INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Ludwig
Maximilians
Universität
München

LMU

Automatic Translation Between XQuery and Xcerpt

Benedikt Linse

Diplomarbeit

Beginn der Arbeit:	01.08.2005
Abgabe der Arbeit:	31.01.2006
Betreuer:	Prof. Dr. François Bry, Tim Furche

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 31.01.2006 Benedikt Linse

Abstract

XQuery is a flexible, functional language for querying all kinds of XML data. As a W3C candidate recommendation it is likely to see widespread use and implemented in several commercial and open-source applications.

Xcerpt is a versatile, rule based query language for semistructured graph data in general and can be used to transform XML and RDF. In contrast to the navigational approach taken by XQuery, it uses patterns augmented by variables to extract information from documents.

This thesis investigates automatic translation possibilities among both languages and is organized in two parts. In the first part, it is shown how to translate sublanguages of Xcerpt query terms including multiple variables, negated and optional subterms, and entire queries to XQuery. Equally expressive sublanguages of XQuery are defined and automatic translation algorithms for both directions are specified and discussed.

The second part of this thesis discusses translation possiblities for expressions including construct parts. A translation algorithm for Xcerpt construct terms including grouping constructs and rules is provided. Furthermore, it is shown how to translate intertwined XQuery expressions to construct query rules. As an extension, it is demonstrated that XQuery expressions constructing intermediate results are translatable by either simulating dynamic environments in Xcerpt or by eliminating intermediate results before the translation algorithm is applied.

Zusammenfassung

XQuery ist eine flexible funktionale Anfragesprache für XML. Es steht kurz vor der Annahme als W3C Empfehlung, wird schon vielerorts eingesetzt und ist in einigen kommerziellen und quelloffenen Anwendungen implementiert.

Xcerpt ist eine vielseitige, regelbasierte Anfragesprache für semistrukturierte Graphdaten und kann zur Transformation von XML und RDF verwendet werden. Im Gegensatz zum navigationsbasierten Ansatz von XQuery verwendet Xcerpt mit Variablen angereicherte Muster, um Daten aus Dokumenten zu extrahieren.

Diese Diplomarbeit untersucht die Möglichkeiten der automatischen Übersetzung zwischen beiden Sprachen und gliedert sich in zwei Teile. Im ersten Teil wird gezeigt, wie Teilmengen von Xcerpts Anfragetermen mit mehreren Variablenvorkommen, negierten und optionalen Subtermen und schließlich Queries nach XQuery übersetzt werden können. In der Ausdrucksstärke übereinstimmende Teilsprachen von XQuery werden definiert und Algorithmen für beide Übersetzungsrichtungen werden angegeben und diskutiert.

Der zweite Teil der Arbeit beschäftigt sich mit der automatischen Übersetzung von Ausdrücken, die nicht nur Daten anfragen, sondern auch Ergebnisse konstruieren. Übersetzungsregeln für Xcerpts Gruppierungskonstrukte und ganze Regeln werden angegeben. Darüber hinaus werden solche XQuery Ausdrücke übersetzt, in denen Elementkonstruktion und Datenanfrage verflochten sind. Als Erweiterung werden zwei Möglichkeiten zur Übersetzung von Ausdrücken, die Zwischenergebnisse verwenden, aufgezeigt.

Acknowledgments

Many persons helped and supported me during the writing of this thesis. Without their efforts and patience, many interesting aspects could not have been included in this work. Above all, I would like to thank:

- *Prof. Dr. François Bry*, for giving me the chance to investigate this interesting and challenging topic, for making fruitful suggestions througout the work on this thesis, and for discussing the ideas I came up with.
- *Tim Furche*, who helped me to better understand both Xcerpt and XQuery, suggested the overall structure of this thesis, gave illuminative answers to all of my questions, and put this thesis to the acid test.
- *Andreas Schroeder*, for many intense discussions about the semantics and evaluation of Xcerpt programs and for proof-reading the entire work.

Contents

Chapter 1

Introduction

With the rise of XML[3] as a universal format for the interchange of information, the need to query and transform semistructured data has become increasingly important in the last years. Languages such as XSLT[9], XQuery[2] and Xcerpt[17, 6, 4] have been invented to simplify these tasks.

Studying the possiblities of automatically translating between XQuery and Xcerpt yields a couple of benefits. One of these benefits is that the differences and commonalities between both languages become apparent. Expressions that are straightforwardly translatable hint at parallelisms in the employed constructs, whereas expressions that are hard to translate indicate that the methods for solving a problem in both languages severly diverge. This leads to a better understanding of both languages. Studying the term complexity of the translation functions with respect to the length of the input gives further insight whether queries can be phrased as briefly as in the source language.

Another very important goal of automatically translating between XQuery and Xcerpt is to ease code migration. Translation schemes have been given for automatically translating between XQuery and XSLT for exactly this purpose. Code migration may not only be of interest to programmers switching from one language to the other, but also for having Xcerpt programs evaluated by an XQuery processor. Xcerpt being an academic research language, only prototypical implementations are available until today. Automatic translation of Xcerpt to XQuery instead of directly executing Xcerpt code bears the advantage that many of the needs common to all XML query languages (such as parsing and validating XML-documents, adressing specific nodes within a document fragment, filtering results based on predicates) do not need to be realized by Xcerpt itself.

Finally, insight on optimization possiblities may be gained by comparing the performance of queries and their translations. On the one hand, a lot of research has been conducted in the area of optimization of XQuery code[15, 16], and therefore, XQuery expressions should be expected to evaluate faster. On the other hand, Xcerpt is the more declarative language, which means that the programmer's intent is specified very explicitely, such that additional optimizations (e.g. the matrix method [18]) may be applied.

1.1 A Comparison of the XQuery and Xcerpt Data Models

The formal semantics of XQuery [11] is neither directly defined on XML files, nor on the XML Information Set [10], but on the XQuery and XPath data model [12], which can be constructed from the former two. Likewise, the semantics of Xcerpt [17, chapter 8] is not defined on XML files, but on so called data terms, which are the input and output of Xcerpt programs. The advantage of this abstraction in both languages is that the data does not necessarily need to originate from XML-files, but can be generated from relational databases, or be the output of other queries. In order to be able to compare and translate between both languages it is necessary to first compare both data models, and define when the data in the different data models is considered to be equal.

1.1.1 Outline of this Thesis

This thesis is structured in six chapters. In this first chapter both Xcerpt and XQuery and their respective data models are briefly described. In the following three chapters, three sublanguages of Xcerpt XC_1, XC_2 and XC_3 as well as three corresponding sublanguages XQ_1, XQ_2 and XQ_3 are defined, and and automatic translation between these sublanguages is discussed. The sublanguages are defined such that XC_2 comprises

XC_1 and XC_3 comprises XC_2 (the same holds for XQ_1, XQ_2 and XQ_3). All of these sublanguages are restricted to querying XML data. Chapter 5 is concerned with the translation of expressions that also construct results. Entire Xcerpt rules are translated to XQuery and XQuery expressions that intertwine construction and query parts are translated to Xcerpt. Chapter 6 presents some ideas for future research in this field, and summarizes the results.

1.1.2 The Xcerpt Data Model

Xcerpt distinguishes between three types of terms: Data terms, query terms and construct terms. Data terms represent data to be queried and are always completely specified, whereas query terms are matched against data terms (and also construct terms) and may therefore feature a pattern-like, incomplete structure. The various kinds of incompleteness within Xcerpt query terms as well as construct terms are introduced later on in this introduction. Since the focus of this section lies on the Xcerpt data model, only data terms are introduced here. The syntax of Xcerpt data terms is given in [17, section 4.2] and is reproduced in Table 1.1.

Table 1.1: The syntax of Xcerpt data terms given in Backus-Naur-form

```
1:  <data-term>      ::= ( oid '@' )?  <ns-label> <list> .
2:  <ns-label>       ::= (<ns-prefix> ':')?  label .
3:  <ns-prefix>      ::= label | '"' iri '"' .
4:  <list>           ::= <ordered-list> | <unordered-list> .
5:  <ordered-list>   ::= '[' <attributes>?  <data-subterms>?  ']'  .
6:  <unordered-list>::= '{' <attributes>?  <data-subterms>?  '}' .
7:  <data-subterms>  ::= <data-subterm> ( ',' <data-subterm> )*
8:  <data-subterm>   ::= <data-term> | '"' string '"' | number | '^'oid .
9:  <attributes>     ::= 'attributes' '{'<attribute> (',' <attribute>)* '}'.
10: <attribute>      ::= <ns-label> '{' '"' string '"' '}' .
```

There are two features of Xcerpt data terms that cannot be directly represented in XML: unordered lists (line six in Table 1.1), and Xcerpt references (line one and eight). Both of these issues are briefly discussed in the sequel.

While the order of elements in an XML document is always given by the document order, siblings in the Xcerpt data model can be unordered, which is indicated by curly braces. The idea behind differentiating between ordered and unordered data becomes apparent when querying this data. An ordered Xcerpt query term can never match an unordered Xcerpt data term, because the intended semantics of an unordered data term is to give no guarantee about the order of its children, while the intended semantics of an ordered query term is to retrieve only data for which the order is guaranteed by square brackets. A possible approach of carrying over this semantics would be to introduce a special attribute (e.g `xc:ordered`[1]) which may be set to false to indicate that the order of the children of the node is not ensured. In order to translate Xcerpt programs that also operate on unordered data terms, the data terms could be translated to XML as usual, including the additional `xc:ordered='false'` attribute-value pair for the translations of curly braces as specified by the rule below. Due to more interesting aspects in translating between Xcerpt and XQuery, this approach is not further investigated in this thesis.

$[\![$ `a[ b{ c, d } ]` $]\!]_{toXML}$ ==[2]

```
<a xmlns:xc='http://www.pms.ifi.lmu.de/xcerpt2xquery'>
  <b xc:ordered='false'><c/><d/></b>
</a>
```

The second outstanding difference between the Xcerpt and XQuery data model is that Xcerpt terms may not only be trees, but also graphs. This means that beyond the traditional XML reference mechanism through `id` and `idref` attributes, Xcerpt provides a different referencing mechanism that is used to represent true

[1] The namespace prefix `xc` is associated with `http://www.pms.ifi.lmu.de/xcerpt2xquery` and distinguishes metadata used for the translation of Xcerpt to XQuery (and for the reverse direction) throughout this thesis.

[2] The notation $[\![\ldots]\!]_{toXML}$ == denotes the translation of Xcerpt data terms to an XML representation that may be queried by the XQuery translations of Xcerpt expressions.

graph structures. In the data term below, the reference `^oid1` within the element `b` results in `b` having two child nodes – one named `c` and the other one being the only child element of element `c`. It is important to note that there is no semantic difference between parent-child relationships represented by references and by nesting of subterms in Xcerpt.

```
a[ b[ c[ oid1@ d[] ], ^oid1 ] ]
```

A fully-fledged translation of Xcerpt to XQuery would require an XML representation for Xcerpt references. In XML, the data term above might be represented as follows:

```
<a xmlns:xc='http://www.pms.ifi.lmu.de/xcerpt2xquery'>
   <b><c><d xc:oid='oid1'></c><xc:reference xc:oid='oid1' /></b>
</a>
```

Prescinding from Xcerpt references and unordered siblings, all Xcerpt data terms have canonical XML representations. This holds true also for attributes, despite the syntactical differences. As exhibited in Table 1.1, attributes are enclosed in an extra `attributes`-element within the element they belong to. To reflect the absence of order among attributes curly braces are used in line nine of the grammar productions. Also in the XQuery data model, the order of attributes is not important – a fact that is of interest when comparing the `fn:deep-equal`-function with the Xcerpt way of establishing value based equality in Section 3.4.2.

Representing attributes as flat elements within the special `attributes`-element recently has been given up in favor of a syntax closer to the XML representation. This new Xcerpt syntax (specified in [6]) encloses attributes in either double (query terms only) or single parentheses. Each attribute is represented by a key value pair as in Table 1.2.

Table 1.2: An example Xcerpt data term using the new Xcerpt syntax

```
books:book ( language='english', price='215.39 Euros' ) [
  books:title [ 'Automatic translation between XQuery and Xcerpt' ]
  books:year [ '2006' ]
]
```

1.1.3 The XQuery Data Model

This section answers the following questions:

1. What is the XQuery data model, and why is it important for this thesis?
2. How are XQuery values accessed in the XQuery formal semantics?
3. Which parts of the XQuery data model are relevant for this thesis, and which can be left out?
4. The XQuery data model includes type annotations. Xcerpt gets by without type annotations. Is it still possible to translate between both data models?
5. How are data terms in the Xcerpt data model – e.g. the data term from the last section – translated into the XQuery data model?

While the XML Information Set [10] represents solely XML documents, the XQuery and XPath data model represents "all permissible values of expressions in the XSLT, XQuery, and XPath languages" [12]. The importance of the data model for this thesis stems from the fact that in the XQuery formal semantics the semantics of many constructs is defined in terms of the data model. As an example consider the rule which specifies the semantics of the application of the `child::` axis to a node value.

$$\text{dynEnv} \vdash \textbf{axis}\ \text{child::}\ \text{of element ElementName}\ \{\text{AttributeValue, ElementValue}\} \Rightarrow \text{ElementValue}$$

Rule 1.1 simply states, that applying the `child::` axis on an arbitrary element – represented by `element ElementName {AttributeValue, ElementValue}` in the XQuery data model – yields the value `ElementValue`. No premises need to be fulfilled for this rule to be applicable. `AttributeValue` and `ElementValue` are so-called properties of the element with name `ElementName`. The syntax of formal values like `element ElementName {AttributeValue, ElementValue}` is specified in the XQuery formal semantics [11, Section 2.3.1] and reproduced in Table 1.3. These formal values are the means for accessing the XQuery data model in the formal semantics.

Table 1.3: XQuery formal values productions

```
 1: Value              ::= Item | (Value ',' Value) | ('(' ')') .
 2: Item               ::= NodeValue | AtomicValue .
 3: AtomicValue        ::= AtomicValueContent TypeAnnotation?  .
 4: AtomicValueContent ::= String | Boolean | Decimal | Float | Double | Duration |
 5:                        DateTime | Time | Date | GYearMonth | GYear |
 6:                        GMonthDay | GDay | GMonth | HexBinary | Base64Binary |
 7:                        AnyURI | expanded-QName | NOTATION .
 8: TypeAnnotation     ::= 'of type' TypeName .
 9: ElementValue       ::= 'element' ElementName 'nilled'?
10:                        TypeAnnotation?  '{' Value '}' '{' NamespaceBindings '}' .
11: AttributeValue     ::= 'attribute' AttributeName TypeAnnotation? '{'SimpleValue'}' .
12: SimpleValue        ::= AtomicValue | (SimpleValue ',' SimpleValue) | ('(' ')') .
13: DocumentValue      ::= 'document' '{' Value '}' .
14: CommentValue       ::= 'comment' '{' String '}' .
15: ProcInstValue      ::= 'processing-instruction' NCName '{' String '}' .
16: TextValue          ::= 'text' '{' String '}' .
17: NodeValue          ::= ElementValue | AttributeValue | DocumentValue | TextValue |
18:                        CommentValue | ProcInstValue .
19: ElementName        ::= QName .
20: AttributeName      ::= QName .
21: TypeName           ::= QName .
22: NamespaceBindings  ::= NamespaceBinding (',' NamespaceBinding)* .
23: NamespacBinding    ::= 'namespace' NCName '{' String '}' .
```

As can be seen in the first line of Table 1.3, all XQuery values are a sequence of Items. This is a fundamental difference to Xcerpt, where sequences exist only as lists of subterms of a term. Therefore, it is not entirely possible to translate an XQuery expression like `(1,2,3)` to Xcerpt. As soon as this expression is wrapped by an element constructor `element result { (1,2,3) }`, the value can also be represented in the Xcerpt data model and would read `result[ 1, 2, 3 ]`.

Another striking difference between both data models is that while XQuery supports the concept of an explicit *document node* (see line 13), Xcerpt only provides the possibilty of reading and writing documents with the constructs `in` and `out`. Therefore, the simple query `fn:doc('tags.xml')`[3] which returns a document node is not translatable to Xcerpt, whereas the query `fn:doc('tags.xml')/child::*` is very well translatable to `in { resource [ 'file:tags.xml' ], var X }`. An additional document node could be easily introduced either in Xcerpt itself (and is likely to be included in the near future), or for the purpose of translation between XQuery and Xcerpt, a new `xc:document` element could be employed to reflect the usage of document nodes in the XQuery data model. Since document nodes are of no importance in this thesis, these methods are not further investigated nor used.

XQuery supports static typing and distinguishes between a wide range of data types. Therefore AtomicValueContent (line 4), which denotes the value space that any attribute or content of a an element of type `xsd:simpleType` may ever adopt, draws from a long list of XML-Schema types. For the purposes of this thesis, it is sufficient to restrict `AtomicValueContent` to Strings. While XQuery supports static type checking, it can also operate on untyped data. Although a type system for Xcerpt is under development, Xcerpt still operates on untyped data. This leads to the most important simplification of the above production rules that still allows to represent any untyped value in the XQuery data model and answers question 4 above: Type

[3]The namespace prefix `fn` is mapped to http://www.w3.org/2005/xpath-functions and is used for predefined XQuery functions as specified in [14]

annotations (line 8) are not taken into account. To be more precise, it is assumed that the expressions that are translated in this thesis operate on non-validated XML.

In the XQuery data model, elements have an optional 'nilled' marker (line 9). "This marker can only be present if the element has been validated against an element type in the schema which is 'nillable', and the element has no content and an attribute `xsi:nil` set to 'true'." [11, 2.3.1 Formal values] In other words, the usage of the attribute `xsi:nil` in an XML instance document in combination with the attribute `nilled` in the corresponding schema, is just a way to say that some information has been deliberately left away. Since schema validation is not of interest in this thesis, the `xsi:nil` attribute must be handled just as any other attribute when querying data with Xcerpt and XQuery.

Similarly, comments and processing instructions (lines 14 and 15) are not of any particular interest to this thesis. Consequently, the constructs to be translated do not operate on these items, though they could be easily added to the translation if required. Namespace bindings (line 23) are only used to distinguish meta-data introduced during the translation process. The expressions themselves do not query or construct namespace bindings. They do, however, query and return data including namespace prefixes. As a simplification to the translation process, the assumption is made that equivalent namespace prefixes are bound to the same namespaces in the context of an expression in Xcerpt and its translation in XQuery, or vice versa.

A final important characteristic of the XQuery data model, which cannot be derived from Table 1.3, is that every single node that is read from an input resource or constructed during the evaluation of an expression is assigned a unique node identifier. While this is certainly useful (e.g. to find out whether two variables are bound to the same node – see Section 3.4.3 for details), it also causes problems for referential transparency as exposed in Section 5.4.3. Since Xcerpt does not assign node identifiers and therefore cannot distinguish between structurally identical (value-equal) data terms that have been extracted from different nodes of the input document, certain XQuery expressions are hard to translate to Xcerpt. An obvious (partial) solution to this problem is to preprocess all XML documents to be queried and assign unique node identifiers to each element node. In this manner, the XML value

```
<a><b><c/></b></a>
```

is preprocessed to read:

```
a(xc:id='1')[ b(xc:id='2')[ c(xc:id='3')[] ] ]
```

Obviously, this method cannot be applied for attribute nodes. The Xcerpt translations of XQuery expressions considered in this thesis get by without comparing the origin of values. Thus, methods for augmenting XML with node identifiers are not further investigated.

In the XQuery data model, the example Xcerpt data term depicted in Table 1.2 would be formulated as follows.

Table 1.4: An example for a node value in the XQuery data model

```
element books:book {
  attribute language { 'english' },
  attribute price { '215.39 Euros' },
  element books:title { 'Automatic translation between XQuery and Xcerpt' }
  element books:year { '2006' }
}
```

The conclusion of this section is that both data models are very similar and that despite the obvious syntactic differences in representation, it is easy to see when the results of an XQuery and an Xcerpt expression coincide. By enforcing the mentioned restrictions (only tree-like, ordered Xcerpt data terms and only untyped XQuery formal values) it is possible to ensure that the results of the translations are comparable.

1.2 Fundamentals of XPath and XQuery

XQuery [2] is a functional XML query language developed by the XML Working group of the W3C. Its development is closely coordinated with that of XSLT 2.0 [9] and with that of XPath 2.0 [1], which is a subset

of both XSLT and XQuery 1.0 and is used also in XPointer and XML Schema. XPath allows to navigate to specific nodes within an XML document by specifying their paths from the root node or any other previously determined node, and by stating conditions (so-called predicates) about the node itself, or any related node.

Navigation is possible over a set of axes, the most important among them being the *child axis* to be used for selecting child nodes, the *parent axis* for selecting the parent node, and the *attribute axis* for selecting attributes of the current context node. An example XPath expression selecting all new `book` elements (that is only those `book`-elements with an attribute-value-pair `new='true'`) within a `library` element, for which an `author` subelement is given, would be `//library/book[author][@new='true']`.

Three different kinds of axes are used within this example: the *child axis* is always assumed when no other axis has been specified – such as in `book`, which could also be written `child::book` –, the *attribute axis* is abbreviated by the symbol '@' (therefore `@new` could also be written `attribute::new`) and `//` is a shorthand for `/descendant-or-self::node()/` axis.

Subexpressions within square brackets (such as `[author]`) are called predicates and serve to filter the result of the expression they are appended to. Multiple predicates may be appended to each other.

The example demonstrates that XPath provides a concise and easy way to extract sequences of nodes from an XML document. On the other hand, using XPath alone, it is not possible to transform one document into another document, iterate over the values of a sequence, construct new element nodes that were not present in the input document, or bind the result of path expressions to variables in order to perform further computations. This functionality is provided by the 'enclosing' languages XQuery and XSLT[4]. XQuery being an offspring of the database community, one of its main constructs, the *FLWOR*-expressions[5], resembles closely `Select-From-Where` clauses in SQL, and provides all of the above stated features. An example taken from [2, 3.8 FLWOR Expressions] is depicted in Table 1.5.

Table 1.5: An example FLWOR expressions

```
for $d in fn:doc('depts.xml')/depts/deptno
let $e := fn:doc('emps.xml')/emps/emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
   <big-dept> {
      $d,
      <headcount>{fn:count($e)}</headcount>,
      <avgsal>{fn:avg($e/salary)}</avgsal> }
   </big-dept>
```

New element nodes such as `<big-dept>` are constructed in the example expressions using XML-notation. The results of XPath expressions are bound to variables `$d` and `$e` for being processed later on. Calls to the functions `fn:count` and `fn:avg` aggregate the sequences their arguments evaluate to, and the `for`-clause iterates over all department numbers addressed by the path expression `/depts/deptno`. Note that querying data and constructing results in XQuery is not required to be separate as one could believe when looking at Table 1.5. Sometimes it may be beneficial to intertwine construction and query parts, e.g. to wrap all results computed by the example expression in one single `result` element.

1.3 Fundamentals of Xcerpt

Xcerpt is a versatile, declarative language for querying all kinds of semistructured data. Contrasting the navigational approach of XPath and XQuery where each path expressions yields at most one variable, Xcerpt takes a positional one, returning multiple variable bindings for patterns representing trees (and sometimes even graphs). These patterns, which are used to extract variable bindings from semistructured data are called query terms and are discussed in Section 1.3.1.

A further difference between Xcerpt and XQuery is the clear separation of construct and query parts in Xcerpt programs. This is achieved by the introduction of substitution sets, which may be considered as an

[4] An interesting discussion about whether XQuery is a reinvention of XSLT can be found on http://lists.xml.org/archives/xml-dev/200102/threads.html#00483, but XSLT is of no major interest in this thesis

[5] FLWOR is an acronym for `for`, `let`, `where`, `order by` and `return`.

interface between the querying and construction of data. Substitution sets are generated by matching query terms with semistructured data and consumed by their application to so-called *construct terms* (which are treated in Section 1.3.2). Construct and query terms are connected via Xcerpt *construct-query-rules* (see Section 1.3.4). Several construct query rules make up an Xcerpt program and are evaluated using forward or – similar to Prolog programs – backward chaining (Section 1.3.4).

1.3.1 Xcerpt Query Terms

Xcerpt query terms are patterns that are to be unified with the queried data. In the context of querying the Web, query terms must be able to handle incompleteness of patterns with respect to the actual input documents in various ways. Queries may either be complete or incomplete with respect to order, in depth, and also in breadth. In many cases it is even beneficial to extract data whenever it is present, without causing the query to fail if it is missing. This last requirement is realized by using *optional query terms*.

1.3.1.1 Incompleteness with Respect to Order

Semistructured data may either be explicitly ordered or unordered. In the data model of Xcerpt this is specified by either curly braces denoting incompleteness with respect to order or by square brackets for completeness with respect to order. Since XML does not differentiate between ordered and unordered data, the example data term in Table 1.2 is specified with square brackets, reflecting the implicitly given document order of the corresponding XML data. Also query terms may be complete or incomplete with respect to order, which is expressed by the same syntax. Query terms that are incomplete with respect to order may match data that is inherently ordered, but query terms that are complete with respect to order cannot be matched with data terms that are incomplete with respect to order. Naturally, if both the data and the query term feature the same order specification, they may match. Some canonical query terms, that would match with the example data in Table 1.2 are the data term itself (interpreted as a query term), or any query term that can be derived from it by substituting pairs of square brackets by pairs of curly braces. Neglecting incompleteness in breadth and in depth, optional query terms and variables, these are also the only query terms we can formulate so far that would match with the data.

1.3.1.2 Incompleteness in Breadth and in Depth

Usually, the exact structure of data on the Web is unknown to the query author. Nevertheless, it should be possible to issue queries that require the existence of only certain subterms. While data terms are not allowed to be incomplete in breadth, query terms are, which is denoted by double braces or brackets. An example query term that would match with the data in Table 1.2 would be $Expr^{a}_{1.3.1.2}$. It would also match any `books:book` elements that include additional `author`-elements, or ones with `title`-elements that include child elements for translations to foreign languages.

$Expr^{a}_{1.3.1.2} :=$ `books:book [[ books:title [[ ]] ]]`

Not only the siblings of certain subterms of a query term may be of no particular significance, also its ancestors may be unknown or irrelevant to the query. In XPath such queries are formulated using the `descendant`-axis, and there exists a very similar construct in Xcerpt, which is called `desc` and may precede query terms, indicating that they need not be a child of their enclosing query term, but may occur at arbitrary depth. This could be used to select all titles of a bookstore in XML representation:

$Expr^{b}_{1.3.1.2} :=$ `desc books:title [[ ]]`

1.3.1.3 Xcerpt Variables and Substitution Sets

Up to now, the only information gained by the evaluation of a query term was whether it matches with the data or not. In other words, with the language constructs introduced thus far, only boolean queries on the structure and content of data terms are feasible. Xcerpt variables may be included in query terms to extract more information, i.e. arbitrary parts of the data. Expression $Expr_{1.3.1.3}$ queries the title and the entire nodes representing the year of a `books:book` element.

$Expr_{1.3.1.3}$:=
```
books:book [[
  books:title [[ var Title ]],
  var Year as books:year [[ ]]
]]
```

To be exact, two different kinds of variables are used in $Expr_{1.3.1.3}$. Bindings for variable `Year` having to fulfill the pattern `books:year [[ ]]`, `Year` is called a variable with *pattern restriction*. In contrast, the bindings for variable `Title` only need to appear at the right position within the data.

The answer to such queries is given in form of a *substitution set*, which are – as the name suggests – sets of substitutions. A substitution is a mapping from the set of all Xcerpt variables to all data terms (to be more precise, variables may also be mapped to construct terms, but this is not important for the moment). By default, all variables are mapped to themselves, only those variables appearing in the query term are mapped to data terms. Of course, the written representation of substitutions is restricted to variables which are not mapped to themselves. The substitution set generated by the evaluation of expression $Expr_{1.3.1.3}$ with respect to the data in Table 1.2 consists of only one single substitution:

```
{ { Title -> title [ 'Automatic translation between Xcerpt and XQuery' ],
    Year -> '2006' } }
```

In general, substitution sets consist of more than one substitution, but the same substitution may not appear twice within a substitution set. Two substitutions are considered equal, if their variables are mapped to the same data term values. By introducing node identity, also other definitions of substitution equality and for substitution sets are conceivable, paving the way for the translation of certain XQuery expressions to Xcerpt, which would be harder to translate otherwise (see Section 5.1).

1.3.1.4 Negated and optional Query Terms

In the same way as predicates within XPath may preclude paths that contain a certain element or attribute value (such as in `/books:book[fn:not(title)]/author`), Xcerpt provides the keyword `without` for ensuring the absence of certain subterms.

```
books:book {{ var Author as author {{ }}, without title {{ }} }}
```

Subterm negation is not the only necessity that arises from the heterogeneous nature of data on the web. Retrieving certain data fragments in the case that they are available, while not requiring them to be present is another issue. Although it would certainly be possible to issue multiple queries covering all possible structures, a more elegant solution is provided by Xcerpt's `optional` keyword. Given a series of `books:book` data terms, some of which include author subelements, the following query could be used to extract author names whenever possible.

```
books:book {{
   var Title as books:title {{ }},
   optional var Author as books:author {{ }}
}}
```

Applied to the data in Table 1.2, the optional `books:author` subterm would not match, but the entire query term would neither fail. The single generated substitution would be { `Title -> title [ 'Automatic ...' ]` }.

The expressiveness of Xcerpt query terms exceeds the possibilities presented here by far. Just to name some extensions, it is possible to match regular expressions, include position specifications for subterms, make use of label and namespace variables, and even give multiple arguments to the keywords `without` and `optional`. Furthermore, query terms may be connected by the boolean operators `and`, `or` and `not` to form so-called *queries*, which may be associated with resources (e.g XML-documents). For a complete description of the language see [17] and [6].

1.3.2 Xcerpt Construct Terms

Having introduced query terms and the substitution sets that are calculated by their application to data terms, the foundations are laid for the introduction of construct terms. As mentioned above, substitution sets may be

considered as an interface between query and construct terms, and they are consumed by their application to construct terms. As one might guess, variable occurrences within construct terms are replaced by the values to which they are mapped in a substitution.

When a substitution set is applied to a construct term, the result is a set of data terms (the result might as well be a set of construct terms, but this case is neglectable in this thesis). The application of the substitution set of Section 1.3.1.3 to the construct term $Expr^a_{1.3.2}$ yields the data term $Expr^b_{1.3.2}$.

$Expr^a_{1.3.2} :=$ `publication [ var Year, var Title ]`

$Expr^b_{1.3.2} :=$ `publication [ '2006', title [ 'Automatic ...' ] ]`

Of course, there is much more one can achieve by using construct terms than simple renaming of tags and changing the order of siblings, as it was the case in the previous example.

1.3.3 Xcerpt Grouping Constructs

A salient aspect of construct terms are the powerful grouping constructs that may be employed. To see their full power, consider a more extensive substitution set:

```
{  { Sport -> 'Soccer', C -> country ['Germany'], Skills -> 'Good' },
   { Sport -> 'Soccer', C -> country ['France'], Skills -> 'Good' },
   { Sport -> 'Soccer', C -> country ['US'], Skills -> 'Suboptimal' },
   { Sport -> 'Soccer', C -> country ['England'], Skills -> 'Good' },
   { Sport -> 'Cricket', C -> country ['England'], Skills -> 'Perfect'},
   { Sport -> 'Football', C -> country ['US'], Skills -> 'Perfect'} }
```

In order to get a listing of countries that exercise a particular type of sport, one could apply the above substitution set to the construct term below. It encloses all sports together with a list of countries practicing that sport in a view called `sports_and_countries`. The result of this operation is given by $Expr^a_{1.3.3}$.

```
sports_and_countries [ all sport [ var Sport, all var C ] ]
```

$Expr^a_{1.3.3} :=$

```
sports_and_countries [
  sport [ 'Soccer', country [ 'Germany' ], country [ 'France' ],
          country [ 'US' ], country [ 'England' ] ],
  sport [ 'Cricket', country [ 'England' ] ],
  sport [ 'Football', country [ 'US' ] ]
]
```

Of course, this is not the only possiblity to process the substitution set. One might just as well be interested in the types of sports that are popular in particular countries. An appropriate construct term to materialize this view upon the data would be the construct term below that generates the data in $Expr^b_{1.3.3}$.

```
countries_and_sports [ all sportlist [ var C, all var Sport] ]
```

$Expr^b_{1.3.3} :=$

```
countries_and_sports [
  sportlist [ country [ 'Germany' ], 'Soccer' ]
  sportlist [ country [ 'France' ], 'Soccer' ]
  sportlist [ country [ 'England' ], 'Soccer', 'Cricket' ]
  sportlist [ country [ 'US' ], 'Soccer', 'Football' ]
]
```

Apart from the construct terms presented above, it might also be beneficial to find out the skills of a country in all sports it practices, rank all soccer-playing nations according to their skills, etc. All this is possible by using Xcerpt's grouping constructs `all`, `some` and the additional clauses `group by` and `order by`. For a complete description of these language elements see [17][Section 4.6.2].

1.3.4 Xcerpt Rules and Programs

Rules connect query and construct terms with each other, thereby determining to which data terms substitution sets are to be applied. They are of the following generic form:

```
CONSTRUCT
  <CONSTRUCT TERM>
FROM
  <QUERY>
END
```

As mentioned above, queries are possibly nested conjunctions, disjunctions and negations of query terms which may also be associated with a specific resource to be queried. Queries not associated with any resource are evaluated on the results of other rules.

Xcerpt programs are sets of construct-query-rules and are evaluated by rule chaining. Besides the structure given above, rules may also start out with the keyword `GOAL` instead of `CONSTRUCT`. The difference between these two forms is that data terms produced by rules with the keyword `GOAL` are considered results of the program they belong to, whereas data terms resulting from the evaluation of rules beginning with `CONSTRUCT` are only intermediate results.

The wording in the last paragraph assumes a forward chaining evaluation of Xcerpt programs. As in Prolog, Xcerpt programs may also be evaluated using backward chaining, and this is the method that was chosen for the current prototype of the language.

Chapter 2

Translating simple XPath Expressions

In this section a first mapping between the limited sublanguages XQ_1 and XC_1 of XQuery and Xcerpt respectively is given. First an example expression is considered (Section 2.1). Then the syntaxes and semantics of both sublanguages is introduced (Sections 2.2 and 2.3). Section 2.4 concludes the discussion of XQ_1 and XC_1 by providing translation rules between both languages and a formal proof of equivalence for the examined expressions.

2.1 An Example XPath Expression and its Xcerpt Equivalent

XPath being a part of XQuery, the formal semantics of both languages are treated together in the W3C document "XQuery 1.0 and XPath 2.0 Formal Semantics" [11]. But not every XPath or XQuery expression is directly associated with its semantics. The predominant part of both languages is normalized to the so-called XQuery core. Not even the familiar path expressions belong to the XQuery core, but are normalized to `for`-clauses and step-expressions. As an example consider the simple query `/tag1/tag2/tag3`. It is called a *composite* path expression, because it contains an intermediate '/'. Composite path expressions are normalized by the following normalization rule (among others).

(Rule 2.1) [StepExpr / RelativePathExpr]$_{Expr}$ ==
fs:apply-ordering-mode(fs:distinct-doc-order-or-atomic-sequence(
let $fs:sequence as node()* := [StepExpr]$_{Expr}$ return
let $fs:last := fn:count($fs:sequence) return
for $fs:dot at $fs:position in $fs:sequence return
[RelativePathExpr]$_{Expr}$
))

Since this rule is not quite self-explanatory, a brief illustration follows:

- A *step expression* (abbreviated StepExpr in the formula above) e.g. `child::tag1` is made up of an *axis* specification such as `child::` and a *node test*. In the example `child::tag1` the node test checks the label of the context node. The child axis is assumed as the default axis, and hence it may also be omitted. Step expressions are treated more thoroughly later in this chapter.
- A *relative path expression* is a sequence of step expressions concatenated by '/' or '//'. Applying the above rule to `tag1/tag2/tag3`, the step expression StepExpr would be `tag1` and the relative path expression RelativePathExpr would be `tag2/tag3`.
- The subscript $Expr$ indicates that this normalization rule is used to map top-level expressions to the XQuery core, differentiating it from other families of normalization rules such as normalization rules for function calls, for sequence types, and for axes.
- The behavior of the function fs:apply-ordering-mode() depends on the value of the global ordering mode. The ordering mode can be set to 'ordered' or 'unordered' by an XQuery programmer. If it is set to 'ordered', the function is equivalent to the identity function, because its input sequence is already given in document order. Otherwise, the order of the result of fs:apply-ordering-mode() is implementation dependent, meaning that no guarantee is given for the order of the results. The default

behaviour of Xcerpt being not to give a guarantee about the order of results, it is easier to translate XQuery expressions with the ordering mode set to unordered. This is assumed in the rest of this chapter, and translations are considered correct if they produce the same set of results – no matter if the order coincides.

- The `fs:distinct-doc-order-or-atomic-sequence()` function takes either a sequence of only nodes, sorts them by document order and removes duplicates based on node identity, or it takes a sequence of atomic values and returns it unchanged. Since the order of the results is insgnificant in this chapter, neither of the last two functions are made use of in the XQuery expressions to be translated to Xcerpt.

- Within XQuery step expressions the size of the current context may be retrieved by a call to the function `fn:last()`, and the context position by `fn:position()`. These functions rely upon the formal variables `$fs:last` and `$fs:position` being added to the *dynamic environment* of XQuery, which is illustrated by the rule above. Since neither of the two functions are important in this chapter, the binding of both variables is insignificant for the moment.

