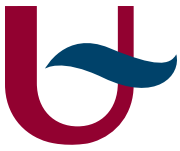


Universiteit Antwerpen

Faculteit Wetenschappen

Informatica

XML Transformations, Views and Updates based on XQuery Fragments

Proefschrift voorgelegd tot het behalen van de graad van doctor in de wetenschappen aan de Universiteit Antwerpen, te verdedigen door

Roel VERCAMMEN

Promotor: Prof. Dr. Jan Paredaens
Co-promotor: Dr. Ir. Jan Hidders

Antwerpen, 2008

XML Transformations, Views and Updates based on XQuery Fragments

Roel Vercammen
Universiteit Antwerpen, 2008

`http://www.universiteitantwerpen.be`

Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). – *Onderzoek gefinancierd met een specialisatiebeurs van het Instituut voor de Aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen (IWT-Vlaanderen).*
Grant number / *Beurs nummer*: 33581.
`http://www.iwt.be`

Typesetting by LaTeX

Acknowledgements

This thesis is the result of the contributions of many friends and colleagues to whom I would like to say "thank you". First and foremost, I want to thank my advisor Jan Paredaens, who gave me the opportunity to become a researcher and teached me how good research should be performed. I had the honor to write several papers in collaboration with him and will always remember the discussions and his interesting views on research, politics and gastronomy.

I am greatly indebted to my co-advisor, Jan Hidders, who helped me assessing what good research questions are and whose expertise helped me a lot. Moreover, I will also remember him for non-scientific reasons, such as the badminton games and his fascination and broad knowledge of history.

Philippe Michiels deserves a special thank you for the many vivid discussions we had and the interesting research questions on XQuery optimization he came up with and that we tried to tackle together. Even though the research we did together is not reflected in this thesis, it broadened my view on the field of database research and he brought me in touch with many great researchers, such as Mary Fernández, Jérôme Siméon and Maurice van Keulen. I am also very thankful to the rest of the ADReM team for giving me a nice stimulating environment the past four years.

I want to acknowledge Toon Calders, Serge Demeyer, Bart Goethals and Maarten Marx, members of my PhD Jury, for giving feed-back on this thesis, which helped to improve this thesis. I also want to thank Stefania Marrara, who was in Antwerp for six months and with whom I had the honor to collaborate on parts of the research that resulted in Chapter 3.

My family deserves my gratitude as well, for the continuous support and encouragement they gave me and the nice and fun environment which I am privileged to call home.

Last but not least, I want to say thank you to the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) for making it possible for me to do the research that resulted in this thesis. Not only their financial support, but also the frequent reporting and the interesting project interviews have helped me a lot into the direction of this thesis.

Abstract

During the past decade, XML has become ubiquitous when it comes to sharing data and publishing information on the Web. Because of the ever increasing amount of data available in XML format and the data model which differs from previous work in database research, this new standard has drawn the attention of the database community.

Several query and updating languages have been proposed, which eventually led to the W3C recommendation for XQuery 1.0 and update facilities for this query language. We define a formal framework called LiXQuery to study and investigate properties of this language and its constructs. More precisely, we compare the relative expressive power of some of the most important features of XQuery in terms of performing transformations.

Data integration and sharing is an interesting problem which can benefit from XML research. XML views can be defined to transform data that several parties want to share to a common schema. It is desirable that such virtual XML trees behave as much as possible as ordinary XML documents. We introduce the concept of projection views based on XPath expressions and show how we can translate XPath queries on the view documents to XPath queries on the base document. Moreover, we introduce three simple XPath-based update operations, i.e., insertion, attribute update and deletion, and show that it is decidable whether propagating the primitive update operations to the base tree introduces view side-effects or not.

Contents

List of Figures

CHAPTER 1

Introduction

Sharing data has been one of the main reasons why the Web has become so popular the past decade. The HyperText Markup Language (HTML) and graphical browsers have been killer applications that caused a tremendous increase of Internet connections during the nineties. One of the major challenges posed by this evolution is managing all this available information. In this thesis we investigate one of the many facets of integrating and managing the loosely structured data on the Web. This introduction highlights the topics that are studied in this thesis. First, we briefly discuss in Section 1.1 the use of the eXtensible Markup Language (XML) to represent data on the Web. Then, we talk about querying and updating XML documents in Section 1.2 and Section 1.3. In Section 1.4 we introduce the notion of XML Views and point out some of the problems that this raises with respect to performing updates. Finally, we discuss our contributions and sketch the organisation of this dissertation in Section 1.5.

1.1 XML

The idea of using text-based formats to store, structure and publish information is almost as old as computer science. However, binary storage formats implementing advanced data structures were far more popular for most applications, mainly for performance and efficiency reasons. Nevertheless, people understood the need for interchange formats that still needed to be readable for decades already early on, for example in big governemental projects. These formats were required to be readable by both machines and humans, and eventually this need resulted in the Standard Generalized Markup Language (SGML) in the 1980s. SGML is a metalanguage in which markup languages can be defined. Such a language could be defined by means of a Document Type Definition (DTD). HTML, one of the applications of SGML, revolutionized the Internet in the early 1990s, by defining a way

to graphically represent data sent over the internet. However, many critisisms on SGML as a data-interchange format remained. The main concern was that SGML was too complicated, which resulted in the emergence of a simplified version of SGML, called eXtensible Markup Language (XML). Nowadays, many web applications use XML-based formats for publishing information. For example, consider Rich Site Summary (RSS), Scaling Vector Graphics (SVG) and the Extensible HyperText Markup Language (XHTML).

Almost at the same time as the emergence of the Web, the notion of semistructured data[Buneman, 1997, Abiteboul et al., 1999, Quass et al., 1996, Suciu, 1998] started to get attention in the database community. Semistructured data is assumed to be self-describing since there is often no seperate schema and the data is represented in a graph-like or tree-like structure. Some of the main reasons why semistructured data started to get attention was the way data was represented on the Web and the challenge of data-integration. Several models for semistructured data were proposed, but eventually the community shifted towards the new XML standard. The XML Data Model shows some differences with some of the previous semistructured data models in the sense that data is represented in ordered trees and mixed contents is allowed, i.e., elements can contain both elements and text at the same time.

The XML data model defines seven kinds of nodes, but in this dissertation we only consider four different node kinds, i.e., we ignore processing instruction, namespace and comment nodes and only consider element, attribute, text and document nodes. An XML tree rooted with a document node is called an XML document and document nodes can only occur as a root in the XML data model. Every document node has exactly one child node, which is an element node and is called the root element. Element nodes have a name and can contain an arbitrary number of attribute, element and text nodes. Attribute nodes also have a name, but can only contain one value and their name is unique among the attribute nodes within the same element. Text nodes only contain a value. Some additional constraints are added to the XML data model, e.g. two text nodes cannot be adjacent sibling in an XML tree, but we refer the reader to textbooks such as [Box et al., 2000] for more details on these. When all these constraints are satisfied, an XML document is said to be *well-formed*. From now on, whenever we talk about XML documents, we implicitly assume it is well-formed.

Within the context of an XML processor, every node of the instance is given an identifier. This identifier is only assigned to XML nodes when it is first needed and gives every node its node identity, which is similar to object identity in object-oriented programming languages.

Every instance of the data model has a text representation, which is called *serialization*. An example of this is shown in Figure 1.1. In the text representation, elements are surrounded by an opening and a closing tag and the name of the element can be found between angle brackets. For example `<ta>S. Lebowsky</ta>` is an element with name `ta` and one text node with value `"S. Lebowsky"`. Attribute names and values appear within the opening tag of an element, e.g., the `course` element with opening tag `<course name="Database Theory">` has a `name` attribute with value `"Database Theory"`. Other features that are shown in this example are comments (written between `<!--` and `-->`),

entities (e.g., `>` used to represent `>` to avoid parsing problems) and abbreviations for empty elements (e.g, `<section title="Further reading"/>` is a shorthand for `<section title="Further reading"></section>`. Note that the tag names and attribute names are used to make the data model self-describing.

Notwithstanding the fact that semistructured data in general and XML in particular do not need any schema information, it makes sense to define a schema for certain applications. For example, when XML is used as an intermediary format for communication between applications, then both parties can agree upon constraints to which the data has to obey. Therefore, similar to SGML, several schema languages exist for XML, such as DTD, XML Schema and RelaxNG.

1.2 Querying XML Documents

The ability to retrieve information effectively through queries is essential for all kinds of data, and hence also for data stored in XML documents. Therefore, it should be no surprise that already during the late nineties, when the XML standard came about, many query languages for XML and semistructured data were proposed, e.g., Lorel [Abiteboul et al., 1997], UnQL [Buneman et al., 2000], XQL [Ishikawa et al., 1998], XML-QL [Deutsch et al., 1998], Quilt [Chamberlin et al., 2000], and YaTL [Cluet and Siméon, 2000]. Eventually, these languages and proposals led to several W3C recommendations, such as XPath, XSL and XQuery. In this dissertation we study the query language XQuery, which became recently a recommendation [Boag et al., 2007a], and its sublanguage XPath, which is used in other XML languages as well to select nodes and atomic values from XML data. Whenever we refer to XPath, we refer to XPath 2.0 [Berglund et al., 2007] and not to XPath 1.0 [Clark and DeRose, 1999]. Note that the latter is less expressive than the former, more precisely, XPath 1.0 is not expressively complete w.r.t. first order logic, while XPath 2.0 is [Marx and de Rijke, 2005].

The results of both XQuery and XPath expressions are sequences defined by the common data model [Fernández et al., 2007], i.e., sequences of atomic values such as strings, integers and booleans, and references to nodes in an XML document. This suggests that node identity is important. However, in practice the result of a query is often serialized [Boag et al., 2007b] and hence we lose the information on node identity.

The main construct in XPath is a path expression. A path expression is used for navigation through an XML document. The navigation is performed in several steps, seperated by a slash. Each step contains an axis, expressing a direction, and a node test, used for testing the name of a node or the node kind, which are seperated by a double colon. Moreover, predicates can be added between square brackets to test certain conditions and in XPath 2.0 variables and first order quantifiers were added. Finally, XPath expressions can be abbreviated, for example if only a label test `title` is specified in a step, then the child axis is assumed to be followed, i.e., `child::title`, a double slash is used as an abbreviation for `/descendant-or-self::node()/`, and `@`n is an abbreviation for `attribute::`n. For a detailed discussion on XPath and XML, we refer the reader to [Kay, 2004]. We now give a

```
<?xml version="1.0"?>
<courses>
  <course name="Database Theory">
    <instructors>
      <professor>D. Woods</professor>
      <ta>S. Lebowsky</ta>
    </instructors>
    <revisions>
      <revision>
      <section title="Introduction">This course is about database theory.</section>
      <section>You can write queries in SQL.
        <exercise>Write a query to select all persons that visit a bar.
          <solution>SELECT person FROM visits</solution> </exercise>
        A JOIN clause combines records from two tables and results in a new
        (temporary) table, also called a joined table. </section>
      </revision>
    </revisions>
  </course>
  <course name="Algebra">
    <instructors>
      <lecturer>E. Edwards</lecturer> <ta>A. Adams</ta> <ta>F. Murrow</ta>
    </instructors>
    <revisions>
      <revision>
        This course is about algebra. For more details, check your textbooks.
        <!-- Maybe we should at least try to make it look more like a real course -->
      </revision>
      <revision>
        <section title="Introduction">
          Welcome to Algebra 101. <!-- This revision is much better than the
                                       previous attempt -->
          <section title="Course material">Check the textbook
            Algebra &gt; Elementary Algebra</section>
          <section title="Further reading"/>
        </section>
      </revision>
    </revisions>
  </course>
</courses>
```

Figure 1.1: Example of an XML document for a virtual learning environment.

simple example based on the XML document of Figure 1.1 to illustrate how we can select all sections that contain exercises with no solution:

```
/descendant::section[child::exercise[fn:not(solution)]]
```

The XQuery language extends upon XPath and is a full-fledged XML query language. Arguably the most important construct in XQuery is the FLWOR expression, which somewhat resembles the select-from-where expressions in SQL. FLWOR is an abbreviation of five XQuery keywords, i.e., `for`, `let`, `where`, `order by`, and `return`. Note that the `for` keyword suggests some kind of iteration, but since XQuery is side-effect free, the `return` clause can actually be evaluated for each item in the result of the `for` clause in parallel and afterwards be merged together according to the `order by` clause to give the correct result sequence. Other features in XQuery include node construction, typeswitches, (recursive) function definitions, arithmetic and many built-in functions. Moreover, XQuery implementations may provide facilities where external functions are implemented using a host programming language. For a more detailed discussion on XQuery, we refer the reader to either textbooks, such as [Brundage, 2004, Katz et al., 2003], or to Chapter 2, where the non-updating part of the language LiXQuery, which is introduced there, is compatible with XQuery while containing most typical XQuery constructs. The following expression is an illustration of an XQuery query that returns all teaching assistants and for each assistant shows which courses they support, assuming there are no teaching assistants with the same name.

```
for $ta in fn:distinct-values(//ta/text())
return
  element {"ta"} {
    attribute {"name"} {$ta},
      for $course in //course
    where $course/instructors/ta/text() = $ta
    return element {"course"} {$course/@name }
  }
```

In this example, the function `fn:distinct-values` takes a sequence of text nodes as input, and returns a new sequence such that the set of values of all nodes in the input sequence is the same as the corresponding set of values for the output sequence. The order of the output sequence is implementation dependant, which enables some optimizations and introduces some degree of non-determinism into the query language. The path expression `$course/name` selects an attribute node, which is not converted to an atomic value, but is inserted as an attribute in the element that is being constructed. The serialized result for the previous query when applied to the document in Figure 1.1, can be the following:

<**ta** name="*A. Adams*"><**course** name="*Algebra*"></**ta**>,
<**ta** name="*S. Lebowsky*"><**course** name="*Database Theory*"></**ta**>,
<**ta** name="*F. Murrow*"><**course** name="*Algebra*"></**ta**>

Note that the result of an XPath and XQuery expression does not have to be well-formed XML and can even contain atomic values.

1.3 Updating XML Documents

XML updates are operations that can make persistent changes to instances of the XML data model. While query languages for XML documents and semistructured data were ubiquitous from the start, work on a W3C XML Update recommendation started rather late. This is also shown by the fact that the first public working draft on XML Query requirements [Fankhauser et al., 2000] was published half a decade earlier than the first public working draft on the XQuery Update Facility requirements [Chamberlin and Robie, 2005]. Two possible reasons for why this took a while are the following:

XML used as interchange language Originally XML was mostly used for interchanging data between applications. Users would then parse these XML fragments and update their relational database accordingly if needed.

Updates done using DOM in a host programming language When XML was used for storing data rather than for interchange between two parties, there already was the ability to update XML documents by using Document Object Model (DOM) operations. In the DOM level 2 specification users were able to modify XML documents and in DOM level 3 it became possible to save XML documents in order to make the changes persistent. The DOM operations needed for performing updates can be accessed by a programming language in which the application is written.

However, XML is increasingly supported by both native XML databases and extensions of relational databases and therefore the need for update languages increased. As a consequence, several XML update languages have been proposed [Laux and Martin, 2000, Sur et al., 2004, Tatarinov et al., 2001]. Many of those are extending XQuery, based upon the statement from the XML Query Requirement which states that "Version 1.0 of the XML Query Language MUST not preclude the ability to add update capabilities in future versions.". The update language that we investigate in this dissertation is based on XQuery and influenced by a number of these XQuery extensions. We now briefly discuss three proposals which influenced the update language used in this thesis.

UpdateX [Sur et al., 2004] includes FLWUpdate statements for complex updates and snapshot semantics to enforce consistency of these complex updates. Intuitively, snapshot semantics means that a snapshot of the XML store is being made before the evaluation of the expression and the resulting updates are not yet performed, but instead they are added to a list of pending updates. After the evaluation of the top-level expression, this list of pending updates is applied in the order that the updates are generated. Several optimizations have been investigated concerning the order of applying the list of pending updates and whether certain pending updates

can be applied as soon as they are generated without changing the result of the query [Benedikt et al., 2005b, Benedikt et al., 2005a].

XQuery Update Facility [Chamberlin et al., 2007] is the W3C proposal to extend XQuery with update operations. Expressions can result in either a result sequence or a list of pending updates. The list of pending updates is applied in an order which does not depend on an execution order of the expression, but does depend on the kind of the primitive update operations. The update operations include a copy, an insertion, an edge deletion[1], renaming and changing values of nodes. Moreover, a `transform` operation is added, which creates a modified copy of existing nodes and is technically speaking not an update operation. This `transform` operation can be evaluated efficiently, as is shown in [Fan et al., 2007a].

To illustrate this language, we now give a small example of an XQuery expression that uses the update facility to remove "old revisions" from the document in Figure 1.1:

```
delete (
  for $rev in //revisions
  where not(empty($rev/following-sibling::revision))
  return $rev
)
```

The `delete` expression takes a sequence of nodes and removes the incoming parent-child edges from these nodes, i.e., the input nodes and its descendants are moved out of the tree.

XQuery! (read: "XQuery Bang") [Ghelli et al., 2006] introduces side-effects to enhance programming XML applications in XQuery. They introduce a `snap` operation to control the applications of updates and enable optimizations. More precisely, to evaluate `snap` { e}, we first evaluate e and then the list of pending updates that is generated by e is applied, either in the order it is generated or in an arbitrary order (which enables certain optimizations). Moreover, XQuery! expressions can return both a value and a list of pending updates. We illustrate this language with an example that adds a (nested) section number to each section.

```
for $sec in //section
let $sibsec := $sec/preceding-sibling::section
let $parsec := $sec/parent::section
return
  snap {
    insert {
      if ($parsec) then
         attribute {"secnr"}
              { concat(concat($parsec/@secnr, count($sibsec)+1), ".") }
```

[1] An edge deletion moves a subtree T_1, rooted at a node n within a tree T_2, out of T_2 such that n becomes a root node now.

```
      else
         attribute {"secnr"} {concat(count($sibsec)+1, ".") }
    }
    into $sec
}
```

Note that the `for` expression iterates over the `section` nodes in document order. Since there is a `snap` expression in the body of the `for` expression, we know that the previous `section` nodes in document order are already updated when evaluating the body of the loop for the following `section`.

Remarkably, while node identity can convey information according to the XQuery specification, none of these update languages support moving a node to another place while preserving its identity. From now on, whenever we discuss the XQuery Update Facility in this dissertation, we consider the XQuery Update Facility Working Draft of July 2006 [Chamberlin et al., 2006]. The current version of XQuery Update Facility is a candidate recommendation and there are some minor differences between the revision we discuss and the current revision.

Recently, XQueryU [Ghelli et al., 2007], an XQuery-based language with side-effects has been proposed. This work illustrates a semantics that differs from the other XQuery-based update languages in the way that the semantics is not defined by means of normalization where FLWOR expressions are mapped to (possibly nested) simple `for` expressions, but instead they define a tuple-stream semantics, which is better suited for database compilation. They also show how the nested-for and the tuple-stream semantics for FLWOR expressions is different in the presence of side-effects.

1.4 Updating XML Views

Views are an important feature of database systems to provide abstraction and guarantee a certain level of security. Moreover, views are often used for interchanging data between applications. Intuitively, views are derived instances of a data model which behave as much as possible as non-derived instances of this data model. This means that we should be able to a certain extent to query and update views. For the relational model, Codd initially stated in one of his fameous twelve rules [Codd, 1985] that all views that are theoretically updatable should also be updatable by the system. However, as was pointed out by [Buff, 1988], this is in general not possible.

In this section we first take a look at what views are in the context of the XML data model and how they can be used as an access control mechanism, then we look deeper into how the view-update problem can be defined and finally we discuss some related work.

1.4.1 XML Views

The notion of a view is a very fundamental service to be provided by a database system, however, there is no standard for supporting views on XML databases yet. The first work

on XML views was done by Abiteboul in [Abiteboul, 1999], where a possible view architecture is introduced and updates of XML views are discussed for the very first time. At the moment there is not yet a generally accepted view definition language for defining virtual XML documents. Still, in some sense, it is already possible to define virtual XML documents by transforming existing XML documents through a query/transformation language like XSL or XQuery. We now illustrate how such a view can be created using an XQuery-like syntax and the `transform` expression to indicate that this construct can be useful for this purpose. Suppose the document in Figure 1.1 can be accesed through the URI `courses.xml`. We define the virtual document `myCourses.xml` as follows[2]:

```
declare view { "myCourses.xml" } {
  element {"myCourses"} {
    for $c in (
      transform
        copy $cc := doc("courses.xml")/courses
        modify
          delete ($cc//comment(),
                  $cc//solution)
        return $cc/course
    ) return
        transform
          copy $cc := $c
          modify (
            (insert $cc/revision[last()]/* after $cc/instructors),
            (delete $cc/revision)
          )
          return $cc
  }
}
```

The result for the given input is shown in Figure 1.2. As is illustrated, the `transform` expression is a useful construct when defining views and solves most problems mentioned in [Vercammen, 2005]. However, some problems remain. For example, we needed two `transform` expressions, since we had to move the contents of the final `revision` element up and remove the comments and `solution` descendants. Moreover, the full XQuery language is still too complex to handle as a view definition language, especially for updates through views.

1.4.2 Views as Access Control Mechanism

The need of XML views is partially motivated by their ability to provide security mechanisms, i.e., the user is allowed to access and possibly update the data that he can see. The concept of XML views as security views was first introduced in [Stoica and Farkas, 2002] which studies the problem of generating secure partials views, which are free of semantic

[2]We use our own notation here, since there is no standard way of defining views yet.

```
<?xml version="1.0"?>
<myCourses>
  <course name="Database Theory">
    <instructors>
      <professor>D. Woods</professor> <ta>S. Lebowsky</ta>
    </instructors>
    <section title="Introduction">This course is about database theory.</section>
    <section>You can write queries in SQL.
      <exercise>Write a query to select all persons that visit a bar.</exercise>
      A JOIN clause combines records from two tables and results in a new
      (temporary) table, also called a joined table. </section>
  </course>
  <course name="Algebra">
    <instructors>
      <lecturer>E. Edwards</lecturer>
      <ta>A. Adams</ta> <ta>F. Murrow</ta>
    </instructors>
    <section title="Introduction">
      Welcome to Algebra 101.
      <section title="Course material">Check the textbook
        Algebra &gt; Elementary Algebra</section>
      <section title="Further reading"/>
    </section>
  </course>
</myCourses>
```

Figure 1.2: Example of an XML view for the document for Figure 1.1.

conflicts. An efficient algorithm for deriving security view definitions from security policies is proposed in [Fan et al., 2004]. The concept of security views used in the two previously worked research papers is generalized in [Kuper et al., 2005]. A special kind of subtrees is proposed in [Benedikt and Fundulaki, 2005] where the view contains root-to-leaf paths from the original document, inheriting the tree structure. The root-to-leaf paths that are in the view are determined by the result of XPath expressions and for natural fragments [3] of XPath they show that the resulting subtree query langages are closed under composition. Finally, [Fan et al., 2007b] studies the rewriting of regular XPath queries on XML views defined by an annotated schema.

Views are not the only security mechanism available. Several papers cover other access control mechanisms for XML documents. For example, [Fundulaki and Marx, 2004] shows that the semantics of access control policies can be specified by XPath expressions and use their framework to give a formal specification of five prominent approaches of access control for XML document. In [Böttcher and Steinmetz, 2005] two query execution plans for users with different access on a single master XML document are made and the appropriate plan is chosen by the access control module. The first plan makes a secure copy of the part of the XML document that can be accessed and then performs the query, while the second modifies the query of the user in such a way that every location step checks for which of the result nodes the user has access and then simplifies this query. Finally, another approach is the one presented in [Bertino and Ferrari, 2002], where the XML document is accessible to all users, but different encryption keys are used for different portions of the document and they are distributed selectively to the users according to the access control policies.

1.4.3 Updating Views

The problem of updating databases through views was first reported in 1974 by Codd [Codd, 1974] and has been widely studied ever since. Views are often used as a security mechanism or just for the convenience of the user who knows the base schema, but updates through views are often impossible or ambiguous. Even when it is possible to uniquely identify which updates have to be performed on the original data, updating instances of the base schema can have undesired side effects. In order to eliminate side effects, [Bancilhon and Spyratos, 1981] introduces the notion of constant-complement view updates, where each update is reversible and does not have any effect on the "complement" of the view. Hegner [Hegner, 1990] refines this notion and introduces the notion of open and closed update strategies. In *open view update strategies* it is assumed that the view is only defined for convenience. In this case, an update strategy allows as many updates as possible and the user is expected to be aware of consequences that updates can have. On the other hand, *closed views* are totally encapsulated and it is assumed that the user does not have any knowledge of the base schema. Hence the look and feel of the view should be the same as that of a base schema and the effects of the updates should be limited to that part of the base schema which is reflected in the view.

[3] We refer to sublanguages of query languages, such as XPath or XQuery, as fragments of these languages.

Most work on view updatability in the relational model has concentrated on open view update strategies because closed view update strategies are often found too conservative. However, closed view update strategies are a very interesting subject of study, since they represent the class of updates that users can fully understand [Lechtenbörger, 2003] and they offer more security by limiting the scope of the effects of user updates to that part of the database that can be viewed by these users. Furthermore, closed views are especially interesting in an XML setting, where one can publish XML views on the web, which can be updated by others and where security becomes a much more important issue.

Intuitively, in closed view update strategies all updates of the view can be undone by other updates of the view, i.e., the initial database state of the base schema can be recovered, and updates can be applied sequentially or all at once. More precisely, closed view update strategies have to obey the following rules:

- *Reflexivity.* The identity view update is allowable and corresponds to an identity update on the base schema state.
- *Symmetry.* Every view update is globally reversible.
- *Transitivity.* A series of view updates may be applied incrementally or all at once.

All these conditions have to hold if the updates that occur in them are allowable. Open view update strategies are more liberal in the sense that they allow updates to have some side effects. In what follows we discuss both closed and open update strategies.

1.4.4 Existing Approaches in RDBMS and OODBMS

One of the first efforts to systematically address the view update problem have been made by Dayal and Bernstein. They explored the notion of correct translatability of view updates in [Dayal and Bernstein, 1978]. In this work they required the view update to have a unique set of corresponding updates on the base schema.

Bancilhon and Spyratos introduced the constant-complement approach [Bancilhon and Spyratos, 1981]. A complement of a view is another view, such that together they form a lossless decomposition, i.e., the state of the base schema can be recovered from the combined states of both views. Every complement of a view defines an update strategy. Furthermore, it is shown in [Bancilhon and Spyratos, 1981] that every closed update strategy is defined by a constant complement update strategy. However, it is known that there exist complements that do not define closed update strategies. Cosmadakis and Papadimitriou show that finding a minimal complement of a given view is NP-complete [Cosmadakis and Papadimitriou, 1984]. Lechtenbörger and Vossen demonstrate how to compute reasonable small complements for relational views that only contain projection, selection, renamings and joins [Lechtenbörger and Vossen, 2003]. For some special cases these complements are even shown to be minimal. Hegner presents an order-based approach to find a unique "natural" complement which defines the only reasonable update strategy [Hegner, 2004]. He assumes a partial order on the state of states of the database schema. Many database

transformations preserve this order structure. For example, a natural order structure in the relational model is a relation-by-relation inclusion for which the only base operation of the relational algebra that is not monotonic is the difference. Furthermore, the notion of closed update strategies is extended by adding three extra order-related conditions.

Gottlob et al. relaxed the requirement of a constant complement to capture the entire class of views where the effect of updates on the base schema is unambiguously determined when the effect of an update on the view schema is determined [Gottlob et al., 1988]. This class of "consistent views" is obtained by allowing the complement of a view to decrease according to a suitable partial order. Masunaga presented an approach to design view update translators for relational databases [Masunaga, 1984]. In his approach, views are presented as trees where the base relations are leaves, the root is the view itself and intermediate nodes represent relational algebra operations and rules are introduced to process updates on the view to the lower levels of the trees. Furthermore, it is suggested that end-users should be involved in resolving semantic ambiguities. Keller developed another translation mechanism of view updates, where he considers select-project-join views on BCNF relations [Keller, 1985] and in order to eliminate anomalies, update translations have to be valid and satisfy five criteria. This method gives a complete list of alternative (acceptable) translations for the considered view updates. Tomasic showed that the number of translations for a view update is exponential in the size of the database and that this number can be reduced by using annotations [Tomasic, 1988]. Finally, Buneman et al. investigated the problems of finding a minimal set of tuples T in the source database to delete a tuple t from the view and minimize the side-effects on the view, and the source respectively [Buneman et al., 2002] and they show a dichotomy in the complexity: the problem is either in P or is NP-hard, for queries in the same class.

The view update problem has also been studied for object-oriented databases. For example, [Scholl et al., 1991] shows that the view update problem for object-oriented databases is much more feasible than in the relational model. This conclusion is based on the object preserving operator semantics of the query language they use. In this setting, objects in the view are objects of the database, and hence updating the view is directly an update of the objects of the database. However, these techniques are not applicable to an XML setting if one prefers views to have the same look and feel as normal data, since in this case views should behave like trees.

1.4.5 Existing Work on XML View Updates

In [Kozankiewicz et al., 2003] a stack-based approach is used to create XML views that can be updated. This is made possible by including imperative statements in the view definitions, which have to explicitly state the intents of updates. This approach is very liberal, but puts a high responsibility on the shoulders of the person who defines the view. Moreover, their framework does not use any of the current XML standards.

Foster et al. [Foster et al., 2005] propose a language to define bi-directional tree transformations. This language uses the notion of lenses consisting of two partial functions: the *get* function, which defines the view, and the *putback* function, which is an update

strategy. In order for a lens to be well-behaved, it has to obey the reflexivity and the symmetry requirements of a closed view update strategy, but the transitivity requirement is dropped. Similar work for the relational model has been presented in [Bohannon et al., 2006].

In addition to updating pure XML and relational views, there has also been some work on XML views on relational databases. Braganholo et al. study how to use the view update results of the relational model to assess updatability of XML views over relational databases [Braganholo et al., 2004, Braganholo et al., 2006]. They define XML views by means of query trees, i.e., templates containing information on the values that have to be included. A relational view can be associated with these query trees, and updates on the XML view are translated to updates on the relational view. Hence the problem of updating XML views on relational data is converted to that of updating relational views. Similarly, Wang et al. present a technique, called U-Filter [Wang et al., 2006], to check in advance whether updates of XML views over relational data have view side effects.

1.5 Contributions and Organization

To conclude, we first summarize the problems that are (partially) solved in this thesis.

- There is a lack of a complete and formal semantics for XQuery-based update languages. Moreover, the semantics of these languages is usually too complicated to be useful for a theoretical study and many features are only added for practical purposes.
- The impact in terms of expressive power of several XQuery features is not clear. Properties for more expressive languages are usually harder to establish.
- In some cases, users can see a difference in behavior of view documents and other XML documents. However, for some applications this is not desirable and therefore we should only allow updates for which the result of updating the view document immediately is the same as the result of recomputing the view for the updated base document, but it is in general not clear in advance whether this is the case or not.

We now give the main contributions of this dissertation.

- The LiXQuery language is defined as an XQuery-based query and update language. We give a complete formal definition of the language, together with some illustrating examples in Chapter 2. The definition of this language enables us to study its properties. Parts of this chapter are published in [Hidders et al., 2004, Hidders et al., 2005b, Hidders et al., 2006].
- The relative expressive power of some LiXQuery constructs is studied in Chapter 3 in terms of transforming XML documents and variable bindings to a serialized result. The results in this chapter are a combination of previously published results in [Hidders et al., 2005a, Page et al., 2005, Hidders et al., 2006, Hidders et al., 2007a]

- A specific subproblem of the XML view update problem is studied in Chapter 4. More precisely, we define a simple XPath-based language to define XML queries, views and updates, and show that it is decidable in this setting whether propagating the updates directly to the source database causes view side effects. The motivation for studying this subproblem is as follows:
  - Most existing work studies XML views on relational data, we consider XML views on XML data.
  - We want to avoid that specifications of the view with the same view semantics can have other update semantics, i.e., if for every document two view definitions always show the same view document then the updates on both views have to be translated to the same updates on the base trees. Lenses do not guarantee this.
  - We apply the most straightforward update strategy, similar to what was done in object-oriented databases, i.e., we use the node identity to propagate updates to the base instance.

  Most of the results of this chapter are not yet published, however the related publications, on which this chapter is based, are [Vercammen, 2005, Vercammen et al., 2006].

The discussion in Chapter 5, summarizes the results of this thesis and shows how we can combine, extend and generalize the results of this dissertation towards further research directions.

CHAPTER 2

A Formal Model for XQuery

Notwithstanding the fact that the W3C recommendation for XQuery 1.0 contains a document called "XQuery 1.0 and XPath 2.0 Formal Semantics" [Draper et al., 2006] and that the W3C working draft on the "XQuery Update Facility" [Chamberlin et al., 2006] contains an elaborate description of the update semantics, we introduce our own formalism for an XQuery-based query and update language to facilitate studying the properties. In this model we make some simplifications, for example, we ignore static typing. Moreover, the sublanguage we consider is chosen in such a way that many of the typical XQuery constructs are still there, but we try to avoid having a bloated language. Both restrictions allow us to give a full formal semantics for our XQuery sublanguage in this chapter. The resulting language is refered to as LiXQuery.

The syntax and the informal semantics of this language will be given in Section 2.1 and this will be illustrated with some examples in Section 2.2. In Section 2.3. we define our framework and in Section 2.4 we use that framework to give the formal semantics.

2.1 Syntax and Informal Semantics

The syntax of LiXQuery expressions is given by the grammar depicted in Figure 2.1. This syntax is an abstract syntax, i.e., it assumes that extra brackets and precedence rules are added for disambiguation. The syntax includes some built-in functions, function calls for user-defined functions and a function definition statement. Note that the set of built-in functions is fairly restricted, but these functions together with the ability to define recursive functions, suffice to express all possible computations on the atomic values we consider. The non-terminal $\langle Name \rangle$ refers to the set of names $\mathcal{N}$ which we will not describe in detail here except that the names are strings that must start with a letter or "_". The non-terminal $\langle String \rangle$ refers to strings that are enclosed in double quotes such as in `"abc"` and

⟨*Integer*⟩ refers to integers without quotes such as `100`, `+100`, and `-100`. Therefore the sets associated with ⟨*Name*⟩, ⟨*String*⟩ and ⟨*Integer*⟩ are pairwise disjoint. The ambiguity between rule (G20) and (G21) is resolved by giving precedence to (G20), and for path expressions we assume that the operators "`/`" and "`//`" are left associative (rule (G17)). When refering to multiple rules, we often use the following shorthand: a dash is used to express a range of rules and a comma for expressing the union of a set of rules, e.g., (G18-G20,G23) refers to four rules, i.e., (G18), (G19), (G20) and (G23).

Every expression is evaluated against a store, which contains the Web[1] and XML fragments loaded into memory, and an environment containing variable assignments and function definitions. The evaluation of an expression results into a new store and a list of values over that store. Moreover, a list of pending updates can be returned. These are updates that are not yet performed, but that we will apply to the store later on. All LiXQuery expressions are either updating expressions, i.e., those expressions that generate a list of pending updates or apply this list to the store, or non-updating expressions. Note that non-updating expressions can propagate a list of pending updates generated by their subexpressions. We now give a brief and informal description of the semantics of non-updating and updating expressions.

2.1.1 Non-Updating Expressions

The expression `()`, defined in rule (G7), returns the empty sequence. In rule (G8) the expression e_1`,`e_2 denotes sequence concatenation that returns the concatenation of the results of e_1 and e_2. For example, if e_1 returns $\langle 1, 2\rangle$ and e_2 returns $\langle 3, 4\rangle$, then e_1`,`e_2 returns $\langle 1, 2, 3, 4\rangle$. In the semantics of XQuery single values such as numbers strings and nodes are assumed to be identical with a singleton sequence that contains them. So the expression `(1, 2)` in fact denotes the concatenation of the sequences $\langle 1\rangle$ and $\langle 2\rangle$, which is indeed the sequence $\langle 1, 2\rangle$. In rule (G9) we find the boolean conjunction and disjunction of two singleton sequences with boolean values. The rule (G10) introduces the comparison operators for basic values. Note that `2 < 10` and `"10" < "2"` both hold. These comparison operators have existential semantics, i.e., they are true for two sequences if there is a basic value in one sequence and a basic value in the other sequence such that the comparison holds between these two basic values. Rule (G11) defines basic arithmetic operations.

In rule (G12) the expression `if (`e_1`) then` e_2 `else` e_3 denotes the usual conditional expression where e_1 is required to return a singleton sequence containing a boolean value. In rule (G13) the expression `for $`x `in` e_1 `return` e_2 expresses iteration where the result is computed by iterating over each element in the sequence that is the result of e_1, binding this element to `$x` and evaluating e_2, and finally concatenating all the sequences that resulted from the evaluation of e_2. For example, the expression `for $x in (1, 2, 3) return ($x, $x)` returns the sequence $\langle 1, 1, 2, 2, 3, 3\rangle$. The optional `at` clause can be used to bind a variable to the position of the current item in a sequence during the iteration of

[1] In XQuery implementations, documents from the Web are added to the store on a call-by-need basis. However, for our research purposes, this does not matter and hence we say that conceptually the documents from the Web are already in the store.

