UNIVERSITEIT ANTWERPEN
Departement Wiskunde en Informatica

Academiejaar 2004-2005

Expressieve kracht van de node constructie in XQuery

Wim Le Page

Eindwerk ingediend met het oog op het behalen van
de graad van Licentiaat in de Wetenschappen

Promotor:	prof. dr. J. Paredaens
Co-promotor:	dr. ir. J. Hidders
Medelezers:	P. Michiels, R. Vercammen

Dankwoord

In de eerst plaats wil ik graag prof. dr. Jan Paredaens bedanken. Daarnaast dank ik ook dr. ir. Jan Hidders, Philippe Michiels en Roel Vercammen voor hun hulp en medewerking aan deze thesis. Verder wil ik ook nog Jeroen Avonts en Pieter Wellens bedanken voor de vlotte samenwerking bij het project 'Blixem'.

Contents

List of Figures

List of XQuery Examples

List of Defined XQuery Functions

Samenvatting

0.1 Korte Samenvatting

XQuery is een krachtige en gestandaardiseerde manier op Extensibele Markup Language (XML) data te ondervragen. Omdat XML meer en meer gebruikt als d taal om data uit te wisselen, en XQuery als d taal om deze date te ondervragen is dan ook zinvol dat van zulk een belangrijke taal de eigenschappen worden bestudeerd. Het onderwerp van dit werk is het bestuderen van de expressieve kracht van de node constructie in XQuery. In het relationele model is er aangetoond dat de vlakke relationele algebra dezelfde expressieve kracht heeft als de geneste relationele algebra, wat betreft queries over vlakke relaties met vlakke resultaten [13]. Voor elke query die gebruik maakt van het geneste relationele model en die met een vlakke tabel als input een vlakke tabel als output geeft, bestaat er een equivalente vlakke query die enkel gebruik maakt van het vlakke relationele model. In analogie hiermee bestuderen we een gelijkaardig probleem voor XQuery. Voor elke expressie die operaties bevat die nieuwe nodes construeert en wiens resultaat enkel originele nodes bevat, bestaat eer een equivalente 'vlakke' query in XQuery die geen nieuwe nodes construeert. Dit tonen we in dit werk aan door een transformatie te definiren die zulke expressies omzet in equivalente expressies waarbij de constructie is gesimuleerd. Dit resultaat geeft ons een indicatie van de expressieve kracht van de node constructie in XQuery. Als basis voor de formele definitie van deze transformatie gebruiken we LiXQuery, een correct deeltaal van XQuery die bijna dezelfde expressieve kracht heeft. We gaan dan ook in op wat het verband is tussen de semantiek van de constructie in XQuery en LiXQuery.

0.2 Uitgebreide Samenvatting

Extensibele Markup Language (XML) [1] wordt meer en meer gebruikt als d taal om data uit te wisselen tussen vele verschillende applicatie domeinen en dan ook deze te bewaren. Daardoor zijn er grote hoeveelheden van verschillende datatypes in XML gecodeerd. De noodzaak om deze te ondervragen is dan ook een logisch gevolg. Om deze reden heeft het World Wide Web Consortium (W3C) *XQuery* [4] ontworpen als een krachtige en gestandaardiseerde manier op XML data te ondervragen. Omwille van het belang van structuur in XML is deze taal bovendien ook in staat om naast de inhoud de structuur te ondervragen. Het is dan ook zinvol dat van zulk een belangrijke taal de eigenschappen

worden bestudeerd. Het onderwerp van dit werk is dan ook het bestuderen van zo een bepaalde eigenschap van XQuery, namelijk de expressieve kracht van de node constructie.

0.2.1 XQuery en constructie

In XQuery is het mogelijk om XML te construeren. Dit laat ons toe om gestructureerde output te geven. Dit heeft verschillende toepassingen. Zo kunnen we nu gebruik maken van conceptuele groepen, vergelijkbaar met de views in SQL. Nog een andere mogelijkheid is het toepassen van transformaties. Denk hierbij bijvoorbeeld aan het omzetten van xml data naar xhtml, die dan getoond kan worden aan de gebruiker via een standaard browser. Ook tussentijdse datastructuren behoren tot de mogelijkheden. Deze bieden de mogelijkheid om gegevens uit verschillende xml datasets samen te voegen vergelijkbaar met joins in SQL.

XQuery heeft twee soorten constructie syntax. Enerzijds de direct constructors. Deze hebben dezelfde syntax als XML, maar hun dynamiek is beperkt. Zo is het dan bijvoorbeeld niet mogelijk om een XML element te creren waarvan de naam een resultaat is van de evaluatie van een andere expressie. Anderzijds hebben we de computed constructors. Deze hebben een XQuery specifieke syntax en laten daardor wel toe om bijvoorbeeld het resultaat van een expressie te gebruiken als naam.

Constructoren voegen nodes toe. In het XQuery data model is elke node node uniek, identiek aan zichzelf en verschillend van elke andere node. Atomaire waarden hebben geen identiteit. Elke instantie van een atomaire waarde is identiek aan elke andere zelfde instantie. De nodes gecreerd door constructie zijn nieuw en hebben dus elk hun eigen identiteit.

Node identiteit onderscheid dus nodes die er hetzelfde uitzien maar wel degelijk een verschillende oorsprong hebben (hetzij apart geconstrueerd, hetzij geselecteerd uit andere data sets). Natuurlijk willen we ook een manier om de gelijkheid tussen nodes te bepalen. Daarom definieert men deep equality. Twee nodes zijn deep equal als hetzelfde type, dezelfde naam, dezelfde verzameling attributen hebben en de sequentie van kinderen deep equal is. Voor atomaire waarden is deep equality hetzelfde als identiteit

Bij het herstructureren van XML met behulp van constructie in XQuery kunnen we geselecteerde nodes onder een nieuwe ouder hangen. De semantiek van XQuery definieert dat niet de originele geselecteerde nodes onder deze nieuwe nodes geplaatst dienen te worden, maar kopien ervan. Zulke gekopieerde nodes zijn deep equal aan de nodes waarmee ze gecreerd zijn, maar ze zijn niet identiek

0.2.2 Semantiek van XQuery

XQuery's typeringssysteem en de bijhorende impliciete semantiek die er in aanwezig is maken de taal moeilijk te overzien. XQuery is dan ook op twee manieren gespecificeerd. Enderzijds in gewone taal [4]. Deze specificatie is goed bruikbaar voor de normale gebruiker die XQuery dient te gebruiken om XML te ondervragen. Anderzijds in er een specificatie in formele symbolen [6]. Deze specificatie is dan weer noodzakelijk voor personen die een implementatie van XQuery willen maken, of onderzoekers die de eigenschappen van de taal willen bestuderen. Aangezien we tot de tweede categorie behoren spitsen we ons nu toe op de formele semantiek.

De formele semantiek heeft drie componenten. Ten eerste de *dynamische semantiek*, die de relatie tussen input data, de XQuery expressie en de output data bepaalt. Ten tweede de *statische semantiek*, die de relatie tussen type van input data, de XQuery expressie en het type van de output data bepaalt. Ten derde de *normalisatie regels*, deze transformeren XQuery in een kleine kerntaal die makkelijker te definiren is. Hierdoor wordt ook alle impliciete semantiek, zoals bijvoorbeeld een sorteer operatie, duidelijk zichtbaar. Wij gaan hier verder op de dynamische semantiek in aangezien deze relevant is voor ons verder verloop.

De dynamische semantiek is gespecificeerd via een operationele semantiek. Dit komt neer op verscheidene inferentie regels bestaande uit verschillende types oordelen. Zo zijn er bijvoorbeeld evaluatie oordelen die een verband tussen een expressie en de resulterende waarde uitdrukken. Er bestaan er matching oordelen die dan weer het verband tussen een bepaalde waarde en een bepaald type uitdrukken. Ook zijn er error oordelen die het verband bepalen tussen een expressie en een bepaalde error. Maar daarnaast ook nog vele anderen. Een inferentie regel bestaat uit hypotheses, opgebouwd met deze oordelen, die waar moeten zijn en een conclusie, ook bestaande uit zulke oordelen, die er dan uit volgt.

Als we de formele semantiek van de constructie bekijken zien we echter dat deze bijhoorlijk ingewikkeld is omwillen van het gebruik van namespaces en types. Als we deze echter achterwege laten, omdat we ze niet relevant beschouwen voor onze verdere aanpak, blijken deze echter al veel eenvoudiger te zijn. We bespreken deze semantiek uitgebreid in SubSectie 1.2.1.

0.2.3 LiXQuery

Zoals we zien is XQuery redelijk complex en is het niet eenvoudig de semantiek ervan op een precieze en beknopte manier te definiren. Er is dan ook behoefte aan een elegante en simpele deeltaal van XQuery die bijna dezelfde expressieve kracht heeft. Hiervoor is de *LiXQuery* [10, 11] deeltaal ontworpen. Bij het ontwerp is er rekening gehouden met een publiek van onderzoekers die de expressieve kracht van XQuery bestuderen. In deze deeltaal zijn namelijk enkel die features weggelaten die niet van belang zijn vanuit een theoretisch standpunt en op een zodanige manier dat het LiXQuery een juiste deeltaal is. Zo geldt het namelijk dat alle syntactisch correcte LiXQuery expressies ook syntactisch correct XQuery expressies zijn. Bovendien os het de set van de resultaten van een query gevalueerd met LiXQuery is een juiste subset van de set van de resultaten van de qeury gevalueerd met XQuery.

LiXQuery expressies worden gevalueerd tegen een XML store en een evaluation environment. Een XML store bevat al de documenten bereikbaar via het web (indien hij enkel deze documenten bevat spreken we van de initial XML store), en mogelijk XML fragmenten gecreerd (door middel van constructie) als intermediaire resultaten in een expressie. Een evaluation environment bevat essentieel informatie die functies en variabelen respectievelijk verbindt met hun body en inhoudt, alsook context informatie. Het resultaat van een expressie evaluatie in LiXQuery is een een (mogelijk uitgebreide) XML store genaamd de result store, en een sequentie van n of meerdere items uit deze store, genaamd de result sequence. We noteren dit geheel als $St, Env \vdash e \Rightarrow St', v$.

De semantiek van LiXQuery expressies wordt net zoals de formele semantiek van XQuery gedefinieerd via inferentie regels. Een LiXQuery inferentie regel bestaat uit een premise die bestaan bestaat uit verschillende oordelen die meestal de resultaten definiren van de evaluatie van subexpressies en condities erop en een conclusie, het oordeel dat men wil afleiden.

We geven alle semantische regels van LiXQuery in Sectie 2.2

We gaan wat dieper in op de definities van de semantiek van de constructie. Deze maken gebruik van de notie van deep equality zoals we dat voorheen hebben gedefinieerd. We geven een uitgebreide bespreking van de verschillende soorten constructie en leggen voor de element constructie het verband met de semantiek in XQuery in Sectie 2.3

Gebaseerd op de formele semantiek van expressie evaluatie en constructie in LiXQuery kunnen we verschillende soorten nodes die in een result sequence kunnen voorkomen definiren. Dit zijn ten eerste original nodes, deze zijn identiek aan een node in de initial XML store. Ten tweede new node, deze zijn nieuw geconstrueerd in de expressie (niet aanwezig in de initial XML store) . Ten derde deep equal nodes, deze zijn deep equal met een node in de initial XML store. Ten vierde copied nodes, deze zijn een kopie van een node in de initial XML store. Een diagram van hoe deze zich tot elkaar verhouden is gegeven.

We bespreken ook beknopt *Blixem*, de referentie implementatie van LiXQuery. Deze implementatie voorziet een volledige LiXQuery BNF, toegevoegd als appendix. Om correct te kunnen zijn volgt Blixem de LiXQuery formele semantiek zo dicht mogelijk en is hij bovendien robuust getest.

0.2.4 Expressieve kracht van de node constructie

Vervolgens komen we tot het hoofdonderwerp van dit werk? Een studie van de expressieve kracht van de node constructie. We tonen aan dat voor bepaalde expressies die node constructie kunnen bevatten er een expressie bestaan zonder node constructie die hetzelfde resultaat geeft. Dit resultaat geeft ons een indicatie van de expressieve kracht van de node constructie in LiXQuery. We bewijzen dit door de expressie te transformeren naar een andere expressie waarbij alle node constructie gesimuleerd wordt.

Expressies die nieuwe nodes in hun resultaat hebben kunnen niet worden gesimuleerd zonder node constructie. Daarom introduceren we de notie van een *Node Conservative Expression* (NCE). Een Node-Consercative Expression is een LiXQuery expressie zodanidg dat voor elke store en environment het geldt dat als $St, Env \vdash e \Rightarrow St', v$ dan alle nodes in v nodes in St zijn.

Het creren van nodes is een bron van non-determinisme in LiXQuery. Het gecreerde fragment kan namelijk op een arbitraire plaats in de document volgorde geplaatst worden. Non-determinisme kan dus niet gesimuleerd worden zonder node constructie omdat het er de enige bron van is. We kunnen ons echter beperken tot deterministische expressies omdat het niet determinisme geen fundamenteel feature van LiXQuery is. We definiren determinisme zeer strikt. Een LiXQuery expressie is *deterministisch* als er voor elke store en environment geldt dat als $St, Env \vdash e \Rightarrow St', v$ en $St, Env \vdash e \Rightarrow St'', w$ dan $v = w$.

De simulatie die we definiren wijkt op twee punten af van de originele expressie. Ten eerste mag de simulatie mag namelijk een gedefinieerd resultaat geven als de originele expressie dat niet doet. Ten tweede moeten de resulterende stores van het origineel en de simulatie maar dezelfde zijn op *garbage collection* na. Garbage collection komt er op neer dat we die nodes in de store verwijderen die niet meer bereikt kunnen worden in volgende expressies. Met andere woorden ze zijn niet meer te bereiken via een node in het resultaat of de doc functie. We noteren $[St]_v$ als garbage collection toegepast op St ten opzichte van v. Hiermee kunnen we dan een simulatie formeel definiren:

We zeggen dat een LiXQuery expressie e' een *simulatie* is van een LiXQuery expressie e als voor elke store St en environment Env met ongedefinieerde x, k en m geldt dat als $St, Env \vdash e \Rightarrow St', v$ dan er een store St'' bestaat zodanig dat $St, Env \vdash e' \Rightarrow St'', v$ en $[St'']_v = [St']_v$.

Met deze notie van simulatie definiren we dan het theorema dat het hoofdresultaat is van dit werk:

Voor elke deterministische node-conservative expressie e bestaat er een simulatie e' die geen constructoren bevat.

We bewijzen deze stelling door aan te tonen hoe de expressie e in zo een simulatie e' kan getransformeerd worden. We willen een constructie vrije simulatie van een expressie bekomen door de constructie te simuleren. Om constructie te simuleren moeten we de store simuleren. Hiervoor definiren we speciale variabelen in de environment om dat deel van de store te simuleren waar nieuwe nodes in zullen zitten, maar ook die nodes die door de doc functie worden bekomen. Een gesimuleerde doc() functie moet er dan nu ook voor zorgen dat de opgevraagde nodes gecodeerd zullen worden in deze speciale variabelen. Het opvragen van nodes uit de store is nu gesimuleerd door het opvragen in de gecodeerde store. Nodes worden gesimuleerd door node identifiers. Dit zijn nummer die verwijzen naar gecodeerde nodes in de gecodeerde store. Om atomaire waarden te kunnen onderscheiden van deze identifiers worden ze ook gecodeerd: atomaire waarden worden voorafgegaan door een 0, indentifiers door een 1. Zo bijvoorbeeld: $\langle \texttt{5}, v_1, \texttt{"string"}, v_2 \rangle \rightarrow \langle \texttt{0}, \texttt{5}, \texttt{1}, \texttt{5}, \texttt{0}, \texttt{"string"}, \texttt{1}, \texttt{3} \rangle$ als v_1 en v_2 respectievelijk 5 en 3 als node identifier hebben. Alle expressies moeten getransformeerd worden rekening houdend met deze wijzigingen. Op het einde zal de simulatie de node identifiers terug vervangen door de corresponderende nodes in de echte store. Dit is mogelijk als de originele expressie node-conservative was.

Om de simulatie uit te voeren hebben we een *transformatie functie* nodig die alle LiXQuery functies transformeert in hun gesimuleerde varianten. Het resultaat van de transformatie van een expressie e is inductief gedefinieerd op de structuur van de expressies e.

We overlopen de belangrijkste principes van deze simulatie. Enerzijds is er de gesimuleerde `doc()` functie. Deze is zeer uitgebreid omwille van het werken met de gesimuleerde store. Als een `doc()` call wordt uitgevoerd dient het gevraagde document volledig gecodeerd te worden en toegevoegd te worden aan de gecodeerde store. Dit moet op een zodanige wijze gebeuren dat we later in staat zijn de gesimuleerde nodes terug te vervangen door de originele echte nodes. In de for-expressie, filter-expressie en path-expressies wordt telkens een volgende expressie geevalueerd met als context de result store van de vorige Bij dit soort expressies is het aantal te evalueren subexpressies onbepaald We gebruiken recursieve functies om dit te simuleren. Deze functies worden gegenereerd aangezien hun inhoud ook een vooraf onbepaalde gesimuleerde expressie is. De simulaties van de constructoren vereisen dezelfde gecodeerde store manipulatie functies als aanwezig bij de `doc()` functie. Bovendien dienen deze simulaties ook een deep-copy uit te voeren om deep equal kinderen te kunnen creren.