- Finally, the formal variable `$fs:dot` denotes the current context item. Just as `$fs:position` and `$fs:last`, `$fs:dot` is not an ordinary XQuery variable, and therefore does usually not occur within XQuery programs. It rather serves to specifiy the formal semantics of XQuery and is used internally to remember the context item. In order to stay close to the formal semantics of XQuery, `$fs:dot` is used in this chapter. Later in this thesis, this approach is abandoned in favor of including more constructs not part of the XQuery core, which results in more concise and readable XQuery expressions.

Rule 2.1 is not the only normalization rule that needs to be applied to transform `/tag1/tag2/tag3` to the XQuery core, but it is the most important one. Other normalization rules are necessary to treat the leading '/' and the step expressions, and they are formally defined in [11]. An XQuery core expression equivalent (except for the order of the results) to `/tag1/tag2/tag3` is depicted in Table 2.1.

Table 2.1: $Expr_{2.1}^{XQ}$:An example expression in XQ_1

```
for $fs:dot in
  for $fs:dot in
    let $fs:dot := fn:doc('bib.xml')
    return child::tag1
  return child::tag2
return child::tag3
```

What would be the Xcerpt equivalent to the query in Table 2.1? Due to the fact that $Expr_{2.1}^{XQ}$ both queries XML data and returns a result, the Xcerpt equivalent can neither be a pure query term nor a pure data term. It must be a construct-query rule (see [17, Section 4.7]). The values that $Expr_{2.1}^{XQ}$ returns in its final `return`-clause `child::tag3` must be bound to a variable - say `var X` - in the query part of the rule, and included in the construct part. Since all bindings of `X` shall be returned, a grouping construct is used in the translation to collect them. Unlike its traditional semantics, in this thesis it is assumed that `all` does not perform value-based duplicate elimination. In Chapter 5 grouping constructs are discussed in more detail, and a new grouping construct named `all-distinct` is introduced to take over the original role of `all`. Grouping constructs in Xcerpt must always be enclosed by some kind of term, and therefore the special label `xc:result` is used to enclose the term `all var X` in the translation. The namespace prefix `xc` serves to distinguish this meta-data from the actual result.

Figuring out the query part of the rule is more interesting. The steps in $Expr_{2.1}^{XQ}$ are translated to the labels of the query term. Double parenthesis rather than single ones are the right choice to construct the Xcerpt translation, because $Expr_{2.1}^{XQ}$ does not constrain the number of children within the nodes on the path. Since query terms with single subterms are sufficient for the translation of $Expr_{2.1}^{XQ}$, the completeness with respect to order is not important in this context. For the translation in Table 2.2 curly braces were chosen, but square brackets would also be correct.

Table 2.2: $Expr_{2.1}^{XC}$: The translation of $Expr_{2.1}^{XQ}$

```
GOAL xc:result { all var X }
FROM {
  in resource ['bib.xml'],
  tag1 {{ tag2 {{ var X as tag3 {{ }} }} }}
}
END
```

2.2 The XQuery Sublanguage XQ_1

In this section the grammar productions for XQ_1, a sublanguage comprising queries like the example query in Table 2.1, are given. In terms of the XQuery core, XQ_1 is a small subset of nested FLWOR-expressions that fulfills the following constraints:

- The only variable allowed is the built-in variable `$fs:dot`.
- The innermost construct is a `let`-clause binding the *context variable* `$fs:dot` to a document node by calling the `fn:doc()`-function on a URI. Its `return`-clause is given by a step expression.
- This `let`-clause constitutes the binding sequence of an enclosing `for`-clause, which may itsself be the binding sequence of yet another `for`-clause. Just as with `let`-clauses, the `return`-clauses of the `for`-clauses are given by step expressions.

Table 2.3 gives the grammar productions for XQ_1 in EBNF. `<QNAME>` denotes an arbitrary qualified name as defined in [2].

Table 2.3: Grammar productions for XQ_1

```
<EXPR> ::= for $fs:dot in (<EXPR> | <LET>) return <STEP>
<LET>  ::= let $fs:dot := fn:doc('<URI>') return <STEP>
<STEP> ::= child::<QNAME>
```

In the rest of this section, the semantics of XQ_1 is thoroughly studied based upon the XQuery formal semantics. The following constructs are examined: `let`-clauses, `for`-clauses with empty or non-empty binding sequences, the `fn:doc` function, and step expressions consisting of a child axis and node tests.

2.2.1 The Formal Semantics of `let`-Clauses in XQ_1

The semantics of XQuery expressions is given with respect to a *static environment* `statEnv` and a *dynamic environment* `dynEnv`. The *static environment* is primarily used to perform static type checking in case of schema validation of the XML input data. Not considering validated XML, most rules that only use or alter the *static environment* can be safely ignored for the purposes of this thesis. In some cases though, rules concerning the static environment can be give further insight. Since types of expressions restrict their values, conclusions affecting the dynamic environment – and thus the results of expressions in XQ_1 – can be drawn also from rules concerning only the static environment.

As in other functional programming languages, XQuery `let`-clauses are processed by extending the dynamic environment by a variable binding and evaluating the returned expression in this updated environment. This procedure is formalized by Rule 2.2. Variable references such as VarRef must be expanded before being registered in the dynamic environment DynEnv, because they may contain namespace-prefixes.

$$\text{(Rule 2.2)}\qquad \frac{\begin{array}{c}\text{dynEnv} \vdash \text{Expr}_1 \Rightarrow \text{Value}_1 \\ \text{statEnv} \vdash \text{VarRef of var } \textbf{expands to} \text{ Variable} \\ \text{dynEnv + varValue(Variable} \Rightarrow \text{Value}_1) \vdash \text{Expr}_2 \Rightarrow \text{Value}_2\end{array}}{\text{dynEnv} \vdash \text{let VarRef := Expr}_1 \text{ return Expr}_2 \Rightarrow \text{Value}_2}$$

The example expression $Expr_{2.2}^{XQ}$ from the last section uses a `let`-clause to bind the special variable `$fs:dot` to the document node of `'bib.xml'`. In this extended environment, the step expression `child::tag1` is evaluated.

2.2.2 The Formal Semantics of `for`-Clauses in XQ_1

The `for`-clause being the central construct of XQ_1 its dynamic semantics are treated next. The following rule states that if the binding sequence of a `for`-clause is empty, then the `for`-clause itself evaluates to the empty sequence.

(Rule 2.3)
$$\frac{\text{dynEnv} \vdash \text{Expr}_1 \Rightarrow ()}{\text{dynEnv} \vdash \text{for VarRef TypeDeclaration? in Expr}_1 \text{ return Expr}_2 \Rightarrow ()}$$

Generally, Expr_1 is not empty. The semantics of this case is specified by Rule 2.4, which is read as follows:

- "Let (Item_1, ..., Item_n) be the sequence which Expr_1 evaluates to." (In XQ_1, Expr_1 is a `for` or `let`-clause and evaluates to a sequence of nodes. In the innermost `for`-clause of $Expr_{2.2}^{XQ}$ the binding sequence Expr_1 is the list consisting only of the root node of `bib.xml`.)
- "Let Variable be the expanded QName of VarRef". (The expanded QName is needed to add a correct variable binding to the dynamic environment in the next step.)
- "Let Value_i ($1 \leq i \leq n$) be the value of Expr_2 in the current environment complemented by the binding of the variable Variable to the value Item_i."
- "Then the given for-clause evaluates to the sequence of values (Value_1 ,..., Value_n).

(Rule 2.4)
$$\frac{\begin{array}{c}\text{dynEnv} \vdash \text{Expr}_1 \Rightarrow \text{Item}_1, \ldots, \text{Item}_n \\ \text{statEnv} \vdash \text{VarRef of var expands to Variable} \\ \text{dynEnv + varValue(Variable} \Rightarrow \text{Item}_1) \vdash \text{Expr}_2 \Rightarrow \text{Value}_1 \\ \ldots \\ \text{dynEnv + varValue(Variable} \Rightarrow \text{Item}_n) \vdash \text{Expr}_2 \Rightarrow \text{Value}_n\end{array}}{\text{dynEnv} \vdash \text{for VarRef in Expr}_1 \text{ return Expr}_2 \Rightarrow \text{Value}_1, \ldots, \text{Value}_n}$$

2.2.3 The Formal Semantics of Step Expressions in XQ_1

While the semantics of `for`-clauses is certainly helpful in demonstrating the equivalence of certain expressions in XQ_1 and XC_1, it is still necesarry to examine how the *binding sequences* ($\texttt{Expr}_1$ in Rule 2.4) of the `for`-clauses, which can either be a `fn:doc` function or a step expression (e.g. `child::tag1`), are evaluated. Otherwise Rule 2.4 is not particular useful.

The dynamic semantics of the `fn:doc` function is not specified in [11], but can be described as a "string-to-document-node mapping" [14, section 15.5.4]. As a way out, it is assumed that the document nodes returned by the XQuery `fn:doc` function and by the Xcerpt `in resource` construct are equal (except for differences in the data model), provided they are applied to the same URL.

The other type of expression a binding sequence in XQ_1 can be made up of, is a *step expression*. In order to clarify the vocabulary used in formal semantics of step expressions, some of the terminology of XPath step expressions (see Table 2.2.3) is recapitulated: A *step expression* is either an *axis step* or a *primary expression*. In XQ_1, only axis steps are allowed at the place of step expressions, and therefore the productions for `PrimaryExpr` are not given in Table 2.2.3. An *axis step* is either a *forward step* or a *reverse step*. Again, reverse steps are not included in XQ_1 and no grammar productions are provided in table 2.2.3. A *forward step* is specified by an axis such as the *child axis* or the *attribute axis* and a *node test* which shall not be confused with *predicates*, which are denoted by square brackets in XPath and excluded from XQ_1. *Node tests* are either *kind tests* selecting only nodes of a special kind (not part of XQ_1) or *name tests* selecting nodes of a given name. In XQ_1, only forward steps on the child axis with name tests without wildcards are used.

The formal semantics of XQuery defines the evaluation of step expressions by Rule 2.5, which builds upon a set of other rules that must also be considered to fully understand the evaluation. In the rest of this section, the term 'judgement' is often used. A judgement is simply a statement which expresses whether a property holds or not. The two most important judgements in the XQuery formal semantics are the judgements `Expr => Value` and `Expr : Type`. The former states that the expression `Expr` evaluates to the value `Value`,

Table 2.4: The core syntax of step expressions as defined in [11, section 4.2.1]

```
 1: StepExpr     ::= AxisStep | PrimaryExpr .
 2: AxisStep     ::= ForwardStep | ReverseStep .
 3: ForwardStep  ::= ForwardAxis NodeTest .
 4: ForwardAxis  ::= ('child' '::') | ('descendant' '::') | ('attribute' '::') |
 5:                  ('self' '::') | ('descendant-or-self' '::') |
 6:                  ('following-sibling' '::') | ('following' '::') |
 7:                  ('namespace' '::') .
 8: NodeTest     ::= KindTest | NameTest .
 9: NameTest     ::= QName | Wildcard .
10: Wildcard     ::= * | (NCName ':' '*' | '*' ':'NCName) .
```

and the latter judgement holds if `Expr` is of type `Type`. Judgements are used in the premises of rules to constrain the rule's applicability.

(Rule 2.5)

$$\frac{\begin{array}{c}\text{dynEnv.varValue(\$fs:dot)} = \text{Value}_1 \\ \text{Value}_1 \text{ matches node} \\ \text{dynEnv} \vdash \text{axis Axis of Value}_1 \Rightarrow \text{Value}_2 \\ \text{Axis principal PrincipalNodeKind} \\ \text{dynEnv} \vdash \text{test NodeTest with PrincipalNodeKind of Value}_2 \Rightarrow \text{Value}_3\end{array}}{\text{dynEnv} \vdash \text{Axis NodeTest} \Rightarrow \text{fs:distinct-doc-order(Value}_3\text{)}}$$

In Rule 2.5 the dynamic environment is used to look up the context variable `$fs:dot`. The second premise makes use of the `matches` judgement in order to assure that the rule is only applicable if the value of the context variable is a node value. While there is no definition of the `matches` judgement for nodes in the XQuery formal semantics [11], there are definitions for the `matches` judgement for each kind of node, i.e. element nodes, attribute nodes, document nodes, text nodes, comment nodes and processing instruction nodes (see Table 1.3 for the productions of node values). As an example consider the `matches` judgement for text nodes in Rule 2.6. Although not specifically stated in [11], it is assumed in this thesis that the `matches` judgement used in Rule 2.5 holds true for general nodes, if it is true for one of the node types.

(Rule 2.6)

$$\frac{}{\text{statEnv} \vdash \text{text } \{ \text{ String } \} \text{ matches text}}$$

The third premise of Rule 2.5 asserts that the application of the axis `Axis` to the value of the context node $\texttt{Value}_1$ yields the value $\texttt{Value}_2$. In the case of XQ_1 the specified axis must be the child axis. Application of the child axis to an element (see Rule 2.7) simply returns an element value – that is all the children, but none of the attributes of that element:

(Rule 2.7)

$$\frac{}{\begin{array}{c}\text{dynEnv} \vdash \textbf{axis child::} \textbf{ of element ElementName } \{\text{AttributeValue,ElementValue}\} \\ \Rightarrow \text{ElementValue}\end{array}}$$

Although there are several other rules specifying the semantics of the application of the child axis to a node other than an element node, the most interesting case remains the one of element nodes. Application of the child axis to an attribute node, text node, processing instruction node or comment node yields the empty sequence `()`. In [11] this is specified by rules similar to the following.

(Rule 2.8)

$$\frac{}{\text{statEnv} \vdash \textbf{axis child::} \textbf{ of text: empty}}$$

Note that Rule 2.8 is a statement about the *static* semantics of XQuery that is always applicable (no premises need to be fulfilled). To be exact, it does not directly say anything about the value that is computed by the application of the child axis, but only about its type. Even though type checking – and with it a great part of the information available in the static environment – is ignored when translating from XQuery to Xcerpt, this rule is still significant, since the only value that is of type `empty` is the empty sequence [11, section 2.4.3]. Up to now the semantics of applying the `child` axis to all kinds of nodes but the document node have been treated. The dynamic semantics of this last remaining part being left unspecified in [11], Rule 2.9 specifies its static semantics.

(Rule 2.9)
$$\frac{}{\text{statEnv} \vdash \textbf{axis}\ \text{child::}\ \textbf{of}\ \text{document}\ \{\ \text{Type}\ \}\text{: Type \& processing-instructions* \& comment*}}$$

It is obvious that the value of the documents only child element (possibly interleaved by processing instructions and comment nodes) is returned when applying the child axis to the document node, however, being a static typing rule, Equation 2.9 does not assert that. This dynamic semantics is assumed when calculating the value of expressions in XQ_1.

The fourth premise in Rule 2.5 is simple. The `principal` judgement for the child axis is true whenever `PrincipalNodeKind` is `element`. Hence this premise determines the principal node kind of the axis - which in XQ_1 must always be the *child axis*. The result is bound to PrincipalNodeKind to be used in the fifth and last premise.

The last judgement that needs to be examined to be certain about the semantics of XQ_1 is the *test judgement*, which is used in the fifth and last premise of Rule 2.5. As mentioned earlier, the only type of node tests allowed in XQ_1 are name tests without wild cards. Considering only steps on the child axis, the principle node kind is `element`. Thus the judgements appearing in Rule 2.5 is of the following type:

$$\text{dynEnv} \vdash \text{test Prefix:LocalPart with element of NodeValue}$$

This judgement is true, when the premises in Rule 2.10 are fulfilled.

- The node that is to be matched must be an element node. No attribute nodes, comment nodes, etc shall be matched.
- `Prefix` must be bound to the namespace URI of the expanded QName of `NodeValue`. This simply means that only those elements are matched that belong to the same namespace as specified by `Prefix`.
- `LocalPart` must match the local name of the expanded QName of `NodeValue`. In other words, only elements with tag name `LocalPart` fulfill this condition.

(Rule 2.10)
$$\frac{\begin{array}{c}\text{dm:node-kind(NodeValue) = PrincipalNodeKind}\\ \text{fn:node-name(NodeValue) = expanded-QName}\\ \text{fn:namespace-uri-from-QName(expanded-QName) = statEnv.namespace(Prefix)}\\ \text{fn:local-name-from-QName(expanded-QName) = LocalPart}\end{array}}{\text{dynEnv} \vdash \text{test Prefix:LocalPart with PrincipalNodeKind of NodeValue} \Rightarrow \text{NodeValue}}$$

Summing up the evaluation of step expressions in XQ_1, the following steps are carried out: The context variable `$fs:dot` is looked up in the dynamic environment, and it is checked whether it is bound to a node. After that, the child axis is applied to the node, and it is checked whether the namespace prefixes and the local names of the labels in the step expression and the ones of the child nodes correspond. Only those nodes for which these tests succeed are returned.

With rules 2.2 to 2.10 the entire formal semantics of XQ_1 is given. Looking back, one may say that `for`- and `let`-clauses generate bindings for the context variable, whereas the step expressions are evaluated depending on exactly these bindings.

2.3 The Xcerpt Sublanguage XC_1

This section introduces the formal semantics of XC_1. First the grammar productions for expressions in XC_1 are given, then its formal semantics as specified in [17, Chapter 8] is studied.

Table 2.5: Grammar productions for XC_1

```
<XC1EXPR>   ::= 'GOAL xc:result [ all var X ] FROM' <DOCQUERY> 'END'
<DOCQUERY>  ::= 'in { resource[ "' uri '"],' <QUERYTERM> '}'
<QUERYTERM> ::= <NSLABEL> '{{' <QUERYTERM> '}}'
                | 'var X as' <NSLABEL> '{{ }}'
<NSLABEL>   ::= (label ':')? label
```

As can be seen in Table 2.5, XC_1 contains simple construct-query rules. All of these rules wrap their results in a special element named `xc:result`. The query parts of the rules are of arbitrary depth, but include only one single variable `X`, which is always constrained by a childless, unordered and incomplete query term pattern.

XC_1 is supposed to include translations for all expressions in XQ_1 and examining its formal semantics is a fundamental prerequisite to see the correctness of the translation rules in Section 2.4. It is assumed that the reader has a basic understanding of the operational semantics of Xcerpt as given in [17, Chapter 8].

The rest of this section is structured by the constructs that need to be examined: Xcerpt programs as a set of construct-query-rules, resource specifications, partial unordered query term specifications, Xcerpt construct terms, and the Xcerpt `as` construct.

An Xcerpt program $\mathcal{P}$ generally consists of a set of Xcerpt rules $\mathcal{R}_1, \ldots, \mathcal{R}_k$. For each rule in an Xcerpt program the query part is inserted as a constraint into a new constraint store, the store is solved by the application of simplification rules, and transformed into a substitution set. Application of this substitution set to the construct part of the Xcerpt rule being processed yields the result. In the case of XC_1 we only consider single rules, so we do not have to bother with rule chaining.

Initially the constraint store corresponding to a query in XC_1 has the following form:

$$\langle in\{RSpec, Q\}\rangle_{\emptyset}$$

The resource specification $RSpec$ is a URL identifying an XML document or some other source of Xcerpt data terms such as another Xcerpt program. Q is the query part of the XC_1 expression. Application of the query unfolding rule for resource specifications [17, Section 8.3.2] changes the constraint store to: $\langle Q\rangle_R$ with R being the set of data terms obtained by parsing the resource $RSpec$. This single query term Q with the associated resource $R = \{t_1, \ldots, t_n\}$ is transformed in a disjunct of simulation constraints:

$$\frac{\langle t^q\rangle_{\{t_1,\ldots,t_n\}}}{Q \preceq_u t_1 \vee \ldots \vee Q \preceq_u t_n} \tag{2.1}$$

At this point, the constraint store is a disjunction of constraints, composed of single query terms consisting of two constructs: labels with double curly braces (possibly together with namespace prefixes) denoting incomplete unordered query term specifications, and variable bindings (the `as` construct). For both of these constructs so-called *decomposition rules* are provided by the formal semantics of Xcerpt [17, Section 8.2.2].

2.3.1 The Formal Semantics of Incomplete Query Term Specifications in XC_1

Xcerpt query terms whose outermost construct is a label are decomposed by *root elimination*. Xcerpt permitting a wide range of incompleteness specifications, its decomposition rules are numerous. In selecting the appropriate decomposition rule the following scenarios need to be considered: *label mismatch*, *brace incompatibility*, *left term without subterms*. *Label mismatches* correspond to XQuery name tests on nodes. If the label of the query term does not equal the label of the data term, the constraint resolves to `false`, otherwise Rule 2.2 eliminates the root of the current query term. *Brace incompatibility* is not a problem with unordered query term specifications as they are used in XC_1. Even ordered query terms would not result in brace incompatibility, given that XQuery (and hence also XQ_1 and XC_1) is restricted to ordered data terms. The third scenario, *left term without subterms* arises exactly once for each expression in XC_1 and evaluates always to true with XC_1's double curly braces. To be more specific, only the innermost label in the query term does not have any child terms, all other labels have exactly one child term in XC_1. In the case that the label of the query and data term match, these considerations restrict the set of applicable decomposition rules for all but the innermost label to the following single rule:

$$\frac{l\{\{t_1^1, \ldots, t_n^1\}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi\in\Pi_{pr}} \bigwedge_{1\le i\le n} t_i^1 \preceq_u \pi(t_i^2)} \tag{2.2}$$

Equation 2.2, which is part of the formal semantics of Xcerpt as defined in [17, Section 8.2] has the following meaning: A constraint that unifies an unordered, partial query term $l\{\{t_1^1, \ldots, t_n^1\}\}$ with an ordered data term $l[t_1^2, \ldots, t_m^2]$ can be decomposed into the conjunction of constraints $\bigwedge_{1\le i\le n} t_i^1 \preceq_u \pi(t_i^1)$, if the following condition is satisfied:

- π is a partial, *index injective* and *position respecting* function that maps subterms of the query term on the left hand side to subterms of the data term on the right hand side. This function is furthermore demanded to be total on the set of non-optional and non-negated subterms t_i^1. With XC_1 excluding optional and negated subterms, this postulation can be substituted by considering only total functions on all query subterms. *Index injectivity* requires that no two different terms t_i^1 and t_j^1 are mapped to the same term t_k^2. The condition *position respecting* is of no relevance, because it lays an additional restriction only on those subterms that feature an Xcerpt position specification, which are not present in XC_1.

As we are interested in *all* data terms that match with the specified query term, the simulation constraint is not replaced by *one single* simulation constraint obtained by choosing a *single* function π that satisfies the above condition, but it is replaced by the disjunction of *all possible* simulation constraints that can be produced considering all potential functions π that satisfy the above condition.

Carrying over the semantics of this rule to XC_1, the first thing to note is that the number of subterms is syntactically limited to one. With the further insight from above that each π needs to be total, we can infer that the number of functions π to be considered equals the number of subterms of the data term, which the query term is to be matched with. These thoughts are summed up by the following alternative decomposition rule for XC_1:

$$\frac{l\{\{t_1^1\}\} \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{1 \leq k \leq m} t_1^1 \preceq_u t_k^2} \tag{2.3}$$

Note the resemblance of Equation 2.3 with the formal semantics of XQuery `for`-clauses (Rule 2.4). Whereas the XQuery evaluation rule for `for`-clauses produces a sequence of values (which in general is a sequence of sequences) from a single binding sequence, the root elimination rule for XC_1 2.3 transforms a single constraint into a disjunction of multiple constraints. In the case that the XQ_1 `for`-clause is combined with a step expression on the child axis as its binding sequence, the parallelism goes even further. The step expression evaluates to all those child elements (in the XML data) that match the specified tag name. Similarly, Rule 2.3 can only be applied if the labels match; otherwise the constraint evaluates to false. In XQ_1, the inner expression of the `for`-clause is evaluated in the context of each element in the binding sequence. Accordingly, Rule 2.3 tries to unify the only child element of an XC_1 query term with each subterm of the data term. These analogies give rise to the assumption that root eliminating decomposition rules in XC_1 take on the same role as `for`-clauses with step expressions as binding sequences in XQ_1. However, due to the different evaluation paradigms, these constructs cannot be compared isolated from other constructs such as the Xcerpt `as` construct and the XQuery context variable.

2.3.2 The Formal Semantics of the Xcerpt `as` Construct in XC_1

In most cases, subterms in XC_1 are of the form `label {{<SUBTERM>}}`. This is the recursive case in the grammar productions (see Table 2.5) for terms in XC_1. In order to return some kind of result, however, one needs to make use of the `as` construct, which is the finalizing case in building XC_1 terms. In this section, a simplified version of Xcerpt's `as` elimination rule that is sufficient for XC_1 is derived to better understand the formal semantics of XC_1.

As a result of the decomposition Rule 2.3 or also the query unfolding Rule 2.1, the constraint store may contain a constraint of the form $Var\ X \rightarrow l\ \{\{\ \}\} \preceq_u d$ for some data term d at some point during the evaluation of an XC_1 expression. This constraint does not appear for all combinations of XC_1 queries and associated data resources, because the evaluation of the query may fail due to label mismatches before a variable binding is reached. Such constraints are decomposed by the `as` elimination rule from the formal semantics [17, Section 8.2.2]:

$$\frac{X \rightarrow t^1 \preceq_u t^2}{t^1 \preceq_u t^2 \wedge t^1 \preceq_u X \wedge X \preceq_u t^2} \tag{2.4}$$

Only simple constraints for variables being included in the XC_1 grammar productions, the term t^1 is syntactically limited to $l\ \{\{\ \}\}$. Hence, the first resulting constraint in 2.4 becomes $l\ \{\{\ \}\} \preceq_u t^2$ which - in the absence of node mismatches - evaluates to $True$ by application of the decomposition rule for "left term without subterms" defined in [17, Section 8.2.2]. The transitivity Rule [17, section 8.1.4] can be used to

further simplify the result to $t^1 \preceq_u t^2 \wedge X \preceq t^2$. In fact, adding the lower bound t^1 for the variable X in Xcerpt is necessary to check whether there exist any incompatible upper bounds for X. The number of constraints for the variable X being limited to one single constraint in XC_1, an incompatible upper bound cannot exist, and the `as` elimination rule may be simplified to read:

$$\frac{X \to l\ \{\{\ \}\} \preceq_u t^2}{l\ \{\{\ \}\} \preceq_u t^2 \wedge X \preceq_u t^2} \tag{2.5}$$

In the following sections both of the derived rules from this section together with the formal semantics of QX_1 are taken advantage of to show that the translation rules for mapping expressions between the two sublanguages are correct.

2.4 Translating Between XC_1 and XQ_1

In this section the discussion of XC_1 and XQ_1 is concluded by giving translation rules between both sublanguages and give proof of the equivalence of two generic expressions that represent the entire sublanguages.

The translation rules are denoted by $[\![\ldots]\!]_{toxc1}$ for the translation from XQ_1 to XC_1 and $[\![\ldots]\!]_{toxq1}$ for the other direction. In tables 2.6 and 2.7 names enclosed in angle brackets are non-terminal symbols, *label* and uri are variables that represent qualified names with namespace prefixes and uniform resource locators, respectively. $toxc1$ takes a list of labels as its parameter, and $toxq1$ an additional uri.

Unfortunately, it is necessary to entirely parse and collect the label names of the expressions to be translated before any output can be written. This disutility stems from the definitions of the sublanguages that demand that the innermost labels in XC_1 correspond to the outermost labels in XQ_1. The grammar productions for XQ_1 could be changed to allow translation rules without parameters as follows: `for`-clauses contain step expressions in their binding sequences rather than in their `return`-clauses, and the nested `for`-clauses are moved in the reverse way: from the binding sequence to the `return`-clause. In this way, the outermost labels in XQ_1 would correspond to the outermost labels in XC_1, and could be translated as soon as they are discovered. On the other hand this approach would make the proof of equivalence below less readable and has therefore not been adapted.

Three translation rules, one for each of the constructs `for`, `let` and `fn:doc(...)`, represent the function $toxc1$. `for`-clauses are translated by simply collecting the label in their `return`-clause and recursively calling the tranlsation function on the expression in their binding sequence. Labels are added to the beginning of the parameter list, which means that the labels used in the innermost `for`-clauses appear at the beginning of the list when all `for`-clauses are processed. `let`-clauses are handled in the same way, and $toxc1$ is recursively called on the `fn:doc()`-function. At this point, all labels are collected, and the translation is constructed. It is apparent that the length of the translation grows linearily with the number of labels in the original expression. Since exactly one label occurs within a `let`- or `for`-clause, the number of labels is also a measure for the length of an XQ_1 expression. Therefore the length of the XC_1 translation is linear in the original XQ_1 expression.

Table 2.6: Tranlsation rules XQ_1 to XC_1

$[\![$ `for $fs:dot in <EXPR> return child::`$label$ $]\!]^{<l_1,\ldots,l_k>}_{toxc1}$ `==` $[\![$`<EXPR>`$]\!]^{<label,l_1,\ldots,l_k>}_{toxc1}$

$[\![$ `let $fs:dot := fn:doc('` uri `') return child::`$label$ $]\!]^{<l_1,\ldots,l_k>}_{toxc1}$
`==` $[\![$ `fn:doc('` uri `') `$]\!]^{<label,l_1,\ldots,l_k>}_{toxc1}$

$[\![$ `fn:doc('` uri `') `$]\!]^{<l_1,\ldots,l_k>}_{toxc1}$

```
== GOAL xc:result [ all var X ]
   FROM in { resource [ uri ],
        l_1 {{ ... l_{k-1} {{ var X as l_k {{ }} }} ... }} }
   END
```

The translation rules from XC_1 to XQ_1 (Table 2.7) are similar to those of $toxc1$ in that they also consume

the outermost constructs first. This allows a straightforward implementation based on the abstract syntax of XC_1. On the other hand, it requires that an additional parameter be reserved for the uri reference.

In the same manner as with the reverse direction, the list of labels is empty at the beginning. The first rule in Table 2.7 takes care of the entire construct-query-rule, and calls the translation function on the query term, thereby handing over the uri reference. The second rule collects the labels appearing in the query term one by one, appending them to the end of the list. The third rule handles the only variable together with its pattern restriction and collects the last label. The last rule generates the XQuery translation, using the first label in the list in the innermost construct, because it is the outermost label of the query term.

With the same reasoning as for $toxc1$, it can be shown that the results of $toxq1$ scale linearily with the verbosity of the XC_1 input expression.

Table 2.7: Translation rules XC_1 to XQ_1

$[\![$ GOAL xc:result [all var X]
FROM in { resource [' *uri* '], <QUERYTERM> }
END $]\!]_{toxq1}$ == $[\![$ <QUERYTERM> $]\!]^{uri,<>}_{toxq1}$

$[\![$ *label* {{ <QUERYTERM> }} $]\!]^{uri,<l_1,...,l_k>}_{toxq1}$ == $[\![$ <QUERYTERM> $]\!]^{uri,<l_1,...,l_k,label>}_{toxq1}$

$[\![$ var X as *label* {{ }} $]\!]^{uri,<l_1,...,l_k>}_{toxq1}$ == $[\![$ $]\!]^{uri,<l_1,...,l_k,label>}_{toxq1}$

$[\![$ $]\!]^{uri,<l_1,...,l_k>}_{toxq1}$

```
  == for $fs:dot in
       ...
         for $fs:dot in
           let $fs:dot := fn:doc(' uri ') return child::l1
         return child::l2
       ...
     return child::lk
```

Equivalence of expressions in XC_1 and XQ_1 In the remainder of this section it is shown that the two generic expressions $Expr_{2.4_{XQ}}$ and $Expr_{2.4_{XC}}$ below – which cover the entire sublanguages – are equivalent.

$Expr_{2.4_{XQ}} :=$

```
element xc:result { {
  for $fs:dot in
    ...
      for $fs:dot in
        let $fs:dot := fn:doc(' uri ') return child::tag1
      return child::tag2
    ...
  return child::tagn
} }
```

$Expr_{2.4_{XC}} :=$

```
GOAL
    xc:result [ all var X ]
FROM
    in { resource [' uri '],
       tag1 {{ ... {{ var X as tagn {{ }} }} ... }} }
END
```

In order to show the equivalence of $Expr_{2.4_{XQ}}$ and $Expr_{2.4_{XC}}$, the following assumptions are made:

1. The `fn:doc` function does not raise an error and returns a document node. In [14] `fn:doc` may also return an empty sequence, but this case is not considered.

2. Let d_c be the data term returned by the `resource` construct and let d_q be the document node returned by the `fn:doc` function. By definition (see Rule 2.9), d_q posesses exactly one element child node. It may contain other child nodes, such as processing instructions, comments, etc, but no other element child nodes. It is assumed that this only element child node corresponds to d_c.

3. If an arbitrary label l contains a namespace prefix, and the static environment of XQuery maps this namespace prefix to the namespace ns, then also in Xcerpt, the prefix is bound to ns.

4. If l does not contain a namespace prefix, but the default namespace in XQuery is bound to ns, then also the default namespace in Xcerpt is bound to ns.

5. The sequence returned by $Expr_{2.4_{XQ}}$ does not contain any two nodes that are deep-equal, because duplicate nodes are automatically eliminated in Xcerpt. An alternative to guarantee both expressions to be equal would be to wrap $Expr_{2.4_{XQ}}$ in a call to the `distinct-elements` function (see Table 5.2).

The equivalence of both expressions is shown by a series of transformations, starting with the query part of $Expr_{2.4_{XC}}$ and finishing with a variation of $Expr_{2.4_{XQ}}$, which returns constraints instead of the context variable. Apart from allowing expressions in XQ_1 to return constraints, the proof is eased by the introduction of the context variable in the constraint store. The semantics of both of these extensions is straightforward:

- The value of an XQuery expression returning a series of constraints is defined by the application of the solution of the constraint store to the construct term of $Expr_{2.4_{XC}}$. If the expression returns more than one constraint, the constraint store is given by the disjunction of these constraints.

- A constraint containing a context variable resolves to the constraint obtained by substituting the context variable by the node to which it is bound in the XQuery part of the expression.

After the final transformation, it is easy to see that for each node n which would be returned by $Expr_{2.4_{XQ}}$, the constraint store of $Expr_{2.4_{XC}}$ contains the constraint $X \preceq_u n$. The transformations $=^1$ to $=^6$ are justified as follows:

- In the first step, substituting the root node of the Xcerpt data term d_c by the context variable `$fs:dot`, which is bound to the only element child node of the XQuery document node, the second assumption above is used. Note that the step expression in the `return`-clause of the inner `let`-clause must be `child::element()` rather than `child::*`, because the latter expression would also return processing instructions and other non-element child nodes. Though the formal semantics of kind tests, such as `child::element()` has not yet been examined, the meaning of this specific step expression is evident: It returns all those children of the context node that are element nodes. Using `child::tag0` instead would bring forward the selection of nodes by tag names, which in Xcerpt is represented by the constraint `tag0 {{ ... }}` $\preceq$ `$fs:dot` that remains to be resolved in the following step.

- The second transformation is justified by the root elimination rule of XC_1 (Equation 2.3). There are two cases to be considered: Either the data term d_c is of the form `tag0 {{ ... }}`, in which case the result depends on the descendants of d_c, or, in the other case, a node mismatch occurs. In the second case, the node test `tag0` guarantees that the evaluation fails (the context variable would be bound to the empty sequence `()` in the outer `let`-clause, and therefore no valid constraints are in the constraint store). In the first case, however, the remaining constraints in the `return`-clause control the further evaluation of $Expr_{2.4_{XC}}$. The number of disjunctions k is determined by the number of child elements of the context variable. Again, processing instructions and other non-element nodes are of no interest, so the kind test `element()` is used.

- The disjunction of constraints produced by the root elimination rule is transformed to a `for`-clause in the third step, shifting the context variable one level lower in the data term. While there is one constraint for each child element of the context node before the transformation, now there is one constraint for the shifted context node, which adds up to the same result considering that the context variable is bound once to every child element in the `for`-clause. The innermost `let`-clause returning only a single node, it would not matter whether it is surrounded by a `for`- or `let`-clause. The grammar productions of XQ_1 demand a `for`-clause and therefore this change is also applied in this step.