(G1) ⟨*Program*⟩ ::= ((⟨*VarDecl*⟩ | ⟨*FunDecl*⟩) “;”)* ⟨*Expr*⟩
(G2) ⟨*VarDecl*⟩ ::= “`declare`” “`variable`” ⟨*Name*⟩ “`:=`” ⟨*Expr*⟩
(G3) ⟨*FunDecl*⟩ ::= “`declare`” “`function`” ⟨*Name*⟩
“`(`” (⟨*Var*⟩ (“`,`” ⟨*Var*⟩)*)? “`)`” “{” ⟨*Expr*⟩ “}”
(G4) ⟨*Expr*⟩ ::= ⟨*Var*⟩ | ⟨*Literal*⟩ | ⟨*EmpSeq*⟩ | ⟨*Concat*⟩ | ⟨*AndOr*⟩ | ⟨*ValCmp*⟩ |
⟨*NodCmp*⟩ | ⟨*Arithm*⟩ | ⟨*IfExpr*⟩ | ⟨*LetExpr*⟩ | ⟨*ForExpr*⟩ |
⟨*Step*⟩ | ⟨*Path*⟩ | ⟨*TypeSw*⟩ | ⟨*FunCall*⟩ | ⟨*BuiltIn*⟩ | ⟨*SeqGen*⟩ |
⟨*Constr*⟩ | ⟨*Insert*⟩ | ⟨*Rename*⟩ | ⟨*Replace*⟩ | ⟨*Delete*⟩ |
⟨*Snap*⟩ | ⟨*Transform*⟩
(G5) ⟨*Var*⟩ ::= “`$`”⟨*Name*⟩
(G6) ⟨*Literal*⟩ ::= ⟨*String*⟩ | ⟨*Integer*⟩
(G7) ⟨*EmpSeq*⟩ ::= “`()`”
(G8) ⟨*Concat*⟩ ::= ⟨*Expr*⟩“`,`”⟨*Expr*⟩
(G9) ⟨*AndOr*⟩ ::= ⟨*Expr*⟩(“`and`” | “`or`”)⟨*Expr*⟩
(G10) ⟨*ValCmp*⟩ ::= ⟨*Expr*⟩(“`=`” | “`<`”)⟨*Expr*⟩
(G11) ⟨*Arithm*⟩ ::= ⟨*Expr*⟩ (“`+`” | “`-`” | “`*`” | “`idiv`”) ⟨*Expr*⟩
(G12) ⟨*IfExpr*⟩ ::= “`if `”“`(`”⟨*Expr*⟩“`)`” “`then`”⟨*Expr*⟩ “`else`”⟨*Expr*⟩
(G13) ⟨*ForExpr*⟩ ::= “`for`”⟨*Var*⟩ (“`at`” ⟨*Var*⟩)? “`in`”⟨*Expr*⟩ “`return`” ⟨*Expr*⟩
(G14) ⟨*LetExpr*⟩ ::= “`let`”⟨*Var*⟩“`:=`”⟨*Expr*⟩ “`return`” ⟨*Expr*⟩
(G15) ⟨*NdCmp*⟩ ::= ⟨*Expr*⟩(“`is`” | “`<<`”) ⟨*Expr*⟩
(G16) ⟨*Step*⟩ ::= “`.`” | “`..`” | “`*`” | “`@*`” | “`text()`” | ⟨*Name*⟩ | “`@`” ⟨*Name*⟩
(G17) ⟨*Path*⟩ ::= ⟨*Expr*⟩(“`/`” | “`//`”)⟨*Expr*⟩
(G18) ⟨*TypeSw*⟩ ::= “`typeswitch `”“`(`”⟨*Expr*⟩“`)`” ((“`case`” ⟨*Type*⟩ “`return`”⟨*Expr*⟩)$^+$
“`default`” “`return`”⟨*Expr*⟩)
(G19) ⟨*Type*⟩ ::= “`xs:boolean`” | “`xs:integer`” | “`xs:string`” | “`element()`” |
“`attribute()`” | “`text()`” | “`document-node()`”
(G20) ⟨*BuiltIn*⟩ ::= “`doc(`”⟨*Expr*⟩“`)`” | “`string(`”⟨*Expr*⟩“`)`” | “`name(`”⟨*Expr*⟩“`)`” |
“`xs:integer(`”⟨*Expr*⟩“`)`” | “`root(`”⟨*Expr*⟩“`)`” |
“`concat(`”⟨*Expr*⟩, ⟨*Expr*⟩“`)`” | “`count(`” ⟨*Expr*⟩ “`)`” |
“`true()`” | “`false()`” | “`not(`”⟨*Expr*⟩“`)`”
(G21) ⟨*FunCall*⟩ ::= ⟨*Name*⟩ “`(`”(⟨*Expr*⟩(“`,`”⟨*Expr*⟩)*)?“`)`”
(G22) ⟨*SeqGen*⟩ ::= ⟨*Expr*⟩ “`to`” ⟨*Expr*⟩
(G23) ⟨*Constr*⟩ ::= “`element`” “{” ⟨*Expr*⟩ “}” “{” ⟨*Expr*⟩ “}” |
“`attribute`” “{” ⟨*Expr*⟩ “}” “{” ⟨*Expr*⟩ “}” |
“`document`” “{” ⟨*Expr*⟩ “}” | “`text`” “{” ⟨*Expr*⟩ “}”
(G24) ⟨*Insert*⟩ ::= “`insert`” ⟨*Expr*⟩ (“`into`” | “`before`” | “`after`”) ⟨*Expr*⟩
(G25) ⟨*Rename*⟩ ::= “`rename`” ⟨*Expr*⟩ “`as`” ⟨*Expr*⟩
(G26) ⟨*Replace*⟩ ::= “`replace`” “`value`” “`of`” ⟨*Expr*⟩ “`with`” ⟨*Expr*⟩
(G27) ⟨*Delete*⟩ ::= “`delete`” ⟨*Expr*⟩
(G28) ⟨*Snap*⟩ ::= “`snap`” ((“`unordered`” (“`nondeterministic`” |
“`deterministic`”)) | “`ordered`”) “{” ⟨*Expr*⟩ “}”
(G29) ⟨*Transform*⟩ ::= “`transform`” “`copy`” ⟨*Var*⟩ “`:=`” ⟨*Expr*⟩
“`modify`” ⟨*Expr*⟩ “`return`” ⟨*Expr*⟩

Figure 2.1: Syntax of LiXQuery

the `for` expression. As an example the expression `for $x at $y in ("a", "b", "c") return ($y, $x)` returns the sequence ⟨`1`, `"a"`, `2`, `"b"`, `3`, `"c"`⟩. In rule (G14) the expression `let` $\$x$ `:=` e_1 `return` e_2 returns the result of evaluating e_2 with $\$x$ bound to the result of e_1.

Rule (G15) gives the comparison operators for singleton sequences containing nodes where "`is`" tests the equality of nodes and "`<<`" compares nodes in document order. In rule (G16) we find the basic steps in path expressions that navigate starting from a so-called *context item* which is either a node or a basic value. The expression `.` simply returns the context item. The expression `..` returns the parent node of the context item if the context item is a node that is not a root node. An expression of the form N, with N a valid element name, returns all children of the context item which are element nodes with the name N. Likewise the expression `@`N retrieves the attribute nodes under the context item with name N. The wildcard expressions `*` and `@*` return respectively all element nodes and all attribute nodes under the context item. Finally the `text()` expression returns all text nodes under the context item. In rule (G17) the expressions for composing path expressions are defined. An expression e_1`/`e_2 returns the sequence of nodes that is obtained by evaluating e_1 and then for each node in its result evaluate e_2 with this node as the context item, and finally take the union of all nodes in the results of e_2. The resulting sequence is without duplicates and sorted in document order, i.e., in the order that the associated elements, attributes, etc. are encountered in the document. The expression e_1`//`e_2 has the same semantics except that we evaluate e_2 not only for all the nodes in the result of e_1 but also for all their descendants[2].

The type-switch expression, defined by rules (G18-G19), allows us to check the type of a node or basic value. For an example consider the following expression:

```
for $x in ($y/@*, $y/*, $y/text())
return (
  typeswitch ($x)
    case element() return string($x)
    case attribute() return concat("@",string($x))
    default return "text" )
```

If it is evaluated while `$y` is bound to the root node of an XML fragment `<test id="5"> <result/> scheduled </test>` then the result will be the sequence ⟨`"@id"`, `"result"`, `"text"`⟩. In rule (G20) the built-in functions are declared. The function `string()` gives the string value of an attribute node or text node, and converts integers to strings. The function `xs:integer()`[3] converts strings to integers. The function `doc()` returns the document node that is the root of the tree that corresponds to the content of the file with the name that was given as its argument, e.g., `doc("file.xq")` indicates the document root of the content of the file `file.xq`. The function `name()` gives the tag name of an element node or the attribute name of an attribute node. The function `root()` returns for a node the root

[2] In XQuery the `//` expression is a shorthand for `/descendant-or-self::node()/`

[3] "`xs:`" indicates a namespace. Although we do not handle namespaces we use them here to be compatible with XQuery.

of the tree it belongs to. The function `concat()` concatenates strings and the function `count()` counts the items in a sequence. The functions `true()` and `false()` return the boolean values **true** and **false**, respectively. The function `not()` inverts the the boolean value of its argument.

The rules (G21,G3) allow the application and definition of functions, and specifically recursive functions. An expression of the form N`(`e_1`, ..., `e_n`)` applies the function with name N to the results of $e_1, \ldots, e_n$. An expression of the form `declare function` N`(`v_1`,...,`v_n`)`$\{e\}$ declares a function with name N, formal arguments $v_1, \ldots, v_n$ and body e.

The rule (G22) defines expressions of the form e_1 `to` e_2 which construct a sequence of numbers beginning from the result of e_1 up to and including the result of e_2. For example the expression `1 to 4` returns the sequence $\langle 1, 2, 3, 4 \rangle$. It is an interesting operation since it is one of the few operations that allows the construction of large results where the size depends not so much on the size of the input but on the magnitude of the numbers in the input.

Rule (G23) introduces expressions for creating new nodes. An expression `document`$\{e\}$ creates a new document node with as its contents a deep copy of the resulting sequence of e. An expression `element`$\{e_1\}\{e_2\}$ creates a new element node with a name computed by e_1 and a deep copy of the result of e_2 as its contents. Note that these contents include any attribute nodes that should be associated with the new node. Finally, an expression `text`$\{e\}$ constructs a new text node with the string computed by e as its string value.

All previous operations are non-updating expressions, and as opposed to the XQuery Update Facility, they always allow updating subexpressions. In the formal semantics, presented in Section 2.4, we will make clear how this subtly changes the informal semantics that we have just described.

2.1.2 Updating Expressions

We now describe the semantics of the update expressions, i.e., the `insert`, `rename`, `replace` and `delete` operations. The `insert` e_1 `(into | before | after)` e_2 operation makes a copy of the nodes in the result of the first expression and adds these (afterwards) at the position that is indicated by either `into`, `before`, or `after` and which is relative to the singleton result node of the second expression. Note that the `insert into` inserts a new node n_2 as a child of an existing node n_1, but does not specify the relative position in document order of the newly inserted node n_2 among the children of the target node n_1, i.e., the `insert into` operation has a non-deterministic semantics. The `rename` e_1 `as` e_2 operation renames an element or an attribute, and the `replace value of` e_1 `with` e_2 operation replaces the value of a text or an attribute node with a new atomic value. Both operations are node-identity preserving, i.e., the identity of the updated node is not changed. The `delete` e expression removes the incoming edges[4] for a set of nodes, which

[4]Whenever we refer to edges here, we refer to parent-child edges in a forest and not other edges that occur into some data models, such as DOM.

can then be garbage collected iff they are not accessible anymore through variable bindings or the result sequence.

For most expressions we assume a snapshot semantics, i.e., update expressions are not executed when evaluated but translated to primitive updates which are collected in a list of pending updates. This intuitively corresponds to making a snapshot of the store before evaluating the expression. In general, the list of pending updates is applied in the same order the updates are generated at the end of the evaluation of the program. There are three exceptions to this: the `snap` operation, expressions at the right-hand side of a variable declaration and the `transform` expression. We discuss these three cases in the following.

A `snap` operation applies the list of pending updates that is generated by the subexpression to the store and returns an empty update list. If the snap expression contains the keyword `ordered`, then the pending updates are applied in the same order as they were generated. Else the `snap` expression contains the keyword `unordered` to state that the order of application of the pending updates is undefined. The keyword `unordered` is followed by either `deterministic` or `nondeterministic`, which specify whether the order of the application of pending updates is respectively allowed to affect the set of possible results or not. As an illustration of the `snap` expression consider:

```
for $d in //dept return (
  snap ordered { replace value of $d/salarytotal with 0 },
  for $e in $d/emp return
    snap ordered {
      replace value of $d/salarytotal
      with $d/salarytotal + $e/salarytotal } )
```

This expression computes for each department the total of the salaries of its employees. Note that if we replace the two `snap` operations with one big `snap` operation around the whole expression then it will compute for each department the salary of the last employee since the value of `$d/salarytotal` is not updated during the evaluation.

When evaluating an expression at the right-hand side of a variable declaration, an implicit top-level "`snap ordered`" is presumed, i.e., the list of pending updates that is generated by the expression is applied to the store.

The final exception to the snapshot semantics is the "`transform`" operation. It makes a deep copy of the result of the first subexpression, evaluates the second subexpression and applies the resulting pending updates provided these are only on the deep copy, and finally evaluates the return clause and returns its result. As an illustration of the "`transform`" expression consider:

```
transform copy $d := //dept[@name = "Security"]
          modify delete $d//*[@security-level > 3]
          return $d
```

This expression retrieves all information about the security department except the subtrees which have a security level higher than three. Note that the transform operation can update only fragments it creates itself and not an existing fragment in the XML store. This suggests that it does not add expressive power to LiXQuery, but as can be shown

from the results in Chapter 3, we need recursive function definitions and node construction to simulate the `transform` expression in general.

2.2 Some Examples and Syntactic Sugar

Before proceeding with the formal semantics of LiXQuery, we first give some examples to demonstrate the expressive power of LiXQuery and show how we can express some XQuery constructs that are not in LiXQuery. To allow for a shorter notation of certain very common expressions we will also introduce some short-hands.

The Empty Function The first shorthand we define is used to test emptiness of a sequence. This can be done by using the existential semantics of atomic value comparisons. Hence, the function `empty()` is assumed to be declared as follows:

```
let $seq := (for $x in e1 return 1)
return ($seq = 1)
```

Quantified Formulas The expression `some` $\$v$ `in` e_1 `satisfies` e_2 is introduced as a shorthand for `not(empty(for` $\$v$ `in` e_1 `return if (`e_2`) then` $\$v$ `else ()))`, and `every` $\$v$ `in` e_1 `satisfies` e_2 is introduced as a shorthand for `empty(for` $\$v$ `in` e_1 `return if (`e_2`) then () else` $\$v$`)`.

FLWOR Expression When for- and let-expressions are nested we allow that the intermediate "`return`" is removed. E.g., `for` $\$v_1$ `in` e_1 `return let` $\$v_2$ `:=` e_2 `return` e_3 may be written as `for` $\$v_1$ `in` e_1 `let` $\$v_2$ `:=` e_2 `return` e_3. Furthermore we allow in for- and let-expressions the shorthand `where` e_1 `return` e_2 for `return if` e_1 `then` e_2 `else ()`.

Coercion Let e_1 (or e_2) have the form `string(`e`)` where the result of e is a sequence containing a single text node or a single attribute node. Then e_1 (or e_2) can be replaced by e in the following expressions: `xs:integer(`e_1`)`, `concat(`e_1`,`e_2`)`, e_1`=`e_2, e_1`<`e_2 and `attribute{`e_3`}{`e_2`}`.

Simulating Deep Equality The first example shows that we can express deep equality of two sequences. This essentially means that we have to check whether two fragments are isomorphic except that we have to take into account that attributes are unordered.

```
(: detects whether the attributes of $e are equal in name and value
   with those of $f :)
declare function deepat1($e,$f) {
  every $ae in $e/@* satisfies
    some $af in $f/@* satisfies
      ( name($ae)=name($af) and string($ae)=string($af) )
```

```
  and
  every $af in $f/@* satisfies
    some $ae in $e/@* satisfies
      ( name($ae)=name($af) and string($ae)=string($af) )
};

(: verifies whether $e is a textnode :)
declare function typetext($e) {
  typeswitch ($e) case text() return true() default return false()
};

(: detects whether $se and $sf are sequences of pairwise
   deep-equal items :)
declare function deepequal($se,$sf) {
  if (empty($se) and empty($sf)) then true()
    else
  if (empty($se) or empty($sf)) then false()
    else
  if (typetext($se[1]))
    then if (typetext($sf[1]))
          then ( string($se[1])=string($sf[1]) and
                 deepequal($se[1 < position()], $sf[1 < position()]) )
          else false()
    else if (typetext($sf[1]))
          then false()
          else ( name($se[1])=name($sf[1]) and
                 deepat1($se[1],$sf[1]) and
                 deepequal($se[1]/(*|text()), $sf[1]/(*|text())) and
                 deepequal($se[1 < position()], $sf[1 < position()])
                 )
};
```

Simulation of other Axes We can simulate all the axes that are not already directly supported in the syntax of LiXQuery. For example, the following expression is a simulation of `following-sibling::node()`:

```
for $dot in ../(*, text()) return (
  if (. << $dot) then $dot else () )
```

A second axis we give as an illustration of the above claim is `ancestor::node()`:

```
let $dot := . return (
  for $anc in root($dot)//.
  where some $node in $anc//. satisfies $node is $dot
  return $anc)
```

Simulation of the full `string()` Function In LiXQuery the `string()` function is only defined for integers, attribute nodes and text nodes, but in XQuery it is defined for all items. We can simulate this more general function as follows.

```
(: concatenate all strings in $x :)
declare function concatAll($x) {
  if ( empty( $x ) )
  then ""
  else concat($x[position()=1], concatAll($x[position()>1]))
};

(: simulates full xquery string function :)
declare function xqString($x) {
  if ( empty( $x ) )
  then ""
  else typeswitch ($x)
       case document-node() return concatAll($x/text())
       case element() return concatAll($x/text())
       default return string($x)
};
```

Simulation of Predicates A filter expression e_1`[`e_2`]` iterates over the result of e_1 and select each node for which e_2 evaluates to **true** while taking this node as the context item. For example, `(1, 2, 3, 4)[. > 2]` returns the sequence $\langle 3, 5\rangle$. A simulation of e_1`[`e_2`]` can be done in two steps. First, we construct e_2' from e_2 by replacing every subexpression that depends upon the context item from e_1, i.e., is not nested in the e_2' of e_1'`[`e_2'`]`, e_1'`/`e_2' or e_1'`//`e_2', is replaced as follows: the expression `.` is replaced with `$dot`, and all expressions e of the forms N, `*`, `@`N, `@*` and `text()` are replaced with `$dot/`e. After this we can simulate e_1`[`e_2`]` with:

```
for $dot in e1 return (
  if (e2') then $dot else ())
```

Note that another possible simulation is e_1`/(if (`e_2`) then . else ())` but only if the result contains only nodes and is already sorted in document order. We now show that we can also simulate `position()` and/or `last()`, which can be used in every expression that is evaluated for a context item that is taken from some sequence. These are the subexpressions e_2 in expressions of the forms e_1`/`e_2, e_1`//`e_2 and e_1`[`e_2`]`. The meaning of these functions is for e_1`/`e_2 and e_1`[`e_2`]` the same: `position()` refers to the position of the context item in the result sequence of e_1 and `last()` refers to the position of the last item in this sequence, i.e., the length of the sequence. For example, `("a", "b", "c", "d")[position() > 2]` returns $\langle$`"c","d"`$\rangle$ and `("a", "b", "c", "d")[position() = last()]` returns $\langle$`"d"`$\rangle$. Another example is the expression `a/(if (position() = 2) then . else ())` which returns the second `a` element child of the context item. For expressions of the form e_1`//`e_2 the semantics of `position()` and `last()`

are similar except that the context sequence to which `position()` and `last()` refer is differently defined. Recall that for each node n in the result of e_1 the evaluation iterates over n and its descendants and evaluates e_2. During the evaluation of e_2 the sequence consisting of n and its descendants is assumed to be the context sequence.

We can simulate e_1`[`e_2`]` with `position()` and/or `last()` in e_2 as follows, assuming e_2' is constructed from e_2 as previously explained for the preceding simulation of e_1`[`e_2`]` plus that in addition `position()` and `last()` are replaced by the variables `$pos` and `$last`[5]:

```
let $seq := e1 return
let $last := count($seq) return
for $dot at $pos in e1 return
  if (e2') then $dot else ()
```

The simulation of e_1`/`e_2 with `position()` and/or `last()` in e_2 is similar except that here the results of e_2 are returned and the final result is sorted in document order by applying `/.`:

```
let $seq := e1 return
let $last := count($seq) return
(for $dot at $pos in e1 return e2')/.
```

The simulation of e_1`//`e_2 with `position()` or `last()` in e_2 is accomplished by rewriting it as e_1`/((.//.)/`e_2`)` and reducing it thus to the previous case.

Simulation of the `order by` clause The final XQuery feature that we use to illustrate the claim that most typical XQuery expressions can be expressed in LiXQuery is the `order by` clause:

```
for $x at $y in e1 order by e2
return e3
```

We show that we can simulate this expression without using recursive functions. Assume that the expression in e_2 yields exactly one item when evaluated against an item in the result of e_1. If the evaluation of e_1 yields n items, then the sequence that we have to order according to the order by clause e_2 has also n items. We will create a permutation of the numbers 1 to n in a variable `$ordByPos` such that the i^{th} item in this sequence is j iff the j^{th} item in the result sequence of e_1 is the smallest item in the result of e_1 with at least $i - 1$ items smaller or equal. In order to obtain a stable order, we also have to incorporate the position of the items in the result of e_1. The simulation of the `order by` can then be performed as follows:

```
let $inExpr := e1 return
let $unordBy := (for $x at $y in $inExpr return (e2)) return
let $ordByPos := (
  for $n at $p in $unordBy return (
  for $n2 at $p2 in $unordBy return (
```

[5] We assume, w.l.o.g., that `$pos` and `$last` do not occur in e_2.

```
<list>
  <list> <atom> b </atom> <atom> c </atom> </list> <atom> d </atom>
</list>
```

Figure 2.2: Simulation of the LISP list `((b c) d)`

```
    let $smaller := (for $n3 at $p3 in $unordBy
                      return if ($n3 < $n2) then $n3 else ()) return (
    let $equalBef := (for $n3 at $p3 in $unordBy
                       return if (($n3 = $n2) and ($p3 < $p2))
                              then $n3 else ()) return
      if (count(($smaller,$equalBef)) = $p) then $p2 else ()
    )
  ))
) return
let $ordInExpr := (
  for $x at $pos in $inExpr return
    for $y in $ordByPos return
      if ($y = $pos) then $x else ()
) return
for $x at $pos in $ordInExpr return
let $y := (for $xx at $yy in $ordByPos
          return if ($xx = $pos) then $yy else ())
return e3
```

Turing Completeness It is easy to see that the amount of arithmetic and recursion in LiXQuery allows us to express all partial recursive functions over numbers. It is also possible to simulate LISP, which is known to be a Turing-complete programming language. For this purpose we represent a LISP list `((b c) d)` as shown in Figure 2.2. Given this representation we simulate the `car`, `cdr`, `cons` functions:

```
declare function car($x) { $x/*[1] };
declare function cdr($x) { element{ "list" }{ $x/*[1 < position()] } };
declare function cons($x,$y) { element{ "list" }{ $x,$y/* } };
```

Since we can also compare strings and have conditional expressions, it is easy to see that by using recursion we can define all partial recursive functions over LISP lists.

2.3 Formal Framework

We now proceed with the formal semantics of LiXQuery. We assume a set of *booleans* $\mathcal{B} = \{\mathbf{true}, \mathbf{false}\}$, a set of *strings* $\mathcal{S}$ and a set of *integers* $\mathcal{I}$. Furthermore a set of *names* $\mathcal{N} \subseteq \mathcal{S}$ is identified that contains those strings that may be used as tag names. For each of

these sets a strict total ordering, written as $<$, is presumed to exist. The set of all atomic values is $\mathcal{A} = \mathcal{B} \cup \mathcal{S} \cup \mathcal{I}$. We also assume four countably infinite sets of nodes $\mathcal{V}^d$, $\mathcal{V}^e$, $\mathcal{V}^a$ and $\mathcal{V}^t$ which respectively represent the set of *document*, *element*, *attribute* and *text nodes*. These sets are pairwise disjoint with each other and with the set of atomic values. The set of all nodes is denoted as $\mathcal{V}$, i.e., $\mathcal{V} = \mathcal{V}^d \cup \mathcal{V}^e \cup \mathcal{V}^a \cup \mathcal{V}^t$. The following notation is used: v for values, x for items, n for nodes, r for roots, s for strings and names, f for function names, b for booleans , i for integers and e for expressions. The empty string is denoted as "", non-empty strings as for example "a" and the concatenation of two strings s_1 and s_2 as $s_1 \cdot s_2$. The empty sequence is denoted as $\langle\rangle$, non-empty sequences as for example $\langle 1, 2, 3\rangle$ and the concatenation of two sequences l_1 and l_2 as $l_1 \circ l_2$. The set of sequences over a set S is denoted by S^*. Finally, $\mathcal{X}$ denotes the set of all LiXQuery expressions.

2.3.1 XML Store

Statements will be evaluated against an *XML store* which contains XML fragments. This store contains the fragments that are created as intermediate results, but also the web documents that are accessed by the expression. Although in practice these documents are materialized in the store when they are accessed for the first time, we will assume here that all documents are in fact already in the store when the expression is evaluated.

Definition 2.1 (XML Store)**.** *An XML store is a 6-tuple* $St = (V, E, \ll, \nu, \sigma, \delta)$ *with*

- V *is a finite subset of* $\mathcal{V}$*; we write* V^d *for* $V \cap \mathcal{V}^d$ *(resp.* V^e *for* $V \cap \mathcal{V}^e$*,* V^a *for* $V \cap \mathcal{V}^a$*,* V^t *for* $V \cap \mathcal{V}^t$*);*
- (V, E) *is an acyclic directed graph (with nodes* V *and directed edges* $E \subset V \times V$*), i.e., a forest, where each node has an in-degree of at most one, and hence it is composed of trees; if* $(m, n) \in E$ *then we say that* n *is a child of* m*;*[6] *we denote by* E^* *the reflexive transitive closure of* E*;*
- $\ll$ *is a total order on the nodes of* V*;*
- $\nu : V^e \cup V^a \to \mathcal{N}$ *labels the element and attribute nodes with their* node name*;*
- $\sigma : V^a \cup V^t \to \mathcal{S}$ *labels the attribute and text nodes with their* string value*;*
- $\delta : \mathcal{S} \to \mathcal{V}^d$ *a partial function that associates with a URI or a file name, a document node. It is called the* document function*. This function represents all the URIs of the Web and all the names of the files, together with the documents they contain. We suppose that all these documents are in the store.*

The following properties have to hold for an XML store:

[6] As opposed to the terminology of XQuery, we consider attribute nodes as children of their associated element node. The definitions of parent, descendant and ancestor are straightforward.

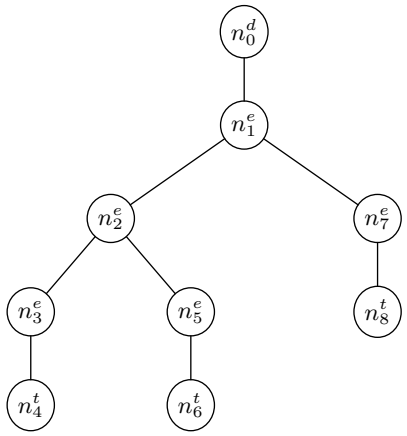


Figure 2.3: XML tree for the Example of Definition 2.1

1. *each document node of* V^d *is the root of a tree and has only one child which is an element node;*

2. *attribute nodes of* V^a *and text nodes of* V^t *do not have any children;*

3. *the* $\ll$ *-order is the document order over* (V, E) *such that for all trees it corresponds to its preorder, i.e.:*

   (a) *if* $(n_1, n_2) \in E^*$ *and* $n_1 \neq n_2$ *then* $n_1 \ll n_2$*;*

   (b) *if* $(m, n_1), (m, n_3) \in E$*,* $(n_1, n_2) \in E^*$*, and* $n_1 \ll n_3$ *then* $n_2 \ll n_3$*;*

4. *nodes of two different trees are not "mixed" in document order, i.e., if* (n_1, n_2)*,* $(n_1, n_4) \in E^*$ *and* $n_2 \ll n_3 \ll n_4$ *then* $(n_1, n_3) \in E^*$*.*

5. *in the* $\ll$ *-order attribute children precede the element and text children, i.e., if* $(m, n_1), (m, n_2) \in E$*,* $n_1 \ll n_2$ *and* $n_2 \in V^a$ *then* $n_1 \in V^a$*;*

6. *there are no adjacent text children, i.e., if* (m, n_1)*,* $(m, n_2) \in E$*,* $n_1, n_2 \in V^t$*, and* $n_1 \ll n_2$ *then there is an* $n_3 \in V^e$ *with* $n_1 \ll n_3 \ll n_2$*;*

7. *for all text nodes* n_t *of* V^t *holds* $\sigma(n_t) \neq$ *"";*

8. *all the attribute children of a common node have a different name, i.e., if* (m, n_1)*,* $(m, n_2) \in E$ *and* $n_1, n_2 \in V^a$ *then* $\nu(n_1) \neq \nu(n_2)$*.*

We write V_{St} to denote the set of nodes of the store St, and similarly we write E_{St}, $\ll_{St}$, ν_{St}, σ_{St} and δ_{St} to denote respectively the second to the sixth component of the 6-tuple St. An *item* of an XML store St is an atomic value in $\mathcal{A}$ or a node in St.

We now give an example to illustrate Definition 2.1. Assume that we have only one document on the web, with URI `"doc.xml"` which contains the following XML fragment:

```
<a>
  <b>
    <c>t1</c>
    <c>t2</c>
  </b>
  <b>t3</b>
</a>
```

The following store $St = (V, E, \ll, \nu, \sigma, \delta)$ corresponds to this XML fragment:

- The set of nodes V consists of $V^d = \{n_0^d\}$, $V^e = \{n_1^e, n_2^e, n_3^e, n_5^e, n_7^e\}$, $V^t = \{n_4^t, n_6^t, n_8^t\}$, $V^a = \emptyset$.
- The set of edges is $E = \{(n_0^d, n_1^e), (n_1^e, n_2^e), (n_1^e, n_7^e), (n_2^e, n_3^e), (n_2^e, n_5^e), (n_3^e, n_4^t), (n_5^e, n_6^t), (n_7^e, n_8^t)\}$.
- The document order $\ll$ is defined by $n_0^d \ll n_1^e \ll n_2^e \ll n_3^e \ll n_4^t \ll n_5^e \ll n_6^t \ll n_7^e \ll n_8^t$.
- Furthermore $\nu(n_1^e) =$ `"a"`, $\nu(n_2^e) = (n_7^e) =$ `"b"`, $\nu(n_3^e) = \nu(n_5^e) =$ `"c"`, $\sigma(n_4^t) =$ `t1`, $\sigma(n_6^t) =$ `t2`, $\sigma(n_8^t) =$ `t3`, and $\delta(\text{“}doc.xml\text{”}) = n_0^d$.

2.3.2 Evaluation Environment

Expressions are evaluated against an environment, which is defined as follows.

Definition 2.2 (Environment)**.** *An environment of an XML store St is a tuple* $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ *with*

1. *a partial function* $\mathbf{a} : \mathcal{N} \to \mathcal{N}^*$ *that maps a function name to its formal arguments,*
2. *a partial function* $\mathbf{b} : \mathcal{N} \to \mathcal{X}$ *that maps a function name to the body of the function,*
3. *a partial function* $\mathbf{v} : \mathcal{N} \to (\mathcal{V} \cup \mathcal{A})^*$ *that maps variable names to their values, and*
4. $\mathbf{x}$ *which is undefined* ($\perp$) *or an item of St and indicates the context item.*

2.3.3 Auxiliary Notions

Before continuing the specification of the formal framework by defining the list of pending updates, we give some notational tools.

First, we give some notations for sets, bags and lists/sequences. We use c to denote collections that can be either sets, bags or lists. The set of items in a list or bag c is denoted by $\mathbf{Set}(c)$. The bag representation of a list c is denoted by $\mathbf{Bag}(c)$.

We now define some auxiliary operations on stores.

Definition 2.3 (Union of Stores)**.** *Two stores St and St' are disjoint, denoted as $St \cap St' = \emptyset$, iff $V_{St} \cap V_{St'} = \emptyset$. The definition of the union of two disjoint stores St and St', denoted as $St \cup St'$, is straightforward. The resulting document order is extended to a total order in a non-deterministic way.*

Given a sequence of nodes l in an XML store St we let $\mathbf{Ord}_{St}(l)$ denote the unique sequence $l' = \langle y_1, \ldots, y_m \rangle$ such that $\mathbf{Set}(l) = \mathbf{Set}(l')$ and $y_1 \ll_{St} \ldots \ll_{St} y_m$.

Sometimes we are only interested in certain substores of an XML store. Let St be a store and n an element of V_{St}. We define V_{St}^n as $\{n' | (n, n') \in E_{St}^*\}$, i.e., the set of nodes in the subtree rooted at n in St. The projection of St to a set of nodes $V \subseteq V_{St}$ is denoted by $\Pi_V(St)$ and is a new store St' such that $V_{St'} = V$, $E_{St'} = E_{St} \cap (V \times V)$, $\ll_{St'} = \ll_{St} \cap (V \times V)$, $\nu_{St'} = \nu_{St} \cap (V \times \mathcal{S})$, $\sigma_{St'} = \sigma_{St} \cap (V \times \mathcal{S})$, and $\delta_{St'} = \delta_{St} \cap (\mathcal{S} \times V)$. The restriction of St to n is defined as $\Pi_{V_{St}^n}(St)$ and is denoted by $St[n]$. The exclusion of n from St is defined as $\Pi_{V_{St} - V_{St}^n}(St)$ and is denoted by $St \setminus n$. For both restriction and exclusion it is not hard to see that the projection always results in a store. Finally, if St is a store, n a node in St, and s a string, then we let $St[\delta(n) \mapsto s]$ ($St[\nu(n) \mapsto s]$) denote the store that is equal to St except that $\delta_{St'}(n) = s$ ($\nu_{St'}(n) = s$).

We now define an isomorphism between stores, for which it can be shown that we cannot distinguish between them.

Definition 2.4 (Isomorphic Stores)**.** *Given the XML stores St_1 and St_2. St_1 and St_2 are said to be* isomorphic *iff there is a bijective function $h : V_{St_1} \to V_{St_2}$ such that for each $n, n' \in V_{St_1}$ the following holds:*

1. *if $n \in \mathcal{V}^d$ ($\mathcal{V}^e$, $\mathcal{V}^a$, $\mathcal{V}^t$) then $h(n) \in \mathcal{V}^d$ ($\mathcal{V}^e$, $\mathcal{V}^a$, $\mathcal{V}^t$)*
2. *if $\nu(n) = s$ then $\nu(h(n)) = s$*
3. *if $\sigma(n) = s'$ then $\sigma(h(n)) = s'$*
4. *$(n, n') \in E$ iff $(h(n), h(n')) \in E$*
5. *$n \ll_{St_1} n' \Leftrightarrow h(n) \ll_{St_2} h(n')$*
6. *$\forall s \in \mathcal{S} : \delta_{St_1}(s) = n \Leftrightarrow \delta_{St_2}(s) = h(n)$*

Two trees can be equal up to node identity, meaning that they represent the same XML fragment. We then call these two trees deep-equal, more precisely:

Definition 2.5 (Deep Equal)**.** *Given the XML stores St_1, St_2 and nodes $n_1 \in V_{St_1}$ and $n_2 \in V_{St_2}$, n_1 in St_1 is said to be* deep-equal *with n_2 in St_2, denoted as $\mathbf{DpEq}_{St_1,St_2}(n_1, n_2)$, iff n_1 and n_2 refer to two isomorphic trees, i.e., $St_1[n_1]$ is isomorphic with $St_2[n_2]$. We use $\mathbf{DpEq}_{St}(n_1, n_2)$ as a shorthand for $\mathbf{DpEq}_{St,St}(n_1, n_2)$.*

Finally, give some additional notations for environments. If En is an environment, n a name and y an item then we let $En[\mathbf{a}(n) \mapsto y]$ ($En[\mathbf{b}(n) \mapsto y]$, $En[\mathbf{v}(n) \mapsto y]$) denote the environment that is equal to En except that the function $\mathbf{a}$ ($\mathbf{b}$, $\mathbf{v}$) maps n to y. Similarly, we let $En[\mathbf{x} \mapsto y]$ denote the environment that is equal to En except that $\mathbf{x}$ is defined as y if $y \neq \perp$ and undefined otherwise.

2.3.4 List of Pending Updates

Recall that updates are not executed when updating expressions are evaluated but are collecting in a list of pending updates, which is executed afterwards. This list of pending updates contains a number of primitive update operations.

Definition 2.6 (Primitive Update Operations)**.** *Let $n, n_1, \ldots, n_m$ be nodes in a store St, and $s \in \mathcal{S}$. A primitive update operation on the store St is one of following operations:*

- $insBef(n, \langle n_1, \ldots, n_m \rangle)$ *(insert $n_1, \ldots, n_m$ as sibling before n),*
- $insAft(n, \langle n_1, \ldots, n_m \rangle)$ *(insert $n_1, \ldots, n_m$ as sibling after n),*
- $insInto(n, \langle n_1, \ldots, n_m \rangle)$ *(insert $n_1, \ldots, n_m$ as child from n),*
- $ren(n, s)$ *(rename n to s),*
- $repVal(n, s)$ *(replace value of n with s),*
- $del(n)$ *(delete n).*

Before proceeding with the formal semantics, we first give some intuition about these primitive update operations. The operation $insBef$ ($insAft$, $insInto$) moves nodes n_1 to n_m before (after, into) the node n. In the formal semantics of LiXQuery, we will see that the nodes n_1 to n_m are always copies of other nodes. The operations ren and $repVal$ change respectively the name and the value of n to s. Finally, the operation del removes the incoming edge from n and hence detaches the subtree rooted at n. Note that both del and $insInto$ can have more than one result store, due to the resulting document order.

We write $St \vdash o \Rightarrow^{\mathtt{U}} St'$ to denote that applying the primitive update operation o to St can result in the store St'. The definition of $\Rightarrow^{\mathtt{U}}$ is given in Figure 2.4 by means of inference rules. Each rule consists of a set of premises and a conclusion of the form $St \vdash o \Rightarrow^{\mathtt{U}} St'$. The free variables in the rules are always assumed to be universally quantified.

The target of a primitive update operation is the node that appears as the first paramater of the primitive update operation. The source of a primitive update operation is a bag that is either empty when the operation is $ren, repVal, del$ or the bag representation of the list of nodes which is the second argument for the operations $insBef, insInto, insAft$.

Definition 2.7 (List of Pending Updates)**.** *The* list l of pending updates over a store St *is a list of primitive update operations on St. The set of target nodes of l is denoted by* $\mathbf{Targets}(l)$ *and defined as the set of targets of the primitive update operation appearing in l. The bag of source nodes of l is denoted by* $\mathbf{Sources}(l)$ *and defined as the bag union of the sources of the primitive update operations appearing in l. We say that a list of pending updates is* well-formed *if* $\mathbf{Set}(\mathbf{Sources}(l)) \cap \mathbf{Targets}(l) = \emptyset$ *and there are no nodes that appear twice or more in* $\mathbf{Sources}(l)$.

$$\text{(S1)}\ \frac{St' = St[\nu(n) \mapsto s]}{St \vdash ren(n,s) \Rightarrow^{\mathtt{U}} St'} \qquad \text{(S2)}\ \frac{St' = St[\sigma(n) \mapsto s]}{St \vdash repVal(n,s) \Rightarrow^{\mathtt{U}} St'} \qquad \text{(S3)}\ \frac{St' = (St \setminus n) \cup St[n]}{St \vdash del(n) \Rightarrow^{\mathtt{U}} St'}$$

$$\text{(S4)}\ \frac{\begin{array}{c} St \setminus n_1 \setminus \ldots \setminus n_m = St' \setminus n_1 \setminus \ldots \setminus n_m \qquad St[n_1] = St'[n_1] \qquad \ldots \qquad St[n_m] = St'[n_m] \\ (n,n_1) \in E_{St'} \qquad \ldots \qquad (n,n_m) \in E_{St'} \qquad n_1 \ll_{St'} n_2 \qquad \ldots \qquad n_{m-1} \ll_{St'} n_m \end{array}}{St \vdash insInto(n, \langle n_1, \ldots, n_m \rangle) \Rightarrow^{\mathtt{U}} St'}$$

$$\text{(S5)}\ \frac{\begin{array}{c} n_1, \ldots, n_m \in \mathcal{V}^e \cup \mathcal{V}^t \qquad St \setminus n_1 \setminus \ldots \setminus n_m = St' \setminus n_1 \setminus \ldots \setminus n_m = St'' \qquad St[n_1] = St'[n_1] \\ \ldots \qquad St[n_m] = St'[n_m] \qquad (n' \in V_{St''} \wedge (n,n') \notin E^*_{St}) \Rightarrow (n \ll_{St'} n' \Leftrightarrow n_m \ll_{St'} n') \\ (n',n) \in E_{St} \Rightarrow \{(n',n_1), \ldots, (n',n_m)\} \subset E_{St'} \\ n \ll_{St'} n_1 \qquad n_1 \ll_{St'} n_2 \qquad \ldots \qquad n_{m-1} \ll_{St'} n_m \end{array}}{St \vdash insAft(n, \langle n_1, \ldots, n_m \rangle) \Rightarrow^{\mathtt{U}} St'}$$

$$\text{(S6)}\ \frac{\begin{array}{c} n_1, \ldots, n_m \in \mathcal{V}^e \cup \mathcal{V}^t \qquad St \setminus n_1 \setminus \ldots \setminus n_m = St' \setminus n_1 \setminus \ldots \setminus n_m = St'' \\ St[n_1] = St'[n_1] \qquad \ldots \qquad St[n_m] = St'[n_m] \\ n' \in V_{St''} \Rightarrow (n' \ll_{St'} n \Leftrightarrow n' \ll_{St'} n_1) \qquad (n',n) \in E_{St} \Rightarrow \{(n',n_1), \ldots, (n',n_m)\} \subset E_{St'} \\ n_m \ll_{St'} n \qquad n_1 \ll_{St'} n_2 \qquad \ldots \qquad n_{m-1} \ll_{St'} n_m \end{array}}{St \vdash insBef(n, \langle n_1, \ldots, n_m \rangle) \Rightarrow^{\mathtt{U}} St'}$$

Figure 2.4: Semantics of the Primitive Update Operations.

The notion of well-formed lists of pending updates is needed for Proposition 2.1. It can be shown that every list of pending updates generated by LiXQuery is well-formed.