Een voorbeeld van een transformatie is gegeven in Sectie 4.5.

Vooraleer we echter zulke getransformeerde expressies kunnen uitvoeren dienen we de input ervan te coderen, en nadat we zo een getransformeerde expressie hebben uitgevoerd dienen we het resultaat te decoderen. De basesitechniek van de codering is het maken van een sequentie van de root nodes van alle variabele bindingen in de echte environment en sorteer deze in document order. Deze encoderen levert de initile store op. We kiezen gecodedeerde node identifiers zodanig dat ze corresponderen met de positie in de gecodeerde store. Op deze manier kunnen we eenvoudig de variabele bindingen vervangen door gecodeerde versies om zo de gecodeerde environment bekomen. De methode van het decoderen hangt af van de origine van de nodes in het resultaat. Nodes die in de initial store zaten zijn eenvoudig te vervangen als we de mapping ervan hebben bijgehouden. Nodes waarvan de root een document node in delta is, zijn te vervangen door een echte doc() call uit te voeren en de juiste node te selecteren op basis van positie (deze zit namelijk gecodeerd in de gecodeerde versie van het document).

0.2.5 Conclusie

We concluderen dat de stelling inderdaad klopt voor node-conservatie expressies. We bekijken echter ook nog in Sectie 5.1 wat de mogelijkheden zijn voor andere types expressies. Expressies waarbij er ook gekopieerde nodes voorkomen in het resultaat, en expressies waarbij we alle deep equal nodes in het resultaat voorkomen.

Introduction

Extensible Markup Language (XML) [1] has become the language of choice for storing and transmitting data across diverse application domains. It is used in corporate IT departments, academic research institutions or small programming projects. XML therefore encodes a huge amount of datatypes scattered across a large number of diverse application domains. Of course, with this vast store of XML-encapsulated information, there are people who are going to need to query it. *XQuery*, an XML Query Language [4] invented by the World Wide Web Consortium, offers a powerful, standardized way to do just that. XQuery provides a flexible and easy-to-use mechanism for querying not only content, but structure as well. With its ability to integrate XML and non-XML data, XQuery seems to be able do for XML what SQL has done for relational data. It is therefore important to study the properties of this powerful query language. However this language is rather complex and its not easy to define its semantics in a precise and concise manner. For this reason, J. Hidders, J. Paredaens and R. Vercammen have defined *LiXQuery* [10, 11], an elegant and simple sublanguage of XQuery. LiXQuery has almost the same expressive power as XQuery, but has the advantage that it's syntax and semantics can be written down in a few pages. This sublanguage was designed with the audience of researchers investigating the expressive power of XQuery in mind. It will therefore form the basis of our study.

In XQuery, a query result can contain atomic values and different types of nodes. The nodes can be original (selected from the input) or they can be new nodes, not occurring in the input. These new nodes are constructed during the evaluation of the expression. Constructing new XML nodes is useful for several purposes, including creating a new result shape (transformation), representing temporary intermediate data structures (composition), and organizing data into conceptual groups (views). It is important to realize that these new nodes can therefore be copies of nodes occurring in the input or can be created to be (deep) equal to nodes in the input. Nevertheless, it is still possible that only original nodes occur in the final result. We call such expressions *node-conservative*. For example, the query in Example 0.1 creates new nodes not occurring in the result. In this example we perform a join and a projection of two XML documents in XQuery.

In the relational model it has been shown that the flat relational algebra has the same expressive power as the nested relational algebra, as far as queries over flat relations and with flat results are concerned [13]. Hence, for each query that uses the nested relational model and that, with a flat table as input always has a flat table as output, there exists an equivalent flat query that only uses the flat relational model. In [14] a very direct proof

Example 0.1 Node-Conservative Expression

The following XQuery expression

```
let $jointtable :=
        element {"table"}{
            for $b1 in doc("table.xml")/table/row
            for $b2 in doc("table2.xml")/table/row
            where $b1/a = $b2/a
            return element{"row"}{$b1/*,$b2/*} }
return
    for $b in $jointtable/row/b return string($b)
```

has the result sequence `"one","two"`when given the input documents `table.xml` and `table2.xml` which look as follows

```
<table>
  <row><a>1</a><b>one</b></row>
  <row><a>2</a><b>two</b></row>
  <row><a>3</a><b>three</b></row>
</table>

<table>
  <row><a>1</a><c>red</c></row>
  <row><a>2</a><c>blue</c></row>
</table>
```

is given of this fact using a simulation technique. In analogy, we study a related flat-flat problem for XQuery. We show that for each node-conservative expression there exists an expression without node construction yielding the same result. For example, the query in Example 0.1 can be rewritten to the query shown in Example 0.2. In this work we will show how to generate automatically equivalent constructor-free expressions for node-conservative expressions. This result gives an indication of the expressive power of the node construction.

Furthermore it can be interesting for query optimization, since optimizing node construction can be hard. For example, consider the translation of XQuery expressions to SQL, where complete translations also have to deal with simulating node construction. In [9] such a translation is given, where the construction of new elements yields larger SQL statements which are harder to optimize.

Example 0.2 Constructor-Free Expression

The following XQuery expression

```
for $b1 in doc("table.xml")/table/row
for $b2 in doc("table2.xml")/table/row
where $b1/a = $b2/a
return
   for $b in ($b1/*, $b2/*)/b return string($b)
```

is equivalent to the query of Example 0.1 and does not contain node constructors.

Other work studied the effect of adding object creation to query languages on the expressive power of these languages. For example, in [7] the effect of object identity on the power of query languages is studied and a notion of determinate transformations is introduced as a generalization of the standard domain-preserving transformations. However, obvious extensions of complete database programming languages with object creation are not complete for determinate transformations. In [15] this mismatch is solved by introducing the notion of constructive transformations, a special kind of determinate transformations which are precisely the transformations that can be expressed by these obvious extensions.

This work is structured as follows. In Chapter 1 we give some background information about the creation of nodes in XQuery. Chapter 2 is an introduction to LiXQuery, which we will use as a formal model for XQuery and for proving our theorems. In Chapter 3 we discuss the *Blixem* project, which is the reference implementation of LiXQuery. Chapter 4 investigates the expressive power of the construction. Finally, the conclusion of this work is presented in Chapter 5.

Chapter 1

XQuery, An XML Query Language

In this Chapter we look at XQuery, the standard language for querying XML documents. We will focus on the mechanisms of construction in XQuery and the related topics of Node Identity, Deep-equality and copying, and shed some light on the formal semantics which describes this language.

1.1 Construction in XQuery

In XQuery it is possible to construct XML directly in a query. This is a very useful feature which allows us to create structured output. This structured output can organize data in conceptual groups similar to the use of views in SQL. Another possibility is to use this ability for transformations. One can for example have xml files to store the data of a system and transform this data into *xhtml* to display it via a standard web browser to the users of the system. Construction can also be used to represent temporary intermediate data structures which allow for composition, similar to joins in SQL.

1.1.1 Constructors

In XQuery expressions exist for creating all the XML node kinds. The basic element, attribute and text nodes can be created, but also comment, processing-instruction and namespace nodes. The way these can all be created is using the same syntax as XML. But this syntax, called the *direct constructor* syntax, is not sufficient. Sometimes you will want to simulate a document loaded by the built-in `doc()` function, by writing a function that will return a computed document instead of one loaded from XML. Sometimes you will want to create element or attribute nodes that have a name which is the result of an another XQuery expression. For these reasons there exists an alternative, more powerful XML construction syntax in XQuery, the so called *computed constructor* syntax. An example of such syntax is given in the expression in Example 1.1 where the new element `$book1` and its subelements are constructed. We also demonstrate the feature to have the name of the element be the result of an expression in this example. The value of the attribute `name` of the first `extra` element in the input document is used as the name for the fourth element in `$book1`.

One of the basic functions of construction is the creation of new nodes which have certain selected elements as their children. These children will be created as copies of the selected nodes. In Example 1.1 we can see that the `author` element of the newly constructed book is created as a copy of the `author` of the first `book` in the input document. We can clearly see that it is a copy in the result sequence of the expression.

In the result sequence of the expression in Example 1.1 the `publisher` element and the `extra_field` element are totally new nodes, the first `author` element is an original node, the second `author` element is a copied node and `14.95` and "`Someone Else`" are atomic values.

So informally we can say that in a result of an expression which contains construction we can get three types of nodes:

- orginal nodes, selected by an expression
- copies of original nodes, created as children of constructed nodes
- totally new (constructed) nodes, as a result of constructor expressions

and besides these nodes, atomic values (strings, numbers, ...).

1.1.2 Node Identity

In XQuery it is necessary to distinguish nodes from each other. Even if they are for example element nodes that have exactly the same name, attributes and values, they can still differ. They can for example originate from totally different documents. To be able to make this distinction in XQuery each node has a unique identity. Every node in an instance of the XQuery data model is unique: **identical** to itself, and not identical to any other node. It is important to note that atomic values do not have an identity. Every instance of the value "5" as an integer is **identical** to every other instance of the value "5" as an integer.

So in the light of this the expression in Example 1.2 returns true, because it compares the exact same nodes. The `is-operator` provides us with a test for node identity.

But the expression in Example 1.3 returns false, because two different nodes are compared which were created here by use of the XQuery element constructor (using the XML syntax).

Two sequences of items are **identical** if they have the same number of items and every item in the first sequence is identical to the item at the same position in the second sequence.

1.1.3 Deep equality

We now now it is important to be able to distinguish two nodes even if they look the same. It is however equally important to be able to know if two nodes indeed do look the same,

Example 1.1 Computed Element

The following XQuery expression

```
let $book1 := element {"book"}
{
    attribute {"year"} { 1977 },
    doc("input.xml")/books/book[1]/author,
    element {"publisher"} {"Puzzin Books"},
    element {"price"} { 14.95 },
    element {fn:string(doc("input.xml")/books/extra[1]/@name)}
      {fn:data(doc("input.xml")/books/extra[1])}
}
return
  ($book1/publisher, doc("input.xml")/books/book[1]/author,
  $book1/author, $book1/*[4], fn:data($book1/price),
  fn:data($book1/author))
```

has the result sequence

```
<publisher>Puzzin Books</publisher>,
<author>Someone Else</author>,
<author>Someone Else</author>,
<extra_field>Extra content</extra_field>,
14.95,
Someone Else
```

when given this input document (input.xml)

```
<books>
<book>
<author>Someone Else</author>
</book>
<extra name="extra_field">Extra content</extra>
</books>
```

Example 1.2 Identical Nodes

```
let $book1 :=
  doc("input.xml")/books/book[1]
return
$book1 is doc("input.xml")/books/book[1]
```

Example 1.3 Non-identical Nodes

```
<book1 /> is <book1 />
```

or to put it an another way, to know if two nodes are equal. That is why we introduce the notion of *deep equality*. Informally we can say that two nodes are *deep equal* if they are of the same kind[1], with the same name and have the same attributes (the order of which may vary between the two) and deep equal children (the order of which must be the same).

Note that these nodes may differ in their parent. For atomic values the notion of *deep equality* coincides with *identity*.

To illustrate this notion we can see that the arguments of the `fn:deep-equal` function in Example 1.4 are indeed the same except for attribute order and node identity. The expression in this example therefore returns true. In a XQuery expression deep equality can be tested with the `deep-equal` function.

Example 1.4 Deep equal items

```
fn:deep-equal(
<book year="1998" ISBN="1111111">
  <author>Someone</author>
</book>,
<book ISBN="1111111" year="1998">
  <author>Someone</author>
</book>)
```

We will now define deep equality for items and sequences in a more formal way:

Definition 1.1 (Deep equal)**.** Two items are **deep equal** if they are both atomic values that compare equal, or they are nodes of the same kind, with the same name, whose sets[2] of attributes are deep equal and whose sequences[3] of children are deep equal.

Two sequences of items are **deep equal** if they have the same number of items and every item in the first sequence is deep equal to the item at the same position in the second sequence.

1.1.4 Copying

One of the things construction in XQuery was useful for was the ability to structure the output or intermediate nodes. This is achieved by creating new element nodes and selecting

[1]one of the seven kinds of nodes (`document, element, attribute, text, namespace, processing instruction, and comment`) defined in [5].

[2] The order of the attributes may differ

[3]The order of the children must be the same

nodes from the input to be their children. As mentioned before the children of these new nodes are not the original nodes selected from the output, but are deep equal versions of these nodes with new node identities. We will call these nodes *copies* of the original nodes. Now we will define the notion of a *copy* more formally.

Definition 1.2 (Direct copy of)**.** Item A is a **direct copy of** item B if it is created as a descendant of a newly constructed element C, by selecting[4] the item B or an ancestor of B in the constructor of element C.

Item A is therefore deep-equal to item B but as it is a new node it has a new node identity and therefore it is not identical to B. Therefore in Example 1.1 the two `author` elements in the result sequence are deep-equal but not identical because the second `author` element is a direct copy of the first.

Definition 1.3 (Copy of)**.** We define the binary relation **copy of** as the transitive closure[5] of the binary relation *direct copy of.*

This enables us to include copies of copies, copies of copies of copies, etc.. in the notion of *copy.*

1.2 Formal Semantics

XQuery is not a simple language. The typing system it contains and the many implicit casts that go with it makes it hard to oversee. The XQuery language is specified by in two ways. In the XQuery language document[4] normal prose is used to describe the language specifications of XQuery. In the XQuery Formal Semantics document[6] symbols are used to do the same. These two documents have their own benefits. The prose document can be easily read by people who want to be able to use the language to query XML data. But prose always lacks some degree precision that only symbols can achieve. Therefore these Formal Semantics are also vital, especially for implementers writing an complete, correct, and conforming XQuery engine and for researchers who are investigating expressive power, optimizations or other features of the language.

The formal semantics of XQuery has three components: a dynamic semantics, a static semantics, and normalization rules. The dynamic semantics specifies the relationship between input data, an XQuery expression, and output data. The static semantics specifies the relationship between the type of the input data, an XQuery expression, and the type of the output data. The normalization rules transform full XQuery into a small core language, which is easier to define, implement, and optimize. In this process the implicit

[4]This selection can be as a result of a XPath expression or a complete FLWOR expression (more details see [4])

[5] A transitive closure is an extension or superset of a binary relation such that whenever (a,b) and (b,c) are in the extension, (a,c) is also in the extension

functionality and casts (e.g. a atomic string value converted to a text node) will become apparent. The dynamic and static semantics are defined in terms of the core language.

The technique that is used to specify XQuery's static and dynamic semantics is commonly known as an *operational semantics.* An operational semantics is specified using *inference rules* (similar to those used in the study of logic). Inference rules consists mainly of *evaluation judgments.* In their general form they are written as $Env \vdash Expr \Rightarrow Value$, which you read as "In environment Env, the evaluation of expression $Expr$ yields the value $Value$." The environment can for example be the dynamic environment which contains among other things the values of variables. To make up an inference rule with evaluation judgements you have zero or more judgments above a line, called the *hypotheses* or *premises*, and one judgement below the line, called the *conclusion.* Such a rule means to say that when all the *hypotheses* are true, then the conclusion must also be true. A trivial example may look like this:

$$\frac{dynEnv \vdash Expr_1 \Rightarrow true \qquad dynEnv \vdash Expr_2 \Rightarrow true}{dynEnv \vdash Expr_1 \texttt{ and } Expr_2 \Rightarrow true}$$

Of course not all rules are as simple as this, even this rule is preceded by a normalization phase where the *effective boolean value*[6] is extracted from the expressions. Besides these standard judgments there are amongst others special *matching judgments*, to enable typing, and *error judgments* for raising errors, and ways to indicate the values of variables in the environment. The following rules illustrate this:

$$\frac{\begin{array}{c} dynEnv \vdash Expr_1 \Rightarrow Value_1 \qquad Type_0 = [SequenceType]_{sequencetype} \\ Value_1 \ \textbf{\texttt{matches}}\ Type_0 \qquad statEnv \vdash VarRef \ \textbf{\texttt{of var expands to}}\ Variable_1 \\ dynEnv + varValue(Variable_1 \Rightarrow Value_1) \vdash Expr_2 \Rightarrow Value_2 \end{array}}{dynEnv \vdash \ \texttt{let \$}VarRef_1 \ \texttt{as}\ SequenceType \ \texttt{:=}\ Expr_1 \ \texttt{return}\ Expr_2 \Rightarrow Value_2}$$

$$\frac{dynEnv \vdash Expr_1 \Rightarrow Value_1 \qquad Type_0 = [SequenceType]_{sequencetype} \qquad \textbf{\texttt{not}}(Value_1 \ \textbf{\texttt{matches}}\ Type_0)}{dynEnv \vdash \ \texttt{let \$}VarRef_1 \ \texttt{as}\ SequenceType \ \texttt{:=}\ Expr_1 \ \texttt{return}\ Expr_2 \ \textbf{\texttt{raises}}\ typeError}$$

The **`of var expands to`** judgment is for expanding qualified names in the light of namespaces, and the *sequencetype* normalization rule resolves the type name to a type.