- Steps two and three are repeated until the variable constraint is reached, producing the nested `for`-clauses that are characteristic for XQ_1. The rationale is the same as in the steps above.
- As discussed in Section 2.3.2 and summarized in Equation 2.4 each `as` constraints is resolved to two constraints: The first one assuring that the data term satisfies the structure of the query term, the second establishing the variable binding.
- In the last transformation, the duty of sorting out non-matching child elements is transferred from the constraints `tagn {{ }}` $\preceq$ `$fs:dot` to the name test `tagn`, leaving only variable constraints in the `return`-clauses. Remember that a sequence of constraints returned by an XQ_1 fragment is understood as a disjunction of these constraints.

```
{ tag0 {{ ... {{ var X as tagn {{ }} }} ... }} ⪯u dc }
```

$=^1$

```
let $fs:dot :=
    let $fs:dot := fn:doc(' uri ') return child::element()
return { tag0 {{ ... {{ var X as tagn {{ }} }} ... }} ⪯u $fs:dot }
```

$=^2$

```
let $fs:dot :=
    let $fs:dot := fn:doc(' uri ') return child::tag0
return
    { ⋁i=1...k {
        tag1 {{ ... var X as tagn {{ }} ... }} ⪯u (child::element())[i]
    } }
```

$=^3$

```
for $fs:dot in
    for $fs:dot in
        let $fs:dot := fn:doc(' uri ') return child::tag0
    return child::element()
return { tag1 {{ ... var X as tagn {{ }} ... }}  ⪯u $fs:dot }
```

$= \ldots =^4$

```
for $fs:dot in
    ...
        for $fs:dot in
            let $fs:dot := fn:doc(' uri ') return child::tag0
        return child::tag1
    ... return child::element()
return
    { var X as tagn {{ }} ⪯u $fs:dot }
```

$=^5$

```
for $fs:dot in
    ...
        for $fs:dot in
            let $fs:dot := fn:doc(' uri ') return child::tag0
        return child::tag1
    ... return child::element()
return
    { tagn {{ }} ⪯u $fs:dot ∧ X ⪯u $fs:dot }
```

$=^6$

```
for $fs:dot in
    ...
        for $fs:dot in
            let $fs:dot := fn:doc(' uri ') return child::tag0
        return child::tag1
    ... return child::tagn
return
    { X ≼u $fs:dot }
```

The resulting substitution set Σ is the set consisting of mappings from X to the nodes specified by the context node. Applying this substitution set to the construct term `<xc:result>all var X</xc:result>` returns a new `xc:result` node filled with all possible substitutions for X. Note that neither XQuery nor Xcerpt specify the order of the child elements in the outer `xc:result` node. Therefore $Expr_{2.4_{XC}}$ and $Expr_{2.4_{XQ}}$ are only equal abstaining from the order of child elements. Thus, curly instead of square brackets could have been used in the XC_1 construct term.

With the above proof of equivalence, the treatment of the first pair of sublanguages is finished. The major discoveries of this first chapter are recapitulated below.

- Simple XPath expressions which are normalized to nested `for`-clauses with *name tests* as binding sequences, can be translated to Xcerpt query terms with single child elements.
- Special attention has to be laid on the treatment of duplicates and element order.
- Xcerpt root elimination rules adopt the roles of two XQuery constructs: *name tests* and `for`-clauses.
- The complexity of the translation rules between both sublanguages is linear in the number of constructs employed.
- Translating between XC_1 and XQ_1 is a structure conserving process: Translating an XC_1 expression to XQ_1 and back to XC_1 yields the original expression. The same property holds for an expression in XQ_1 which is translated to XC_1 and back.

In the following chapter, XC_1 and XQ_1 are expanded to include multiple child nodes in query terms, an arbitrary number of variables, and variable constraints. Special care is taken to retain the beneficial complexity and stability properties discovered in the first pair of sublanguages, despite the partial alleviation of the extensive syntactical constraints.

Chapter 3

Translating Simple Xcerpt Query Terms

This Chapter introduces the sublanguages XC_2 and XQ_2 of Xcerpt and XQuery, respectively. In contrast to the last chapter, the starting point in this Chapter is not a fragment of XQuery, but a subset of Xcerpt query terms. Another important change with respect to the last Section is given by a different notion of equality among XC_2 and XQ_2 expressions (see Section 3.2).

This Chapter is organized as follows: In Section 3.1 the grammar productions for XC_2 are introduced, and the enhancements over XC_1 are discussed. Subsequently, example expressions within XC_2 are translated to XQuery in Section 3.2. Grammar productions for an equally expressive sublanguage of XQuery are presented in Section 3.3. The formal semantics of the constructs of this new sublanguage XQ_2 are studied in Section 3.4. Automatic translation from XC_2 to XQ_2 is discussed in Section 3.5, and the reverse direction is treated in Section 3.6.

3.1 The Sublanguage XC_2

XC_2 differs from XC_1 in the following ways:

- For each query term an arbitrary number of child subterms, instead of only one as in XC_1 is allowed.
- It is also possible to specify an arbitrary number of variables in XC_2. Moreover, the same variable may occur any number of times anywhere in the expression.
- The pattern restrictions for variables are no more limited to simple labels, but may be themselves any type of query term in XC_2.
- As a consequence, also nested pattern restrictions are allowed meaning that restricted variables might appear within the pattern restrictions of other variables.

The grammar productions for XC_2 are given in Table 3.1. `<LABEL>` denotes any qualified name as defined in [2], and `<XCVAR>` may be any Xcerpt variable name. Although this grammar is very short, the corresponding language is much more expressive than XC_1.

Table 3.1: Grammar productions for XC_2

```
<QTERM>  ::= <LABEL> {{ <QTERMS>? }} | var <XCVAR> as <QTERM> .
<QTERMS> ::= <QTERM> ( , <QTERM> )* .
```

3.2 Example XC_2 Expressions and Translations

In order to introduce the challenges that have to be faced when translating XC_2, and to get an idea of the required constructs that need to be included in XQ_2, several translation possibilities for Xcerpt example expressions are discussed in this section. Among these challenges are all the extensions listed above. Since the

separation of query and construct terms is a characteristic only of Xcerpt, there is no equivalent to an Xcerpt query term in XQuery. Therefore, it needs to be defined when an Xcerpt query term is considered equal to an XQuery expression.

When evaluated, Xcerpt query terms merely produce substitution sets, whereas XQuery expressions always produce some data. To make XQ_2 expressions comparable to Xcerpt query terms, only expressions returning an XML representation of substitution sets are be included (The exact grammar productions are given in Section 3.3). A query term qt in XC_2 is considered to be equal to a query qu in XQ_2, if the XML-representation of the substitution set produced by qu is equivalent to the substitution set produced by qt. To be more precise, the substitution sets returned by XQuery expressions are allowed to be multi-sets, thus possibly including the same substitution more than once. The transformation of multi-sets of substitutions to real substitution sets can be easily achieved by applying the `distinct-elements` function introduced in listing 5.2.

An example for a substitution set in XML-representation is given in Table 3.2. The substitutions contain bindings for the variables `X` and `Y`, computed element constructors are used to represent the metadata, and direct element constructors for the bindings.

Table 3.2: An XML representation of a substitution set

```
element xc:substitution_set {
   element xc:substitution {
      element xc:X { <a><b/></a> }, element xc:Y { <c/> } },
   element xc:substitution {
      element xc:X { <a><c/></a> }, element xc:Y { <b/> } },
   element xc:substitution {
      element xc:X { <a><c/></a> }, element xc:Y { <d/> } }
}
```

Another important aspect of defining the equivalence of Xcerpt and XQuery expressions is the data the expressions are evaluated on. In Xcerpt, query terms are either associated with input resources, or they participate in rule chaining. This chapter does not make any assumption about the origin of the data. Instead it is assumed that the same data - wherever it comes from - is present as child elements in the XQuery variable `$data`. In other words, the resource of the data of query terms is left unspecified, but it is demanded that this data is equivalent to the XML-value in `$data`. Although sequences are the primary data structures in XQuery, it is assumed that `$data` contains only a single rooted tree. In the case of a resource being associated with a query term, `$data` may be thought of as the document node of this resource.

3.2.1 Multiple Child Subterms

The most important enhancement of XC_2 with respect to XC_1 is the possibility to specify more than one child subterm. Of course, this makes the semantics of XC_2 more complex. As a consequence, the root elimination Rule 2.2 can not be simplified to 2.3 as it was possible in XC_1.

When translating expression $Expr_{3.2.1_{XC}}$, it needs to be ensured that the nodes that are bound to the variable `X` have siblings with tag names `tag2` and `tag3`. In XQuery, this is achieved by simple `if-then-else` clauses. `if-then-else` clauses were not included in XQ_1 and therefore their semantics will be briefly discussed in Section 3.4. The conditional expression in the `if` clauses are `fn:boolean` functions that take a step expression as the only argument. If the argument evaluates to the empty sequence, the `fn:boolean` function returns false, otherwise it returns true. Its detailed semantics will also be discussed in Section 3.4.

$Expr_{3.2.1_{XC}} :=$ `tag0 {{ var X as tag1 {{ }}, tag2 {{ }}, tag3 {{ }} }}`

$Expr_{3.2.1_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in v0/child::tag1 return
      if fn:boolean($v0/child::tag2) then
        if fn:boolean($v0/child::tag3) then
          element xc:substitution { element xc:X {$v1} } else ()
      else () }
```

The `for`-clauses in $Expr_{3.2.1_{XQ}}$ differ from those in the last Chapter in that their binding sequences consist of variable references together with axis steps, and the more complex subexpressions are found in the `return`-clauses. This is almost exactly the opposite way as before. In fact, both represenations may be transformed into each other (see Section 5.4.3 for details). Another important point to note is that with all child subterms of `tag0` having different labels, ensuring Xcerpt injectivity is not an issue.

3.2.2 Dealing with Xcerpt Injectivity: Multiple Subterms with Overlapping Labels

Whereas XC_1 allows at most one child subterm in a query term, this restriction is not present in XC_2. Therefore it may happen that an Xcerpt query term has multiple children with the same label. Hence, a way has to be found to guarantee that the nodes we find in XQ_2 are distinct if they are to be matched with siblings of the same parent node. This is not only necessary for the case of identical labels, but also for the case of *overlapping* ones.

Overlapping of labels is defined as follows: Let $t_1 := p_1{:}l_1$ and $t_2 := p_2{:}l_2$ be qualified names. t_1 and t_2 overlap, if and only if their prefixes p_1, p_2 and their local names l_1, l_2 overlap. To simplify the problem, regular expressions for local names and namespace prefixes are excluded. Thus, the values that p_1, p_2, l_1 and l_2 can assume, are only strings or the character '*' denoting any name. Two namespace prefixes or two local names s_1 and s_2 overlap iff: s_1 = '*', s_2 = '*' or $s_1 = s_2$.

Consider the XC_2 example expression in Table 3.3. For each of the three subterms of the root node `tag0`, there must exist a matching subterm in the root node of the data term. How is it possible to ensure that all three of these subterms are distinct?

In the data model of XQuery and XPath, every node has a unique identifier. The function `op:is-same-node($x,$y)` can be used to compare the node identifiers of the nodes `$x` and `$y`. Its formal semantics is discussed in Section 3.4.3 together with the formal semantics of the `fn:not` function which is also used in the example expression below. An alternative way to check the equality of nodes would be to remember the position of each node within its parent node and to compare the position variables. This could be achieved by using `for`-clauses with an additional `at`-construct. This method is further discussed in Chapter 4 to tranlsate ordered Xcerpt query terms, but in this section the `op:is-same-node` function is preferred.

In the tables below three possibilities for translating $Expr_{3.2.2}$ are presented. They differ in their efficiency, phrase complexity and use of XQuery constructs. The first one is straightforward in that it consists only of `for` and `if`-clauses, but is expected to be less efficient than the second and third one, which depend on the more complex `some-satisfies` construct. The third translation takes advantage of the greatest variety of constructs, making its semantics more intricate than the ones of the other two, but at the same time reducing its phrase complexity to a linear level.

Table 3.3: $Expr_{3.2.2}$: An XC_2 expression with overlapping tag names within the same parent

`tag0 {{ var X as tag1 {{ }}, tag1 {{ }}, tag1 {{ }} }}`

While this first translation does find all solutions to the original query, in many cases it returns duplicates. To see this take a look at the following data term:

$d =$ `tag0 {{ tag1 {{ tag2 {{ }} }}, tag1 {{ }}, tag1 {{ }}, tag1 {{ }} }}`

In the first cycle of the outermost for clause, `$x1` will be bound to the first child subterm of d, and there are six alternative ways to assign distinct child subterms to the variables `$x2` and `$x3`, without infringing the injectivity constraints. Thus the expression will return a list of 24 nodes, containing each subterm of d six

Table 3.4: $Expr_{3.2.2}$ translated with `for`, `if` and `let`-clauses

```
element xc:subtitution_set {
   for $v0 in $data/child::tag0 return
     for $v1 in $v0/child::tag1 return
       for $v2 in $v0/child::tag1 return
         for $v3 in $v0/child::tag1 return
           if (fn:not(op:is-same-node($v1, $v2))) then
             if (fn:not(op:is-same-node($v1, $v3))) then
               if (fn:not(op:is-same-node($v2, $v3))) then
                 element xc:subtitution { element xc:X { $v1 } }
               else ()
             else ()
           else () }
```

times. It would be possible to eliminate the duplicates with a `distinct-elements`-function 5.2, but this doesn't prevent the XQuery compiler from doing extra work. Therefore, the expression is better translated by using the `some-satisfies` construct as in Table 3.5, checking only for the existence of an expression that satisfies the given condition:

Table 3.5: $Expr_{3.2.2}$ translated with the additional `some-satisfies` construct

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      if (some $v2 in $v0/child::tag1 satisfies (
        fn:not(op:is-same-node($v1, $v2)) and
        (some $x3 in child::tag1 satisfies (
          fn:not(op:is-same-node($v1, $v3) and
            fn:not(op:is-same-node($v2, $v3)))
          ))))
      then element xc:substitution {
        element xc:X { $v1 } } else ()
```

In this translation, no futile work is carried out, but a considerable number of injectivity preserving node comparisons: Let n be the number of child nodes of a parent node in the XC_2 query term. Then $\frac{n\cdot(n-1)}{2}$ is an upper bound for the number of necessary node comparisons [1]. Assuming that all tag names overlap, it is not even possible to do any better. Generally, however, the set of child nodes C may be transformed in a set of subsets $C_1, ..., C_k$ of C such that for each $c_i, c_j \in C_m$ the labels of c_i and c_j overlap, and such that for each $c_i \in C_m$ and $c_j \in C_n$ with $n \neq m$ the labels do not overlap. The subsets $C_1, ..., C_k$ can be computed easily, if regular expressions are excluded. If one of the labels is $*$, the corresponding child node must be included in every subset. Note that if we disallowed the $*$ in tag names, the subsets would be equal to the equivalence classes of C with respect to the equivalence relation $=_{String}$ testing the equality of strings. By including $*$, however, the equivalence relation does not satisfy the requirement *transitivity*: From $* =_{String} tag1$ and $* =_{String} tag2$ does not follow $tag1 =_{String} tag2$. Having computed the subsets $C_1, ..., C_k$ the total number of comparisons required for the given parent node equals $\Sigma_{i=1...k} \frac{|C_i|(|C_i|-1)}{2}$.

Decreasing the phrase complexity. In the case that the number of subsets $C_1, ..., C_k$ is small, the complexity of the resulting XQuery expression in terms of the number of constructs of the translated Xcerpt query term is unsatisfactory. As shown above, in the worst case, the number of required node comparisons is quadratic in the number of child subterms. Luckily, there exists an easy way out of this unlovely situation that restores linear

[1] $\frac{1}{2}n \cdot (n-1)$ is the number of possiblities to pick two arbitrary elements out of a set of n elements. For any two of the n siblings, injectivity has to be ensured in the worst case

complexity: It makes use of the `op:except` function and ensures from the start that no two variables are bound to the same data subterm. This solution is presented in Table 3.6 as the third translation of expression $Expr_{3.2.2}$

Table 3.6: $Expr_{3.2.2}$ translated with the `op:except` function

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    let $seq1 := $v0/child::tag1* return
      for $v1 in $seq1 return
        let $seq2 := op:except($seq1, $x1) return
          if (some $v2 in $seq2 satisfies (
            let $seq3 := op:except($seq2, $x2) return
              some $x3 in $seq3 satisfies true
          ) then element xc:substitution
            element xc:X { $v1 } else ()
}
```

It remains to be shown that the semantics of the root elimination rule (Equation 2.2) can be imitated as described above. The following reasoning builds upon the corresponding argumentation for XC_1 from Section 2.3.1.

The root elimination rule transforms a simulation constraint between a query term q and a data term d into a disjunction of conjunction of simulation constraints between the children of q and d. The conjunctions are reflected in the XQuery translations by the fact that all expressions are nested. Only if bindings for *all* XQuery variables (that represent the Xcerpt subterms) can be found, a substitution set is returned. Disjunctions are mirrored by `for`-clauses. Evaluating an expression e including a variable v once for every binding of v to the data subterms $d_i, (1 \leq i \leq k)$ is the same as evaluating e together with the disjunction $\bigvee_{1 \leq i \leq k} v = d_i$.

One disjunction is produced by the root elimination rule for every total and injective function mapping the subterms of q to the subterms of d (for the exact definition scroll back to Section 2.3.1). Totality of the functions is ensured in the translations if *every* subterm of a query term is associated with an XQuery variable bound in a nested `for`-clause. Injecitvity of the functions is guaranteed by the injectivity constraints between these variables in the XQuery translation.

3.2.3 Multiple Variables

Another relaxation of the strict syntactic constraints of XC_1 lies in the possibility to specify multiple variables in XC_2 query terms like in $Expr_{3.2.3_{XC}}$. Ignoring construct terms and focusing only on the production of correct substitution sets in this part of the thesis, the treatment of multiple variables is straightforward: The substitutions must contain one sub-element for each Xcerpt variable occurring in the query term. As before, subterms containing variables must be translated using `for`-clauses. Those without variables are better translated with `some-satisfies` clauses.

$Expr_{3.2.3_{XC}} :=$

```
tag0 {{ var X as tag1 {{ }}, tag2 {{ var Y as tag2 {{ }} }} }}
```

$Expr_{3.2.3_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      for $v2 in $v1/child::tag2 return
        for $v3 in $v2/child::tag2 return
          element xc:substitution {
            element xc:X { $v1 },
            element xc:Y { $v2 },
          }
}
```

3.2.4 Dealing with Xcerpt Injectivity: Multiple Variables with Overlapping Labels

Combining the major issues of the last two sections – multiple variables and overlapping tag names – is straightforward: Xcerpt injectivity is guaranteed by the node comparison with the negated `op:is-same-node`-function and for all variables occurring within the query term, an element must be constructed within the substitutions.

For a change, $Expr_{3.2.4_{XQ}}$ presents yet another way to ensure Xcerpt injectivity: It counts the number of distinct nodes among the variable bindings for all siblings. On the one hand, this method allows to check the adherence to the injectivity constraints with one single construct. On the other hand, this check needs to be delayed until the last one of the variables is bound, infringing on the principle of carrying out selections (elimination of tuples of XQuery variables by node-comparisons) before joins (nested `for`-clauses).

$Expr_{3.2.4_{XC}} :=$ `tag0 {{ var X as tag1 {{ }}, var Y as tag1 {{ }} }}`

$Expr_{3.2.4_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      for $v2 in $v0/child::tag1 return
        if (fn:count(distinct-nodes-stable($v1, $v2)) = 2) then
          element xc:substitution {
            element xc:X { $v1 },
            element xc:Y { $v2 }
          }
}
```

3.2.5 Deep Pattern Restrictions

Another lifted restriction in XC_2 with respect to XC_1 concerns pattern restrictions for variables. To be precise, pattern restrictions that are *deep* in the sense that they do not only specify the label of the variable to be bound, but also its structure including its descendants, are now included in XC_2. When querying XML data, selecting nodes with a certain structure is a very important aspect, and the expressivity of XC_2 allows to formulate such queries, which is demonstrated by $Expr_{3.2.5_{XC}}$.

$Expr_{3.2.5_{XC}} :=$ `tag0 {{ var X as tag1 {{ tag2 {{ tag3 {{ }} }} }} }}`

$Expr_{3.2.5_{XC}}$ can be translated in a similar fashion to $Expr_{3.2.2}$, since in a way, the required siblings for variable `X` in $Expr_{3.2.2}$ can be seen as a pattern restriction for `X`.

$Expr_{3.2.5_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      if some $v2 in $v1/child::tag2 satisfies ($v2/child::tag3)
      then element xc:substitution { element xc:X { $v1 } }
}
```

3.2.6 Multiple Constraints for one Variable

An interesting situation that may arise in XC_2 are multiple constraints for the same variable. In this section it is argued that it is necessary to introduce a new function, the `fn:deep-equal` function, in order to be able to translate expressions of this kind. To substantiate this insight, the transformations within the Xcerpt constraint store are observed during the evaluation of the example expression $Expr_{3.2.6_{XC}}$.

$Expr_{3.2.6_{XC}} :=$ `tag0 {{ var X as tag1 {{ tag2 }}, var X as tag1 {{ }} }}`

In the representations of the constraint store below, the sign `:<` denotes a simulation constraint. Initially, the constraint store contains a single simulation constraint $Expr_{3.2.6_{XC}} \preceq d$ for some data term d.The first transformation is achieved by applying the root elimination rule (Equation 2.2). To take away some of the complexity of the transformations, only a single disjunct of the disjunction resulting from Equation 2.2 is considered (it is assumed that the disjunction is not empty). Let π be the function that is used to map query subterms to data subterms in this disjunct, and let `d1` and `d2` be the distinct data terms that the first and the second subterm of $Expr_{3.2.6_{XC}}$ are mapped to by π. Note that `d1` and `d2` are distinct, because π is injective. Then the constraint store contains the following conjunction after this first transformation.

```
CS = { var X as tag1 {{ tag2 {{ }} }} :< d1, var X as tag1 {{ }} :< d2 }
```

These constraints are further decomposed by the `as` elimination rule (Equation 2.4), which also adds the necessary lower and upper bounds for both occurrences of variable `X`:

```
CS = {
   tag1 {{ tag2 {{ }} }} :< d1, tag1 {{ tag2 {{ }} }} :< X, X :< d1,
   tag1 {{ }} :< d2, tag1 {{ }} :< X, X :< d2
}
```

This is where the consistency rule ([17, Section 8.1.4]) comes into play. It guarantees that multiple upper bounds for the same variable are consistent. Two upper bounds t_1 and t_2 for the same variable are replaced by the bisumlation constraint $t_1 \preceq_u t_2 \wedge t_1 \preceq_u t_2$. Besides, one of the upper bounds remains in the constraint store. In the constraint store above, variable `X` has the two upper bounds `d1` and `d2`. In the transformation the upper bound `d1` is kept, and the bisumulation constraint is added. The remaining constraints remain untouched.

```
CS = {
   tag1 {{ tag2 {{ }} }} :< d1, tag1 {{ tag2 {{ }} }} :< X, X :< d1,
   tag1 {{ }} :< d2, tag1 {{ }} :< X,
   d1 :< d2, d2 :< d1
}
```

Luckily, both data terms `d1` and `d2` are accessible in the XQuery translations by the variables to which they are bound in `for`- or `some`-clauses. Therefore they can be easily compared by a call to the `fn:deep-equal`-function. The exact conditions under which Xcerpt bisimulation and the `fn:deep-equal`-function yield the same results are elaborated in Section 3.4.2.

In the next step, the transitivity rule is used to replace variable occurrences within upper bounds of a simulation constraint by their own upper bounds. The upper bound for `X` being `d1`, `d1` is substituted for all occurrences of `X` on the right hand side of simulation constraints.

```
CS = {
   tag1 {{ tag2 {{ }} }} :< d1, tag1 {{ tag2 {{ }} }} :< d1, X :< d1,
```

```
   tag1 {{ }} :< d2, tag1 {{ }} :< d1,
   d1 :< d2, d2 :< d1
}
```

This results in some redundancy in the constraint store. The first two constraints being exactly equivalent, one of them can be omitted. Furthermore the constraints `tag1 {{ }} :< d2` and `tag1 {{ }} :< d1` are equivalent, because their upper bounds bisimulate. Hence, one of them can be omitted.

```
CS = {
   tag1 {{ tag2 {{ }} }} :< d1, X :< d1,
   tag1 {{ }} :< d1,
   d1 :< d2, d2 :< d1
}
```

As mentioned before, the bisimulation constraints are taken care of by the `fn:deep-equal`-function in the XQuery translation. The remaining constraints are translated in just the same way as before.

$Expr_{3.2.6_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      if (some $v2 in $v1/child::tag2 satisfies true
          and some $v3 in $v0/child::tag1 satisfies
            (fn:not(op:is-same-node($v3, $v1))
            and fn:deep-equal($v1, $v3))
      then element xc:substitution {
        element xc:X { $v1 } }
```

Producing a `some-satisfies` clause for the second subterm, $Expr_{3.2.6_{XQ}}$ breaks with the principle of translating all subterms containing variables with `for`-clauses. The reason for this is that a binding for variable `X` is already generated by the translation of the first subterm. The fact that `X` occurs more than once in the expression is a constraint on the structure of the data terms, but does not induce additional substitutions. Translating the second subterm by means of a `for` clause would result in duplicates in the substitution set.

In the case that the query shall distinguish between nodes in the input documents that are deep-equal, adding value-based duplicates to the constraint store may be exactly the right thing to do. As a matter of fact, in the second part of this thesis, it is argued that a constraint store containing duplicates is a prerequisite for translating certain XQuery expressions to Xcerpt. But since the focus of this section lies on translating query terms from Xcerpt to XQuery, the substitution sets generated shall equate those that are produced according to the formal semantics of Xcerpt.
$Expr_{3.2.6_{XQ}}$ pinpoints the difference between *value based node equality* and *identity based node equality*. While the nodes bound to the variables `$a` and `$b` need to have distinct node identifiers, their values must equal.

3.2.7 Nested Pattern Restrictions for Different Variables

An obvious enhancement contained in XC_2 over simple pattern restrictions are nested pattern restrictions as showcased in $Expr_{3.2.7_{XC}}$. If Variable `Y` were left out, the expression would be an example for *deep pattern restrictions* as treated in Section 3.2.5. Variable `Y` included, however, all bindings for variable `Y` must be collected. Simply checking for the existence of a node that fulfills the constraint for Y would not suffice. Hence, $Expr_{3.2.7_{XC}}$ cannot be translated by using the `some-satisfies` construct, but is translated by means of ordinary `for`-clauses.

$Expr_{3.2.7_{XC}} :=$ `tag0 {{ var X as tag1 {{ var Y as tag2 {{ }} }} }}`

$Expr_{3.2.7_{XQ}} :=$

```
element xc:subtitution_set {
  for $v0 in $data/child::tag0
    for $v1 in $v0/child::tag1 return
      for $v2 in $v1/child::tag2 return
        element xc:substitution {
          element xc:X { $v1 }, element xc:Y { $v2 } } }
```

3.2.8 Nested Constraints for the Same Variable

A sophistication of nested pattern restrictions is presented in $Expr_{3.2.8_{XC}}$: A constraint for variable X appears within a constraint for X itself. It is easy to see that this query only matches with cyclic graph data terms built with Xcerpt references (which are considered as true parent child relationships). XML references via `id` and `idref` attributes are in general not regarded as true parent child relationships, and therefore $Expr_{3.2.8_{XC}}$ yields no results. An equivalent expression in XQuery would be the empty sequence `()`, however, restituting this result postulates that the compiler has found out that the original query contained nested constraints for the same variable. Another feasible approach would be to translate expression 3.2.8 in the same way as $Expr_{3.2.7_{XC}}$ and $Expr_{3.2.6_{XC}}$, demanding that both nodes bound to the Xcerpt variable X (in XQuery two variables would be introduced for X) be deep-equal.

$Expr_{3.2.8_{XC}} :=$ `tag0 {{ var X as tag1 {{ var X as tag2 {{ }} }} }}`

3.2.9 Optimization: Execute Selections Before Joins

Looking closely at the translations of the example expressions, one identifies a considerable number of `for`- and `if`-clauses. In general the `for`-clauses are used to bind variables and the `if`-clauses sort out those variable bindings that are not desirable. As in logical SQL query optimization it is possible to safe spare computations by executing the "selections" (`if`-clauses) before the "joins" (`for`-clauses). Naturally, this is only possible as long as all variables appearing in the `if`-clauses remain bound.

3.2.10 A Complex Example

To see that arbitrary complex XC_2 expressions can be translated in just the same way as the example expressions above, and to sum up the main ideas in one translation, a large query term is translated to XQuery in this section:

$Expr_{3.2.10^{XC}} :=$

```
tag0 {{
    var X as tag1 {{
        var Y as tag2 {{ tag3 {{ tag4 {{ }} }} }},
        var Z as tag0 {{ tag2 {{ tag1 {{ }} }} }}
    }},
    var Z as tag0 {{ var X as tag1 {{ tag2 }} }}
}}
```

There are four subterms within $Expr_{3.2.10^{XC}}$ that need to be bound to XQuery variables using `for`-clauses. These include all subterms that bind variables and those that have children that bind a variable. They are marked red in $Expr_{3.2.10^{XC}}$. Since there are multiple occurrences of variables X and Z, one binding occurence for each of these variables has to be picked and translated by a `for`-clause. The other occurrences of X and Y are considered consuming occurrences and are therefore translated by `some`-clauses. Subterms to be translated by `some`-clauses are marked blue in $Expr_{3.2.10^{XC}}$.

In the XQuery translation the variable `$v0` is associated with the root of the query term, the variables `$v1` and `$v2` with its children. `$v3` represents the subterm binding variable Y and is therefore also translated by a `some`-clause. The `if`-clause beginning in line 6 forms the deep pattern constraint for Y. As the second XQuery variable associated with a subterm containing X (`$v2` was the first such variable), `$v6` is translated with an existential quantification (line 10). The same holds for `$v9` as the second representative of Y (line 17). As

soon as both representatives for an Xcerpt variable are bound, the deep-equality constraint is emitted. Finally the substitution set is returned in line 21 if all checks are successfull. Note that the three `if`-clauses could also be combined to a single `if`-clause by `and`-connecting the conditions to form a single larger condition. This is treated in more detail when translating XC_3.

```
1  element xc:substitution_set {
2    for $v0 in $data/child::tag0 return
3      for $v1 in $v0/child::tag1 return          (: var X is bound here :)
4        for $v2 in $v0/child::tag0 return        (: var Z is bound here :)
5          for $v3 in $v1/child::tag2 return      (: var Y is bound here :)
6            if (                        (: deep pattern restriction for Y :)
7              some $v4 in $v3/child::tag3 satisfies
8                (some $v5 in $v4/child::tag4 satisfies true))
9            then
10             if (some $v6 in $v1/child::tag0 satisfies
11                              (: all representatives of Z must be equal :)
12               (fn:deep-equal($v6, $v2) and
13                                  (: deep pattern restriction for $v6 :)
14                 (some $v7 in $v6/child::tag2 satisfies ($v7/child::tag1)
15                 )))
16             then
17               if (some $v9 in $v2/child::tag1 satisfies
18                        (: all representatives for Y must be deep-equal:)
19                  (fn:deep-equal($v9, $v1) and ($v9/child::tag2)))
20               then                (: deep pattern restriction for $v9 :)
21                  element xc:substitution {
22                    element xc:X {$v1}, element xc:Y {$v3},
23                    element xc:Z {$v2} }
24               else ()
25             else ()
26           else ()
27 }
```

The findings of this section are used in the following Section to develop an equally expressive XQuery sublanguage named XQ_2.

3.3 XQ_2 Grammar Productions

An expression in XQ_2 always starts with an element constructor for the substitution set to be returned. From then on, nested `some` and `for` clauses, injectivity and deep equality constraints may be constructed finalizing in an `xc:substitution` element containing all variable bindings calculated so far. XQuery variable names are denoted by `<XQVAR>` and must be preceded by a dollar sign.

Table 3.7: Grammar productions for XQ_2