The notation $St \vdash o \Rightarrow^{\mathtt{U}} St'$, used to specify the semantics of primitive update operations, is overloaded for sequences of primitive update operations. For such a sequence $l = \langle o_1, \ldots, o_m \rangle$ we define $St \vdash l \Rightarrow^{\mathtt{U}} St'$ by induction on m such that (1) $St \vdash \langle\rangle \Rightarrow^{\mathtt{U}} St$ and (2) if $St \vdash \langle o_1, \ldots, o_{m-1} \rangle \Rightarrow^{\mathtt{U}} St'$ and $St' \vdash o_m \Rightarrow^{\mathtt{U}} St''$ then $St \vdash \langle o_1, \ldots, o_m \rangle \Rightarrow^{\mathtt{U}} St''$. For some lists of pending updates, we can reorder the application of these primitive update operations without changing the semantics. Therefore we say that l is *execution-order independent* if for every sequences l' such that $\mathbf{Bag}(l) = \mathbf{Bag}(l')$ and store St' it holds that $St \vdash l \Rightarrow^{\mathtt{U}} St'$ iff $St \vdash l' \Rightarrow^{\mathtt{U}} St'$. Finally, the following proposition gives an algorithm to decide execution-order independence of a well-formed list of pending updates:

Proposition 2.1. *A well-formed list of pending updates $l = \langle o_1, \ldots, o_k \rangle$ over a store St is execution-order dependent iff there are two primitive update operations o_i and o_j in l such that $i \neq j$, and there are $m, n, n_1, \ldots, n_{k'}, n'_1, \ldots, n'_{k''} \in V_{St}$ and $s, s' \in \mathcal{S}$, such that $s \neq s'$, $\langle n_1, \ldots, n_{k'} \rangle \neq \langle n'_1, \ldots, n'_{k''} \rangle$, $k' > 0$, $k'' > 0$, and one of the following holds:*

1. $o_i = ren(n,s) \wedge o_j = ren(n,s')$
2. $o_i = repVal(n,s) \wedge o_j = repVal(n,s')$

3. $o_i = insBef(n, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = insBef(n, \langle n'_1, \ldots, n'_{k''} \rangle)$

4. $o_i = insBef(n, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = del(n)$

5. $o_i = insAft(n, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = insAft(n, \langle n'_1, \ldots, n'_{k''} \rangle)$

6. $o_i = insAft(n, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = del(n)$

7. $o_i = insInto(m, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = insBef(n, \langle n'_1, \ldots, n'_{k''} \rangle) \wedge (m, n) \in E_{St}$

8. $o_i = insInto(m, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = insAft(n, \langle n'_1, \ldots, n'_{k''} \rangle) \wedge (m, n) \in E_{St}$

Proof. First, if one of the eight conflicts stated in the proposition occurs then suppose $l' = \langle o'_1, \ldots, o'_{k-1}, o'_k \rangle$ is a permutation of l such that o'_{k-1} and o'_k are a conflict according to the proposition. Assume l'' is obtained by swapping o'_{k-1} and o'_k in l'. For each of the eight conflicts it can easily be verified that l' can have result stores that cannot be a result store of l'' or vice versa:

1. Let $o'_{k-1} = ren(n, s)$ and $o'_k = ren(n, s')$. For every result store of l' it holds that n has name s, while for every result store of l'' the node n has a different name s'.

2. Let $o'_{k-1} = repVal(n, s)$ and $o'_k = repVal(n, s')$. For every result store of l' it holds that n has value s, while for every result store of l'' the node n has a different value s'.

3. Let $o'_{k-1} = insBef(n, \langle n_1, \ldots, n_{k'} \rangle)$ and $o'_k = insBef(n, \langle n'_1, \ldots, n'_{k''} \rangle)$. For every result store of l' it holds that nodes $\langle n'_1, \ldots, n'_{k''} \rangle$ are immediately before n, while for every result store of l'' nodes $\langle n_1, \ldots, n_{k'} \rangle$ are between $n'_{k''}$ and n.

4. Let $o'_{k-1} = insBef(n, \langle n_1, \ldots, n_{k'} \rangle)$ and $o'_k = del(n)$. For every result store of l'' it holds that n is immediately after $\langle n_1, \ldots, n_{k'}$, while this does not hold for l' since the deletion detaches n from its parent and moves it to an arbitrary position in document order.

5. Similar to 3.

6. Similar to 4.

7. Let $o'_{k-1} = insInto(m, \langle n_1, \ldots, n_{k'} \rangle) \wedge o_j = insBef(n, \langle n'_1, \ldots, n'_{k''} \rangle)$ and $(m, n) \in E_{St}$. If St' is a result of applying $\langle o'_1, \ldots, o'_{k-2} \rangle$ on St then either (m, n) is in $E_{St'}$ or n is deleted earlier and hence has no parent in St'. If n is deleted earlier then conflict 4 occurs. Hence we can assume w.l.o.g. (without loss of generality) that $(m, n) \in E_{St'}$. For the result stores of l' then there are no nodes between n and $n'_{k''}$ while for l'' there can be nodes from $\{n_1, \ldots, n_{k'}\}$ between n and $n'_{k''}$.

8. Similar to 7.

Now, suppose none of the eight conflicts stated in the proposition occurs in $l = \langle o_1, \ldots, o_k \rangle$. We show that every finite number of swaps of adjacent operations in l cannot change the set of output stores for a given input store, by showing for every swap of two adjacent operations that given the same input store, they result in the same set of output stores. Since there are 6 update primitives, we have to consider 21 cases.

1. $ren(n_1, s_1)$ **and** $ren(n_2, s_2)$: Since conflict 1 does not occur, we know that either $n_1 \neq n_2$, or $n_1 = n_2$ and $s_1 = s_2$. In both cases swapping the primitive update operations does not change the result store.

2. $ren(n_1, s_1)$ **and** $repVal(n_2, s_2)$: Trivial.

3. $ren(n_1, s_1)$ **and** $del(n_2)$: Trivial.

4. $ren(n_1, s_1)$ **and** $insBef(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

5. $ren(n_1, s_1)$ **and** $insAft(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

6. $ren(n_1, s_1)$ **and** $insInto(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

7. $repVal(n_1, s_1)$ **and** $repVal(n_2, s_2)$: Similar to 1.

8. $repVal(n_1, s_1)$ **and** $del(n_2)$: Trivial.

9. $repVal(n_1, s_1)$ **and** $insBef(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

10. $repVal(n_1, s_1)$ **and** $insAft(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

11. $repVal(n_1, s_1)$ **and** $insInto(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

12. $del(n_1)$ **and** $del(n_2)$: The new place of n_1 and n_2 in document order is chosen non-deterministically and hence swapping the primitive update operations does not change the result store.

13. $del(n_1)$ **and** $insBef(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Since conflict 4 does not occur, we know that $n_1 \neq n_2$. The nodes $n'_1, \ldots, n'_{k_2}$ will be immediately before n_2 and n_1 will be a root in the result store at an arbitrary place and hence swapping the primitive update operations does not change the result store.

14. $del(n_1)$ **and** $insAft(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Similar to 13.

15. $del(n_1)$ **and** $insInto(n_2, \langle n'_1, \ldots, n'_{k_2} \rangle)$: Trivial.

16. $insBef(n_1, \langle n'_1, \ldots, n'_{k_1} \rangle)$ **and** $insBef(n_2, \langle n''_1, \ldots, n''_{k_2} \rangle)$: Since conflict 3 does not occur, we know that $n_1 \neq n_2$. Hence, it is easy to see that swapping the update operations does not change the result store.

17. $insBef(n_1, \langle n'_1, \ldots, n'_{k_1}\rangle)$ **and** $insAft(n_2, \langle n''_1, \ldots, n''_{k_2}\rangle)$: If $n_1 = n_2$ or n_1 and n_2 are non-adjacent nodes St then this is trivial. If n_1 is the first following sibling of n_2 then in the result store the siblings between n_2 and n_1, ordered by document order, will be $n''_1, \ldots, n''_{k_2}, n'_1, \ldots, n'_{k_1}$, independent of the order in which the two primitive update operations are performed.

18. $insBef(n_1, \langle n'_1, \ldots, n'_{k_1}\rangle)$ **and** $insInto(n_2, \langle n''_1, \ldots, n''_{k_2}\rangle)$: Since conflict 7 does not occur, we know that n_2 is not the parent of n_1. Hence, it is easy to see that swapping the update operations does not change the result store.

19. $insAft(n_1, \langle n'_1, \ldots, n'_{k_1}\rangle)$ **and** $insAft(n_2, \langle n''_1, \ldots, n''_{k_2}\rangle)$: Similar to 16.

20. $insAft(n_1, \langle n'_1, \ldots, n'_{k_1}\rangle)$ **and** $insInto(n_2, \langle n''_1, \ldots, n''_{k_2}\rangle)$: Similar to 18.

21. $insInto(n_1, \langle n'_1, \ldots, n'_{k_1}\rangle)$ **and** $insInto(n_2, \langle n''_1, \ldots, n''_{k_2}\rangle)$: Trivial.

From this we know that we can never obtain different result stores by swapping adjacent operations and hence l is execution-order independent. □

2.3.5 Program Semantics

We now define the semantics of programs. We write $(St, En) \vdash p \Rightarrow (St', v)$ to denote that the *program* p, evaluated against the XML store St and environment En of St, can result in the new XML store St' and value v of St'. Similarly, $(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l)$ means that the evaluation of *expression* e against St and En may result in St', v, and the list of pending updates l over St'. The semantics of expressions is given in Section 2.4. A function declaration changes the environment by extending **a** and **b**. The last expression of the program is then evaluated against the extended environment. Function declarations are allowed to be mutually recursive.

$$\text{(S7)}\ \frac{(St, En[\mathbf{a}(f) \mapsto \langle s_1, \ldots, s_m\rangle][\mathbf{b}(f) \mapsto e]) \vdash p \Rightarrow (St', v')}{(St, En) \vdash \texttt{declare function } f\texttt{(\$}s_1, \ldots, \texttt{\$}s_m\texttt{)\{ } e \texttt{ } \texttt{;} \ p \Rightarrow (St', v')}$$

$$\text{(S8)}\ \frac{(St, En) \vdash e \Rightarrow (St', v) \qquad St', En[\mathbf{v}(s) \mapsto v] \vdash p \Rightarrow (St'', v'')}{(St, En) \vdash \texttt{declare variable \$}s := e \ ; \ p \Rightarrow (St'', v'')}$$

$$\text{(S9)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l) \qquad St' \vdash l \Rightarrow^{\mathtt{U}} St''}{(St, En) \vdash e \Rightarrow (St'', v)}$$

Note that in the last rule, v is a value of St'', since $V_{St'} = V_{St''}$. Also note that updates in a variable declaration are applied immediately after the declaration.

2.4 Expression Semantics

In what follows we give the reasoning rules that are used to define the semantics of LiX-Query expressions. Each rule consists of a set of premises and a conclusion of the form $(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l)$, where St' is the result store, v is the result value and l is the resulting list of pending updates. The free variables in the rules are assumed to be universally quantified.

Variables and Literals (G5-G7) The result of a literal is simply a sequence with one element, viz., the atomic value the literal represents.

$$\text{(S10)}\ \frac{}{(St, En) \vdash \$s \Rightarrow^{\mathtt{E}} (St, \mathbf{v}_{En}(s), \langle\rangle)} \qquad \text{(S11)}\ \frac{a \in \mathcal{A}}{(St, En) \vdash a \Rightarrow^{\mathtt{E}} (St, \langle a\rangle, \langle\rangle)}$$

$$\text{(S12)}\ \frac{}{(St, En) \vdash () \Rightarrow^{\mathtt{E}} (St, \langle\rangle, \langle\rangle)}$$

Sequence Concatenation (G8) The concatenation glues the results of two expressions together. Note that if the subexpressions contain no snap operations then the actual computation of the result can be done in parallel, since the modifications to the store done by e_1 cannot be observed by e_2.

$$\text{(S13)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, v_2, l_2)}{(St, En) \vdash e_1,\ e_2 \Rightarrow^{\mathtt{E}} (St_2, v_1 \circ v_2, l_1 \circ l_2)}$$

Binary Expressions on Atomic Values (G9-G10) The boolean binary operators `and` and `or` require both input sequences to have only one item, while the value comparison operators accept two sequences of arbitrary size and have an existential semantics.

$$\text{(S14)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle b_1\rangle, l_1) \qquad (St', En) \vdash e_2 \Rightarrow (St_2, \langle b_2\rangle, l_2) \qquad b_1, b_2 \in \mathcal{B}}{\begin{array}{c}(St, En) \vdash e_1\ \mathtt{and}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle b_1 \wedge b_2\rangle, l_1 \circ l_2) \\ (St, En) \vdash e_1\ \mathtt{or}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle b_1 \vee b_2\rangle, l_1 \circ l_2)\end{array}}$$

$$\text{(S15)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle x_1, \ldots, x_m\rangle, l_1) \\ x_1, \ldots, x_m \in \mathcal{A} \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle x'_1, \ldots, x'_{m'}\rangle, l_2) \\ x'_1, \ldots, x'_{m'} \in \mathcal{A} \qquad b_= \Leftrightarrow \exists_{1\le i\le m, 1\le j\le m'}(x_i = x'_j) \qquad b_< \Leftrightarrow \exists_{1\le i\le m, 1\le j\le m'}(x_i < x'_j)\end{array}}{(St, En) \vdash e_1 = e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle b_=\rangle, l_1 \circ l_2) \qquad (St, En) \vdash e_1 < e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle b_<\rangle, l_1 \circ l_2)}$$

Arithmetic (G11) The semantics of arithmetic operations is straightforward. Note that `idiv` denotes the integer division.

$$\text{(S16)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle d_1 \rangle, l_1) \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle d_2 \rangle, l_2) \qquad d_1, d_2 \in \mathcal{I}}{\begin{array}{c}(St, En) \vdash e_1 + e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle d_1 + d_2 \rangle, l_1 \circ l_2) \\ (St, En) \vdash e_1 - e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle d_1 - d_2 \rangle, l_1 \circ l_2) \\ (St, En) \vdash e_1 * e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle d_1 \times d_2 \rangle, l_1 \circ l_2) \\ (St, En) \vdash e_1 \ \mathtt{idiv}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle d_1 / d_2 \rangle, l_1 \circ l_2)\end{array}}$$

If-expression (G12) The semantics of the if-expression is given by two inference rules: one for the case the condition evaluates to **true** and one for **false**. Note that in each case only one of the branches is executed and hence only updates of one branch are added to the list of pending updates.

$$\text{(S17)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle \mathbf{true} \rangle, l) \qquad (St', En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1)}{(St, En) \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Rightarrow^{\mathtt{E}} (St_1, v_1, l \circ l_1)}$$

$$\text{(S18)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle \mathbf{false} \rangle, l) \qquad (St', En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, v_2, l_2)}{(St, En) \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, v_2, l \circ l_2)}$$

For-expression (G13) The rule for `for $`s `at $`s' `in` e `return` e' specifies that first e is evaluated and then e' for each item in the result of e but with s and s' in the environment bound to the respectively the item in question and its position in the result of e. Finally the results for each item are concatenated to a single sequence.

$$\text{(S19)}\ \frac{\begin{array}{c}(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St_0, \langle x_1, \ldots, x_m \rangle, l) \qquad (St_0, En[\mathbf{v}(s) \mapsto x_1]) \vdash e' \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \\ \ldots \qquad (St_{m-1}, En[\mathbf{v}(s) \mapsto x_m]) \vdash e' \Rightarrow^{\mathtt{E}} (St_m, v_m, l_m)\end{array}}{(St, En) \vdash \mathtt{for}\ \$s\ \mathtt{in}\ e\ \mathtt{return}\ e' \Rightarrow^{\mathtt{E}} (St_m, v_1 \circ \ldots \circ v_m, l \circ l_1 \circ \ldots \circ l_m)}$$

$$\text{(S20)}\ \frac{\begin{array}{c}(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St_0, \langle x_1, \ldots, x_m \rangle, l) \\ (St_0, En[\mathbf{v}(s) \mapsto x_1][\mathbf{v}(s') \mapsto 1]) \vdash e' \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \\ \ldots \qquad (St_{m-1}, En[\mathbf{v}(s) \mapsto x_m][\mathbf{v}(s') \mapsto m]) \vdash e' \Rightarrow^{\mathtt{E}} (St_m, v_m, l_m)\end{array}}{(St, En) \vdash \mathtt{for}\ \$s\ \mathtt{at}\ \$s'\ \mathtt{in}\ e\ \mathtt{return}\ e' \Rightarrow^{\mathtt{E}} (St_m, v_1 \circ \ldots \circ v_m, l \circ l_1 \circ \ldots \circ l_m)}$$

Let-expression (G14) The `let` expression evaluates the subexpression on the righthand side and binds the result to a variable. Note that even if $\$x$ does not occur in e_2 then we still have to evaluate e_1 since the resulting list of pending updates l_1 has to be added to the result list of pending updates of the `let` expression.

$$\text{(S21)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \qquad (St_1, En[\mathbf{v}(s) \mapsto v_1]) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, v_2, l_2)}{(St, En) \vdash \mathtt{let}\ \$s := e_1\ \mathtt{return}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, v_2, l_1 \circ l_2)}$$

Node Comparison (G15) Given two nodes, we can compare their position in the document order of the store. Note that these operations do not have an existential semantics, as oposed to value comparisons of rule (G9-G10).

$$\text{(S22)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle n_1 \rangle, l_1) \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle n_2 \rangle, l_2) \\ n_1, n_2 \in \mathcal{V} \qquad b_{\mathtt{is}} \Leftrightarrow (n_1 = n_2) \qquad b_{\ll} \Leftrightarrow (n_1 \ll_{St_2} n_2)\end{array}}{\begin{array}{ll}(St, En) \vdash e_1\ \mathtt{is}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle b_{\mathtt{is}} \rangle, l_1 \circ l_2) & (St, En) \vdash e_1 << e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle b_{\ll} \rangle, l_1 \circ l_2)\end{array}}$$

Path Expressions (G16-G17) The semantics of a step consisting of an element name s is that all element children of the context node (indicated in the environment by $\mathbf{x}$) with name s are returned in document order. The semantics of the step consisting of the wild-card $*$ is the same except that all element children of the context node are returned.

$$\text{(S23)}\ \frac{\mathbf{x}_{En}\ is\ defined}{(St, En) \vdash . \Rightarrow^{\mathtt{E}} (St, \langle \mathbf{x}_{En} \rangle, \langle\rangle)} \qquad \text{(S24)}\ \frac{(n, \mathbf{x}_{En}) \in E_{St}}{(St, En) \vdash .. \Rightarrow^{\mathtt{E}} (St, \langle n \rangle, \langle\rangle)}$$

$$\text{(S25)}\ \frac{\nexists n : (n, \mathbf{x}_{En}) \in E_{St}}{(St, En) \vdash .. \Rightarrow^{\mathtt{E}} (St, \langle\rangle, \langle\rangle)} \qquad \text{(S26)}\ \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^e \wedge \nu_{St}(n) = s\}}{(St, En) \vdash s \Rightarrow^{\mathtt{E}} (St, \mathbf{Ord}_{St}(W), \langle\rangle)}$$

$$\text{(S27)}\ \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^a \wedge \nu_{St}(n) = s\}}{(St, En) \vdash @s \Rightarrow^{\mathtt{E}} (St, \mathbf{Ord}_{St}(W), \langle\rangle)}$$

$$\text{(S28)}\ \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^e\}}{(St, En) \vdash * \Rightarrow^{\mathtt{E}} (St, \mathbf{Ord}_{St}(W), \langle\rangle)} \qquad \text{(S29)}\ \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^a\}}{(St, En) \vdash @* \Rightarrow^{\mathtt{E}} (St, \mathbf{Ord}_{St}(W), \langle\rangle)}$$

$$\text{(S30)}\ \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^t\}}{(St, En) \vdash \mathtt{text()} \Rightarrow (St, \mathbf{Ord}_{St}(W), \langle\rangle)}$$

The semantics of (e_1 `/` e_2) is as follows. First e_1 is evaluated. Then for each item in its result we bind in the environment $\mathbf{x}$ to this item, and with this environment we evaluate

e_2. The results of all these evaluations are concatenated and finally this sequence is sorted by document order and the duplicates are removed. The result is only defined if all the evaluations of e_2 contain only nodes.

$$\text{(S31)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_0, \langle x_1, \ldots, x_m\rangle, l_0) \qquad (St_0, En[\mathbf{x} \mapsto x_1]) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \quad \ldots \quad (St_{m-1}, En[\mathbf{x} \mapsto x_m]) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_m, v_m, l_m) \qquad v_1, \ldots, v_m \in \mathcal{V}^*}{(St, En) \vdash e_1 \ / \ e_2 \Rightarrow^{\mathtt{E}} (St_m, \mathbf{Ord}_{St_m}(\cup_{1\leq i\leq m}\mathbf{Set}(v_i)), l_0 \circ l_1 \circ \ldots \circ l_m)}$$

$$\text{(S32)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_0, \langle x_1, \ldots, x_m\rangle, l_0) \\ W_1 = \{x \in V_{St_0} | (x_1, x) \in (E_{St_0})^*\} \qquad \ldots \qquad W_m = \{x \in V_{St_0} | (x_m, x) \in (E_{St_0})^*\} \\ \langle x'_1, \ldots, x'_{m'}\rangle = \mathbf{Ord}_{St_0}(\cup_{1\leq i\leq m} W_i) \qquad (St_0, En[\mathbf{x} \mapsto x'_1]) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \\ \ldots \qquad (St_{m-1}, En[\mathbf{x} \mapsto x'_{m'}]) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_{m'}, v_{m'}, l_{m'}) \qquad v_1, \ldots, v_{m'} \in \mathcal{V}^*\end{array}}{(St, En) \vdash e_1 \ // \ e_2 \Rightarrow^{\mathtt{E}} (St_{m'}, \mathbf{Ord}_{St_{m'}}(\cup_{1\leq i\leq m'}\mathbf{Set}(v_i)), l_0 \circ l_1 \circ \ldots \circ l_{m'})}$$

Typeswitch-expression (G18-G19) The following semantics is assumed for types: $[\![\mathtt{xs:boolean}]\!] = \mathcal{B}$, $[\![\mathtt{xs:integer}]\!] = \mathcal{I}$, $[\![\mathtt{xs:string}]\!] = \mathcal{S}$, $[\![\mathtt{document\text{-}node()}]\!] = \mathcal{V}^d$, $[\![\mathtt{attribute()}]\!] = \mathcal{V}^a$, $[\![\mathtt{text()}]\!] = \mathcal{V}^t$ and $[\![\mathtt{element()}]\!] = \mathcal{V}^e$.

$$\text{(S33)}\ \frac{\begin{array}{c}(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St_1, \langle x\rangle, l) \\ (x \in [\![t_j]\!] \vee j = m+1) \qquad \forall_{1\leq i<j}(x \notin [\![t_i]\!]) \qquad St_1, En \vdash e_j \Rightarrow^{\mathtt{E}} (St_2, v, l')\end{array}}{\begin{array}{c}(St, En) \vdash \mathtt{typeswitch}(e)\ \mathtt{case}\ t_1\ \mathtt{return}\ e_1 \ldots\ \mathtt{case}\ t_m\ \mathtt{return}\ e_m \\ \mathtt{default\ return}\ e_{m+1} \Rightarrow^{\mathtt{E}} (St_2, v, l \circ l')\end{array}}$$

Functions (G20-G21) We now give the semantics of the built-in functions.

$$\text{(S34)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle s\rangle, l) \qquad \delta_{St'}(s) = n}{(St, En) \vdash \mathtt{doc}(e) \Rightarrow (St', n, l)}$$

$$\text{(S35)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle n\rangle, l) \qquad n \in \mathcal{V}^e \cup \mathcal{V}^a}{(St, En) \vdash \mathtt{name}(e) \Rightarrow^{\mathtt{E}} (St', \langle \nu_{St'}(n)\rangle, l)}$$

$$\text{(S36)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle n\rangle, l) \qquad n \in \mathcal{V}^a \cup \mathcal{V}^t}{(St, En) \vdash \mathtt{string}(e) \Rightarrow^{\mathtt{E}} (St', \langle \sigma_{St'}(n)\rangle, l)}$$

$$\text{(S37)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle x\rangle, l) \qquad x \in \mathcal{A} \qquad \mathit{AtValueToString}(x) = s}{(St, En) \vdash \mathtt{string}(e) \Rightarrow^{\mathtt{E}} (St', \langle s\rangle)}$$

$$\text{(S38)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle s\rangle, l) \qquad s \in \mathcal{S} \qquad \mathit{StringToInteger}(s) = i}{(St, En) \vdash \mathtt{xs:integer}(e) \Rightarrow^{\mathtt{E}} (St', \langle i\rangle, l)}$$

$$\text{(S39)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle n \rangle, l) \qquad m \in V_{St'} \qquad (m, n) \in E^*_{St'} \qquad \nexists n' : (n', m) \in E^+_{St'}}{(St, En) \vdash \mathtt{root}(e) \Rightarrow^{\mathtt{E}} (St', \langle m \rangle, l)}$$

$$\text{(S40)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle s_1 \rangle, l_1) \qquad s_1 \in \mathcal{S} \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle s_2 \rangle, l_2) \qquad s_2 \in \mathcal{S}}{(St, En) \vdash \mathtt{concat}(e_1, e_2) \Rightarrow^{\mathtt{E}} (St_2, \langle s_1 \cdot s_2 \rangle, l_1 \circ l_2)}$$

$$\text{(S41)}\ \frac{}{(St, En) \vdash \mathtt{true}() \Rightarrow^{\mathtt{E}} (St, \langle \mathbf{true} \rangle, \langle\rangle)} \qquad \text{(S42)}\ \frac{}{(St, En) \vdash \mathtt{false}() \Rightarrow^{\mathtt{E}} (St, \langle \mathbf{false} \rangle, \langle\rangle)}$$

$$\text{(S43)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle b \rangle, l) \qquad b \in \mathcal{B}}{(St, En) \vdash \mathtt{not}(e) \Rightarrow^{\mathtt{E}} (St', \langle \neg b \rangle, l)} \qquad \text{(S44)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', \langle x_1, \ldots, x_m \rangle, l)}{(St, En) \vdash \mathtt{count}(e) \Rightarrow^{\mathtt{E}} (St', \langle m \rangle, l)}$$

For user-defined functions, we introduce the following semantics.

$$\text{(S45)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, v_1, l_1) \\ \ldots \qquad (St_{m-1}, En) \vdash e_m \Rightarrow^{\mathtt{E}} (St_m, v_m, l_m) \qquad En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x}) \qquad \mathbf{a}(f) = \langle s_1, \ldots, s_m \rangle \\ En' = (\mathbf{a}, \mathbf{b}, \{(s_1, v_1), \ldots, (s_m, v_m)\}, \bot) \qquad (St_m, En') \vdash \mathbf{b}(f) \Rightarrow^{\mathtt{E}} (St', v', l')\end{array}}{(St, En) \vdash f(e_1, \ldots, e_m) \Rightarrow^{\mathtt{E}} (St', v', l_1 \circ \ldots \circ l_m, l')}$$

Sequence Generation (G22) The sequence generation operation computes a sequence of increasing integers and is defined as follows.

$$\text{(S46)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle d_1 \rangle, l_1) \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle d_2 \rangle, l_2) \\ d_1, d_2 \in \mathcal{I} m = d_2 - d_1 + 1 \qquad m \geq 1 \qquad x_i = d_1 + (i - 1)\end{array}}{(St, En) \vdash e_1\ \mathtt{to}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle x_1, \ldots, x_m \rangle, l_1 \circ l_2)}$$

Node Construction (G23) The semantics of the element constructor `element` $\{e_1\}$ $\{e_2\}$ is defined as follows. First e_1 is evaluated and assumed to result in a single legal element name. Then e_2 is evaluated and for the result we create a new store St_3 that contains the new element with the result of e_1 as its name and with contents that are deep-equivalent with the result of e_2 if we compare them item by item. Finally we add St_3 to the original store and return the newly created element node.

$$\text{(S47)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle s\rangle, l_1) \qquad s \in \mathcal{N} \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle n_1, \ldots, n_m\rangle, l_2) \\ n_1, \ldots, n_m \in \mathcal{V} \qquad St_4 = St_2 \cup St_3 \qquad n \in V_{St_3} \Rightarrow (r, n) \in E^*_{St_3} \qquad r \in \mathcal{V}^e \\ \nu_{St_3}(r) = s \qquad \mathbf{Ord}_{St_3}(\{n' | (r, n') \in E_{St_3}\}) = \langle n'_1, \ldots, n'_m\rangle \qquad \mathbf{DpEq}_{St_4}(n_1, n'_1) \\ \ldots \qquad \mathbf{DpEq}_{St_4}(n_m, n'_m) \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n'))\end{array}}{(St, En) \vdash \mathtt{element}\{e_1\}\{e_2\} \Rightarrow^{\mathtt{E}} (St_4, \langle r\rangle, l_1 \circ l_2)}$$

$$\text{(S48)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle s\rangle, l_1) \\ s \in \mathcal{N} \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle s'\rangle, l_2) \qquad s' \in \mathcal{S} \qquad St_4 = St_2 \cup St_3 \qquad V_{St_3} = \{r\} \\ r \in \mathcal{V}^a \qquad \nu_{St_3}(r) = s \qquad \sigma_{St_3}(r) = s' \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n'))\end{array}}{(St, En) \vdash \mathtt{attribute}\{e_1\}\{e_2\} \Rightarrow^{\mathtt{E}} (St_4, \langle r\rangle, l_1 \circ l_2)}$$

$$\text{(S49)}\ \frac{\begin{array}{c}(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St_1, \langle s\rangle, l) \qquad s \in \mathcal{S} - \{\text{“”}\} \qquad St_3 = St_1 \cup St_2 \\ V_{St_2} = \{r\} \qquad r \in \mathcal{V}^t \qquad \sigma_{St_2}(r) = s \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_1} n') \Rightarrow (n \ll_{St_3} n'))\end{array}}{(St, En) \vdash \mathtt{text}\{e\} \Rightarrow^{\mathtt{E}} (St_3, \langle r\rangle, l)}$$

$$\text{(S50)}\ \frac{\begin{array}{c}(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St_1, \langle n_1\rangle, l) \\ n_1 \in \mathcal{V}^e \qquad St_3 = St_1 \cup St_2 \qquad n \in V_{St_2} \Rightarrow (r, n) \in E^*_{St_2} \qquad r \in \mathcal{V}^d \\ (r, n_2) \in E_{St_2} \qquad \mathbf{DpEq}_{St_3}(n_1, n_2) \qquad \forall\, n, n' \in \mathcal{V}((n \ll_{St_1} n') \Rightarrow (n \ll_{St_3} n'))\end{array}}{(St, En) \vdash \mathtt{document}\{e\} \Rightarrow^{\mathtt{E}} (St_3, \langle r\rangle, l)}$$

Basic Update Expressions (G24-G27) The `delete` results into a set of pending updates which will delete the incoming edges of the selected nodes. The `rename` and `replace value` expressions evaluate two subexpressions which have to result in respectively one node and one string value. Similar to the `delete` expression we add new primitive operation to the list of pending updates. An `insert` expression makes a copy of the nodes that are selected by the first subexpression and puts these copies at a certain place w.r.t. the node that is returned by the second expression. The position is indicated by either `before`, `after`, or `into`. In case of insertion into a node n, the relative place of the copied nodes among the children of n is chosen arbitrarily, but the relative order of the copies has to be preserved.

$$\text{(S51)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St_1, \langle n_1, \ldots, n_m\rangle, l)}{(St, En) \vdash \mathtt{delete}\ e \Rightarrow^{\mathtt{E}} (St_1, \langle\rangle, l \circ \langle del(n_1), \ldots, del(n_m)\rangle)}$$

$$\text{(S52)}\ \frac{(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle n \rangle, l_1) \qquad (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle s \rangle, l_2)}{\begin{array}{c}(St, En) \vdash \texttt{rename}\ e_1\ \texttt{as}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle\rangle, l_1 \circ l_2 \circ \langle ren(n, s) \rangle) \\ (St, En) \vdash \texttt{replace value of}\ e_1\ \texttt{with}\ e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle\rangle, l_1 \circ l_2 \circ \langle repVal(n, s) \rangle)\end{array}}$$

$$\text{(S53)}\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle n \rangle, l_1) \\ (St_1, En) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, \langle n_1, \ldots, n_m \rangle, l_2) \qquad St' = St_2 \cup St'_1 \cup \ldots \cup St'_m \\ \mathbf{DpEq}_{St'}(n_1, n'_1)\ \ldots\ \mathbf{DpEq}_{St'}(n_m, n'_m) \qquad V^{n'_1}_{St'_1} = V_{St'_1}\ \ldots\ V^{n'_m}_{St'_m} = V_{St'_m}\end{array}}{\begin{array}{c}(St, En) \vdash \texttt{insert}\ e_2\ \texttt{into}\ e_1 \Rightarrow^{\mathtt{E}} (St_3, \langle\rangle, l_1 \circ l_2 \circ \langle insInto(n, \langle n'_1, \ldots, n'_m \rangle) \rangle) \\ (St, En) \vdash \texttt{insert}\ e_2\ \texttt{before}\ e_1 \Rightarrow^{\mathtt{E}} (St_3, \langle\rangle, l_1 \circ l_2 \circ \langle insBef(n, \langle n'_1, \ldots, n'_m \rangle) \rangle) \\ (St, En) \vdash \texttt{insert}\ e_2\ \texttt{after}\ e_1 \Rightarrow^{\mathtt{E}} (St_3, \langle\rangle, l_1 \circ l_2 \circ \langle insAft(n, \langle n'_1, \ldots, n'_m \rangle) \rangle)\end{array}}$$

Snap Expression (G28) The snap operation comes in three different flavours: ordered, unordered deterministic and unordered nondeterministic. The ordered mode specifies that the pending updates have to be applied in the same order as they were generated, the unordered deterministic mode requires that the list of pending updates has to be execution-order independent, while the unordered nondeterministic mode applies the pending updates in an arbitrary order.

$$\text{(S54)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l) \qquad St' \vdash l \Rightarrow^{\mathtt{U}} St''}{(St, En) \vdash \texttt{snap ordered}\ \{\ e\ \}\ \Rightarrow^{\mathtt{E}} (St'', v, \langle\rangle)}$$

$$\text{(S55)}\ \frac{(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l) \qquad \mathbf{Bag}(l) = \mathbf{Bag}(l') \qquad St' \vdash l' \Rightarrow^{\mathtt{U}} St''}{(St, En) \vdash \texttt{snap unordered nondeterministic}\ \{\ e\ \}\ \Rightarrow^{\mathtt{E}} (St'', v, \langle\rangle)}$$

$$\text{(S56)}\ \frac{\begin{array}{c}(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l) \\ l\ \textit{is execution-order independent} \qquad \mathbf{Bag}(l) = \mathbf{Bag}(l') \qquad St' \vdash l' \Rightarrow^{\mathtt{U}} St''\end{array}}{(St, En) \vdash \texttt{snap unordered deterministic}\ \{\ e\ \}\ \Rightarrow^{\mathtt{E}} (St'', v, \langle\rangle)}$$

For every LiXQuery expression, it can be verified that the resulting list of pending updates is always well-formed and hence we can apply Proposition 2.1 to decide whether l is execution-order independent.

Transform Expressions (G29) The transform expression first evaluates the first subexpression which should result in a sequence of nodes. Then it makes deep copies of each of these nodes, placed relatively in document order as the original nodes were ordered in the result sequence. The second subexpression is evaluated with the variable bound to the deep-copied nodes, and if the resulting list of pending updates only affects nodes in the deep copies then these are applied to the store and the last subexpression is evaluated.

$$(\text{S57})\ \frac{\begin{array}{c}(St, En) \vdash e_1 \Rightarrow^{\mathtt{E}} (St_1, \langle n_1, \ldots, n_m\rangle, \langle\rangle) \\ St'_1 = St_1 \cup St_{1,1} \cup \ldots \cup St_{1,m} \qquad \mathbf{DpEq}_{St'_1}(n_1, n'_1)\ \ldots\ \mathbf{DpEq}_{St'_1}(n_m, n'_m) \\ V^{n'_1}_{St_{1,1}} = V_{St_{1,1}}\ \ldots\ V^{n'_m}_{St_{1,m}} = V_{St_{1,m}} \qquad n'_1 \ll_{St'_1} n'_2 \ll_{St'_1} \ldots \ll_{St'_1} n'_m \\ En_1 = En[\mathbf{v}(s) \mapsto \langle n'_1, \ldots, n'_m\rangle] \qquad (St'_1, En_1) \vdash e_2 \Rightarrow^{\mathtt{E}} (St_2, v, l) \\ \mathbf{Targets}(l) \subseteq V_{St'_1} - V_{St_1} \qquad St_2 \vdash l \Rightarrow^{\mathtt{U}} St'_2 \qquad (St'_2, En_1) \vdash e_3 \Rightarrow^{\mathtt{E}} (St_3, v', \langle\rangle)\end{array}}{(St, En) \vdash \texttt{transform copy } \$s := e_1 \texttt{ modify } e_2 \texttt{ return } e_3 \Rightarrow^{\mathtt{E}} (St_3, v', \langle\rangle)}$$

Note that we require that the evaluation of e_3 returns an empty list of pending updates, since `transform` expressions are only to be used for creating a modified copy of a node, not for updating other parts of the store.

2.5 Conclusion

This chapter introduced the LiXQuery language for querying and updating XML fragments. The design of LiXQuery is strongly based on the XQuery W3C Recommendation [Boag et al., 2007a] and the XQuery Update Facility (XQUF) Working Draft [Chamberlin et al., 2007]. More precisely, we claim that is is related to these standards in the following way:

- For LiXQuery expressions that do not contain updating subexpressions and that are syntactically in XQuery, it holds that the semantics is downwards compatible with the XQuery semantics, i.e., if we obtain a certain result sequence in LiXQuery then we can also obtain the same result sequence in XQuery.

- We allow to fully mix updating and non-updating operations, while this is restricted in XQUF.

- The semantics of our updating expressions is also a subset of the corresponding XQuery expressions. For example, in LiXQuery `replace value` is not defined when it is applied on an element node, while in XQUF the value of the single text node within that element node is replaced, if applicable.

- We added the `snap` operator, as introduced in [Ghelli et al., 2006], to identify when to apply pending updates. In Chapter 3 we will see that this construct adds expressive power when we discard recursive function definitions.

- The `unordered deterministic` mode of the `snap` corresponds somewhat to the behavior of the list of pending updates in the XQUF, while the default behavior for LiXQuery is the `ordered` mode. More precisely, in XQUF there is first a check for conflicts to see whether there are `rename` or `replaceValue` operations with the same target node and then the list of pending updates is applied in a semi-ordered fashion, i.e., every pending update is classified into one of several categories, based on their

primitive update operation, and then the updates are applied category by category while applying the updates within one category in the same order as they were generated. Note that in an older version [Chamberlin et al., 2006] there were checks for all conflicts, which in fact corresponds exactly to our `unordered deterministic` mode.

We claim that LiXQuery captures the essence of XQuery as a query language and can therefore be used for educational purposes, e.g., teaching XQuery, and research purposes, e.g., investigating the expressive power of XQuery fragments, whereas the full LiXQuery language can be used to improve the understanding of the relative expressive power of constructs in XQuery-based updating languages.

CHAPTER 3

Expressing Transformations in XQuery Fragments

The XQuery-based language introduced in the previous chapter is a powerful language, as is hinted by the fact that it is Turing-complete. However, this does not mean that we can express all computable mappings from stores to stores in LiXQuery. For example, we cannot move a node to another place in a tree while preserving its identity. Moreover, some constructs seem to add expressive power to the language while others appear to be merely syntactic sugar. In this chapter we compare the relative expressive power of some key constructs of the LiXQuery language in terms of expressing forest transformations by studying LiXQuery fragments defined by the presence or absence of these constructs.

3.1 LiXQuery Fragments

We now define the constructs and fragments for which we will study the expressive power in this chapter. We also give some motivation for our choices of fragments. It can be easily verified that the restrictions used in this chapter, can also be obtained by changing the grammar rules of Figure 2.1. For example, removing semantic rule (S20) changes grammar rule (G13) as follows: ⟨*ForExpr*⟩ ::= "`for`"⟨*Var*⟩ "`in`"⟨*Expr*⟩ "`return`" ⟨*Expr*⟩.

First, we define the following four base fragments of LiXQuery:

- The fragment XQ corresponds to non-recursive XQuery without node construction or count aggregation, i.e., rules (S1-S7, S20, S44-S45,S47-S57) are omitted from LiXQuery. In this language we can only select nodes from the input store and return computed atomic values.

- The fragment XQ^{constr} corresponds to non-recursive XQuery without count aggregation, i.e., rules (S1-S7, S20, S44-S45, S51-S57) are omitted from LiXQuery. In this

language we can construct new nodes and make deep-copies of nodes from the input store.

- The fragment XQ^{upd} corresponds to non-recursive XQuery extended with the update operations, but without the `snap` operation. More precisely, rules (S7, S20, S44-S45, S54-S56) are omitted from LiXQuery. This language corresponds to a subset of the XQuery update facility, however there are some subtle differences, which will be discussed in Section 5.1.