These were just some illustrative examples form the *dynamic semantics.* Other types of judgments exist for the dynamic semantics and a way to write down normalization rules. We will not go into these, and refer to the XQuery Formal Semantics document[6]. Next we will take a closer look at one inference rule in particular, that of the element constructor.

[6]In XQuery special rules exist that determine how to convert an item or a sequence to its boolean value

1.2.1 Formal Semantics of the Element Constructor

In Section 1.1 we gave an introduction of construction in XQuery. Now we will examine it more closely using the formal semantics. We will limit ourselves to the dynamic semantics of computed element constructor since the formal semantics of other computed constructors are similar, and the direct constructor variants are normalized to their computed version. The The dynamic semantics for computed element constructors is the most complex of all expressions in XQuery. If we look at the formal inference rule for element construction as it appears in [6] it looks like this[7]:

$$\frac{\begin{array}{c} \mathit{dynEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Value}_0 \qquad \mathit{statEnv} \vdash \mathit{Value}_0 \ \texttt{matches xs:Qname} \\ \mathit{Expr}_2 = \mathit{CompElemNamespace}_1, ..., \mathit{CompElemNamespace}_n, (\mathit{Expr}_3) \\ \mathit{CompElemNamespace}_1 = \ \texttt{namespace} \ \mathit{NCName1}\{\mathit{URI}_1\} \\ ... \qquad \mathit{CompElemNamespace}_n = \ \texttt{namespace} \ \mathit{NCNamen}\{\mathit{URI}_n\} \\ \mathit{statEnv}_1 = \mathit{statEnv} + \texttt{namespace}(\mathit{NCName} \Rightarrow (\mathit{active}, \mathit{URI}_1)) \\ ... \qquad \mathit{statEnv}_n = \mathit{statEnv}_{n-1} + \ \texttt{namespace}(\mathit{NCName} \Rightarrow (\mathit{active}, \mathit{URI}_n)) \\ \mathit{statEnvn}, \mathit{dynEnv} \vdash \texttt{fs:item-sequence-to-node-sequence}(\mathit{Expr}_3) \Rightarrow \mathit{Value}_1 \\ \mathit{statEnvn} \vdash \mathit{Value}_1 \ \texttt{matches} \ (\mathit{attribute*},\ \mathit{(element} \mid \mathit{text} \mid \mathit{processing\text{-}instruction} \mid \mathit{comment)*}) \\ \mathit{NamespaceBindings} = (\mathit{CompElemNamespace}_1, ...\mathit{CompElemNamespace}_n), \\ \texttt{fs:active\_ns}(\mathit{statEnv.namespace}), \texttt{fs:get\_static\_ns\_from\_items}(\mathit{statEnv}, \mathit{Value}_1) \end{array}}{\mathit{statEnv}, \mathit{dynEnv} \vdash \texttt{element} \ \{\mathit{Expr}_1\} \ \{\mathit{Expr}_2\} \ \Rightarrow \texttt{element} \ \{\mathit{Value}_0\} \ \texttt{of type xdt:untyped} \ \{\mathit{Value}_1\}}$$

This rule is rather complex due to it's use of namespaces. As we do not consider namespaces to be essential for our further approach, we can omit them here. This way the rule becomes a lot simpler:

$$\frac{\begin{array}{c} \mathit{dynEnv} \vdash \mathit{Expr}_1 \Rightarrow \mathit{Value}_0 \qquad \mathit{statEnv} \vdash \mathit{Value}_0 \ \texttt{matches xs:Qname} \\ \mathit{statEnvn}, \mathit{dynEnv} \vdash \texttt{fs:item-sequence-to-node-sequence}(\mathit{Expr}_2) \Rightarrow \mathit{Value}_1 \\ \mathit{statEnvn} \vdash \mathit{Value}_1 \ \texttt{matches} \ (\mathit{attribute*},\ \mathit{(element} \mid \mathit{text} \mid \mathit{processing\text{-}instruction} \mid \mathit{comment)}*) \end{array}}{\mathit{statEnv}, \mathit{dynEnv} \vdash \texttt{element} \ \{\mathit{Expr}_1\} \ \{\mathit{Expr}_2\} \ \Rightarrow \texttt{element} \ \{\mathit{Value}_0\} \ \texttt{of type xdt:untyped} \ \{\mathit{Value}_1\}}$$

First we see that the name expression (Expr_1) is evaluated and its result value (Value_0) is checked to see that it matches `xs:QName`, which is the type for qualified names.

Second, the function `fs:item-sequence-to-node-sequence()` is applied to the element's content expression (Expr_2). This function call is evaluated in the static and dynamic environment. This function converts a sequence of item values to a sequence of nodes called the *content sequence*, by applying the normative rules of the element construction. It has most of the construction semantics packed in to it, but its semantics is only described informally as it refers to the XQuery language document[4]. We will look at these

[7] In fact, it does not appear like this in the considered working draft of the formal semantics [6]. The conclusion of the dynamic semantics of the element constructor differs, and is written as: $\mathit{statEnv}, \mathit{dynEnv} \vdash \texttt{element} \ \{\mathit{Expr}_1\} \ \{\mathit{Expr}_2\} \ \Rightarrow \mathit{Value}_1$. This is in fact wrong, as it would imply that an element constructor returns the children of the constructed element. As we can deduce from the XQuery language document[4], the static semantics of the element constructor and the semantics of the element constructor this is certainly not the case. Therefore we corrected this error here.

normative rules, but we will leave out the rules concerning namespaces and typing (as this also adds extra complexity but we do not consider it essential in our further approach):

- For each adjacent sequence of one or more atomic values returned by an enclosed expression, a new text node is constructed, containing the result of casting each atomic value to a string, with a single space character inserted between adjacent values.
- For each node returned by an enclosed expression, a new copy is made of the given node and all nodes that have the given node as an ancestor, collectively referred to in what follows as *copied nodes*.

  Each copied node receives a new node identity. The parent, children, and attributes properties of the copied nodes are set so as to preserve their inter-node relationships. For the topmost node (the node directly returned by the enclosed expression), the parent property is set to the newly-constructed element node.

  All other properties of the copied nodes are preserved.
- Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.
- If the content sequence contains a document node, the document node is replaced in the content sequence by its children.
- If the content sequence contains an attribute node following a node that is not an attribute node, a type error is raised

The resulting value $Value_0$ must match zero-or-more attributes followed by zero-or-more element, text, processing-instruction or comment nodes.

The properties of the newly constructed element node are determined as follows:

- It has a new node identity.
- The node-name is the qualified name resulting from $Expr_1$.
- The parent is the node constructed by the nearest containing element or document node constructor, if such a constructor exists; otherwise parent is empty.
- The attributes consist of all the attribute nodes in the content sequence. The parent property of each of these attribute nodes has been set to the newly constructed element node.
- The children consist of all the element, text, comment, and processing instruction nodes in the content sequence. The parent property of each of these nodes has been set to the newly constructed element node.

– The string-value property is equal to the concatenated contents of the text-node descendants in document order.

As we can see, the main semantic of the element constructor is not stated formally but is captured informally in the meaning of the `fs:item-sequence-to-node-sequence()` function call.

Chapter 2

LiXQuery, A Formal Foundation for XQuery Research

This XQuery language is rather complex and its not easy to define its semantics in a precise and concise manner. For this reason, LiXQuery [10, 11] has been defined. LiXQuery is a fully downwards compatible sublanguage of XQuery that has almost the same expressive power as XQuery and that has a compact and well defined syntax and semantics. In this Chapter we give a short introduction to LiXQuery, which we will use as a basis for studying the expressive power of node construction in XQuery.

2.1 LiXQuery Syntax and Semantics

The LiXQuery language was designed with the audience of researchers investigating the expressive power of XQuery in mind. The XQuery features that are omitted in LiXQuery are therefore only those that are not essential from a theoretical perspective. However, to ensure the validity of LiXQuery, it is designed as a proper sublanguage. Specifically, all syntactically valid LiXQuery expressions do also satisfsfy the XQuery syntax. Moreover, the LiXQuery semantics is defined in such a way that the result of a query evaluated using the LiXQuery semantics will be a proper subset of the same query evaluated by XQuery. The lack of a complete formal sematics for XQuery does not allow this to be proved.

LiXQuery is largely the same as XQuery but has only a few built-in functions and no primitive data-types, order by clause, namespaces, comments, programming instructions and entities. Furthermore it ignores typing and only provides `descendant-or-self` and `child` as navigational axes, but the other navigational axes can be simulated using these 2 axes. Although the features that LiXQuery lacks, are important for practical purposes, they are not relevant to the study of the expressive power. Note that LiXQuery does support recursive functions, positional predicates and atomic values, which will prove to be essential in our later approach. The syntax of LiXQuery is given in Fig. 2.1 as an abstract syntax, i.e., it assumes that extra brackets and precedence rules are added for disambiguation.

The non-terminal $\langle \mathit{Name} \rangle$ 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 $\langle \mathit{String} \rangle$ refers to strings that are enclosed in double quotes such as in `"abc"` and $\langle \mathit{Integer} \rangle$ refers to integers such as `100`, `+100`, and `-100`.[1] Therefore the sets associated with $\langle \mathit{Name} \rangle$, $\langle \mathit{String} \rangle$ and $\langle \mathit{Integer} \rangle$ are pairwise disjoint.

The ambiguity between rule [5] and [24] is resolved by giving precedence to [5], and for path expressions we will assume that the operators "`/`" and "`//`" (rule [18]) are left associative and are preceded by the filter operation (rule [17]) in priority.

A full BNF of LiXQuery can be found as Appendix A, as part of the Blixem project discussed in Chapter 3.

We define LQE as the set of LiXQuery expressions. In LiXQuery, *expressions* are evaluated against an *XML store* and an *evaluation environment.* The XML store contains the XML fragments that are created as intermediate results in an expression, as well as all the documents accessible from the web. The store that only contains all the web documents is called the *initial XML store.* The evaluation environment essentially contains mapping information for function names, variable names and the context item (including context position in the context sequence and the context sequence size). Formally, the XML store is a defined as a 6-tuple[2] $St = (V, E, \ll, \nu, \sigma, \delta)$ where: V is the set of available nodes; (V, E) forms an acyclic directed graph to represent the tree-structures; $\ll$ defines a total order over the nodes in V; ν labels element and attribute nodes with their node name; σ labels the attribute and text nodes with their string value; δ is a partial function that uniquely associates with an URI or a file name, a document node.

The environment in LiXQuery is denoted by a tuple $Env = (a, b, v, x, k, m)$ where a is a partial function that maps a function name to its formal argument; b is a partial function that maps a function name to the body of the function; v is a partial function that maps variable names to their values; x is an item of St and indicates the context item or x is undefined; k is an integer denoting the position of the context item in the context sequence or k is undefined; m is an integer denoting the size of the context sequence, or m is undefined.

The result of an expression evaluated against an XML store and environment is a (possibly expanded) XML store (*result store*) and a sequence of one or more items over the result store (*result sequence*). Items in the result sequence can either be atomic values or nodes. Like in XQuery, each node has a unique identity while atomic values do not.

Formal definitions of the XML store and the environment can be found in [10, 11].

[1]Integers are the only numeric type that exists in LiXQuery.

[2]This tuple is the same as in [11] except that the sibling order $<$ is replaced by the document order $\ll$.

[1] ⟨*Query*⟩	→	(⟨*FunDef*⟩“;”)* ⟨*Expr*⟩
[2] ⟨*FunDef*⟩	→	“declare” “function” ⟨*Name*⟩ “(”(⟨*Var*⟩(“,”⟨*Var*⟩)*)?“)”
		“{”⟨*Expr*⟩“}”
[3] ⟨*Expr*⟩	→	⟨*Var*⟩ | ⟨*BuiltIn*⟩ | ⟨*IfExpr*⟩ | ⟨*ForExpr*⟩ | ⟨*LetExpr*⟩ | ⟨*Concat*⟩ |
		⟨*AndOr*⟩ | ⟨*ValCmp*⟩ | ⟨*NodeCmp*⟩ | ⟨*AddExpr*⟩ | ⟨*MultExpr*⟩ |
		⟨*Union*⟩ | ⟨*Step*⟩ | ⟨*Filter*⟩ | ⟨*Path*⟩ | ⟨*Literal*⟩ | ⟨*EmpSeq*⟩ |
		⟨*Constr*⟩ | ⟨*TypeSw*⟩ | ⟨*FunCall*⟩
[4] ⟨*Var*⟩	→	“$”⟨*Name*⟩
[5] ⟨*BuiltIn*⟩	→	“doc(”⟨*Expr*⟩“)” | “name(”⟨*Expr*⟩“)” | “string(”⟨*Expr*⟩“)” |
		“xs:integer(”⟨*Expr*⟩“)” | “root(”⟨*Expr*⟩“)” |
		“concat(”⟨*Expr*⟩, ⟨*Expr*⟩“)” | “true()” | “false()” |
		“not(”⟨*Expr*⟩“)” | “count(”⟨*Expr*⟩“)” | “position()” | “last()”
[6] ⟨*IfExpr*⟩	→	“if ”“(”⟨*Expr*⟩“)” “then”⟨*Expr*⟩ “else”⟨*Expr*⟩
[7] ⟨*ForExpr*⟩	→	“for”⟨*Var*⟩(“at”⟨*Var*⟩)? “in”⟨*Expr*⟩ “return”⟨*Expr*⟩
[8] ⟨*LetExpr*⟩	→	“let”⟨*Var*⟩“:=”⟨*Expr*⟩ “return”⟨*Expr*⟩
[9] ⟨*Concat*⟩	→	⟨*Expr*⟩“,”⟨*Expr*⟩
[10] ⟨*AndOr*⟩	→	⟨*Expr*⟩(“and” | “or”)⟨*Expr*⟩
[11] ⟨*ValCmp*⟩	→	⟨*Expr*⟩(“=” | “<”)⟨*Expr*⟩
[12] ⟨*NodeCmp*⟩	→	⟨*Expr*⟩(“is” | “<<”)⟨*Expr*⟩
[13] ⟨*AddExpr*⟩	→	⟨*Expr*⟩ (“+” | “-”) ⟨*Expr*⟩
[14] ⟨*MultExpr*⟩	→	⟨*Expr*⟩ (“*” | “idiv”) ⟨*Expr*⟩
[15] ⟨*Union*⟩	→	⟨*Expr*⟩“|”⟨*Expr*⟩
[16] ⟨*Step*⟩	→	“.” | “..” | ⟨*Name*⟩ | “@”⟨*Name*⟩ | “*” | “@*” | “text()”
[17] ⟨*Filter*⟩	→	⟨*Expr*⟩“[”⟨*Expr*⟩“]”
[18] ⟨*Path*⟩	→	⟨*Expr*⟩(“/” | “//”)⟨*Expr*⟩
[19] ⟨*Literal*⟩	→	⟨*String*⟩ | ⟨*Integer*⟩
[20] ⟨*EmpSeq*⟩	→	“()”
[21] ⟨*Constr*⟩	→	“element”“{”⟨*Expr*⟩“}” “{”⟨*Expr*⟩“}” |
		“attribute”“{”⟨*Expr*⟩“}” “{”⟨*Expr*⟩“}” |
		“text”“{”⟨*Expr*⟩“}” | “document”“{”⟨*Expr*⟩“}”
[22] ⟨*TypeSw*⟩	→	“typeswitch ”“(”⟨*Expr*⟩“)” (“case” ⟨*Type*⟩ “return”⟨*Expr*⟩)$^+$
		“default” “return”⟨*Expr*⟩
[23] ⟨*Type*⟩	→	“xs:boolean” | “xs:integer” | “xs:string” | “element()” |
		“attribute()” | “text()” | “document-node()”
[24] ⟨*FunCall*⟩	→	⟨*Name*⟩“(”(⟨*Expr*⟩(“,”⟨*Expr*⟩)*)?“)”

Figure 2.1: Syntax for LiXQuery queries and expressions

2.2 Formal Semantics of LiXQuery Expressions

The semantics of a LiXQuery expression is defined by statements of the form $St, Env \vdash e \Rightarrow St', v$, which state that when e is evaluated against a store St and an environment Env then St' is the result store and v is the result sequence over St. We derive such statement by using inference rules, which are given in [11]. The LiXQuery inference rules are similar to those of the formal semantics of XQuery as described in Chapter 1 Section 1.2. Each LiXQuery inference rule consists of a set of premises and a conclusion which will be the statement we want to derive. The premise generally consists of statements which define the result of the evaluation of the subexpressions and how they relate to the result of the statement we are inferring. It also contains conditions on the expressions and their results which must be true in order to apply the inference rule. The free variables in the rules are always assumed to be universally quantified. In these rules 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. We denote the empty sequence by $\langle\rangle$, non-empty sequences by, for example, $\langle 1, 2, 3\rangle$ and the concatenation of two sequences l_1 and l_2 by $l_1 \circ l_2$.