```
<XQ_2>     ::= element xc:substitution_set { <EXPR> } .
<EXPR>     ::= <FOR> | <IFCLAUSE> | <SUBST> .
<IFCLAUSE> ::= if <COND> then <EXPR> else () .
<COND>     ::= fn:deep-equal(<XQVAR>, <XQVAR>) | <SOME> | <AND> .
               fn:not(op:is-same-node(<XQVAR>, <XQVAR>)) .
<SOME>     ::= some <XQVAR> in <STEP> satisfies <SOMEEXPR> .
<FOR>      ::= for <XQVAR> in <STEP> return <EXPR> .
<SUBST>    ::= element xc:substitution{ (element xc:<XCVAR> {<XQVAR>})* } .
<STEP>     ::= <XQVAR>/child::<QNAME> .
```

In order to retain the exact expressiveness of XC_2 and to exclude redundant or unreasonable expressions, several limitations are placed on the above grammar productions:

- Naturally, references to XQuery variables may only appear within the scope of these variables. This restricts the usage of deep-equality and injectivity constraints.
- The outermost bound variable `$v0` must be child of the special variable `$data`. All other variables must be descendants of `$v0`.
- For any two variables that are bound in `for` or `some` clauses with the same parent variable and overlapping labels, there must be an `if` clause that assures the injectivity constraint for the two siblings. Otherwise it is impossible to find an equivalent XC_2 expression. The translation of such expressions in XC_3 is discussed in Section 4.2.2.
- The inversion of the above requirement – that for every injectivity constraint the compared variables must be siblings – is not a prerequisite for XQ_2 expressions to be translatable to XC_2, however, expressions not satisfying this condition are redundant, in that child nodes of different parents are always distinct nodes (this would not be true for a graph data model). Therefore such expressions are not considered part of XQ_2.
- For all variables `$a` appearing in a deep-equality constraint, there must exist a variable `$b` (possibly with `$a` = `$b`) that appears in the `result` element and is transitively linked to `$a` by deep-equality constraints. The reverse requirement is not necessary. If an Xcerpt variable only appears once in the translated query term, its XQuery translation does not include any deep-equality constraints for this variable, but must nevertheless appear within the substitution set.

3.4 Building Blocks of XQ_2

As can be seen when reconsidering the translations in Section 3.2, the following constructs are necessary for translating simple, breadth-incomplete Xcerpt query terms:

- `for` clauses
- step expressions on the child axis with name tests
- let clauses
- the `fn:doc` function
- `if-then-else` clauses
- the `op:is-same-node()` function
- the `deep-equal` function
- indirect element constructors
- the `some-satisfies` clauses

The first four of are also part of XQ_1 and their semantics are discussed in detail in Section 2.2, so in this section special emphasize will be laid upon `if-then-else` clauses in combination with the two node comparison functions, indirect element constructors, and `some-satisfies` clauses for existential quantification in XQuery. The most basic of the four being probably `if-then-else` clauses, they will be treated first.

3.4.1 `if`-Clauses

The formal semantics of `if Expr1 then Expr2 else Expr3` clauses is the same as in other functional programming languages. Some interesting XQuery specific aspects are: Static type checking makes sure that the type of the condition `Expr1` is boolean. `if-then-else` is a non-strict function, because evaluating one of its arguments may throw an error, whereas the whole expression may still be evaluated without errors. The conditional of `if` clauses in XQ_2 is either the `deep-equal()` function or the `is-same-node()` function, which will be discussed next.

3.4.2 The `fn:deep-equal` Function

The examples above make use of the `deep-equal` function to test whether two nodes have the same value. In Xcerpt, the value-based equality is monitored by the constraint store. In this section the conditions under which both of these methods of establishing value-based node equality semantically comply with each other, are derived.

The `deep-equal` function is defined in [14] as a function that operates on two sequences of nodes. Whenever the function is used in the translations of XC_2 expressions, only a pair of nodes is compared. Nevertheless the function must also be discussed when called upon arguments that are sequences, because `deep-equal` calls itself recursively on the lists of child nodes of the original arguments. But the discussion starts out by examining the semantics of the `deep-equal` function in the case that both of its arguments are single nodes.

While `deep-equal` is defined to operate on all kinds of nodes – including document nodes, comment nodes, and processing instructions – the focus in this thesis is laid upon elements, attributes and text nodes, as they are the only ones included in Xcerpt data terms. Let a and b be the two arguments to the function. If the node types of a and b differ, then the function returns false. In the following only comparisons of nodes of the same type are considered.

3.4.2.1 Comparing Element Nodes

Since all variables in XQ_2 expressions are bound to element nodes, the non-recursive calls to the `fn:deep-equal`-function take element nodes as their actual parameters. There are three conditions to be fulfilled for two element nodes to be equal.

Expanded QName equality: Both nodes need to have the same name. To be more precise, the expanded QNames are compared. In XQuery expanded QNames are computed from local names by looking up the namespace prefix in the *statEnv.namespace environment*. In Xcerpt, these node comparisons are performed by the root elimination rules. For a translation between XC_2 and XQ_2 it is easiest to assume that the namespace prefixes are already expanded, such that the namespace prefixes themselves do not need to be translated.

Matching attributes: It is apparent that for two nodes to be deep-equal, their attributes must coincide. In the description of the `fn:deep-equal` function [14] this is formulated as follows. a and b must have the same number n of attributes, and for each attribute in a there must exist an attribute in b, such that those attributes are deep-equal.

An obvious implication of this definition is that there must exist a total function mapping the attributes $\{a_1, \ldots a_n\}$ of a to those of b. But there's more than that: One could assume that two attributes a_i and a_j of a might be mapped to the same attribute b_k of b with $1 \leq i, j, k \leq n$, $i \neq j$, however, since a_i and a_j must be deep-equal to b_k, they must also carry the same name, which is forbidden in XML-documents. Thus, it can be concluded that not not only a total function from $\{a_1, \ldots a_n\}$ to $\{b_1, \ldots b_n\}$ exists, but a total, injective function. With the number of attributes in a and b being equal, this function is also bijective.

While it would be syntactically possible to formulate queries in XC_2 selecting nodes based on the existence of attributes (using the old syntax for attributes), these queries are not considered for the moment. Nevertheless it is necessary to discuss the resolution of Xcerpt constraints which involve attribute nodes, because such constraints may be the byproduct of ordinary node simulations. Note that in this case data terms are compared rather than a query term and a data term as in Equation 2.2.

Attributes in Xcerpt are denoted like ordinary elements and are surrounded by an additional `attributes` element with curly braces indicating incompleteness with respect to order. As a further natural restriction, attributes may not have any child elements. The rest of XML documents being represented as ordered Xcerpt data terms, the root elimination rules for attribute nodes has not yet been examined. To show that the `deep-equal` function and establishing node equality through simulation in Xcerpt yield the same result, it is necessary to catch up on this issue now.

The corresponding root elimination rule is the one with single curly braces on both sides, which means incompleteness with respect to order, but completeness in breadth, and is taken from [17, Section 8.2].

$$\frac{l\{t_1^1, \ldots, t_n^1\} \preceq_u l\{t_1^2, \ldots, t_m^2\}}{\bigvee_{\pi \in \Pi_{bij} \cap \Pi_{pp}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^2)} \tag{3.1}$$

It only differs from Rule 2.2 in that instead of the set of position respecting functions, the set of bijective position preserving functions is considered. The restriction position preserving is always fulfilled if two data terms are simulated. Thus the only remaining constraint is *bijectivity*, which is also the only demand laid down by the definition of the *matching attributes* requirement of the `fn:deep-equal` functions, as shown above.

Matching child nodes Finally, the children of a and b must match. This intuitive understanding is formalized in [14, 15.3.1 fn:deep-equal] as follows: the sequence a`/(*|text())` must be deep-equal to the sequence b`/(*|text())`. Note that all child nodes returned by this expression must either fulfill the *name test* `*` or the *kind test* `text()`. Both tests do not overlap, because `*` expands to `child::*`, and returns only element child nodes of the context node [1, Section 3.2.3]. To cut a long story short, a`/(*|text())` returns all children of a that are either element nodes or text nodes in document order. Interestingly, comments and processing instructions have no influence on the value based equality of nodes as given by the `deep-equals` function.

For two sequences `$parameter1` and `$parameter2` to be deep-equal, they must be of the same length, and "every item in the sequence `$parameter1`" must be "deep-equal to the item at the same position in the sequence `$parameter2`" [14][Section 15.3.1 fn:deep-equal].

In order to show that the ways the `deep-equal` function and Xcerpt ensure the equality of child nodes comply, another variant of the root elimination rule must be introduced: the one with ordered complete terms on both sides as presented in Equation 3.2. *Ordered*, because elements rather than attributes are compared, and *complete*, because data terms rather than query terms are the subject of discussion.

$$\frac{l[t_1^1, \ldots, t_n^1] \preceq_u l[t_1^2, \ldots, t_m^2]}{\bigvee_{\pi \in \Pi_{mon} \cap \Pi_{bij}} \bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u \pi(t_i^2)} \tag{3.2}$$

The set of index monotonic functions Π_{mon} is defined in [17, Definition 4.6] and has the expected meaning. It is easy to see that there exists only one monotonic, index bijective function which is given by $\pi(t_i^1) = t_i^2$ for all left subterms t_i^1 and all right subterms t_i^2. Thus, the result of Rule 3.2 will be the conjunction of constraints $\bigwedge_{1 \leq i \leq n} t_i^1 \preceq_u t_i^2$. According to the definition in XQuery, each subterm of the left hand side is simulated with the subterm at the same position of the right hand side.

3.4.2.2 Comparing Attribute and Text Nodes

As discussed in the last section, for two element nodes to be deep-equal, their attributes must also be deep-equal. For two attribute nodes to be deep-equal, their qualified names must match, and their "typed values" must equal. Only under the assumption that all attributes are untyped, and that the equality of atomic values is the same as the equality of Strings in Xcerpt, it can be guaranteed that the comparison of two attribute nodes in Xcerpt and XQuery yields the same result.

Equality of text nodes is established even easier: Two text nodes "are deep-equal if and only if their string-values are equal" [14][Section 15.3.1 fn:deep-equal]. In Xcerpt, text nodes are "represented as compound terms with the string or regular expression as label, no subterms, and a total term specification" [17, Section 4.4]. As a result, the root elimination rule for "left term without subterms", single curly braces, and right term without subterms covers the role of the `deep-equal` function in case of text nodes as parameters.

3.4.2.3 Conditions for Compliance of Value Based Node Equality in Xcerpt and XQuery

Summing up the comparison between the ways one can check value based equality in Xcerpt and XQuery, the conditions under which the `fn:deep-equal` function with parameters a and b and a constraint store of the form $CS\{a \preceq b \wedge b \preceq a\}$ yield the same result, are recapitulated:

- The parameters a and b of the `deep-equal` function are two single element nodes.
- Namespace prefixes are expanded both in the XC_2 and XQ_2 expressions.
- In XQuery all elements are typed `xdt:untyped` and all attribute nodes are typed `xdt:untyped-Atomic`.
- Atomic value equality and Xcerpt string equality coincide.
- Label comparisons in Xcerpt root elimination rules comply with XQuery string comparisons.

Although these conditions are quite natural, they have to be kept in mind when relying on the translation mechanisms in this thesis.

3.4.3 The `op:is-same-node` Function

As mentioned in the introduction, the XQuery data model includes unique node identifiers for every single node read from an input resource or constructed during the evaluation of an expression. These node identifiers are used by the `op:is-same-node`-function to find out whether two variables have been bound to the same nodes. Unfortunately, XQuery node identifiers get lost when nodes are inlcuded in element constructors. This problem is further discussed in Section 5.2.3, because it prevents finding out if two bindings in the substitution set returned by the translation of an Xcerpt query term stem from the same node of the processed in put resource. In this section, the arguments to the `op:is-same-node` function are variables, that are directly queried from the input resources, and therefore their node identifiers do not change.

3.4.4 The `some-satisfies` Construct

During the translation of deep Xcerpt pattern restrictions, existential quantification was preferred over casting the value of a step expression to a boolean value (See $\text{Expr}_{3.2.2}$ and its translations), because it may prevent the compiler from executing dispensable work. The formal untyped dynamic semantics of `some-satisfies` is defined in [11, 4.11 Quantified Expressions] by rules 3.1 and 3.2. In this section it is shown that the translation of deep pattern restrictions by means of existential quantification is indeed correct.

$$\text{(Rule 3.1)}\quad \frac{\begin{array}{c}\text{dynEnv} \vdash \text{Expr1} \Rightarrow \text{Item}_1, ..., \text{Item}_n \\ \text{dynEnv} \vdash \text{VarRef}_1 \text{ of var } \textbf{expands to } \text{Variable}_1 \\ \text{dynEnv + varValue}(\text{Variable}_1 \Rightarrow \text{Item}_i)) \vdash \text{Expr}_2 \Rightarrow \text{true}, 1 \leq i \leq n\end{array}}{\text{dynEnv} \vdash \text{some VarRef}_1 \text{ in Expr}_1 \text{ satisfies Expr}_2 \Rightarrow \text{true}}$$

Rule 3.1 is read as follows: Let $\texttt{Item}_1, \ldots, \texttt{Item}_n$ be the sequence that $\texttt{Expr}_1$ evaluates to. If there exists one $\texttt{Item}_i$ in this sequence such that $\texttt{Expr}_2$ evaluates to `true` in the initial dynamic environment augmented by the variable binding $\texttt{Variable}_1 \Rightarrow \texttt{Item}_i$, then the existential quantification with variable $\texttt{Variable}_1$, binding sequence $\texttt{Expr}_1$ and condition $\texttt{Expr}_2$ yields true. To allow the use of namespace prefixes in variable names, $\texttt{VarRef}_1$ is used instead of $\texttt{Variable}_1$ in the conclusion of Rule 3.1.

Rule 3.2 specifies the contrary situation. If for all $\texttt{Item}_i$ in $\texttt{Expr}_1$ the condition $\texttt{Expr}_2$ evaluates to `false` in the respective extended environment, also the existential quantification in the conclusion yields false.

$$\text{(Rule 3.2)}\quad \frac{\begin{array}{c}\text{dynEnv} \vdash \text{Expr1} \Rightarrow \text{Item}_1, ..., \text{Item}_n \\ \text{dynEnv} \vdash \text{VarRef}_1 \text{ of var } \textbf{expands to } \text{Variable}_1 \\ \text{dynEnv + varValue}(\text{Variable}_1 \Rightarrow \text{Item}_1)) \vdash \text{Expr}_2 \Rightarrow \text{false} \\ \ldots \\ \text{dynEnv + varValue}(\text{Variable}_1 \Rightarrow \text{Item}_n)) \vdash \text{Expr}_2 \Rightarrow \text{false}\end{array}}{\text{dynEnv} \vdash \text{some VarRef}_1 \text{ in Expr}_1 \text{ satisfies Expr}_2 \Rightarrow \text{false}}$$

To show the equality of deep pattern restrictions in XC_2 and its translations above, consider the following constraint store:

```
CS = { var X as tag0 {{ tag1 {{ }} }} ≼u d }
```

where d is a data term. The constraint store is simplified by the root elimination rule for incomplete unordered query term specifications 2.2. Let $c_1, \ldots, c_k$ be the children of d. Under the assumption that the outermost label of d matches with `tag0`, the application of the `as` elimination rule followed by the root elimination rule transforms the constraint store to:

```
CS = { { tag1 {{ }} ≼u c1, tag0 {{ tag1 {{ }} }} ≼u X, X ≼u d };
     ...  ;
     { tag1 {{ }} ≼u ck, tag0 {{ tag1 {{ }} }} ≼u X, X ≼u d } }
```

This constraint store yields the substitution X = d if at least one $c_i, 1 \leq i \leq k$ has `tag1` as its outermost label. On the contrary, if all c_i are incompatible with the label `tag1`, the constraint store yields false, and no binding for X is returned. This is exactly the semantics as specified by rules 3.1 and 3.2.

3.5 Translating From XC_2 to XQ_2

In this section, an automatic translation algorithm for XC_2 expressions is given in form of translation rules. The algorithm builds upon the following principles and methods, some of which are results from the previous sections.

- It is possible to translate a query term recursively by finding first a translation for the parent, and taking care of its children later.
- A query term and all of its subterms need to be associated with fresh XQuery variables. These variables may either be bound in a `for`-clause or a `some-satisfies` construct. In order to not return duplicates in the substitution sets to be produced, existential quantifications are preferred when there is no new Xcerpt variable at any level in the query term to be translated.
- In order to decide which variables are new in the sense that no such variable has yet been translated, it is necessary to keep track of a list of Xcerpt variables that have already been translated. This list is named `XCVs` in the translation rules below.
- XQuery variables that are associated with subterms of the form `var varname as label {{...}}` need to be remembered and returned as subterms in the substitutions produced. The associations between XQuery variables and Xcerpt variables are stored in a list `As` such as `[(XCVar1, XQVar1), ..., (XCVark, XQVark)]`[2].
- Subterms including new Xcerpt variables (variables not yet contained in `XCVs`) must be translated before siblings that do not include new variables. This is best understood by reconsidering the XQuery expression in Section 3.2.10. The red subterms in this expression include new Xcerpt variables and are therefore translated by `for`-clauses. The ones without new variables are marked blue and translated by `some`-clauses. Emitting a `some`-clause before a `for`-clause would give rise to the following problem. The `for`-clause would have to be included in the `then`-clause of the surrounding `if`-clause. For good reasons, however, the variable bound by the `some`-clause is not available in its surrounding `if`-clause, and injectivity or deep-equality constraints between the variables of the `for`- and `some`-clause could not be emitted in the scope of both variables.
- Injectivity constraints need to be produced for each pair of siblings whose tag names overlap. Naturally, these injectivity constraints may only appear within the `for`- or `some`-clauses binding both variables. Injectivity constraints such as `[($v1, $v2), ($v1, $v3), ($v2, $v3)]` are stored in a list named `Injs`.

Two kinds of translation rules are used: The first type is denoted by $[\![\ldots]\!]_{toXQ}^{\texttt{As, XCVs, XQVs, Injs}}$ and transforms a list of triples into ordinary XQuery expressions, taking into account the variable association `As`, the list of already translated Xcerpt variables `XCVs`, the list of bound XQuery variables `XQVs` and the list of injectivity constraints `Injs`. The triples consist of a query term, an XQuery variable associated with the query term itself, and an XQuery variable associated with the parent node of the query term. In the initial call to translate a query term `qt`, this list consists only of the single triple `(qt, $v0, $data)`. The four parameters are all initialized with the empty list `[]`. The second type of rules is written $[\![\ldots]\!]_{toXQBool}$ and covers the transformation of tuples of variables to boolean XQuery expressions enforcing injectivity constraints. The following functions are used to reduce the verbosity of the translation rules:

- `label(qt)` returns the label of a query term: `label(HTML{{ }}) = HTML`
- `subterms(qt)`: A function returning the list of subterms of a query term:
  `subterms( HTML {{ body {{ }}, head {{ }} }} ) = [body {{ }}, HTML {{ }}]`.
- `fresh_vars(n)` returns `n` XQuery variables that have not been used before:
  `fresh_vars(3) = [$v7, $v8, $v9]`.
- `make_injs(vars)` = `[($v1, $v2) | $v1, $v2 <- vars, $v1 < $v2 ]` (Haskell notation) returns all pairs of variables out of a list of siblings. From the resulting list, injectivity constraints

[2]Haskell notation is chosen to syntactically represented lists. The list items are separated by commas and square brackets are used as list delimiters

are produced. An enhanced version of this function might check, whether an injectivity constraint needs to be produced at all for a pair of variables. Obviously, this may be omitted whensoever the labels of the subterms associated with the variables do not overlap.

- `vars(qt)` computes a list containing all variables of a query term `qt`.
- `ripe_injs(injs, vars)` takes a list of injectivity constraints `injs` and a list of variables `vars`, and deletes all but those injectivity constraints, whose variables are included in the list. The variables handed over to the function is the list of already bound variables `XQVs`. Thus it is safe to emit the injectivity constraints returned by `ripe_injs`.
- `insert_assoc((XCVar, XQVar), Assocs)` inserts a new association between an XQuery variable and an Xcerpt variable into the list of associations `Assocs`. If the Xcerpt variable is already comprised within `Assocs`, the list remains unchanged.
- `getXQVar(XCVar, assocs)` returns the XQuery variable associated with the `XCVar` in the list of associations `assocs`.
- `binding_siblings(qt)` returns the list of variable-binding siblings of the term `qt` that have not yet been translated. The function is used to ensure that terms binding variables are translated before siblings that do not bind any variables.
- `bindsvar(qt)` takes a query term as its argument and returns true if it binds a variable at the top level. In the translation rules it is used to find out whether a deep-equality constraint must be emitted and whether the list `As` of associations must be updated.
  `bindsvar(var X as html{{}}) = true` and `bindsvar(html{{}}) = false`.
- The functions `++` (concatenation of lists), `\\` (difference of lists) `length`, `elem`, `replicate`, `not` and `zip3` assume their respective Haskell meaning.

As defined by Rule 3.1, the application of a translation rule to a query term `qt1` with associated XQuery variable `$v1` results in the subterms being added to the list of triples, and the necessary injectivity constraints `new_injs` being added to `Injs`. A fresh variable is associated with each of the subterms, and the current variable `$v1` is stored as the parent variable of the subterms in the newly created triples. Moreover, `$v1` is added to the list of bound variables, and "ripe" injectivity constraints are removed from `Injs` and emitted as a boolean XQuery expression within the condition of an `if`-clause.

(Rule 3.1)

```
                    vars(subterms(qt1)) \\ XCVs != [],
                          XQVs' = XQVs ++ [$v1],
                            not(bindsvar(qt1)),
                fvars = fresh_vars(length(subterms(qt1))),
                      new_injs = make_injs(fvars),
                   ripe_injs = ripe_injs(Injs, XQVs'),
                  Injs' = Injs ++ new_injs \\ ripe_injs
    triples = zip3 (subterms(qt1), fvars, replicate (length(fvars)) $v1)
-------------------------------------------------------------------------
⟦ [(qt1,$v1,$p1), ... (qtk,$vk,$pk)] ⟧^{As, XCVs, XQVs, Injs}_{toXQ} =
    for $v1 in $p1/child::label(qt1) return
      if (⟦ ripe_injs ⟧_{toXQBool}) then
        ⟦ triples ++ [(qt2,$v2,$p2), ..., (qtk,$vk,$pk)] ⟧^{As, XCVs, XQVs', Injs'}_{toXQ}
    else ()
```

The first premise of Rule 3.1 guarantees that the query term `qt1` contains a new Xcerpt variable. If `qt1` does not contain any new Xcerpt variable, it is translated with an existential quantification instead (Translation Rule 3.2). In the same way as above, new triples, new injectivity constraints, the set of "ripe" and the set of remaining injectivity constraints `Injs'` are calculated, and `$v1` is added to the list of bound XQuery variables. The difference to Rule 3.1 is that instead of a `for`-clause together with a nested `if`-clause, a `some`-clause *within* an `if`-clause is produced. The transformation of the injectivity constraints to a boolean XQuery expression takes place in the `satisfies`-clause, rather than in the condition of the `if`-clause. A premise for this rule (the second one) is that no siblings of `qt1` contain defining variable occurrences. If there are siblings of this kind, the triples in which they are contained must be brought to the front of the list of triples to

```
                    vars(subterms(qt1)) \\ XCVs == [],
                       binding_siblings(qt1) == [],
                            not(bindsvar(qt1)),
                          XQVs' = XQVs ++ [$v1],
                  fvars = fresh_vars(length(subterms(qt1))),
                        new_injs = make_injs(fvars),
                     ripe_injs = ripe_injs(Injs, XQVs'),
                   Injs' = Injs ++ new_injs \\ ripe_injs,
      triples = zip3 (subterms(qt1), fvars, replicate (length(fvars)) $v1)
```

(Rule 3.2)

$$[\![\ [(\texttt{qt1},\$v1,\$p1),\ \ldots\ (\texttt{qtk},\$vk,\$pk)]\ ]\!]_{toXQ}^{\texttt{As, XCVs, XQVs, Injs}} =$$

```
     if (some $v1 in $p1/label(qt1) satisfies ([[ ripe_injs ]]_toXQBool))
     then
       [[ triples ++ [(qt2, $v2, $p2), ..., (qtk, $vk, $pk)] ]]^As, XCVs, XQVs',Injs'_toXQ
     else ()
```

be translated first. A rule that formally specifies this reordering is presented during the translation of XC_3 in Section 4.3.

Note that in rules 3.1 and 3.2, the set of translated Xcerpt variables `XCVs` and the set of associations `As` remain untouched. As a matter of fact these data structures must only be adjusted, if `qt1` binds a variable at the top level, and this case was precluded by the third premises in the rules above. If these premises are inverted to read `bindsvar(qt1)` as in rules 3.3 and 3.4, these parameters come into play. Additionally, the query term `qt1` is marked with the superscript `X` in these rules to denote that `X` is the variable bound by the query term.

The whole sense of keeping track of variable associations is to issue the right value comparisons (two XQuery variables associated with the same Xcerpt variable need to be deep-equal), and to know which variables to return within the substitution set. In fact, only those XQuery variables that are associated with a query term featuring an Xcerpt variable must be returned. If an Xcerpt variable appears multiple times in the entire query term, it is sufficient to remember only one of its XQuery representatives. The function `insert_assoc` takes care of this.

Note that it must be differentiated between the case of a top level variable not being contained in the list `XCVs`, because it is treated for the first time, and the case that it already occurred elsewhere in the query term. In the first case (Rule 3.3), the new association is added to the list of associations `As` and in the subsequent recursive call to the translation function, the updated association list is given as an argument. In the second case, however, the list remains unchanged, and a deep-equality constraint is emitted to ensure value-based equality among the nodes bound to both XQuery variables (Rule 3.4). Similarly, the list of known Xcerpt variables only needs to be updated in the first case.

```
                          bindsvar(qt1),
                       not(elem(X, XCVs)),
          XQVs' = XQVs ++ [$v1], XCVs' = XCVs ++ [X],
               As' = insert_assocs(X, $v1), As),
          fvars = fresh_vars(length(subterms(qt1))),
                 new_injs = make_injs(fvars),
              ripe_injs = ripe_injs(Injs, XQVs'),
            Injs' = Injs ++ new_injs \\ ripe_injs,
triples = zip3 (subterms(qt1), fvars, replicate (length(fvars)) $v1)
```

(Rule 3.3)

$$[\![\ [(\texttt{qt1}^{X},\$v1,\$p1),\ \ldots\ (\texttt{qtk},\$vk,\$pk)]\ ]\!]_{toXQ}^{\texttt{As, XCVs, XQVs, Injs}} =$$

```
     for $v1 in $p1/child::label(qt1) return
       if ([[ ripe_injs ]]_toXQBool) then
         [[ triples ++ [(qt2,$v2,$p2), ..., (qtk,$vk,$pk)] ]]^As', XCVs', XQVs', Injs'_toXQ
       else ()
```

Rules 3.1 to 3.4 produce `for`-and `some`-clauses, and deep equality constraints. The construction of the substitutions is taken care of by another rule for $toXQ$, whereas injectivity constraints are emitted by $toXQBool$.

Processing injectivity constraints is simple compared to the handling of query terms. Each of the tuples of XQuery variables is enclosed in a negated `op:is-same-value` function, and all these constraints are connected by `and` (Rule 3.6). There is not even a premise for this rule.

(Rule 3.4)

```
                                    bindsvar(qt1),
                                    elem(X, XCVs),
                                XQVs' = XQVs ++ [$v1],
                        fvars = fresh_vars(length(subterms(qt1))),
                              new_injs = make_injs(fvars),
                           ripe_injs = ripe_injs(Injs, XQVs'),
                         Injs' = Injs ++ new_injs \\ ripe_injs,
        triples = zip3 (subterms(qt1), fvars, replicate (length(fvars)) $v1)
```

$[\![$ [(qt1^X,$v1,$p1), ... (qtk,$vk,$pk)] $]\!]_{toXQ}^{As,\ XCVs,\ XQVs,\ Injs}$ =

```
    for $v1 in $p1/child::label(qt1) return
      if (([[ ripe_injs ]]_toXQBool) and
         fn:deep-equal(X, getXQVar(X,assocs)))
      then [[ triples ++ [(qt2,$v2,$p2), ..., (qtk,$vk,$pk)] ]]_toXQ^As, XCVs, XQVs', Injs'
      else ()
```

After all of the triples have been processed by one of the rules above, the element constructors for the substitution sets are returned (Rule 3.5). A premise for this rule is that both the query terms to be translated, and the injectivity constraints are empty. The case that injectivity constraints are left, but no more query terms are to be processed can never occur: When the last XQuery variable together with the last subterm is processed by one of the rules 3.1 - 3.4, all XQuery variables that have ever been assigned to a subterm are included in the list `XQVs`. At this point, the function `ripe_injs` returns all of the remaining constraints, and `Injs'` is the empty list `[]`.

(Rule 3.5)

```
                          triples = [], Injs = [],
                  As = [(XCV1, XQV1), ..., (XCVk, XQVk)]
```

$[\![$ triples $]\!]_{toXQ}^{As,\ XCVs,\ XQVs,\ Injs}$ =

```
element xc:substitution_set {
    element xc:XCV1 { XQV1 }, ..., element xc:XCVk { XQVk } }
```

(Rule 3.6)

$[\![$ [($a1, $b1), ..., ($ai, $bi)] $]\!]_{toXQBool}$ =

```
(fn:not(op:is-same-node($a1,$b1)) and ...
     fn:not(op:is-same-node($ai, $bi)))
```

Concluding the automatic translation from XC_2 to XQ_2 two obvious questions about the translation algorithm are discussed: At an arbitrary point in the translation, is it always possible to find a rule that may be applied? Does the application of rules terminate?

To answer the first question, two types of premises in the rules above must be distinguished. Ones that may prevent the rule from being applicable (they are called *hard* premises in this thesis), and ones that are only used to assign values to data structures that are passed on in recursive calls to the translation function (*soft* premises). Soft premises are irrelevant for this first question. The following hard premises occur in the rules for $toXQ$:

- `vars(subterms(qt1)) \\ XCVs == []` and `vars(subterms(qt1)) \\ XCVs != []`,
- `bindsvar(qt1)` and `not(bindsvar(qt1))`,
- `binding_siblings(qt1) == []` and `binding_siblings(qt1) != []`,
- `bindsvar(qt1)` and `not(bindsvar(qt1))`.
- The list of triples is empty (`triples = []`) or not.

These premises can be used to build the decision tree depicted in Table 3.8, which is complete in the sense that for each leaf node a rule is specified. Therefore one can always find a rule that is applicable. For the application of the only rule for $toXQBool$, no premises have to be fulfilled.

Table 3.8 is also helpful in answering the second question. Each of the rules 3.1 to 3.4 consumes one query subterm and adds its children to the list of query terms to be processed. Each subterm is therefore

Table 3.8: Decision tree for the automatic translation algorithm

`1 not(triples = [])`	
`1.1 not(bindsvar(qt1))`	
`1.1.1 vars(subterms(qt1)) \\ XCVs != []`	Rule 3.1
`1.1.2 vars(subterms(qt1)) \\ XCVs == []`	
`1.1.2.1 binding_siblings(qt1) == []`	Rule 3.2
`1.1.2.2 binding_siblings(qt1) == []`	Reorder list of triples
`1.2 bindsvar(qt1)`	
`1.2.1 not(elem(X,XCVs))`	Rule 3.3
`1.2.2 elem(X,XCVs)`	Rule 3.4
`2 triples = []`	Rule 3.5

processed exactly once, and the list of triples is empty when all subterms have been treated, which leads to the non-recursive case `2`. The only questionable leaf node of the decision tree is `1.1.2.2`. It can only occur if `qt1` does not bind any variables and results in the siblings of `qt1` that do bind variables to be processed first. Obviously, this case can occur at most once for each subterm.

3.6 Translating From XQ_2 to XC_2

The translation procedure from XQ_2 to XC_2 is straightforward because XQ_2 was trimmed to include only a tight superset of the expressions produced by translating from XC_2. Note that not all expressions in XQ_2 may be the result of the translation of a query term in XC_2: While the translation algorithm for XC_2 emits constraints always as early as possible to generate reasonably efficient translations, these constraints are not required to appear as early as possible in all valid XQ_2 expressions. As an unpleasant consequence for the translation from XQ_2 to XC_2, the XQuery expression needs to be examined in its entirety before one can decide whether all required injectivity constraints are present for a particular set of siblings. This suggests splitting the translation process into an *analysis phase* and a *construction phase*. The *analysis phase* builds appropriate data structures dedicated to hold all parent-child-relationships between XQuery variables, injectivity and deep-equality constraints. The *construction phase* constructs the resulting query term by consulting these data structures. While this approach would certainly be more efficient, the algorithm described in the following queries the original XQuery expression as the only data structure, omitting the *analysis phase* and abbreviating the exposition of the translation process.

1. Find the root query term q and remember its XQuery variable name and label `l`. There is exactly one root query term which can be identified by searching for a step expression with `$data` as the parent. The variable bound in the corresponding `for` or `some`-clause is the one of the root query term.

2. Associate a fresh Xcerpt variable `X` with q only in the case that the XQuery variable of q appears within the returned substitution set, or that it appears within a deep-equality constraint. Store the association of both variables in a list `Assocs`. Depending on whether a variable has been assoiciated with it, the query term to be produced is either of the form `l{{`$subts$`}}` or `var X as l{{`$subts$`}}`, with the list $subts$ of subterms to be determined in the following step.

3. Find all children $subts = c_1, \ldots, c_j$ of q, their respective XQuery variable names `$v1,...,$vj` and their respective labels `l1,...,lj`. The children are found in the same way, as the root query term above, only that this time step expressions including the variable for q are searched for instead of `$data`. Check that for each pair of variables `$vi`, `$vk` with overlapping labels, there exists an injectivity constraint `fn:not( op:is-same-node($vi, $vk) )`. If one of these constraints is missing, the translation procedure fails.

4. For each child c_i of q determine whether to associate an Xcerpt variable with it. As above, the condition for the association is that the variable appears either within the returned `substitution` element, or within a deep-equality constraint. If this holds true, detect whether the list of associations `Assocs` already contains an XQuery variable v transitively connected to the variable `$vi` of c_i by deep-equality constraints. If this check is positive, associate the Xcerpt variable of v with the query term c_i. Otherwise use a fresh Xcerpt variable. Now one of the two patterns `var X as li{{`$subts_c$`}}` and

`li{{`$subts_c$`}}` with `li` being the label of c_i has been determined for c_i. The list of subterms $subts_c$ of c_i can be calculated by executing steps 3 and 4, substituting q by c_i and $subts$ by $subts_c$.

Obviously, the translation algorithm described above terminates, because as with the reverse direction, each XQuery variable (each subterm for the direction XC_2 to XQ_2) is processed exactly once.

A further interesting question is whether the translations are stable. Naming the translation function of the algorithm above $toXC$, this amounts to the question if for an arbitrary XC_2 query term q and for an arbitrary XQ_2 expression e the equations 3.3 and 3.4 are satisfied.

$$toXC(toXQ(q)) = q \tag{3.3}$$

$$toXQ(toXC(e)) = e \tag{3.4}$$

To answer this question it first needs to be defined when two query terms and two XQ_2 expressions are considered to be equal. Syntactical equivalence should not be taken as a measure for query terms, because XQ_2 query terms which are identical except for subterm ordering are semantically equivalent. As a matter of fact $toXQ$ does not define the order in which subterms are to be constructed from an XQ_2 expression e. Ignoring the order of subterms within XC_2 query terms, Equation 3.3 is in fact fulfilled. A formal proof is not given in this thesis.

Equation 3.4 does not hold true for arbitrary expressions e. To see this consider an expression with an injectivity constraint between two XQuery variables that represent sibling subterms of an XC_2 query term. Such an injectivity constraint is superfluous, because it is always satisfied. Nevertheless the grammar productions allow such constraints, and the translation from XQ_2 to XC_2 simply ignores them. When the resulting query term is translated back to XQuery, only necessary injectivity constraints are produced. Therefore, the Equation 3.4 is not satisfied for this example. But the less strict requirements of equations 3.5 and 3.6 are fulfilled, which is a direct consequence of Equation 3.3.

$$toXQ(toXC(toXQ(q))) = toXQ(q) \tag{3.5}$$

$$toXC(toXQ(toXC(e))) = toXC(e) \tag{3.6}$$

Chapter 4

Translating Complex Xcerpt Query Terms

The sublanguages treated in this Chapter are called XC_3 and XQ_3 and are proper supersets of XC_2 and XQ_2 respectively. With the treatment of these two sublanguages the Xcpert constructs `without` and `optional`, descendants as well as all kind of subterm specifications (ordered, unordered, complete and incomplete), query conjunctions, disjunctions and negations are translated to XQuery and back. As in the last chapter, the translation functions are called $toXQ$ and $toXC$, respectively. The sublanguage XQ_3 of XQuery is chosen such that $toXQ(q) \in XQ_3$ and $toXC(e) \in XC_3$ holds for all queries q in XC_3 and all expressions e in XQ_3. The length of the translations scales linearly with the number of constructs in the original expressions.

This Chapter is divided in four sections. In Section 4.1, XC_3 is introduced and some example expressions are translated to XQuery. An analysis of XQ_3 follows in 4.2. In sections 4.3 and 4.4, automatic translation algorithms between both sublanguages are presented.

4.1 The Sublanguage XC_3

There are several aspects of XC_2 that are unsatisfactory, because they limit admissible queries to a rather small part of Xcerpt query terms. That is why an extension of XC_2 is considered in this section. Enhancements over XC_2 are:

- Ordered and complete query terms are included. Recall that XC_2 only includes curly braces denoting incompleteness in breadth and with respect to order. In this chapter, all kinds of braces may be used.
- Subterm negation is allowed. It is shown how to translate `without` constructs, including nested `without` and ones that contain variable bindings.
- A further extension is the inclusion of the `desc` construct which allows to refer to arbitrary descendants of a query term.
- Besides these enhancements of query terms, also conjunctions (`and (...)`), disjunctions (`or (...)`) and negations (`not (...)`) of queries are translated.

Although this is a considerable increase in expressivity there are still many Xcerpt features, which are not included:

- namespace variables and label variables
- While *optional subterms* are discussed in this section, they are not included in the new sublanguage, because their translation to XQuery would mean that the linear complexity of the translation is lost. Nevertheless the transformation of query terms including optional subterms to optional-free query terms is presented in detail. Applying this transformation, it is possible to translate an extended version of XC_3 including optional subterms to XQ_3.
- *Construct terms* are not discussed until Chapter 5, where the emphasis is laid on translating entire construct-query-rules.

- The translation of Xcerpt regular expressions is not examined in this thesis, but with the possibility to use regular expressions also in XQuery [14, Section 7.6.1], it seems likely that a canonical translation is achievable.
- The translation of position specifications and positional variables is elaborated, but also they are not included in XC_3.
- While their tranlsation does not seem to be complicated, *conditions* to Xcerpt rules are excluded for the sake of brevity.
- The filtering constructs `except` (als referred to as `minus` in [17][Section 9.1.3]), and `plus` are neither included in XC_3 nor even discussed.
- In contrast to the boolean connectives of queries `and`, `or` and `not`, the homonymous connectives for query terms, though discussed in this chapter, are not included in XC_3.

4.1.1 Grammar Productions

The following grammar productions comprise all of the queries expressible in XC_3, but not all queries derivable by this grammar are valid Xcerpt expressions. Although it would be possible to exclude such queries by a finer grained grammar, this grammar is preferred for the sake of brevity.

Table 4.1: Grammar productions for XC_3

```
<QUERY>     := and ( <QUERY>*  ) | or ( <QUERY>* ) | not ( <QUERY> ) |
               <QUERYTERM>
<QUERYTERM> := <QNAME> {{ <QUERYTERM>* }} | <QNAME> { <QUERYTERM>* } |
               <QNAME> [[ <QUERYTERM>* ]] | <QNAME> [ <QUERYTERM>* ] |
               var varname | var varname as <QUERYTERM> |
               without <QUERYTERM> | desc <QUERYTERM>
```

In order to exclude illegal Xcerpt expressions, several natural syntactical restrictions are placed upon the above grammar productions.

- The `without` keyword may not appear twice in a row. A term of the form `without without html {{ var X }}` is illegal. However, it is often desirable to use *nested* `without`, and these are included in XC_3.
- Similarly, there must not be more than one `desc`-construct within the same level of a query term, and only one variable may be bound to the same query term.
- In the case that a query term uses more than one of the constructs `desc`, `without`, `var`, the correct order of these constructs is `without var varname as desc <QUERYTERM>`.

In the following sections, each of the enhancements is discussed by introducing example Xcerpt expressions, and by giving possible translations to XQuery, highlighting advantages and disadvantages of the translation possibilities. Emphasis is laid upon retaining linear complexity of the length of XQuery expressions in terms of the number of constructs employed in the original Xcerpt query. As in the last chapter, the XQuery results will be included in element constructors that represent the substitution sets of the corresponding Xcerpt queries. This methode eases the translation of entire construct query terms in Chapter 5. The XML representation of these substitution sets is the same as in the last chapter.

It is assumed that the data for the Xcerpt query terms is internalized, and that the same data is contained within an XQuery variable named `$data`.

4.1.2 Translating Ordered and Complete Query Term Specifications

The discussion of the enhancements in XC_3 starts out by introducing other types of brackets for query terms than just those with unordered incomplete query term specifications of XC_2. The first objects of interest are single square brackets, and two translation possibilities are compared. Consider expression $Expr_{4.1.2_{XC}}$.

$Expr_{4.1.2_{XC}} :=$ `tag0 [ var X as tag1 [ tag2 {{ }}, tag3 {{ }}, tag1 {{ }} ]]`

$Expr_{4.1.2_{XC}}$ selects all those `tag1` subterms of a root term labeled `tag0` that do not have any siblings, and that include exactly three subterms labeled `tag2`, `tag3` and `tag1` in the given order. No further restrictions are placed on these innermost subterms. The first tranlsation $Expr_{4.1.2^a_{XQ}}$ checks that the number of subterms of the root node is exactly one, and that the number of child elements on the second level equals three. Furthermore it checks that the node names of the elements on the second level equal `tag2`, `tag3` and `tag1` respectively.

$Expr_{4.1.2^a_{XQ}} :=$

```
element xc:substitution_set {
  for $v1 in $data/child::tag0 return
    for $v2 in $v1/child::tag1 return
      let $sequence = fs:distinct-document-order($v2/child::*) return
        if ((fn:count($sequence) == 3) and
           (fn:node-name($sequence[1]) == tag2) and
           (fn:node-name($sequence[2]) == tag3) and
           (fn:node-name($sequence[3]) == tag1))
        then element xc:substitution { element xc:X { $v2 } } else ()
```

In $Expr_{4.1.2^a_{XQ}}$, the `distinct-document-order` function is used to ensure that the children of `$v2` appear in document order in `$sequence`. In a second and third step, the number of children is checked using the `fn:count` function and the labels are tested for equality. Note that numeric predicates on lists (e.g. `$sequence[2]`) are not part of the XQuery core, but are normalized at the aid of the `subsequence` function to `fn:subsequence($sequence,2,1)`.

While this is tranlsation is very straightforward, there is another interesting translation that is also easily adaptable to translating incomplete ordered query terms, and that can do without predicates. It is represented by $Expr_{4.1.2^b_{XQ}}$.

$Expr_{4.1.2^b_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    if (fn:count($v0/child::*) == 1) then
      for $v1 in $v0/child::tag1 return
        if (fn:count($v1/child::*) == 3) then
          if (some $v2 in $v1/child::tag2 satisfies (
            some $v3 in $v1/child::tag3 satisfies (
              ($v2 << $v3) and
              (some $v4 in $v1/child::tag1 satisfies ($v3 << $v4)))))
          then element xc:substitution { element xc:X { $v1 } }
          else ()
    else ()
```

Instead of specifying the exact position of child subterms, they are compared with the `<<` operator, which is also not part of the XQuery core, but normalized to the function `fs:node-before`. Obviously, restricting the number of subterms to one single value and demanding that there exist the same amount of label-matching totally ordered subterms, equates to restricting the number of children and checking the labels for the subterms at each position.