- The fragment XQ^{snap} corresponds to non-recursive XQuery extended with updates and `snap` operations, i.e., rules (S7, S20, S44-S45) are omitted from LiXQuery. The addition of this operation makes it possible to have `for` loops with side effects and the `snap` operation in this fragment indicates the correspondence to XQuery! [Ghelli et al., 2006].

Each of these fragments can be extended by combinations of the following three constructs. The addition of these constructs will be denoted by adding one or more subscripts to the name of the fragment.

- Extending a fragment with the count aggregation function corresponds to adding rule (S44) and is denoted by a subscript C. This construct is made optional since, although in practice it is often used, the added expressive power is often not easy to establish. See for example [Libkin, 2003] for a discussion of this for SQL.

- Extending a fragment with the position information in a `for` loop corresponds to adding rule (S20) and is denoted by a subscript at. This construct is made optional since it seems to be the only construct that can refer to the position of an item in a sequence.

- Extending a fragment with (recursive) function definitions corresponds to adding rules (S7,S45) and is denoted by a subscript R. Observe that if we disallow recursion these constructs would not increase the expressive power of the language since all function calls could be in-lined. Adding recursion will obviously greatly increase the expressive power and make the language computationally complete for operations over strings and integers but, as will be shown later, there may still be operations over XML data and/or sequences that cannot be expressed. For example, XQ_R cannot express the `count()` function, which can be informally explained by saying that there is no operation in XQ_R that can distinguish sequences that contain the same elements. There are for example no operations in XQ that allow us to select the head or the tail of a list, as is possible in LISP. Clearly with such operations the `count()` could have been expressed with recursive functions. This will be discussed more formally later on.

We have now defined 32 sublanguages of LiXQuery. For example, $XQ^{constr}_{C,R}$ denotes the language corresponding to XQuery without the `at` clause in `for` expressions, XQ_{at}

denotes non-recursive XQuery without node construction, count aggregation function but with position information in `for` expressions, and finally $XQ^{snap}_{at,C,R}$ corresponds to full LiXQuery. The language $\mathbf{L}(XF)$ of a LiXQuery fragment XF is the (infinite) set of all expressions that can be generated by the syntax rules for this fragment with $\langle Program \rangle$ as start symbol. The set Φ is the set of all 32 LiXQuery fragments.

3.2 Expressiveness Relationships between Fragments

We study expressiveness in terms of the ability to perform certain transformations and say that two programs express the same transformation if they map the same input (store and variable assignment) to deep-equal result sequences. The expressive power of a LiXQuery fragment is defined as the *transformations* that can be expressed in this fragment. A transformation is the mapping of a store and an environment to a serialized result sequence. We now define when a program simulates another program and when two programs are equivalent.

Definition 3.1 (Program Simulation and Equivalence)**.** *Consider two LiXQuery programs e_1 and e_2. Program e_1* simulates *e_2, denoted by $e_1 \rightsquigarrow e_2$, if for every store St and environment $En = (\emptyset, \emptyset, \mathbf{v}, \bot)$ over St it holds that $(St, En) \vdash e_1 \Rightarrow (St', \langle i_1, \ldots, i_k \rangle)$ implies that there exists a store St'', a value $v' = \langle i'_1, \ldots, i'_k \rangle$ over St'', and an evaluation $(St, En) \vdash e_2 \Rightarrow (St'', \langle i'_1, \ldots, i'_k \rangle)$ such that for every j from 1 to k it holds that either*

- *i_j and i'_j are atomic values and $i_j = i'_j$, or*
- *i_j and i'_j are nodes and $\mathbf{DpEq}_{St',St''}(i_j, i'_j)$.*

Programs e_1 and e_2 are equivalent*, denoted by $e_1 \sim e_2$, if $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_1$.*

This measure of expressive power can be justified by the XQuery Processing Model [Draper et al., 2006]. There it is possible to set variables in an initial environment. Moreover, we assume that the (optional) serialization phase is applied at the end so deep-equal results cannot be distinguished.

Definition 3.2 (Equivalent Fragments)**.** *Recall that Φ is the set of XQuery fragments as defined in Section 3.1. Consider two XQuery fragments XF_1, $XF_2 \in \Phi$.*

- $XF_1 \preceq XF_2 \iff \forall e_1 \in \mathbf{L}(XF_1) : \exists e_2 \in \mathbf{L}(XF_2) : e_1 \sim e_2$
  (XF_2 can simulate XF_1)
- $XF_1 \equiv XF_2 \iff ((XF_1 \preceq XF_2) \wedge (XF_2 \preceq XF_1))$
  (XF_1 is equivalent to XF_2)
- $XF_1 \prec XF_2 \iff ((XF_1 \preceq XF_2) \wedge (XF_1 \not\equiv XF_2))$
  (XF_1 is less expressive than XF_2)

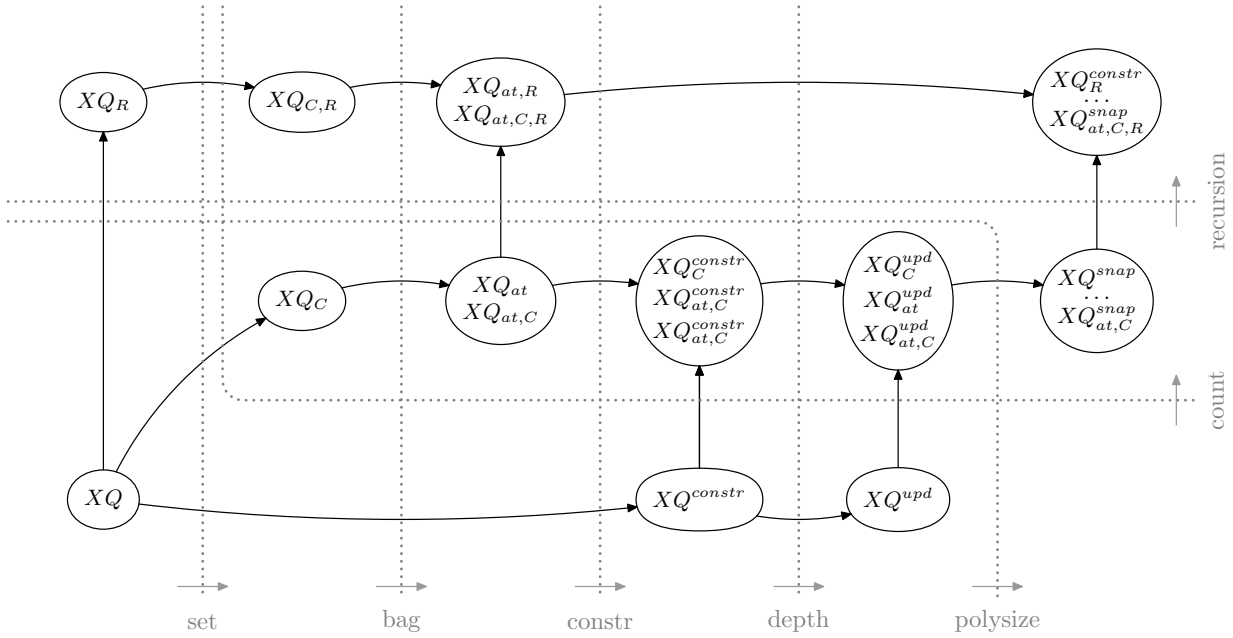


Figure 3.1: Equivalence classes of XQuery fragments

In this definition, the relation $\preceq$ is a partial order on Φ, and $\equiv$ is an equivalence relation on Φ.

Note that the four base fragments can be ordered in such a way that every expression in the smaller fragment is also an expression in the larger fragment and hence the larger fragments can simulate the smaller fragments. More precisely, $XQ \preceq XQ^{constr} \preceq XQ^{upd} \preceq XQ^{snap}$.

We use these relations to investigate the relationships between all LiXQuery fragments defined in Section 3.1. We show that the equivalence relation $\equiv$ partitions Φ (containing 32 fragments) into 12 equivalence classes. In Figure 3.1 we show these 12 equivalence classes and their relationships. Each node of the graph represents an equivalence class, i.e., a class of LiXQuery fragments with the same expressive power. Each edge is directed from a less expressive class C_1 to a more expressive one C_2 and points out that each fragment in C_1 is less expressive than all fragments of C_2, i.e., for every $XF_1 \in C_1$ and $XF_2 \in C_2$ it holds that $XF_1 \prec XF_2$.

Theorem 3.1. *For the graph in Figure 3.1 and for all fragments $XF_1, XF_2 \in \Phi$ it holds that*

- $XF_1 \equiv XF_2 \iff$ *XF_1 and XF_2 are within the same node*
- $XF_1 \prec XF_2 \iff$ *there is a directed path from the node containing XF_1 to the node containing XF_2*

The proof of this theorem is given in Section 3.5. The lemmas of Section 3.3 will be used to show that all different nodes in the graph have a different degree of expressive power and the lemmas of Section 3.4 to show that all fragments that are in the same node have the same expressive power.

Informally, the dotted borders in Figure 3.1 divide the set of fragments (Φ) in two parts and the arrows that cross the borders all go in one direction, i.e., from the part in which we cannot express a certain feature to the part that can express this feature. The fragments that are in the part from which the arrows start is called the left-hand side and the other half is called the right-hand side of the border. The correctness of the dotted borders is proven by showing that something can be expressed in the least expressive fragments of the right-hand side that cannot be expressed in any of the most expressive fragments of the left-hand side. In the following two sections we give the necessary lemmas needed to complete this proof.

3.3 Properties of the Fragments

In this section we prove that certain properties hold for some fragments that do not hold for other fragments to show that they have different degrees of expressive power.

3.3.1 Reachable Substores from Input/Output

First, we give some additional notations. For functions $\alpha : A \to B$ we use $\text{dom}(\alpha)$ to denote the domain, i.e., the set of elements $a \in A$ for which $\alpha(a)$ is defined, and $\text{rng}(\alpha)$ for the range, i.e., the subset of the co-domain B containing the elements b for which there is an element $a \in A$ such that $\alpha(a) = b$.

The nodes reachable from a given store St and set of nodes $V \subseteq V_{St}$ are those that are in a tree for which there is either a node in V or that are reachable by a `document` call. More precisely, it is defined as $\mathbf{reach}(St, V) = \{n \mid \exists n_1 : (n_1, n) \in E^*_{St} \wedge ((n_1 \in \text{rng}(\delta_{St})) \vee (\exists n_2 \in V : (n_1, n_2) \in E^*_{St}))\}$. The set of nodes reachable from a given input store St and environment En over St is $\mathbf{reachIn}(St, En) = \mathbf{reach}(St, \bigcup_{s \in \text{dom}(\mathbf{v})} \mathbf{Set}(\mathbf{v}(s)))$. The set of nodes reachable from an output store St and sequence over this store St is $\mathbf{reachOut}(St, v) = \mathbf{reach}(St, \{n \mid n \in V_{St} \wedge n \in \mathbf{Set}(v)\})$

This notion is used for several properties. If the expression of the `return` clause of a `for` expression does not contain a (possibly nested) `snap` operation then every iteration observes a part of the same input store, which is the output store of the `in` clause. Hence results of a previous iteration do not influence the result of the next iteration. More precisely, we obtain the following property:

Property 3.1. *Consider the expression* `for $x at $y in` e_1 `return` e_2 *where* e_2 *is an* $XQ^{upd}_{at,C,R}$ *expression, i.e.,* e_2 *does not contain a possibly nested* `snap` *expression, and an environment En for which all function bodies are* $XQ^{upd}_{at,C,R}$ *expressions. Suppose the evaluation of the* `in` *clause is as follows:* $(St, En) \vdash e_1 \Rightarrow^E (St_1, \langle x_1, \ldots, x_m \rangle, l_0)$. *Then if*

for every $i = 1, \ldots, m$ *it holds that* $(St_i, En_i) \vdash e_2 \Rightarrow^E (St_{i+1}, v_i, l_i)$ *where* En_i *denotes* $En[\mathbf{v}(x) \mapsto x_i][\mathbf{v}(y) \mapsto i]$*, then it also holds that* $St_i[\mathbf{reachIn}(St_i, En_i)]$ *is a substore of* St_1*.*

The correctness of this property can be easily verified by showing that for all pending update operations o that are generated during the evaluation of e_2 one of the two following holds:

- o is not yet applied and hence in one of the lists of pending updates l_i
- o is applied by a `transform` expression, but then the update is performed on a copy which is not accessible through a document call or a variable binding in En_i

This result will be used further on to prove properties for some fragments.

3.3.2 Set Equivalence and Bag Equivalence

The first two properties just claim that there are fragments in which it is not possible to distinguish between sequences with the same set or bag representation. To formalize this notion we define set equivalence and bag equivalence between environments and between sequences.

Definition 3.3. *Consider a store* St *and two environments* $En_1 = (\mathbf{a_1}, \mathbf{b_1}, \mathbf{v_1}, \mathbf{x_1})$ *and* $En' = (\mathbf{a_2}, \mathbf{b_2}, \mathbf{v_2}, \mathbf{x_2})$ *over the store* St*. We call* En_1 *and* En_2 set-equivalent *iff it holds that* $\mathbf{a_1} = \mathbf{a_2}$*,* $\mathbf{b_1} = \mathbf{b_2}$*,* $\mathrm{dom}(\mathbf{v_1}) = \mathrm{dom}(\mathbf{v_2})$ *and* $\forall s \in \mathrm{dom}(\mathbf{v_1}) : \mathbf{Set}(\mathbf{v_1}(s)) = \mathbf{Set}(\mathbf{v_2}(s))$*, and finally* $\mathbf{x_1} = \mathbf{x_2}$*. The environments* En_1 *and* En_2 *are called* bag-equivalent *iff they are set-equivalent and it holds that* $\forall s \in \mathrm{dom}(\mathbf{v_1}) : \mathbf{Bag}(\mathbf{v_1}(s)) = \mathbf{Bag}(\mathbf{v_2}(s))$*.*

To illustrate why we introduce this notion, consider the following example XQ program:

```
for $x in $seq return
  for $y in $seq return (if ($x > $y) then $x else ($x * $y))
```

If `$seq` is $\langle 1, 2 \rangle$ then the result is $\langle 1, 2, 2, 4 \rangle$, and if `$seq` is $\langle 2, 1, 2 \rangle$ then the result is $\langle 4, 2, 4, 2, 1, 2, 4, 2, 4 \rangle$. Both result values have the same set representation $\{1, 2, 4\}$. We now generalize this observation.

Lemma 3.1. *Let* St *be a store,* En_1, En_2 *two set-equivalent environments that have only function bodies which are* XQ_R *expressions, and* e *an expression in* XQ_R*. If the result of* e *is defined for both* En_1 *and* En_2*, then for each sequence* v_1 *and* v_2 *for which it holds that* $(St, En_1) \vdash e \Rightarrow^E (St, v_1, \langle\rangle)$ *and* $(St, En_2) \vdash e \Rightarrow^E (St, v_2, \langle\rangle)$[1]*, it also holds that* $\mathbf{Set}(v_1) = \mathbf{Set}(v_2)$*.*

[1]Since e does not contain node constructors in its subexpressions, it is easy to see that all subexpressions are evaluated against the same store St and that the result store of all these subexpressions will also be St.

Proof. This lemma is proven by induction on the derivation tree in which each node corresponds to one of the semantic rules (S10-S19,S21-S43,S45-S46) in Figure 3.1. Obviously, variables, literals, and empty sequences return sequences with the same set representation when evaluated against set-equivalent environments. If we apply a comparison between two sequences v_1 and v_2 then the result is the same as when applied to v_1' and v_2' if v_1' and v_2' have the same set representation as respectively v_1 and v_2. This is due to the existential semantics of the value comparison operators and the fact that node comparison operators are only defined for single nodes.

We now consider the `for` expression. By induction we know that the set of items i in the result sequence of the `in` clause of a `for` expression is the same when evaluated against set-equivalent environments En_1 and En_2. Let E_1 and E_2 denote the set of environments which contain all extensions of respectively En_1 and En_2 with a binding of the loop variable to an item in the result sequence of the evaluation of the in-clause against respectively En_1 and En_2. It can be shown that the relation "is a set-equivalent environment" between E_1 and E_2 is a total and surjective function, that is, for each evaluation against an environment En_1' in E_1 there is an evaluation against a set-equivalent environment En_2' in E_2 and vice versa. This follows intuitively from the fact that En_1' is En_1 but with the loop variable bound to a certain item i and En_2' is En_2 but with the loop variable also bound to i. By induction we then know that En_1' yields a result sequence with the same set representation as the result of En_2' and vice versa. Hence it follows that the concatenation of the result sequences of all evaluations yields result sequences with the same set representation.

In a similar way, we can show that all other expressions in this LiXQuery fragment also return sequences with the same set representation when applied to set-equivalent environments, since the result sequences of their subexpressions have the same set representation. Note however that for two set-related environments En_1 and En_2 it is possible that the evaluation of an expression e is defined for En_1 but not for En_2, e.g., consider `1 to $x` with $\mathbf{v}_1(x) = \langle 2 \rangle$ and $\mathbf{v}_2(x) = \langle 2, 2 \rangle$. □

The previous lemma is combined with the following lemma for proving that fragments on the left-hand side of the *count*-border in Figure 3.1 cannot express `count`.

Lemma 3.2. *The fragment XQ_C does not have the property of Lemma 3.1.*

Proof. Consider two set-equivalent environments $En_1 = (\emptyset, \emptyset, \{(\text{“}seq\text{”}, \langle 1, 1 \rangle)\}, \perp)$ and $En_2 = (\emptyset, \emptyset, \{(\text{“}seq\text{”}, \langle 1 \rangle)\}, \perp)$. The expression `count($seq)` returns $\langle 2 \rangle$ in the evaluation against En_1 and $\langle 1 \rangle$ against En_2. □

Note that the previous lemma implies that we cannot define full list equality in XQ_R. Similar to the result of Lemma 3.1, there is also a fragment in which we cannot distinguish between lists with the same bag representation. The following lemma states this more precisely.

Lemma 3.3. *Let St be a store, En_1, En_2 two bag-equivalent environments that have only function bodies which are $XQ_{C,R}$ expressions, and e be an expression in $XQ_{C,R}$. If the result of e is defined for both En_1 and En_2, then for each sequence v_1 and v_2 for which it*

holds that $(St, En_1) \vdash e \Rightarrow^E (St, v_1, \langle\rangle)$ *and* $(St, En_2) \vdash e \Rightarrow (St, v_2, \langle\rangle)$*, it also holds that* $\mathbf{Bag}(v_1) = \mathbf{Bag}(v_2)$.

Proof. For all $XQ_{C,R}$ expressions we can show similar to the proof of Lemma 3.1 that evaluations against bag-equivalent environments result in bag-equivalent result sequences. The only new feature is the `count()` function, which returns the same value when applied to sequences with the same bag representation. Moreover, we have to show for the `for` expression that there is a bijection between the sets E_1 and E_2, as defined in the proof of Lemma 3.1. □

The previous lemma is combined with the following lemma for proving that in fragments on the left-hand side of the *at*-border in Figure 3.1 we cannot simulate the `at` clause in `for` expressions.

Lemma 3.4. *The fragment* XQ_{at} *does not have the property of Lemma 3.3.*

Proof. Consider an environment $En_1 = (\emptyset, \emptyset, \{(\text{“}seq\text{”}, \langle 1, 2\rangle)\}, \bot)$ and another bag-equivalent environment $En_2 = (\emptyset, \emptyset, \{(\text{“}seq\text{”}, \langle 2, 1\rangle)\}, \bot)$. Both environments have no user-defined functions. Now, consider the following expression:

```
for $i at $pos in $seq
return if ($pos=1) then $i else ()
```

The evaluation of this expression returns $\langle 1\rangle$ when evaluated against environment En_1 and $\langle 2\rangle$ when evaluated against En_2. □

3.3.3 Relationships between Input and Output Values and Length

The maximum integer value or sequence length in the output for all programs in certain LiXQuery fragments can be identified as being bounded by a class of functions w.r.t. the input. For proving the inexpressibility results related to the input-output, we first introduce some auxiliary notations.

Let $St = (V, E, \ll, \nu, \sigma, \delta)$ be a store, $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ an environment over St and v a sequence over St. The set of integer values in a sequence v is defined as $I_v = \{i \mid (s \in \mathcal{S} \cap \mathbf{Set}(v) \wedge \mathit{StringToInteger}(s) = i) \vee (i \in \mathcal{I} \cap \mathbf{Set}(v))\}$, the set of integer values in a store St is $I^{St} = \{i \mid \exists s : \mathit{StringToInteger}(s) = i \wedge s \in \mathrm{rng}(\sigma)\}$, while the set of integer values in the environment En is $I^{En} = \bigcup_{v \in \mathrm{rng}(\mathbf{v})} I_v$. The number of nodes in the largest tree of the forest in St is $\Delta_{St}^{tree} = \max(\bigcup_{n_1 \in V}\{c \mid c = |\{n_2 \mid (n_1, n_2) \in E^*\}|\})$.[2] Finally, $\mathbf{abs}(i)$ denotes the absolute value of an integer i and we overload this notation for sets of integers, i.e., $\mathbf{abs}(I) = \{i \mid \exists i' \in I : i = \mathbf{abs}(i')\}$.

Definition 3.4 (Largest Integer Values and Sequence Lengths)**.** *Consider the evaluation* $(St_1, En) \vdash e \Rightarrow^E (St_2, v, l)$*, where* $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$*. Assume* $St_3 = St_1[\mathbf{reachIn}(St_1, En)]$,

[2] We assume $\max(S)$ to be defined in such a way that it selects either the largest integer value from S or 0 if S is the empty set.

i.e., that part of the store that can be accessed by expression e, and similarly $St_4 = St_2[\mathbf{reachOut}(St_2, v)]$*, i.e., the part of the store that can be accessed through document calls or the result sequence of e.*

- *The* largest input sequence length *is defined as* $d_I^s = \max(\{|s| \mid s \in \mathrm{rng}(\mathbf{v})\} \cup \{\Delta_{St_3}^{tree}\})$.
- *The* largest input integer value *is defined as* $d_I^v = \max(\mathbf{abs}(I^{St_3} \cup I^{En}))$.
- *The* largest output sequence length *is defined as* $d_O^s = \max(\{|v|, \Delta_{St_4}^{tree}\}) + |\mathbf{Set}(\mathbf{Sources}(l))|$.
- *The* largest output integer value *is defined as* $d_O^v = \max(\mathbf{abs}(I^{St_4} \cup I_v \cup \{i \mid \exists n, s : repVal(n, s) \in \mathbf{Set}(l) \wedge StringToInteger(s) = i\}))$.

We now illustrate the previous definition with an example. Consider the following program e:

```
for $x at $y in doc("doc.xml")//c return ($x, $y*100)
```

The evaluation of this expression does not change the store and when evaluated against the store in Figure 2.3 and the empty environment $En_0 = (\emptyset, \emptyset, \emptyset, \bot)$ we obtain the evaluation $(St, En_0) \vdash e \Rightarrow^{\mathtt{E}} (St, \langle n_3^e, 100, n_5^e, 200\rangle, \langle\rangle)$.

- The largest input sequence length is $d_I^s = 9$, which is the size of the largest (and only) tree in the input store.
- The largest input integer value is $d_I^v = 0$, since there are no integers in the input.
- The largest output sequence length is $d_O^s = 9$, which is the size of the largest (and only) tree in the output store.
- The largest output integer value is $d_O^v = 200$.

In the definition of the largest sequence lengths we include the size of the largest tree in the store, since one can generate such a sequence by using the descendant-or-self axis at the root of this tree. Moreover, the number of source nodes of the list of pending updates is added to the largest sequence, since the updates can add the source nodes to one tree.

The following inexpressibility results use the observation that, for the output, the maximum integer value and/or sequence length can be bounded by a certain class of functions in terms of the input. If such a function is a polynomial p that has $\mathbb{N}$ or $\mathbb{N}^2$ as its domain then there always exists an increasing polynomial p' such that p' is an upper bound for p. Therefore we assume that all such functions that are used as an upper bound in the following lemmas to be increasing functions.

Lemma 3.5. *For every* XQ^{upd} *expression e there are polynomials* p_1 *and* p_2 *such that for each evaluation* $(St, En) \vdash e \Rightarrow^{E} (St', v, l)$ *it holds that* $d_O^v \leq p(d_I^v)$ *and* $d_O^s \leq p(d_I^v, d_I^s)$.

Proof. We prove the lemma by induction on the size of the derivation tree of the expression e. Since we consider XQ^{upd} expressions, the nodes in this tree the nodes correspond to rules (S10-S19,S21-S43,S46-S53,S57).

First, consider the leafs of the derivation tree. Note that all off these have an empty list of pending updates in their result.

Variables (S10) The result only contains values from the input (store and environment) of the expression.

Literals (S11-S12) The result only contains constant values.

Steps (S23-S30) Similar to variables, i.e., items from the input store are returned.

Hence, the lemma holds for the previous expressions. All other expressions have subexpressions. We denote the largest input/output integer values of the k^{th} subexpression by $d^v_{I_k}$ and $d^v_{O_k}$. From the induction hypothesis it follows that for each subexpression it holds that $d^v_{O_k} \leq p_{k,1}(d^v_{I_k})$ and $d^s_{O_k} \leq p_{k,2}(d^v_{I_k}, d^s_{I_k})$ for polynomials $p_{k,1}$ and $p_{k,2}$. Note that many expressions (S13-S18,S22,S33-S43,S46-S53) do not alter the environment nor do they alter the reachable part of the store before passing them to their subexpressions, so $d^v_{I_k} = d^v_I$ for all their subexpressions, and hence $d^v_{O_k} \leq p_{k,1}(d^v_I)$ and $d^s_{O_k} \leq p_{k,2}(d^V_I, d^s_I)$.

Binary expressions (S13-S16,S22), `if` expressions (S17-S18), typeswitches (S33) and most basic built-in functions (S34-S43) All these expressions return result sequences which length is bound by the sum of lengths of the output of these subexpressions and hence $d^s_O \leq p_1(d^V_I, d^s_I)$ for some polynomial p_1. The integer values in the result are in the result of one of the subexpressions and hence $d^v_O \leq p_2(d^v_I)$ for some polynomial p_2.

Sequence Generation (S46) The largest integer value is the largest integer value of both subexpressions and the largest output sequence length is at most the maximum of the sum of the largest output sequence length of both subexpressions and twice the largest output integer value.

Constructors (S47-S50) These can worst-case for every item in the input sequence of the first subexpression the biggest tree in the input store, such that the output sequence length is still within the bounds that we have to show. Note that no additional pending updates are generated.

Update expressions (S51-S53) These expressions generate lists of pending updates, which are bound polynomially by the length of the output sequences of the subexpressions. All integer values in the result are also in the result of one of the subexpressions.

The expressions in XQ^{upd} that do change the environment are:

`for` expressions (S19) The environment is only changed for the second subexpression by changing the variable binding to items in the result of the first subexpression. Moreover, as shown in Property 3.1 the reachable part of the input store does not change for the evaluations of e_2. By induction we know, the largest integer value in the result of the first subexpression is $d^v_{O_1} \leq p_1(d^v_I)$ and the largest sequence length is $d^s_{O_1} \leq p_2(d^v_I, d^s_I)$, for some polynomials p_1 and p_2. From the induction hypothesis it follows that for each iteration of e_2 it holds that $d^v_{O_2} \leq p_3(d^v_{I_2})$ and $d^v_{O_2} \leq p_4(d^s_{I_2}, d^v_{I_2})$ for some polynomials p_3 and p_4, and hence $d^v_{O_2} \leq p_3(p_1(d^v_I))$ and $d^s_{O_2} \leq p_4(p_2(d^v_I, d^s_I), p_1(d^v_I))$. Since the result of a `for` expression contains only items that are in the result of an evaluation of e_2, we know that there exists a polynomial p such that $d^v_O \leq p(d^v_I)$ and moreover the largest ouput sequence length is at most $d^s_{O,1}.d^s_{O,2}$.

`let` expressions (S21) Similar to the `for` expression, only difference is that we now evaluate e_2 only once, but the entire result sequence of e_1 is now bound to a variable in the input of e_2. Hence the polynomial bounds also apply for these expressions.

Path expressions (S31-S32) These also obviously have output integer values and sequence lengths within these polynomial bounds, since they are in fact a special kind of `for` expressions with an extra selection at the end, i.e., a node test and removal of duplicate nodes.

`transform` expressions (S57) The expression e_1 in the `clause` has by induction largest sequence length $d^s_{O_1} \leq p_{1,1}(d^v_I, d^s_I)$ and largest integer value $d^v_{O_1} \leq p_{1,2}(d^s_I)$ for some polynomials $p_{1,1}$ and $p_{1,2}$. Deep copies of these result sequences are made before evaluating the expression in the `modify` clause. Hence the largest integer value in the input of e_2 is $d^v_{I_2} = d^v_{O_1}$ and the largest sequence length is $d^s_{I_2} \leq (d^s_{O_1})^2$ and similar input bounds hold for e_3. The expression e_3 is then evaluated and afterwards the list of pending updates generated in e_2 is applied and the result sequence of e_3 is returned. Hence it can be easily be verified that the polynomial bounds hold.

□

The previous lemma is combined with the following lemma to show that we cannot express `count` in fragments at the left-hand side of the *count*-border in Figure 3.1.

Lemma 3.6. *The fragment XQ_C does not have the property of Lemma 3.5.*

Proof. Consider the empty store St_0, the environment $En = (\emptyset, \emptyset, \{(\text{“}\$input\text{”}, \langle 1, \ldots, 1\rangle)\}, \bot)$, and the expression $e =$ `count($input)` where the length of the sequence bound to variable `$input` equals k, then the evaluation $(St_0, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, \langle\rangle)$ has largest input integer value $d^v_I = 1$ and output integer value $d^v_O = k$, which obviously also depends on d^s_I. □

The following lemma gives upperbounds for the largest output sequence lengths and integer values for evaluations in $XQ^{upd}_{at,C}$.

Lemma 3.7. *For every $XQ^{upd}_{at,C}$ expressions e there are polynomials p_1 and p_2 such that for each evaluation $St, En \vdash e \Rightarrow^{E} (St', v, l)$ it holds that $d^s_O \leq p_1(d^s_I, d^V_I)$ and $d^v_O \leq p_2(d^s_I, d^v_I)$.*

Proof. We prove the lemma by induction on the size of the derivation tree of the expression e. Since we consider $XQ^{upd}_{at,C}$ expressions, the nodes in this tree the nodes correspond to rules (S10-S44,S46-S53,S57). The proof is mostly similar to the proof of Lemma 3.5 and hence we omit most of the details. The only new expression is the `count` function (S44), for which it clearly holds that $d^v_O = \max(\{d^v_{O_1}, d^s_{O_1}\})$. From the induction hypothesis it follows that the output sequence lengths and integer values for the subexpression are bounded as follows: $d^s_{O_1} \leq p_1(d^V_I, d^s_I)$ and $d^v_{O_1} \leq p_2(d^v_I, d^s_I)$ for some increasing polynomials p_1 and p_2. Hence $d^v_O \leq p(p_2(d^v_I, d^s_I), p_1(d^V_I, d^s_I))$ for some polynomial p, which is clearly within the bounds we have to show. Moreover, the length of the output sequence is one, the length of the list of pending updates is the same as that of the subexpression and also the output store is the same as the output store of the subexpression.

Expressions in $XQ^{upd}_{at,C}$ that change the environment can contain a value that is obtained by using a count. We now illustrate how the induction works for these expressions by illustrating the `let` expressions. The reasoning for other expressions is similar.

`let` expression (S21) From the induction hypothesis it follows that the output sequence lengths and integer values for the first subexpression are bounded as follows: $d^s_{O_1} \leq p_1(d^s_I)$ and $d^v_{O_1} \leq p_2(\log(d^s_I), d^v_I)$ for some increasing polynomials p_1 and p_2. These upper bounds also apply to $d^s_{I_2}$ and $d^v_{I_2}$. From the induction hypothesis it follows that $d^s_{O_2} \leq p_3(d^s_{I_2})$ and $d^v_{O_2} \leq p_4(\log(d^s_{I_2}), d^v_{I_2})$ for some polynomials p_3 and p_4. Hence $d^s_O = d^s_{O_2} \leq p_3(p_1(d^s_I)) \leq p_5(d^s_I)$ and $d^v_O = d^v_{O_2} \leq p_4(p_1(\log(d^s_I)), p_2(\log(d^s_I), d^v_I)) \leq p_6(\log(d^s_I), d^v_I)$ for some increasing polynomials p_5 and p_6.

The previous results suffice to show that $d^s_O \leq p_1(d^v_I, d^s_I)$ and $d^v_O \leq p_2(d^v_I, d^s_I)$ where p_1 and p_2 are some polynomials that only depend on the expression itself and the functions in the environment and not on the values in the store or the environment. □

The previous lemma is combined with the following two lemmas for the separation of the *polysize*-border in Figure 3.1.

Lemma 3.8. *The fragment XQ_R does not have the property of Lemma 3.7.*

Proof. Clearly there are expressions in XQ_R that do not have this property. Indeed, if we consider the empty store St_0, the environment $En = (\emptyset, \emptyset, \{(\text{“\$}input\text{”}, k)\}, \bot)$, and the expression $e =$

```
declare function mpowern($m, $n) {
  if ($n = 1) then $m else ($m * mpowern($m, $n - 1))
};
mpowern($input, $input)
```

then the evaluation $(St_0, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, \langle\rangle)$ has largest input integer value $d^v_I = k$ and largest output integer value k^k, which is clearly not polynomial in terms of d^v_I and d^s_I. □

Lemma 3.9. *The fragment XQ^{snap} does not have the property of Lemma 3.7.*

Proof. Consider the empty store St_0, the environment $En = (\emptyset, \emptyset, \{(\text{“\$}input\text{”}, k)\}, \perp)$, and the expression $e =$

```
let $m := element {"result"} { 1 } return (
  (for $n in (1 to $input) return
    snap ordered {
      replace value of $m/text() with (xs:integer($m/text()) * $input)
    }
  ),
  xs:integer($m/text())
)
```

then the evaluation $(St_0, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, \langle\rangle)$ has largest input integer value $d^v_I = k$ and largest output integer value k^k, which is clearly not polynomial in terms of d^v_I and d^s_I. □

3.3.4 Depth of Transformations

The ability to change nodes at an arbitrary depth is something that is not possible in some LiXQuery fragments. The following lemma states that in XQ^{constr}_{at} all nodes in the result that are nested at least d levels, have to be deep equal to nodes in the input store, where this d is linearly bounded by the size of the program.

Lemma 3.10. *For all XQ^{constr}_{at} programs there is a depth d such that all nodes that are in the result store, but not in the input store and that have at least d ancestors are deep-equal to nodes in the input store.*

Proof. This property is shown by induction on the structure of the program. Only node construction can create new nodes and the result is a new tree in the store, where all nodes except for the root are deep-equal to nodes that already existed, i.e., that are in the result store of the subexpression. Hence, it can be easily seen that every expression e can only add one more level to the newly created nodes, so there is a number d, linearly bounded by the size of the expression, such that all nodes deeper than d are deep-equal to nodes in the input store. □

Lemma 3.11. *The property of Lemma 3.10 does not hold for XQ^{upd} programs.*

Proof. Consider the following XQ^{upd} program:

```
transform copy $x := doc("a.xml")
modify (for $y in $x//a return rename $y as "b")
return $x
```

This program does not satisfy the property of Lemma 3.10, because it makes a copy of the document tree of `a.xml` and relabels all a nodes at *any level* in this tree. □

3.3.5 Termination of Programs

The last separation result is based on the observation that for some fragments all programs can be simulated by a Turing machine that always halts, while for other fragments this does not hold.

Lemma 3.12. *There are programs in XQ_R that cannot be simulated by a $XQ^{snap}_{at,C}$ program.*

Proof. As shown in Section 2.2, XQ_R is Turing-complete. However, it can easily be seen that $XQ^{snap}_{at,C}$ programs can be simulated by a Turing machine that always halt, since we have no recursion and only iteration occurs through `for` expressions and path expressions which first compute a *finite* input, before performing the iteration over the items in this sequence. □

We conjecture that the expressive power of $XQ^{snap}_{at,C}$ is more limited than recursive languages and that the bounds of expressive power of this LiXQuery fragment corresponds to primitive recursive functions. We now show how we can simulate all primitive recursive functions in $XQ^{snap}_{at,C}$. We give a translation τ which maps primitive recursive functions to $XQ^{snap}_{at,C}$ expressions with one free variable. This free variable, `$x` models the arguments of primitive recursive functions, which are tuples of natural numbers, by a sequence of integers. First, we translate the basic primitive recursive functions to $XQ^{snap}_{at,C}$ in a straightforward manner, i.e., the zero function is translated to `0`, the successor function `$x + 1`), and the projection of the i^{th} item in a tuple to `(for $y at $z in $x return (if ($z =` i` then ($y) else ()))`. More complex primitive recursive functions can be translated by first translating the functions that occur as an argument and then combining these expressions to a new expression as follows. The composition of a k-ary primitive recursive function f and k l-ary primitive recursive functions $g_0, \ldots, g_k$ is translated to `let $x :=` $(\tau(g_0)$`, ...,` $\tau(g_{k-1}))$ `return` $(\tau(f))$. Finally, we show the translation of primitive recursion. Let f be a k-ary primitive recursive function, and g, a $(k+2)$-ary primitive recursive function. If h is a $(k+1)$-ary function defined as the primitive recursion of f and g, then $\tau(h)$ is defined as follows:

```
let $n := (for $y at $z in $x return (if ($z = 1) then ($y) else ()))
let $x := (for $y at $z in $x return (if ($z = 1) then () else ($y)))
let $comp := element {"comp"} {for $xi in (τ(f)) return element {"x"} {$xi}}
for $m in (1 to $n) return
  snap ordered {
    let $newh := (for $xi in $comp/comp/x return xs:integer($xi/text()))
    let $x := ($newh, $m - 1, $x)
    let $newcomp := (for $xi in τ(g) return element {"x"} {$xi})
    return (delete $comp//x, insert $newcomp into $comp)
  }
```

An example of a computable function that we conjecture to be not expressible in $XQ^{snap}_{at,C}$ is the Ackermann-function.

3.4 Expressibility Results

Adding extra features to LiXQuery fragments does not always extend the set of transformations expressible in the fragment. In this section we show how we can simulate expressions and programs in LiXQuery fragments.

3.4.1 Expression Simulations

The first expressibility results are obtained by simulating expressions, i.e., operations in fragments that syntactically do not include the feature that we are simulating. In order to do this, we have to show that both the result sequence and the list of pending updates are the same (up to node identity of the newly created nodes) in the simulated expression and the simulating expression. For the list of pending updates, this will be easy to see, since for all simulations that we introduce here, it can be verified that the subexpressions which may generate pending updates are still only executed once and in the right order.

First we show that we can count the number of items in a sequence in all fragments that include an `at` in `for` expressions, which is needed to show we can count in fragments at the right-hand side of the *count*-line in Figure 3.1.

Lemma 3.13. *The* `count` *operator can be expressed in* XQ_{at}, XQ_{at}^{constr}, XQ_{at}^{upd}, XQ_{at}^{snap} *and* $XQ_{at,R}$

Proof. From Section 2.2 we know that `empty(`e_1`)` can be expressed in XQ. Counting the items of a sequence corresponds to finding the maximal position of an item in a sequence. Hence `count(`e_1`)` is equivalent to :

```
let $positions := (for $i at $pos in (e1) return $pos) return
for $a in (0, $positions) return
  if (empty(
    for $b in $positions return
      if ($b > $a) then 1 else ()
  )) then $a else ()
```

This expression always returns exactly one item, since `(0, $positions)` does not contain duplicate values, and hence there is exactly one item which is the largest. □

The following Lemma gives another way to simulate `count`. In this simulation we use node construction and recursive function definitions.

Lemma 3.14. *The* `count` *operator can be expressed in* XQ_R^{constr}.