We will now continue by listing all the semantic rules of LiXQuery. We will focus on some of the rules to compare them with the XQuery counterparts given in Section 1.2 in Chapter 1. We will also explain the rules for the construction in detail separately in Section 2.3.

2.2.1 Semantic Rules

Rule 2.1 (Query (Rules [1] and [2])). A function declaration extends **a** and **b** and then the last expression is evaluated with these **a** and **b**. We express this using the following notation. 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 **a** (**b**, **v**) maps n to y. Similarly, we let $En[\mathbf{x} \mapsto y]$ ($En[\mathbf{k} \mapsto y]$, $En[\mathbf{m} \mapsto y]$) denote the environment that is equal to En except that **x** (**k**, **n**) is defined as y if $y \neq \bot$ and undefined otherwise. Function declarations are allowed to be mutually recursive.

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

Rule 2.2 (Variable (Rule [4])).

$$\frac{}{St, En \vdash \$s \Rightarrow (St, \mathbf{v}_{En}(s))}$$

Rule 2.3 (Built-in Functions (Rule [5])). These rules use $root()$,**true** and **false** formally defined in [11].

$$\frac{St, En \vdash e \Rightarrow (St', \langle s \rangle) \qquad \delta_{St'}(s) = n}{St, En \vdash \texttt{doc}(e) \Rightarrow (St', n)} \qquad \frac{St, En \vdash e \Rightarrow (St', \langle n \rangle) \qquad n \in \mathcal{V}^e \cup \mathcal{V}^a}{St, En \vdash \texttt{name}(e) \Rightarrow (St', \langle \nu_{St'}(n) \rangle)}$$

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

$$\frac{St, En \vdash e \Rightarrow (St', \langle x \rangle) \qquad x \in \mathcal{A} \qquad \mathit{AtValueToString}(x) = s}{St, En \vdash \texttt{string}(e) \Rightarrow (St', \langle s \rangle)}$$

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

$$\frac{St, En \vdash e \Rightarrow (St', \langle n \rangle) \qquad n \in V_{St'}}{St, En \vdash \texttt{root}(e) \Rightarrow (St', \langle \mathit{root}(n) \rangle)}$$

$$\frac{St, En \vdash e_1 \Rightarrow (St_1, \langle s_1 \rangle) \qquad s_1 \in \mathcal{S} \qquad St_1, En \vdash e_2 \Rightarrow (St_2, \langle s_2 \rangle) \qquad s_2 \in \mathcal{S}}{St, En \vdash \texttt{concat}(e_1, e_2) \Rightarrow (St_2, \langle s_1 \cdot s_2 \rangle)}$$

$$\frac{}{St, En \vdash \texttt{true}() \Rightarrow (St, \langle \mathbf{true} \rangle)} \qquad \frac{}{St, En \vdash \texttt{false}() \Rightarrow (St, \langle \mathbf{false} \rangle)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle b \rangle) \qquad b \in \mathcal{B}}{St, En \vdash \texttt{not}(e) \Rightarrow (St', \langle \neg b \rangle)} \qquad \frac{St, En \vdash e \Rightarrow (St', \langle x_1, \ldots, x_m \rangle)}{St, En \vdash \texttt{count}(e) \Rightarrow (St', \langle m \rangle)}$$

$$\frac{}{St, En \vdash \texttt{position}() \Rightarrow (St, \langle \mathbf{k}_{En} \rangle)} \qquad \frac{}{St, En \vdash \texttt{last}() \Rightarrow (St, \langle \mathbf{m}_{En} \rangle)}$$

Rule 2.4 (**If-expression (Rule [6])**). 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**. This is similar to the formal specification of `and` in XQuery, as seen in Section 1.2, where their are different inference rules for the different values of the operands of the `and` operator. Note that in each case only one of the branches is executed.

$$\frac{St, En \vdash e \Rightarrow (St', \langle \mathbf{true} \rangle) \qquad St', En \vdash e_1 \Rightarrow (St_1, v_1)}{St, En \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow (St_1, v_1)}$$

$$\frac{St, En \vdash e \Rightarrow (St', \langle \mathbf{false} \rangle) \qquad St', En \vdash e_2 \Rightarrow (St_2, v_2)}{St, En \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow (St_2, v_2)}$$

Rule 2.5 (**For-expression (Rule [7])**). 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.

$$\frac{\begin{array}{cc} St, En \vdash e \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) & St_0, En[\mathbf{v}(s) \mapsto x_1][\mathbf{v}(s') \mapsto 1] \vdash e' \Rightarrow (St_1, v_1) \\ \ldots & St_{m-1}, En[\mathbf{v}(s) \mapsto x_m][\mathbf{v}(s') \mapsto m] \vdash e' \Rightarrow (St_m, v_m) \end{array}}{St, En \vdash \texttt{for } \$s \texttt{ at } \$s' \texttt{ in } e \texttt{ return } e' \Rightarrow (St_m, v_1 \circ \ldots \circ v_m)}$$

Rule 2.6 (**Let-expression (Rule [8])**). We can see that this rule is simpler than its XQuery counterpart given in Section 1.2. But if we omit the typing in the XQuery rule we can see that the semantics correspond. As you can also so there are no error judgments in LiXQuery. LiXQuery does not raise errors, instaid the results are undefined.

$$\frac{St, En \vdash e \Rightarrow (St', v) \qquad St', En[\mathbf{v}(s) \mapsto v] \vdash e' \Rightarrow (St'', v')}{St, En \vdash \texttt{let}\ \$s := e\ \texttt{return}\ e' \Rightarrow (St'', v')}$$

Rule 2.7 (**Concatenation (Rule [9])**).

$$\frac{St, En \vdash e' \Rightarrow (St', v') \qquad St', En \vdash e'' \Rightarrow (St'', v'')}{St, En \vdash e',\ e'' \Rightarrow (St'', v' \circ v'')}$$

Rule 2.8 (**Boolean Operators (Rule [10])**). We can see some differences with the corresponding XQuery formal semantics inference rule as given in Section 1.2. That rule represents only one instance, while this rule represents them all at once. But it is clear that they are basically the same.

$$\frac{St, En \vdash e' \Rightarrow (St', \langle b' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle b'' \rangle) \qquad b', b'' \in \mathcal{B}}{St, En \vdash e'\ \texttt{and}\ e'' \Rightarrow (St'', \langle b' \wedge b'' \rangle) \qquad St, En \vdash e'\ \texttt{or}\ e'' \Rightarrow (St'', \langle b' \vee b'' \rangle)}$$

Rule 2.9 (**Atomic Value Comparisons (Rule [11])**).

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St', \langle x'_1, \ldots, x'_{m'} \rangle) \qquad x'_1, \ldots, x'_{m'} \in \mathcal{A} \qquad St', En \vdash e'' \Rightarrow (St'', \langle x''_1, \ldots, x''_{m''} \rangle) \\ x''_1, \ldots, x''_{m''} \in \mathcal{A} \qquad b_= \Leftrightarrow \exists_{1 \leq i \leq m', 1 \leq j \leq m''}(x'_i = x''_j) \qquad b_< \Leftrightarrow \exists_{1 \leq i \leq m', 1 \leq j \leq m''}(x'_i < x''_j) \end{array}}{St, En \vdash e' = e'' \Rightarrow (St'', \langle b_= \rangle) \qquad St, En \vdash e' < e'' \Rightarrow (St'', \langle b_< \rangle)}$$

Rule 2.10 (**Node Comparisons (Rule [12])**).

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St', \langle n' \rangle) \\ St', En \vdash e'' \Rightarrow (St'', \langle n'' \rangle) \qquad n', n'' \in \mathcal{V} \qquad b_{\texttt{is}} \Leftrightarrow (n' = n'') \qquad b_{\ll} \Leftrightarrow (n' \ll_{St''} n'') \end{array}}{St, En \vdash e'\ \texttt{is}\ e'' \Rightarrow (St'', \langle b_{\texttt{is}} \rangle) \qquad St, En \vdash e' << e'' \Rightarrow (St'', \langle b_{\ll} \rangle)}$$

Rule 2.11 (**Additions (Rule [13])**).

$$\frac{St, En \vdash e' \Rightarrow (St', \langle d' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle d'' \rangle) \qquad d', d'' \in \mathcal{I}}{St, En \vdash e' + e'' \Rightarrow (St'', \langle d' + d'' \rangle) \qquad St, En \vdash e' - e'' \Rightarrow (St'', \langle d' - d'' \rangle)}$$

Rule 2.12 (**Multiplications (Rule [14])**).

$$\frac{St, En \vdash e' \Rightarrow (St', \langle d' \rangle) \qquad St', En \vdash e'' \Rightarrow (St'', \langle d'' \rangle) \qquad d', d'' \in \mathcal{I}}{St, En \vdash e' * e'' \Rightarrow (St'', \langle d' \times d'' \rangle) \qquad St, En \vdash e'\ \texttt{idiv}\ e'' \Rightarrow (St'', \langle d'/d'' \rangle)}$$

Rule 2.13 (**Union (Rule [15])**).

$$\frac{St, En \vdash e' \Rightarrow (St', v') \qquad St', En \vdash e'' \Rightarrow (St'', v'') \qquad v', v'' \in \mathcal{V}^*}{St, En \vdash e' \mid e'' \Rightarrow (St'', \mathbf{Ord}_{St''}(\mathbf{Set}(v') \cup \mathbf{Set}(v'')))}$$

Rule 2.14 (Axis Steps (Rule [16])). The semantics of a step consisting of an element name s is that all element children of the context node (indicated in the envorment 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.

$$\frac{\mathbf{x}_{En} \textit{ is defined}}{St, En \vdash \texttt{.} \Rightarrow (St, \langle \mathbf{x}_{En} \rangle)} \qquad \frac{(n, \mathbf{x}_{En}) \in E_{St}}{St, En \vdash \texttt{..} \Rightarrow (St, \langle n \rangle)} \qquad \frac{\nexists n (n, \mathbf{x}_{En}) \in E_{St}}{St, En \vdash \texttt{..} \Rightarrow (St, \langle\rangle)}$$

$$\frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^e \wedge \nu_{St}(n) = s\}}{St, En \vdash s \Rightarrow (St, \mathbf{Ord}_{St}(W))}$$

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

$$\frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^e\}}{St, En \vdash * \Rightarrow (St, \mathbf{Ord}_{St}(W))} \qquad \frac{W = \{n | (\mathbf{x}_{En}, n) \in E_{St} \wedge n \in \mathcal{V}^a\}}{St, En \vdash \texttt{@}* \Rightarrow (St, \mathbf{Ord}_{St}(W))}$$

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

Rule 2.15 (Filter-expression (Rule [17])). The semantics of e' `[` e'' `]` is that first e' is evaluated, then for each item in the result of e' the expression e'' is evaluated with $\mathbf{x}$ bound to this item, $\mathbf{k}$ to the position of the item in the result of e' and $\mathbf{m}$ to the number of items in the result of e'. The result of e'' is a boolean or an integer, in which case it is converted to **true** if this integer is equal to $\mathbf{k}$ and to **false** otherwise. Finally, the result is the subsequence of the result of e' that contains exactly all items for which e'' evaluated to **true**.

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) \qquad En' = En[\mathbf{m} \mapsto m] \\ St_0, En'[\mathbf{x} \mapsto x_1][\mathbf{k} \mapsto 1] \vdash e'' \Rightarrow (St_1, \langle x'_1 \rangle) \qquad \ldots \qquad St_{m-1}, En'[\mathbf{x} \mapsto x_m][\mathbf{k} \mapsto m] \vdash e'' \Rightarrow (St_m, \langle x'_m \rangle) \\ x'_1, \ldots, x'_m \in \mathcal{B} \cup \mathcal{I} \qquad v = \langle x_i | (x'_i \in \mathcal{I} \wedge x'_i = i) \vee (x'_i \in \mathcal{B} \wedge x'_i) \rangle \end{array}}{St, En \vdash e' \texttt{ [} e'' \texttt{]} \Rightarrow (St_m, v)}$$

Rule 2.16 (Path Expression (Rule [18])). The semantics of (e' `/` e'') is as follows. First e' is evaluated. Then for each item in its result we bind in the environment $\mathbf{x}$ to this item, $\mathbf{k}$ to the position of $\mathbf{x}$ in the result of e', and $\mathbf{m}$ to the number of items in the result of e', and with this environment we evaluate e''. 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'' contain only nodes.

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_0, \langle x_1, \ldots, x_m \rangle) \qquad En' = En[\mathbf{m} \mapsto m] \qquad St_0, En'[\mathbf{x} \mapsto x_1][\mathbf{k} \mapsto 1] \vdash e'' \Rightarrow (St_1, v_1) \\ \ldots \qquad St_{m-1}, En'[\mathbf{x} \mapsto x_m][\mathbf{k} \mapsto m] \vdash e'' \Rightarrow (St_m, v_m) \qquad v_1, \ldots, v_m \in \mathcal{V}^* \end{array}}{St, En \vdash e' \texttt{ / } e'' \Rightarrow (St_m, \mathbf{Ord}_{St_m}(\cup_{1 \leq i \leq m} \mathbf{Set}(v_i)))}$$

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

Rule 2.17 (**Literal (Rule [19])**)**.** The result of a literal is simply a sequence with one element, viz., the atomic value the literal represents.

Rule 2.18 (**Empty Sequence (Rule [20])**)**.**

$$\frac{}{St, En \vdash () \Rightarrow (St, \langle\rangle)}$$

Rule 2.19 (**Typeswitch-expression (Rules [22] and [23])**)**.** Let $[\![\texttt{xs : boolean}]\!] = \mathcal{B}$, $[\![\texttt{xs : integer}]\!] = \mathcal{I}$, $[\![\texttt{xs : string}]\!] = \mathcal{S}$, $[\![\texttt{document-node()}]\!] = \mathcal{V}^d$, $[\![\texttt{attribute()}]\!] = \mathcal{V}^a$, $[\![\texttt{text()}]\!] = \mathcal{V}^t$ and $[\![\texttt{element()}]\!] = \mathcal{V}^e$.

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

Rule 2.20 (**Function Call (Rule [24])**)**.** The semantics of $f(e_1, \ldots, e_m)$ is that $e_1, \ldots, e_m$ are consecutively evaluated, and then the expression $\mathbf{b}(f)$ is evaluated with the variable names of $\mathbf{a}(f)$ bound to the results of $e_1, \ldots, e_m$.

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

2.3 Construction in LiXQuery

As the goal of our work is to examine the expressive power of the node construction using LiXQuery it is important to understand how the construction is formally defined for LiXQuery. In LiXQuery you can write all the XQuery constructors using the computed constructor syntax, except for the constructors of comment nodes, processing instructions and namespace nodes.

Essential to the definitions of the semantics of the constructors is the notion of *deep equality*. We introduced the notion for XQuery in Section 1.1.3. In [11] we can find the following formal definition of deep equality in LiXQuery, which defines what it means for two nodes in an XML store to represent the same XML fragment using the notion of isomorphic trees.

Definition 2.21 (LiXQuery Deep Equal)**.** Given the XML store $St = (V, E, \ll, \nu, \sigma, \delta)$ and two nodes n_1 and n_2 in St. n_1 and n_2 are said to be *deep equal*, denoted as $\mathbf{DpEq}_{St}(n_1, n_2)$, if n_1 and n_2 refer to two isomorphic trees, i.e., there is a one-to-one function $h : C_{n_1} \to C_{n_2}$ with $C_{n_i} = \{n | (n_i, n) \in E^*\}$ for $i = 1, 2$, such that for each $n, n' \in C_{n_1}$ it holds that (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 $(h(n), h(n')) \in E$ and (5) if $n, n' \notin \mathcal{V}^a$ then $n \ll n'$ iff $h(n) \ll h(n')$.

This definition essentially says that deep equal nodes must have the same type of children at the same positions (except for attributes) with the same names and same values. The only thing that can differ between the nodes in the subtree of n_1 and that of n_2 is the fact that they can have a different node identity.

With this notion the semantics of the element constructor ($\texttt{element}\{e'\}\{e''\}$) is the defined as follows. First e' is evaluated and assumed to result in a single legal element name ($St, En \vdash e' \Rightarrow (St_1, \langle s \rangle)$ and $s \in \mathcal{N}$). Then e'' is evaluated ($St_1, En \vdash e'' \Rightarrow (St_2, \langle n_1, \ldots, n_m \rangle)$) and for the result ($n_1, \ldots, n_m \in \mathcal{V}$) we create a new store (St_3) that contains the new element ($r \in \mathcal{V}^e$) with the result of e' as its name ($\nu_{St_3}(r) = s$) and with contents that are deep-equivalent with the result of e'' if we compare them item by item ($\mathbf{Ord}_{St_3}(\{n' | (r, n') \in E_{St_3}\}) = \langle n'_1, \ldots, n'_m \rangle \mathbf{DpEq}_{St_4}(n_1, n'_1) \ldots \mathbf{DpEq}_{St_4}(n_m, n'_m)$). Finally we add St_3 to the original store ($St_4 = St_2 \cup St_3$) and return the newly created element node. During this the document order of the nodes in the original store St_2 remains unchanged in the new extended store St_4 ($\forall\, n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n'))$). This semantics is represented as a LiXQuery inference rule as follows:

Rule 2.22 (Element Constructor (Rule [21])).