A disadvantage of this method is that dispensable computations are carried out, because the `for` loops generate a great number of tuples of which the major part is filtered out. On the other hand, the close resemblance of this translation method to not only the translation of incomplete ordered query terms, but also to that of unordered query terms gives rise to a homogenous treatment of all term specifications in the automatic translation process (4.3).

In fact the only difference in translating ordered and unordered query terms is that for the former ones order constraints are generated and for the later injectivity constraints. In general more injectivity constraints need to be generated for the same amount of label overlapping subterms, which is due to the transitivity of the

<< operator: From `$a << $b` and `$b << $c` one can deduce `$a << $c` while it is not possible to deduce `$a != $c` from `$a != $b` and `$b != $c`. As discussed during the treatment of XC_2, it is necessary to produce $n(n-1)/2$ injectivity constraints for ensuring injectivity among n label-overlapping subterms, however, $n-1$ order constraints suffice to ensure a total ordering – and thus injectivity – among n ordered label-overlapping subterms.
As mentioned above, translating ordered incomplete query terms is very similar to this second translation method for ordered complete query terms. For ordered incomplete query terms the `fn:count` constraint on the number of children is simply omitted.

An efficiency improvement for translations of both complete and incomplete ordered query terms would be to check after each variable assignment in a for-clause, whether the number of following siblings of the assigned element nodes suffices for the rest of the query subterms to be matched. A more advanced optimization would be to not generate bindings to such nodes in the first place as exemplified in $Expr_{4.1.2^c_{XQ}}$. Note that the `fn:subsequence` function may have two or three arguments. The first argument is the input sequence, the second one the starting position, and the optional third one is the length of the result sequence. In absence of the third argument, all nodes until the end of the input sequence are returned.

$Expr_{4.1.2^c_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    if (fn:count($v0/child::*) == 1) then
      for $v1 in $v0/child::tag1 return
        let $seq := fs:distinct-document-order($v2/child::*) return
          if (fn:count($v1/child::*) == 3) then
            if (some $v2 in $v1/child::tag2 satisfies (
              some $v3 in fn:subsequence($seq, 2)/self::tag3
                satisfies (($v2 << $v3) and
                (some $v4 in fn:subsequence($seq,3)/self::tag1
                  satisfies ($v3 << $v4)))))
            then element xc:substitution { element xc:X { $v1 } }
            else ()
    else () }
```

Single curly braces are the final type of brackets to be discussed in this section. The difference between translating single curly braces and double curly braces is the same as the difference in the translation of single square brackets and double ones: One just needs to check that the number of subterms of the data node equals the number of subterms of the query term. This is achieved again with the `fn:count` function.

4.1.3 Translating `without`

Another important Xcerpt construct is `without` and therefore it is included in XC_3. Recall that `without` only makes sense with double square or curly braces. All subterms not mentioned within a breadth-complete parent term may not occur within simulating data terms anyway. The translation of `without` follows exactly the definition of the decomposition rules for query terms with negated subterms. After successfully searching for simulating data terms for all the positive subterms of the query term, it is checked whether any simulating nodes for the negated subterms that fulfill the order, deep-equality constraints and injectivity constraints exist. Only if this check fails, the simulation succeeds and a substitution is returned. As a first example consider $Expr_{4.1.3_{XC}}$.

$Expr_{4.1.3_{XC}} :=$ `var X as tag0{{ tag1{{ }}, without tag2{{ }}, tag2{{ }} }}`

It matches with all those nodes in the data that are named `tag0`, have a child node named `tag1`, exactly one named `tag2` and arbitrary siblings. A straightforward translation of this query would be $Expr_{4.1.3_{XQ}}$. In the first and second some clause, the positive children of `$v0` are simulated. If this succeeds, the negated subterm `without tag2 {{ }}` is tried to be simulated, taking the necessary injectivity constraints into account. Injectivity constraints for the pairs of siblings `($v1, $v2)` and `($v1, $v3)` are not necessary, since their node names do not coincide. Only if this simulation for `$v3` fails, the entire simulation succeeds.

$Expr_{4.1.3^a_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    if some $v1 in $v0/child::tag1 satisfies true then
      if some $v2 in $v0/child::tag2 satisfies true then
        if (fn:not(some $v3 in $v0/child::tag2 satisfies
           fn:not(op:is-same-node($v3, $v2)))) then
             element xc:substitution { element xc:X { $v0 } }
        else ()
      else ()
    else ()
```

If there were more than one negated query subterm, the translation would have to assure that the simulations for these other subterms fail as well. To see this, imagine that also the first subterm of $Expr_{4.1.3_{XC}}$ were negated. For the resulting expression, $Expr_{4.1.3^b_{XQ}}$ would be a correct translation.

$Expr_{4.1.3^b_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    if (some $v1 in $v0/child::tag2 satisfies (
      fn:not(some $v2 in $v0/child::tag1 satisfies true) and
      fn:not(some $v3 in $v0/child::tag2 satisfies
        fn:not(op:is-same-node($v3, $v2)))))
    then element xc:substitution { element xc:X { $v0 } }
    else () }
```

$Expr_{4.1.3^b_{XQ}}$ differs from $Expr_{4.1.3^a_{XQ}}$ in several ways: The first subterm of the original Xcerpt expression is simulated later than the third one. This is due to the necessity of first searching for a valid mapping for all the positive subterms (the third subterm is the only positive one) and then checking the absence of negated subterms. In contrast to $Expr_{4.1.3_{XQ}}$, the absence (rather than the existence) of the first subterm must be checked by encapsulating its simulation in a `fn:not` function. The third and final deviation is that there are not as many `if`-clauses. Note that also expression $Expr_{4.1.3_{XQ}}$ could get by with less `if`-clauses by transforming code of the form

```
if some <BINDING1> satisfies true then
   if some <BINDING2> satisfies true then <CONSEQUENCE> else ()
else ()
```

into

```
if some <BINDING1> satisfies (some <BINDING2> satisfies true)
then <CONSEQUENCE>
else ().
```

The `satisfies` conditions in general being some other boolean expression than just `true`, it is desirable to also simplify these expressions. This can be achieved by `and`-connecting the outermost `satisfies`-condition and the condition of the second `if`-clause to form the new condition of the outermost `satisfies`-clause. In this manner, the second `if`-clause can be eliminated. In the automatic translation process described later in this chapter, these abbreviations are used whenever applicable.

Two other potentially interesting cases of Xcerpt expressions including negated subterms, namely the ones of ordered query terms together with `without` and the one of nested `without` turn out to be translatable straightforwardly: The only difference in translating square brackets is to use order constraints instead of injectivity constraints. Similarly, nested `without` is translated with nested negated existential quantifications, as demonstrated by the following two expressions.

```
var X as tag0 {{ without tag1 {{ without tag2 }} }}
```

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    if fn:not(some $v1 in $v0/child::tag1 satisfies (
       fn:not(some $v2 in $v1/child::tag2 satisfies true)))
    then element xc:substitution { element xc:X { $v0 } }
    else ()
}
```

In this last expression, the second some clause is trivial in the sense that its satisifes clause consists only of the boolean constant `true`. As a matter of fact, any clause of the form `some $var in <STEP> satisfies true` may be substituted by `<STEP>`. This is because the node-sequence which the step expression evaluates to, is automatically transformed into a boolean value in a boolean context. If it is the empty sequence, the result is `false`, in case its first element is a node, the result is `true`. This transformation is called finding *the effective boolean value* of an expression [2, Section 2.4.3].

4.1.4 `without` and Variables

As described in [17], variables occurring within the scope of a `without` never yield variable bindings, but must be considered as mere constraints of its siblings and of its enclosing term. Two consequences can be drawn from this fact: First, each variable within a `without` must be bound elsewhere in the query term. Second, expressions of this kind cannot be translated straightforwardly as in the examples above. To see this, consider expression $Expr_{4.1.4_{XC}}$. Before translating the negated subterm `without var X as tag2`, one has to make sure that all defining occurrences of variable X have already been treated. Hence, the usage of `without` in conjunction with variables imposes a restriction on the order of translating subterms to XQuery. The correct translation $Expr_{4.1.4_{XQ}}$ first associates the XQuery variable `$v2` with the Xcerpt subterm `var X as tag2` before checking for the existence of a child data term within `$v1` that simulates with X. Range restrictedness of Xcerpt query terms guarantees that there is always a positive occurrence for each variable.

$Expr_{4.1.4_{XC}} :=$ `tag0 {{ tag1 {{ without var X as tag2 }}, var X as tag2 }}`

$Expr_{4.1.4_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag1 return
    for $v1 in $v0/child::tag2 return
      if (some $v2 in $v0/child::tag1 satisfies (
        fn:not(some $v3 in $v2/child::tag2 satisfies
          fn:deep-equal($v3,$v1))))
        then element xc:substitution { element xc:X { $v1 } }
        else ()
```

4.1.5 Order and Injectivity Constraints Among Multiple Negated Subterms

Injectivity constraints were thouroughly discussed during the treatment of XC_2, and order constraints in Section 4.1.2. The introduction of the `without` construct complicates the question about whether to produce constraints for a given pair of siblings. Consider $Expr_{4.1.5_{XC}}$. Let `$v0` be the XQuery variable representing the root query term, and `$v1`, `$v2` and `$v3` the variables for the three subterms. In the absence of negations, order constraints would be produced for the pairs of variables `($v1, $v2)` and `($v2, $v3)`. An order constraint between the last two subterms in $Expr_{4.1.5_{XC}}$ does not make sense, however, because the successfull simulation of only one of the two terms causes the entire simulation to fail. Nevertheless it is important to ensure the order among the first and last subterm, such that the pair of order constraints `($v1, $v2)`, `($v1, $v3)` must be generated to enforce Xcerpts intended semantics. The correct translation of $Expr_{4.1.5_{XC}}$ is therfore given by $Expr_{4.1.5_{XQ}}$.

$Expr_{4.1.5_{XC}} :=$

```
tag0 [[ var X as tag1 {{ }}, without tag2 {{ var X }},
        without tag2 {{ }} ]]
```

$Expr_{4.1.5_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::f return
    for $v1 in $v0/child::a return
      if (fn:not(some $v2 in $v0/child::b satisfies ($v1 << $v2)))
      then
        if (fn:not(some $v3 in $v0/child::c satisfies ($v1 << $v3)))
        then
          element xc:substitution { element xc:X { $v1 } }
        else ()
      else ()
}
```

4.1.6 Translating `optional`

Regarding the importance of querying possibly incomplete and heterogenous data on the web, the `optional` construct deserves to be discussed here. Declaring a subterm as `optional` means that it shall be tried to simulate the subterm with a subterm of the data term, but that the simulation need not fail if no simulating data term can be found. Using the `optional` keyword in an Xcerpt query term demands that the same variable be marked `optional` also in the construct part of a construct query rule. Furthermore, optional query terms must always include a variable binding.

4.1.6.1 Optional Terms without Interrelations between Variables

A single optional term is translated by means of a `for`-clause just in the same way as it would be without the `optional` marker. In the case that there is no data term that the query term simulates with, nothing will be returned. This simple translation is feasible, because the `optional` is rather insignificant in expressions with a single variable occurrence such as $Expr_{4.1.6.1_{XC}}$ – it would have almost the same meaning without the `optional`. The only difference between including and omitting the `optional` is that in the case of no subterm matching `var X as tag1 {{ }}`, the usage of `optional` would prevent the query term from failing, returning the empty subtitution set instead. In other words, the usage of optional subterms is more beneficial in the case of multiple variables within a query term.

$Expr_{4.1.6.1_{XC}} :=$ `tag0 {{ optional var X as tag1 {{ }} }}`

$Expr_{4.1.6.1_{XQ}} :=$

```
element xc:substitution_set {
  for $v1 in $data/child::tag0 return
    for $v2 in $v1/child::tag1 return
      element xc:substitution { element xc:X { $v1 } }
```

4.1.6.2 Optional Terms with Related but Distinct Variables

In contrast to the example above, the construct query Rule $Expr_{4.1.6.2_{XC2}}$ may actually produce heterogeneous substitutions. They may either include bindings for all three variables, only for variable `X` and `Y`, only for variables `X` and `Z`, or only for variable `X`. Therefore the translation is not as trivial as above. A possible translation checks for each binding of variable `X` if there exist siblings with tag name `tag2` and `tag3`. Depending on the result of these tests, different substitutions are returned. Bindings for variable `Y` are only included if the test for `tag2` succeeds, those for `Z` only if the test for `tag3` succeeds.

$Expr_{4.1.6.2_{XC}} :=$

```
tag0 {{ var X as tag1,
        optional var Y as tag2,
        optional var Z as tag3
     }}
```

$Expr_{4.1.6.2_{XQ}} :=$

```
element xc:substitution set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      if ($v0/child::tag2) then
        for $v2 in $v0/child::tag2 return
          if ($v0/child::tag3) then
            for $v3 in $v0/child::tag3 return
              element xc:substitution {
                element xc:X { $v1 },
                element xc:Y { $v2 },
                element xc:Z { $v3 } }
          else element xc:substitution {
            element xc:X { $v1 },
            element xc:Y { $v2 } }
      else if ($v0/child::tag3) then
        for $v4 in $v0/child::tag3 return
          element xc:substitution {
            element xc:X { $v1 },
            element xc:Z { $v4 } }
      else element xc:substitution { element xc:X { $v1 } }
}
```

$Expr_{4.1.6.2_{XQ}}$ exposes that the length of an XQuery translation of of a query term including `optional` grows exponentially with the number of `optional` in the original Xcerpt expression. The formal justification for this is that for each optional variable, two substitutions must be returned – one including the optional variable and one without. Hence the number of different possible substitutions is 2^o where o is the number of optional subterms.

4.1.6.3 Transforming `optional` into `without`

A different, elegant approach to translating $Expr_{4.1.6.2_{XC}}$ is to first convert query terms featuring optional subterms into a disjunction of query terms with negated subterms, as exemplified in [17]. $Expr_{4.1.6.3_{XC}}$ is the result of transforming $Expr_{4.1.6.2_{XC}}$ in this manner. Having already examined the possibilities for translating without, the translation of $Expr_{4.1.6.3_{XC}}$ is rather straightforward. The only remaining question about how to translate disjunctions of query terms will be treated in Section 4.1.7.1.

$Expr_{4.1.6.3_{XC}} :=$

```
or ( tag0 {{ var X as tag1, var Y as tag2, var Z as tag3 }},
     tag0 {{ var X as tag1, without tag2,  var Z as tag3 }},
     tag0 {{ var X as tag1, var Y as tag2, without tag3  }},
     tag0 {{ var X as tag1, without tag2,  without tag3  }} )
```

Note that the variables Y and Z must be left away in the negated subterms, because they do not occur unnegatedly elsewhere in the expression. If they were bound in some other part of the query term, we would need to include them in the negated subterm. This issue is deepened in the following section.

At first glance, the fact that one may use `optional` but not `without` within breadth-complete query terms causes a problem. Is it still possible to transform breadth-complete query terms containing `optional`? The solution is quite simple: The query term is transformed just as if it were breadth incomplete, but the negated subterms are omitted.

4.1.6.4 Variables within Multiple `optionals`

This section lifts the restriction from the last Section that a variable appears within at most one optional subterm. In the case that a variable occurs within more than one optional subterm, the order of simulating optional subterms with data terms affects the resulting substitution sets. To see this consider expression $Expr_{4.1.6.4_{XC}}$.

$Expr_{4.1.6.4^{a}_{XC}} :=$ `tag0 {{ optional var X as tag1, optional var X as tag2 }}`

Querying the data `tag0 [ tag1 [], tag2 [] ]` with $Expr_{4.1.6.4_{XC}}$ we would expect the substitution set `{ {X` $\mapsto$ `tag1[]},{X` $\mapsto$ `tag2[]} }`. An intuitive translation like $Expr_{4.1.6.4^{a}_{XQ}}$, however, yields only the first one of the substitutions. This problem originates from scheduling a stiff simulation order. The query subterms are simulated in the order they appear in the query term. Thus the subterm `var X as tag1` is simulated first, succeeding with the example data. Now there's no chance for the second subterm to succeed, because the variable constraints for `X` are incompatible.

$Expr_{4.1.6.4_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    if ($v0/child::tag1) then
      for $v1 in $v0/child::tag1 return
     if (some $v2 in $v0/child::tag2 satisfies
          fn:not(op:is-same-node($v2, $v1)) and
          fn:deep-equal($v2, $v1))
        then for $v2 in $v0/child::tag2 return
          element xc:substitution { element xc:X { $v2 } }
        else
          element xc:substitution { element xc:X { $v1 } }
    else if ($v0/child::tag2) then
      for $v3 in $v0/child::tag2 return
        element xc:substitution { element xc:X { $v3 } }
    else ()
}
```

Interestingly, this problem is solved by transforming the `optional` into `without` as exhibited in $Expr_{4.1.6.4^{b}_{XC}}$. It is obvious that the first disjunct of $Expr_{4.1.6.4^{b}_{XC}}$ cannot simulate with any data term. The expected substitution set is generated by the second and third disjunct, each of which contribute one substitution. The last disjunct does not simulate with the data. A final noteworthy issue is that `var X` should be left out in the last disjunct if $Expr_{4.1.6.4^{a}_{XC}}$ is not part of a larger query or query term. Otherwise it might happen, that a defining occurrence of `X` appears elsewhere in the query term.

$Expr_{4.1.6.4^{b}_{XC}} :=$

```
or ( tag0 {{ var X as tag1, var X as tag2 }},
     tag0 {{ without var X as tag1, var X as tag2 }},
     tag0 {{ var X as tag1, without var X as tag2 }},
     tag0 {{ without var X as tag1, without var X as tag2 }} )
```

4.1.6.5 Nested `optional` Terms

A further interesting case is the one of nested optional terms as in expression $Expr_{4.1.6.5^{a}_{XC}}$.

$Expr_{4.1.6.5^{a}_{XC}} :=$

```
tag0 [[ var X as tag1 [[]],
  optional var Y as tag2 [[
    optional var Z as tag3 [[]]
  ]]
]]
```

Apparently, it may not occur that a substitution resulting from this query term contains a binding for variable `Z` and at the same time no binding for `Y`. To reflect this the translation of $Expr_{4.1.6.5_{XC}}$ does not try to find a simulating node for the subterm including variable `Z`, when the simulation for `var Y as tag2 [[ ... ]]` fails. The `else`-clause in question simply returns a substitution with `X` as the only included variable. Note that there is no need to ponder about the simulation order within $Expr_{4.1.6.5_{XQ}}$, because there is no variable that occurs within multiple optionals.

$Expr_{4.1.6.5_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      if ($v0/child::tag2) then
        for $v2 in $v0/child::tag2 return
          if ($v2/child::tag3) then
            for $v3 in $v2/child::tag3 return
              element xc:substitution {
                element xc:X { $v1 }
                element xc:Y { $v2 }
                element xc:Z { $v3 } }
          else element xc:substitution {
            element xc:X { $v1 }
            element xc:Y { $v2 } }
      else element xc:substitution { element xc:X { $v1 } } }
```

An alternative approach to translating $Expr_{4.1.6.5^{a}_{XC}}$ is to first transform the optionals into without as has been shown above. With nested `optional` terms this is not as canonical as before. Similarly to the direct translation, the disjunct that binds variable Z and negates the term that would be bound to Y must be omitted, otherwise the translation would not even be range restricted, with variable Z only occurring in a negated subterm.

$Expr_{4.1.6.5^{b}_{XC}} :=$

```
or (
  tag0 [[ var X as tag1 [[]], var Y as tag2 [[ var Z as tag3 [[]] ]],
  tag0 [[ var X as tag1 [[]], var Y as tag2 [[ without tag3 [[]] ]],
  tag0 [[ var X as tag1 [[]], without tag2 [[ ]] )
```

Summing up the translation possibilities of `optional`, it needs to be emphasized that there is no way of translating Xcerpt expressions including `optionals` to XQuery such that the length of the translation scales linearly with the length of the archetype. The revealed transformation method does not solve this complexity problem, but presents a viable alternative for directly translating optional subterms.

4.1.7 From Query Terms to Queries: Translating `and`, `or` and `not`

As exemplified by the transformation of `optional` to `without` in the previous section, disjunctions – and also conjunctions and negations – are important extensions to simple query terms. Furthermore translating certain XQuery expressions that lack injectivity constraints is outright impossible without conjunctions and disjunctions. Thus it is important to discuss these constructs.

4.1.7.1 Translating `or`

Reconsider $Expr_{4.1.6.5^{b}_{XC}}$, which is the result of the elimination of the optional subterms of $Expr_{4.1.6.5^{a}_{XC}}$. How is this disjunction of query terms translated to XQuery? Let $\mathcal{Q} := or\{\mathcal{Q}_1, \ldots, \mathcal{Q}_n\}$ be a disjunction of query terms. As Schaffert states in his thesis, `or` "merely merges the resulting sets of substitutions resulting from the queries $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$" [17, Section 4.5.2]. As an XML representation of the substitution set, it is thus sufficient to append the translations of the $\mathcal{Q}_i$ within the same enclosing `xc:substitution_set` element as shown in $Expr_{4.1.7.1_{XQ}}$.

$Expr_{4.1.7.1_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      for $v2 in $v0/child::tag2 return
        for $v3 in $v2/child::tag3 return
          element xc:substitution {
```

```
            element xc:X { $v1 },
            element xc:Y { $v2 },
            element xc:Z { $v3 } },
    for $v0 in $data/child::tag0 return
      for $v1 in $v0/child::tag1 return
        for $v2 in $v0/child::tag2 return
          if fn:not($v2/child::tag3) then
            element xc:substitution {
              element xc:X { $v1 },
              element xc:Y { $v2 } }
          else (),
    for $v0 in $data/child::tag0 return
      for $v1 in $v0/child::tag1 return
        if fn:not($v1/child::tag2) then
          element xc:substitution {
            element xc:X { $v1 } }
        else ()
}
```

Note that the injectivity of nodes `$v1` and `$v2` is not an issue, because their labels differ. Repeatedly binding the same variable names `$v0`, `$v1`, and `$v2` is not a problem in $Expr_{4.1.7.1_{XQ}}$, because their scopes do not overlap. Prescinding from $Expr_{4.1.6.5^{b}_{XC}}$ and taking into account also those disjunctions that may yield equal substitutions for different disjuncts, it may be argued that just appending the substitutions from different disjuncts will result in duplicates in the substitution set. Though this is certainly true, duplicates may also be returned by single query terms, and this is accepted. In many cases it is not satisfactory to eliminate duplicates. This is why duplicates in this thesis are not eliminated during query translation, but only when explicitly asked for in construct terms – for example by using the duplicate eliminating `all`-construct. As a consequence, substitution sets in this thesis are not sets in the mathematical meaning of the word, but rather multisets or bags of substitutions.

As a final observation, the presented treatment of disjunctions works fine also for the special cases that the number of disjuncts is zero or one. For $n = 0$ the empty substitution set is returned, and for $n = 1$ the result coincides with the isolated translation of $\mathcal{Q}_1$.

4.1.7.2 Translating and

Connecting query terms with `and` is especially useful when querying different XML sources with the `resource` construct. Apart from that `and` is needed to translate certain XQuery expressions that lack some order or injectivity constraints for pairs of siblings to Xcerpt. As a trivial example for an `and`-connected query, consider $Expr_{4.1.7.2_{XC}}$, which queries all those `tag1` subterms of a root node named `tag0` that also occur within a root node named `tag1` and include a subterm named `tag2`.

$Expr_{4.1.7.2_{XC}} :=$

```
and (
  tag0 {{ var X as tag1 }},
  tag1 {{ var X }}
)
```

$Expr_{4.1.7.2_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      for $v2 in $data/child::tag1 return
        for $v3 in $data/child::* return
          if (fn:deep-equal($v1, $v3))
          then element xc:substitution {
            element xc:X { $v1 } }
          else () }
```

The translation $Expr_{4.1.7.2_{XQ}}$ differs from translations of disjunctions of queries in that an `xc:substitution` element appears in only one `return` clause, whereas in $Expr_{4.1.7.1_{XQ}}$ the number of `return`-clauses with `xc:substitution`-elements equals the number of disjuncts in the corresponding Xcerpt query.

The translation procedure can be described as follows: According to a *selection strategy* one of the conjuncts is chosen. This conjunct is translated almost in the same way as if it were a single query term. The case in which it is not a query term but a query, is precluded for the moment. The only difference in translating this first conjunct from translating a single query term is that the final return clause does not return the XML representation of a substitution set, but the translation of the conjunct which is chosen next according to the selection strategy. Only after translating the final conjunct the substitution, including bindings for the union of all variables of all conjuncts, is returned. But what happens if the conjunct is a query? The case that it contains a negated query term is treated in the following section, while it is easy to take measures to circumvent the case of disjunctions in the conjunct.

The selection strategy mentioned above needs to ensure that conjuncts including defining occurrences of a variable are translated before conjuncts with consuming occurrences of the same variable.

It is easy to adapt the translation algorithm presented during the treatment of XC_2 to include also conjunctions of query terms. Usually, the translation algorithm started out with one single step represented by a triple including the special variable `$data`, a fresh variable name `$v0`, and the query term to be translated. When translating a conjunction of queries the list of triples is not intitialized with a single element but one for each conjunct. The parent variable for each query term is `$data` and a fresh variable name for each of the conjunct query terms is reserved. This is also exactly the way the translation of conjunctions is handled in Section 4.3.

The final question which remains to be answered in this sectionis how to guarantee that a conjunct which is to be translated, does not contain a disjunction of query terms. The answer is to bring the query in disjunctive normal form by applying the distributive law:

$$\begin{aligned} &and(or(\mathcal{Q}_1, \ldots, \mathcal{Q}_n), \mathcal{Q}_{n+1}, \ldots, \mathcal{Q}_{n+m}) = \\ &\quad or(and(\mathcal{Q}_1, \mathcal{Q}_{n+1}, \ldots, \mathcal{Q}_{n+m}), \ldots, and(\mathcal{Q}_n, \mathcal{Q}_{n+1}, \ldots, \mathcal{Q}_{n+m})) \end{aligned}$$

4.1.7.3 Translating `not`

Semantically, `not` is very similar to `without`. The only difference is that `without` may only occur within query terms, while `not` serves to negate queries, usually within a conjunction of queries. Assuming that all `not` is free of conjunctions and disjunctions, the translation procedure – which is treated in Section 4.3 – does not differentiate between query term negation and query negation. De Morgan's laws can be used to transform queries such that query negations only appear as the innermost constructs of `and`, `or` and `not`.

$$\begin{aligned} not(and(\mathcal{Q}_1, \mathcal{Q}_2)) &= or(not(\mathcal{Q}_1), not(\mathcal{Q}_2)) \\ not(or(\mathcal{Q}_1, \mathcal{Q}_2))) &= and(not(\mathcal{Q}_1), not(\mathcal{Q}_2)) \end{aligned}$$

As an example of a conjunction that includes a negated query term, consider a slight variation of $Expr_{4.1.7.2_{XC}}$:

$Expr_{4.1.7.3_{XC}} :=$

```
and (
  not(tag1 {{ var X as tag1 {{ tag2 {{ }} }} }}),
  tag0 {{ var X as tag1 }}
)
```

Similarly to negated and positive subterms with common variables within ordinary query terms, a correct translation must first translate the second query term, because it yields a variable binding for `X`, whereas in the first query term there is only a consuming occurrence of `X`. This is reflected in the following translation.

$Expr_{4.1.7.3_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::tag1 return
      if fn:not(some $v2 in $data/child::tag1 satisfies(
```

Table 4.2: Grammar productions for XQ_3

```
<XQ_3>  ::= element xc:substitution_set { <EXPR> (, <EXPR>)* }
<EXPR>  ::= <FOR> | <IF> | <SUBST>
<FOR>   ::= for <VARIABLE> in (<CHILD>|<DESC>) return <EXPR>
<IF>    ::= if (<BEXPR>) then <EXPR> else ()
<BEXPR> ::= <INJ> | <DEQ> | <BEF> | <COUNT> |
            <SOME> | <AND> | <NOT> | true
<SUBST> ::= element xc:substitution { <BINDG> }
<BINDG> ::= element xc:<XCVAR> { <VARIABLE> }
<CHILD> ::= <VARIABLE>/child::<QNAME>
<DESC>  ::= <VARIABLE>/descendant::<QNAME>
<INJ>   ::= fn:not(op:is-same-node(<VARIABLE>, <VARIABLE>))
<DEQ>   ::= fn:deep-equal(<VARIABLE>, <VARIABLE>)
<BEF>   ::= fn:node-before(<VARIABLE>, <VARIABLE>)
<SOME>  ::= some <VARIABLE> in (<Child> | <DESC>) satisfies <BEXPR>
<AND>   ::= <BEXPR> and <BEXPR>
<NOT>   ::= fn:not(<SOME>)
<COUNT> ::= fn:count(<VARIABLE>/child::*) == <INTEGER>
```

```
      some $v3 in $v2/child::tag1 satisfies ($v3/child::tag2)
        and fn:deep-equal($v1, $v3)))
    then element xc:substitution {
      element xc:X { $v1 } }
    else () }
```

4.2 The Sublanguage XQ_3

As mentioned in the introduction of this chapter, XQ_3 is a proper superlanguage of XQ_2. The following constructs are included in XQ_3, but not in XQ_2:

- The `descendant` axis is necessary to translate the `desc`-construct of Xcerpt.
- In addition to injectivity constraints, order constraints such as `fn:node-before($v1, $v2)` are allowed to enforce Xcerpt order constraints in XQuery.
- Constraints on the number of children are put into effect by the `fn:count()` function. Arguments to this function in XC_3 must be of the form `<VARNAME>/child::*`.
- The `and` connective is a shorthand for nested `if`-clauses as has been shown in Section in 4.1.3. Therefore it is not strictly necessary to include `and` in XQ_3, but it improves the readability of the translations. The other newly added boolean operator `fn:not` serves to translate query negation (`not()`) and query term negation (`without`). Note that while `not()` was already used in injectivity constraints in XC_2, it may be used to negate any boolean expression produced by `<BEXPR>` in the grammar productions for XC_3 (Table 4.1).
- The outermost XQuery element constructor, namely the `xc:substitution_set` element constructor, may contain a *sequence* of expressions, which allows the translation of Xcerpt query disjunctions.

All other constructs are already part of XQ_2. All expressions within XQ_3 can be derived by the grammar in Table 4.2, but some additional restrictions are placed upon valid XQ_3 expressions.

4.2.1 Grammar Productions

In Table 4.2 `<QNAME>` is a qualified name as defined in [2], `<VARIABLE>` is a valid XQuery variable name, `<XCVAR>` is an Xcerpt variable name. An expression E in XC_3 must additionally satisfy the following restrictions:

- XQuery variables are unique, meaning the same XQuery variable name is bound only once within a valid XQ_3 expression. This constitutes only a syntactical restriction, since expressions infringing this constraint can be easily transformed into a valid XQ_3 expression by renaming variables.
- Order constraints are only allowed for variables referring to siblings. It would be possible to translate unrestricted order constraints to XC_3 but this is not considered in this thesis. For injectivity constraints this restriction is lifted, since children of different nodes in tree structure are always distinct.
- A restriction inherited from XC_2 is that the `else` clause of conditionals is always the empty sequence. It would be interesting to examine a sublanguage freed from these last two restrictions and check whether XC_3 may be used to express its translations, but this is not treated in this thesis.
- For each XQuery variable, there must be at most one count constraint. It would also be possible to lift this restriction, ignoring all but one of the `fn:count` functions in the translation, if they coincide, and restituting `False` if there are contradicting constraints on the number of children of a variable.

4.2.2 Translating Partial Injectivity Constraints

When translating Xcerpt query terms to XQuery, injectivity is preserved either by generating injectivity or order constraints depending on the type of brackets used. It is easy to reverse this translation process if all required injectivity constraints are present. *Partial* or even absent injectivity constraints as in $Expr_{4.2.2_{XQ}}$ complicate this reversal.

$Expr_{4.2.2_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::* return
      for $v2 in $v0/child::* return
        element xc:substitution {
          element xc:X { $v1 }, element xc:Y { $v2 } } }
```

$Expr_{4.2.2^a_{XC}} :=$ `tag0 {{ var X, var Y }}`

If this query were translated using $Expr_{4.2.2^a_{XC}}$, substitutions mapping `$v1` and `$v2` to the same node would be mistakenly eliminated. Note that this problem could not occur if the tag names of both siblings were different. The correct translation needs to take into account both cases: Either `$v1` and `$v2` are bound to distinct nodes, or to the same nodes. Both of these cases are connected via a disjunction:

$Expr_{4.2.2^b_{XC}} :=$ `or (tag0 {{ var X, var Y }}, tag0 {{ var X }})`

The problem with $Expr_{4.2.2^b_{XC}}$ is that there are substitutions which do not include bindings for variable `Y`. In all these substitutions `Y` should be mapped to the same value as `X`. Range restrictedness requires all non-optional variables which appear in the construct part of a rule to be mandatory (non-optional) in all disjuncts of the query part of the rule. This means it would not be allowed to use `Y` in the construct part of a rule with query part $Expr_{4.2.2^b_{XC}}$. A solution to this problem is to mark `Y` optional both in the query part and the construct part. Occurrences of `Y` in the construct part may further be extended by a default value `var X`, which is exemplified by the following rule:

$Expr_{4.2.2^c_{XC}} :=$

```
CONSTRUCT
  xc:substitution_set [
    all xc:substitution [ xc:X [ var X ],
      xc:Y [ optional var Y with default var X] ] ]
FROM
  or (tag0 {{ var X, var Y }},
      tag0 {{ var X, optional var Y as xyz:abc {{ }})
END
```

A singularity of $Expr_{4.2.2.2^c_{XC}}$ is the namespace prefix `xyz` before `tag1`. The namespace bound to `xyz` must be uniquely chosen in order to make sure that this second subterm cannot accidentally match. It is only included to make $Expr_{4.2.2.2^c_{XC}}$ range restricted. Range restrictedness demands that all optional terms in the construct part appear also negatively (defining) in each disjunct of the query part. Alternatively, range restrictedness could be ignored and this subterm could we left out.

An obvious disadvantage of this kind of translation is that it causes interdependencies between the translation of construct and query parts. Furthermore it benefits from `optional`, which is actually not part of the language XC_3. This is why other translations have to be found. Last but not least it is not extensible to the case of a variable lacking more than one injectivity constraint.

An obvious alternative is to put aside injectivity by using the `and` connective of queries:

$Expr_{4.2.2.2^d_{XC}} :=$ `and (tag0 {{ var X }}, tag0 {{ var Y }})`

This translation works fine as long as there is only one rooted tree, e.g. an XML document, associated with the query. If the query term is part of a rule which participates in rule chaining, however, it may occur that the disjuncts simulate with different rooted trees. This means there is no way of knowing that the terms referred to by `X` and `Y` belong to the same parent node, which was clearly the semantics of the XQuery archetype. Luckily, the translation is rescued by the possibility of using `and` to connect not only multiple queries, but also query terms.

While it is being discussed to allow `and` to connect query terms, this has not officially been added to the Xcerpt standard.

4.2.2.1 Making use of the `and` Connective for Query Terms

Keep in mind that the semantics of the `and` connective for query terms differs significantly from the semantics of the `and` connective for queries. In the case of queries, both conjuncts need to simulate with arbitrary data terms and the substitution sets are joined. The difference is that in the case of the conjuncts being query terms, the same data term must simulate with both of the conjuncts. Thus if $Expr_{4.2.2.2^d_{XC}}$ is considered as a conjunction of query terms, the translation is correct. With the `and` connective for query terms, also more complex queries can be translated:

$Expr_{4.2.2.1_{XQ}} :=$

```
element xc:substitution_set {
  for $v0 in $data/child::tag0 return
    for $v1 in $v0/child::* return
      for $v2 in $v0/child::* return
        for $v3 in $v0/child::* return
          if (fn:not(op:is-same-node($v1, $v3)) and
             fn:not(op:is-same-node($v2,$v3))) then
               element xc:substitution {
                 element xc:X { $v1 },
                 element xc:Y { $v2 },
                 element xc:Z { $v3 } } }
```

In $Expr_{4.2.2.1_{XQ}}$ only the nodes bound to the variables `$v1` and `$v2` may coincide, all other pairs are guaranteed to be distinct. Using the same method as above, the correct translation is $Expr_{4.2.2.1_{XC}}$:

$Expr_{4.2.2.1_{XQ}} :=$ `and (tag0 {{var X, var Z}}, tag0 {{var Y, var Z}})a`

Prescinding from $Expr_{4.2.2.1_{XQ}}$, consider a query term with n children with *overlapping* tag names, and k injectivity constraints among these children. It is easy to see that $n + k$ is an upper bound for the number of conjuncts needed: Since the number of variables is n, one conjunct can be reserved for each variable, and as well one conjunct for each of the k injectivity constraints. At the same time, the approximation cannot be much better. Obviously, k is not an upper bound for the number of conjuncts, because also in the case of no injectivity constraints, each variable must appear in at least one conjunct. Furthermore, n is not an upper bound for the number of conjuncts for $n > 5$. To see this, consider the variables `U, V, W, X, Y, Z` and injectivity constraints among the pairs `(U,V), (V,W), (W,X), (X,Y), (Y,Z), (Z,W), (U,X)`.