Proof. We will show how to define a recursive function `count-distinct-nodes` such that `count(`e_1`)` is equivalent to following XQ_R^{constr} expression:

```
count-distinct-nodes(
  for $e in e1 return element {"e"} {()}
)
```

This expression generates as many new nodes as there are items in the input e_1 and then applies a newly defined function `count-distinct-nodes` to this sequence, which counts the number of distinct nodes in a sequence. This can be done by decreasing the input sequence of the function call to `count-distinct-nodes` by exactly one node in each recursion step, which is possible since all items in the input sequence of `count-distinct-nodes` have a different node identity and hence we can remove each step the first node (in document order) of the newly created nodes. More precisely, the function `count-distinct-nodes` can be defined as follows:

```
declare function count-distinct-nodes($seq) {
  if (empty($seq)) then 0
  else (
    let $newseq := (
      for $e1 in $seq return
        if (empty(
          for $e2 in $seq return
            if ($e2 << $e1) then 1 else ()
        )) then () else $e1
    )
    return (1 + count-distinct-nodes($newseq))
  )
}
```

□

We now show how to simulate the `at` clause of a `for` expression. First we simulate the position information using the `snap` construct and node construction. Together with Lemma 3.13 and Lemma 3.14 this shows that we can simulate `count` in all fragments at the right-hand side of the *count*-border in Figure 3.1.

Lemma 3.15. *The* `at` *clause in a* `for` *expression can be expressed in* XQ^{snap}.

Proof. In order to simulate the position information, we make a new node that will be used as a counter and modify its value every iteration. We assume, w.l.o.g., that `$position` is not a free variable in e_1. The simulation of `for` $\$x$ `at` $\$y$ `in` e_1 `return` e_2 can be done as follows:

```
let $position := (element {"position"}{0})/text() return
for $x in e1 return
  let $y := xs:integer($position) return (
    snap ordered {
      replace value of $position with ($y + 1);
    }, e2
  )
```

Note that the expression e_2 is not within the scope of the snap operation and hence the generated list of pending updates in the simulated expression is also the resulting list of pending updates for the entire simulating expression. □

The last expressibility result simulates the `at` clause by using node construction and `count`. Together with the previous lemma, we use this result to show that we can simulate `at` in all fragments at the right-hand side of the *at*-border in Figure 3.1.

Lemma 3.16. *The* `at` *clause in a* `for` *expression can be expressed in* XQ_C^{constr} *and* XQ_C^{upd}.

Proof. First we proof this lemma for XQ_C^{upd}. We transform sequence order into document order by creating new nodes as children of a common parent such that the new nodes contain all information of each item in the sequence and they are in the same order as the items in the original sequence. First, in XQ_C^{count} we can express the (non-recursive) functions `pos` and `atpos`, which respectively give the position of a node in a document-ordered duplicate-free sequence and return a node at a certain position in such sequence. This can be done as follows:

```
declare function pos($node, $seq) {
  count(for $e in $seq return
    if ($e << $node) then 1 else ()
  ) + 1
};

declare function atpos($seq, $pos) {
  for $node in $seq return
    if (pos($node, $seq) = $pos) then $node else ()
};
```

Let us assume that we can define XQ_C^{constr} functions `encode` and `decode` such that `encode` translates an arbitrary sequence to an ordered and duplicate-free sequence of nodes while encoding each item in the original sequence into one node at the same position and the function `decode` can retrieve the original item given this node and the original sequence. Then the following XQ_C^{upd} expression is equivalent to the $XQ_{at,C}^{upd}$ expression `for` $\$x$ `at` $\$pos$ `in` e_1 `return` e_2 (where e_1 and e_2 are XQ_C^{upd} expressions):

```
let $seq := (e1) return
let $newseq := encode($seq) return
for $x in $newseq
return (
  let $pos := pos($x, $newseq) return
  let $x := decode($x, $seq)
  return (e2))
```

Because the result sequence of e_1, `$seq`, is used both in the `in` clause of the `for` expression and as actual parameter for the `decode` function, we have to assign this result to a new variable, since by simple substitution a node construction that is done in e_1 would be evaluated more than once. Furthermore the expression e_2 is guaranteed to have the right values for the variables $\$x$ and $\$pos$ iff the function `decode` behaves as desired. We can assume, w.l.o.g., that e_2 does not use variables `$seq` and `$newseq`, since they are used in the simulation.

We now take a closer look at how to define the functions `decode` and `encode`. The function `encode` needs to create a new sequence in which we simulate all items by creating a new node for each item. By adding these nodes as children of a newly constructed element (named `newseq`) we ensure that the original sequence order is reflected in the document order for the newly constructed sequence. Atomic values are simulated by putting their value as text node in an element which denotes the type of atomic value. Encoding nodes cannot be done by making a copy of them, since this would discard all information we have about the node identity. Therefore we store for a node all information we need to retrieve the node later using the function `decode`. We do this by storing the root of the node and the position where the node is located in the descendant-or-self list of its root node.

For example, consider the Store St of Figure 2.3, which only has one tree. Encoding the sequence $\langle 1, n_2^e, \text{“}c\text{”} \rangle$ over this store results into the creation of the following new element:

```
<newseq>
  <int>1</int>
  <node root="1" descpos="3"/>
  <str>c</str>
</newseq>
```

Note that this coding is possible because of Property 3.1. The encoding and decoding is performed by the following two functions:

```
declare function encode($seq) {
  let $rootseq := (
    for $e in $seq return
      typeswitch($e)
      case element() return root($e)
      case attribute() return root($e)
      case document-node() return root($e)
      default return ()
  )/. return
  let $newseq := element {"newseq"} {
    for $e in $seq
    return
      typeswitch($e)
      case xs:integer return element {"int"} {$e}
      case xs:string return element {"str"} {$e}
      case xs:boolean return element {"bool"} {if ($e) then 1 else 0}
      default return element {"node"} {
        attribute {"root"} {pos(root($e), $rootseq)},
        attribute {"descpos"} {pos($e, root($e)//.)}
      }
  }
  return $newseq/*
};

declare function decode($node, $seq) {
```

```
let $rootseq := (
    for $e in $seq return
      typeswitch($e)
      case element() return root($e)
      case attribute() return root($e)
      case document-node() return root($e)
      default return ()
  )/. return
  if (name($node) = "int") then xs:integer($node/text())
  else if (name($node) = "str") then string($node/text())
  else if (name($node) = "bool") then
    if (xs:integer($node/text()) = 1) then true() else false()
  else if (name($node) = "node") then (
    let $root := atpos(rootseq($seq), xs:integer($node/@root))
    return atpos($root//., xs:integer($node/@descpos))
  )
  else ()
};
```

Note that none of the previous functions used recursion, so we do not actually need functions since we could inline the function definitions in the expressions. Hence `at` can be expressed in XQ_C^{upd}. Finally, it can be easily verified that we never introduce new updating expressions and hence when the simulated expression is an $XQ_{C,at}^{constr}$ expression then the simulated expression is a XQ_C^{constr} expression. □

3.4.2 Program Simulations

The following expressibility result introduces a simulation for programs only in the sense that they will ensure that the correct serialized result will be returned, but no guarantees are given for node identity. We will show the following expressibility result for $XQ_{at,C,R}^{constr}$.

Lemma 3.17. *For all XQ_R^{snap} programs e it holds that there is a $XQ_{at,C,R}^{constr}$ program e' that $e \sim e'$.*

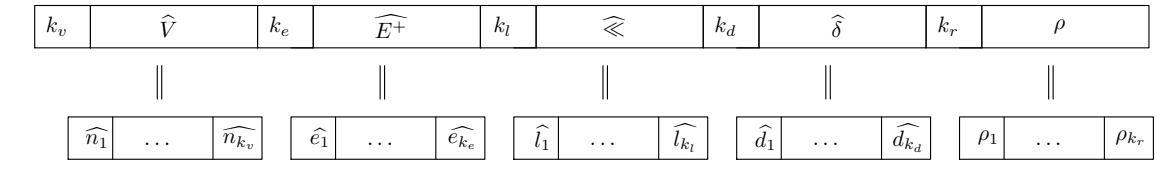
Proof. This simulation is done by simulating the computational context, i.e., the store, the evaluation environment and the list of pending updates will be encoded into one sequence. This simulation technique is similar to the one we used in [Page et al., 2005] to show that adding node construction to XQ_R does not add expressive power for "node-conservative deterministic queries", however we now include some node creation and nodes in the encoded store which are needed to simulate the non-deterministic behavior. Note that the simulation that we give in this section is a little bit too complicated and powerful for the lemma we need in the proof of Theorem 3.1, but in the discussion in Chapter 5 we will show how to extend this simulation to show that we can separate queries and updates in $XQ_{at,C,R}^{upd}$ expressions.

Simulating the Computation Context

First, we show how the structure of the simulated computational context. Expressions will return a simulating sequence, which we create by combining sequences that respectively encode the store, the list of pending updates and the result sequence. The result of each simulating expression is the concatenation of these three sequences, each one prepended by one atomic value indicating its length. It can easily be seen that in this way we preserve all information that we need to perform computations. Moreover we specify the simulated computation context in such a way that queries or expressions that we do not need recursive functions in the simulation of expressions, except for when we have nested `snap` expressions or recursive function calls. Finally, note that enough information is preserved in order to be able to generate a *deep-equal result sequence*.

- **Simulated Store**: To encode a store $St = (V, E, \ll, \nu, \sigma, \delta)$, we create a sequence which consists of 5 components which are needed to maintain the necessary information:
  - Encoded set of nodes: $\widehat{V}$ contains node identifiers, along with an indication of the type of node (i.e., element, text, attribute or document node), the name and the text value of the node.
  - Encoded transitive closure of the edge relation: $\widehat{E^+}$ contains for all couples of nodes $(n_1, n_2) \in E^+$ the identifiers, as they are used in $\widehat{V}$.
  - Encoded document order: $\widehat{\ll}$ contains similarly for all couples of nodes (n_1, n_2) where $n_1 \ll n_2$, the identifiers, as they are used in $\widehat{V}$.
  - Encoded URI-document mapping: $\widehat{\delta}$: contains for every $(s, n) \in \delta$, the couple (s, i) where s is a string (URI) and i is the node identifier of $n = \delta(s)$, as used in $\widehat{V}$.
  - Encoded root order: ρ contains for all root nodes n in the store St the couple (i, n) where i is the identifier of n, as it is used in $\widehat{V}$. We need this part to preserve the full document order information during the simulation.

  The following figure illustrates the encoded store:



with $\widehat{n_i} = \langle id(n_i), type(n_i), \nu(n_i), \sigma(n_i)\rangle$ and $id(n_i) = i$
$\widehat{e_i} = \langle id(n_{1,i}), id(n_{2,i})\rangle$ $type(n_i) =$ "element" $(n_i \in \mathcal{V}^e)$
$\widehat{l_i} = \langle id(n'_{1,i}), id(n'_{2,i})\rangle$ "attribute" $(n_i \in \mathcal{V}^a)$
$\widehat{d_i} = \langle s_i, id(\delta(s_i))\rangle$ "document" $(n_i \in \mathcal{V}^d)$
$\rho_i = \langle id(n''_i), n''_i\rangle$ "text" $(n_i \in \mathcal{V}^t)$

- **Simulated List of Pending Updates** The encoding $\widehat{l}$ of a list of pending updates l is defined as follows: $l = \langle o_1, \ldots, o_k \rangle \Leftrightarrow \widehat{l} = \langle b_1, \widehat{o_1}, \ldots, b_k, \widehat{o_k} \rangle$, with with $\widehat{o}$ defined as:

o	$\widehat{o}$
$a(n, \langle n_1, \ldots n_m \rangle)$	$\langle$"a", $id(n), id(n_1), \ldots, id(n_m)\rangle$
$a(n, s)$	$\langle$"a", $id(n), value(s)\rangle$
$a(n)$	$\langle$"a", $id(n)\rangle$

  where a is the name of a primitive update operation. We use nodes b_i before each $\widehat{o_i}$ in $\widehat{l}$ only for determining the beginning and the end of an encoded primitive update, since we want to iterate over the encoded primitive update operations by using a limited number of `for` expressions (without side-effects) and the encoded primitive update operations can contain any atomic value.

- **Simulated Sequence over a Store** The simulation of a sequence $\langle x_1, \ldots, x_m \rangle$ is a sequence of atomic values $\langle kind(x_1), value(x_1), \ldots, kind(x_m), value(x_m)\rangle$, where $kind(x_i)$ is 0 if x_i is an atomic value and 1 if x_i is a node. Moreover, $value(x_i)$ is x_i if x_i is an atomic value or $id(x_i)$ if x_i is a node.

Translating Expressions

We now illustrate how we can simulate expressions. Unfortunately, we cannot assume that the entire store is encoded in advance (it is possibly infinitely big), so we need to load the relevant parts of the store when needed. This complicates the simulation, since only at run time we can see which parts of the store are actually used. We show what the preconditions, postconditions and invariants are for the simulating expression in order to compute the correct (encoded) result. We have to show that the simulation ϵ is defined such that for every expression e, stores St_1, St_2, St_3, St_4, and environments $En_1 = (\mathbf{a}_1, \mathbf{b}_1, \mathbf{v}_1, \mathbf{x})$) over St_1, $En_2 = (\mathbf{a}_2, \mathbf{b}_2, \mathbf{v}_2, \mathbf{x})$ over St_3 it holds that $(St_1, (\mathbf{a}_1, \mathbf{b}_1, \mathbf{v}_1, \mathbf{x})) \vdash e \Rightarrow^{\mathtt{E}} (St_2, v_1, l)$ iff $(St_3, En_2) \vdash \epsilon(e) \Rightarrow^{\mathtt{E}} (St_4, v_2, \langle\rangle)$ if following conditions hold:

- $\mathbf{a}_1(s) = \langle s_1, \ldots, s_m \rangle \Leftrightarrow \mathbf{a}_2(s) = \langle s_1, \ldots, s_m, \text{“}store\text{”}\rangle$, i.e., the encoded store is added as a parameter to the function signatures.

- $\mathbf{b}_1(s) = e' \Leftrightarrow \mathbf{b}_2(s) = \epsilon(e')$, i.e., the function bodies are replaced by their simulating function bodies.

- $\mathbf{v}_1(s) = \langle x_1, \ldots, x_m \rangle \Rightarrow \mathbf{v}_2(s) = \langle kind(x_1), value(x_1), \ldots, kind(x_m), value(x_m)\rangle$, i.e., variables are stored encoded.

- $\mathbf{v}_2(\text{“}dot\text{”}) = 0 \Leftrightarrow \mathbf{x} = \perp$, i.e., `$dot` is 0 iff there is no context node.

- $\mathbf{v}_2(\text{“}dot\text{”}) = id(n) \Leftrightarrow \mathbf{x} = n$, i.e., `$dot` contains the id of the context node.

- $\mathbf{v}_2(\text{“}store\text{”})$ is $\widehat{St_5}$ for some St_5 such that $\delta_{St_5} \subseteq \delta_{St_1}$, $St_5[\mathbf{reachIn}(St_1, En_1)] = St_1[\mathbf{reachIn}(St_1, En_1) - (\text{rng}(\delta_{St_1}) - \text{rng}(\delta_{St_5}))]$, and $St_3[\text{rng}(\delta_{St_1}) - \text{rng}(\delta_{St_5})] = St_1[\text{rng}(\delta_{St_1}) - \text{rng}(\delta_{St_5})]$, i.e., `$store` contains the encoded store for St_5 which contains some documents which were at some point already accessed through `doc()` calls, encoded trees for the nodes accessible by the simulated expression, and possibly some “garbage”. Informally, St_5 is the substore of St_1 that is already loaded by document calls or that is reachable by variables in the environment. The documents accessible through URIs that are not yet loaded in St_5 are the same in St_1 and St_3.
- $\text{dom}(\mathbf{v}_2) - \{\text{“}store\text{”}, \text{“}dot\text{”}\} = \text{dom}(\mathbf{v}_1)$, i.e., the variables *store* and *dot* do not occur in the variable binding $\mathbf{v}_1$ and no other new variables are needed in the simulation $\epsilon(e)$[3].

Since the simulation will be defined inductively, we assume $\epsilon(e)$ denotes the simulation of the expression e. We now show how to simulate sequence concatenation, document loading, a for-loop, an edge deletion and a snap operation. The simulation of the other expressions is obtained using techniques similar to those introduced in the proofs for the five previously mentioned expressions, and are briefly discussed at the end.

Sequence Concatenation The simulation of the sequence concatenation e_1 , e_2 is straightforward, but we give it here to serve as an example of how the general simulation is done.

```
let $res := ε(e1) return
let $store := store($res) return
let $val1 := val($res) return
let $upd1 := upd($res) return
  let $res := ε(e2) return
  let $store := store($res) return
  let $val2 := val($res) return
  let $upd2 := upd($res) return
  return resultEnc($store, ($upd1, $upd2), ($res1, $res2))
```

Note that we can assume, w.l.o.g., that `$res, $val1, $upd1` do not occur as free variables in e_2. In the rest of this proof we assume that all newly introduced variables except `$store` and `$dot` are no free variables in the translated subexpressions. The previous translation used four non-built-in functions. The functions `store($res)`, `val($res)` and `upd($res)` extract respectively the encoded values for St, v, and l from the encoded result sequence `$res`. The function `resultEnc` is used to combine an encoded store, encoded list of pending updates and an encoded value to one result sequence.

Document loading The translation of a document call checks whether the document that would be loaded by the URI that is the result of the subexpression is already

[3]Note that in the actual simulation we sometimes define new variables, but they never occur as a free variable in a simulating (sub)expression.

in $rho and if this is not the case, the entire document is loaded. The expression doc(e) is simulated as follows:

```
let $res := ϵ(e) return
let $store := store($res) return
let $val := val($res) return
let $upd := upd($res)
let $docURI := $val[2] return
let $docNode := doc($docURI) return
let $docId := (
  let $rho := rho($store) return
  for $x at $p in $rho
  return
    if (isOdd($p) and ($rho[p+1] is $docNode))
    then ($x)
    else ()
)
return
  if (not(empty($docId))) then (
    let $delta := delta($store) return
    let $deltaLookup := (
      for $x at $p in $delta
      return
        if (isOdd($p) and ($x = $docURI)) then ($docURI, $delta[$[+1])
    )
    return
      if (empty($deltaLookup))
      then (
         let $newDelta := ($delta, ($docURI, $docId))
         let $newStore := storeEnc(V($store),E($store),docord($store),
                                   $newDelta,rho($store))
         return resultEnc($newStore, $upd, (1, $docId))
      )
      else resultEnc($store, $upd, (1, $docId))
  )
  else (
    let $nodes := $docnode//. return
    let $maxId := maxId($store) return
    let $V := (V(store($res)),
    for $n at $pos in $nodes return
    typeswitch ($n)
    case element() return ($maxId+$pos,"element",name($n),"")
    case attribute() return ($maxId+$pos,"attribute",name($n),string($n))
    case text() return ($maxId+$pos,"text","",string($n))
    case document-node() return ($maxId+$pos,"document","","")
    default return ()
```

```
)
let $E := (E(store($res)),
   for $n1 at $pos1 in $nodes return
     for $n2 at $pos2 in $nodes return
       if ($n1//. = $n2) then ($maxId+$pos1,$maxId+$pos2) else ()
 )
 let $rho := rho(store($res), ($maxId+1, $docnode)) return
 let $docord := (
   let $incomplDocord := (docord(store($res)),
     for $n1 at $pos1 in $nodes return
       for $n2 at $pos2 in $nodes return
         if ($n1 << $n2) then ($maxId+$pos1,$maxId+$pos2) else ()
   ) return complDocord($incompl, $rho)
 ) return
 let $delta := (delta(store($res)), ($val[2]), $maxId+1)
 return resultEnc(storeEnc($V, $E, $docord, $delta, $rho),
                  $upd, (1, $maxId+1))
)
```

Note that we used positional predicates in the assignment of $\$s'$, which is a shorthand. Some new functions appear in this simulation. The functions `V($store)`, `E($store)`, `docord($store)`, `delta($store)`, and `rho($store)` retrieve components of the encoded store. The function `maxId($store)` returns the highest number that is used in the simulated store St as a node identifier and the function `isOdd($number)` checks whether the integer value in `$number` is an odd number. Finally, The function `complDocord($incomplDocord)` completes the encoded document order using the document order of the root nodes referenced in ρ assuming the document order is already complete for all trees of the forest that is represented in the encoded store. This function can clearly be expressed in XQ_R^{constr}.

`for` expression The `for` expression is one of the most important constructs in XQuery. For each `for`-loop we show how to generate a function that exactly computes the result of this for-loop. Assume that for each for-loop in the orginal expression we have associated a unique number x, used to define for every for-expression a unique function `for-`x`()`. The parameter $vars_x$ of this function represent all free variables in e'. Recursion is used here to simulate the iteration over a sequence where the resulting store of the previous step is passed on to the following step. The expression `for` $\$s$ `at` $\$s'$ `in` e `return` e' can then be simulated as follows:

```
let $res := ϵ(e) return
let $store := store($res) return
let $upd := upd($res) return
let $val := val($res) return
let $tempRes := for-x(1, $val, $store, vars_x)
return prependUpdList($upd, $tempRes)
```

Note that the result is stored in `$tempRes` and the list of pending updates generated during the evaluation is prepended to the list of pending updates previously generated during the evaluation of this expression. This is done by the function `prependUpdList($upd,$tempRes)`. Finally, the function `for-`x`()` is defined as follows:

```
declare function for-x($pos, $seq, $store, vars_x) {
  if ($pos <= (count($seq) idiv 2)) then
    let $s := ($seq[$pos*2-1], $seq[$pos*2]) return
    let $s' := (0, $pos) return
    let $res1 := ε(e') return
    let $store1 := store($res1) return
    let $upd1 := upd($res1) return
    let $val1 := val($res1) return
    let $res2 := for-x($pos + 1, $seq, $store1, vars_x)
    let $store2 := store($res2) return
    let $upd2 := upd($res2) return
    let $val2 := val($res2)
    return resultEnc($store2, ($upd1, $upd2), ($val1, $val2))
  else resultEnc($store, (), ())
}
```

Note that we have to use recursive functions to simulate the behavior of for-loops, since the encoded result store of one iteration has to be the input encoded store of the next iteration. Also note that we used positional predicates in the assignment of $\$s'$, which is a shorthand.

Edge deletion The simulation of the edge deletion `delete` e, simply adds for each result node n of the expression e the encoded primitive update operation $\langle$"*del*", $id(n)\rangle$ to the encoded list of pending updates as follows:

```
let $res := ε(e) return
let $val := val($res) return
let $newUpd := ((count($val) idiv 2),
  for $node at $pos in $val return
    if (not(isOdd($pos)))
      then ("del", $node)
      else ()
) return appendUpdList($newUpd,$res)
```

Note that the function `appendUpdList` is the counterpart of the `prependUpdList` function. We assume that e only returns nodes. We can easily modify the previous simulation to return an error (have an undefined result) iff e yields an item that is not a node, but we have opted not to do this here for the sake of clarity.

`snap` operation To ensure a correct computation, we have to apply updates on the encoded store as soon as they are applied in the XQ_R^{snap} expression. We illustrate the

simulation of the snap-operation by simulating the ordered mode. The unordered deterministic mode can be simulated by adding the conflict checks and after this simulation we show how to obtain the non-determinism needed for the unordered non-deterministic mode.

```
let $res := ϵ(e) return
let $store := store($res) return
let $val := val($res) return
let $upd := upd($res) return
return resultEnc(applyUpd($store, $upd), (), $val)
```

We assume the function `applyUpd`, which can be defined as follows:

```
declare function applyUpd($store, $upd) {
  let $firstUpd := (
    let $firstBorder := min(
      for $x at $p in $upd
      return if ($x/.) then $p else ()
    )
    return
      for $x at $p in $upd
      return if ($p < $firstBorder) then $x else ()
  )
  if (empty($firstUpd)) then (
    $store
  ) else if ($firstUpd[1] = "del") then (
    del($store, $firstUpd[2])
  ) else if ($firstUpd[1] = "ren") then (
    rename($store, $firstUpd[2], $firstUpd[3])
  ) else if ($firstUpd[1] = "repVal") then (
    repVal($store, $firstUpd[2], $firstUpd[3])
  ) else if ($firstUpd[1] = "insInto") then (
    insInto($store, $firstUpd[2], $firstUpd[position() >= 3])
  ) else if ($firstUpd[1] = "insAft") then (
    insAft($store, $firstUpd[2], $firstUpd[position() >= 3])
  ) else if ($firstUpd[1] = "insBef") then (
    insBef($store, $firstUpd[2], $firstUpd[position() >= 3])
  )
};
```

Note that the `$x/.` checks whether the sequence bounds to `$x` contains a node and since `$x` is a singleton sequence, it checks whether the item in it is a node. The function `min()` is assumed to return the minimal value in a list of integers. For each primitive update operation, we define one function that maps the parameters of this operation and the store to its result store. We illustrate how to write such a simulation by giving the simulation of the `del` primitive update operation:

```
declare function del($store, $node) {
  let $V := V($store) return
  let $E := (
    let $oldE := E($store) return
    for $x at $p in $oldE
    return if (isOdd($p)) then (
      if ($x = $node) then ()
      else if ($oldE[$p+1] = $node) then ()
      else ($x, $oldE[$p+1])
    ) else ()
  ) return
  let $rho := (
    let $oldRho := rho($store) return
    let $oldFilteredRho := (
      for $x at $p in $oldRho
      return if (isOdd($p)) then (
        if ($x = $node) then ()
        else ($x, $oldFilteredRho[$p+1])
      )
    ) return ($oldFilteredRho, ($node, element {"node"} {()} ))
  )
  let $docord :=
    let $oldComplDocord := docord($store) return
    let $newIncomplDocord := (
      for $x at $p in $oldComplDocOrd
      return if (isOdd($p)) then (
        if ($x = $node) then ()
        else if ($oldComplDocord[$p+1] = $node) then ()
        else ($x, $oldComplDocord[$p+1])
     ) else ()
    ) return complDocord($newIncomplDocord, $rho)
  ) return
  let $delta := (
    let $oldDelta := delta($store)
    for $x at $p in $oldE
    return if (isOdd($p)) then (
      if ($oldDelta[$p+1] = $node) then ()
      else ($x, $oldDelta[$p+1])
    ) else ()
  )
  return storeEnc($V, $E, $docord, $delta, $rho)
};
```

Note that we use node creation to obtain a new position for the deleted node in the store. The simulation of the other primitive update operations is straightforward

and also consists of first updating all four components (`$V, $E, $docord, $delta`) separately and then encoding the newly obtained store.

Other Expressions We finish this proof by indicating how the simulation of other expressions can be obtained. The translation for function and variable declarations are straightforward and so are literal values, variable references and the empty sequence. The root function looks for a node that is ancestor of the input node and that has no parents. This information is read from the encoded E^+ and if no node is found, the input node is returned because it is already a root node. The simulation of the other built-in functions is trivial. The condition check, let binding, concatenation, and all boolean operators are straightforward. Also typeswitches, function calls, sequence generation are not hard to simulate.

Path expressions are simulated by using a variable `$dot` to store the context node and the translation of the axes and node tests is also not difficult to express on the simulated store. For example, the child axis is simulated by looking for the nodes `$n1` in E^+ that are a descendant of `$dot`, but for which there are no nodes `$n2` such that `$n2` is a descendant from `$dot` and an ancestor from `$n1`. Sorting by document order and the removal of duplicates is done in three steps. First, the duplicates can be removed by filtering out all nodes that appear also at a lower position in the node sequence. After this, we count for each node n_i how many nodes of the sequence $\langle n_1, \ldots, n_k \rangle$ are before the current node in document order. The result of this step is the sequence $\langle c(n_1), \ldots, c(n_k) \rangle$ and is a permutation of $\langle 0, \ldots, k-1 \rangle$. Finally, we iterate over all numbers from 0 to $k-1$ and return the node n_i for which it holds that $c(k_i)$ is the current value of the iteration variable.

The only construct left to explain is node construction. We briefly sketch how the simulation of element construction can be done, the other node construction expressions are similar. First, we get a new identifier i, which can be done by using `maxId($store)+1`. Then we make deep-copies of all the nodes that are selected by the second subexpression during which i is added to the value of their node identifiers. This deep-copy adds not only encoded nodes to $\widehat{N}$, but also couples to both the $\widehat{E^+}$ and $\widehat{\ll}$. Finally, the document order is completed by performing `addInDocord(`i`, $incomplDocord, $E)`.

Generating a Deep-Equal Result Sequence

Given a simulated sequence over an encoded store, we can create a new sequence which is deep-equal to the sequence that is being simulated. We now show briefly how to create an item that is deep-equal to a simulated item $\langle kind, value \rangle$

- If the item is an atomic value ($kind = 0$) then return the $value$.
- Else the item is a node and $value = id(n_i)$.
  - For all its children (check $\widehat{E^+}$) generate deep-equal items.

- Check the type of the node n_i, the value and the label by looking for the encoded node $\langle id(n_i), type(n_i), \nu(n_i), \sigma(n_i)\rangle$ in $\widehat{V}$
- Now create a new node of the correct type and give it the correct label and value. If the node has descendants then add them to the body of the node constructor.
- Return the newly created node.

It is easy to see that this can be done in XQ_R^{constr}. □

3.5 Proving the Relationships between the Fragments

In this Section we show the correctness of Theorem 3.1 by combining the results of the two previous sections.

First, we prove that the dotted borders in Figure 3.1 are correct by showing that something can be expressed in each of the least expressive fragments of the right-hand side that cannot be expressed in any of the most expressive fragments of the left-hand side.

***set*-border** The most expressive fragment on the left-hande side is XQ_R. The least expressive fragment on the right-hand side is XQ_C. From Lemma 3.1 and Lemma 3.2 it follows that we cannot write a XQ_R program to decide whether two input sequences, having the same set representation, are the same sequence, while in XQ_C this can be done.

***bag*-border** The most expressive fragment on the left-hande side is $XQ_{C,R}$. The least expressive fragment on the right-hand side is XQ_{at}. From Lemma 3.3 and Lemma 3.4 it follows that we cannot write a XQ_R program to decide whether two input sequences, having the same bag representation, are the same sequence, while in XQ_{at} this can be done.

***constr*-border** The most expressive fragment on the left-hande side is $XQ_{at,C,R}$. The least expressive fragment on the right-hand side is XQ^{constr}. We can return new nodes in XQ^{constr} which are not deep-equal to nodes in the input sequence, for example the program `element "a"()` cannot be expressed in $XQ_{at,C,R}$, since we would need an `a` node with no children in the input store, but a simulation has to hold for every store and hence also for the empty input store.

***depth*-border** The most expressive fragments on the left-hande side are $XQ_{at,C}^{constr}$ and $XQ_{at,C,R}$. The least expressive fragment on the right-hand side is XQ^{upd}. Since we cannot create new nodes in $XQ_{at,C,R}$, it is easy to see that all nodes in the result have to be also in the input store. From Lemma 3.10 and Lemma 3.11 it follows that we cannot write $XQ_{at,C}^{constr}$ programs such that nodes at an arbitrary depth are not deep-equal to nodes in the input store, while in XQ^{upd} this can be done.

***polysize*-border** The most expressive fragment on the left-hand side is $XQ^{upd}_{at,C}$. The least expressive fragments on the right-hand side are XQ_R and XQ^{snap}. From Lemma 3.7, Lemma 3.8 and Lemma 3.9 it follows that the polynomial size bounds for XQ^{upd}_C do not hold for XQ_R and XQ^{snap}.

***count*-border** The most expressive fragments on the left-hande side are XQ^{upd} and XQ_R. The least expressive fragment on the right-hand side is XQ_C. From Lemma 3.5 and Lemma 3.6 we know that we cannot express `count()` in XQ^{upd}. From Lemma 3.1 and Lemma 3.2 it follows that we cannot expression `count()` in XQ_R.

***recursion*-border** The most expressive fragment on the left-hand side is $XQ^{snap}_{at,C}$. The least expressive fragment on the right-hand side is XQ_R. From Lemma 3.12 we know that there are programs in XQ_R that cannot be expressed by $XQ^{snap}_{at,C}$ programs.

All previous results can now be combined to complete the proof:

- If XF_1 and XF_2 are in the same node then it follows that they are equivalent:
  This can easily be shown by the lemmas from Section 3.4.

- If XF_1 and XF_2 are equivalent then they occur in the same node:
  Suppose that XF_1 and XF_2 are not in the same node. It follows from the figure that they are separated by a dotted border and hence we know that there is something in one fragment that you cannot express in the other fragment, so $XF_1 \not\equiv XF_2$.

- If there is a directed path from the node containing XF_1 to the node containing XF_2 then we know that $XF_1 \preceq XF_2$ and since XF_1 and XF_2 appear in a different node they are not equivalent, so $XF_1 \prec XF_2$:
  This follows from the fact that there is a fragment XF_1' equivalent to XF_1 and XF_2' equivalent to XF_2 such that $\mathbf{L}(XF_2') \subseteq \mathbf{L}(XF_1')$.

- If $XF_1 \prec XF_2$ then there is a directed path from the node containing XF_1 to the node containing XF_2:
  Suppose that $XF_1 \prec XF_2$ and there is no directed path from XF_1 to XF_2. Then either there is a directed path from XF_2 to XF_1 such that $XF_2 \prec XF_1$ and hence $XF_1 \not\prec XF_2$ or there is no directed path at all between the nodes of both fragments. In this case we know by inspecting Figure 3.1 that there are (at least) two borders separating the nodes of both fragments where for the first border XF_1 is in the more expressive set of fragments and for the second border XF_2 is in the more expressive set of fragments. Hence XF_1 and XF_2 are incomparable so $XF_1 \not\prec XF_2$.

3.6 Conclusion

The relative expressive power of certain LiXQuery constructs was studied in this chapter. More precisely, we investigated `count` aggregation, position information in `for` loops, recursive function definitions, node construction, update operations and the `snap` operation.

We defined 32 LiXQuery fragments which can be into 12 equivalence classes, i.e., classes including fragments with the same expressive power in terms of the transformations they can perform. The transformation of a program is defined by the mapping of input stores and variable bindings over this store to serialized result sequences. We also proved that the 12 equivalence classes are really different and possess a different degree of expressive power. The smallest fragment (of the 32 fragments studied in this chapter) that has the expressive power of full LiXQuery is XQ_R^{constr}.

CHAPTER 4

Expressing Views and Updates with XPath Transformations

The problem of updating a database through view updates is well-established and has been widely studied in the context of relational databases. In this chapter we study the view-update problem in an XML setting, which consists of finding a systematic translation of updates on the view document to updates on the base document. One desirable property of such an update translation strategy is that it is well-behaved in the sense that a user cannot see the difference between applying an update directly on the view document or applying the translated update on the base document and then recomputing the view, i.e., both results are isomorphic. We consider XPath-based projection views [Vercammen et al., 2006] and a simple propagation update strategy which translates atomic updates on nodes in the view directly to atomic updates on the corresponding nodes in the base document. We show that well-behavedness of this simple propragation update strategy is decidable for the considered update operations and projection views.

4.1 Introduction

A view mechanism allows a DBMS to present to certain users and applications only the data that is relevant for them and allowed for them to see. As such it can play an important role in the access management in a DBMS. However, if users can only access the data through certain views then any update they want to perform has to be done through these views, which means that the database should be able to translate a sufficiently large set of updates on the view to updates on the base tables. Finding such a translation that is both natural and intuitive is what is generally known as the *view-update problem* and has been studied extensively for relational databases [Dayal and Bernstein, 1978, Bancilhon and Spyratos, 1981, Cosmadakis and Papadimitriou, 1984, Masunaga, 1984, Keller, 1985, Gottlob et al., 1988, Tomasic, 1988, Hegner, 1990, Lechtenbörger and Vossen, 2003]. In this chapter we

study this problem for XML databases and views that are defined as a projection of a base XML document, i.e., a restriction of the XML document to a subset of its nodes while maintaining the ancestor-descendant relationships between these nodes. As an example consider the following XML document:

```
<?xml version="1.0"?>
<doctors>
  <doctor name="Meredith Grey">
    <patient name="Will Brown">
      <treatment date="2007−05−24" name="digoxin" qty="0.125 mg"/>
      <treatment date="2007−06−01" name="metoprolol" qty="50 mg"/>
      <treatment date="2007−06−03" name="digoxin" qty="0.125 mg"/>
    </patient>
    <patient name="Mary Johnson">
      <treatment date="2007−06−02" name="premarin" qty="1.25 mg"/>
    </patient>
  </doctor>
  <doctor name="Christina Yang">
    <patient name="Peter Willard">
      <treatment date="2007−06−01" name="warfarin" qty="5 mg"/>
      <treatment date="2007−06−04" name="lanoxin" qty="0.125 mg"/>
    </patient>
  </doctor>
</doctors>
```

This base document describes medecine prescriptions by doctors to hospital patients. The following XML document is one of the many views possible on the base document:

```
<?xml version="1.0"?>
<treatments>
  <doctor name="Meredith Grey">
    <treatment date="2007−05−24" name="digoxin" qty="0.125 mg"/>
    <treatment date="2007−06−01" name="metoprolol" qty="50 mg"/>
    <treatment date="2007−06−03" name="digoxin" qty="0.125 mg"/>
    <treatment date="2007−06−02" name="premarin" qty="1.25 mg"/>
  </doctor>
</treatments>
```

This view document is a projection of the base document where the patient information is removed. Such a projection can be described by an XPath expression that selects all the nodes that are retained and a label for the creation of new root. In this case the new root is given element name `treatments` and the XPath expression might be for example the following:

```
/*/doctor[*[@name="digoxin"]]/(.|@*|patient/*/(.|@*))
```

This means that in the view we only see the treatments of the doctors that have prescribed digoxin.

Let us first consider so-called primitive updates [Chamberlin et al., 2006] on the view, which are simple updates such as for example the renaming of a particular element node, assigning a new value to a particular attribute, removing a certain element and all its descendants, adding a new element with a certain label under a certain old element, and adding a new element with a certain label directly after or before a certain old element. Since the view is essentially a projection of the base document, the most straightforward and intuitive update strategy seems to be the one where the primitive update that is applied to a certain node in the view is mapped to the same update to the corresponding node in the base fragment. For example, under this strategy the deletion of the node describing the metoprolol treatment is translated to the deletion of the metoprolol treatment in the base document, an update of the value of the `qty` attribute node of this treatment node to `0.256 mg` would be translated to the same update on the corresponding attribute node in the base document, and an `insert before` of a new treatment before the metoprolol treatment is translated to an `insert before` of such an element before the metoprolol treatment in the base document. Note that in the latter case the new treatment will be added to the same patient as the metoprolol treatment.

Given the described update strategy we can ask which primitive updates have "side effects" in the view, in the sense that if the translated update is applied to the base document then the recomputed view is not the same as the result of the original update applied to the view document. For example, if we relabel a `treatment` node with `medication` then there are no such side effects, but if we update the name of all the digoxin treatments then all treatments will disappear from the recomputed view.

4.2 Preliminaries

In this section we introduce a simplified version of the XML store mentioned in the LiXQuery data model in the sense that we only consider one tree and one node kind. Moreover, we abstract from the fact that element nodes have a name and attribute nodes by replacing these with a function for that node which maps names to values. We take a look at the view-update problem for this data model and introduce the notion of *well-behavedness* in order to define formally the absence of view side-effects.

In general, we use the following notations: blackboard symbols (e.g. $\mathbb{A}$) for the postulated infinite sets, capitals (e.g. V) for finite sets, lowercase characters (e.g. v) for elements of a set, Greek symbols (e.g. τ) for functions and binary relations, and caligraphic symbols (e.g. $\mathcal{I}$) for the query, view, and update operations we introduce. Capitals are also used for tuples, e.g., T. For binary relations α we write $\alpha(x, y)$ to denote $(x, y) \in \alpha$. For (possibly partial) functions $\alpha : A \to B$ we use, similar to the notation in Chapter 3, $\mathrm{dom}(\alpha)$ to denote the domain, i.e., the set of elements $a \in A$ for which $\alpha(a)$ is defined, and $\mathrm{rng}(\alpha)$ for the range, i.e., the subset of the co-domain B containing the elements b for which there is an element $a \in A$ such that $\alpha(a) = b$. For functions $\alpha : A \to B$ and subsets A' of A, we introduce the shorthand $\alpha(A')$ to denote the set $\{b \mid \exists a \in A' : \alpha(a) = b\}$. Finally, the symbol "$\circ$" denotes the composition of binary relations (and therefore also functions), i.e.,

$(\beta \circ \alpha)(x, y) \Leftrightarrow \exists z : \alpha(x, z) \wedge \beta(z, y)$ and ι_X denotes the identity relation over X.

4.2.1 Data Model

Our data model is a simplification and abstraction of the full XML Data Model[Fernández et al., 2005]. We only have one type of nodes, viz. element nodes, which have attributes[1]. Clearly, element names can be modeled using these attributes. We postulate an infinite set of nodes $\mathbb{V}$, and two disjoint infinite sets $\mathbb{A}$ and $\mathbb{S}$, which are used respectively for attribute names and attribute values. Moreover $\mathbb{S}$ is totally-ordered, countable and non-discrete, i.e., for every two elements s_1 and s_2 from $\mathbb{S}$ it holds that $s_1 \neq s_2 \Leftrightarrow \exists s_3 : s_1 < s_3 < s_2$. For example, $\mathbb{S}$ can be the set of rational numbers or strings.