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

Let us now compare this to the semantics of the element construction in XQuery as described in Subsection 1.2.1. We omitted namespaces and typing in our discussion of the XQuery semantics, but the LiXQuery semantics omits these too. It is straightfore-ward to see that the basic evaluation syntax of the name expression corresponds. But as we stated in Subsection 1.2.1 much of the semantics there is informally captured in the `fs:item-sequence-to-node-sequence()` function call. First of all the semantics of LiX-Query differs as it does not perform implicit 'casts' of atomic values with the evaluation of the context expressions. This means that the content expression of the element construc-tor may only evaluate to nodes (as stated clearly in the semantic rule). It is however not fundamental that we should be able to have atomic values here. For example, in XQuery, a string literal in the context expression would be converted implicitly to a new text node. In LiXQuery this semantics must be stated explicitly using a text node constructor (which now in LiXQuery consequently only allows for atomic string values to be in its context expression, as can be seen in the following Rule 2.24). The LiXQuery constructor seman-tics also misses the merging of adjacent text nodes, and document nodes are no longer allowed in the context sequence, as there is no implicit conversion of a document node to it's children. This fact that no document nodes are allowed in the context sequence and the fact that attribute nodes must appear together and first in the sequence, is not apparent in the LiXQuery semantic rule. But these facts follows from the semantics of the store, which does not allow document nodes to be the children of element nodes, or attributes in the wrong place. It is important to note however that these restrictions of LiXQuery with respect to XQuery are not essential. All these implicit semantics can still be achieved explicitly in LiXQuery as demonstrated in Example refex:constrestr.

Example 2.1 Restricted Element Constructor

```
element {"new"}{"just a ","content string",doc("table.xml")}
```

must be written in LiXQuery as

```
element {"new"}{text{"just a content string"},doc("table.xml")/*}
```

Taking these restrictions in to account the semantics that remain to be compared are the essential copying semantics. The XQuery semantics states that for each node returned by an enclosed expression, a new copy is made of the given node and all nodes that have the given node as an ancestor. Collectively each of these ‘copied’ nodes has a new node identity while the parent, children, and attributes properties of the copied nodes are set so as to preserve their inter-node relationships. As we can see in the LiXQuery semantics this corresponds to the new nodes $n'_1, \ldots, n'_m$ and their deep eqality to the nodes in the context sequence (result sequence of expression e'') as well as the statement that says the document order must correspond $(\forall\ n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n')))$.

The XQuery semantics also states that for the topmost nodes (the node directly returned by the enclosed expression), the parent property is set to the newly-constructed element node. This corresponds to the statement $\mathbf{Ord}_{St_3}(\{n'|(r, n') \in E_{St_3}\}) = \langle n'_1, \ldots, n'_m\rangle$ in the LiXQuery semantics.

Essentially we can conclude that taking into account the restrictions, the semantics correspond.

With the attribute constructor (`attribute`$\{e'\}\{e''\}$) the expression e' is evaluated first and again assumed to result in a single legal element name $(St, En \vdash e' \Rightarrow (St_1, \langle s\rangle)$ and $s \in \mathcal{N})$. Then e'' is evaluated and assumed to result in a single legal string value $(St_1, En \vdash e'' \Rightarrow (St_2, \langle s'\rangle)$ and $s' \in \mathcal{S})$. We then create a new store (St_3) which contains the new attribute $(V_{St_3} = \{r\}$ and $r \in \mathcal{V}^a)$ with the result of e' as its name $(\nu_{St_3}(r) = s)$ and the result of e'' as its value $(\sigma_{St_3}(r) = s')$. Finally we add St_3 to the original store $(St_4 = St_2 \cup St_3)$ and return the newly created attribute node. Again during this the document order of the original store remains unchanged. Written as an inference rule:

Rule 2.23 (Attribute Constructor (Rule [21])).

$$\frac{\begin{array}{c} St, En \vdash e' \Rightarrow (St_1, \langle s\rangle) \qquad s \in \mathcal{N} \qquad St_1, En \vdash e'' \Rightarrow (St_2, \langle s'\rangle) \qquad s' \in \mathcal{S} \qquad St_4 = St_2 \cup St_3 \\ V_{St_3} = \{r\} \qquad r \in \mathcal{V}^a \qquad \nu_{St_3}(r) = s \qquad \sigma_{St_3}(r) = s' \qquad \forall\ n, n' \in \mathcal{V}((n \ll_{St_2} n') \Rightarrow (n \ll_{St_4} n')) \end{array}}{St, En \vdash \texttt{attribute}\{e'\}\{e''\} \Rightarrow (St_4, \langle r\rangle)}$$

The text node constructor ($\texttt{text}\{e\}$) is similar to the attribute constructor. The only expression e is evaluated and assumed to result in a stringvalue different from the empty string ($St, En \vdash e \Rightarrow (St_1, \langle s \rangle)$ and $s \in \mathcal{S} - \{``"\}$). A new store ($St_2$) is created containing the text node ($V_{St_2} = \{r\}$ and $r \in \mathcal{V}^t$) that has the result of expression e as its value ($\sigma_{St_2}(r) = s$). Again this store is added to the original store while preserving the document order of the original store. As an inference rule we write:

Rule 2.24 (**Text Node Constructor (Rule [21])**)**.**

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

The document constructor ($\texttt{document}\{e\}$) also evaluates just one expression but this time the expression (e) must evaluate to a *single node* ($St, En \vdash e \Rightarrow (St_1, \langle n_1 \rangle)$ and $n_1 \in \mathcal{V}^e$). A new store (St_2) is created containing a document node ($r \in \mathcal{V}^d$), of which all nodes in St_2 are descendants ($n \in V_{St_2} \Rightarrow (r, n) \in E^*_{St_2}$), that has a node ($n_2$) deep equal to the result of expression e ($\mathbf{DpEq}_{St_3}(n_1, n_2)$) as its only child node ($(r, n_2) \in E_{St_2}$). In the form of an inference rule we write:

Rule 2.25 (**Document Constructor (Rule [21])**)**.**

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

2.4 Types of Result Nodes

Based on this knowledge of node construction and the formal expression evaluation in LiXQuery and XQuery, we can now formally define the different types of nodes that can occur in the result sequence of an evaluated LiXQuery expression. These types of nodes will allow us to later define specific properties of expressions.

Definition 2.26 (Result node types)**.** We call a node in the result sequence of an expression an **original node** if it is identical to some node in the initial XML store of the expression. We call an it a **new node** if it is newly constructed in the expression (it was not present in the initial XML store).
We call it a **deep-equal node** if it is deep-equal to some node in the initial XML store of the expression.
We call it a **copied node** if it is a copy of some node in the initial XML store of the expression.

In Figure 2.2 we see the relations between these types of nodes. It is straightforward to see that the copied and original nodes are also deep-equal. The copied nodes are also new, as they have a new node identity. But it is important to know that a portion of the new

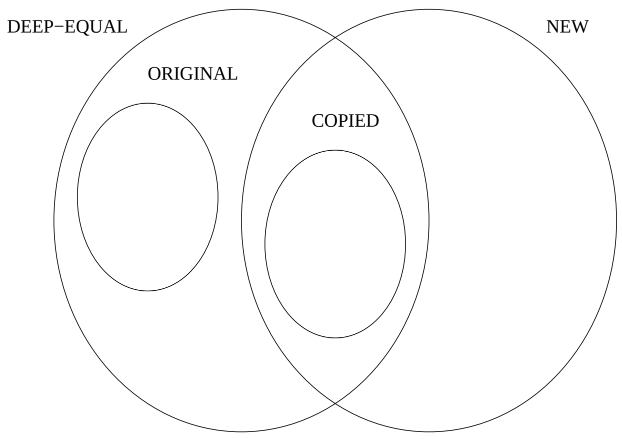

Figure 2.2: Result node types

nodes is also deep equal but was not created as a copy. These nodes are in fact *coincidently* deep-equal to an original node. This coincidental deep equal nodes are made up of new nodes and it is therefore possible that they are only deep equal in some evaluations of an expression, as we will expand on later.

Chapter 3

Blixem, A LiXQuery Engine

In this Chapter we will introduce *Blixem*[1], an engine to evaluate LiXQuery queries. We discuss it here because it is the reference implementation for LiXQuey and can therefore provide some insight in the LiXQuery semantics as it tries to follow them as close as possible. The Blixem project is being developed by Jeroen Avonts, Pieter Wellens and myself.

3.1 Goal and approach

The main goal of the Blixem project is to provide a reference implementation for LiXQuery. This implementation will allow us to easily compare LiXQuery with XQuery, by comparing the results of Blixem to that of a standards compliant XQuery implementation. It also provides us with a basis for running test based on hypotheses about LiXQuery or subsets of LiXQuery.

We opted to create the LiXQuery eginine on top of an existing XML DOM implementation. We chose the Apache Xerces XML DOM parser[2]. It is able to parse documents written according to the XML 1.1 Recommendation [2] and handles namespaces according to the XML Namespaces 1.1 Recommendation [3], and will correctly serialize XML 1.1 documents. We used the Java version of Xerces as we have chosen this to be the implementation language for our LiXQuery egine. Java was chosen because it is cross-platform and easy to develop. The LiXQuery parser was written using the JavaCC[3] parser generator together with the JJTree tool to generate an abstract syntax tree. The abstract syntax of LiXQuery as given in [11] misses extra brackets and precedence rules to be disambiguated. To be able to generate a parser we therefore devised a formal BNF syntax for LiXQuery where we added the missing information based on the XQuery syntax [4]. This BNF can be found in Appendix A. Because the goal is to provide a reference implementation of LiXQuery, the correctness of the implementation is essential. First this is achieved by having

[1]http://www.adrem.ua.ac.be/ blixem
[2]http://xml.apache.org/xerces2-j/
[3]https://javacc.dev.java.net/

the code very close to the formal semantics of LiXQuery, as you will see in Section 3.2. Second is to intensively test the engine. For this we devised a data-oriented testing framework on top of the JUnit testing tool[4]. An XML file is used to specify LiXQuery expressions and their results. These results can be parse trees for parser testing or result sequences for engine testing. Our framework then reads these XML files and creates the associated JUnit tests which we are then able to run. All the programming is done using the eclipe java IDE[5] providing the integration with JUnit and refactoring abilities.

3.2 Design

The generated parser generates an Abstract Syntax Tree (AST) of the parsed LiXQuery expression. We then make use of a *treewalker* class to walk the nodes of the tree and evaluate the function associated with the specific AST node. We are using a treewalker instead of an evaluation method in the AST nodes because of the generated nature of these nodes. As the BNF syntax changed quite a bit during the development so did the Abstract Syntax Tree associated with it, and the use of a treewalker lessened the impact of these changes.

In Figure 3.1 we provide a simple UML diagram of the basic classes in our implementation. As you can see the different components of the LiXQuery formal semantics are present as classes, of which the subcomponents adhere to the LiXQuery naming conventions. We have an `Environment` class which contains the `aMap`, `bMap`, `vMap`, `k`, `s` and `x` mappings, and an `XMLStore` which contains a `deltaMap` and a `docordMap`. We can see that the `XMLStore` makes use of `XMLDocuments` and `XMLFragments`, and that the `Environment` uses `Items`. An Item is an abstract class which has the subclass `NodeItem`, which is a concrete class of which the underlying implantation is based on the XML DOM provided by Xerces, and the subclass `AtomicValue`, which is an abstract class which has the `IntegerValue`, `BooleanValue` and `StringValue` subclasses (not shown on the diagram).

To understand how our code closely relates to the formal semantic of LiXQuery we will explain the implemtation of the for-expression an an example. The semantics of the for-expression is provided as Rule 2.5 in Subsection 2.2.1 on page 17. The associated code is given in Example 3.1. The implented semantics is slightly extended as it accounts for the possibility of ommitting '`at $`s''. This is achieved by computing an offset based on the number of childnodes of the `for` AST node. As you can see we evalute expression e as described in the semantics (`ItemSequence resE = eval(root.jjtGetChild(1 + offset), env)`). After this we clone the Environment. This is necessary as the semantic rules prescribes that all the expressions in the for loop must be evaluated in the same environment. We then loop over the result sequence, each time adding the current item as the specified variable ($\$s$, where s is `root.jjtGetChild(0).getText()`) and optionally adding the specified postion variable. With this environment the second expression e' is evaluated (`ItemSequence resEP = eval(root.jjtGetChild(2 + offset), envP)`) as specified in

[4]http://www.junit.org/

[5]http://www.eclipse.org/

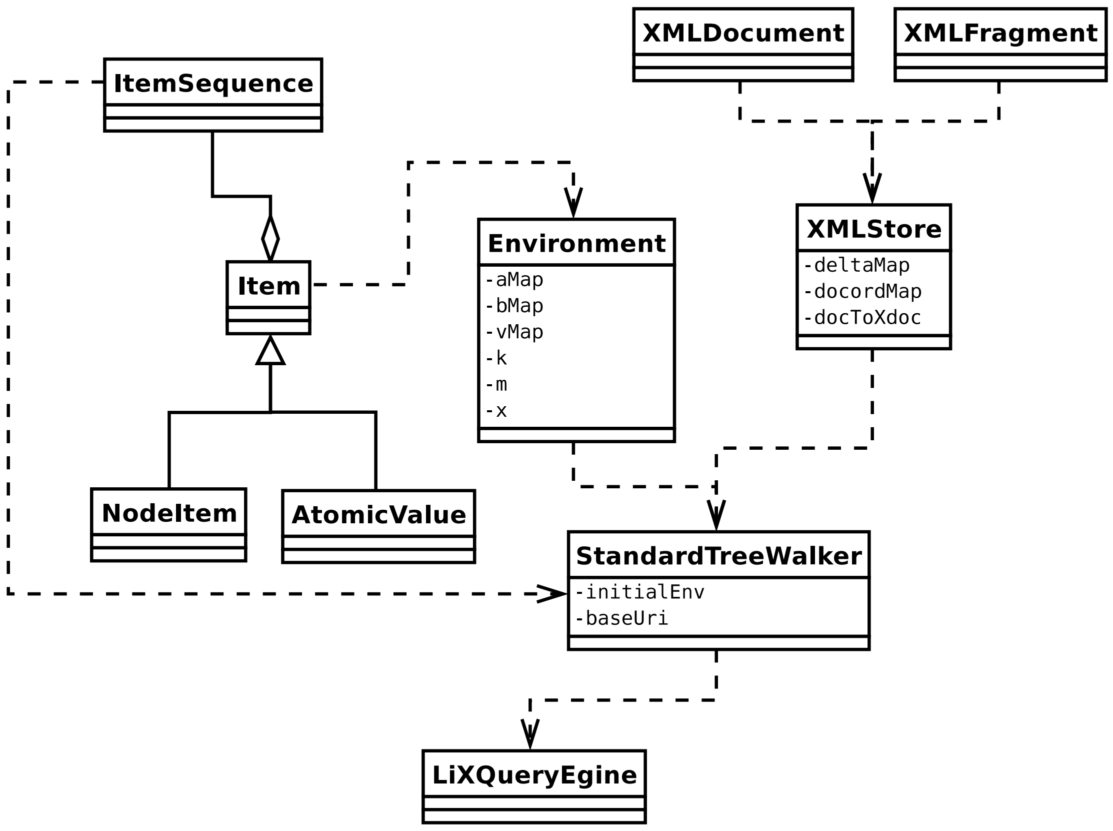

Figure 3.1: Blixem components

the semantic rule $(St_{m-1}, En[\mathbf{v}(s) \mapsto x_m][\mathbf{v}(s') \mapsto m] \vdash e' \Rightarrow (St_m, v_m))$. Finally result is added to the result sequence (`result.addNodeList(resEP)`), which evetually is returned as specified in the semantic rule $(v_1 \circ \ldots \circ v_m)$.

Example 3.1 For-expression implementation

```
private ItemSequence forExpr(blixem.parser.lixquery.Node root,
    Environment env)  throws EngineException,
      InternalErrorException {
  ItemSequence result = new ItemSequence();
  int offset = (root.jjtGetNumChildren() == 3 ? 0 : 1);
  ItemSequence resE = eval(root.jjtGetChild(1 + offset), env);
  Environment envP = env.cloneEnv();
  for (int i = 0; i < resE.getLength(); i++) {
    Item node = resE.item(i);
    envP.addVariable(root.jjtGetChild(0).getText(),
      new ItemSequence(node));
    if (offset == 1) {
      envP.addVariable((root.jjtGetChild(1)).getText(),
        new ItemSequence(new IntegerValue(i + 1)));
    }
    ItemSequence resEP = eval(root.jjtGetChild(2 + offset), envP);
    result.addNodeList(resEP);
  }
  return result;
}
```

3.3 Construction in Blixem

Construction will be supported in Blixem, but work on this is still continuing at this moment of writing. The Xerces XML DOM has no built-in support for XML fragments which are essential to support construction. Therefore we created our own `XMLFragment` class and `XMLDocument` class using the Xerces XML DOM documents internally. The `XMLDocument` class is a straightforward wrapper. With the `XMLFragment` we simply simulate the fragment by using an Xerces XML DOM document as a base, but taking care of fragment specific properties. These include making sure that the fragments don't have a parent node and that during element and document construction the selected nodes in the body of the constructor are being copied.