Note that there is no triple of variables that is required to be bound to mutually distinct nodes and could therefore be contained in one single conjunct. As a result, every single constraint has to be realized by its own conjunct, which yields a total number of seven constraints for six variables.

There is, however, another upper bound for the number of required conjuncts in terms of the number of missing injectivity constraints $\frac{n\cdot(n-1)}{2}-k$. Only one conjunct, namely the one including all variables, is needed to express the presence of all injectivity constraints. This is evident, because this conjunct is the preimage of the translation of an unordered partial query term to XQuery. Upon deleting the injectivity constraint that distinguishes the variables `X1` and `X2`, the conjunct is split up into two conjuncts, one excluding the variable `X1` and the other excluding the variable `X2`. Taking away another injectivity constraint among variables distinct from `X1` and `X2` causes again the number of conjuncts to double. Taking away an injectivity constraint among variables that are not "fresh", causes only those conjuncts which contain both variables to divide. Thus, $2^{\frac{n\cdot(n-1)}{2}-k}$ is another upper bound for the number of required conjuncts. An optimized upper bound is given by the minimum of the first two:

$$min(n+k, 2^{\frac{n\cdot(n-1)}{2}-k}) \tag{4.1}$$

The discussion of the upper bounds for the number of conjuncts give rise for an algorithm to construct them. For a given number of subterms n and a given number of present injectivity constraints k it is determined which of the upper bounds is lower. In the case of few injectivity constraints, $n+k$ is less than $\frac{n\cdot(n-1)}{2}-k$, and the conjuncts are constructed by creating one conjunct for each injectivity constraint and adding single variable conjuncts for all those variables that do not occur within any injectivity constraint. In the case of almost complete injectivity constraints, one single conjunct is created including all variables, and for each missing injectivity constraint, the conjuncts are split as described above.

So far, the assumption has been made that the labels of all child subterms overlap, which is the worst case scenario. In general, there are only few overlapping labels, and most of the injectivity constraints – although not explicitly stated – are enforced by distinct tag names. Before translating XQ_3 queries with partial explicit injectivity constraints, it needs to be verified if there is any chance that injectivity may be violated. In the major part of every day queries, this is certainly not the case. Otherwise, the sets $C_1, \ldots, C_m$ of overlapping tag names as defined in 3.2.2 need to be calculated. An upper bound for the number of indispensable conjuncts is the sum of the application of Equation 4.1 to each of the C_i.

4.2.2.2 Making use of the `or` Connective for Query Terms

Similar to the `and` connective for query terms, the corresponding `or` has not been added to the official Xcerpt standard yet. Nevertheless, this sectionexamines whether it may be helpful for translating partial injectivity constraints. The idea is to distinguish for each missing injectivity constraint the cases that both variables refer to the same node and the case in which they refer to distinct nodes. While the first case is realized by an `and` connective, the second case can be represented by an ordinary Xcerpt query subterm. The following Xcerpt query is a correct translation of $Expr_{4.2.2.2_{XQ}}$. Note that in this case it is insignificant whether the `or` operator is a connective for query terms or entire queries.

$Expr_{4.2.2.2_{XC}} :=$
```
or (tag0 {{ and (var X, var Y) }}, tag0 {{ var X, var Y }})
```

Although $Expr_{4.2.2.2_{XC}}$ is adequately short, this translation method scales very poorly. As mentioned above, for each missing injectivity constraint it must be distinguished between the case that it would be violated (if it were present) and the case that the variables are assigned to distinct nodes. Thus, the number of required disjuncts rises exponentially with the number of missing injectivity constraints. As has been shown above, one can do better by only using the `and` connective.

4.2.3 Translating Partial Order Constraints

A very similar problem that may occur when translating from XQ_3 to XC_3 is that some of the order constraints are absent:

$Expr_{4.2.3.0_{XC}} :=$

```
element xc:substitution_set {
  for $v0 in $data/tag0 return
    for $v1 in $v0/* return
      for $v2 in $v0/* return
        for $v3 in $v0/* return
          if ($v2 << $v3) then
            element xc:substitution {
              element xc:X { $v1 }, element xc:Y { $v2 },
              element xc:Z { $v3 } }
          else () }
```

It turns out that also for these kinds of expressions the best translation method is the use of the `and` connective for query terms:

$Expr_{4.2.3.0_{XC}} :=$ `and (tag0 [[ var Y, var Z ]], tag0 [[ var X ]])`

Thanks to the transitivity of order constraints, the maximum number of conjuncts in the translations of partial order constraints is less than the number of required injectivity constraints above: In the case of no order constraints, one conjunct is reserved for each variable – this is the same case as the one without injectivity constraints. To enforce a total ordering among n label-overlapping subterms, $n - 1$ order constraints suffice.

4.3 Automatic Translation of XC_3 to XQ_3

In this and the following part of this Chapter automatic translation algorithms are introduced that generalize the translation of the example expressions from the last sections. This section discusses the function $toXQ$ that maps XC_3 queries to XQ_3, and is divided into two parts covering the translation of query terms and entire queries.

4.3.1 Translating XC_3 Query Terms

Translating query terms in XC_3 is achieved in a very similar fashion to the translation of query terms in XC_2, as presented in 3.5: Two kinds of rules are employed. The function $toXQ$ is the main translation function. It recursively calls itself multiple times during a translation, and each call produces a fragment of the result such as element nodes and `for`-clauses. In contrast, $toXQBool$ produces boolean XQuery expressions such as `some`-clauses, before constraints, injectivity constraints, deep-equality constraints, negations and conjunctions of boolean expressions. Both of these recursive functions are extended versions of the ones introduced during the treatment of XC_2 and XQ_2. $toXQ$ is equipped with an extra argument `Befs` (abbreviation for "befores") holding the order constraints.

With respect to the translation procedure for XC_2, there is only a slight difference in translating unordered breadth-incomplete subterms without negations and descendant constructs: Besides checking for ripe injectivity constraints, it must also be checked for ripe order constraints as can be seen in Rule 4.1. In this sectionrules are only provided for ordered query terms, breadth complete specifications, negated subterms and descendant constructs. Negated query terms are identified by the superscript "-", positive ones by a superscript "+", ordered ones by superscript square brackets, breadth-incomplete ones by double square or curly braces, query terms binding a variable are marked with the name of the variable, query terms that are descendant of their enclosing query term are flagged by `Desc`. To give an example, `without desc var X as tag1 [[ ]]` is denoted $\texttt{qt1}^{\texttt{[[]],-,X,Desc}}$.

The rules rely on the functions described in 3.5 and the following additional ones.

- `make_befs(vars)` transforms a list of XQuery variables `vars = [$v1, ..., $vk]` into a list of tuples `[($v1, $v2), ..., ($v(k-1), $vk)]` which represent order-constraints among the siblings of a parent term with ordered query term specification.
- `ripe_befs(befs, vars)` operates in the same way as `ripe_injs`, only that its first argument is a list of order constraints.

- The function `vars` from Section 3.5 needs to adjusted to also operate on negated query terms containing variables. In the translation rules for XC_2, `vars` was used to decide whether a query term is to be translated with a `for` or a `some`-clause. Only query terms yielding new variable bindings were translated with `for`. Variables in negated subterms never yield any bindings, and will therefore not be returned by the function. If the argument to `vars`is a list of query terms, `vars` is applied to each of the items in the list, and the union of the results is returned.
- `all_vars(qts)` determines the list of all consuming and defining variable occurrences of the list of terms `qts`.

Rule 4.1 takes care of query terms which are at the same time positive, ordered and breadth-complete. Instead of injectivity constraints, new order constraints `new_befs` are produced and added to `Befs`. It is checked whether any injectivity or order constraints are "ripe" in the sense that both variables are bound. Naturally, only constraints including the variable `$v1` might be "ripe" – since `$v1` is the only new variable in the set of bound XQuery variables `XQVars`, and constraints consisting of two other bound XQuery variables have already been sorted out in the previous applications of translation rules. Ripe constraints are handed over to the $toXQBool$ function. $toXQBool$ takes as its first argument a list of triples to be translated as `some`-clauses. This is only necessary in the case of negated query terms (4.2). Rule 4.1 differs in one more way from Rule 3.1. The number of subterms of the data term to be bound to `$v1` is checked by an additional condition in the `if`-clause.

```
                    vars(subterms(qt1)) \\ XCVs != [],
                          XQVs' = XQVs ++ [$v1],
                fvars = fresh_vars(length(subterms(qt1))),
                      new_befs = make_befs(fvars),
                   ripe_injs = ripe_injs(Injs, XQVs'),
                  Injs' = Injs ++ new_injs \\ ripe_injs
                   ripe_befs = ripe_befs(Befs, XQVs'),
                  Befs' = Befs ++ befores \\ ripe_befs
    triples = zip3 (subterms(qt1), fvars, replicate (length(fvars)) $v1)
```

(Rule 4.1)

$$
\begin{aligned}
&[\![\ [(\texttt{qt1}^{[],+},\$v1,\$p1),\ \ldots\ (qtk,\$vk,\$pk)]\ ]\!]^{\text{As, XCVs, XQVs, Injs, Befs}}_{toXQ} = \\
&\quad \texttt{for } \$v1 \texttt{ in } \$p1/\texttt{child::(label(qt1)) return} \\
&\qquad \texttt{if ((fn:length(\$v1/child::*) == length(subterms(qt1))) and} \\
&\qquad\quad [\![\ [],\ \texttt{ripe\_injs},\ \texttt{ripe\_befs}\ ]\!]^{\text{As}}_{toXQBool}) \texttt{ then} \\
&\qquad\qquad [\![\ \texttt{triples ++ [(qt2,\$v2,\$p2), ..., (qtk,\$vk,\$pk)]}\ ]\!]^{\text{As, XCVs, XQVs', Injs', Befs'}}_{toXQ} \\
&\qquad \texttt{else ()}
\end{aligned}
$$

Translating unordered complete query terms $\texttt{qt1}^{\{\},+}$ and ordered incomplete query terms $\texttt{qt1}^{[[]],+}$ is very similar to the translation Rule 4.1. In the first case, injectivity constraints are produced instead of order constraints, and in the latter case the constraint on the number of children of `$v1` is omitted.

Query terms that bind variables at the top level $\texttt{qt1}^{[],+,X}$ are translated in the same way as in XC_2: The list of associations must be updated, and in the case that some other XQuery variable has already been associated with `X`, both nodes must be checked for deep-equality. The definition of rules for combinations of variables at the top level and different kinds of brackets or braces is straightforward.

Section 3.5 differentiates between terms containing variables not known before the translation of a query term, and subterms without such variables. Needless to say, this distinction cannot be given up in XC_3. As a result, translation rules producing `some`-clauses with the premise that no new variables occur in the query term, must be formulated. Luckily, the rules discussed above can be adapted canonically to this situation: Besides substituting the `for`-clauses by `some`-constructs, the conditions of the enclosed `if`-clause must be moved to the `satisfies`-clause.

Translating negated query terms $\texttt{qt1}^{-}$ is more challenging. Only in one respect it is easier than translating positive subterms: A negated subterm is always translated by means of a `some` clause, since by definition it does not contain any defining variable occurrences. This is why the first premise is missing in Rule 4.2. On the other hand, translating $\texttt{qt1}^{-}$ is more difficult, because all the triples generated by the subterms of `qt1` must be understood as mere constraints to `$v1`, and are therefore included in the `satisfies`-clause. For $toXQBool$ this means that it must be able to handle entire subterms besides injectivity and order constraints.

Moreover, all defining variable occurrences must be translated before consuming ones (as demonstrated in Section 4.1.4). Hence, for each variable `Y` appearing in the negated query term `qt1` it has to be checked

whether there exists a query term in the list of triples that includes a defining occurrence of Y. All such query terms (which are returned by the function `binding_triples(vars(qt1))`) must be translated first together with the siblings of `qt1` that include defining variable bindings (`binding_siblings(qt1)`). Consequently, Rule 4.2 is only applicable if there are no query terms to be translated first. Rule 4.3 covers the reverse case and draws all necessary subterms to the front.

```
                    binding_triples(vars(qt1)) == [],
                       binding_siblings(qt1) == []
                          XQVs' = XQVs ++ [$v1],
                fvars = fresh_vars(length(subterms(qt1))),
                      new_injs = make_injs(fvars),
                   ripe_injs = ripe_injs(Injs, XQVs'),
                 Injs' = Injs ++ new_injs \\ ripe_injs
                   ripe_befs = ripe_befs(Befs, XQVs'),
                 Befs' = Befs ++ new_befs \\ ripe_befs
  triples = zip3 (subterms(qt1), fvars, replicate (length(fvars)) $v1)
```

(Rule 4.2)

$$[\![\ \texttt{[(qt1}^{\{\{\}\},-}\texttt{,\$v1,\$p1), (qt2,\$v2,\$p2), ... (qtk,\$vk,\$pk)]}\ ]\!]_{toXQ}^{\texttt{As, XCVs, XQVs, Injs, Befs}} =$$

```
  if (fn:not(some $v1 in $p1/child::(label(qt1)) satisfies (
    [[ triples, ripe_injs, ripe_befs ]]^As_toXQBool)) then
      [[ [(qt2,$v2,$p2), ..., (qtk,$vk,$pk)] ]]^As, XCVs, XQVs', Injs', Befs'_toXQ
  else ()
```

(Rule 4.3)

```
      binding_triples(vars(qt1)) ++ binding_siblings(qt1) != [],
 predraw_triples = binding_triples(vars(qt1)) ++ binding_siblings(qt1),
other_triples = [(qt1, $v1, $p1), ..., (qtk, $vk, $pk)] \\ predraw_triples
```

$$[\![\ \texttt{[(qt1}^{\{\{\}\},-}\texttt{,\$v1,\$p1), (qt2,\$v2,\$p2), ... (qtk,\$vk,\$pk)]}\ ]\!]_{toXQ}^{\texttt{As, XCVs, XQVs, Injs, Befs}} =$$

```
      [[ predraw_triples ++ other_triples ]]^As, XCVs, XQVs, Injs, Befs_toXQ
```

4.3.2 Translating XC_3 Queries

Xcerpt queries differ from query terms in that they may also be conjunctions, disjunctions, negations of queries or query terms, and are potentially wrapped in resource constructs. It is quite easy to extend the translation of Xcerpt query terms to handle entire queries as well. Rules for these extensions are given in this section. Associating an XQuery variable with an entire Xcerpt query would be premature, in that queries may contain multiple query terms requiring one variable each. Associating a parent variable with an Xcerpt query, however, does make sense and is used amongst other things to translate resource specifications as in 4.4.

For each translation of a resource declaration a fresh variable is bound to the document node of the entity referenced by `uri`. By default, queries are always associated with the resource `$data`, which means that they participate in rule chaining. Resource declarations overshadow this default. The significance of the special variable `$data` is further discussed in Chapter 6.1.1.

$$[\![\ \texttt{(in \{ resource [uri], query \}, \$data), q2, ..., qk}\ ]\!]_{toXQ}^{[],[],[],[],[]} =$$

(Rule 4.4)

```
      let $doc := fn:document(uri) return
        [[ [(query, $doc), q2, ..., qk] ]]^[], [], [], [], []_toXQ
```

For the automatic translation of conjunctions of queries reconsider the example in 4.1.7.2. It shows that the only correct way to translate conjunctions is by nesting the translations of each conjunct. Query conjuncts are in fact very similar to siblings within query terms. The only difference is that the conjuncts are not required to match with distinct nodes. Remember that siblings are translated by adding each of them together with the same parent variable to the list of terms to be processed. Aside from the fact that no fresh variables are associated with the conjuncts of a query, and that no injectivity constraints are created, the proceeding in translation Rule 4.5 is the same.

$[\![$ `[(and(sq1, ..., sqj), $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$ = (Rule 4.5)
$[\![$ `[(sq1, parent), ..., (sqj, parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$

Xcerpt variables always being associated with query terms, but never with entire queries, the list of associations is always empty for the translation of queries. Similar reasoning applies for the list of injectivity and order constraints, and the lists of variables.

The ability to translate disjunctions is of elevated importance in this thesis, because they were used during the compilation of `optional` into `without`. Also for disjunctions of queries, an example expression has been treated in Section 4.1.7.1. Since the evaluation of disjuncts in Xcerpt may be thought of as merging the substitution sets obtained by evaluating each disjunct separately, the best way for us to translate `or`-connected queries is to include the substitutions produced by each disjunct in the same `xc:substitution_set` element. This is formally realized by Rule 4.6.

$[\![$ `[(or(sq1, ..., sqj), $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$ = (Rule 4.6)
$[\![$ `[(sq1, $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$,
`...,`
$[\![$ `[(sqj, $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$

Translating negated queries is even simpler. Assuming that the queries to be translated are in disjunctive normal form, only query terms appear within query negations. A negated query `not(query_term)` is therefore equivalent to its enclosed negated query term `without query_term` (Rule 4.7).

$[\![$ `[(not(query_term), $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$ = (Rule 4.7)
$[\![$ `[(without(query_term), $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$

Finally, the gap between translating queries and query terms must be closed. In the above translation rules, queries are represented as tuples of queries together with their associated resource in form of an XQuery variable. In constrast, query terms are shown as triples including an additional variable for identification purposes. When the translation function comes across a query consisting of a single query term, it simply associates a fresh variable with the query term (Rule 4.8):

`[$fvar] = fresh_vars(1)`

$[\![$ `[(query_term, $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$ = (Rule 4.8)
$[\![$ `[(query_term, $fvar, $parent), q2, ... qk]` $]\!]^{[],[],[],[],[]}_{toXQ}$

4.4 Automatic Translation from XQ_3 to XC_3

As discussed in the last section, equivalent expressions for any XC_3 query can be formulated in XQ_3. At the same time, XQ_3 is designed to not include any expressions that cannot be translated back to XC_3. Moreover, the equivalence of Xcerpt and XQuery expressions is defined as the equivalence over the substitution sets they produce. Therefore, every expression in XQ_3 returns substitution sets that preferably contain bindings for the same set of variables. Both these design guidelines result in XQ_3 being a rather synthetical sublanguage of XQuery, but in this way the expressiveness of both sublanguages coincides, and a sensible notion of equivalence over expressions in XQ_3 and XC_3 is available. Enforcing the additional constraints on the grammar of XQ_3 described in Section 4.2.1, it is in fact possible to translate XQ_3 back to Xcerpt.

Resulting from the design principles, a great part of XQ_3 is part of the range $\{E_{XQ_3} \mid \exists E_{XC_3} \in XC_3 : toXQ(E_{XC_3}) = E_{XQ_3}\}$ of the translation function $toXQ : XC_3 \mapsto XQ_3$. For these expressions it may seem reasonable to define their translations in Xcerpt as their preimage under $toXQ$. Remembering that preimages are generally sets of arbitrary cardinality, this definition of $toXC$ would either need to have the powerset of XC_3 as its codomain, or an arbitrary element of the preimage would have to be picked instead. Both solutions are unsatisfactory in that they do not provide a deterministic way of calculating the results of

$toXC$. Nevertheless, the consideration above shows that the more interesting aspects of $toXC$ is the treatment of expressions not in the range of $toXQ$. The deterministic translation rules presented in the following are designed to fulfill the condition $toXC(E_Q) \in toXQ^{-1}(E_Q)$.

4.4.1 Identifying Xcerpt Disjunctions and Conjunctions

As has been mentioned in Section 4.3, an Xcerpt query is transformed into disjunctive normal form before it is handed over to the translation function $toXQ$. The grammar productions of XQ_3 ensure that it is straightforward to translate every thereby derivable XQuery expression E_Q back to an Xcerpt query in disjunctive normal form. In this section, it is demostrated how to determine the outermost disjunction and enclosed conjunctions of the Xcerpt query $toXC(E_Q)$.

Remember that the element constructors of substitution sets are allowed to contain more than one XQuery expression (see Table 4.2) so that it is possible to translate disjunctions of Xcerpt queries to XQuery. Each of these expressions returned substitutions in XML-representation. Needless to say, multiple expressions within the same `xc:substitution_set` element constructor are translated back to Xcerpt disjuncts. Therefore, each expression within the `xc:substitution_set` element constructor can be translated separately, and the results are appended to form an Xcerpt disjunction (Translation Rule 4.9). If the substitution set contains only one XQuery expression, the outermost `or` may be left away.

$[\![$ `element xc:substitution_set { e1, ..., ek }` $]\!]_{toXC}$ = (Rule 4.9)
`or (` $[\![$ `e1` $]\!]_{toXC}$`, ...,` $[\![$ `ek` $]\!]_{toXC}$ `)`

Identifying conjunctions to be constructed from XQuery expressions is harder, because they do not appear as clearly separated in XQ_3 as the expressions representing the disjuncts. The difficult part is to construct trees from parent-child and ancestor-descendant relationships, and finding out how many trees to construct.

For each XQ_3 subexpression E_{Q_S} within the `xc:substitution_set`, at most one conjunction is formed. E_{Q_S} is derivable from the non-terminal symbol `<EXPR>` of the grammar productions in Table 4.2, and therefore its translation builds upon the translation of entire XQ_2 expressions. This time, however, the translation procedure is split into the two phases *analysis of* E_{Q_S} and *construction of the Xcerpt Conjunction* K. The first phase, which builds up data structures for use in the second phase, is very straightforward, and is given by Rule 4.10. The following data structures are assumed to be constructed in the analysis phase and are used in the rules of the construction phase:

- V is the set of XQuery variables used in E_{Q_S} excluding the special variable `$data`.
- X is the set of Xcerpt variables used as labels in the element constructors within the XML representation of the substitution sets. In other words, for an element constructor `element xc:X1 { $v1 }`, `X1` is added to X.
- $I \subseteq Q \times Q$ is the set of pairs of variables for which there exist injecitivity constraints in E_{Q_S}.
- Analogously, $B \subseteq Q \times Q$ denotes the set of pairs of variables for which order constraints are found in E_{Q_S}.
- $S \subseteq Q \times Q \times QName$ is the set of parent-child and ancestor-descendant relationships among XQuery variables, saved together with their labels. The source of this data structure are `some` and `for`-clauses: A `for`- or `some`-clause binding the variable `$v` to the binding sequence `$w/child::html` is transformed into the triple `($v, $w, html)` and added to S during the analysis phase.
- $A \subseteq V \times X$ is the set of associations between XQuery and Xcerpt variables. For each `element XCVAR { $XQVar }`, the tuple `(XCVar, $XQVar)` is added to A. Furthermore, two variables appearing within a deep-equality constraint `fn:deep-equal($v1, $v2)`, must be associated with the same Xcerpt variable in A. There may be multiple associations for the same Xcerpt variable, but only one for each XQuery variable.
- $C \subseteq V \times \mathbb{N}$ is the set of XQuery variables for which constraints on the number of children are found in E_{Q_S} together with the required number of children. Query terms constructed from variables in C must naturally be breadth-complete.

- $N \subseteq V$ is the set of XQuery variables that are bound in `some`-clauses directly surrounded by a `fn:not`-function. Query terms constructed from these variables will exhibit the `without`-marker.
- $D \subseteq V$ is the of XQuery variables whose step expressions used the `descendant` axis. $V \setminus D$ is the set of XQuery variables that are direct children of their parent variables.

$$[\![E_{Q_S}]\!]_{toXC} = [\![\texttt{\$data}]\!]_{toXC}^{V,X,I,B,S,A,C,N,D} \qquad \text{(Rule 4.10)}$$

Building upon the analysis phase, which is summarized by Rule 4.10conjunctions must be extracted from XQ_3-subexpressions E_{Q_S} inside of the `substitution_set` element constructor. For this purpose, it is useful to remember how these conjuncts are produced during the reverse translation procedure. For each conjunct, one fresh XQuery variable is introduced by $toXQ$, and the parent of these fresh XQuery variables is always the special variable `$data`. Hence, to identify the conjuncts to be produced by $toXC(E_{Q_S})$, it is necessary to search for the variables $c_1, \ldots c_k \in V$ that are descendants (including children) of `$data`. If there exists only one child c_1 of `$data`, it is not necessary to produce a conjunction with one single element, but the translation is better given by a simple query term. Note that it is necessary that each variable $v \in V$ is either within $\{c_1, \ldots, c_k\}$ or descendant of exactly one of the c_i. As soon as a single XQuery variable $v \in V$ occurs as a descendant of c_i and c_j ($1 \leq i < j \leq k$), the XQuery expression is not translatable. The situation that v does not have an ancestor within $\{c_1, \ldots, c_k\}$ is forbidden, because then v would be free within E_{Q_S}, and only `$data` is allowed to be free within XQ_3.

Besides the procedure described above, there exists another approach to answer the question whether $toXC(E_{Q_S})$ is a simple, possibly negated query term or a conjunction of possibly negated query terms: First the equivalence relation $=_S \subseteq V \times V$ is introduced. It is defined as the smallest reflexiv symmetric and transitive set such that for all $v_1, v_2 \in V$, (v_1, v_2) is within $=_S$, if there exists a `for`- or `some`-clause within S in which v_2 is bound to child nodes or descendants of v_1. If the set of equivalence classes of $V/_{=_S}$ under this equivalence relation contains only a single element, it is not necessary to produce a conjunction of query terms, otherwise one (possibly negated) query term is computed for each equivalence class to form a conjunction. An interesting characteristic of the equivalence classes V_i is that in each of them, there exists one "top" variable $top(V_i)$. This is easy to see remembering that all $v_i \in V_i$ must be transitively connected, and that for each v_i there exists at most one parent. These top variables of the equivalence classes $V/_{=_S}$ are the c_i identified above. Using the notation $[\![c_l]\!]$ for the equivalence class of c_l, this may also be written $V/_{=_S} = \{[\![c_1]\!], \ldots, [\![c_k]\!]\}$.

The construction of the conjuncts starts out with these top variables (See Rule 4.11). For the rule to be applicable, it is necessary to ensure that there are no injecitivity constraints among the top variables $top(V_1), \ldots, top(V_k)$, because Xcerpt allows conjuncts within `and` to simulate with the same data terms. It is computationally reasonable to split the data structures collected in the analysis phase into smaller disjunct subsets and pass these over to the recursive calls, as exemplified with the set of XQuery variables V and the injectivity constraints I. This would also make sense for the sets B, A, C, N and D, but was omitted in Rule 4.11 for the sake of brevity.

$$\begin{gathered} V/_{=_S} = \{V_1, \ldots, V_k\} \\ \forall\, 1 \leq i \leq k:\ S_i = \{(\texttt{\$v, \$p, label}) \mid \texttt{\$v} \in V_i, \texttt{\$p} \in V_i\} \\ \forall\, 1 \leq i \leq k:\ I_i = \{(\texttt{\$i1, \$i2}) \mid \texttt{\$i1}, \texttt{\$i2} \in V_i\} \\ \nexists V_i, V_j \in V/_{=_S} : (top(V_i), top(V_j)) \in I \end{gathered} \qquad \text{(Rule 4.11)}$$

$$[\![\texttt{\$data}]\!]_{toXC}^{V,X,I,B,S,A,C,N,D} = \texttt{and } \{[\![top(V_1)]\!]_{toXC}^{V_1,X,I_1,B,S_1,A,C,N,D}, \ \ldots, \ [\![top(V_k)]\!]_{toXC}^{V_k,X,I_k,B,S_k,A,C,N,D}\}$$

4.4.2 Automatic Construction of Query Terms From XQ_3

Within Xcerpt queries in disjunctive normal form, query negations appear as the innermost construct of the three query connectives `or`, `and` and `not`. At this position, `not` is semantically equivalent to the negation of query terms using `without`. As mentioned above, the translation rules $toXC$ produce only Xcerpt queries in DNF, and therefore they always use query term negation instead of `not`.

The construction of a query term qt from an XQuery expression always starts out with one single variable `$parent` that represents the root of qt. With XC_3 including all kinds of brackets, descendants, query term

negations and Xcerpt variables, 32 alternative rules would have to be given. Since they are very similar to each other, this sectiononly presents two of them and describes how to derive the other 30. The premises of the rules differ in that they either require order constraints or injectivity constraints and the presence or absence of a constraint on the number of children of `$parent`. Furthermore the premises either require the existence or lack of an association with an Xcerpt variable, and demand that `$parent` is included in or excluded from the set of negated variables N and from the set of descendants D. Altogether this sums up to $2^5 = 32$ different rules.

Rule 4.12 transforms the single XQuery variable `$parent` into a negated, unordered and breadth-incomplete query term taking into account the data structures generated during the analysis phase of the translation. For the rule to be applicable, `$parent` must not be associated with an Xcerpt variable, must be within N, but neither within D nor C, and for each pair of label-overlapping children (for the definition of *overlapping tag names* see Section 3.2.2), there must exist an injectivity constraint within I. The children of `$parent` are determined by searching for triples in S which have `$parent` as their parent variable. If all requirements are fulfilled, the subterms are constructed by recursive calls to $toXC$.

$$\texttt{\$parent} \in N,\ \texttt{\$parent} \notin D,\ \nexists n \in \mathbb{N} : (\texttt{\$parent}, n) \in C,\ \nexists v \in X : (\texttt{\$parent}, v) \in A,$$
$$(\texttt{\$parent\_parent},\ \texttt{\$parent},\ \texttt{parent\_label}) \in S,$$
$$C = \{\texttt{\$c1}, \ldots, \texttt{\$ck}\} = \{\texttt{\$child} \mid (\texttt{\$child},\ \texttt{\$par},\ \texttt{label}) \in S, \texttt{\$par}{=}\texttt{\$parent}\},$$
$$\forall c_1, c_2 \in C : (c_1 \neq c_2) \wedge (label(c_1) \sim label(c_2)) \Rightarrow (c_1, c_2) \in I$$

$$[\![\ \texttt{\$parent}\ ]\!]_{toXC}^{V,X,I,B,S,A,C,N,D} = \qquad \text{(Rule 4.12)}$$
$$\texttt{without parent\_label \{\{}\ [\![\texttt{\$c1}]\!]_{toXC}^{V,X,I,B,S,A,C,N,D},\ \ldots,\ [\![\texttt{\$ck}]\!]_{toXC}^{V,X,I,B,S,A,C,N,D}\ \texttt{\}\}}$$

Rule 4.12 can be easily adapted to produce several similar query terms: If the step expression of `$parent` uses the `descendant` axis instead of the `child` axis, then `$parent` would be included in D and translated by a rule with the premise `$parent` $\in D$ and the additional `desc` construct in the translation. If $(\texttt{\$parent}, n) \in C$ were true for some $n \in \mathbb{N}$, the translation would use single curly braces instead of double ones, and if `$parent` $\notin N$ were true, the translation would omit the `without` construct. Finally, if an associated Xcerpt variable $v \in X$ could be found within the set of associations A, the translation would read `without var` v `as parent_label {{ ... }}`. All possible combinations of these adjustments yield 15 more rules and generate all kinds of unordered query terms.

Generating ordered query terms is a bit different and covered by rules like 4.13. A premise for generating an ordered query term is of course the existence of appropriate order constraints. Therefore it is demanded that there exists an ordering of the elements of C, such that for each subsequent pair of variables in this sequence, an order constraint can be found within B. The query term produced by Rule 4.13 is not only ordered, but also breadth complete, because `$parent` is within the set C. As a consequence, it must be ensured that the number of children k coincides with the number of required children n. This rule can be adjusted in a very similar way as above to cover the generation of all kinds of ordered query terms.

$$\texttt{\$parent} \notin N,\ \texttt{\$parent} \in D,\ (\texttt{\$parent}, n) \in C,\ \exists v \in X : (\texttt{\$parent}, v) \in A,$$
$$(\texttt{\$parent\_parent},\ \texttt{\$parent},\ \texttt{parent\_label}) \in S,$$
$$C = \{\texttt{\$c1}, \ldots, \texttt{\$ck}\} = \{\texttt{\$child} \mid (\texttt{\$child},\ \texttt{\$par},\ \texttt{label}) \in S, \texttt{\$par}{=}\texttt{\$parent}\},\ n = k$$
$$\exists f : C \to \{1 \ldots k\}, f \textit{ is bijective} : \forall c_i, c_j \in C : f(c_i) + 1 = f(c_j) \Rightarrow (c_i, c_j) \in B$$

$$[\![\ \texttt{\$parent}\ ]\!]_{toXC}^{I,B,S,A} = \qquad \text{(Rule 4.13)}$$
$$\texttt{desc var}\ v\ \texttt{as parent\_label [}\ [\![f^{-1}(1)]\!]_{toXC}^{I,B,S,A},\ \ldots,\ [\![f^{-1}(k)]\!]_{toXC}^{I,B,S,A}\ \texttt{]}$$

Given the above rules and their derivatives, it is obvious that for each call to the tranlsation function, a corresponding rule can be found. Termination of the translation algorithm is also evident, because the expressions in XQ_3 are not allowed to include cyclic parent child relationships between variables (this is a direct consequence of `$data` being the only free variable in a XQ_3 expression).

The translation rules for $toXC$ are devised with the original query term in mind. Except for subterm ordering within unordered query terms and the order of conjuncts and disjuncts, the equation $toXC(toXQ(q)) = q$ holds for an abritrary query q within XC_3. The same example as in Section 3.6 can be shown that Equation 3.4 does not hold for the third pair of sublanguages.

Chapter 5

Translation of Construct Parts

This chapter examines an entirely different challenge of the automatic translation between Xcerpt and XQuery. In contrast to the last chapter, the central topic is not query terms and the extraction of substitution sets, but the construction of results.

This chapter is structured as follows: In Section 5.1 a new Xcerpt grouping construct is introduced that is necessary to translate certain XQuery expressions. In Section 5.2 Xcerpt grouping constructs are translated in close correspondency with their formal semantics. The section builds upon the translation of query terms to XQuery expressions returning substitution sets and shows how both methods can be combined to tranlsate entire construct-query-rules. The remaining two sections cover the reverse direction: Section 5.3 shows how to disentangle intertwined XQuery expressions to translate them to Xcerpt rules. Section 5.4 goes one step further by also considering the construction of intermediate results within XQuery expressions.

5.1 Proposal of a Duplicate-preserving Grouping Construct for Xcerpt

As a preliminary to the translation of construct terms, it is necessary to point out that not all expressions can be easily translated from XQuery to Xcerpt. This is due to the fact that Xcerpt is designed to perform duplicate elimination in two ways: First, Xcerpt query terms return substitution sets, although in some cases it may be preferable to work with multi-sets of substitutions:

"Note that in practice, it would be desirable to define substitution sets as multi-sets that may contain duplicate elements: if an XML document contains two persons named 'Donald Duck', then it should be assumed that these are different persons with the same name." [17, Section 7.3.1]

Second, the grouping construct `all` performs duplicate elimination. For the objective of this thesis, these inherent characteristics of Xcerpt pose a challenge, which is exemplified by $Expr_{5.1_{XQ}}$ and its Xcerpt counterpart $Expr_{5.1_{XC}}$: When applied to $Data_{5.1}$, the XQuery expression returns a `result` element containing two persons named 'Donald Duck', while the translation would return only one such subterm.

As a workaround to this problem, in this chapter it is assumed that Xcerpt query terms in fact do return multi-sets of substitutions. Note that also the translations of Xcerpt queries return XML-representations of multi-sets of substitutions. Furthermore, the usual duplicate eliminating grouping construct `all` is renamed to read `all-distinct`, and a semantically new grouping construct named `all` is introduced, the application of which retains duplicates.

$Expr_{5.1_{XQ}} :=$
```
element result { for $p in $data/child::person return $p }
```

$Expr_{5.1_{XC}} :=$
```
CONSTRUCT result [all var X] FROM var X as person {{ }} END
```

$Data_{5.1} :=$
```
data [ person [ 'Donald Duck' ],
       person [ 'Daisy Duck' ],
       person [ 'Donald Duck' ] ]
```

The duplicate elimination by the `all-distinct` keyword is formally defined in [17, Section 7.3.3] by the following equation:

(5.1) $[\![\sigma]\!](\texttt{all-distinct t}) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!], \ldots, [\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{FV(t)}}$

Since this grouping construct performs value based duplicate elimination, it is renamed to `all-distinct`. The equivalence classes are calculated according to the simulation unification relation among query terms. Defining the equivalence classes based on node identity $=^{id}$, it would be possible to specify an `all` construct that preserves duplicates as in Equation 5.2.

(5.2) $[\![\sigma]\!](all\ t) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t)$ where $\{[\![\tau_1]\!], \ldots, [\![\tau_k]\!]\} = [\![\sigma]\!]/_{=^{id}_{FV(t)}}$

The equivalence classes in 5.2 are not calculated according to the values of the bindings of the free variables, but with respect to their node identitities. Obviously, this would demand from an Xcerpt implementation to assign identitifiers to each node acquired from an input resource, and remember them during the evaluation of rules. Furthermore, unique node identifiers would also have to be assigned to the elements constructed by Xcerpt rules, so that they can participate in rule chaining.

With this definition for the grouping construct `all`, $Expr_{5.1_{XQ}}$ would be a correct translation of $Expr_{5.1_{XC}}$. In the rest of this chapter, it is assumed that this new `all` construct may be used along with the ordinary grouping constructs `all-distinct` in Xcerpt construct terms.

