQuot ::= '""'
[185] EscapeApos ::= "''"
[186] ElementContentChar ::= Char - [{}<&]
[187] QuotAttrContentChar ::= Char - ["{}<&]
[188] AposAttrContentChar ::= Char - ['{}<&]
[189] Comment ::= "(:" (CommentContents | Comment)* ":)"
                  /* ws: explicitXQ */
                  /* gn: commentsXQ */
[190] PITarget ::= [http://www.w3.org/TR/REC-xml#NT-PITarget]
                   /* xgc: xml-versionXQ */
[191] CharRef ::= [http://www.w3.org/TR/REC-xml#NT-CharRef]
                  /* xgc: xml-versionXQ */
[192] QName ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]
                /* xgc: xml-versionXQ */
[193] NCName ::= [http://www.w3.org/TR/REC-xml-names/#NT-NCName]
                 /* xgc: xml-versionXQ */
[194] S ::= [http://www.w3.org/TR/REC-xml#NT-S] /* xgc: xml-version */
[195] Char ::= [http://www.w3.org/TR/REC-xml#NT-Char]
               /* xgc: xml-versionXQ */
```

Bibliography

[1] Andy Tripp . Manual tree walking is better than tree grammars. http://jazillian.com/articles/treewalkers.html. Online: 2008.02.06.

[2] Terence Parr . Translators should use tree grammars. http://antlr.org/article/1100569809276/use.tree.grammars.tml. Online: 2008.02.06.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Hardcover, 2006.

[4] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[5] Kjell Bratbergsengen. *Lagring og behandling av store datamengder*, chapter 11. Tapir akademisk forlag, 2003.

[6] David Brownell and David Megginson. SAX Project. http://www.saxproject.org/. Online: 2008.02.05.

[7] Sudarshan S. Chawathe. A Quick Introduction to Relational Algebra. http://www.umcs.maine.edu/~chaw/200609/mat500db/relnalg.pdf. Online: 2008.02.01.

[8] City University of New York. Database Managment Systems – Relational Algebra. http://cisnet.baruch.cuny.edu/holowczak/classes/3400/relationalalgebra/. Online: 2008.01.28.

[9] Dan Olteanu and Holger Meuss and Tim Furche and Francois Bry. XPath: Looking Forward. In *EDBT Workshop on XML Data Management (XMLDM)*, Prague, Czech Republic, 2002.

[10] Don Chamerlin, Jonathan Robie and Daniela Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf. Online: 2008.4.28.

[11] Dr Gordon Russel. Database eLearning. http://db.grussell.org/. Online: 2008.01.29.

[12] Emiran Curtmola et.al. GalaTex: An XML Full Text Search Engine. http://www.galaxquery.com/galatex/. Online: 2007.10.16.

[13] Jens Teubner. Pathfinder: Compiling XQuery for Execution on the Monet Database Engine. http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-76/teubner.pdf. Online: 2007.12.11.

[14] Michael Kay. XSLT and XPath Optimization. In *IDEAlliance XML 2002 Conference*, Baltimore, Maryland, USA, December 2002.

[15] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[16] Lester I. McCann. On Making Relational Division Comprehensible. `http://fie.engrng.pitt.edu/fie2003/papers/1057.pdf`. Online: 2008.01.29.

[17] Jim Melton. *Advanced SQL:1999-Understanding Objec-Relational and Other Advanced Features*. Morgan Kaufmann, 2002.

[18] Leon Moonen. Generating Robust Parsers using Island Grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, October 2001.

[19] Mads Nyborg and Andreas Ravnestad. Development of an XQuery Parser with Full-Text Extensions. Specialization project, Norwegian University of Science and Technology, December 2007, `http://printf.no/tmp/report.pdf`.

[20] Terrence Parr. *The Definitive ANTLR Reference*. The Pragmatic Bookshelf, 2007.

[21] Philippe Michiels. XQuery Optimization. In *Proceedings of the VLDB 2003 PhD Workshop*, "Berlin, Germany", September 2003.

[22] Pierre Genevés, Jean-Yves Vion-Dury. Logic-based XPath optimization. In *ACM sumposium on Document engineering*, Milwaukee, Wisconsin, USA, 2004.

[23] Michael Rys. An introduction to the xquery (and xpath 2.0) type system: The general concepts. `http://sqljunkies.com/WebLog/mrys/archive/2004/05/13/2480.aspx`. Online: 2008.05.12.

[24] The iAD Research Centre. iAd: Information Access Disruptions. `http://www.iad-centre.no/`. Online: 2007.12.13.

[25] the Pathfinder Team. Pathfinder: A Purely Relational XQuery Processor. `http://pathfinder-xquery.org/`. Online: 2007.12.13.

[26] Torsten Grust. Purely Relational FLWORs. In *ACM SIGMOD 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2005)*, Baltimore, MD, USA, June 2005.

[27] Torsten Grust, Jens Teubner, Manuel Mayr, Jan Rittinger, and Sherif Sakr. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *SIGMOD International Conference on Management of Data*, Beijing, China, June 2007.

[28] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue – XQuery: Fluent. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM 2004)*, Enschede, The Netherlands, June 2004.

[29] Wolfgang Meier. eXist fact sheet. `http://exist.sourceforge.net/facts.html`. Online: 2008.04.23.

[30] Wolfgang Meier. eXist XQuery documentation. `http://exist.sourceforge.net/xquery.html`. Online: 2007.11.12.

[31] Wolfgang Meier. Index-driven XQuery processing in the eXist XML database. http://exist.sourceforge.net/xmlprague06.html. Online: 2007.11.12.

[32] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/. Online: 2008.06.02.

[33] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 formal semantics. http://www.w3.org/TR/xquery-semantics/. Online: 2008.06.01.

[34] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text 1.0. http://www.w3.org/TR/xquery-full-text/. Online: 2008.06.02.

[35] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xquery-operators/. Online: 2008.06.02.

[36] World Wide Web Consortium. XQuery 1.0 Specification. http://www.w3.org/TR/xquery/. Online: 2008.06.02.