As this part of the engine is still in heavy development and testing I will not go into it any further and will return to the main topic of this work, the study of the expressive power of the node construction.

Chapter 4

Expressive Power of the Node Construction

In this chapter we show that for certain expression that can contain node construction there exists an expression without node construction yielding the same result. This result gives us an indication on the expressive power of the node construction in LiXQuery.

Section 4.1 contains the definition of the theorem for the elimination of node construction in expressions that have results that do not contain newly constructed nodes. Section 4.2 gives an outline of the simulation that will be used to prove our theorem. Section 4.3 and 4.4 define this simulation. In Section **??** we explain how this simulation can be used to create a constructor-free expression. Finally in Section 4.5 we give an illustrative example.

4.1 Eliminating Node Construction

The elimination of construction in an expression basically means that we will have to find an other expression that simulates the expression but which does not use construction. It is clear that an expression that returns new nodes cannot be simulated by an expression without constructors. This is why we introduce the notion of *node-conservative expression.*

Definition 4.1. A *node-conservative expression* (NCE) is an expression $e \in LQE$ such that for all stores St and environments Env it holds that if $St, Env \vdash e \Rightarrow St', v$ then all nodes in v are nodes in St.

Another restriction we make is that we only consider deterministic expressions. Node creation is a source of nondeterminism in LiXQuery (and XQuery) because the fragment that is created by a constructor is placed at an arbitrary position in document order between the already existing trees in the store. Since node construction is the only source of non-determinism in LiXQuery, it is clear that we cannot simulate that there are many possible results without it. This is however not a fundamental feature of XQuery so we ignore non-deterministic expressions.

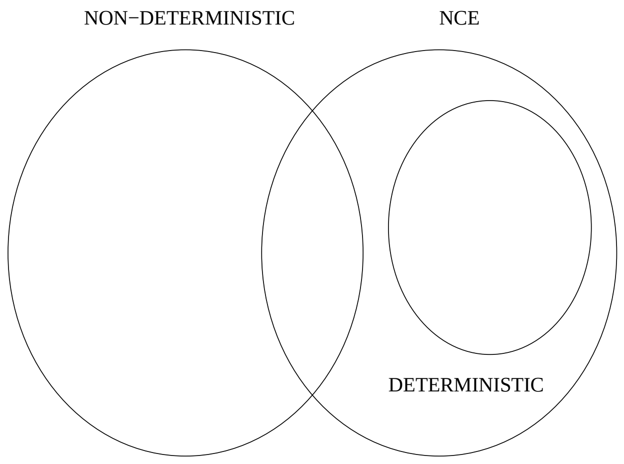

Figure 4.1: NCE and determinism

Definition 4.2. An expression $e \in LQE$ is said to be deterministic if for every store St and environment Env it holds that if $St, Env \vdash e \Rightarrow St', v$ and $St, Env \vdash e \Rightarrow St'', w$ then $v = w$.

Note that this is a very strict definition of determinism. These deterministic expressions are in fact a subset of the node-conservative expressions as depicted in Figure 4.1. Note that node-conservative expressions are possibly non-deterministic, an example of a non-deterministic node-conservative expression is given in Example 4.1. In Example 0.1 we considered a join and a projection of two XML documents in LiXQuery. This expression is an example of a deterministic node-conservative expression.

Example 4.1 A Non-deterministic Expression

```
declare function eps:nondeterm() {
  let $x := <a/>
  let $y := <a/>
  return
    if $x << $y then true() else false()
};
if (eps:nondeterm()) then
  doc("table.xml")/table/row[1]
else
  doc("table2.xml")/table/row[2]
```

Using the input documents as defined in Example 0.1.

We could have allowed multiple results that were equivalent up to isomorphism over the nodes, but this would make things unnecessarily complex.

Next to restricting the types of expressions we consider, we also allow a simulation to differ in its semantics from the the original in two ways. The first is that a simulation may have a defined result where the original does not. Note that we still require that whenever an expression has a defined result then the simulation has the same defined result, but not necessarily the reverse. We conjecture that the theorem also holds when we also require the reverse but proving this would add a lot of overhead to this work without adding much extra insight in the expressive power of node construction.

The second way in which the semantics of a simulation differs from that of the original is that resulting stores are only the same up to *garbage collection*, i.e., after removing the trees that are not reachable by the LiXQuery δ function (used in the `fn:doc()` function) or contain nodes from the result sequence. If we denote the store that results from garbage collection on a store St and a result sequence v as $[St]_v$ then this leads to the following definition:

Definition 4.3. Given two expression $e, e' \in LQE$ we say that e' is a *simulation* of e if for all stores St and environments Env with undefined x, k and m it holds that if $St, Env \vdash St', v$ then there exists a store St'' such that $St, Env \vdash St'', v$ and $[St'']_v = [St']_v$.

With these definitions to define the following theorem, wich is the main result of this work:

Theorem 4.4. *For every deterministic node-conservative*[1] *expression e there exists a simulation $e' \in LQE$ that does not contain constructors.*

4.2 Outline of the simulation

Our goal is to create a construction free simulation of an expression. To simulate construction we will need to simulate the store, because it is there that the information concerning the newly constructed nodes will reside. In short, the simulation performs the following steps:

1. We use a few special variables in the environment to encode a part of the store. This part will contain the newly created nodes but also parts of the old store that are retrieved with the `doc()` function;

2. Whenever a `doc()` call occurs in the original expression, the simulation will add the encoding of the document tree to the simulated store on the condition that it is not already there;

3. Accessing nodes in the store is simulated by accessing the encoded store;

4. Nodes are simulated by node identifiers which are numbers that refer to the encoded nodes in the store;

[1]Since every deterministic expression is also node-conservative we can strictly speaking drop the second requirement.

5. In order to be able to distinguish encoded atomic values from node identifiers within sequences, we let the normal atomic values be preceded by a `0` and the node identifiers by a `1`. Note that this means that in the simulation, a sequence will be twice as long and every item that was at position i will now be at position $2i$;

6. Finally, the simulation replaces the node identifiers with the corresponding nodes from the store. If the original expression is indeed a deterministic node-conservative expression, the result – and thus also the result of the simulation – will contain no newly constructed nodes. Consequently, this last step is always possible if the original expression is node-conservative.

The transformation of an expression to its simulation, is expressed by a transformation function. A transformation function is a function $\epsilon : LQE \to LQE$. The commuting diagram in Figure 4.2 illustrates what should hold for such a transformation function ϵ for it to be correct. We show this by induction on the subexpressions e'' of an expression e.

$$\begin{tikzcd}
(St, Env) \arrow[r, "\tau"] \arrow[d, "e''"'] & (\widehat{St}, \widehat{Env}) \arrow[d, "\epsilon(e'')"'] \\
(St', v) \arrow[r, "\tau'"] & (\widehat{St}, \widehat{v})
\end{tikzcd}$$

Figure 4.2: Relations between the several components in the translation.

On the left-hand side we see that starting from a store St and an environment Env, the evaluation of the expression e'', which may add new nodes to St, will result in a new store $St' \supseteq St$ and a result v. On the right-hand side we see that starting from a store $\widehat{St}$ and an environment $\widehat{Env}$, the evaluation of the transformed constructor-free expression $\epsilon(e'')$, which will not add new nodes to $\widehat{St}$, will result in the same store $\widehat{St}$ and a result $\widehat{v}$.

At the top of the diagram we see the encoding τ which encodes a store $St_{\widehat{Env}} \subseteq St$ into sequences of atomic values that are bound to special variables in the environment $\widehat{Env}$. Moreover, τ replaces the values of all variables in Env with sequences of atomic values and the bodies of all functions are transformed by ϵ to constructor-free expressions. At the bottom of the diagram we see the encoding τ' which encodes a store $St_{\widehat{v}} \subseteq St'$ and the value v as a sequence of atomic values $\widehat{v}$.

When we use this schema to show by induction that we can correctly transform an expression e to a constructor-free expression $\epsilon(e)$ it will hold for the evaluation of the subexpression e'' that $\widehat{St}$ is the store against which e is evaluated. Moreover, if during the evaluation of e nodes where created before the evaluation of e'' then (1) these nodes have been added to St and (2) in the evaluation of $\epsilon(e)$ they were added to the encoded store in $\widehat{Env}$. So it will hold that $St = \widehat{St} \cup St_{\widehat{Env}}$. Obviously it has to be shown by induction that this remains true after the evaluation of e'' so it has to be shown that $St' = \widehat{St} \cup St_{\widehat{v}}$. An overview of all these relationships between the involved stores is illustrated in Figure 4.3.

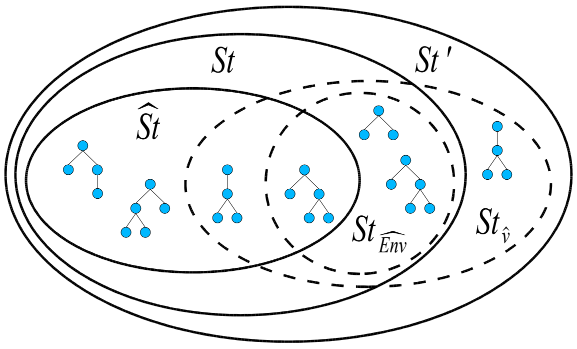

Figure 4.3: The stores $\widehat{St}$, $St_{\widehat{Env}}$, St, $St_{\hat{v}}$ and St'

4.3 Encoding the Store and Environment

Before we describe how to transform LiXQuery expressions into their constructor-less simulations, we first have to look into the encodings of the store and environment based on their formal semantics.

We first describe how to encode a store in sequences of atomic values. We will define this given an injective function $id : V \to \mathbb{N}$ that provides the unique node identifier for each node and which will be used to represent the nodes in the encoding.

Definition 4.5. Given an XML store $St = (V, E, \ll, \nu, \sigma, \delta)$ and an injective function $id : V \to \mathbb{N}$ then we call a tuple of XML values $(\hat{V}, \hat{E}, \hat{\delta})$ a *store encoding of St under id* if

– $\hat{V} = \langle id(v_1), t_1, n_1, s_1\rangle \circ \ldots \circ \langle id(v_k), t_k, n_k, x_k\rangle$
where (1) $\{v_1, \ldots, v_k\} = V$, (2) $v_1 \ll \ldots \ll v_k$, (3) t_i equals `"text"`, `"doc"`, `"attr"` or `"elem"` if v_i is a text node, a document node, an attribute node or an element node, respectively, (4) n_k is $\nu(v_k)$ if it is defined and `""` otherwise, and (5) s_k is $\sigma(v_k)$ if it is defined and `""` otherwise,

– $\hat{E} = \langle id(v_1), id(v'_1)\rangle \circ \ldots \circ \langle id(v_m), id(v'_m)\rangle$
where $\{(v_1, v'_1), \ldots, (v_m, v'_m)\} = E$,

– $\hat{\delta} = \langle s_1, id(v_1)\rangle \circ \ldots \circ \langle s_p, id(v_p)\rangle$
where $\delta = \{(s_1, v_1), \ldots, (s_p, v_p)\}$.

Note that a store encoding is not uniquely determined given St and id because we can choose the order in $\hat{E}$ and $\hat{\delta}$.

We have to encode sequences of atomic values and nodes as sequences of atomic values only. When we directly replace each node v with $id(v)$ we cannot always tell if a number represents itself or encodes a node identifier. Therefore we let atomic values that encode themselves be preceded by `0` and atomic values that are node identifiers be preceded by `1`. For illustration consider the examples in Example 4.2.

Example 4.2 Encoded Values

Given a function $id = \{(v_1, 5), (v_2, 3)\}$:

value	*value encoding*
$\langle 5 \rangle$	$\langle 0, 5 \rangle$
$\langle v_1 \rangle$	$\langle 1, 5 \rangle$
$\langle 5, v_1, \texttt{"string"}, v_2 \rangle$	$\langle 0, 5, 1, 5, 0, \texttt{"string"}, 1, 3 \rangle$

Definition 4.6. Given an XML value $v = \langle x_1, \ldots, x_k \rangle$ over a store $St = (V, E, \lll, \nu, \sigma, \delta)$ and an injective function $id : V \to \mathbb{N}$, we call an XML value $\tilde{v}$ the *value encoding of v under id* if $\tilde{v} = \langle m_1, \hat{x}_1 \rangle \circ \ldots \circ \langle m_k, \hat{x}_k \rangle$ where $m_i = 1$ and $\hat{x}_i = id(x_k)$ if x_k is a node and $m_i = 0$ and $\hat{x}_i = x_i$ otherwise.

Note that the encoding of value v is written as $\tilde{v}$ and not as $\hat{v}$ to distinguish it from the $\hat{v}$ in the commuting diagram in Figure 4.2 which encodes both a store and a value.

We now proceed with formalizing the the τ relationship introduced in Figure 4.2. Recall that the relations in this diagram hold by induction on the subexpressions e'' of a simulated expression e. The resulting store $\widehat{St}$ is the store against which e is evaluated, because all nodes that are created by e'' are in $\epsilon(e'')$ encoded in $\widehat{Env}$. We will refer to the part of St encoded in $\widehat{Env}$ as $St_{\widehat{Env}}$. Since $St_{\widehat{Env}}$ describes the part of St that is retrieved or created by preceding evaluations it holds that $St = \widehat{St} \cup St_{\widehat{Env}}$ where $\widehat{St} \cap St_{\widehat{Env}}$ contains the documents that were retrieved with the `doc()` function before e'' was evaluated (see Figure 4.3).

Definition 4.7. Given a store $St = (V, E, \lll, \nu, \sigma, \delta)$, an environment $Env = (a, b, v, x, k, m)$ over this store and a transformation function ϵ we call a pair $(\widehat{St}, \widehat{Env})$ with store $\widehat{St}$ and environment $\widehat{Env} = (\hat{a}, \hat{b}, \hat{v}, \hat{x}, \hat{k}, \hat{m})$ a *store-environment encoding of St and Env under tr* if there is a store $St_{\widehat{Env}}$ and an injective function $id : V_{\widehat{Env}} \to \mathbb{N}$ such that

- $St = \widehat{St} \cup St_{\widehat{Env}}$,
- all nodes in values of variables in Env are in $St_{\widehat{Env}}$
- $\hat{a} = a$,
- $\hat{b} = \{(s, tr(y)) | (s, y) \in b\}$,
- in $\hat{v}$ (1) all variable names s bound by v are bound to the value encoding of $v(s)$ under id, (2) the variables `tau:E`, `tau:V` and `tau:delta` contain $\hat{V}$, $\hat{E}$ and $\hat{\delta}$, respectively, where $(\hat{V}, \hat{E}, \hat{\delta})$ is the store encoding of $St_{\widehat{Env}}$ under id and (3) the variables `tau:x`, `tau:k` and `tau:m` contain value encodings of x, k and m, respectively, under id, and
- $\hat{x}$, $\hat{k}$ and $\hat{m}$ are all undefined.

As described in Section 2.1 result of a LiXQueryS expression is defined in terms of its result sequence and the result store. This is also apparent from the LiXQuery the formal semantics where each semantic rule has a conclusion of the form $St, Env \vdash e \Rightarrow (St', v)$. Because of the language specifications of LiXQuery, in order to achieve similar behavior in our simulation, we must encode this modified store in a result sequence only. Therefore we define the τ' encoding in Figure 4.2. Here we refer to the part of the store that is encoded in the environment as $St_{\widehat{v}}$. Since $St_{\widehat{v}}$ describes the part of St that is retrieved or created by preceding evaluations it must hold that $St' = \widehat{St} \cup St_{\widehat{v}}$ where $\widehat{St'} \cap St_{\widehat{v}}$ contains the documents that were retrieved with the `doc()` function before or during e'' was evaluated (see Figure 4.3).

Definition 4.8. Given a store $St' = (V, E, \ll, \nu, \sigma, \delta)$ and a value v over this store then a pair $(\widehat{St}, \widehat{v})$ with a store $\widehat{St}$ and an XML value $\widehat{v}$ is called a *store-value encoding of St and v* if there is a store $St_{\widehat{v}}$ and an injective function $id : V_{\widehat{v}} \rightarrow \mathbb{N}$ such that (1) $St' = \widehat{St} \cup St_{\widehat{v}}$, (2) all nodes in v are in $St_{\widehat{v}}$ and (3) $\hat{v} = \langle|V|\rangle \circ \hat{V} \circ \langle|E|\rangle \circ \hat{E} \circ \langle|\delta|\rangle \circ \hat{\delta} \circ \tilde{v}$ where $(\hat{V}, \hat{E}, \hat{\delta})$ is the store encoding of $St_{\widehat{v}}$ under id, and $\tilde{v}$ is the value encoding of v under id.

Based on this input/output encoding we can give the formal meaning of the diagram in Figure 4.2 and define when a transformation function defines a correct simulation.