Definition 4.1 (Document Tree)**.** *An* Attribute Assignment α *is a partial function* $\mathbb{A} \to \mathbb{S}$ *with a finite domain that maps attribute names to attribute values.*
A Document Tree *is a tuple* $T = (V, \lhd, r, \lambda, \prec)$ *such that* $(V, \lhd, r)$ *is a rooted tree with* $V \subset \mathbb{V}$ *a finite set of* nodes*,* $\lhd$ *is the* parent-child relationship *and* $r \in V$ *the root of the tree. Moreover* $\lambda : V \to (\mathbb{A} \to \mathbb{S})$ *labels a node with an attribute assignment, and* $\prec$ *is a total order over* V *that represents the* document order*, i.e., (1) every child is greater than its parent, and (2) if two nodes are siblings then all descendants of the smaller sibling are smaller than the larger sibling.*

In the rest of the chapter, when we talk about trees, we actually mean document trees. In the following we let $\rhd$ denote the inverse relation of $\lhd$, $\lhd^+$ and $\rhd^+$ the transitive closure of respectively $\lhd$ and $\rhd$, and $\lhd^*$ and $\rhd^*$ the reflexive and transitive closure of resp. $\lhd$ and $\rhd$. Moreover, $\dot{\prec}$ denotes the following sibling relation, defined by $(\lhd \circ \rhd) \cap \prec$, and the inverse of this relation is denoted by $\dot{\succ}$. The set of all document trees is denoted by $\mathbb{T}$. Finally, if we write V_T then this denotes the set of nodes V of the tree T and similarly we use the notation $\lhd_T, r_T, \lambda_T$, and $\prec_T$.

Document trees can represent the same XML document but still have other node identifiers. In this case we say that the document trees are (strongly) isomorphic. Similarly, for some documents we can discard the sibling order without changing the semantics. If this is the case then weakly isomorphic document trees are said to represent the same document. More precisely, strong and weak isomorphism is defined as follows.

Definition 4.2 (Strong and Weak Isomorphism)**.** *Let* T_1, T_2 *be two document trees and* ρ *a bijection from* V_{T_1} *to* V_{T_2}*. If* $\lhd_{T_2} = \{(\rho(v_1), \rho(v_2)) \mid v_1 \lhd_{T_1} v_2\}$ *and* $\lambda_{T_2} = \{(\rho(v), \alpha) \mid \lambda_{T_1}(v) = \alpha\}$ *then* ρ *is said to be a* weak isomorphism*. Moreover, if also* $\prec_{T_2} = \{(\rho(v_1), \rho(v_2) \mid v_1 \prec_{T_1} v_2)\}$ *then* ρ *is said to be a* strong isomorphism*. Two trees* T_1, T_2 *are strongly (weakly) isomorphic if there exists a strong (weak) isomorphism between* T_1 *and* T_2*, which is denoted by* $T_1 \equiv T_2$ *(*$T_1 \cong T_2$*).*

Note that in this definition it follows from the constraints for $\lhd_{T_2}$ that $r_{T_2} = \rho(r_{T_1})$.

[1]Note that the XML Data Model considers attributes to have a seperate node type, while in our data model we include attributes in element nodes.

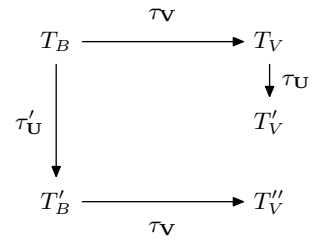


Figure 4.1: General Setting for the XML View-Update Problem

4.2.2 View-Update Problem

A tree transformation is defined as binary relation τ over $\mathbb{T}$ that is generic in terms of node identifiers, i.e., if ρ is a permutation of $\mathbb{V}$, T_1, T_2 two trees and T'_1, T'_2 the result of applying ρ to respectively T_1, T_2 then $\tau(T_1, T_2) \Leftrightarrow \tau(T'_1, T'_2)$. Note that this is a binary relation and not a function because it may be undefined and non-deterministic. The set of all tree transformations is denoted by $\mathbb{F}$. Both update operations and view definitions are defined as tree transformations. Figure 4.1 shows the general setting for the view-update problem, i.e., we have a view definition τ_V which maps a certain tree T_B (base tree) to a tree T_V (view tree) and an update operation τ_U that maps the tree T_V to a new tree $\tau_U(T_V) = T_{V'}$. In general we look for an update τ'_U, given τ_V and τ_U. An update strategy is a (partial) function $\sigma : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ and maps a view definition and an update operation on the view tree to an update operation on the base tree. If σ is the update strategy used in Figure 4.1 then $\tau'_U = \sigma(\tau_V, \tau_U)$.

Definition 4.3 (Well-behavedness)**.** *An update strategy σ is said to be* well-behaved *for a view τ_V and an update τ_U iff for all trees T_1, T_2 it holds that*

- *if $(\tau_V \circ \sigma(\tau_V, \tau_U))(T_1, T_2)$ then there is a tree T_3 such that $(\tau_U \circ \tau_V)(T_1, T_3)$ and $T_2 \equiv T_3$*
- *if $(\tau_U \circ \tau_V)(T_1, T_2)$ then there is a tree T_3 such that $(\tau_V \circ \sigma(\tau_V, \tau_U))(T_1, T_3)$ and $T_2 \equiv T_3$*

In this definition, in the first bullet T_1 corresponds to T_B, T_2 to T''_V and T_3 to T'_V in Figure 4.1 and in the second bullet, T_1 corresponds to T_B, T_2 to T'_V and T_3 to T''_V. Well-behavedness is a desirable property since users of a view do not see side-effects when updating the view.

4.3 Queries, Updates and Views

Many XML query, update and transformation languages use XPath expressions to select nodes within a more complex expression. In this section we introduce the language $\mathbb{P}$ containing path expressions and use this language to define queries, updates and views.

4.3.1 Queries

The language $\mathbb{P}$ is defined as follows:

$$\begin{array}{ll} e ::= & \epsilon \mid p \mid \Uparrow \mid \downarrow^+ \mid \uparrow^+ \mid \twoheadleftarrow \mid \twoheadrightarrow \mid e/e \mid e \cap e \mid e \cup e \mid e - e \\ p ::= & q \mid p \vee p \\ q ::= & c \mid q \wedge q \\ c ::= & a(<|\leq|=|\neq|\geq|>)(a \mid s) \end{array}$$

Where a is a symbol from $\mathbb{A}$ and s from $\mathbb{S}$. The expression p is used for local tests on nodes in a tree, called *predicates*. The symbols $\downarrow^+$, $\uparrow^+$, $\twoheadleftarrow$, and $\twoheadrightarrow$ are used to denote respectively the descendant, ancestor, preceding, and following axis, $\Uparrow$ denotes a jump to the root r_T, e_1/e_2 represents the concatenation of e_1 and e_2, and finally $\cap$, $\cup$ and $-$ represent the set intersection, set union and set difference. For disambiguation, parentheses are added and the concatenation is assumed to have the highest precedence, e.g., $\downarrow^+/a \cup b$ is the same as $(\downarrow^+/a) \cup b$. We now give the semantics of path expressions.

Definition 4.4 (Path Semantics)**.** *The semantics of a predicate p is $\mathcal{L}[p] \subseteq (\mathbb{A} \to \mathbb{S})$ such that $\alpha \in \mathcal{L}[p]$ if α satisfies p. If an attribute in p is not in the domain of α then $\alpha \notin \mathcal{L}[p]$. The semantics of a path expression e is a function $\mathcal{P}[e] : \mathbb{T} \to 2^{\mathbb{V}\times\mathbb{V}}$, defined as follows:*

$$\begin{array}{llll} \mathcal{P}[\epsilon](T) & = \{(v,v) \mid v \in \mathbb{V}\} & \mathcal{P}[\Uparrow](T) & = \{(v, r_T) \mid v \in \mathbb{V}\} \\ \mathcal{P}[p](T) & = \{(v,v) \mid \mathcal{L}[p](\lambda_T(v))\} & & \\ \mathcal{P}[\downarrow^+](T) & = \lhd_T^+ & \mathcal{P}[\uparrow^+](T) & = \rhd_T^+ \\ \mathcal{P}[\twoheadleftarrow](T) & = \succ_T - \rhd_T^+ & \mathcal{P}[\twoheadrightarrow](T) & = \prec_T - \lhd_T^+ \\ \mathcal{P}[e_1/e_2](T) & = \mathcal{P}[e_2](T) \circ \mathcal{P}[e_1](T) & \mathcal{P}[e_1 \cap e_2](T) & = \mathcal{P}[e_1](T) \cap \mathcal{P}[e_2](T) \\ \mathcal{P}[e_1 \cup e_2](T) & = \mathcal{P}[e_1](T) \cup \mathcal{P}[e_2](T) & \mathcal{P}[e_1 - e_2](T) & = \mathcal{P}[e_1](T) - \mathcal{P}[e_2](T) \end{array}$$

Note that the only XPath axes that are syntactically in $\mathbb{P}$ are $\downarrow^+, \uparrow^+, \twoheadleftarrow, \twoheadrightarrow$, and ϵ. These 5 axes select for each node 5 disjoint sets of nodes and we call them strict-recursive axes. All other XPath axes can be expressed by combining these 5 axes. For example, the descendant-or-self axis corresponds to $\downarrow^+ \cup \epsilon$, the child axis to $\downarrow^+ - \downarrow^+/\downarrow^+$ and the following-sibling axis to $\twoheadrightarrow \cap ((\uparrow^+ - \uparrow^+/\uparrow^+)/(\downarrow^+ - \downarrow^+/\downarrow^+))$.

Moreover, note that predicates do not perform any navigation, but just check the attribute assignment of the context node. For example, $a = a$ returns the context node v if the attribute a is defined for v, i.e., $a \in \mathrm{dom}(\lambda_T(v))$.

We now define queries by path expressions to return a set of nodes that we can select by evaluating the path expression against the root.

Definition 4.5 (Query)**.** *Let e be a path expression. The query $\mathcal{Q}[e]$ is a function $\mathbb{T} \to 2^{\mathbb{V}}$, defined as $\mathcal{Q}[e](T) = \{v \mid (r_T, v) \in \mathcal{P}[e](T)\}$.*

4.3.2 Properties

We now give some properties of $\mathbb{P}$ that are used in this chapter. In terms of expressive power, the language $\mathbb{P}$ is related to XPath 2.0 without value comparisons [2], which is expressively complete for first-order queries [Marx, 2005] in the following way: if we ignore attributes, and element names are mapped to a fixed attribute in our data model then this language corresponds to XPath 2.0 without value comparison. However, in Section 4.2 we map attribute nodes of the XML Data Model to properties of an element node in our data model, to which we always have to refer explicitely in path expressions, i.e., the name has to be used and we cannot use wildcards. An implication of this restriction is that, given a path expression e, we can always add an attribute that is not referred to in e, to nodes in a tree T without changing the result of $\mathcal{Q}[e](T)$, while in XPath 2.0 this does not always hold when `@*` occurs in the path expression. Obviously, we can also extend trees by adding nodes in addition to attributes.

Definition 4.6 (Extension)**.** *Let A be a set of attribute names and T_1, T_2 be document trees. Then T_2 is said to be an extension of T_1 w.r.t. A (T_1 is a restriction of T_2 w.r.t. A) if $V_{T_1} \subseteq V_{T_2}$, $\lhd_{T_1}^+ \subseteq \lhd_{T_2}^+$, $\prec_{T_1} \subseteq \prec_{T_2}$, and for all $v \in V_{T_1}$ and $a \in A$ it holds that if $(\lambda_{T_1}(v))(a)$ is defined then $(\lambda_{T_2}(v))(a)$ is defined and $(\lambda_{T_1}(v))(a) = (\lambda_{T_2}(v))(a)$.*

If $A = \mathbb{A}$ then T_1 is said to be a projection of T_2 on V_{T_1}, denoted by $T_2 = \pi_{V_{T_1}}(T)$.

Observe that the notions of extension and restriction define a partial order on the set of document trees. The notions of *minimal extension* and *maximal restriction* of a document are defined with respect to this partial order.

From [Guo et al., 1996] we know that satisfiability for the predicates that we consider, can be decided in linear time. Moreover, if a predicate is satisfiable then there is an attribute assignment which needs only polynomial space.

Lemma 4.1. *If $S \subsetneq \mathbb{S}$ is a finite set of attribute values, p is a satisfiable predicate in $\mathbb{P}$ such that all attribute values used as a constant in p are in S then there exists an attribute assignment α such that $\mathcal{L}[p](\alpha)$ holds and which can be stored in polynomial space in terms of the space needed to store p and S.*

Proof. From the proof of Theorem 3.2.1 of [Guo et al., 1996], it follows that for all satisfiable expressions q of the form $\bigwedge(a_{i,1}(<|\leq|=|\neq|\geq|>)(a_{i,2} \mid s_i))$ we can find for every attribute an open, halfopen or closed interval for which both the left and the right boundaries are either s_i values occuring in the expression or infinity such that for every attribute assignment that maps attributes to values in that interval it holds that p is true. Moreover, we can assume that for every two elements $s_1, s_2 \in \mathbb{S}$ it holds that there is a value $s_3 \in \mathbb{S}$ between s_1 and s_2 and for which we need at most only constant space more than we need for storing s_1 and s_2. For example, if $\mathbb{S}$ is the set of strings, $s_1 =$ "aa" and $s_2 =$ "ab" then s_3 can be "aaa". Finally, for disjunctions $p = q_1 \vee \ldots \vee q_m$ it holds that p is satisfiable if one of the

[2] We can express comparisons between values within the same node, which are essentially predicates, but we cannot express comparisons between values of different nodes which can be used for joins.

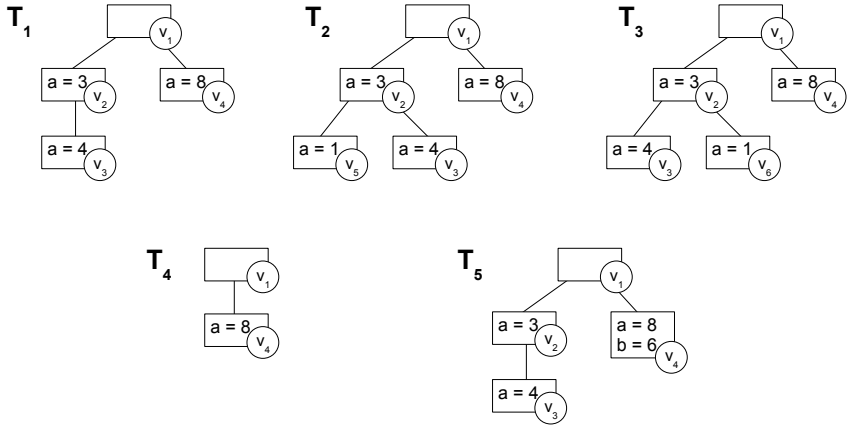


Figure 4.2: Example Document Trees for illustrating update operations

q_i is satisfiable and hence linear space suffices to encode a satisfying attribute assignment if e is satisfiable. □

Moreover [ten Cate and Lutz, 2007] implies the following decidability result.

Lemma 4.2. *Containment of path expressions in $\mathbb{P}$ is decidable, but the decision problem is non-elementary*[3]*.*

Finally, path expressions are generic as defined in database theory for query langauges [Chandra and Harel, 1980], i.e., let p be a path expression, T a document tree, ρ a permutation of $\mathbb{V}$, and T' the document tree obtained by applying ρ to T, then $(v_1, v_2) \in \mathcal{P}[e](T) \Leftrightarrow (\rho(v_1), \rho(v_2)) \in \mathcal{P}[e](T')$. This implies that we cannot observe the difference between isomorphic trees.

4.3.3 Updates

An update is a tree transformation which modifies a tree. We consider three possible updates. An update is either an attribute update, a deletion or an insertion.

The attribute update assigns a value to a certain attribute. If this attribute already existed, then the attribute value is overwritten with the new value.

Definition 4.7 (Attribute Update)**.** *Let e be a path expression, a an attribute name, and s an attribute value. Then $\mathcal{U}[e, a, s](T_1, T_2)$ iff T_2 is a minimal extension w.r.t. $\mathbb{A} - \{a\}$ such that $(\lambda_{T_2}(v))(a) = s' \Leftrightarrow (v \in \mathcal{Q}[e](T_1) \wedge s = s') \vee (v \notin \mathcal{Q}[e](T_1) \wedge (\lambda_{T_1}(v))(a) = s')$.*

For example, consider tree T_1 and T_5 of Figure 4.2. Then T_5 can be obtained from T_1 by performing an attribute update as follows: $\mathcal{U}[\downarrow^+/(a > 3 \wedge a \neq 4), b, 6](T_1, T_5)$.

The deletion deletes all subtrees rooted at the nodes selected by a path expression.

[3]A decision problem is non-elementary if the time needed to solve it cannot be bounded by any exponential tower of constant height.

Definition 4.8 (Deletion)**.** *Let e be a path expression. Then $\mathcal{D}[e](T_1, T_2)$ iff T_2 is a maximal restriction w.r.t. $\mathbb{A}$ such that for every $v_1 \in \mathcal{Q}[e](T_1)$ and $v_2 \in V_{T_1}$ it holds that if $v_1 \lhd^*_{T_1} v_2$ then $v_2 \notin V_{T_2}$.*

For example, consider tree T_1 and T_4 of Figure 4.2. Then T_4 can be obtained from T_1 by performing a deletion as follows: $\mathcal{D}[\downarrow^*/(a < 4 \vee b < 3)](T_1, T_4)$.

Finally, the insertion adds new nodes with a certain attribute assignment as first preceding sibling or as last child of nodes selected by a path expression.

Definition 4.9 (Insertion)**.** *Let e be a path expression, α an attribute assignment, and w one of $\{\blacktriangleleft, \blacktriangledown, \blacktriangleright\}$. Then $\mathcal{I}[e, \alpha, w](T_1, T_2)$ iff T_2 is a minimal extension w.r.t. $\mathbb{A}$ such that for every $v_1 \in \mathcal{Q}[e](T_1)$ there exists a $v_2 \in \mathbb{V} - V_{T_1}$ for which it holds that $\lambda_{T_2}(v_2) = \alpha$ and*

- *if $w = \blacktriangleleft$ then $(v_2, v_1) \in \dot{\prec}_{T_2} - (\dot{\prec}_{T_2} \circ \dot{\prec}_{T_2})$*
- *if $w = \blacktriangledown$ then $v_1 \lhd_{T_2} v_2$*
- *if $w = \blacktriangleright$ then $(v_1, v_2) \in \dot{\prec}_{T_2} - (\dot{\prec}_{T_2} \circ \dot{\prec}_{T_2})$*

For example, consider trees T_1, T_2 and T_3 of Figure 4.2. Then the insertion $\mathcal{I}[\downarrow^+/(a = 3), \{(a, 1)\}, \blacktriangledown]$ can map T_1 to T_2 or T_3. Note that we can add multiple nodes at once, but all newly inserted nodes will be leafs in the result tree. Moreover, it can be shown that if there are no nodes in document order between v_1 and v_2 in a tree, then there can be at most one node between v_1 and v_2 in the document order of the resulting tree.

4.3.4 Views

We only consider views that are projections of trees onto nodes selected by a path expression e and an additional node is added as root.

Definition 4.10 (View)**.** *Let e be a path expression and α an attribute assignment. The view $\mathcal{V}[e, \alpha]$ is a tree transformation from a tree T_1 to another tree T_2 such that T_2 is the projection of T_1 on the result nodes of $\mathcal{Q}[e](T_1)$ where a new node v_r is added as root to ensure that the result is still a tree. This new node gets the attributes from α, i.e., $\lambda_{T_2}(v_r) = \alpha$. More precisely, let T_1 and T_2 be document trees. Then $\mathcal{V}[e, \alpha](T_1, T_2)$ iff there is a 'new' node v_r that only appears in T_2 and a bijection $\rho \subset \mathcal{Q}[e](T_1) \times (V_{T_2} - \{v_r\})$, called* projection relation*, such that*

- $v_1 \lhd^+_{T_2} v_2 \Leftrightarrow (\rho^{-1}(v_1) \lhd^+_{T_1} \rho^{-1}(v_2)) \vee (v_1 = v_r)$
- $r_{T_2} = v_r$
- $\lambda_{T_2}(v) = \alpha' \Leftrightarrow (\alpha = \alpha' \wedge v = v_r) \vee (\alpha' = \lambda_{T_1}(\rho^{-1}(v)))$
- $v_1 \prec_{T_2} v_2 \Leftrightarrow (\rho^{-1}(v_1) \prec_{T_1} \rho^{-1}(v_2) \vee v_1 = v_r)$

Intuitively, the previous definition stated that T_1 restricted to $\mathcal{Q}[e](T_1)$ has to be isomorphic to T_2 without its root. This isomorphism is actually our projection relation.

Proposition 4.1. *The projection relation from Definition 4.10 is unique.*

Proof. Consider the view $\mathcal{V}[e, \alpha]$ and two trees T_1 and T_2 for which it holds that $\mathcal{V}[e, \alpha](T_1, T_2)$. Suppose there are two different projection relations ρ_1, ρ_2 between T_1 and T_2. By definition the domain and range of ρ_1 and ρ_2 are the same. Since $\rho_1 \neq \rho_2$ there must be a node v in T_1 such that $\rho_1(v) = v_1$, $\rho_2(v) = v_2$, and $v_1 \neq v_2$. Now let n denote the number of nodes in the document order $\prec_{T_1}$ before v. Then it can be easily seen that there also must be exactly n nodes in the document order $\prec_{T_2}$ before v_1 and v_2 and since this is a total order, we get the contradiction that $v_1 = v_2$. □

The unique projection relation for a view τ_V and two trees T_1 and T_2 is denoted by $\mathbf{Proj}[T_1, \tau_V, T_2]$.

4.3.5 Additional Notations

We now introduce some auxiliary notations that we use throughout the rest of this chapter. We introduce a shorthand $a \doteq s$ for $(a = s) \cup (\epsilon - (a = a))$, i.e., if a is defined then its value equals s, and $a \doteq a'$ for $(a = a') \cup (\epsilon - ((a = a) \cup (a' = a')))$, i.e., a is defined iff a' is also defined and when they are defined then they have the same value. Finally, $\texttt{empty}(e) = (\Uparrow - e/\Uparrow)$, i.e., the root is returned iff e returns an empty result.

The set of attribute names that occur in a test in a path expression e is denoted by A_e. For update and view operations τ, we define a (finite) set of attribute names A_τ as the set of attribute names that occur in predicates within path expressions or in attribute assignments in τ. More precisely:

$$\begin{aligned} A_{\mathcal{U}[e,a,s]} &= A_e \cup \{a\} \\ A_{\mathcal{D}[e]} &= A_e \\ A_{\mathcal{I}[e,\alpha,w]} &= A_e \cup \mathrm{dom}(\alpha) \\ A_{\mathcal{V}[e,\alpha]} &= A_e \cup \mathrm{dom}(\alpha) \end{aligned}$$

Finally, we introduce a few shorthands for axes that are syntactically not in $\mathbb{P}$:

$$\begin{aligned} \downarrow^* &= \downarrow^+ \cup \epsilon & \uparrow^* &= \uparrow^+ \cup \epsilon \\ \downarrow &= \downarrow^+ - \downarrow^+/\downarrow^+ & \uparrow &= \uparrow^+ - \uparrow^+/\uparrow^+ \\ \dot{\twoheadrightarrow} &= \twoheadrightarrow \cap ((\uparrow^+ - \uparrow^+/\uparrow^+)/(\downarrow^+ - \downarrow^+/\downarrow^+)) & \dot{\twoheadleftarrow} &= \twoheadleftarrow \cap ((\uparrow^+ - \uparrow^+/\uparrow^+)/(\downarrow^+ - \downarrow^+/\downarrow^+)) \end{aligned}$$

4.4 Simple Propagation Update Strategy

The simple propagation update strategy σ_p corresponds intuitively to simply propagating all updates on nodes in the view tree T_V defined by τ_V to nodes in the base tree T_B, i.e., updates on a node v in the view tree are performed in the same way on the node $\mathbf{Proj}^{-1}[T_B, \tau_V, T_V](v)$ in the base tree. Since the update operations are formulated in terms of a path expression e we need to assume a translated path expression $\epsilon(\tau_V, e)$

which select the corresponding nodes in the base tree, i.e., for all trees T_1 it holds that $\mathcal{Q}[\epsilon(\tau_V, e)](T_1) = \{v \mid \exists T_2 : \tau_V(T_1, T_2) \wedge (\exists v' : v' \in Q[e_2](T_2) \wedge \mathbf{Proj}[T_1, \tau_V, T_2](v, v'))\}$. Assuming this translated expression, we now define the simple propagation update strategy.

Definition 4.11 (Simple Propagation Update Strategy)**.** *Let e_1, e_2 be path expressions, α_1, α_2 attribute assignments, and $\tau_V = \mathcal{V}[e_1, \alpha_1]$. The simple propagation update strategy σ_p is defined as follows:*

- $\sigma_p(\mathcal{V}[e_1, \alpha_1], \mathcal{U}[e_2, a, s]) = \mathcal{U}[\epsilon(\tau_V, e_1), a, s]$
- $\sigma_p(\mathcal{V}[e_1, \alpha_1], \mathcal{D}[e_2]) = \mathcal{D}[\epsilon(\tau_V, e_1)]$
- $\sigma_p(\mathcal{V}[e_1, \alpha_1], \mathcal{I}[e_2, \alpha_2, w]) = \mathcal{I}[\epsilon(\tau_V, e_1), \alpha_2, w]$

In the rest of this section, we show that there always exists such a path expression $\epsilon(\tau_V, e)$ by first adding a new operation to $\mathbb{P}$. Afterwards we show that this new operation does not add expressive power.

4.4.1 Adding an Escape to Path Expressions

From Definition 4.10 we know that the root v_r of the view tree T_V has no "corresponding node" v in the base tree T_B such that $\mathbf{Proj}[T_B, \tau_V, T_V](v, v_r)$. However, path expressions on the view tree can use this node to compute their result and to perform the computation on the base tree we simulate v_r. We do this by defining a new relation $\rho[T_B, \tau_V, T_V] = \mathbf{Proj}[T_B, \tau_V, T_V] \cup \{(v, v_r) \mid v \in \mathbb{V} - V_{T_B}\}$ and adding to $\mathbb{P}$ an operation to jump to nodes that are not in the tree against which we are evaluating the path expression. This operation is called *escape*, is denoted by $\diamondsuit$ and the extension of the language $\mathbb{P}$ including this operation is referred to as $\mathbb{P}_\diamondsuit$. The semantics of this operation is defined as follows: $\mathcal{P}[\diamondsuit](T) = \{(v_1, v_2) \mid v_1 \in \mathbb{V} \wedge v_2 \in \mathbb{V} - V_T\}$.

Adding the $\diamondsuit$ operation does not allow us to express more queries iff we restrict queries to only return nodes that are in the tree against which we are evaluating the path expression.

Lemma 4.3. *For every expression e_1 in $\mathbb{P}_\diamondsuit$ there is an expression e_2 in $\mathbb{P}$ such that $\mathcal{Q}[e_1 \cap \downarrow^*] = \mathcal{Q}[e_2]$.*

Proof. The semantics of a path expression is a relation between nodes, i.e., $\mathcal{P}[e](T) \subseteq 2^{\mathbb{V} \times \mathbb{V}}$. The expression $\epsilon^I = \epsilon \cap \Uparrow/\downarrow^*$ is the restriction of the self axis to nodes in the tree and $\epsilon^O = \epsilon \cap \diamondsuit$ is the restriction of the self axis to nodes outside the tree. We show how to obtain 4 path expressions, e^{II}, e^{IO}, e^{OI}, and e^{OO} such that $\mathcal{P}[e] = \mathcal{P}[(\epsilon^I/e^{II}) \cup (\epsilon^I/e^{IO}/\diamondsuit) \cup (\epsilon^O/e^{OI}) \cup (\epsilon^O/e^{OO}/\diamondsuit)]$ and e^{II}, e^{IO}, e^{OI}, and e^{OO} do not contain the $\diamondsuit$ operation, i.e., they are in $\mathbb{P}$. Let $\emptyset$ be a shorthand for a unsatisfiable path expression, e.g., $\Uparrow/\uparrow^+$. Figure 4.3 shows how to rewrite a path expression into this form by induction. The correctness of this rewriting can easily be verified. Note that this rewriting causes an exponential blow-up. □

Note that $\mathbb{P}_\diamondsuit$ might be exponentially more succinct than $\mathbb{P}$, but we could not establish this.

e	e^{II}	e^{IO}	e^{OI}	e^{OO}
$\downarrow^+$	$\downarrow^+$	$\emptyset$	$\emptyset$	$\emptyset$
$\uparrow^+$	$\uparrow^+$	$\emptyset$	$\emptyset$	$\emptyset$
$\twoheadleftarrow$	$\twoheadleftarrow$	$\emptyset$	$\emptyset$	$\emptyset$
$\twoheadrightarrow$	$\twoheadrightarrow$	$\emptyset$	$\emptyset$	$\emptyset$
p	p	$\emptyset$	$\emptyset$	$\emptyset$
ϵ	ϵ	$\emptyset$	$\emptyset$	ϵ
$\Uparrow$	$\Uparrow$	$\emptyset$	$\Uparrow$	$\emptyset$
$\diamondsuit$	$\emptyset$	$\Uparrow$	$\emptyset$	$\Uparrow$
e_1/e_2	$e_1^{II}/e_2^{II} \cup e_1^{IO}/e_2^{OI}$	$e_1^{IO}/e_2^{OO} \cup e_1^{II}/e_2^{IO}$	$e_1^{OI}/e_2^{II} \cup e_1^{OO}/e_2^{OI}$	$e_1^{OO}/e_2^{OO} \cup e_1^{OI}/e_2^{IO}$
$e_1 \cap e_2$	$e_1^{II} \cap e_2^{II}$	$e_1^{IO} \cap e_2^{IO}$	$e_1^{OI} \cap e_2^{OI}$	$e_1^{OO} \cap e_2^{OO}$
$e_1 \cup e_2$	$e_1^{II} \cup e_2^{II}$	$e_1^{IO} \cup e_2^{IO}$	$e_1^{OI} \cup e_2^{OI}$	$e_1^{OO} \cup e_2^{OO}$
$e_1 - e_2$	$e_1^{II} - e_2^{II}$	$e_1^{IO} - e_2^{IO}$	$e_1^{OI} - e_2^{OI}$	$e_1^{OO} - e_2^{OO}$

Figure 4.3: Rewriting $\mathbb{P}_\diamondsuit$ to $\mathbb{P}$ expressions.

4.4.2 View Composition

The simple propagation update strategy assumes a path expression e_3 to select the nodes v for which there is a v' such that $\mathbf{Proj}[T_B, \tau_V, T_V](v, v')$ is in the result of $\mathcal{Q}[e_2] \circ \mathcal{V}[e_1, \alpha_1]$. We now show that there always exists such an e_3 and hence σ_p is defined for all views and updates that we consider.

Lemma 4.4. *For every path expression e_1, e_2 in $\mathbb{P}$ and attribute assignment α there exists a path expression $e_3 = \epsilon(e_1, \mathcal{V}[e_2, \alpha])$.*

Proof. We prove this lemma by constructing such an expression e_3. We denote by $(e_1, \alpha) : e_2$ the path expression for which it holds that $\forall T : (v_1, v_2) \in \mathcal{P}[(e_1, \alpha) : e_2](T) \Leftrightarrow (\forall T' : \exists v_1', v_2' \in V_{T'} : (\mathcal{V}[e_1, \alpha](T, T') \wedge \{(v_1, v_1'), (v_2, v_2')\} \subseteq \rho[T_B, \tau_V, T_V]\}$ where $\rho[T_B, \tau_V, T_V]$ is $\mathbf{Proj}[T, \tau_V, T'] \cup \{(v_1, v_2) \mid v_1 \notin T \wedge v_2 = r_{T'}\}$. The following table shows how this path expression $(e_1, \alpha) : e_2$ can be constructed.

$$
\begin{array}{lll}
(e_1,\alpha) : \downarrow^+ & = (\epsilon \cap \diamondsuit)/\Uparrow/e_1 \cup (\epsilon \cap \Uparrow/e_1)/(\downarrow^+ \cap \Uparrow/e_1) & \\
(e_1,\alpha) : \uparrow^+ & = (\epsilon \cap \Uparrow/e_1)/((\uparrow^+ \cap \Uparrow/e_1) \cup \diamondsuit) & \\
(e_1,\alpha) : \twoheadrightarrow & = (\epsilon \cap \Uparrow/e_1)/(\twoheadrightarrow \cap \Uparrow/e_1) & \\
(e_1,\alpha) : \twoheadleftarrow & = (\epsilon \cap \Uparrow/e_1)/(\twoheadleftarrow \cap \Uparrow/e_1) & \\
(e_1,\alpha) : \Uparrow & = (\epsilon \cap (\diamondsuit \cup \Uparrow/e_1))/\diamondsuit) & \\
(e_1,\alpha) : \epsilon & = (\epsilon \cap (\diamondsuit \cup \Uparrow/e_1)) & \\
(e_1,\alpha) : p & = (\epsilon \cap \Uparrow/e_1/p) & (\neg\mathcal{L}[p](\alpha)) \\
 & = (\epsilon \cap (\diamondsuit \cup \Uparrow/e_1/p)) & (\mathcal{L}[p](\alpha)) \\
(e_1,\alpha) : (e_2/e_3) & = (e_1,\alpha) : e_2/(e_1,\alpha) : e_3 & \\
(e_1,\alpha) : (e_2 \cap e_3) & = (e_1,\alpha) : e_2 \cap (e_1,\alpha) : e_3 & \\
(e_1,\alpha) : (e_2 \cup e_3) & = (e_1,\alpha) : e_2 \cup (e_1,\alpha) : e_3 & \\
(e_1,\alpha) : (e_2 - e_3) & = (e_1,\alpha) : e_2 - (e_1,\alpha) : e_3 &
\end{array}
$$

Note that when we jump to the root in the view tree, we jump out of the base tree in $(e_1, \alpha) : e_2$ and $\epsilon \cap \diamondsuit$ checks for nodes outside the base tree to see whether the root of the view was in the result of the path expression on the view tree. Moreover, the correctness from the axes follows from the fact that a view is basically a projection and for all 5 axes it holds that a projection preserves the relative position of two (projected) nodes w.r.t. these

axes. Finally, since only nodes within the tree have attributes, we compute the result of predicates for the root in the view tree during the construction of $(e_1, \alpha) : e_2$, which can be done since the attribute assignment α for the constructed root node of the view tree is given. □

From this result now follows that for every view and update, the simple propagation update strategy defines a translated update on the view. Note that in all transformations defined in Section 4.3, only path expressions in $\mathbb{P}$ are allowed.

Corollary 4.1. *For every view τ_V and update τ_U there is an update τ'_U such that $\sigma_p(\tau_V, \tau_U) = \tau'_U$.*

4.5 Deciding Well-Behavedness

We show that for all updates and views we can reduce checking well-behavedness to deciding satisfiability. First we define a configuration which contains the five trees of Figure 4.1 for a given view and update. Deciding whether σ_p is well-behaved for a certain view and update is reduced to deciding whether a certain configuration tree exists, which can be checked by checking satisfiability of a path expression.

4.5.1 Configurations

Update strategies define 5-ary relations between trees, as is shown in Figure 4.1. Hence, it can also be looked at as an infinite set of 5-tuples. A configuration is then defined as such a 5-tuple for the simple propagation update strategy.

Definition 4.12 (Configuration)**.** *A* configuration *C for an update τ_U and view τ_V is a 5-tuple of document trees $(T_B, T'_B, T_V, T'_V, T''_V)$ such that $\sigma_p(\tau_V, \tau_U)(T_B, T'_B)$, $\tau_V(T_B, T_V)$, $\tau_U(T_V, T'_V)$, and $(\tau_V \circ \sigma_p(\tau_V, \tau_U))(T_B, T''_V)$.*
We also define four properties for configurations:

- *C is* conflicting *if $T'_V \ncong T''_V$ or $\mathbf{Old}_C^{V'} \neq \mathbf{Old}_C^{V''}$, with*
  - *$\mathbf{Old}_C^{V'} = \{v \in V_{T_B} \mid \exists v' \in V_{T'_V} \wedge \mathbf{Proj}[T_B, \tau_V, T_V](v, v')\}$, i.e., the nodes of the base tree which are projected and in the updated view T'_V*
  - *$\mathbf{Old}_C^{V''} = \{v \in V_{T_B} \mid \exists v' \in V_{T''_V} \wedge \mathbf{Proj}[T'_B, \tau_V, T''_V](v, v')\}$, i.e., the nodes of the base tree which are in the view after updating the base tree.*
- *C is* node-conservative *if $r_{T_V} = r_{T''_V}$, and the projection relations $\mathbf{Proj}[T_B, \tau_V, T_V]$ and $\mathbf{Proj}[T'_B, \tau_V, T''_V]$ are both a restriction of the identity relation.*
- *C is* node-creation independent *if it is node-conservative and $(V_{T'_V} - V_{T_V}) \cap (V_{T'_B} - V_{T_B}) = \emptyset$.*

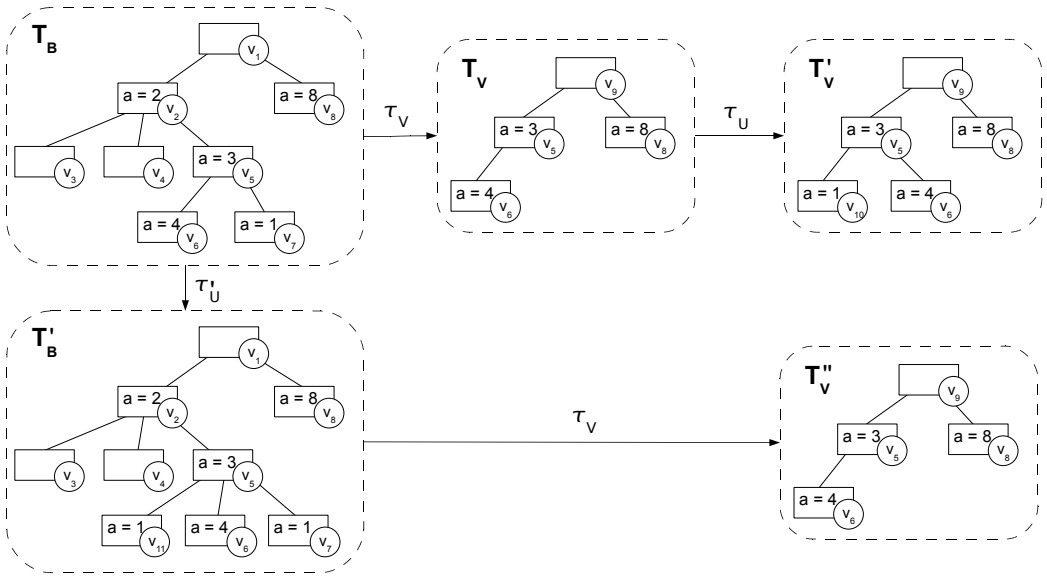


Figure 4.4: Node-creation independent and attribute-minimal configuration.

- *C is* attribute-minimal *if the domain of attribute assignments for all nodes in all five trees is restricted to attributes that are refered to in τ_V or τ_U.*

The notion of conflicting is introduced to correspond to configurations for which T''_V is too different from T'_V, i.e., configurations that we will use later on to construct a counterexample to show that σ_p is not well-behaved for τ_V and τ_U. The other three properties for configurations are defined to help us find such a conflicting configuration by reducing the search space as we will show in Lemma 4.8.