5.2 From Xcerpt Construct Terms to XQuery

Not giving up the separation of querying and constructing data when translating from Xcerpt to XQuery eases the translation process notably, and allows to see that the translation is indeed correct. Therefore, the translations of query terms to XQuery return an XML representation of the Xcerpt notion of substitution sets, and the translations of construct terms query these substitution sets in order to construct the results. In this way, entire construct query rules can be translated to XQuery (Rule 5.1). Both kinds of rules, the ones treating Xcerpt query terms, and the ones converting Xcerpt construct terms to XQuery, carry the same name. In some cases this may result in syntactically equivalent construct and query terms being treated in diverse ways, but from the context it is always evident which translation function is to be applied.

$[\![$ `CONSTRUCT ct FROM qt END` $]\!]_{toXQ}$ = (Rule 5.1)

$[\![$ `let $ss := (` $]\!]_{toXQ}$ `)/* return (if $ss then (` $[\![$ `ct` $]\!]^{\$ss}_{toXQ}$ `) else () )`

So as to cover the case of empty substitution sets, the above translation rule checks if `$ss` actually contains any elements. Only if this check succeeds, the translation proceeds, otherwise the empty sequence is returned.

In the following, it is assumed that the query part of an Xcerpt construct query rule has already been translated and evaluated and that the sequence of substitutions in XML-representation is bound to the XQuery variable named `$ss`. An example assignment for `$ss` would be:

```
$ss := (
   element xc:substitution {
     element xc:X { <a><b/></a> }, element xc:Y { <c/> } },
   element xc:substitution {
     element xc:X { <a><c/></a> }, element xc:Y { <b/> } },
   element xc:substitution {
     element xc:X { <a><c/></a> }, element xc:Y { <d/> } } )
```

For the sake of clarity, direct element constructors are used for the data and computed element constructors for the meta-data in the substitution set above. Although there are no duplicate substitutions, they would be allowed.

Construct terms including only breadth-complete terms, and XML Data always being ordered, it suffices to focus on the case of single square brackets. In the next sections example Xcerpt construct terms are translated to XQuery. The formal justification for these translations are taken from [17, Section 7.3.3].

Just as with query terms, only a subset of the construct terms of Xcerpt is translated. Not included in this subset are, e.g., positional variables, `some`-constructs, `order by`-constructs and aggregations. Nested grouping constructs are taken care of, as can be seen by the grammar productions for the subset of construct terms considered (Table 5.1).

Table 5.1: Grammar productions for a subset of Xcerpt construct terms

```
<CTERM>    ::= var X | <QNAME> [ <CTERM>* ] | all <CTERM> <GROUP-BY>? |
               all-distinct <CTERM> <GROUP-BY>?
<GROUP-BY> ::= group by { <XCVAR>+ }
```

5.2.1 Minimal Construct Terms: Single Terms and Variables

In this section, translations for single variables and construct terms without children are given. The formal semantics of Xcerpt with respect to the application of substitution sets to construct terms are studied to argue that the translation is indeed correct.

The construct term to be first discussed term is a simple label without children such as `tag0[]`. Two cases are distinguished when applying a substitution set Σ to a construct term ct in Xcerpt. The first case is the one in which all substitutions $\sigma_i \in \Sigma$ have the same bindings with respect to the free variables $FV(ct)$ of the construct term. In other words, the substitution set is equal to the single equivalence class $[\![\sigma]\!]$ with respect to the equivalence relation $\simeq_{FV(ct)}$ for an arbitrary $\sigma \in \Sigma$ (For the formal definition of $\simeq_{FV(ct)}$ see [17, Definition 7.3]). The second case is that there is more than one element in $\Sigma/_{\simeq_{FV(ct)}}$. Different formulas are applied for each of these cases.

Since the simple construct term `tag0[]` does not include any variable, the rule for the first case must be applied in its translation:

$$[\![\sigma]\!](f[t_1, \ldots, t_n]) = \langle [\![\sigma]\!](f)[\ [\![\sigma]\!](t_1) \circ \cdots \circ [\![\sigma]\!](t_n)\ ] \rangle \tag{5.3}$$

In formula 5.3 σ is a representative for all substitutions to be applied. Having neglected label variables in the treatment of Xcerpt query terms, the label f is mapped to itself. Therefore, the result of the application of an arbitrary non-empty substitution set to `tag0[]` is simply the construct term itself. In the case that the substitution set `$ss` is empty, its application to the construct term yields nothing. Hence, the correct translation is `if ($ss) then element tag0 {} else ()`.

The translation of a construct term consisting of a single variable, say `var X`, is just as simple. In contrast to the first example, the set of free variables of `var X` is not the empty set, but the set containing `X` as its single member. Generally `X` does not have the same assignment in all substitutions stemming from the evaluation of the query part. For this case the following equation is found in [17]:

$$\Sigma(t^c) = \{t^{c'} \mid [\![\sigma]\!] \in \Sigma/_{\simeq FV(t^c)} \wedge \langle t^{c'} \rangle = [\![\sigma]\!](t^c)\} \tag{5.4}$$

As mentioned before, $\Sigma/_{\simeq FV(t^c)}$ is the set of equivalence classes of the substitution set Σ with respect to the simulation equivalence over the bindings of the free variables of the construct term t^c. Only considering substitutions containing data terms, $\Sigma/_{\simeq FV(t^c)}$ is also the set of equivalence classes of Σ with respect to the *deep-equality* function over the bindings of the free variables of t^c. From this formula it can be derived that the number of results equals the number of different assignments for `X` in the substitution set. To be more precise, in the case of t^c being a single variable, the results are the set of different bindings for this variable. Thus the correct translation of `var X` to XQuery is $Expr_{5.2.1_{XQ}}$.

$Expr_{5.2.1_{XQ}} :=$ `distinct-elements($ss/child::xc:X)`

The `distinct-elements()`-function (see Table 5.2) eliminates duplicates in the bindings for `X` and must be employed to reflect the fact that for each equivalence class, which may contain any positive number of substitutions, only one result term is produced. Its definition is derived from the definition of the `distinct--nodes-stable` function from [14, E.5 eg:distinct-nodes-stable]. The `fn:distinct-values`-function eliminates duplicates based on the equality of string values of nodes rather than checking if they are deep-equal, and thus cannot be used for translating variable occurrences in construct terms.

The equivalence classes are calculated according to the bindings of `X`, which is the only free variable in the construct term. Note that although the formula above specifies that the result of the application of a substitution set to a construct term is a set, without an `all`-construct, the Xcerpt interpreter would only return one single result term. In the translation this may be reflected by taking only the first element of the list returned by the

Table 5.2: The `distinct-elements` function

```
declare function distinct-elements($arg as node()*) as node()*
{
   for $a at $apos in $arg
   let $before_a := fn:subsequence($arg, 1, $apos - 1)
   where every $ba in $before_a satisfies not(fn:deep-equal($a,$ba))
   return $a
};
```

`distinct-elements` function. In the rest of this section, variables always appear within the scope of an `all`-construct to avoid this issue.

5.2.2 Grouping with Respect to a Single Variable

In the case of more than one variable occurring within a construct term, the translation method depends upon whether the variables appear within the scope of the same grouping constructs or not. The easiest case is the one with all variables appearing within their own private grouping construct as in $Expr_{5.2.2_{XC}}$.

$Expr_{5.2.2_{XC}} :=$ `tag0 [ all-distinct var X, all-distinct var Y, all var Z]`

Applying a substitution set to the outermost term in $Expr_{5.2.2_{XC}}$ is handled by Equation 5.3, because the set of free variables of $Expr_{5.2.2_{XC}}$ is empty. As mentioned above, only expressions whose outermost terms do not contain any free variables are considered, and thus this reasoning applies to all example expressions treated in this section. The application of Equation 5.3 to $Expr_{5.2.2_{XC}}$ yields exactly one result element named `tag0` – if the substitution set is not empty. To reflect this, the XQuery element constructor for `tag0` in $Expr_{5.2.2_{XQ}}$ must not appear within a `for` clause and all further output must be generated within this element. Upon translating the outermost construct term, the translation of its subterms follows: The application of the substitution set to terms of the form `all-distinct var X` is handled by equations 5.1 and 5.5 taken from [17, Section 7.3.3]. Likewise, the application of a substitution set to terms of the from `all var X` is handled by equations 5.2 and 5.5. Remember that although set notation is used in all of these equations, $[\![\sigma]\!]$ may be a multi-set of substitutions.

$$[\![\sigma]\!](Var\ V) = \langle \sigma(V) \rangle \tag{5.5}$$

Apparently, the set of free variables $FV(t)$ for a simple variable occurrence is the set consisting of the variable itself. Consequently, duplicates must be eliminated based on the values or identities of these single variables. The `distinct-elements`-function must be used to eliminate duplicates based on value, and the `distinct-nodes-stable`-function to eliminate duplicates based on node identity. The definition of the `distinct-nodes-stable` function differs from the `distinct-elements`-function (Table 5.2) only in that the nodes are not compared with respect to deep equality, but with respect to node identity.

$Expr_{5.2.2_{XQ}} :=$

```
if ($ss) then element tag0 {
  (distinct-elements($ss/child::xc:X),
   distinct-elements($ss/child::xc:Y),
   distinct-nodes-stable($ss/child::xc:Z)) }
else ()
```

A final important observation is that without the `if`-clause, $Expr_{5.2.2_{XQ}}$ would always return an element, while the original Xcerpt construct term does not produce anything in the case of an empty substitution set. To fix this, an XQuery translation of a construct query rule must check whether the substitution set returned by the translation of the query part contains any substitutions, and only in this case, the construct part must be evaluated on the returned substitution set (as in Rule 5.1). This additional check needs only be carried out once, and is unnecessary in the recursive calls to the translation functions, because empty substitution sets

are not generated by recursive calls. Therefore, the `if`-clause could be left away, if $Expr_{5.2.2_{XC}}$ were to be translated as a subterm of an enclosing construct term.

To give a first taste of automatic translation of construct terms, a translation rule for simple construct terms without grouping constructs is introduced. Translation rules for `all` and `all-distinct` are presented in the following section, after the discussion of two other example expressions. The translation function takes one argument $\$ss$, which is an XQuery variable representing the substitution set to be applied to the construct term. The initial assignment for $\$ss$ is generated by the translation of the query part of a rule. The rules discussed in this section either pass over the same substitution set to a recursive call of the translation function, or they compute a set of equivalence classes and hand over those as the substitution set. If the construct term is a variable, or a term without subterms, the translation terminates.

$$[\![\texttt{label [ct1, ..., ctn]}]\!]^{\$ss}_{toXQ} = \qquad \text{(Rule 5.2)}$$
$$\texttt{element label \{} [\![\texttt{ct1}]\!]^{\$ss}_{toXQ}\texttt{, ...,} [\![\texttt{ctn}]\!]^{\$ss}_{toXQ} \texttt{\} else ()}$$

A more complex example than $Expr_{5.2.2_{XC}}$ is one in which variables are grouped according to the values of other variables. Xcerpt distinguishes between explicit grouping with the `group by`-clause and implicit grouping which takes place (not only) if a construct term includes both free and bound variables, such as `tag1 [all-distinct var X, var Y]` in $Expr_{5.2.2^b_{XC}}$. `Y` does not occur within an all construct in this subterm, while `X` does. Hence `X` is grouped according to the values of `Y`.

$Expr_{5.2.2^b_{XC}} :=$ `tag0 [ all-distinct tag1 [all-distinct var X, var Y] ]`

The translation, $Expr_{5.2.2^b_{XQ}}$, calculates the set of distinct values for `Y`, and for each of these values, the corresponding equivalence class $[\![\sigma]\!] \in \Sigma/_{\simeq_{\{Y\}}}$ is assigned to `$class_y`. Since `X` shall be grouped by `Y`, all of its distinct bindings within the same equivalence class are emitted within the same `tag1`-element.

$Expr_{5.2.2^b_{XQ}} :=$

```
element tag0 {
  for $y in distinct-elements($ss/child::xc:Y) return
    let $class_y := (
      for $s in $ss return
        if fn:deep-equal($s/child::xc:Y, $y)
        then $s else () )
    return
      element tag1 { distinct-elements($class_y/child::xc:X), $y }
```

As a slight variation, the correct translation of $Expr_{5.2.2^c_{XC}}$, which differs from $Expr_{5.2.2^b_{XC}}$ only in that the innermost `all-distinct` is substituted by an `all`, can be obtained by substituting the second `distinct-elements`-function in $Expr_{5.2.2^b_{XQ}}$ by the `distinct-nodes-stable`-function.

$Expr_{5.2.2^c_{XC}} :=$ `tag0 [ all-distinct tag1 [all var X, var Y] ]`

5.2.3 Explicit and Implicit Grouping with Respect to more than one Variable

Grouping with respect to multiple variables means that a substitution set shall be applied to a construct term $all\ t$ with $|FV(t)| > 1$. In XQuery, the computation of equivalence classes of substitution sets according to not only one single variable, but a set of variables, is more difficult.

The subterm `tag1 [all var X, var Y, var Z]` of $Expr_{5.2.3_{XC}}$ exemplifies this situation.

$Expr_{5.2.3_{XC}} :=$ `tag0 [ all tag1 [all var X, var Y, var Z] ]`

In its translation ($Expr_{5.2.3_{XQ}}$), the substitution set given by the variable `$ss` is tested for emptiness as usual. Subsequently, the set of representatives `yzs` of all equivalence classes is computed making use of the `distinct-elements()` function. Note that it is necessary to create a new element to enclose the values of `$s/child::xc:Y` and `$s/child::xc:Z`. Simply returning both values as a tuple would result in the argument of the `distinct-elements()`-function being a nested list of elements, which in XQuery is

immediately flattened to an ordinary list. Upon calculating the representatives of the equivalence classes, it is easy to identify the equivalence classes themselves by iterating over the set of all substitutions for each representative and comparing the binding for the variables `Y` and `Z`.

Let $vars$ be the number of variables to be grouped by, n the number of substitutions, $dist(k)$ the complexity of the distinct elements function called upon a sequence of k values and deq the complexity of the `deep-equal`-function called upon two elements. Then the time complexity of this algorithm amounts to $O(n \cdot (dist(n) + n \cdot deq \cdot vars))$, which can be seen as follows: In lines 10 and 11, the `fn:deep-equal` function is used once for each variable. These comparisons are enclosed in a `for`-clause, which is executed n times. The `distinct-elements`-function is applied on the results, and the whole procedure is repeated once for each representative (which equals the number n of substitution sets in the worst case). The computation of the representatives in lines three to five does not have an effect on the overall complexity.

Already the `fn:deep-equal` function is computationally expensive and depends on the size of the input data. The `distinct-values` function compares each possible combination of two nodes of the input sequence ($\frac{n \cdot (n-1)}{2}$ combinations) and therefore its complexity amounts to $dist(n) = O(n^2 \cdot deq)$. As a result, the overall complexity of calculating a substitution set according to the above method is $O(n^3 \cdot deq \cdot vars)$. In Section 5.2.4, the complexity is reduced to $O(n^2 \cdot deq \cdot vars)$.

$Expr_{5.2.3_{XQ}} :=$

```
1   if ($ss) then
2     element tag0 {
3       let $yzs := distinct-elements(
4         for $s in $ss return
5           element xc:value { $s/child::xc:Y, $s/child::xc:Z })
6       return
7         for $yz in $yzs return
8           let $subs_yz := distinct-elements(
9             for $s in $ss return
10              if (fn:deep-equal($s/child::Y, $yz/child::Y) and
11                  fn:deep-equal($s/child::Z, $yz/child::Z))
12            then $s else ())
13          return
14            element tag1 {
15              $subs_yz/child::xc:X,
16              $yz/child::xc:Y,
17              $yz/child::xc:Y } }
18  else ()
```

Explicit grouping being no harder to translate than implicit grouping, no special example expression is treated here. Instead, imagine that variables `Y` and `Z` in $Expr_{5.2.3^{c}_{XC}}$ appeared within a `group by`-clause following `all tag1 [var X]` instead of within the subterm. The rule handling the application of substitution sets to explicit grouping constructs reads:

$$[\![\sigma]\!](all\ t\ group\ by\ V) = [\![\tau_1]\!](t) \circ \cdots \circ [\![\tau_k]\!](t) \text{ where } \{[\![\tau_1]\!], \ldots, [\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{FV(t) \cup V}} \tag{5.6}$$

For the altered example expression this means that the equivalence classes are again calculated according to the same variables `Y` and `Z` - just as before. The semantics of the resulting expression would only differ in that the bindings for `Y` and `Z` would not show up in the result.

Automatic translation of the grouping construct `all-distinct` is achieved by Rule 5.3, in which `ct` denotes a construct term, `X1`, ..., `Xn` are its free variables, and `$ss` is the substitution set to be applied to `ct`. The translation function calls itself recursively, thereby replacing the substitution set `$ss` given as a parameter by the equivalence class `$class_r` which is calculated with respect to the free variables of `ct`.

There are two related rules that can be derived from Rule 5.3. The first one is concerned with the translation of the duplicate-preserving `all`, and the second one treats an additional `group-by`-clause. Since these two extensions are orthogonal to each other, a third derivative treating both extensions is obtained by applying both adjustments to Rule 5.3.

$[\![$ `all-distinct ct` $]\!]^{\$ss}_{toXQ}$ = (Rule 5.3)

```
let $reps := distinct-elements (
  for $s in $ss return
    element xc:value { $s/child::xc:X1, ..., $s/child::xc:Xn } )
  return for $r in $reps return
    let $class_r := distinct-elements(
      for $s in $ss return
        if (fn:deep-equal($s/child::xc:X1, $r/child::xc:X1)
          ... and ...
          fn:deep-equal($s/child::xc:Xn, $r/child::xc:Xn))
        then $s else ()
      return
```

$[\![$ `ct` $]\!]^{\$class\_r}_{toXQ}$

If duplicates are to be preserved by the grouping construct, the representatives of the substitution sets must be calculated according to the node identities of the variable bindings for `X1, ..., Xn`. Unfortunately, the `distinct-nodes-stable`-function does not achieve this aim, because all of the newly constructed `xc:value`-elements are assigned unique and different node identifiers by XQuery. Neither can the comparison operate on the children of these elements instead, because the identities of the nodes within the originally queried data get lost as soon as `xc:substitution`-elements are generated to surround them. To give an example, the expression `op:is-same-node(<a>$v1</a>/*, <a>$v1</a>/*)` returns false in XQuery, while comparing the variable references directly would return true. A way out to this problem would be to extend the representation of substitution sets to include additional artificially created node identifiers for all variable values. Since the duplicate preserving `all` has not yet found its way into the Xcerpt language, no alternative translation methods are explored.

The attempt to adapt Rule 5.3 to explicit grouping constructs is more successful. It is sufficient to unify the set of free variables and the set of variables in the `group by` clause, calculate the representatives of the equivalence classes based upon this union of variables, and call the `deep-equal` on all of them.

There is one other important part of Xcerpt construct terms that can neither be translated by Rule 5.2 nor by 5.3: Variable occurrences. In the example expressions such as $Expr_{5.2.3_{XQ}}$ they are translated by simply returning the value (e.g. `$ss/child::xc:X`) of their bindings in the XML-representation of the substitution set.

$[\![$ `var X` $]\!]^{\$ss}_{toXQ} =$ `$ss/child::xc:X` (Rule 5.4)

Theoretically, such a step expression may return a sequence of values of arbitrary length, but since there do not occur any free variables in the construct terms examined in this section, the equivalence classes applied to a variable occurrence always contain exactly one binding for each variable. There is, however, also the opportunity to return all values of a variable next to each other within the same parent node, such as in `tag1 [ all var X]`. This case is covered collaboratively by rules 5.3 and 5.4.

5.2.4 Easing the Translation of Construct Terms with XQuery Functions

The translations presented in the last section are rather verbose, and applying Rule 5.3 and its derivatives more than once when translating complex construct terms, leads to great redundancy of code. As in other programming languages, XQuery functions are used to factor out common code, and the required functions are presented in this section.

The ultimate goal of using functions to translate grouping constructs is the following: Given a list of substitutions in XML-representation accessible by the variable `$ss` and a list of variable names `$vars` (e.g `("X", "Y")`), the equivalence classes with respect to the bindings of the variables in `$vars` shall be calculated by a single function call `xc:equivalence-classes($ss, $vars)`. This allows a much briefer translation of construct terms.

The most fundamental function needed for this purpose is one that checks whether two substitutions `$s1` and `$s2` belong to the same equivalence class with respect to the bindings of the variables in `$var`. The recursive function in Table 5.3 achieves this end.

Table 5.3: XQuery function xc:has-same-bindings

```
declare function xc:has-same-bindings($s1, $s2, $vars) {
  if ($vars) then        (: recursive case: still variables to compare :)
    if (fn:deep-equal($s1/*[fn:local-name(.) = $vars[1]],
                      $s2/*[fn:local-name(.) = $vars[1]]))
    then xc:has-same-bindings($s1, $s2, fn:subsequence($vars,2))
    else false()
  else true()            (: base case: no more variables to compare:)
};
```

Based on this function, computing a list of representatives of the set of equivalence classes could be realized in a very similar way as in the last section. But time complexity can be improved by calculating the classes in one single sweep over the list of substitutions without taking the indirection over representatives. With XQuery functions at hand this enhancement is implemented as follows:

Table 5.4: XQuery functions xc:eq-classes and xc:insert-subst

```
declare function xc:eq-classes($ss, $classes, $vars) as node()* {
  if ($ss) then   (: recursive case :)
    let $newclasses := xc:insert-subst($ss[1], $classes, $vars) return
      xc:eq-classes(fn:subsequence($ss, 2), $newclasses, $vars)
  else $classes  (: base case: all substitutions processed :)
};

declare function xc:insert-subst($s, $classes, $vars) as node()* {
  if ($classes) then    (: try to find an equivalence class for $s :)
    if (xc:has-same-bindings($classes[1]/*[1], $s, $vars)) then
      (element class { $classes[1]/*, $s }, fn:subsequence($classes, 2))
    else ($classes[1],
      xc:insert-subst($s, fn:subsequence($classes, 2), $vars))
  else ($classes, element class { $s })      (: a new class is added :)
};
```

In Table 5.4, the function `xc:eq-classes` iterates over the list of substitutions, inserting one substitution at a time in the list of equivalence classes. In the initial call to the function, the list of equivalence classes is always empty, and therefore a wrapper function taking only `$ss` and `$vars` as its only arguments would be helpful.

The function `xc:insert-subst` is used to iterate over the equivalence classes which built up so far. For each equivalence class, it is checked whether the current substitution `$s` should be part of it. If all these checks fail, a new equivalence class is added with `$s` as its only element.

The realization of `xc:insert-subst` is not as brief as one might expect. This is due to the absence of a built-in function to append a child to a given node, such as the `appendChild`-function in the document object model [8]. Certainly, such a function could be implemented as a user-module, and as a matter of fact, such updates of node values will be natively supported in future versions of XQuery, which would make the above code more concise [7].

Making use of these functions, the translation of the construct term $Expr_{5.2.3_{XQ}}$ becomes less verbose:

```
if ($ss) then {
  element tag0 {
    for $class in xc:eq-classes($ss, (), ("Y", "Z")) return
      element tag1 {
        distinct-elements($class/*/xc:X),
        distinct-elements($class/*/xc:Y),
        distinct-elements($class/*/xc:Z)
```

```
  }   }
else ()
```

5.2.5 Grouping Constructs enclosing Sequences of Construct Terms

A recent development within the language Xcerpt is the admission of multiple construct terms within grouping constructs, such as in $Expr_{5.2.5_{XC}}$. For each unique pair of values for `X` and `Y`, one instance of the tuple enclosed by `all-distinct` is generated. This means that the same value for `Y` may appear multiple times within the enclosing `tag0`-element, interleaved by instances of `tag1 [ var X ]`.

$Expr_{5.2.5_{XC}} :=$

```
CONSTRUCT tag0 [ all-distinct (tag1 [ var X ], var Y) ]
FROM var X as a {{ var Y as b {{ }} }} END
```

This type of grouping can be formally defined by Equation 5.7, which specifies that the equivalence classes of the substitution set $[\![\sigma]\!]$ are calculated according to the union of the sets of free variables of the t_i, plus the explicit variables V – if present. The free variables within the grouping construct `all-distinct` in $Expr_{5.2.5_{XC}}$ are `X` and `Y`, and therefore the equivalence classes in the XQuery translation $Expr_{5.2.5_{XQ}}$ are calculated according to these two variables. As before, the XQuery variable `$ss` in $Expr_{5.2.5_{XQ}}$ contains the substitution set in XML-representation returned by the evaluation fo the translation of the query part of $Expr_{5.2.5_{XC}}$.

(5.7) $$[\![\sigma]\!](\texttt{all}\ (t_1,\ldots,t_l)\ \texttt{group by}\ V) = [\![\tau_1]\!](t_1) \circ \cdots \circ [\![\tau_1]\!](t_l) \circ \cdots \circ [\![\tau_k]\!](t_1) \circ \cdots \circ [\![\tau_k]\!](t_l)$$
$$\text{where } \{[\![\tau_1]\!],\ldots,[\![\tau_k]\!]\} = [\![\sigma]\!]/_{\simeq_{\bigcup_{1\leq i\leq l} FV(t_i)\cup V}}$$

$Expr_{5.2.5_{XQ}} :=$

```
element tag0 {
  let $reps := distinct-elements(
    for $s in $ss return
      element xc:restricted_substitution {
        $ss/child::xc:X, $ss/child::xc:Y })
  return for $r in $reps
    let $class_r := distinct-elements(
      for $s in $ss return
        if (fn:deep-equal($r/child::xc:X, $s/child::xc:X) and
            fn:deep-equal($r/child::xc:y, $s/child::xc:Y))
        then $s else ())
    return
      (element tag1 { $class_r/child::xc:X }, $class_r/chidl::xc:Y) }
```

Automatic translation of n-ary `all`-constructs is achieved by adapting rule 5.3 to treat a sequence of construct terms $ct_1,\ldots,ct_k$ rather than a single construct term ct within the `all-distinct`-construct. Apart from that, Rule 5.5 differs from Rule 5.3 in that it makes use of the helper functions from last section. The variables `X1, ..., Xn` in Rule 5.5 are the union of the free variables of the c_i.

$[\![$ `all-distinct` $(ct_1,\ldots,ct_k)\ ]\!]^{\$ss}_{toXQ}$ = (Rule 5.5)

```
      for $class in xc:eq-classes($ss, (), ("X1", ..., "Xn")) return
          ( [[ct_1]]^$class_toXQ, ..., [[ct_k]]^$class_toXQ )
```

To conclude the translation of construct terms, the following points are emphasized. Grouping constructs are best translated by staying close to the their formal semantics. This requires that some kind of data structure to represent subsitution sets is available, which must be provided by the translations of the query terms. In this

way entire Xcerpt rules are translated seamlessly to XQuery. The use of XQuery functions eases the translation notably. The constructs `order by`, `some` and aggregations remain unadressed in this thesis, but especially the handling of `some` is straightforward.

5.3 XQuery Expressions with Mixed Construction and Query Parts

As exemplified by the translations of Xcerpt construct terms, XQuery expressions often intertwine query and construction parts, such that they are incompatible with the grammar productions of XQ_3. As a consequence, they cannot be translated with the methods introduced so far. This section shows how to translate XQuery expressions consisting of `for`-clauses, variables, indirect element constructors and sequences of these constructs.

The grammar productions for this sublanguage are given in Table 5.5, with `<QNAME>` and `<XQVAR>` denoting qualified names and XQuery variables, respectively. Note that a lot of grammar productions that are included in XQ_3, e.g. those for injectivity, order, and deep-equality constraints, are excluded from XQ_4. Their inclusion would not add any interesting aspect to the focus of this section, but since they are only relevant to the calculation of query terms, they would neither complicate the rules presented here. An important constraint on the grammar productions in Table 5.5 is that XQuery variables may be bound at most once.

Table 5.5: XQuery expressions with mixed construction and query parts: XQ_4

```
<ELEMENT>   ::= 'element' <QNAME> '{' <SUBEXPR> (',' <SUBEXPR> )* '}'
<SUBEXPR>   ::= <ELEMENT> | <FORCLAUSE> | <XQVAR>
<FORCLAUSE> ::= 'for' <XQVAR> 'in' <STEP> 'return'
                '(' <SUBEXPR> (',' <SUBEXPR> )* ')'
<STEP>      ::= <XQVAR>'/child::'<QNAME>
```

A translation of an intertwined XQuery expression results in an entire Xcerpt construct-query-rule. Therefore, the translation procedure is split into two parts. The query components are translated by rules flagged $toXCqt$ in a very similar way expressions in XQ_3 are translated, only that it is not necessary to account for the substitution sets in XML-representation, and that XQuery element constructors are disregarded. The construction components, in other words the element constructors and variable references, are translated by rules named $toXCct$. The interaction between both functions is formalized by Rule 5.6.

As in the previous chapter, A denotes a sequence of associations between XQuery an Xcerpt variables, and may be calculated together with the query part $[\![E_{Intertwined}]\!]^{A}_{toXCqt}$. Within A, multiple XQuery variables may be associated with the same Xcerpt variable, but not vice versa. At least one association for each XQuery variable returned by $E_{Intertwined}$ must be included in A. It would be possible to calculate both the construct part and the query part in one single sweep over $E_{Intertwined}$, but this would require to hold more intermediate data-structures in memory.

The second parameter to $toXCct$ is a sequence containing XQuery variables, which the resulting Xcerpt construct term must be grouped by. It is initialized with the empty sequence [] and is discussed in detail later in this section.

$$A = associations(E_{Intertwined}) \qquad \text{(Rule 5.6)}$$

$[\![E_{Intertwined}]\!]_{toXC}$ = CONSTRUCT $[\![E_{Intertwined}]\!]^{A,[]}_{toXCct}$ FROM $[\![E_{Intertwined}]\!]^{A}_{toXCqt}$ END

In this section, the focus lies upon the construction part of the translation, since the handling of the query part is in fact very similar to the automatic translation from XQ_3 to XC_3. The discussion is split into two episodes. The first one gives a first idea on how to handle intertwined XQuery expressions, while the second one extends the discussion to include sequences of expressions within `for`-clauses.

5.3.1 Translation of Element Constructors and `for`-Clauses

In this section, a slightly restricted version of the language defined by the grammar in Table 5.5 is translated to Xcerpt. The restriction is given by not allowing `for`-clauses to return sequences of subexpressions. Sequences

within `for`-clauses are discussed separately in Section 5.3.2, because they require n-ary grouping constructs.

As an introduction, consider the intertwined expression $Expr_{5.3_{XQ}}$, in which the construction and query part are colored red and blue, respectively. For each element in `$data/child::html` one new node named `b` is constructed, thus the element constructor `b [ ... ]` must be grouped with respect to `$x`. Since `$x` does not belong to the free variables of the expression enclosed by `b[ ... ]`, explicit grouping must be used to specify the translation. A similar reasoning applies for the innermost element constructor with label `c`. For each subterm of `$x` named `body`, one such node is constructed. Consequently, the element constructor `c[...]` in the translation must be grouped by `Y`. `Y` occurs within the element constructor for `c`, and thus no explicit grouping is necessary. In both step expressions of $Expr_{5.3_{XQ}}$ duplicates are preserved, hence the duplicate preserving `all` construct is the right choice as the grouping construct.

An interesting characteristic of $Expr_{5.3_{XQ}}$ is that an element labelled `a` is produced no matter if `$x` and `$y` can be succesfully bound to a node. Similarly, for each binding of variable `$x`, one element named `b` is constructed, irrespectively of the potential bindings for `$y`. To reflect this situation, all Xcerpt labels that represent an XQuery element constructor within a `for`-clause must be declared optional.

$Expr_{5.3_{XQ}} :=$

```
element a {
  for $x in $data/child::html return
    element b {
      for $y in $x/child::body return
        element c { $y } } }
```

$Expr_{5.3_{XC}} :=$

```
CONSTRUCT
  a [ optional all b [ optional all c [ var Y ] ] group by X]
FROM
  optional var X as html {{ optional var Y as body {{ }} }}
END
```

If `element b` were not enclosed by a `for`-clause, the first grouping construct in the translation could be left away. In contrast, if there were more than one `for`-clause enclosing `element b`, it would be necessary to group by more than one variable. When translating an element constructor c to Xcerpt, the sequence of variables $V_{sur}(c)$, that are bound in `for`-clauses surrounding it, must be known.

Keeping track of these variables in a recursive translation function is most easily achieved by starting out with the empty sequence and appending a variable for each `for`-clause consumed. Note that it would not suffice to store $V_{sur}(c)$ within a set, because the Xcerpt translation must return subterms in the same order as the XQuery archetype. In the following, the subset relation is occasionally used when referring to sequences, meaning that all elements in the first sequence must be contained in the second sequence in an arbitrary order. In Rule 5.7 "++" denotes the function concatenating two lists, and `[$var]` denotes the list consisting of the single element `$var`.

$$[\![\texttt{for \$var in step return } e]\!]^{A,Vars}_{toXCct} = [\![e]\!]^{A,Vars\texttt{++[\$var]}}_{toXCct} \qquad \text{(Rule 5.7)}$$

It does not make sense, however, to group an expression twice by the same variable. Therefore, $V_{group}(c)$ denotes the sequence obtained by eliminating all those variables from $V_{sur}(c)$ which also participate in a grouping produced by the translation of an element constructor surrounding c. $V_{group}(c)$ is the sequence of variables that c needs to be grouped by, and is given as a superscript to the rule. It is easiest to specify a translation rule for an element constructor that does not need to be grouped:

$$[\![\texttt{element label \{ } e_1, \ldots, e_n \texttt{ \} }]\!]^{A,[]}_{toXCct} = \texttt{label [} [\![e_1]\!]^{A,[]}_{toXCct} \ldots [\![e_n]\!]^{A,[]}_{toXCct} \texttt{]} \qquad \text{(Rule 5.8)}$$

In the case that $V_{group}(c)$ is not empty, grouping constructs are used in the translation. Let $FV(c) \subseteq V_{sur}(c)$ be the set of free variables (excluding `$data`) of the construct term to be translated. $FV(c)$ is a subset of $V_{sur}(c)$, because expressions that contain unbound variables (except for `$data`) are not translated. Free variables of a subexpression must therefore be bound by some construct within the surrounding expressions. Note that, with implicit and explicit grouping giving no guarantees about the order of the resulting subelements,

these grouping constructs are not an option, even if $FV(c) = V_{group}(c)$ holds. Instead the `order by`-clause must be used to establish an order according to the bindings for the variables in the sequence $V_{group}(c)$.

Rule 5.9 recursively calls the translation function $toXCct$ on the subexpressions $e_1, \ldots, e_n$ of c, substituting $V_{group}(c)$ by the empty list in order to prevent multiple groupings with respect to the same variable. The notation $A(V_{group})$ refers to the Xcerpt variables associated with the XQuery variables in the list V_{group}.

$$\frac{c \texttt{ = element label \{ } e_1, \ldots, e_n \texttt{ \}}}{[\![c]\!]^{A,V_{group}}_{toXCct} \texttt{ = optional all label [} [\![e_1]\!]^{A,[]}_{toXCct}\texttt{, ..., } [\![e_n]\!]^{A,[]}_{toXCct} \texttt{]} \\ \texttt{order by \{ } A(V_{group}) \texttt{ \}}} \quad \text{(Rule 5.9)}$$

The last construct from the grammar to be automatically translated is also the simplest one: XQuery variable occurrences of a variable v. In the same way as with element constructors, two cases need to be distinguished: Either the list $V_{group}(v)$ of variables that v must be grouped by, is empty - which is the case if v is directly surrounded by an element constructor - or v must in fact be grouped by other variables, because it is directly surrounded by possibly multiple nested `for`-clauses. In the first case the translation of v is given by $A(v)$ and in the second case, Rule 5.10 applies.

$$[\![\ v \ ]\!]^{A,V_{group}}_{toXCct} \texttt{ = all } A(v) \texttt{ order by \{ } A(V_{group}) \texttt{ \}} \quad \text{(Rule 5.10)}$$

As before, $A(v)$ denotes the Xcerpt variable associated with v during the translation of the query term, and $A(V_{group})$ the representatives of the variables, that v is to be grouped by.

5.3.2 Returning Sequences Within `for`-Clauses

The translation rules above precluded the case of sequences returned by for-clauses. This section answers the question what to do if e in Rule 5.7 is anything else but a single element constructor, a single `for`-clause or a single variable. Example $Expr_{5.3.2_{XQ}}$ iterates over the `body` child nodes of the `html` elements of `$data` and returns both the `html` element and the `body` element whenever `$v2` is successfully bound to a node. While each node bound to `$v2` is returned exactly one time, the number of times the same binding for `$v1` is returned remains unassured. Additionally, `html`- and `body` nodes appear alternately within the result sequence, and they must be grouped with respect to both variables `$v1` and `$v2`.

$Expr_{5.3.2_{XQ}} :=$

```
element a {
  for $v1 in $data/child::html return
    for $v2 in $v1/child::body return ($v1, $v2) }
```

The translation of the query part of $Expr_{5.3.2_{XQ}}$ is evidently given by

```
optional var V1 as html {{ optional var V2 as body {{ }} }}
```

The construct part of the translation is more difficult to figure out. Applying the substitution set generated by the query part to the term `a [ all var V1, all var V2]` would result in a sequence that includes all of the data terms that are returned by expression $Expr_{5.3.2_{XQ}}$, but neither the correct number of these nodes, nor in the correct order. Moreover, it is impossible to specify a translation of $Expr_{5.3.2_{XQ}}$ with the initial language constructs suggested in [17].