Definition 4.9. A transformation function ϵ is said to be a *correct transformation* if it holds for every store St and environment Env that if $St, Env \vdash e \Rightarrow St', v$ and $(\widehat{St}, \widehat{Env})$ is store-environment encoding of St and Env under tr then it holds that $\widehat{St}, \widehat{Env} \vdash \epsilon(e) \Rightarrow \widehat{St}, \widehat{v}$ where $(\widehat{St}, \widehat{v})$ is a store-value encoding of St' and v.

The translated expression $\epsilon(e)$ will work on the encoded store and environment in $\widehat{Env}$ and may result in an encoded value $\hat{v}$ The store $\widehat{St}$ remain unchanged. Note that in order to obtain an expression e' as required by Theorem 4.4 we have to execute before $\epsilon(e)$ a function that replaces the values of all the variables with the encoded values and if they contain nodes then the connected parts of the store need to be encoded as well. Moreover, after we have executed $\epsilon(e)$ we have to decode the resulting value. This will be explained later in Section **??**

4.4 A Correct Transformation Function

In this section we construct a transformation function $\epsilon : LQE \rightarrow LQE$ and show that the following theorem holds.

Theorem 4.10. *The transformation function ϵ is a correct transformation function.*

The result of $\epsilon(e)$ is defined by induction upon the structure of e. We will now define the translations for all the types of LiXQuery expressions. Helper functions will be defined in the `eps` namespace which is assumed to be distinct from all the used namespaces in e.

First we will define necessary helper functions `eps:V()`, `eps:E()`, `eps:delta()` and `eps:val()` which respectively extract $\widehat{V}$, $\widehat{E}$, $\widehat{\delta}$, and $\widetilde{v}$ from a store-value encoding. And for computing the store-value encoding give $\hat{V}$, $\hat{E}$, $\hat{\delta}$ and $\tilde{v}$ we declare a function `eps:stValEnc()` with formal arguments `$V, $E, $delta` and `$val`.

Function 4.1 eps:V

```
declare function eps:V($stValEnc) {
  for $i at $pos in $stValEnc
  where (($pos > 1) and($pos <= ($stValEnc[1]*4+1)))
  return
    $i
};
```

Function 4.2 eps:E

```
declare function eps:E($stValEnc) {
  for $i at $pos in $stValEnc
  where (($pos > ($stValEnc[1]*4+2))
    and ($pos <= ($stValEnc[1]*4+2)
      + ($stValEnc[$stValEnc[1]*4+2]*2)))
  return
   $i
};
```

4.4.1 Variables

In LiXQuery variables result in their value, in the simulation we must return a store value encoding in order. In this way variables can uniformly be used together with results from other expressions.

Definition 4.11 (variable).

ϵ(`$s`) = `eps:stValEnc($tau:V,$tau:E,$tau:delta,$s))`

4.4.2 Built-in functions

LiXQuery has several built-in functions which correspond to the same functions in XQuery, taking in account the omission of data types. Our simulation of these functions will make heavy use of the above introduced helper functions to extract the necessary information from the encodings.

Function 4.3 eps:delta

```
declare function eps:delta($stValEnc) {
  for $i at $pos in $stValEnc
  where
    (($pos > (($stValEnc[1]*4+2)
    + ($stValEnc[$stValEnc[1]*4+2]*2) + 1))
    and
    ($pos <=
    (($stValEnc[1]*4+2)
    + ($stValEnc[$stValEnc[1]*4+2]*2) + 1)
    + ($stValEnc[ ($stValEnc[1]*4+2)
    + ($stValEnc[$stValEnc[1]*4+2]*2) + 1 ]*2)))
  return
   $i
};
```

Function 4.4 eps:val

```
declare function eps:val($stValEnc) {
  for $i at $pos in $stValEnc
  where
    ($pos >
    (($stValEnc[1]*4+2)
    + ($stValEnc[$stValEnc[1]*4+2]*2) + 1)
    + ($stValEnc[ ($stValEnc[1]*4+2)
    + ($stValEnc[$stValEnc[1]*4+2]*2) + 1 ]*2))
  return
   $i
};
```

Function 4.5 eps:stValEnc

```
declare function eps:stValEnc($tau:V,$tau:E,$tau:delta,$v) {
  (fn:count($tau:V) idiv 4, $tau:V,
  fn:count($tau:E) idiv 2, $tau:E,
  fn:count($tau:delta) idiv 2, $tau:delta,
  $v)
};
```

doc-function

The `doc()` function loads new documents into our encoded store. The LiXQuery formal semantics assumes that all the documents of the web are already loaded into the store. This of course is not true in a real world application. Therefore documents must be loaded. In our transformation this also means that they have to be encoded and added tot the encode store. This is why the transformation of the `doc()` function is the most complex of the built-in functions, and uses several specific helper functions.

Definition 4.12 (doc-function)**.**

```
ϵ(doc(e)) = let $eps:res := ϵ(e)
            return eps:doc($eps:res)
```

Here the function `eps:doc()` checks if the document is already in the encoded store by comparing the URI's tot the URI's already present in $\widehat{\delta}$. If this is the case it just returns the associated simulated node id as found in $\widehat{\delta}$, else the `eps:doc()` function compares the real document node obtained with the given URI, to the real documents obtained via the URI's that are already present in $\widehat{\delta}$. If this is the case, only a new entry is added to $\widehat{\delta}$ linking the new URI to the node identifier of the encoded document. If the document is not present in $\widehat{\delta}$ the document is encoded. First a document node is added to the encoded store and with the resulting node identifier a new entry is added in $\widehat{\delta}$. Then, also using this identifier, the nodes of the document are encoded and added after this document node in $\widehat{V}$ by calling the eps:encodeDesc() function given as Function 4.7. The `eps:doc()` function finally returns a store-value encoding containing the (new) node identifier as the result sequence and the (updated) store, environment and delta.

During this proces several helper functions are used which are given as Function 4.8 to 4.16. Most notably are the functions `eps:addNode()` and `eps:addNodeAfter()` that perform the necessary encode store additions.

To be able to add nodes to the encoded store we define the helper functions `eps:addNode` and `eps:addNodeAfter`.

name-function

The `name()` function returns the name of a node. The transformation is straightforward, using the general helper functions.

Definition 4.13 (name-function)**.**

```
ϵ(name(e′)) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           eps:nu($eps:val[2], $tau:V) )
```

Function 4.6 eps:doc

```
declare function eps:doc($eps:res) {
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  (: check if we already loaded document :)
  let $docid := eps:docId($eps:val[2],$tau:delta)
  return
  if (not(empty($docid))) then
    eps:stValEnc($tau:V,$tau:E,$tau:delta,$docid)
  else
    let $docid := eps:realDocId($eps:val[2],$tau:delta)
    return
    if (not(empty($docid))) then
      (: update delta mapping :)
      let $tau:delta := ($tau:delta,$eps:val[2],$docid[2])
      return
      eps:stValEnc($tau:V,$tau:E,$tau:delta,$docid)
    else
      (: load the document :)
      (: add document node :)
      let $eps:res2 := eps:addNode("document","","",
        $tau:V,$tau:E,$tau:delta)
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      (: add to delta :)
      let $tau:delta := (eps:delta($eps:res2),$eps:val[2],$eps:val2[2])
      (: encode document :)
      let $eps:descendants :=
        (fn:doc($eps:val[2])/descendant-or-self::node()
        | fn:doc($eps:val[2])//@*)

      let $eps:res3 := eps:encodeDesc($eps:descendants,2,$eps:val2[2],
        ($eps:val2[2]),$tau:V,$tau:E,$tau:delta)
      let $tau:V := eps:V($eps:res3)
      let $tau:E := eps:E($eps:res3)
      let $tau:delta := eps:delta($eps:res3)
      return
        eps:stValEnc($tau:V,$tau:E,$tau:delta,(1,$eps:val2[2]))
};
```

Function 4.7 eps:encodeDesc

```
declare function eps:encodeDesc($eps:descendants,$eps:item,$eps:nodeid,$eps:idseq,
    $tau:V,$tau:E,$tau:delta) {
  if (fn:count($eps:descendants) >= $eps:item) then
    let $eps:desc := ($eps:descendants[$eps:item])
    let $eps:v := (
      typeswitch($eps:desc)
      case element()
        return ("element",fn:name($eps:desc),"")
      case text()
        return ("text","", fn:data($eps:desc))
      case attribute()
        return ("attribute",fn:name($eps:desc), fn:data($eps:desc))
      default return ())
    let $eps:res :=
      eps:addNodeAfter($eps:nodeid,
        eps:parentId($eps:desc,$eps:descendants,$eps:idseq),
        $eps:v[1],$eps:v[2],$eps:v[3],$tau:V,$tau:E,$tau:delta)
    let $tau:V := eps:V($eps:res)
    let $tau:E := eps:E($eps:res)
    let $tau:delta := eps:delta($eps:res)
    let $eps:val := eps:val($eps:res)
    return
    eps:encodeDesc($eps:descendants,$eps:item+1,$eps:val[2],
      ($eps:idseq,$eps:val[2]),
      $tau:V,$tau:E,$tau:delta)
  else eps:stValEnc($tau:V,$tau:E,$tau:delta,(0,0))
};
```

Function 4.8 eps:parentId

```
declare function eps:parentId($eps:item, $eps:posparents,$eps:idseq) {
  let $eps:parent := $eps:item/..
  for $pp at $pos in $eps:posparents
  where $pp is $eps:parent
  return $eps:idseq[$pos]
};
```

Function 4.9 eps:docId

```
declare function eps:docId($URI_filename, $tau:delta) {
  for $URI at $pos in $tau:delta
  where (($pos mod 2 = 1)
   and ($tau:delta[$pos]=$URI_filename))
  return
   (1,$tau:delta[$pos+1])
};
```

Function 4.10 eps:realDocId

```
declare function eps:realDocId($URI_filename, $tau:delta) {
  for $URI at $pos in $tau:delta
  where (($pos mod 2 = 1)
   and (doc($tau:delta[$pos]) is doc($URI_filename)))
  return
   (1,$tau:delta[$pos+1])
};
```

Function 4.11 eps:newId

```
declare function eps:newId($tau:V) {
  let $ids := (
    for $id at $pos in $tau:V
    where ($pos mod 4 = 1)
    return
      $id
  )
  return
    fn:max($ids)+1
  (: without max:
  for $id in $ids
  where every $id2 in $ids satisfies ($id2 <= $id)
  return
    $id + 1
  :)
};
```

Function 4.12 eps:addNode

```
declare function eps:addNode($type,$name,$text,$tau:V,$tau:E,$tau:delta) {
  let $eps:newid := eps:newId($tau:V)
  let $tau:V := ($tau:V, $eps:newid, $type,$name,$text)
  return
    eps:stValEnc($tau:V,$tau:E,$tau:delta,(1,$eps:newid))
};
```

Function 4.13 eps:addNodeAfter

```
declare function eps:addNodeAfter($eps:nodeid,$eps:parentid,
    $type,$name,$text,$tau:V,$tau:E,$tau:delta) {
  (: in our implementation new nodes are always added at the end :)
  (: assert $nodeid = nodeid of last node :)
  let $eps:newid := eps:newId($tau:V)
  let $tau:V := ($tau:V, $eps:newid, $type,$name,$text)
  let $tau:E := ($tau:E,$eps:parentid, $eps:newid)
  return
    eps:stValEnc($tau:V,$tau:E,$tau:delta,(1,$eps:newid))
};
```

Function 4.14 tau:less

```
declare function tau:less
  ($node1_id, $node2_id, $tau:E) {
return
if ((
 for $p at $posp in $tau:E
 where ($posp mod 2 = 1)
 return
  if ($tau:E[$posp+1] = $node1_id) then
   $p
  else
   ()
 )
 =
 (
 for $p2 at $posp2 in $tau:E
 where ($posp2 mod 2 = 1)
 return
  if ($tau:E[$posp2+1] = $node2_id) then
   $p2
  else
   ()
  )
  )
then
 $node1_id < $node2_id
else
 ()
};
```

The function `eps:nu()` returns the name of the specified node using the information encoded in $\widehat{V}$, which it accesses using the node id. If the node is not an element or attribute node, the empty sequence is returned.

Function 4.15 eps:nu

```
declare function eps:nu($eps:nodeId, $tau:V) {
let $eps:node := eps:getNode($eps:nodeId,$tau:V)
return
if (($eps:node[2] = "element")
    or
   ($eps:node[2] = "attribute"))
then
 (0,$eps:node[3])
else
 ()
};
```

Function 4.16 eps:getNode

```
declare function eps:getNode($eps:nodeId, $tau:V) {
  for $ids at $pos in $tau:V
  where (($pos mod 4 = 1) and ($ids = $eps:nodeId))
  return ($tau:V[$pos],$tau:V[$pos+1],$tau:V[$pos+2],$tau:V[$pos+3])
};
```

string-function

The `string()` function returns the string value of a node or an atomic value. Our transformation uses the function `eps:sigma()`, which returns the string value of the specified node using the information encoded in $\widehat{V}$, in a similar way to the name-function. This time if it is not a text node or attribute node, the empty sequence is returned.

Definition 4.14 (string-function).

```
ϵ(string(e′)) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return
  if ($eps:val[1] = 1) then
    eps:stValEnc($tau:V, $tau:E, $tau:delta,
             eps:sigma($eps:val[2], $tau:V) )
  else
    eps:stValEnc($tau:V, $tau:E, $tau:delta,
             (0,string($eps:val[2]))
```

Function 4.17 eps:sigma

```
declare function eps:sigma($eps:nodeId, $tau:V) {
let $eps:node := eps:getNode($eps:nodeId,$tau:V)
return
if (($eps:node[2] = "text")
    or
   ($eps:node[2] = "attribute"))
then
 (0,$eps:node[4])
else
 ()
};
```

xs:integer-function

The `xs:integer()` function converts stings to atomic values. We use a straightforward transformation. Due to the nature of our encoding it is easy to decode (selecting the second item in the sequence). We then apply the original function (`xs:integer()`), and encode again (create a $\langle 0, integer \rangle$ sequence).

Definition 4.15 (xs:integer-function)**.**

```
ϵ(xs:integer(e')) =
  let $eps:res := ϵ(e')
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
          (0,xs:integer($eps:val[2])) )
```

root-function

The `root()` function returns the root-node of a node. The transformation is achieved by the recursive `eps:root()` function, which recursively walks op the encoded tree, selecting the parent of the parent (using the `eps:parent()` function) until it can go no further.

Definition 4.16 (root-function)**.**

```
ϵ(root(e′)) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           eps:root($eps:val[2],$tau:E) )
```

Function 4.18 eps:root

```
declare function eps:root($eps:nodeId, $tau:E) {
  let $parentId := eps:parent($eps:nodeId, $tau:E)[2]
  return
  if (empty($parentId)) then
    (1,$eps:nodeId)
  else
    eps:root($parentId,$tau:E)
};
```

Function 4.19 eps:parent

```
declare function eps:parent($eps:nodeId, $tau:E) {
    for $pnid at $pos in $tau:E
    where (($pos mod 2 = 0) and ($pnid = $eps:nodeId))
    return (1,$tau:E[$pos - 1])
};
```

concat-function

The `concat()` function concatenates strings. The transformation is a straightforward decoding, applying the original function and encoding again, similar to the `xs:integer()` simulation.

Definition 4.17 (concat-function).

```
ϵ(concat(e′,e″)) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  let $eps:res2 := ϵ(e″)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val2 := eps:val($eps:res)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, concat($eps:val[2],$eps:val2[2])) )
```

true-, false-, not- and count-function

The `true()` and `false()` functions return the `true` and `false` atomic values. The `not()` function is the boolean not function and the `count()` function return the number of items in the sequence.

Definition 4.18 (true- and false-function).

```
ϵ(true()) =
  eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, true()) )

ϵ(false()) =
  eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, false()) )
```

The transformation of the `true()` and `false()` functions simply returns the encoded `true` and `false` atomic values.

Definition 4.19 (not-function).

```
ϵ(not(e′)) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, not($eps:val[2])) )
```

The `not()` function transformation does a decode, an original function application, and encoding.

Definition 4.20 (count-function).

```
ϵ(count(e′)) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, count($eps:val) idiv 2) )
```

The `count()` function transformation does a decode. Then applies the original function. Before encoding again it does a division by two due to the nature of our encoding.

position- and last-function

The `position()` function returns the position of the context item in the context sequence. The `last()` function returns the size of the context sequence.

Definition 4.21 (position- and last-function).

```
ϵ(position()) =
  eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, $tau:k) )

ϵ(last()) =
  eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, $tau:m) )
```

The transformation of these functions simply return information which is available in the encoded environment.

4.4.3 If-expressions

The *if-expression* is a very important expression in LiXQuery because it provides the semantics of the where function which is not present in LiXQuery. The transformation of it is simple, the condition of the if is changed to account for the encoding an the transformation is recursively applied to the expressions within the body of the *if-expression*.

Definition 4.22 (If-expression).

```
ϵ(if $e$ then $e_1$ else $e_2$) =
  let $eps:res := ϵ($e$)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  if ($eps:val[2] = true() ) then
    return ϵ($e_1$)
  else
    return ϵ($e_2$)
```

4.4.4 For-expressions

The *for-expression* is the most fundamental type of expression in LiXQuery. In it's transformation we assume a number x that is unique for each for-expression that has to be transformed. This is used to define for every for-expression a unique function `eps: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 transformation of the *for-expression* is then defined as follows.