We now illustrate Definition 4.12 by giving an example. Consider the view $\tau_V = \mathcal{V}[(\downarrow^+/a > 2), \emptyset]$ and update over the view $\tau_U = \mathcal{I}[(\downarrow^+/a = 4), \blacktriangleleft, \{(a, 1)\}]$. The view contains all nodes of the base tree with an a attribute that has a value greater than 2 and the update adds a new node with one attribute, i.e., $a = 1$, directly before nodes with an a value equal to 4. The translated update on the view tree is, according to the translation of Lemma 4.4, $\tau'_U = \mathcal{I}[((\epsilon \cap \diamondsuit)/\Uparrow/\downarrow^+/a > 2 \cup (\epsilon \cap \Uparrow/\downarrow^+/a > 2)/(\downarrow^+ \cap \Uparrow/\downarrow^+/a > 2))/(\epsilon \cap \Uparrow/\downarrow^+/a > 2/a = 4), \blacktriangleleft, \{(a, 1)\}]$. According to Lemma 4.3 this can be translated to have a path expression in $\mathbb{P}$ and it can be easily verified that in this example $\tau'_U = \tau_U$. Consider now the configuration of Figure 4.4. The mapping between the graphical representation and document trees is straightforward. This configuration is node-conservative and attribute-minimal for τ_U and τ_V. Moreover, the configuration is also node-creation independent. Finally, it can be easily seen that T''_V and T'_V are not weakly isomorphic and hence this configuration is conflicting.

The notion of conflicting configuration will be used to determine whether the simple propogation update strategy is well-behaved for a certain view and update. We first give some properties of configurations for certain update operations.

Lemma 4.5. *If τ_V is $\mathcal{V}[e_1, \alpha_1]$ and τ_U is one of $\mathcal{D}[e_2]$, $\mathcal{U}[e_2, a, s]$, $\mathcal{I}[e_2, \alpha_2, \blacktriangleleft]$, or $\mathcal{I}[e_2, \alpha_2, \blacktriangleright]$, then for all configurations $C_1 = (T_{B,1}, T'_{B,1}, T_{V,1}T'_{V,1}, T''_{V,1})$ and $C_2 = (T_{B,2}, T'_{B,2}, T_{V,2}T'_{V,2}, T''_{V,2})$ for τ_V and τ_U it holds that $T_{B,1} \equiv T_{B,2}$ implies $T'_{B,1} \equiv T'_{B,2}$, $T_{V,1} \equiv T_{V,2}$, $T'_{V,1} \equiv T'_{V,2}$, and $T''_{V,1} \equiv T''_{V,2}$.*

Proof. It can easily be seen that view trees are strong isomorphic when the base trees are strong isomorphic since the evaluation of path expressions is generic. Moreover, these four update operations are deterministic up to the node identity of the newly created nodes and hence they return strong isomorphic results when applied on strong isomorphic trees. □

Lemma 4.6. *If τ_V is $\mathcal{V}[e_1, \alpha_1]$ and τ_U is $\mathcal{I}[e_2, \alpha_2, \blacktriangledown]$ then for all configurations $C_1 = (T_{B,1}$, $T'_{B,1}$, $T_{V,1}T'_{V,1}, T''_{V,1})$ and $C_2 = (T_{B,2}, T'_{B,2}, T_{V,2}T'_{V,2}, T''_{V,2})$ for τ_V and τ_U it holds that $T_{B,1} \equiv T_{B,2}$ implies $T'_{B,1} \cong T'_{B,2}$, $T_{V,1} \equiv T_{V,2}$, and $T'_{V,1} \cong T'_{V,2}$.*

Proof. Since view trees are strong isomorphic when the base trees are strong isomorphic, we know that $T_{V,1} \equiv T_{V,2}$. The insertion under a node chooses a position among its children in a non-deterministic manner and nodes can only be inserted as leafs into a tree. Hence, $T'_{V,1} \cong T'_{V,2}$ and $T'_{B,1} \cong T'_{B,2}$. □

Lemma 4.7. *The simple propagation update strategy is well-behaved for a view τ_V and update τ_U iff there exists no conflicting configuration for τ_V and τ_U.*

Proof. First assume there exists a conflicting configuration $C_1 = (T_{B,1}, T'_{B,1}, T_{V,1}, T'_{V,1}, T''_{V,1})$. Assume $\mathbf{Old}_C^{V'} \neq \mathbf{Old}_C^{V''}$ and $T'_V \cong T''_V$. Then there must be at least one node $v \in \mathbf{Old}_C^{V'} - \mathbf{Old}_C^{V''}$ which is weakly isomorphic with $v' \in \mathbf{Old}_C^{V''} - \mathbf{Old}_C^{V'}$. Let a be an attribute not in $A_{\tau_V} \cup A_{\tau_U}$. Now consider configuration $C_2 = (T_{B,2}, T'_{B,2}, T_{V,2}, T'_{V,2}, T''_{V,2})$, where in all trees the attribute assignment of v (if it is in the tree) is the attribute assignment of v in the corresponding tree in C_1, extended by adding $a = s$ for some fixed $s \in \mathbb{S}$. Then it can be easily seen that C_2 is indeed a configuration for τ_V and τ_U, since a does not influence the result of the view or update. Moreover, since v' is not changed, it follows that $T'_{V,2} \not\equiv T''_{V,2}$. Hence if there exists a conflicting configuration, then there also exists a configuration $C_1 = (T_B, T'_B, T_V, T'_V, T''_V)$ with $T'_V \not\cong T''_V$ and hence $T'_V \not\equiv T''_V$. By Lemma 4.6 and 4.5 we know that all configurations with T_B as base tree have updated view trees weak equivalent with T'_V and since $T''_V \not\cong T'_V$ we know that there exists no configuration such that the updated view tree, which is weak equivalent with T'_V, is strong equivalent with T''_V and therefore σ_p is not well-behaved for τ_V and τ_U.

We now assume σ_p is not well-behaved for τ_V and τ_U and there are no conflicting configurations. Then there exists a tree T such that for some configuration $C_1 = (T, T'_{B,1}, T_{V,1}T'_{V,1}, T''_{V,1})$ it holds that for all configurations $C_2 = (T, T'_{B,2}, T_{V,2}T'_{V,2}, T''_{V,2})$ it holds that either $T'_{V,1} \not\equiv T''_{V,2}$ or $T'_{V,2} \not\equiv T''_{V,1}$. Moreover, since C_1 and C_2 are both not conflicting we know that $\mathbf{Old}_{C_1}^{V'} = \mathbf{Old}_{C_1}^{V''}$, $\mathbf{Old}_{C_2}^{V'} = \mathbf{Old}_{C_2}^{V''}$, $T''_{V,1} \cong T'_{V,1}$ and $T''_{V,2} \cong T'_{V,2}$.

From Lemma 4.6 and 4.5 it follows that $T_{V,1} \equiv T_{V,2}$, $T'_{V,1} \cong T'_{V,2}$ and it can be easily verified that as a consequence $\mathbf{Old}_{C_1}^{V'} = \mathbf{Old}_{C_2}^{V'}$, since the path expressions in the update statements are evaluated on strong isomorphic trees and hence either the same "old" nodes

are deleted or new nodes are added which cannot be in $\mathbf{Old}_{C_i}^{V'}$ for $i = 1, 2$. Hence it follows that $\mathbf{Old}_{C_1}^{V'} = \mathbf{Old}_{C_1}^{V''} = \mathbf{Old}_{C_2}^{V'} = \mathbf{Old}_{C_2}^{V''}$ and $T''_{V,1} \cong T'_{V,1} \cong T''_{V,2} \cong T'_{V,2}$.

- $\tau_U = \mathcal{D}[e_2]$: Obviously $T'_{V,1}$, $T''_{V,1}$, $T'_{V,2}$, $T''_{V,2}$ only contain "old" nodes and one root node, which does not occur in the base tree. The set of old nodes is the same in all these four trees and their attribute assignment is the same as well, since deletion does not change this. Since the view root node cannot be changed because of the weak isomorphisms, it follows that all these four trees are strong isomorphic, which is a contradiction with the assumption that σ_p is not well-behaved for τ_V and τ_U.

- $\tau_U = \mathcal{U}[e_2, a, s]$: Similar to $\mathcal{D}[e_2]$, but now an attribute update is performed on the same subset of old nodes and hence all four trees are strong isomorphic, resulting in the same contradiction.

- $\tau_U = \mathcal{I}[e_2, \alpha_2, w]$: It can be easily seen that $T'_{V,1}$, $T''_{V,1}$, $T'_{V,2}$, and $T''_{V,2}$ all contain the same number of nodes because they are weak isomorphic. Moreover, they contain the same number of newly inserted nodes, i.e., for every $i = 1, 2$ and every node $v_{s,i}$ that is added under a node $v_{t,i}$ in the update from $T_{V,i}$ to $T'_{V,i}$, there is a node $v'_{s,i}$ added in $T'_{B,i}$ under a node $v'_{t,i}$ such that $\mathbf{Proj}[T_{B,i}, \tau_V, T_{V,i}](v'_{t,i}, v_{t,i})$, and there is a node $v''_{s,i}$ in $T''_{B,i}$ such that $\mathbf{Proj}[T_{V,i}](v'_{s,i}, v''_{s,i})$. Since C_2 can be any configuration and is also not conflicting, we can assume that the nodes added in $T'_{B,2}$ are added at the same relative position under the same parent node, i.e., $v'_{s,1} = v'_{s,2}$ and hence $T'_{B,2}$ is strong isomorphic with $T'_{B,1}$. As a consequence the view trees of the updated base trees are also strong isomorphic, i.e., $T''_{V,1} = T''_{V,2}$.

This shows that if σ_p is not well-behaved for τ_V and τ_U then there exists a conflicting configuration. □

Since we cannot observe the actual node identifiers using path expressions, the choice of newly created nodes does not matter. Moreover, if the base trees of two configurations C_1 and C_2 are extensions of each other w.r.t. $A_{\tau_V} \cup A_{\tau_U}$ then C_1 is a conflicting configuration iff C_2 is a conflicting configuration.

Lemma 4.8. *If there exists a conflicting configuration for τ_V and τ_U then there also exists a conflicting node-creation independent and attribute-minimal configuration for τ_V and τ_U.*

Proof. The node identifiers are chosen non-deterministicaly when computing views and updates, and path expressions are generic. Hence there is a node-creation independent configuration C_2 for which all trees are isomorphic to the trees of C_1 and hence C_2 is also conflicting. Let C_3 be C_2 where in all trees we restrict the domain of the attribute assignments in nodes to $A_{\tau_V} \cup A_{\tau_U}$. Since this restriction does not influence the evaluation of path expressions in the views and updates, it follows that C_3 is also a conflicting configuration. □

From the Lemma 4.7 and Lemma 4.8 we now know that we can reduce deciding well-behavedness as follows.

Corollary 4.2. *The simple propagation update strategy is well-behaved for a view τ_V and update τ_U iff there exists no conflicting, node-creation independent and attribute-minimal configuration for τ_V and τ_U.*

4.5.2 Configuration Trees

We now define configuration trees and show that every node-creation independent and attribute-minimal configuration can be represented by a single configuration tree, i.e., a tree that is a common extension of all trees. We also show that there is a bijective mapping between configurations and configuration trees and how we can translate path expressions on trees of a configuration to path expressions on the configuration tree.

Definition 4.13 (Configuration Tree)**.** *Let τ_V be a view and τ_U an update. Moreover, let $A = A_{\tau_U} \cup A_{\tau_V}$ and $s_t \in \mathbb{S}$. A tree T is a* configuration tree *for τ_V and τ_U if there exists a node-creation independent and attribute-minimal configuration $C = (T_B, T'_B, T_V, T'_V, T''_V)$ for τ_U and τ_V, 4 disjoint sets $A_{old}, A_{new}, A_{tree}, A_{root} \subset \mathbb{A}$, bijections ν_{old} between A and A_{old} and ν_{new} between A and A_{new}, $A_{tree} = \{a_B, a'_B, a_V, a'_V, a''_V\}$, $|A_{tree}| = 5$, and $A_{root} = \{a_{root}\}$, such that T is a minimal common extension of all trees of C w.r.t. $\emptyset$, for which it holds that*

- *$r_T = r_{T_V}$, i.e., the root of the view tree is the root of the configuration tree.*
- *$v \in V_T \Leftrightarrow v \in (V_{T_B} \cup V_{T'_B} \cup V_{T_V} \cup V_{T'_V} \cup V_{T''_V})$, i.e., only nodes that are in a tree of the configuration are in the configuration tree.*
- *The attribute assignment w.r.t. A_{tree} is as follows:*
  - *$v \in V_{T_B} \Leftrightarrow (\lambda_T(v))(a_B) \geq s_t$,*
  - *$v \in V_{T'_B} \Leftrightarrow (\lambda_T(v))(a'_B) \geq s_t$,*
  - *$v \in V_{T_V} \Leftrightarrow (\lambda_T(v))(a_V) \geq s_t$,*
  - *$v \in V_{T'_V} \Leftrightarrow (\lambda_T(v))(a'_V) \geq s_t$,*
  - *$v \in V_{T''_V} \Leftrightarrow (\lambda_T(v))(a''_V) \geq s_t$, and*
  - *$v \in V_T \Leftrightarrow A_{tree} \subseteq \mathrm{dom}(\lambda_T(v))$*

  This means that attributes in A_{tree} indicate in which tree v appears by having a value relative to the treshold value s_t.
- *a_{root} is only defined in the root of the base tree, i.e., $a_{root} \in \mathrm{dom}(\lambda_T(v)) \Leftrightarrow v = r_{T_B}$*
- *if $\alpha_1 = \lambda_{T_B}(v)$ $(\lambda_{T_V}(v))$ and $\alpha_2 = \lambda_{T_C}(v)$ then $a \in \mathrm{dom}(\alpha_1) \Leftrightarrow \alpha_2(\nu_{old}(a)) = \alpha_1(a)$, i.e., "old values" for a are now assigned to $\nu_{old}(a)$.*
- *if $\alpha_1 = \lambda_{T'_B}(v)$ $(\lambda_{T'_V}(v))$ and $\alpha_2 = \lambda_{T_C}(v)$ then $a \in \mathrm{dom}(\alpha_1) \Leftrightarrow \alpha_2(\nu_{new}(a)) = \alpha_1(a)$, i.e., "new values" for a are now assigned to $\nu_{new}(a)$.*

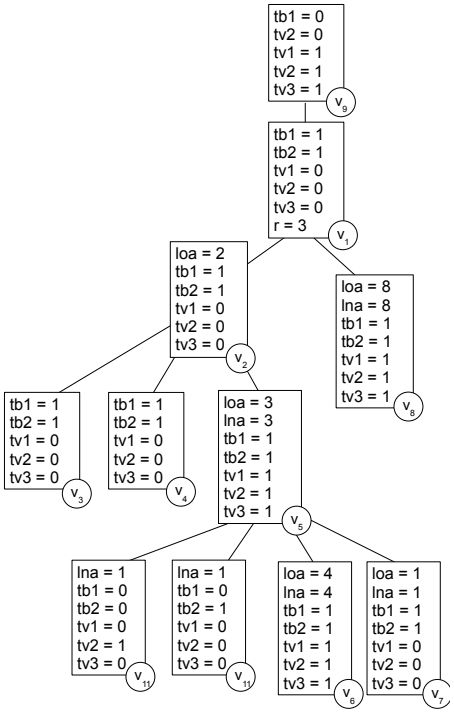


Figure 4.5: A Configuration Tree for the Configuration of Figure 4.4

To illustrate this definition, Figure 4.5 shows a possible configuration tree for the configuration C of Figure 4.4, with $A = \{a\}, A_{new} = \{lna\}$, $A_{old} = \{loa\},$, $A_{tree} = \{tb1, tb2, tv1, tv2, tv3\}$, $A_{root} = \{r\}$, and $s_t = 1$.

The projection T' of T on the nodes for which a_B is defined, corresponds structuraly to T_B, but attributes a of T_B are mapped to attributes $\nu_{old}(a)$ in T' and some new attributes occur in T' that do not correspond to attributes in T_B. We say that T_B is a *configuration restriction* of T for attribute a_B w.r.t. mapping ν_{old} and T' is a *configuration restriction* of T for attribute a_B w.r.t. the identity relation $\iota_{\mathbb{A}}$. In a similar way it holds that T_V is a configuration restriction for a_V of T w.r.t mapping ν_{old} and T'_B, T'_V, T''_V are configuration restrictions for respectively a'_B, a'_V, a''_V w.r.t. ν_{new}. More precisely, we define a configuration restriction as follows:

Definition 4.14 (Configuration Restriction)**.** *Assume $a \in \mathbb{A}$, $\nu : \mathbb{A} \to \mathbb{A}$ and $T \in \mathbb{T}$. Let T' be the projection of T on the nodes for which a is defined. A tree T'' is a configuration restriction of T for attribute a w.r.t. mapping ν iff T' can be obtained from T'' by changing the attribute assignments for every node v such that $(\lambda_{T''}(v))(a') = s \Leftrightarrow a' \in \mathrm{dom}(\nu) \wedge (\lambda_{T'}(v))(\nu(a')) = s$*

If T is a configuration tree then we can identify 3 disjoint subsets V_{root}, V_{old}, and V_{new} of V_T, such that

- $V_{root} = \{r_T\}$,

- $V_{old} = \{v \mid v \neq r_T \wedge (((\lambda_T(v))(a_B) \geq s_t) \vee ((\lambda_T(v))(a_V) \geq s_t))\}$, and
- $V_{new} = V_T - (V_{root} \cup V_{old})$.

The nodes in these sets are refered to as respectively root nodes, old nodes and new nodes of the configuration tree.

Lemma 4.9. *Every node-creation independent and attribute minimal configuration defines a configuration tree.*

Proof. First, it is easy to see that for every node-creation independent configuration $C = (T_B, T'_B, T_V, T'_V, T''_V)$ there exists a minimal common extension w.r.t. $\emptyset$, since there are no two nodes v_1, v_2 such that in one tree T_1 of the configuration C the node v_1 is a descendant (follower) of n_2 and in another tree T_2 of C the node v_1 is not a descendant (follower) of v_2 while both v_1, v_2 occur in T_2. This minimal common extension is called T_C. The attribute labelling for nodes v of T_C is as follows: $\lambda(v) = \{(a, s) \mid \nu^{-1}_{old}(a) = a' \wedge \exists \alpha \in \{\lambda_{T_B}(v), \lambda_{T_V}(v)\} : \alpha(a') = s\} \cup \{(a, s) \mid \nu^{-1}_{new}(a) = a' \wedge \exists \alpha \in \{\lambda_{T'_B}(v), \lambda_{T'_V}(v), \lambda_{T''_V}(v)\} : \alpha(a') = s\}$. □

In a configuration, path expressions are evaluated against T_B, T_V, and T'_B in order to compute updated and view trees. Since a configuration tree T is an extension of the trees of a configuration C and since we need to be able to evaluate path expressions e against the three previously mentioned trees of C for consistency checks, we need a systematic way for each of these 3 trees to translate path expressions over trees of C to path expressions over T. For this we define a function $\gamma_B : \mathbb{P} \to \mathbb{P}$ such that $\mathcal{P}[e](T_B) = \mathcal{P}[\gamma_B(e)](T_C)$, and similarly we define functions γ'_B, and γ_V. If p is a predicate then p_{old} (p_{new}) is p with attribute names a replaced by $\nu_{old}(a)$ ($\nu_{new}(a)$). Using this notation, we define the 3 translation functions as follows:

e	$\gamma_B(e)$	$\gamma'_B(e)$	$\gamma_V(e)$
$\uparrow^+$	$(a_B \geq s_t)/\uparrow^+/(a_B \geq s_t)$	$(a'_B \geq s_t)/\uparrow^+/(a'_B \geq s_t)$	$(a_V \geq s_t)/\uparrow^+/(a_V \geq s_t)$
$\downarrow^+$	$(a_B \geq s_t)/\downarrow^+/(a_B \geq s_t)$	$(a'_B \geq s_t)/\downarrow^+/(a'_B \geq s_t)$	$(a_V \geq s_t)/\downarrow^+/(a_V \geq s_t)$
$\twoheadleftarrow$	$(a_B \geq s_t)/\twoheadleftarrow/(a_B \geq s_t)$	$(a'_B \geq s_t)/\twoheadleftarrow/(a'_B \geq s_t)$	$(a_V \geq s_t)/\twoheadleftarrow/(a_V \geq s_t)$
$\twoheadrightarrow$	$(a_B \geq s_t)/\twoheadrightarrow/(a_B \geq s_t)$	$(a'_B \geq s_t)/\twoheadrightarrow/(a'_B \geq s_t)$	$(a_V \geq s_t)/\twoheadrightarrow/(a_V \geq s_t)$
ϵ	ϵ	ϵ	ϵ
$\Uparrow$	$\Uparrow/\downarrow$	$\Uparrow/\downarrow$	$\Uparrow$
p	$p_{old}/(a_B \geq s_t)$	$p_{new}/(a'_B \geq s_t)$	$p_{old}/(a_V \geq s_t)$
e_1/e_2	$\gamma_B(e_1)/\gamma_B(e_2)$	$\gamma'_B(e_1)/\gamma'_B(e_2)$	$\gamma_V(e_1)/\gamma_V(e_2)$
$e_1 \cap e_2$	$\gamma_B(e_1) \cap \gamma_B(e_2)$	$\gamma'_B(e_1) \cap \gamma'_B(e_2)$	$\gamma_V(e_1) \cap \gamma_V(e_2)$
$e_1 \cup e_2$	$\gamma_B(e_1) \cup \gamma_B(e_2)$	$\gamma'_B(e_1) \cup \gamma'_B(e_2)$	$\gamma_V(e_1) \cup \gamma_V(e_2)$
$e_1 - e_2$	$\gamma_B(e_1) - \gamma_B(e_2)$	$\gamma'_B(e_1) - \gamma'_B(e_2)$	$\gamma_V(e_1) - \gamma_V(e_2)$

The correctness of this translation can be shown straightforwardly.

4.5.3 Checking Configuration Trees

For every view and update operation we now show that we can formulate a sufficient and necessary condition for a tree T for being a configuration tree given A_{new}, A_{old}, A_{tree}, ν_{new}, and ν_{old}. In order to check this, we introduce 3 consistency checks for the tree:

- View Consistency: are the trees marked to be the result of a view transformation (T_V and T''_V) actually the result of computing the view from the corresponding base trees (T_B and T'_B)?
- Update Consistency: are the trees marked to be the result of a update transformation (T'_V and T'_B) actually the result of computing the updates tree from the corresponding input trees (T_V and T_B)?
- Structural Consistency: is r_T the root of T_V, T'_V and T''_V, is the root of T_B and T'_B the only child of r_T, and are all nodes marked as being in one of the five trees of the configuration?

View Consistency

Let $\tau_V = \mathcal{V}(e_1, \alpha_1)$. The subtree T_V is a result of computing the view τ_V from T_B and T''_V of computing the view τ_V from T'_B.

1. a_V is defined for v iff $v \in \gamma_B(e_1) \cup \{r_T\}$
2. a''_V is defined for v iff $v \in \gamma'_B(e_1) \cup \{r_T\}$

Update Consistency

The subtree T'_V is a result of performing the update τ_U on T_V and the subtree T'_B is a result of performing the update τ'_U on T_B. However, since τ'_U is essentially the same update as τ_U we will see that in the actual checks it suffices to only check for τ_U and whether the same nodes in T_B are updated in the same way as those in T_V. We have to perform a number of tests to decide whether a tree is a configuration tree for an update. Let $A = A_{\tau_U} \cup A_{\tau_V}$. We now consider the three possible update operations used in τ_U and τ'_U seperately.

Attribute Update Let τ_U be the attribute update $\mathcal{U}[e, a, s]$.

1. The old nodes that are not changed by the update have the old attribute values, i.e., for every old node v that is not in $\gamma_V(e)$ and which has attribute assignment $\alpha = \lambda_T(v)$, it holds that for every attribute $a' \in A$ that either $\nu_{old}(a')$ and $\nu_{new}(a')$ are not defined or $\alpha(\nu_{old}(a')) = \alpha(\nu_{new}(a'))$.
2. The old nodes that are changed have the new value for the updated attribute and the old value for the non-updated attributes, i.e., for every old node v that is in $\gamma_V(e)$ and which has attribute assignment $\alpha = \lambda_T(v)$, it holds that for every attribute $a' \in A$ that if $a' \neq a$ then either $\nu_{old}(a')$ and $\nu_{new}(a')$ are not defined or $\alpha(\nu_{old}(a')) = \alpha(\nu_{new}(a'))$, else $\alpha(\nu_{new}(a)) = s$.
3. There are no new nodes.
4. The root is not changed, i.e., $r_T \notin \gamma_V(e)$.

Deletion Let τ_U be the deletion $\mathcal{D}[e]$.

1. The old nodes that are in subtrees rooted at nodes selected by the deletion, are effectively deleted, i.e., a'_V and a'_B are not defined for the nodes in $\mathcal{Q}(\gamma_V(e)/\downarrow^*)[T]$.
2. The old nodes that are not a descendant of a node selected by the deletion, are not deleted, i.e., a'_B is defined only when a_B is defined and a'_V is defined only when a_V is defined. Moreover for these nodes v it holds that the old and the new attribute values are the same, i.e., for every $a \in A$ it holds that either $\nu_{old}(a)$ and $\nu_{new}(a)$ are not defined in $\lambda_T(v)$ or $(\lambda_T(v))(\nu_{old}(a)) = (\lambda_T(v))(\nu_{new}(a))$.
3. There are no new nodes.
4. The root is not deleted, i.e., $r_T \notin \gamma_V(e)$.

Insertion Let τ_U be the insertion $\mathcal{I}[e, \alpha, w]$.

1. The old nodes v are not changed, i.e., for every $a \in A$ it holds that $(\lambda_T(v))(\nu_{old}(a)) = (\lambda_T(v))(\nu_{new}(a))$.
2. The new nodes do not have old attribute values, i.e., for every new node v and attribute $a \in A$ it holds that $\nu_{old}(a)$ is not defined in the attribute assignment $\lambda_T(v)$.
3. The new nodes get α as attribute assignment, i.e., if v is a new node then for every $a \in A$ it holds that $(\lambda_T(v))(\nu_{new}(a)) = s \Leftrightarrow \alpha(a) = s$.
4. If $w = \blacktriangleleft$ or $w = \blacktriangleright$ then the new nodes are immediately before of after a target node, i.e., if v is a new node then the next or previous sibling is in $\gamma_V(e)$.
5. If $w = \blacktriangledown$ then the new nodes v are a child of a target node and there are no siblings of v that are also a new node.
6. The root is not the target node for the insertion, i.e., $r_T \notin \gamma_V(e)$.

Structural Consistency

The structural consistency can be checked as follows:

1. The root of T is the root of the view trees T_V, T'_V and T''_V, i.e., attributes a_V, a'_V, and a''_V have values greater than or equal to the treshold value s_t for the root of T, attributes a_B, a'_B have a value smaller than s_t and the old attribute assignment corresponds to α.
2. The root of T has only one child which is the root of the base trees T_B and T'_B, i.e., there is only one node with attribute a_{root} defined and that node has only one ancestor (the root r_T) and no siblings. Moreover attributes a_B, a'_B have a value greater than or equal to s_t and a_V, a'_V, a''_V have values smaller than the treshold value s_t.

3. Every node has to come from at least one tree in the configuration, i.e., every node has at least one of $a_B, a'_B, a_V, a'_V, a''_V$ with a value be greater than s_t.

4. The configuration is node-creation independent, i.e., nodes that are new in T'_B compared to T_B do not occur in T'_V and nodes that are new in T'_V do not occur in T'_B. Node-conservatism already follows from previous constraints.

4.5.4 Checking for Conflicts in Configuration Trees

As we will show in Subsection 4.5.5, the previous constraints can be translated to checking whether $\mathbb{P}$ path expressions return the same result. However, we cannot check for isomorphism between two arbitrary trees using path expressions in $\mathbb{P}$. Luckily, the check for weak isomorphism can be simplified, as we will show now. From Definition 4.13 it follows that we have to check for isomorphism between two trees T'_V and T''_V in an attribute-minimal node-creation independent configuration. Since node identifiers are shared, this simplifies the check and we only have to check the following:

1. The old nodes that are in T'_V are in T''_V and vice versa, i.e., a'_V is defined iff a''_V is defined. This checks whether $\mathbf{Old}_C^{V'} = \mathbf{Old}_C^{V''}$.

2. For every new node in T'_V there is a new node in T'_B which is also projected in T''_V. The newly created nodes in T'_B have another identity than those in T'_V. Hence we check whether the nodes that are new in T'_B and that are projected into T''_V correspond to the new nodes in T'_V by checking whether their target nodes, i.e., the nodes relative to which they are inserted, correspond. This holds iff $T_{V''} \cong T_V$.

4.5.5 Complexity of Deciding Well-Behavedness

In this subsection we show that deciding whether a tree is a configuration tree for a conflicting configuration of a view and update can be done by checking whether a path expression e is satisfiable and use the results from 4.5.3 and 4.5.4. As a consequence, it follows then from Lemma 4.8 that deciding well-behavedness can be polynomially reduced to deciding unsatisfiability for e.

Three disjoint sets of nodes are defined for configuration trees. We can select the nodes of the sets using path expressions as follows:

- $e_{root} = \Uparrow$
- $e_{new} = \Uparrow/\downarrow^+/(a_B < s_t \vee a_V < s_t)$
- $e_{old} = \Uparrow/\downarrow^+/(a_B \geq s_t \wedge a_V \geq s_t)$

These expressions will be used to show the following theorem.

Theorem 4.1. *Well-behavedness of the simple propagation update strategy for views and updates is decidable, but non-elementary.*

Proof. First we show that well-behavedness of the simple update strategy for views and updates can be reduced polynomially to satisfiability for XPath without data value comparisons. Let τ_V be a view and τ_U and update. We now show that we can define a conjunction of equations of the form $\bigwedge_{i=1,\ldots,k}(e_{2i-1} = e_{2i})$ such that for every tree T it holds that T is a configuration tree for a conflicting attribute-minimal and node-creation independent configuration iff $(\mathcal{Q}[e_1](T) = \mathcal{Q}[e_2](T)) \wedge \ldots \wedge (\mathcal{Q}[e_{2k-1}](T) = \mathcal{Q}[e_{2k}](T))$. From this then follows that σ_p is well-behaved the expression e_{unsat}, defined as $\bigcup_{i=1,\ldots,k}((e_{2i-1} - e_{2i}) \cup (e_{2i} - e_{2i-1}))$, is unsatisfiable or $\Uparrow - e_{unsat}/\Uparrow$ is satisfiable.

First we define the checks we need to do to check whether T is a configuration tree. We have three groups of checks, i.e., view consistency, update consistency and structural consistency:

View Consistency Assume $\tau_V = \mathcal{V}[e, \alpha]$. We have to check whether $\downarrow^*/(a_V \geq s_t) = (e_{root} \cup \gamma_B(e_1))$ and $\downarrow^*/(a_V \geq s_t) = (e_{root} \cup \gamma'_B(e_1))$.

Update Consistency Let $A = \{a_1, \ldots, a_k\}$. The checks that we have to perform depend on the kind of update operation:

- $\tau_U = \mathcal{U}[e, a, s]$: We can assume w.l.o.g. that $a = a_1$. First we check the old nodes that are not changed by the update with $(e_{old} - \gamma_V(e_2)) = ((e_{old} - \gamma_V(e_2))/(\nu_{old}(a_1) \doteq \nu_{new}(a_1))/\ldots/(\nu_{old}(a_k) \doteq \nu_{new}(a_k)))$. Then we check the old nodes that are changed by the updates with the expression $(e_{old} \cap \gamma_V(e_2)) = ((e_{old} \cap \gamma_V(e_2))/(\nu_{new}(a_1) \doteq s)/(\nu_{old}(a_2) \doteq \nu_{new}(a_2))/\ldots/(\nu_{old}(a_k) \doteq \nu_{new}(a_k)))$. Afterwards we check whether there are no new nodes, i.e., $e_{new} = \emptyset$ and finally we check the root node with expression $e_{root} \cap \gamma_V(e) = \emptyset$.
- $\tau_U = \mathcal{D}[e]$: First we check whether the old nodes that are to be deleted are effectively deleted, i.e., $\gamma_V(e)/\downarrow^* = \gamma_V(e)/\downarrow^*/(a'_V < s_t \wedge a'_B < s_t)$. Then we check whether the old nodes that are not to be deleted are still there and have the same attribute assignment, i.e., $(e_{old} - \gamma_V(e_2)/\downarrow^*) = (e_{old} - \gamma_V(e_2)/\downarrow^*)/a'_B/((a_V \geq s_t \wedge a'_V \geq s_t) \vee (a_V < s_t \wedge a'_V < s_t))/(\nu_{old}(a_1) \doteq \nu_{new}(a_1))/\ldots/(\nu_{old}(a_k) \doteq \nu_{new}(a_k))$. Afterwards we check whether there are no new nodes, i.e., $e_{new} = \emptyset$ and finally we check the root node with expression $e_{root} \cap \gamma_V(e) = \emptyset$.
- $\tau_U = \mathcal{I}[e, \alpha, w]$: We can assume w.l.o.g. that there is a $l \leq k$ such that $\mathrm{dom}(\alpha) = \{a_1, \ldots, a_l\}$. First we check whether the old nodes are not changed, i.e., $e_{old} = (e_{old}/a'_B/((a_V \geq s_t \wedge a'_V \geq s_t) \vee (a_V < s_t \wedge a'_V < s_t))/(\nu_{old}(a_1) \doteq \nu_{new}(a_1))/\ldots/(\nu_{old}(a_k) \doteq \nu_{new}(a_k)))$ is empty. Then we check whether the new nodes have the correct new attribute assignment, i.e., $e_{new} = e_{new}/(a'_B \geq s_t \vee a'_V \geq s_t)/(\nu_{new}(a_1) = \alpha(a_1))/\ldots/(\nu_{new}(a_l) = \alpha(a_l))$. Afterwards we check whether the newly inserted nodes are inserted at the correct position as follows, depending on the value of w:
  - if $w = \blacktriangleleft$ then $e_{new}/(\dot{\twoheadrightarrow} - \dot{\twoheadrightarrow}/\dot{\twoheadrightarrow}) = \gamma_V(e)$
  - if $w = \blacktriangleright$ then $e_{new}/(\dot{\twoheadleftarrow} - \dot{\twoheadleftarrow}/\dot{\twoheadleftarrow}) = \gamma_V(e)$
  - if $w = \blacktriangledown$ then $e_{new}/\uparrow = \gamma_V(e)$

Finally we check the root node with expression $e_{root} \cap \gamma_V(e) = \emptyset$.

Structural Consistency First, we check whether the tree contains at least two nodes, i.e., $\epsilon = \downarrow^+/\downarrow^+/\Uparrow$. Note that by definition configuration trees need to have at least two nodes, since trees are always rooted and hence both the view tree and the base tree have at least one node. Then we check whether the root node is the root of the view trees: $\Uparrow = a_B < s_t \wedge a'_B < s_t \wedge a_V \geq s_t \wedge a'_V \geq s_t \wedge a''_V \geq s_t)$. Then we check whether the root node has only one child with $\downarrow/\dot{\twoheadrightarrow} = \emptyset$ and whether that node is the root of the base trees with $\downarrow/(a_B \geq s_t \wedge a'_B \geq s_t \wedge a_V < s_t \wedge a'_V < s_t \wedge a''_V < s_t)$. We then check whether there are no newly created nodes that are in both T'_B and T'_V with $e_{new}/(a'_B \geq s_t \wedge a'_V \geq s_t) = \emptyset$. Finally we check whether every node comes from at least one tree of the configuration, i.e., $\downarrow^* = \downarrow^*(a_B \geq s_t \vee a'_B \geq s_t \vee a_V \geq s_t \vee a'_V \geq s_t \vee a''_V \geq s_t)$.

Finally, the check for conflicts can be done as follows:

- The old nodes that are in T'_V are in T''_V and vice versa, i.e., $e_{old}/(a'_V \geq s_t) = e_{old}/(a''_V \geq s_t)$
- The set of target nodes of the new nodes in T'_V corresponds to the set of target nodes of the new nodes in T''_V. This depends on the insertion position w:
  - if $w = \blacktriangleleft$ then $e_{new}/(a'_V \geq s_t)/(\dot{\twoheadrightarrow} - \dot{\twoheadrightarrow}/\dot{\twoheadrightarrow}) = e_{new}/(a''_V \geq s_t)/(\dot{\twoheadrightarrow} - \dot{\twoheadrightarrow}/\dot{\twoheadrightarrow})$
  - if $w = \blacktriangleright$ then $e_{new}/(a'_V \geq s_t)/(\dot{\twoheadleftarrow} - \dot{\twoheadleftarrow}/\dot{\twoheadleftarrow}) = e_{new}/(a''_V \geq s_t)/(\dot{\twoheadleftarrow} - \dot{\twoheadleftarrow}/\dot{\twoheadleftarrow})$
  - if $w = \blacktriangledown$ then $e_{new}/(a'_V \geq s_t)/\uparrow = e_{new}/(a''_V \geq s_t)/\uparrow$

To conclude the proof, we show that XPath containment can be reduced polynomially to checking well-behavedness of the simple propagation update strategy. Let e be a path expression, a an attribute, s an attribute value, and α an attribute assignment. Consider the view $\mathcal{V}[(\epsilon - \downarrow^*/(a = a)/\Uparrow)/\downarrow^*, \alpha]$ and the update $\mathcal{I}[e/\Uparrow/\downarrow, \{(a, s)\}, \blacktriangledown]$. If e returns a non-empty result then a node with an a attribute will be added, but according to the view definition T''_V will only contain one node, i.e., the root with α as attribute assignment, since $\mathcal{Q}[(\epsilon - \downarrow^*/(a = a)/\Uparrow)](T)$ is then empty. Hence, σ_p is well-behaved for τ_V and τ_U iff e is not satisfiable. Similarly, for attribute updates, we can update some nodes to get an a attribute and for deletions we can define a view which returns the entire tree but first checks for a certain satisfiable path expression e' and then delete the nodes in the view corresponding to e' iff e returns a non-empty result.

From both polynomial reductions and the result of [ten Cate and Lutz, 2007] now follows that the decision problem for well-behavedness of σ_p is non-elementary. □

4.6 Conclusion

In this chapter we examined the XML View-Update problem for XPath-based projection views and some basic update operations. We studied a simple propagation update strategy

and introduced the notion of well-behavedness, i.e., can we observe the difference between applying an update directly on the view document or applying the translated update on the base document and then recomputing the view. We showed that well-behavedness of the simple propagation update strategy for our views and updates is decidable. Many questions remain open however, such as the complexity of deciding well-behavedness for fragments of $\mathbb{P}$. Other possible directions for future work include generalizing the update strategy, investigating the implications of adding some schema information, and finding efficient and conservative approximations for checking well-behavedness.

CHAPTER 5

Discussion

To conclude this dissertation we discuss the results presented in the previous chapters, refer to some of our related research and give pointers to possible future work and how the results of this thesis can be used for that research.

5.1 LiXQuery as a Framework

We defined LiXQuery [Hidders et al., 2005b, Hidders et al., 2004] in Chapter 2 as a clean formalisation of XQuery-based queries and updates. This framework for theoretical analysis enabled us to study properties of the XQuery language. We claim that LiXQuery contains most typical features of XQuery 1.0 such as `for` expressions, path expressions, position information, node construction, recursive function definitions, typeswitches, some arithmetic and boolean operators, some built-in functions and document loading. Another querying feature that is in LiXQuery is the `transform` expression, which was introduced in the XQuery Update Facility to construct slightly modified copies of XML fragments. LiXQuery also includes features to make persistent changes to an XML store and introduce side-effects using a `snap` operation, similar to the one introduced in [Ghelli et al., 2006].

The semantics of the XQuery Update Facility differs in some details with the semantics of LiXQuery without the `snap` operation. For example, their semantics of the "`replace value of`" operation allows to change the contents of element nodes, which can be simulated in LiXQuery. An important difference is that the working draft does not allow to mix updating and non-updating expressions anymore, i.e., queries and updates are split. We propose that, as is demonstrated by the presented syntax and semantics of LiXQuery, it is straightforward to define the semantics of a language that does not have this restriction. Moreover, it can be shown that for all LiXQuery programs e there exists an equivalent program e' where all queries and updates are split, i.e., there are no subexpressions that

return both a non-empty result sequence and a non-empty list of pending updates.

5.2 Studying the Expressive Power of LiXQuery

In Chapter 3 we compared of expressive power of LiXQuery fragments. In this section we first look at how we can extend the technique of program simulation presented there to "separate" queries and updates in LiXQuery expressions. In the second part of this section we discuss some other measures of expressive power.