A recent extension to Xcerpt are n-ary variants of the constructs `optional`, `without` and `all`. In fact, a grouping construct that takes a sequence of construct terms is necessary to phrase the correct translation ($Expr_{5.3.2_{XC}}$). To see that this is a correct translation of $Expr_{5.3.2_{XQ}}$, reconsider the formal semantics of this n-ary grouping construct as suggested in Section 5.2.5.

$Expr_{5.3.2_{XC}} :=$

```
CONSTRUCT
  element a [ all (var V1, var V2) order by V1]
FROM
  optional var V1 as html {{ optional var V2 as body {{ }} }}
END
```

In $Expr_{5.3.2_{XC}}$, one pair of variable assignments for `V1` and `V2` is returned for each of the equivalence classes $\{[\![\tau_1]\!] \ldots [\![\tau_k]\!] = \Sigma/_{\simeq_{\{V_1,V_2\}}}$ of the substitution set Σ. As before, this translation is only correct under the assumption that the substitution sets are in fact multisets of substitutions and that the grouping construct `all` preserves duplicates.

Mixed sequences returned by `for`-clauses According to the grammar in Table 5.5, not only multiple variables may occur within the `return`-clause of a `for`-clause. Multiple other `for`-clauses or element constructors are translated in the same way. As an example consider expression $Expr_{5.3.2_{XQ}}$, which returns a heterogeneous sequence within a `for`-clause.

$Expr_{5.3.2_{XQ}} :=$

```
1  element a {
2    for $v1 in $data/child::html return (
3      $v1,
4      element b { $v1/child::* },
5      for $v2 in $v1/child::* return $v2/child::* )
6  }
```

Again, the sequence is echoed in the Xcerpt translation within an `all`-construct. The variable reference `$v1` and the element constructor `element b` are translated in the same manner as in the last subsection.

The advertent reader may have noticed that $Expr_{5.3.2_{XQ}}$ is under strict considerations not part of the language XQ_4, because it is not derivable by the grammar productions in Table 5.5. To be more precise, the deviance lies in the usage of step expressions within the element constructor for `b` in line four and in the `return`-clause of line five. Both of these step expressions may be transformed into a `for`-clause explicitely binding new variables `$v3` and `$v4`. In this way, the translation rules given in the previous sections suffice to derive the correct translation $Expr_{5.3_{XQ}}$.

$Expr_{5.3_{XC}} :=$

```
CONSTRUCT
  a [ optional all (
      var V1, b [ all var V2 ], all var V3 order by {V2}
  ) ]
FROM
  optional var V1 as html {{ var V2 as /.*/ {{ var V3 }} }}
END
```

A further difficulty for the translation procedure is to recognize that the step `$v1/child::*` appears twice in $Expr_{5.3_{XQ}}$ and thus it is convenient to use the same Xcerpt variable to represent both XQuery variables `$v2` and `$v3` of the expanded version of the expression.

5.4 XQuery Expressions with Construction of Intermediate Results

Although XQ_4 includes expressions that both query data and construct results, it is restricted in an important way. Once XQ_4 expressions construct results, the constructed data is never reused later on in the XQuery expressions. In other words, the querying is restricted to the variable `$data` and its descendants. In this section this confinement is lifted by the introduction of additional `let`-clauses. The grammar production for this new sublanguage called XQ_5 is given in Table 5.6.

The newly introduced `let`-clauses may bind their variables to an arbitrary XQuery expression deducable by `<SUBEXPR>`. In this respect, they are privileged in comparison to `for`-clauses, the binding sequences of which are still limited to step expressions. The question whether this kind of extension of XQ_4 is sheer arbitrariness, immediately comes to one's mind. In fact, also `for`-clauses with binding sequences more complex than step expressions are indirectly expressible by the above grammar productions, since such binding sequences may be substituted by a variable bound in a `let`-clause as exemplified by the following transformation:

Table 5.6: Grammar productions for XQ_5

```
<ELEMENT>   ::= element <QNAME> { <SUBEXPR>* }
<SUBEXPR>   ::= <ELEMENT> | <FORCLAUSE> | <XQVAR> | <LETCLAUSE>
<FORCLAUSE> ::= for <XQVAR> in <STEP> return (<SUBEXPR>*)
<LETCLAUSE> ::= let <XQVAR> := <SUBEXPR> return <SUBEXPR>
<STEP>      ::= <XQVAR>/child::<QNAME>
```

```
for $v1 in <SUBEXPR1> return <SUBEXPR2> ==
    let $myvar := <SUBEXPR1> return for $v1 in $myvar return <SUBEXPR2>
```

This example shows that it is often convenient to transform XQuery expressions before they are translated to Xcerpt. The rich equality preserving transformation possibilities of XQuery expressions are of intense interest in later parts of this section. As a matter of fact, there are two possibilities of translating the sublanguage XQ_5 to Xcerpt: The first one is direct translation of `let`-clauses to Xcerpt construct-query-rules, and the second one using a mapping of expressions of XQ_5 into XQ_4. Consequently, the discussion of XQ_5 is split into these two parts. But before both methods are presented, consider a short example expression.

$Expr_{5.4_{XQ}}$ highlights the difficulty of translating XQuery expressions with intermediately constructed and queried results.

$Expr_{5.4_{XQ}} :=$

```
let $var :=
  element b { for $v0 in $data/tag0 return element a { $v0 }}
return element a {
  for $v1 in $var/child::a return
    for $v2 in $data/child::b return
      element c { $v1, $v2 } }
```

The variable `$var` is bound to an expression including element constructors and these element constructors are deconstructed later on. In the following sections different ways are shown to translate $Expr_{5.4_{XQ}}$.

5.4.1 Intermediate Results Translated by Rule Chaining

During a forward chaining evaluation of Xcerpt programs, intermediate results are constructed in a similar fashion to the computation of the binding for `$var` in $Expr_{5.4_{XQ}}$. This suggests the translation of each `let`-clause by its own appropriate Xcerpt rule.

Whenever the variable bound in the `let`-clause is referenced within the XQuery expression, the Xcerpt translation must query the data constructed by the rule associated with the `let`-clause. For these queries to succeed only on the desired data, the construct parts of the generated rules must be uniquely identifiable. This is most easily achieved by using a special namespace, and the variable name of the variable bound in the `let`-clause as an enclosing label for the construct term. Of course this is only true under the easily enforcable assumption that each variable has a unique name in XQ_5 expressions.

In this manner, $Expr_{5.4_{XQ}}$ translates to $Expr_{5.4.1_{XC}}$. As in the last section, all Xcerpt variables must be optional, because results are generated by $Expr_{5.4_{XQ}}$ no matter if any bindings for the variables `$v1` and `$v2` exist.

$Expr_{5.4.1_{XC}} :=$

```
CONSTRUCT
   xc:var { b [ optional all a [ var V0 ] ] }
FROM
   optional var V0 as tag0 {{ }}
END

CONSTRUCT
   a [ optional all c [ var V1, var V2 ] ]
```

```
FROM
   and ( xc:var {{ /.*/ {{ optional var V1 as a {{ }} }} }},
         optional var V2 as b {{ }} )
END
```

Unfortunately, this kind of translation only works for `let`-clauses appearing on a global level and fails for certain kinds of nested `let`-clauses or ones appearing within `for`-clauses. The problem arises from the fact that `let`-clauses that are not the outermost construct of an XQ_5 expression are evaluated in a customized dynamic environment (see sections 2.2.1 and 2.2.2). Nevertheless, some of these wrapped let-clauses are still translatable by the above procedure as the following example shows.

$Expr_{5.4.1_{XQ}} :=$

```
for $v0 in $data/tag0 return
  for $v1 in $v0/tag1 return
    let $v2 := intermediate_result { $v1 } return
      for $v3 in $v2/child::tag0 return
        for $v4 in $v3/child::tag1 return
          element result { $v4 }
```

The `let`-clause and both of its surrounding `for`-clauses in $Expr_{5.4.1_{XQ}}$ may be translated almost as if they were an autonomous XQuery expression into the first rule in $Expr_{5.4.1^b_{XC}}$. The remaining `for`-clauses and the element constructor may then be translated to form the second rule in $Expr_{5.4.1^b_{XC}}$, which queries the construct parts of the first rule.

$Expr_{5.4.1_{XC}} :=$

```
CONSTRUCT
  xc:v2 [ optional all intermediate_result [ var V0, var V1 ] ]
FROM
  optional var V0 as tag0 {{ optional var V1 as tag1 {{ }} }}
END

GOAL
  xc:result [ optional all result [ var V4 ] ]
FROM
  xc:v2 {{
    intermediate_result {{ tag0 {{ optional var V4 as tag1 {{ }} }} }}
  }}
END
```

Note that it is not strictly necessary to group all `intermediate_result`-elements within the same `xc:v2`-element. The program would yield the same results if the `all`-keyword were left away.

As mentioned above, slight variations of $Expr_{5.4.1_{XQ}}$ may cause this translation procedure to miscarry. One such variation would be to not only return `$v4` in the innermost element constructor, but also one of the variables `$v0` and `$v1`. Bindings for their Xcerpt equivalents `var V0` and `var V1` are only generated during the evaluation of the query part of the first rule, and are not present for the construct part of the second one. In order to produce fresh bindings for both of these variables, one could adapt the query part of the second rule to read as follows:

```
and (
  xc:v2 {{ intermediate_result {{
    tag0 {{ optional var V4 as tag1 {{ }} }}
  }} }},
  optional var V0 as tag0 {{ optional var V1 as tag1 {{ }} }}
)
```

While bindings for `V0` and `V1` can now be consumed in the construct part of the second rule, there is no guarantee that the bindings for `V4` are brought together with the correct bindings for `V0` and `V1`. To make

things worse, all possible combinations of bindings for these variables are used, producing a large amount of false results. In the following section, a method for surmounting these shortcomings is presented by simulating XQuery dynamic environments within Xcerpt.

5.4.2 Simulation of XQuery Dynamic Environments Within Xcerpt

As demonstrated in sections 5.2, construct terms are translated most easily by carrying over the notion of substitution sets, which play a crucial role in the evaluation of Xcerpt rules, to XQuery. In this section, dynamic environments as a data structure that plays an important role in the evaluation of XQuery expressions, are emulated in Xcerpt to translate arbitrary expressions in XQ_5.

The example expression to be considered in this section is a more complex version of $Expr_{5.4.1_{XQ}}$:

$Expr_{5.4.2_{XQ}} :=$

```
for $v0 in $data/tag0 return
  for $v1 in $v0/tag1 return
    let $v2 := intermediate_result { $v0, $v1 } return
      for $v3 in $v2/child::tag0 return
        for $v4 in $v3/child::tag1 return
          element result { $v4, $v0 }
```

It does not only return variables bound in immediately enclosing `for`-clauses, but also the outermost variable `$v0` in the innermost element constructor. In the last section it has been shown that when translating `let`-clauses with appropriate construct-query-rules, it may be unknown which variable bindings belong together in the sense that they are appear within the same dynamic environment during an XQuery evaluation. In the translation of $Expr_{5.4.2_{XQ}}$ this problem is solved by making dynamic environments explicit:

$Expr_{5.4.2_{XC}} :=$

```
CONSTRUCT
  xc:env0 [ xc:v0 [ var V0 ] ]
FROM
  var V0 as tag0 {{ }}
END

CONSTRUCT
  xc:env1 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ] ]
FROM
  xc:env0 [ xc:v0 [ var V0 as /.*/ {{ var V1 as tag1 {{ }} }} ] ]
END

CONSTRUCT
  xc:env2 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ],
            xc:v2 [ intermediate_result [ var V0, var V1 ] ] ]
FROM
  xc:env1 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ] ]
END

CONSTRUCT
  xc:env3 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ],
            xc:v2 [ var V2 ], xc:v3 [ var V3 ] ]
FROM
  xc:env2 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ],
            xc:v2 [ var V2 as /.*/ {{ var V3 as tag0 {{ }} }} ] ]
END

CONSTRUCT
  xc:env4 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ], xc:v2 [ var V2 ],
```

```
            xc:v3 [ var V3 ], xc:v4 [ var V4 ] ]
FROM
  xc:env4 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ], xc:v2 [ var V2 ],
            xc:v3 [ var V3 as /.*/ {{ var V4 as tag1 {{ }} }} ],
END

CONSTRUCT
  xc:result [ all result [ var V4, var V0 ] ]
FROM
  xc:env4 [[ xc:v0 [ var V0 ], xc:v4 [ var V4 ] ]]
END
```

The dynamic environment for the final return clause `element result { $v4, $v0 }` is built up step by step in the Xcerpt translation. Bindings for the variables `$v0` ... `$v4` are added one after another, yielding the intermediate environments `xc:env0` ... `xc:env4`. Regrettably, the length of the translation does not scale linearily with the number of constructs in the XQuery preimage. The number of rules is given by the number of `for`- and `let`-clauses c, and the length of these rules also increases linearily with the number of variables in the environment, which is equal to the number of clauses c. Thus, the total complexity is $O(c^2)$.

There is, however, some room for improvement. No intermediate results being constructed between the application of the rules one, two and three, these rules may be merged to form one single rule. With the same reasoning in mind, one may also combine rules four, five and six to obtain the following program as an abbreviated alternative to $Expr_{5.4.2_{XC}}$.

```
CONSTRUCT
  xc:env2 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ],
            xc:v2 [ intermediate_result [ var V0, var V1 ] ] ]
FROM
   var V0 as tag0 {{ var V1 as tag1 {{ }} }}
END

CONSTRUCT
  xc:result [ all result [ var V4, var V0 ] ]
FROM
  xc:env2 [ xc:v0 [ var V0 ], xc:v1 [ var V1 ],
     xc:v2 [ var V2 as /.*/ {{
        var V3 as tag0 {{ var V4 as tag1 {{ }} }}
     }} ] ]
END
```

Let c be the entire number of clauses within an XQ_5 expression as before, and l the number of `let`-clauses. The length of the result of this translation procedure is still not linear in c, but the complexity is $O(c \cdot l)$.

In contrast to the translation in the last section, this method allows to translate all expressions within XQ_5 to Xcerpt. Furthermore, it may be easily implemented to obtain an automatic translation algorithm. Aligned with the formal semantics of XQuery, it is extensible to handle also function calls or other XQuery constructs that manipulate the dynamic environment.

5.4.3 Elimination of `let`-Clauses

As shown in the last section, rule-chaining in conjunction with the emulation of dynamic environments allow to translate all expressions in XQ_5 to Xcerpt. Nevertheless, this section explores a different approach by trying to convert all expressions within XQ_5 to semantically equivalent expressions in XQ_4.

The basic idea is to eliminate `let`-clauses by substituting variable references in the body of `let`-clauses by their definition. Subsequently, the resulting expression is simplified to comply with the grammar productions of XQ_4. Since all expressions in XC4 have been shown to be straight-forwardly translatable to Xcerpt, this is an effective way to circumvent the direct translation of XQ_5.

Note that `let`-elimination in XQuery unlike in most functional programming languages may cause trouble in certain cases. Due to node-identity it is not always possible to guarantee that the result of the evaluation

of an expression stays the same when substituting variables by their bindings. As an example consider the following queries:

```
let $a := element a { } return ($a is $a)

element a { } is element a { }
```

The second query is the result of eliminating the `let`-clause in the first query. Nevertheless, the first query evaluates to true, whereas the second one evaluates to false [1]. This infringement of referential transparency in XQuery extends to functions that make use of the comparison operator for node identities `is` such as `fn:count` and `fn:distinct-nodes`, but since none of these constructs are included in XQ_5, `let`-elimination is a correct transformation in this scope.

This section is structured as follows: At first, an example expression is transformed according to the ideas above. In the second subsection, it is shown that all expressions resulting from the elimination of let-clauses may be transformed to comply with the grammar of XQ_4.

5.4.3.1 An Introductory Example

To get a basic idea of the transformations discussed in this section, consider the example expression $Expr_{5.4.3_{XQ}}$. It is not part of the sublanguage XQ_4 because it includes a `let`-clause.

$Expr_{5.4.3_{XQ}} :=$

```
let $a := element a {
  for $v0 in $data/child::tag0 return element b { $v0 }
} return
  for $v1 in $a/child::b return element c { $v1 }
```

Getting rid of this `let`-clause is easily achieved by substituting all references to variables v which are bound in `let`-clauses by the body of the respective `let`-clause b. In the following generic transformation formula this substitution is denoted by a subscript $\{v \mapsto b\}$.

$$\texttt{let } v \texttt{ := } b \texttt{ return } e \texttt{ == } e_{\{v \mapsto b\}}$$

Applying this formula to $Expr_{5.4.3_{XQ}}$ yields the following equivalent expression, which unfortunately neither complies with the grammar productions for XQ_4 nor XQ_5. While this might seem like a step in the wrong direction at first glance, the result of the `let`-elimination may be transformed to fit into XQ_4.

```
for $v1 in (element a {
    for $v0 in $data/child::tag0 return element b { $v0 }
  }/child::b)
return element c { $v1 }
```

To the advertant reader it may be obvious that $Expr_{5.4.3_{XQ}}$ performs superflous computations. The `let`-elimination sets the stage for removing this redundancy. In fact, the intermediate transformation result above constructs an element `a` and immediately applies the child axis on it. Therefore, one might just as well not construct the new node and apply the self axis on the content of the element, as in the following equivalent expression:

```
for $v1 in (
  (for $v0 in $data/child::tag0 return element b { $v0 })/self::b
) return element c { $v1 }
```

The general form of this transformation is given by

$$\texttt{(element } label_1 \texttt{ \{ } e \texttt{ \})/child::} label_2 \texttt{ == } e\texttt{/self::} label_2$$

The name test `self::b` is applied to the sequence resulting from the evaluation of a `for`-clause. The same result is obtained by applying the name test to the expression in the body of the `for`-clause.

[1] This problem has been pointed out by Daniela Florescu on http://lists.xml.org/archives/xml-dev/200412/msg00228.html and in [5]

```
for $v1 in (
  for $v0 in $data/child::tag0 return (element b { $v0 }/self::b)
) return element c { $v1 }
```

This transformation uncovers another redundant computation, namely the name test itself, which always holds true and might just as well be omitted:

```
for $v1 in (
  for $v0 in $data/child::tag0 return element b { $v0 }
) return element c { $v1 }
```

In the case that the label names do not comply, the body of the `for`-clause may be simplified to the empty sequence. The general form of this transformation is given by:

$$\texttt{element}\ label_1\{e\}/\texttt{self::}label_2 == \begin{cases} \texttt{element}\ label_1\{e\} & \text{if } label_1 = label_2, \\ () & \text{otherwise.} \end{cases} \tag{5.8}$$

The following transformation is the least obvious, but the most important one. It serves to rearrange the query part of the XQuery expression, such that it is easier to translate to Xcerpt. While the first query action in the query above is given by the binding sequence of the first for-clause, this it not obvious. To bring the syntactical order in accordance with the evaluation order, the `for`-clause in the binding sequence may be brought outside of its surrounding `for`-clause:

```
for $v0 in $data/child::tag0 return
  for $v1 in element b { $v0 } return
    element c { $v1 }
```

Note that this transformation works for any expression at the place of `element b { $v0 }`. In this example, the transformation spawns a `for`-clause with a binding sequence containing only one single element. Such `for`-clauses do not actually iterate over anything, but are in fact alternative notations for `let`-clauses, which themselves may be eliminated.

```
for $v0 in $data/child::tag0 return
  element c { element b { $v0 } }
```

In [13, Section 3.3] this transformation of iterations over unit forests (`for` v `in` e_1 `do` e_2 `=` $e_{2\{v \mapsto e_1\}}$) is called the *left unit law* and may only be applied if e_1 is not a sequence.

Finally brought in conformance with the grammar of XQ_4, the expression is translated to Xcerpt as usual:

```
CONSTRUCT
  xc:result [ optional all c [ b [ var V1 ] ] ]
FROM
  var V1 as tag0 {{ }}
END
```

The introductory example in this section shows that it is possible to transform some of the expressions obtained by `let`-elimination to XQ_4. In the following section it is demonstrated that thanks to the monadic nature of XQuery `for`-clauses [13] all such expressions are transformable.

5.4.3.2 Systematic Transformation of Complex Binding Sequences

By means of the `let`-elimination, variable occurences may be substituted by arbitrary XQ_5 subexpressions. The grammar productions for XQ_5 (Table 5.6) allow variable references to appear either within the body of `for`- and `let`-clauses, within element constructors or within step expressions. Variable references within the body of `let`-clauses are of no interest to us, because `let`-elimination takes care of them. In case the replaced variables appear within element constructors, or within the bodies of `for`-clauses, the resulting expression is still within XQ_4, and can be translated to Xcerpt. More difficult cases arise if the variable is part of a binding sequence. In this section, all possible expressions resulting from the replacement of XQuery variables in binding sequences are converted to such expressions that comply with the grammar for XQ_4.

For this end, it must be distinguished between element constructors, `for`-clauses and lists of expressions at the place of variable references in binding sequences. The transformations are given in a general form that is easily implementable in a programming language. As in the previous sections, the meta-variables v_i, e_i, and $l_1, i \in \mathbb{N}$ represent XQuery variables, subexpressions (derivable by <SUBEXPR> in Table 5.6) and labels , respectively.

Element constructors within binding sequences The first case to be considered is the one of element constructors. As has been seen during the treatment of the example expression, this case often eliminates redundancies within the original XQ_5 expression. In fact the application of the child-axis to an element constructor yields the same result irrespectively of the name of the element constructed.

$$\texttt{for } v_1 \texttt{ in (element } l_1 \texttt{ \{ } e_1 \texttt{ \}/} l_2\texttt{) return } e_2 \texttt{ ==} \\ \texttt{for } v_1 \texttt{ in } e_1\texttt{/self::} l_2 \texttt{ return } e_2$$

After eliminating the element constructor, the name test must still be carried out on the subexpression itself. Since also the `self`-axis is excluded from XQ_4, the resulting expression must be further simplified. Again, it must be distinguished between element constructors, let-clauses, variable references, for-clauses and binding sequences.

- As discussed in the last section, the application of a name test on an element constructor can be handled according to Equation 5.8. In the case of matching label names, the resulting `for`-clause together with its undesirably complex binding sequence is disposed by the *left unit law* [13, Section 3.3] (given in Section 5.4.3.1) just as in the example expression.
- `Let`-clauses are not much harder to handle. They are simply eliminated as already described, and the self axis is taken care of by one of the other transformations described in this paragraph:

$$\texttt{for } v_1 \texttt{ in (let } v_2 \texttt{ := } e_1 \texttt{ return } e_2\texttt{)/self::} l_1 \texttt{ return } e_3 \texttt{ ==} \\ \texttt{for } v_1 \texttt{ in } e_{2\{v_2 \mapsto e_1\}}\texttt{/self::} l_1 \texttt{ return } e_3$$

- Applying a name test to a variable reference can be resolved by finding out the binding sequence of the variable. In XQ_5, these binding sequences are always short path expressions consisting of a reference to another variable, a step on the `child`-axis, and a name test. If the name test in this binding sequence and the name-test to be eliminated overlap, the later one may simply be omitted. Otherwise, the whole expression evaluates to the empty sequence. Of course this requires global knowledge about all bound variables during the transformation process. In a recursive translation function this may be achieved by keeping a list of bound variables together with their label names in memory. Name test on variables are formally eliminated by Equation 5.9.

$$\texttt{for } v_1 \texttt{ in } v_2\texttt{/self::} l_1 \texttt{ return } e \texttt{ ==} \begin{cases} e_{\{v_1 \mapsto v_2\}} & \text{if } label(v_2) = l_1, \\ () & \text{otherwise.} \end{cases} \tag{5.9}$$

  As a matter of fact, the case of overlapping label names may be split in two parts: Having disposed the name-test, a simplistic `for`-clause is left over: `for` v_1 `in` v_2 `return` e. According to the *left unit law* ([13, Section 3.3]) for iterations in the XQuery algebra, this simplifies to $e_{\{v_1 \mapsto v_2\}}$.

- Name tests on `for`-clauses may be the most interesting case, although just as with `let`-clauses, the name-test on the `self::` axis is not eliminated but rathermore postponed.

$$\texttt{for } v_1 \texttt{ in (for } v_2 \texttt{ in } e_1 \texttt{ return } e_2\texttt{)/self::} l_1 \texttt{ return } e_3 \texttt{ ==} \\ \texttt{for } v_2 \texttt{ in } e_1 \texttt{ return (for } v_1 \texttt{ in } e_1 \texttt{ return } e_3\texttt{/self::} l_1\texttt{)}$$

  In [13, Section 3.3], this transformation is called the *associative law* among the three monad laws.

  The transformation above is also used in order to get rid of `for`-clauses directly substituted for variable references in binding sequences later on in this section.

- The final subexpressions that a node-test might be applied to, are sequences of XQ_5 values. Again, this is not a final case of the transformation, but the expression must be further simplified in order to get rid of the name tests on the sequence items $i_1, \ldots, i_k$.

```
for v1 in (i1,...ik)/self::l1 return e1 ==
       for v1 in (i1/self::l1, ..., ik/self::l1) return e1
```

Having shown that all name tests on the `self::`-axis can be eliminated to expressions within XQ_4, it is also apparent that element constructors as binding sequences can be transformed to fit into XQ_4. The remaining subexpressions within binding sequences are `for`-clauses and sequences. Both of these cases are similar to the corresponding sub-cases above.

For-clauses within binding sequences As the second type of subexpressions of binding sequences that do not comply with the grammar of XQ_4, `for`-clauses are considered. Unsurprisingly, it is again the *associative law* from [13] that can be applied:

```
for v1 in (for v2 in e1 return e2)/child::l1 return e3 ==
       for v2 in e1 return (for v1 in e2/child::l1 return e3)
```

It is easy to see that in the special case that $e_2 = v_2$ or that $e_3 = v_1$ the simplification can be carried even further. In [13, Section 3.3] a simplification of the form `for` v `in` e `return` v `=` e is called the *right unit law*.

In contrast to element constructors, the single formula above suffices to get rid of all `for`-clauses within binding sequences. The final complex expressions within binding sequences of `for`-clauses are sequences of subexpressions at the place of variable references.

Sequences within binding sequences The binding sequences to be transformed in this paragraph are of the form `(`e_1`, ..., `e_k`)/child::`l, and are generated by substituting the sequence of values `(`e_1`, ..., `e_k`)` for an XQuery variable reference in a valid XQ_4 binding sequence during `let`-elimination. By generating one appropriate `for`-clause for each element in the sequence, the expressions may be transformed to fit into XQ_4.

```
for v1 in (e1, ..., ek)/child::l return e ==
       (for v1 in e1/child::l return e,
        ...,
        for v1 in ek/child::l return e)
```

Therefore any expression in XQ_5 may be translated to Xcerpt by taking the indirection over XQ_4, if one is prepared to put up with the elongation of the expressions caused by `let`-elimination.

Concluding the treatment of XQ_5, it can be recorded that there exist viable translation possibilities for XQuery expressions with intermediately constructed results. Unfortunately neither the emulation of XQuery dynamic environments in Xcerpt nor `let`-elimination yield linear complexity of the length of the Xcerpt translations, although the former method comes close.

Chapter 6

Future Work and Conclusion

Although large parts of XQuery and Xcerpt are shown to be translatable, research in this area is far from finished. Many aspects of the translation could not be considered in their entirety during. The first Section of this Chapter serves to point out some possible directions for future work, presenting ideas that may be extended to form autonomous topics of interest. The second Section concludes this thesis by summing up insights and contributions.

6.1 Future Work

Future work may be classified into three categories:

- Research that builds upon the methods introduced in this thesis like the translation of entire Xcerpt programs, translation of query terms including aggregations, positional and label variables, and translation of XQuery function calls to Xcerpt.
- Entirely different approaches. One of these alternative approaches is the implementation of simulation unification in XQuery, and may be used as the foundation for a backward chaining evaluation of Xcerpt programs in XQuery. Another alternative approach would be to give up the translation of Xcerpt to the XQuery core, but to use XPath expressions instead whenever possible. This may be more efficient for a large number of XQuery implementations.
- As shown in this thesis, there are some language constructs that are hard to translate and some unique features of both languages that complicate the entire translation process. There are three approaches for explaining these differences. Either the expressions that are not easily expressible in one language are of subordinate importance so that they could slip by unnoticed, one or both of the languages miss some features that might be added in the future, or the differences in expressibility simply stem from the fact that the languages are not designed for the same purposes. Unanswered questions in this area are the following: Is it possible to integrate the concept of node identity in Xcerpt without giving up on referential transparency, answer closedness and declarativity? Is it beneficial to include `and` and `or`-connectives for query terms? Would it be possible to conceive modules for XQuery that deliver some of Xcerpts reasoning capabilities or integrate query and construct terms into XQuery?

In the following Sections, some of the above named ideas are described in more detail.

6.1.1 Translation of Xcerpt Programs

The first Chapters of this thesis show how to translate Xcerpt query terms to XQuery. Making use of the substitution sets in XML-representation returned by the translations of query terms, and staying close to the formal semantics of Xcerpt with respect to the application of substitution sets to construct terms, Chapter 5 presents a method for translating entire construct-query-rules. The next step would be to translate entire Xcerpt programs to XQuery.

Xcerpt programs consisting of Xcerpt rules, a counterpart of rules must be found to take over their role in XQuery. XQuery functions operating on the global variable `$data`, which is already used by the translation of query terms, lend themselves for this purpose. The following rule and its translation, which makes use of

the function `xc:eq-classes` from Section 5.2.4 for the sake of brevity, exemplify this idea. The query term is translated in the same way as discussed in Chapter 4, only that slightly longer path expressions are used. In XQ_3 a binding sequence of the form `$v0/exercised_by/*` would have to be expanded to an additional `for`-clause.

```
CONSTRUCT
  countries_and_sports [
    all countries [ var Country, all var Sport ]
  ]
FROM
  sports [[
    type [[ var Sport ]],
    exercised_by [[ var Country ]]
  ]]
END
```

```
declare xc:function1 () as node() {
  let $ss :=
    for $v0 in $data/sports return
      for $v1 in $v0/type/* return
        for $v2 in $v0/exercised_by/* return
          element xc:substitution {
            element xc:Sport { $v1 }, element xc:Country { $v2 },
          }
  return
    if ($ss) then
      element countries_and_sports {
        for $class in xc:eq-classes($ss, (), ("Country")) return
          element countries {
            xc:distinct-elements($class/*/xc:Country),
            xc:distinct-elements($class/*/xc:Sport) }
      }
    else ()
};
```

Rules that are associated with input resources, may be translated to a function that features an additional `let`-clause to bind a special variable `$doc` to the document resource. Subsequently `$doc` must be used at the place of `$data` in the function body. A similar procedure may be applied for multiple resource specifications.

Supposed that all rules of an Xcerpt program P are translated to XQuery functions `xc:function1, ... , xc:functionk`, running the program in XQuery can be achieved as follows:

- Compute the full stratification $P = P_1 \uplus \ldots \uplus P_n$ of P according to [17, Definition 6.8]. If the program is not stratifiable, it is not translatable to XQuery using this forward chaining approach.
- Compute the fixpoint interpretation M_P [17, Definitions 7.8 and 7.9] of the program. Corresponding to the formal semantics, at first the rules classified in the lowest stratum P_1 are evaluated until a fix point is reached, followed by those in P_2, and so forth.
- XQuery being a functional programming language, there is no equivalent to the facts of a logical program known at a particular point in the evaluation of a program. As one might guess, the most straightforward way to simulate the growing set of facts during the run of the program is to append the results of the evaluation of a function to the list of children of the special variable `$data`.

While this approach builds heavily upon the results of this thesis, implementing backward chaining must be considered an alternative approach to this suggestion. Since backward chaining relies on rooted graph simulation, a taste for both of these methods is given in the next Section.

6.1.2 An XQuery Reasoning Module

One of the strengths of the language Xcerpt are its advanced reasoning capabilities. With XQuery becoming a widely supported standard for XML querying, reasoning capabilities for XQuery could help to transfer some of the benefits from Xcerpt. This aim can either be achieved by forward chaining or by backward chaining. The foundations for the first method are provided by this thesis, and forward chaining evaluation of Xcerpt programs is sketched in the previous Section.

Another promising approach is the implementation of simulation unification in XQuery to allow reasoning by backward chaining. The difference between translation of Xcerpt to XQuery and a reasoning module is that the latter would operate on Xcerpt rules and the data at the same time. This means that some kind of representation of Xcerpt terms as XML data is needed. The XML syntax of Xcerpt can be used for this purpose. In a similar way to this thesis, the implementation of such a reasoning module could be structured into two parts: One to cover the calculation of substitution sets by simulation unification, and the second part to support rule chaining. Since XQuery allows to operate on schema validated data, optimizations based on this schema information could be additionally applied.

6.1.3 Efficient Evaluation of n-ary Queries in XQuery

XPath allows the formulation of tree queries, but it does not allow to simultaneously locate more than one node of an input tree. In other words, XPath only allows unary queries. To formulate n-ary queries, XQuery has to be used. The sublanguages XQ_2 and XQ_3 comprehend the translations of n-ary Xcerpt queries in form of iterations, existential quantifications, several kinds of constraints and parent-child relationships. There are, however, many different alternatives to formulate n-ary queries. Since the emphasis of this thesis lies upon finding translation possiblities and algorithms, the performance is not a primary issue. Interesting research in this area would be to empirically compare the efficiency of different XQuery realizations of n-ary queries and the evaluation of n-ary queries in Xcerpt. Since the performance is expected to vary among XQuery implementations, multiple XQuery engines should be incorporated in the analysis.

6.2 Conclusion

The contributions of this thesis are the following: Three pairs of equally expressive sublanguages of Xcerpt and XQuery are identified and automatic translation algorithms for translating in both directions between each pair of sublanguages are discussed. The formal semantics of both XQuery and Xcerpt are thoroughly compared to underline the correctness of the translation rules. In order to properly convey the semantics of Xcerpt query terms, the translation algorithms for XC_2 and XC_3 must distinguish between two kind of nodes. The first type of nodes constitutes bindings for Xcerpt variables to be returned in substitution sets, and must therefore be translated by for-clauses. The second type of nodes represent – just as predicates in XPath – mere constraints for the data to be retrieved and must therefore be translated by existential quantifications (some-clauses). A recursive translation algorithm has been presented that translates XC_2 query terms in one single sweep over the term structure. XC_3 including negated subterms and multiple variable occurrences, the order of translating the subterms to XQuery is non-trivial. Translation rules for XC_3 specify how to recursively translate entire queries in a single sweep and how to ensure the correct translation order of the subterms. Translation rules for the reverse direction identify tree patterns within XQuery in order to translate them to Xcerpt query terms.

Building upon the translation of Xcerpt queries, it has been shown how to translate Xcerpt construct terms directly and with XQuery functions. Automatic translation algorithms for construct terms and entire construct-query-rules are given. It has been demonstrated that intertwined XQuery construction and Xcerpt grouping constructs serve similar purposes in both languages. Intertwined XQuery expressions are automatically translated to Xcerpt rules. As a final enhancement, XQuery expressions constructing intermediate results are translated by the simulation of dynamic environments through rule chaining and by the elimination of let-clauses.

Although only parts of both languages are considered, it can be seen that translation in both directions is feasible under moderate expenses. Difficulties are encountered when confronted with the distinguishing characteristics of both languages, such as node-identity in XQuery, injectivity among siblings and optional subterms in Xcerpt.

Bibliography

[1] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C Recommendation, 2005.

[2] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation, 2005.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). Extensible Markup Language (XML) 1.0 (2nd Edition). W3C Recommendation, 2000.

[4] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of the Intl. Conf. on Logic Programming (ICLP02)*, LNCS 2401, Copenhagen/Denmark, 2002. Springer-Verlag.

[5] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems (IJSWIS)*, 1(2), 2005.

[6] François Bry, Tim Furche, and Sebastian Schaffert. Initial Specification of the Language Syntax, REWERSE Deliverable I4-D6, 2005.

[7] Don Chamberlin and Jonathan Robie. XQuery Update Facility Requirements, W3C Working Draft, 2005.

[8] Mike Champion, Steve Byrne, and Lauren Wood. Document Object Model (Core) Level 1. W3C Recommendation, 1998.

[9] James Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 1999.

[10] John Cowan and Richard Tobin. XML Information Set (Second Edition), W3C Recommendation, 2005.

[11] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Candidate Recommendation, 2005.

[12] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model, W3C Candidate Recommendation, 2005.

[13] Mary Fernández, Jérôme Siméon, and Philip Wadler. A Semi-monad for Semi-structured Data. *Proceedings of International Conference on Database Theory (ICDT), London, UK, Jan. 2001*.

[14] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Candidate Recommendation, 2005.

[15] Philippe Michiels. Xquery optimization. *Proceedings of the VLDB 2003 PhD Workshop, Co-located with the 29th International Conference on Very Large Data Bases. Berlin, September 12-13*, 2003.

[16] Dan Olteanu, Holger Meuss, Tim Furche, and Francois Bry. XPath: Looking Forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490, pages 109–127. Springer, 2002.

[17] Sebastian Schaffert. Xcerpt: A Rule-Based Query and Transformation Language for the Web, PhD Thesis, Institute for Informatics, University of Munich. 2004.

[18] Andreas Schroeder. An Efficient Evaluation of Xcerpt, Diploma Thesis, Institute for Informatics, University of Munich, 2005.