5.2.1 Separating Queries and Updates

We informally sketch how to extend the program simulation presented in the proof of Lemma 3.17 to illustrate how we can separate updating and non-updating subexpressions in LiXQuery. The general idea is that we want to transform an expression e to an expression where no subexpression returns both a non-empty result sequence and a non-empty list of pending updates. This is done as follows:

```
let $result := (ϵ(e))
let $updates := performUpd($result)
return retSequence($result)
```

First we compute the result of the expression $\epsilon(e)$, which does not contain any updating subexpression and hence returns an empty list of pending updates. The result of the simulating expression is bound to the variable `$result` and contains the result sequence which encodes the result store, the result sequence and the list of pending updates generated by e. Then we extract and perform the encoded list of pending updates, and bind the resulting empty sequence to the variable `$updates`. Finally, we can extract the result sequence from the encoded result sequence and return this as result of the program.

First, note that now the node identity of the nodes in the result sequence is important, since we update existing parts of the store and want to claim these get updated in the same way as the original expression does. A simulated computation context similar to the one used in Chapter 3 can be used, but since we allow several update statements and cannot perform the updates immediately, we extend the relation ρ to contain the mapping of all relevant node identifiers to their corresponding nodes. We then first compute the simulated result and afterwards perform the list of pending updates as needed. Note that when `snap` operations occur, we need to keep track of which non-deterministic choices are made in order to return the correct result when the updates are actually performed, which is after the query result is computed. To illustrate this consider the following simple example:

```
let $x := (<root> <a/> <b/> </root>) return
let $a := ($x/a) return
let $b := ($x/b) return
let $upd := snap { delete $x/* } return
let $c := (
  if ($a << $b)
```

```
    then (element {"c"} {()})
    else (element {"d"} {()}))
  return ($a,$b,$c)
```

Note that the deletion removes the incoming edge in a node and assigns a new position in document order to the deleted node in a non-deterministic fashion. When we apply this example to an empty store then the result store will contain 4 nodes, of which 3 are in the result sequence. However, choosing one of the possible result stores for the deletion before performing the actual deletion might result into a result sequence that cannot correspond to the result store, e.g., `<a/>, <b/>, <c/>` is the result sequence when in the result store the `a` node is in document order after the `b` node.

Finally, it can also be shown that if no `snap` operation or recursive function definitions are used, the simulation of the `for` expression can be done without using recursive function definitions. This is possible since we maintain the transitive closure of the parent-child relation in the store at all time. Moreover, in this case each iteration sees the same part of the input store and the newly added fragments to the output store can be merged at the end, where an offset to the identifiers has to be applied to ensure these are still unique in the simulated output store.

5.2.2 Other Measures of Expressive Power

The relative expressive power of LiXQuery constructs can be measured in several ways. In Chapter 3 we looked at the transformations that can be done, i.e., mappings from an input store and an variable assignment to a serialized result. We have chosen to include variable assignments since this enables us to prove properties for some LiXQuery functions in a more general way. It is however an open problem whether the obtained expressiveness results also hold when we assume that programs only can have empty variable assignments as input, since the separatability results are shown using properties of the variable assignment. For example, consider the query to return the node that is in the middle of the document w.r.t. document order or the k^{th} smallest value of a document. Both queries seem to need the `count` function to be expressible, however they are both expressible in XQ^R, while Lemma 3.1 showed that in general we cannot count the number of items in a sequence bound to a variable. The first query can be written as applying a user-defined function `trimBorders()`, which returns its parameter sequence if it is a singleton sequence or otherwise removes th first and the last node in document order of the sequence and then applies the function recursively on this smaller sequence. The second query can be simulated in XQ^R as follows by apply a function that returns a sequence that contains for every value the first node in the document with that value and for each node in this sequence count how many other nodes have a value that is smaller. We can count unique nodes in XQ^R and if we count $k-1$ nodes then we return the value of this node. These two examples illustrate that removing the environment from the input of LiXQuery programs might change the expressiveness relations between the fragments introduced in Chapter 3, such as being now able to count in XQ^R, however whether this is actually the case is

still an open question, since we might need to count the different number of values in a computed sequence and we cannot use node construction to create new nodes in XQ^R and count these.

Several other relations between input and output can be interesting as a way to measure the relative expressive power:

Document to Document Mappings We consider mappings from forests to forests, however often only tree transformations are needed, i.e., one tree is transformed into another tree, as for example is done in XSL Transformations [Clark, 1999].

Update Operations In the XQuery Formal Semantics [Draper et al., 2006], there is an option for XQuery to work on data loaded externally in another data model, e.g., DOM. In this case, node identity can be important since it can contain information from beyond the scope of the XQuery processor. In [Hidders et al., 2006] we studied the expressive power of LiXQuery fragments in terms of node-identity sensitive updates, i.e., we investigated which fragments can express the same set of mappings from stores to stores, up to garbage collection.

Query Operations XQuery can be used to perform mappings from XML stores and environments to sequences only containing atomic values. This can be seen as requesting "flat" information from an XML document. In [Hidders et al., 2006] we briefly study the expressiveness relations in terms of such mappings to atomic values.

Node-Conservative Transformations These transformations allow node creation, however newly created nodes are disallowed in the result and the node identity for the result is important. These transformations are studied in [Page et al., 2005] for $XQ^{constr}_{at,C,R}$ and the results from this work can be generalized in such a way that also for node-conservative $XQ^{constr}_{at,C}$ expressions we can find an equivalent $XQ_{at,C}$ expression.

Moreover, also other features than the one we studied in Chapter 3 can be studied into more details. In [Hidders et al., 2005a, Hidders et al., 2007a] we also studied the relative expressive power of the sequence generator `to` and the `sum` aggregation function. Also the relative expressive power of the `transform` is an interesting topic to study. For example, it is not clear whether `transform` adds expressive power in terms of transformations to XQ^{upd} and whether XQ^{constr} extended with the `transform` expression would be equally expressive as XQ^{upd}.

Instead of comparing the relative expressive power of XQuery fragments, one can also study the expressive power of these fragments by comparing them to several types of logic. We briefly discuss some related work in this direction by Koch and Benedikt [Koch, 2006, Benedikt and Koch, 2006]. In this work also XQuery fragments are defined and studied in terms of computational complexity and compared in expressive power with certain types of first-order logic. Unfortunately the fragments defined in that work have no direct relationship with our fragments. However, we can make some observations on

the relationships between their fragments $AtomXQ$ and XQ which seem similar to our fragments XQ^{constr} and XQ^{constr}_{at}, respectively.

Their fragment $AtomXQ$ is an XQuery fragment in which one can express path expressions, create new trees, compare nodes/values, test sequences for emptiness and use simple `for` expressions and `if` expressions. In terms of expressive power $AtomXQ$ is a subset of our fragment XQ^{constr}. The converse clearly does not hold since XQ^{constr} can do basic arithmetic on values in the XML trees. Even if the arithmetic operations from XQ^{constr} are removed the relationship is not clear because XQ^{constr} allows general `let` expressions for which it is not clear if the can be removed without losing expressive power. Their fragment XQ is basically $AtomXQ$ extended with a deep-equality comparison for trees. In terms of expressive power their fragment XQ (which is different from our XQ) is a subset of our fragment XQ^{constr}_{at}. The most involved part of the proof is showing that deep equivalence of nodes can be expressed, for which the `at` clause seems to be required. Conversely, it is easily observed that XQ^{constr}_{at} can express some functions that cannot be expressed by their XQ because with the `at` clause we can write functions that return integers not in the XML tree. This raises the question whether in terms of expressive power their XQ is a subset of our XQ^{constr}, which seems unlikely, but no proof of a counterexample has been found yet.

5.3 Updating XML Views

The problem of deciding well-behavedness for the simple propagation update strategy introduced in Chapter 4 is a specific subproblem of the XML view-update problem. This work is an important first step into avoiding view side-effects in an instance-independent manner, however the definition off well-behavedness can be a bit too strict in the sense that we require every result that we can obtain through updating the view should also be obtainable by first updating the base instance and then compute the view, while in practice it can suffice that every view obtained by updating the base instance can also be the result of performing the update on the view tree directly. Moreover, many extensions and optimizations can be made and several update statements, such as copying subtrees, are not considered. In this section we discuss two ways of generalizing and extending the results. First, we take a look at restricting the update operations to disallow the use of set difference in its path expressions and then we see how we can consider update strategies where single update statements are translated into a list of update operations on the base tree.

5.3.1 Complexity for Deciding in a Positive Fragment of $\mathbb{P}$

Let $\mathbb{P}^+$ be the fragment of $\mathbb{P}$ without set difference, i.e., the fragment containing set union, intersection, path concatenation, predicates and recursive navigational axes. Note that we cannot express the child, parent and sibling axes in this fragment. This fragment has several interesting properties. For example, if we apply a query e on two trees T_1 and T_2

such that T_1 is a subtree of T_2 then the result of $\mathcal{Q}[e](T_1)$ is a subset of $\mathcal{Q}[e](T_2)$ and the semantics of the axes e_{axis} can be shown to have the following relationship: $\mathcal{P}[e_{axis}](T_1) = \mathcal{P}[e_{axis}](T_2) \cap (V_{T_1} \times V_{T_1})$. Moreover, this fragment is also monotone in the following way.

Lemma 5.1. *Let e be a path expression in $\mathbb{P}^+$, T_1 a document tree. Then for every extension T_2 of T_1 w.r.t. the set of attribute names A_e it holds that $\mathcal{Q}[e](T_1) \subseteq \mathcal{Q}[e](T_2)$.*

Proof. We prove this lemma by induction on $|e|$. Since we retain all attributes that are referred to in predicates and since all axes in our language are transitive, it can be easily seen that if e is an axis or a predicate then $\mathcal{P}[e](T_2) \subseteq \mathcal{P}[e](T_1)$. Moreover, if $\mathcal{P}[e_1](T_2) \subseteq \mathcal{P}[e_1](T_1)$ and $\mathcal{P}[e_2](T_2) \subseteq \mathcal{P}[e_2](T_1)$, then for path expressions e of the form e_1/e_2, $e_1 \cap e_2$ or $e_1 \cup e_2$ it clearly holds that $\mathcal{P}[e](T_2) \subseteq \mathcal{P}[e](T_1)$. □

We can extend satisfiability of attribute assignments from Lemma 4.1 to satisfiability of path expressions in $\mathbb{P}^+$. First, we show a finite model property for the positive fragment $\mathbb{P}^+$.

Lemma 5.2. *Let e be a path expression in $\mathbb{P}^+$. If $\mathcal{Q}[e]$ is satisfiable then there is a tree T with at most $|e| + 1$ nodes such that $\mathcal{Q}[e](T) \neq \emptyset$.*

Proof. Assume there is a tree T such that $\mathcal{Q}[e](T)$ is not empty. A candidate model for a path expression e is a tuple $C = (v_1, v_2, V_e)$ with $V_e \subseteq V_T$ and $v_1, v_2 \in V_e$ for which it holds that if $T' = \pi_{V_e \cup \{v_r\}}(T)$ then (v_1, v_2) has to be in both $\mathcal{P}[e](T)$ and $\mathcal{P}[e](T')$. The size of a candidate model is defined as the number of nodes in V_e. Let M_e denote the set of all minimal candidate models for e, i.e., $(v_1, v_2, V_e) \in M_e$ implies there is no $V'_e \subsetneq V_e$ such that (v_1, v_2, V'_e) is a candidate model for e.

We now show by induction on the size of e that we can compute M_e and that it is not empty iff $\mathcal{P}[e](T)$ is not empty. Moreover, from the construction it follows that the size of all candidate models in M_e is at most $|e| + 1$. If e is an axis or a predicate then clearly $M_e = \{(v_1, v_2, \{v_1, v_2\}) | (v_1, v_2) \in \mathcal{P}[e](T)\}$. Else it can be verified that M_e is obtained by minimizing the following M'_e sets of candidate models[1]:

$$M'_{e_1/e_2} = \{(v_1, v_2, V_e) | \exists V_{e_1}, V_{e_2} : \exists v_3 : (v_1, v_3, V_{e_1}) \in M_{e_1} \wedge (v_3, v_2, V_{e_2}) \in M_{e_2} \wedge V_e = V_{e_1} \cup V_{e_2}\}$$
$$M'_{e_1 \cap e_2} = \{(v_1, v_2, V_e) | \exists V_{e_1}, V_{e_2} : (v_1, v_2, V_{e_1}) \in M_{e_1} \wedge (v_1, v_2, V_{e_2}) \in M_{e_2} \wedge V_e = V_{e_1} \cup V_{e_2}\}$$
$$M'_{e_1 \cup e_2} = M_{e_1} \cup M_{e_2}$$

Since M_e is not empty iff $\mathcal{P}[e](T)$ is not empty we can conclude that $\mathcal{Q}[e](T)$ is not empty iff there is a candidate model $C \in M_e$ such that $C = (v_r, n, V_e)$ and $v_r \in V_e$ and hence the size of $\pi_{V_e \cup \{v_r\}}(T)$ is $|V_e|$, which is at most $|e| + 1$. □

We now use the previous Lemma to establish the complexity of deciding containment for $\mathbb{P}^+$.

[1] It is easy to see that for every set of candidate models there exists exactly one set of all minimal candidate models

Property 5.1. *Containment of path expressions in $\mathbb{P}^+$ is a CoNP-complete decision problem.*

Proof. Suppose $\mathcal{Q}[e_1] \not\subseteq \mathcal{Q}[e_2]$ then there is a tree T such that there is a node $v \in \mathcal{Q}[e_1](T)$ and $v \notin \mathcal{Q}[e_2](T)$ and from Lemma 4.1 it follows that we can assume w.l.o.g. that the size needed to store the attribute assignments is bounded for every node by some polynomial in terms of $|e_1|+|e_2|$. From the proof of Lemma 5.2 we know that there is a set of nodes V_e such that restricting T to V_e yields a tree T_e with at most $|e_1|$ nodes and from Lemma 5.1 we know that if $v \notin \mathcal{Q}[e_2](T)$ then $v \notin \mathcal{Q}[e_2](T_e)$. Hence T_e is a counter example with at most $|e_1| + 1$ nodes. From this, it follows that if there is a counter example, then there is also a counter example with at most $|e_1| + 1$ nodes. We can guess a tree T, attribute assignments for the nodes and evaluate both $\mathcal{Q}[e_1]$ and $\mathcal{Q}[e_2]$ in polynomial time (polynomial in $|e_1|+|e_2|$) and iff for every possible guess T it holds that $\mathcal{Q}[e_1](T) \subseteq \mathcal{Q}[e_2](T)$ then $\mathcal{Q}[e_1]$ is contained in $\mathcal{Q}[e_2]$. Hence we conclude that deciding containment for path expressions in $\mathbb{P}^+$ is in CoNP. CoNP-hardness follows from [Neven and Schwentick, 2006] (Theorem 3.2) and hence the containment problem is a CoNP-complete problem. □

The previous result could be used for getting an idea on the complexity of deciding well-behavedness of the simple propagation update strategy when the views and updates only are allowed to have path expressions in $\mathbb{P}^+$. However, the path expression that we have to check for satisfiability, as is given in Subsection 4.5.5, is of the form $\bigcup_{i=1,\ldots,k}((e_{2i-1} - e_{2i}) \cup (e_{2i} - e_{2i-1}))$ where it can be shown that every e_j is a $\mathbb{P}^+$ expression. The complexity of satisfiability for these expressions is unknown, which is the reason why this section is in the discussion and not in Chapter 4. We conjecture that containment is not likely to be non-elementary hard and hence establishing the exact complexity for well-behavedness for $\mathbb{P}^+$ is an interesting research direction.

5.3.2 Dealing with a Larger Class of Update Strategies

Limiting the update strategy to performing the same primitive update operations on the same nodes in the base tree can be too strict. We now briefly discuss a possible extension of the Simple Propagation Update Strategy to allow sequences of update operations and translated updates that do not perform the same primitive update operations on the same nodes. We illustrate how the framework used in the proof of decidability in Chapter 4 could be extended by means of an example.

Consider the XML tree of Figure 5.1 and a view showing only the active members. When deleting a member from this view it might be desirable to mark them as not active instead of deleting them from the base instance. In SQL we can use `INSTEAD OF` triggers to specify what needs to be done on the base instance, which can be a list of operations. Suppose in this case we want the update strategy to translate the deletion of members τ_U to the update operation sequence $\langle \tau'_U, \tau''_U \rangle$, where τ'_U replaces the value of active with "`no`" and τ''_U removes the activities. In general, if $\sigma(\tau_V, \tau_U) = \langle \tau_1, \ldots, \tau_k \rangle$ then we generalize the notion of configuration to be a $(k+4)$-tuple of trees $(T_1, \ldots, T_{k+4})$ where T_1 is the base instance, T_2 the view tree on T_1, T_3 the updated view tree, T_4 the result of applying τ_1 on

```
<memberList>
 <member>
   <name>S. Wells</name>
   <address>Farm Road 8</address>
   <active> yes </active>
   <activities> Administration </activities>
  </member>
 <member>
   <name>F. Fredericks</name>
   <address>Town Square 6B</address>
   <active> no </active>
  </member>
 <member>
   <name>B. Reach</name>
   <address>Central Street 190</address>
   <active> yes </active>
   <activities> Security </activities>
  </member>
</memberList>
```

Figure 5.1: XML Document containing Members of an Organization

T_1, T_{k+4} the view tree on T_{k+3} and for every $i = 5, \ldots, k-1$ it holds that $\tau_{i-4}(T_{i-1}, T_i)$. In a similar way to what was done in Chapter 4 we can now define configuration trees. Note that when checking whether a tree is a valid configuration tree we now need to check whether the updates τ_i are performed. Moreover, the check for conflicts is more complex in this case as well because the isomorphism check is harder now.

5.4 Updating XML Views with LiXQuery

The querying and updating expressions that we studied in Chapter 4 were restricted to XPath-based operations. However, in practice we often use XQuery for querying and updating XML databases. A possible further investigation is how we can detect LiXQuery expressions which correspond to our update and view operations. A first step in this direction is the work presented in [Hidders et al., 2007b] where we show how to detect XPath expressions in an XQuery context.

APPENDIX A

Dutch Summary

Het XML formaat wordt momenteel vaak gebruikt om informatie te delen, data uit te wisselen en op het Web te publiceren. In essentie zijn XML documenten geordende onbegrensde bomen. Het ondervragen en beheren van de massa semi-gestructureerde gegevens is een uitdaging die voorgenoemde evolutie met zich meebrengt. In dit proefschrift behandelen we daarom enkele deelproblemen op dit vlak. We zullen nu kort overlopen welke problemen er precies in deze thesis behandeld worden.

Vooreerst dienen XML document ondervraagd te kunnen worden. Hiervoor heeft het World Wide Web Consortium (W3C) de querytaal XQuery ontwikkeld. Deze querytaal is recentelijk uitgebreid met updatefaciliteiten om het eenvoudiger te maken om XML documenten te updaten zonder daarbij bijvoorbeeld zelf handmatig het DOM-model van het document te moeten aanpassen. In Hoofdstuk 2 introduceren we LiXQuery als formalisme om eigenschappen van deze taal te bestuderen. LiXQuery heeft een beknopte syntaxis en semantiek die volledig formeel gedefinieerd is. Naast de traditionele XML querytaalonderdelen, zoals iteratie met een `for`-lus, padexpressies, eenvoudige arithmetiek, etc., bevat deze taal ook transformatie-expressies om lichtjes gewijzigde kopiën van XML fragmenten te construeren, update-operaties om bestaande XML documenten te wijzigen en een `snap` operatie, geïntroduceerd in [Ghelli et al., 2006], om neveneffecten toe te laten.

De eigenschappen van verschillende belangrijke operaties in LiXQuery worden bestudeerd in Hoofdstuk 3. Meer bepaald bestuderen we de relatieve expressieve kracht in termen van transformaties van het Web en een variabeletoekenning naar een geserialiseerd resultaat. Concreet bestuderen we de volgende operaties: de aggregatiefunctie `count()`, de positieinformatie door middel van een `at`-clausule in een `for`-lus, `transform`-expressies, update-expressies, de `snap`-operatie en het gebruik van recursieve functiedefinities. In totaal bekomen we zo 32 LiXQuery-fragmenten die we kunnen onderverdelen in twaalf verschillende equivalentieklassen. Als twee fragmenten zich in dezelfde equivalentieklasse bevinden dan kunnen beide deeltalen ook precies dezelfde transformaties uitdrukken. In-

dien twee fragmenten niet in eenzelfde equivalentieklasse terug te vinden zijn, dan zijn er ook effectief transformaties die één fragment kan uitdrukken en die niet in het andere fragment kunnen uitgedrukt worden.

Omdat XML vaak gebruikt wordt als gegevensuitwisselingsformaat tussen verschillende partijen, spreekt het voor zich dat mensen de transformaties van hun eigen gegevensbank naar het overeengekomen formaat waarin men de gegevens deelt, wensen te automatiseren en zelfs zodanig dat er aan de andere partijen een virtueel XML document wordt aangeboden dat in feite een "view" is op de eigen gegevensbank. Zulk een view kan dan ook als toegangscontrolemechanisme gebruikt worden, dat wil zeggen, alle gegevens die ik kan zien, mag ik zelf ook updaten. Een view dient zich echter zo veel als mogelijk te gedragen als een gewoon document. Het updaten van views kan echter voor allerlei problemen zorgen. Ten eerste is het niet altijd duidelijk welke update op het achterliggende document overeenkomt met de gewenste update op het virtuele document. Ten tweede kunnen allerlei neveneffecten ontstaan door het updaten van het achterliggende document.

De views die we beschouwen in Hoofdstuk 4 zijn zogenaamde projectieviews die het basisdocument projecteert op de knopen die geselecteerd worden door een padexpressie en er een nieuwe wortelknoop boven plaatst met een vooraf bepaalde attribuutstoekenning. We tonen aan dat query's over virtuele documenten bepaald door zo'n view vertaald kunnen worden naar query's over de basisdocumenten zodanig dat een knoop in het resultaat van de vertaalde query zit als en slechts als de projectie van deze knoop in het resultaat zit van de originele query.

De updates zijn invoegingen van enkele knopen ten opzichte van andere knopen geselecteerd met een padexpressie, het verwijderen van deelbomen en het aanpassen van de attribuutstoekenning van knopen geselecteerd door een padexpressie. We vertalen updates van geprojecteerde knopen in de view naar dezelfde updates van de overeenkomstige knopen in het basisdocument. Deze strategie noemen we de updatestrategie door middel van eenvoudige propagatie. Het probleem dat we concreet beschouwen is het volgende probleem: gegeven een view en een update, kunnen we dezelfde bekomen door de view te updaten als door eerst het basisdocument te updaten met de door onze strategie bepaalde update en dan de view opnieuw te berekenen. We tonen aan dat dit probleem beslisbaar is, maar niet-elementair. De gevolgde bewijstechniek hiervoor is het reduceren van het beslissingsprobleem naar het beslissen of een padexpressie een niet-leeg resultaat kan opleveren. Als tussenstap in dit bewijs maken we gebruik van zogenaamde configuraties, 5-tupels van bomen die de basisbomen, de view-bomen en de resultaten van de updates bevatten. Deze configuraties kunnen worden weergegeven in één boom, genaamd de configuratieboom. Het bestaan van een bepaald soort configuratieboom wordt dan gebruikt om aan te tonen dat er effectief neveneffecten op de views kunnen zijn en dit kan getest worden door de bevredigbaarheid van een padexpressie te beslissen.

Tot slot worden in Hoofdstuk 5 enkele mogelijke paden geschetst voor verder onderzoek en tonen we aan hoe we de resultaten van dit proefschrift kunnen verder uitbreiden en veralgemenen.

Publications by the Author

[Fernández et al., 2005] Fernández, M. F., Hidders, J., Michiels, P., Siméon, J., and Vercammen, R. (2005). Optimizing sorting and duplicate elimination in XQuery path expressions. In Andersen, K. V., Debenham, J. K., and Wagner, R., editors, *DEXA*, volume 3588 of *Lecture Notes in Computer Science*, pages 554–563. Springer.

[Hidders et al., 2005a] Hidders, J., Marrara, S., Paredaens, J., and Vercammen, R. (2005a). On the expressive power of XQuery fragments. In Bierman, G. M. and Koch, C., editors, *DBPL*, volume 3774 of *Lecture Notes in Computer Science*, pages 154–168. Springer.

[Hidders et al., 2007a] Hidders, J., Marrara, S., Paredaens, J., and Vercammen, R. (2007a). On the expressivity of functions in XQuery fragments. *Information Systems*. Accepted for Publication.

[Hidders et al., 2005b] Hidders, J., Michiels, P., Paredaens, J., and Vercammen, R. (2005b). LiXQuery: a formal foundation for XQuery research. *SIGMOD Record*, 34(4):21–26.

[Hidders et al., 2007b] Hidders, J., Michiels, P., Siméon, J., and Vercammen, R. (2007b). How to recognise different kinds of tree patterns from quite a long way away. In *PLAN-X*, pages 14–24.

[Hidders et al., 2005c] Hidders, J., Michiels, P., and Vercammen, R. (2005c). Optimizing sorting and duplicate elimination in XQuery path expressions. *Bulletin of the EATCS*, 86:199–223.

[Hidders et al., 2006] Hidders, J., Paredaens, J., and Vercammen, R. (2006). On the expressive power of XQuery-based update languages. In Amer-Yahia, S., Bellahsene, Z., Hunt, E., Unland, R., and Yu, J. X., editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 92–106. Springer.

[Hidders et al., 2004] Hidders, J., Paredaens, J., Vercammen, R., and Demeyer, S. (2004). A light but formal introduction to XQuery. In *XSym*, pages 5–20.

[Page et al., 2005] Page, W. L., Hidders, J., Michiels, P., Paredaens, J., and Vercammen, R. (2005). On the expressive power of node construction in XQuery. In Doan, A., Neven, F., McCann, R., and Bex, G. J., editors, *WebDB*, pages 85–90.

[Vercammen, 2005] Vercammen, R. (2005). Updating XML views. In *VLDB PhD Workshop 2005*, pages 6–10.

[Vercammen et al., 2006] Vercammen, R., Hidders, J., and Paredaens, J. (2006). Query translation for xpath-based security views. In Grust, T., Höpfner, H., Illarramendi, A., Jablonski, S., Mesiti, M., Müller, S., Patranjan, P.-L., Sattler, K.-U., Spiliopoulou, M., and Wijsen, J., editors, *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 250–263. Springer.

Bibliography

[Abiteboul, 1999] Abiteboul, S. (1999). On views and XML. In *PODS*, pages 1–9.

[Abiteboul et al., 1999] Abiteboul, S., Buneman, P., and Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann.

[Abiteboul et al., 1997] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. (1997). The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88.

[Bancilhon and Spyratos, 1981] Bancilhon, F. and Spyratos, N. (1981). Update semantics of relational views. *ACM TODS*, 6(4):557–575.

[Benedikt et al., 2005a] Benedikt, M., Bonifati, A., Flesca, S., and Vyas, A. (2005a). Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P*.

[Benedikt et al., 2005b] Benedikt, M., Bonifati, A., Flesca, S., and Vyas, A. (2005b). Verification of tree updates for optimization. In Etessami, K. and Rajamani, S. K., editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 379–393. Springer.

[Benedikt and Fundulaki, 2005] Benedikt, M. and Fundulaki, I. (2005). XML subtree queries: Specification and composition. pages 138–153.

[Benedikt and Koch, 2006] Benedikt, M. and Koch, C. (2006). Interpreting tree-to-tree queries. In Bugliesi, M., Preneel, B., Sassone, V., and Wegener, I., editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 552–564. Springer.

[Berglund et al., 2007] Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J. (2007). XML path language (XPath) 2.0. `http://www.w3.org/TR/xpath20`.

[Bertino and Ferrari, 2002] Bertino, E. and Ferrari, E. (2002). Secure and selective dissemination of xml documents. *ACM Trans. Inf. Syst. Secur.*, 5(3):290–331.

[Boag et al., 2007a] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. (2007a). XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery`.

[Boag et al., 2007b] Boag, S., Kay, M., Tong, J., Walsh, N., and Zongaro, H. (2007b). XSLT 2.0 and XQuery 1.0 serialization. `http://www.w3.org/TR/xslt-xquery-serialization`.

[Bohannon et al., 2006] Bohannon, A., Pierce, B. C., and Vaughan, J. A. (2006). Relational lenses: a language for updatable views. In Vansummeren, S., editor, *PODS*, pages 338–347. ACM.

[Böttcher and Steinmetz, 2005] Böttcher, S. and Steinmetz, R. (2005). Adaptive xml access control based on query nesting, modification and simplification. In Vossen, G., Leymann, F., Lockemann, P. C., and Stucky, W., editors, *BTW*, volume 65 of *LNI*, pages 295–304. GI.

[Box et al., 2000] Box, D., Skonnard, A., and Lam, J. (2000). *Essential XML: Beyond Markup*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Braganholo et al., 2004] Braganholo, V. P., Davidson, S. B., and Heuser, C. A. (2004). From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287.

[Braganholo et al., 2006] Braganholo, V. P., Davidson, S. B., and Heuser, C. A. (2006). Pataxó: A framework to allow updates through xml views. *ACM Trans. Database Syst.*, 31(3):839–886.

[Brundage, 2004] Brundage, M. (2004). *XQuery: The XML Query Language*. Pearson Higher Education.

[Buff, 1988] Buff, H. W. (1988). Why codd's rule no. 6 must be reformulated. *SIGMOD Record*, 17(4):79–80.

[Buneman, 1997] Buneman, P. (1997). Semistructured data. In *PODS*, pages 117–121. ACM Press.

[Buneman et al., 2000] Buneman, P., Fernandez, M. F., and Suciu, D. (2000). UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110.

[Buneman et al., 2002] Buneman, P., Khanna, S., and Tan, W. C. (2002). On propagation of deletions and annotations through views. In Popa, L., editor, *PODS*, pages 150–158. ACM.

[Chamberlin et al., 2007] Chamberlin, D., Florescu, D., Melton, J., Robie, J., and Siméon, J. (2007). XQuery update facility 1.0. W3C Working Draft. `http://www.w3.org/TR/xquery-update-10/`.

[Chamberlin et al., 2006] Chamberlin, D., Florescu, D., and Robie, J. (2006). XQuery update facility. W3C Working Draft. `http://www.w3.org/TR/xqupdate/`.

[Chamberlin and Robie, 2005] Chamberlin, D. and Robie, J. (2005). XQuery update facility requirements. W3C Working Draft. `http://www.w3.org/TR/2005/WD-xquery-update-requirements-20050211/`.

[Chamberlin et al., 2000] Chamberlin, D. D., Robie, J., and Florescu, D. (2000). Quilt: An xml query language for heterogeneous data sources. In Suciu, D. and Vossen, G., editors, *WebDB (Selected Papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer.

[Chandra and Harel, 1980] Chandra, A. K. and Harel, D. (1980). Computable queries for relational data bases. *J. Comput. Syst. Sci.*, 21(2):156–178.

[Clark, 1999] Clark, J. (1999). XSL transformations. `http://www.w3.org/TR/xslt`.

[Clark and DeRose, 1999] Clark, J. and DeRose, S. (1999). XML path language (XPath) 1.0. `http://www.w3.org/TR/xpath`.

[Cluet and Siméon, 2000] Cluet, S. and Siméon, J. (2000). YaTL: a functional and declarative language for XML.

[Codd, 1985] Codd, E. (1985). Is your DBMS really relational? In *Computerworld*.

[Codd, 1974] Codd, E. F. (1974). Recent investigations in relational data base systems. In *IFIP Congress*, pages 1017–1021.

[Cosmadakis and Papadimitriou, 1984] Cosmadakis, S. S. and Papadimitriou, C. H. (1984). Updates of relational views. *J. ACM*, 31(4):742–760.

[Dayal and Bernstein, 1978] Dayal, U. and Bernstein, P. A. (1978). On the updatability of relational views. In *VLDB*, pages 368–377.

[Deutsch et al., 1998] Deutsch, A., Fernández, M. F., Florescu, D., Levy, A. Y., and Suciu, D. (1998). Xml-ql. In *QL*.

[Draper et al., 2006] Draper, D., Frankhauser, P., Fernández, M. F., Malhorta, A., Rose, K., Rys, M., Siméon, J., and Wadler, P. (2006). XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft. `http://www.w3.org/TR/xquery-semantics`.

[Fan et al., 2004] Fan, W., Chan, C. Y., and Garofalakis, M. N. (2004). Secure XML querying with security views. In *SIGMOD Conference*, pages 587–598.

[Fan et al., 2007a] Fan, W., Cong, G., and Bohannon, P. (2007a). Querying xml with update syntax. In Chan, C. Y., Ooi, B. C., and Zhou, A., editors, *SIGMOD Conference*, pages 293–304. ACM.

[Fan et al., 2007b] Fan, W., Geerts, F., Jia, X., and Kementsietsidis, A. (2007b). Rewriting regular xpath queries on XML views. In *ICDE*, pages 666–675. IEEE.

[Fankhauser et al., 2000] Fankhauser, P., Marchiori, M., and Robie, J. (2000). XML Query requirements. W3C Working Draft. `http://www.w3.org/TR/2000/WD-xmlquery-req-20000131`.

[Fernández et al., 2005] Fernández, M., Malhotra, A., Marsh, J., Nagy, M., and Walsh, N. (2005). XQuery 1.0 and XPath 2.0 data model (XDM). `http://www.w3.org/TR/xpath-datamodel/`.

[Fernández et al., 2007] Fernández, M. F., Malhorta, A., Marsh, J., Nagy, M., and Walsh, N. (2007). XQuery 1.0 and XPath 2.0 data model (XDM). `http://www.w3.org/TR/xpath-datamodel`.

[Foster et al., 2005] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2005). Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, pages 233–246.

[Fundulaki and Marx, 2004] Fundulaki, I. and Marx, M. (2004). Specifying access control policies for XML documents with XPath. In *SACMAT 2004*, pages 61–69.

[Ghelli et al., 2007] Ghelli, G., Onose, N., Rose, K., and Siméon, J. (2007). A better semantics for XQuery with side-effects. In *DBPL*.

[Ghelli et al., 2006] Ghelli, G., Ré, C., and Siméon, J. (2006). XQuery!: An XML query language with side effects. In *DataX 2006*, Munich, Germany.

[Gottlob et al., 1988] Gottlob, G., Paolini, P., and Zicari, R. (1988). Properties and update semantics of consistent views. *ACM TODS*, 13(4):486–524.

[Guo et al., 1996] Guo, S., Sun, W., and Weiss, M. A. (1996). Solving satisfiability and implication problems in database systems. *ACM Trans. Database Syst.*, 21(2):270–293.

[Hegner, 1990] Hegner, S. J. (1990). Foundations of canonical update support for closed database views. In *ICDT*, pages 422–436.

[Hegner, 2004] Hegner, S. J. (2004). An order-based theory of updates for closed database views. *AMAI*, 40(1-2):63–125.

[Hidders et al., 2005a] Hidders, J., Marrara, S., Paredaens, J., and Vercammen, R. (2005a). On the expressive power of XQuery fragments. In Bierman, G. M. and Koch, C., editors, *DBPL*, volume 3774 of *Lecture Notes in Computer Science*, pages 154–168. Springer.

[Hidders et al., 2007a] Hidders, J., Marrara, S., Paredaens, J., and Vercammen, R. (2007a). On the expressivity of functions in XQuery fragments. *Information Systems*. Accepted for Publication.

[Hidders et al., 2005b] Hidders, J., Michiels, P., Paredaens, J., and Vercammen, R. (2005b). LiXQuery: a formal foundation for XQuery research. *SIGMOD Record*, 34(4):21–26.

[Hidders et al., 2007b] Hidders, J., Michiels, P., Siméon, J., and Vercammen, R. (2007b). How to recognise different kinds of tree patterns from quite a long way away. In *PLAN-X*, pages 14–24.

[Hidders et al., 2006] Hidders, J., Paredaens, J., and Vercammen, R. (2006). On the expressive power of XQuery-based update languages. In Amer-Yahia, S., Bellahsene, Z., Hunt, E., Unland, R., and Yu, J. X., editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 92–106. Springer.

[Hidders et al., 2004] Hidders, J., Paredaens, J., Vercammen, R., and Demeyer, S. (2004). A light but formal introduction to XQuery. In *XSym*, pages 5–20.

[Ishikawa et al., 1998] Ishikawa, H., Kubota, K., and Kanemasa, Y. (1998). XQL: A query language for XML data. In *QL*.

[Katz et al., 2003] Katz, H., Chamberlin, D., Kay, M., Wadler, P., and Draper, D. (2003). *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Kay, 2004] Kay, M. (2004). *XPath 2.0 Programmer's Reference*. Wrox.

[Keller, 1985] Keller, A. M. (1985). Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS*, pages 154–163.

[Koch, 2006] Koch, C. (2006). On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256.

[Kozankiewicz et al., 2003] Kozankiewicz, H., Leszczylowski, J., and Subieta, K. (2003). Updatable XML views. In *ADBIS*, pages 381–399.

[Kuper et al., 2005] Kuper, G., Fabio, M., and Nataliya, R. (2005). Generalized XML security views. In *SACMAT 2005*, pages 77–84.

[Laux and Martin, 2000] Laux, A. and Martin, L. (2000). XUpdate — XML Update Language.

[Lechtenbörger, 2003] Lechtenbörger, J. (2003). The impact of the constant complement approach towards view updating. In *PODS*, pages 49–55.

[Lechtenbörger and Vossen, 2003] Lechtenbörger, J. and Vossen, G. (2003). On the computation of relational view complements. *ACM TODS*, 28(2):175–208.

[Libkin, 2003] Libkin, L. (2003). Expressive power of SQL. Theoretical Computer Science.

[Marx, 2005] Marx, M. (2005). Conditional xpath. *ACM Trans. Database Syst.*, 30(4):929–959.

[Marx and de Rijke, 2005] Marx, M. and de Rijke, M. (2005). Semantic characterizations of navigational xpath. *SIGMOD Record*, 34(2):41–46.

[Masunaga, 1984] Masunaga, Y. (1984). A relational database view update translation mechanism. In *VLDB*, pages 309–320.

[Neven and Schwentick, 2006] Neven, F. and Schwentick, T. (2006). On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3).

[Page et al., 2005] Page, W. L., Hidders, J., Michiels, P., Paredaens, J., and Vercammen, R. (2005). On the expressive power of node construction in XQuery. In Doan, A., Neven, F., McCann, R., and Bex, G. J., editors, *WebDB*, pages 85–90.

[Quass et al., 1996] Quass, D., Widom, J., Goldman, R., Haas, K., Luo, Q., McHugh, J., Nestorov, S., Rajaraman, A., Rivero, H., Abiteboul, S., Ullman, J. D., and Wiener, J. L. (1996). Lore: A lightweight object repository for semistructured data. In Jagadish, H. V. and Mumick, I. S., editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, page 549. ACM Press.

[Scholl et al., 1991] Scholl, M. H., Laasch, C., and Tresch, M. (1991). Updatable views in object-oriented databases. In *DOOD*, pages 189–207.

[Stoica and Farkas, 2002] Stoica, A. and Farkas, C. (2002). Secure XML views. In Gudes, E. and Shenoi, S., editors, *DBSec*, volume 256 of *IFIP Conference Proceedings*, pages 133–146. Kluwer.

[Suciu, 1998] Suciu, D. (1998). Semistructured data and xml. In Tanaka, K. and Ghandeharizadeh, S., editors, *FODO*, pages 1–12.

[Sur et al., 2004] Sur, G. M., Hammer, J., and Siméon, J. (2004). UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*.

[Tatarinov et al., 2001] Tatarinov, I., Ives, Z. G., Halevy, A. Y., and Weld, D. S. (2001). Updating XML. In *SIGMOD Conference*.

[ten Cate and Lutz, 2007] ten Cate, B. and Lutz, C. (2007). The complexity of query containment in expressive fragments of xpath 2.0. In Libkin, L., editor, *PODS*, pages 73–82. ACM.

[Tomasic, 1988] Tomasic, A. (1988). View update translation via deduction and annotation. In *ICDT*, pages 338–352.

[Vercammen, 2005] Vercammen, R. (2005). Updating XML views. In *VLDB PhD Workshop 2005*, pages 6–10.

[Vercammen et al., 2006] Vercammen, R., Hidders, J., and Paredaens, J. (2006). Query translation for xpath-based security views. In Grust, T., Höpfner, H., Illarramendi, A., Jablonski, S., Mesiti, M., Müller, S., Patranjan, P.-L., Sattler, K.-U., Spiliopoulou, M., and Wijsen, J., editors, *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 250–263. Springer.

[Wang et al., 2006] Wang, L., Rundensteiner, E. A., and Mani, M. (2006). U-filter: A lightweight XML view update checker. In Liu, L., Reuter, A., Whang, K.-Y., and Zhang, J., editors, *ICDE*, page 126. IEEE Computer Society.