Definition 4.23 (For-expression).

```
ϵ(for $$s$ at $$s'$ in $e$ return $e'$) =
  let $eps:res := ϵ($e$)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:for$_x$(1, $eps:val, $tau:V, $tau:E,
      $tau:delta, $vars_x$)
```

with `eps:for`$_x$`()` defined as follows:

```
declare function eps:for$_x$($eps:pos, $eps:seq,
          $tau:V, $tau:E, $tau:delta, $vars_x$) {
  if ($eps:pos <= (count($eps:seq) idiv 2)) then
    let $s := ($eps:seq[$eps:pos*2-1], $eps:seq[$eps:pos*2])
    let $s' := $eps:pos
    let $eps:res1 := ϵ($e'$)
    let $tau:V := eps:V($eps:res1)
    let $tau:E := eps:E($eps:res1)
    let $tau:delta := eps:delta($eps:res1)
    let $eps:val1 := eps:val($eps:res1)
    let $eps:res2 := eps:for_x($eps:pos + 1, $eps:seq,
             $tau:V, $tau:E, $tau:delta, vars_x)
    let $tau:V := eps:V($eps:res2)
    let $tau:E := eps:E($eps:res2)
    let $tau:delta := eps:delta($eps:res2)
    let $eps:val2 := eps:val($eps:res2)
    return eps:stValEnc($tau:V, $tau:E, $tau:delta,
              ( $eps:val1, $eps:val2 ))
  else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
}
```

4.4.5 Let-expression

The *let-expression* provides us with a way to declare variables. The transformation evaluates the expression and extracts its value an binds it using a *let-expression* of which the return clause is the transformation of the original return clause expression.

Definition 4.24 (Let-expression).

```
ϵ(let $s := e return e′) =
  let $eps:res := ϵ(e)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $s := eps:val($eps:res)
  return ϵ(e′)
```

4.4.6 Concatenation

The results of evaluations of expressions can be *concatenated.* The transformation of this instruction is important, due to the specific encoding scheme we use. Both expressions are evaluated, their values extracted from the encoding. Then the concatenation is applied and this result is encoded back. It is important to note that the evaluation of the second expression takes place in the changed environment and store of the first expression. This is now achieved without recursion (unlike the for expression) because the defined number of expressions is defined at the time we are applying the transformation.

Definition 4.25 (Concatenation).

```
ϵ(e′, e″) =
  let $eps:res1 := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
let $tau:delta := eps:delta($eps:res)
let $eps:val1 := eps:val($eps:res)
let $eps:res2 := ϵ(e″)
let $tau:V := eps:V($eps:res)
let $tau:E := eps:E($eps:res)
let $tau:delta := eps:delta($eps:res)
let $eps:val2 := eps:val($eps:res)
return stValEnc($tau:V,$tau:E,$tau:delta,($eps:val1,$eps:val2))
```

4.4.7 Boolean Operators

LiXQuery provides basic *boolean operators* which allow complex conditions in the other expressions. The transformation of these operators is straightforward and uses the decoding-encoding approach.

Definition 4.26 (Boolean Operators).

```
ϵ(e′ and e″)) =
  let $eps:res1 := ϵ(e′)
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e′)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, $eps:val1[2] and $eps:val2[2]) )

ϵ(e′ or e″)) =
  let $eps:res1 := ϵ(e′)
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e′)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
          (0, $eps:val1[2] or $eps:val2[2]) )
```

4.4.8 Atomic Value Comparison

The *atomic value comparison operators* provided in LiXQuery are the less and equal general comparison operators of XQuery restricted to atomic values. In the translation both epressions are evaluated and the extracted values are passed on to the helper functions `eps:atomicEqual()` and `eps:atomicLess()`. These functions in turn, use the helper function `eps:getAtomics()` to tranform the sequence of encoded atomics in the corresponding sequence of dencoded atomics, and apply the original LiXQuery less and equal operators respectively. After this the result is encoded again.

Definition 4.27 (Atomic Value Comparison).

```
ϵ(e' = e'')) =
  let $eps:res1 := ϵ(e')
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e')
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           eps:atomicEqual($eps:val1, $eps:val2) )

ϵ(e' < e'')) =
  let $eps:res1 := ϵ(e')
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e')
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           eps:atomicLess($eps:val1, $eps:val2) )
```

Function 4.20 eps:atomicEqual

```
declare function eps:atomicEqual($eps:val1,$eps:val2) {
  (0, (eps:getAtomics($eps:val1) = eps:getAtomics($eps:val2)))
};
```

Function 4.21 eps:atomicLess

```
declare function eps:atomicLess($eps:val1,$eps:val2) {
  (0, (eps:getAtomics($eps:val1) < eps:getAtomics($eps:val2)))
};
```

4.4.9 Node Comparison

The LiXQuery *node comparison operators* are based on node id and document order. The transformation of node comparison expressions is therefore done by extracting the information of identity and position contained in the store-value encoding.

Definition 4.28 (Node Comparison).

```
ϵ(e' is e'')) =
  let $eps:res1 := ϵ(e')
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e')
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, $eps:val1[2] = $eps:val2[2]) )
```

Function 4.22 eps:getAtomics

```
declare function eps:getAtomics($atomics) {
  for $i at $pos in $atomics
  where ($pos mod 2 = 0)
  return $i
};
```

```
ϵ(e' << e'')) =
  let $eps:res1 := ϵ(e')
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e')
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
         (0, eps:pos($tau:val1[2], $tau:V) <
           eps:pos($tau:val2[2], $tau:V) ) )
```

The function `eps:pos($eps:nodeid, $tau:V)` iterates over `$tau:V` and returns the position of the node with id `$eps:nodeid` which reflects the document order.

Function 4.23 eps:pos

```
declare function eps:pos($eps:nodeid, $tau:V) {
  for $node at $pos in $tau:V
  where (($pos mod 4 = 1) and ($node = $eps:nodeid))
  return ($pos -1) idiv 4 + 1
};
```

4.4.10 Arithmetic

LiXQuery provides the basic *arithmetic* for the numeric type it supports, integers. Here we define how the addition is to be transformed. The subtraction, multiplication and division are defined in a similar way. The transformation is again using decode-encode method only.

Definition 4.29 (Addition).

```
ϵ(e' + e'')) =
  let $eps:res1 := ϵ(e')
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e′)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           (0, $eps:val1[2] + $eps:val2[2]) )
```

4.4.11 Union

The *union operator* in LiXQuery creates the union of two expression results. It is however important to know that this operator implicitly performs duplicate elimination and sorting in document order, this is apparent form the LiXQuery semantics. Due to our encoding we have to make these operations explicit in our transformation. The `eps:docord` function provides these functionalities.

Definition 4.30 (Union).

```
ϵ(e′ | e″)) =
  let $eps:res1 := ϵ(e′)
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := ϵ(e′)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:stValEnc($tau:V, $tau:E, $tau:delta,
           eps:docord(($eps:val1, $eps:val2), $tau:V) )
```

4.4.12 Axis Steps

A path expression in LiXQuery is made up of subsequent *axis steps*. The `.` and `..` are transformed using `$tau:x` and `eps:parent()` respectively which we already defined. The other axis steps are transformed using the new `eps:children()` function, which extracts the information from the encoding (`$tau:E` in particular) and applies the necessary filters on type or name.

Definition 4.31 (Axis Steps).

- `ϵ(.)  = eps:stValEnc($tau:V,$tau:E,$tau:delta,$tau:x)`

Function 4.24 eps:docord

```
declare function eps:docord($eps:sequence, $tau:V) {
  declare function eps:docord($eps:sequence, $tau:V) {
  let $eps:niseq := (
    for $ni at $posi in $eps:sequence
    where ($posi mod 2 = 0)
    return $ni
  )
  for $vn at $posv in $tau:V
  where (($posv mod 4 = 1) and ($vn = $eps:niseq))
  return
    (1,$vn)
};
```

– ϵ`(..)` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,`
  `eps:parent($tau:x,$tau:E)` (Function 4.19)

– For all strings s occurring as axis steps holds
  ϵ`(`s`)` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,`
  `eps:docord(eps:children(`s`, "element", $tau:x,`
  `$tau:V,$tau:E), $tau:V)` (Function 4.25)

– For all strings s occurring as axis steps holds preceded by an “@” holds
  ϵ`(@`s`)` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,`
  `eps:docord(eps:children(`s`, "attribute", $tau:x, $tau:V,$tau:E), $tau:V)`

– ϵ`(*)` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,`
  `eps:docord(eps:children(*, "element", $tau:x,`
  `$tau:V,$tau:E), $tau:V)`

– ϵ`(@*)` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,`
  `eps:docord(eps:children(*, "attribute", $tau:x,`
  `$tau:V,$tau:E), $tau:V)`

– ϵ`(text())` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,`
  `eps:docord(eps:children(*, "text", $tau:x, $tau:V,$tau:E), $tau:V)`

Function 4.25 eps:children

```
declare function eps:children($s, $a_or_e,
        $tau:x, $tau:V, $tau:E) {
  for $c at $posc in $tau:E
  where ($posc mod 2 = 0)
  return
  if (($tau:E[$posc - 1] = $tau:x[2])
      and (eps:getNode($c,$tau:V)[2] = $a_or_e)
      and (($s = "*") or eps:nu($c, $tau:V)[2] = $s))
  then
   (1,$c)
  else
   ()
};
```

4.4.13 Filter-expression

As the semantics of the LiXQuery *filter-expression* is similar to that of the LiXQuery for expression it is no suprise that the filter expression transformation is similar to that of the for expressions. The difference is in its use of the the context item and context position instead of user defined variables.

Definition 4.32 (Filter-expression).

```
$\epsilon(e'$ [$e''$]) =
  let $eps:res := $\epsilon(e')$
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:filter$_x$(1, $eps:val, $tau:V, $tau:E,
      $tau:delta, $vars_x$)
```

with eps:filter$_x$() defined as follows:

```
declare function eps:filter$_x$($tau:k, $eps:seq,
          $tau:V, $tau:E, $tau:delta, $vars_x$) {
  let $tau:m := count($eps:seq) idiv 2
  return
    if ($tau : k <=$tau:m) then
      let $tau:x := ($eps:seq[$tau:k*2-1], $eps:seq[$tau:k*2])
      let $eps:res1 := $\epsilon(e'')$
      let $tau:V := eps:V($eps:res1)
      let $tau:E := eps:E($eps:res1)
      let $tau:delta := eps:delta($eps:res1)
      let $eps:val1 := eps:val($eps:res1)
      let $eps:res2 := eps:filter$_x$($tau:k+1, $eps:seq,
               $tau:V, $tau:E, $tau:delta, $vars_x$)
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $tau:delta := eps:delta($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      return
        typeswitch($eps:val1[2])
        case xs:integer return
          if ($eps:val1[2] = $tau:k)
          then $eps:stValEnc($tau:V, $tau:E, $tau:delta,
                ( $eps:val1, $eps:val2 ))
          else $eps:stValEnc($tau:V, $tau:E, $tau:delta,
                ($eps:val2 ))
        case xs:boolean return
          if ($eps:val1[2] = true())
          then $eps:stValEnc($tau:V, $tau:E, $tau:delta,
                ( $eps:val1, $eps:val2 ))
          else $eps:stValEnc($tau:V, $tau:E, $tau:delta,
                ($eps:val2 ))
        default return
          eps:stValEnc($tau:V, $tau:E, $tau:delta,$eps:val2 ))
    else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
}
```

4.4.14 Path-expressions

The transformation of *path-expression* also uses the technique of recursion in the same way as the for- and filter-expression translation. This is because it also iterates over a sequence where the resulting store of the previous step is passed on to the following step. The `/` and the `//` transformation use the same basis, the eps:path$_x$() function. The `//` just applies an extra preceding step where it gets all descendants with the `eps:decendants-o-s()` function. This function uses recursion to extract the information from `$tau:E`.

Definition 4.33 (/ path-expression).

```
$\epsilon$($e'$ / $e''$]) =
  let $eps:res := $\epsilon$($e'$)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  let $eps:res2 := eps:path_x(1, $eps:val, $tau:V, $tau:E,
      $tau:delta, vars_x)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return $eps:stValEnc($tau:V, $tau:E, $tau:delta,
      eps:docord($eps:val2, $tau:V))
```

with eps:path$_x$() defined as follows:

```
declare function eps:path_x($tau:k, $eps:seq,
          $tau:V, $tau:E, $tau:delta, vars_x) {
  let $tau:m := count($eps:seq) idiv 2
  return
    if ($tau:k <= $tau:m ) then
      let $tau:x := ($eps:seq[$tau:k*2-1], $eps:seq[$tau:k*2])
      let $eps:res1 := ϵ(e'')
      let $tau:V := eps:V($eps:res1)
      let $tau:E := eps:E($eps:res1)
      let $tau:delta := eps:delta($eps:res1)
      let $eps:val1 := eps:val($eps:res1)
      let $eps:res2 := eps:path_x($tau:k+1, $eps:seq,
              $tau:V, $tau:E, $tau:delta, vars_x)
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $tau:delta := eps:delta($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      return eps:stValEnc($tau:V, $tau:E, $tau:delta,
          ( $eps:val1, $eps:val2 ))
    else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
}
```

Definition 4.34 (// path-expression).

```
ϵ(e' / e'']) =
  let $eps:res := ϵ(e')
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  let $eps:res2 :=
    eps:docord(eps:path_x(1, eps:descendants-o-s($eps:val),
      $tau:V, $tau:E, $tau:delta, vars_x))
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return $eps:stValEnc($tau:V, $tau:E, $tau:delta,
      eps:docord($eps:val2))
```

with `eps:path`$_x$`()` defined in Definition 4.33, and `eps:decendants-o-s()` defined as Function 4.26.

Function 4.26 eps:descendants-o-s

```
declare function eps:decendants-o-s($tau:E,$pps) {
let $can := (for $d at $posd in $tau:E
where (($posd mod 2 = 0) and ($tau:E[$posd - 1] = $pps)
    and (not($d = $pps)))
return $d)
return
if (not(empty($can))) then
  eps:decendants-o-s($tau:E,($pps,$can[1]))
else
  for $nid in $pps
  return (1,$nid)
};
```

4.4.15 Literals and the empty sequence

Literals return the atomic values they represent. In our simulation we must encode literals in their special form and return a store-value encoding. For the *empty sequence* this is also true.

Definition 4.35 (variable).

ϵ(*literal*) = `eps:stValEnc($tau:V,$tau:E,$tau:delta,(0,` *literal*`)))`

Definition 4.36 (empty sequence).

ϵ`(())` = `eps:stValEnc($tau:V,$tau:E,$tau:delta,()))`

4.4.16 Constructors

The transformation of a *construction operator* extends the encoded store. This is a crucial part of the simulation. As with the `doc()` function, we have to add nodes to the store and therefore use the same functions `eps:addNode()` defined as Function 4.12 and `eps:addNodeAfer()` defined as Function 4.13. The attribute and text node constructors are straightforward. They evaluate the subexpressions to string values and with this information call the `addNode()` function. The element and the document constructor are similar, they must both create deep equal copies of the node sequence returned by the contents expression. The element constructor has one extra evaluation to determine its name in the same way it is done in the attribute constructor. The copying is performed by the function `eps:addChildren($parEnc, $chEnc, $tau:V, $tau:E, $tau:delta)`. It makes deep copies for all the nodes encoded in `chEnc`,adds these under the node encoded in `$parEnc` and returns a store-value encoding with the new store and the parent node. The function uses recursion in the same way as the transformation of the for-, filter- and path-expressions, in order to be able to iterate with side-effects on the store.

Definition 4.37 (element constructor).

```
ϵ(element {e′}{e″}) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  let $eps:res2 := ϵ(e″)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:addElement($eps:val, $eps:val2,
    $tau:V,$tau:E,$tau:delta)
```

with `eps:addElement()` declared as follows.

```
declare function eps:addElem($eps:nameEnc $eps:chEnc, $tau:V,
              $tau:E, $tau:delta) {
  let $eps:res1 := eps:addNode("element",$nameEnc[2],"",
    $tau:V,$tau:E,$tau:delta)
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val := eps:val($eps:res1)
  let $eps:res2 := eps:addChildren($eps:val1,$eps:chEnc,
    $tau:V,$tau:E,$tau:delta)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return $eps:stValEnc($tau:V, $tau:E, $tau:delta,
      $eps:val1)
}
```

Function 4.27 eps:addChildren

```
declare function eps:addChildren($eps:parentEnc, $eps:chEnc,
    $tau:V, $tau:E, $tau:delta) {
  eps:copyAsChildren($eps:parentEnc[2],$eps:parentEnc[2],
    $eps:chEnc,2,$tau:V,$tau:E,$tau:delta)
};
```

Definition 4.38 (attribute constructor).

```
ϵ(attribute {e′}{e″}) =
  let $eps:res := ϵ(e′)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  let $eps:res2 := ϵ(e″)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return eps:addNode("attribute",$eps:val, $eps:val2,
    $tau:V,$tau:E,$tau:delta)
```

Definition 4.39 (text node constructor).

```
ϵ(text {e}) =
  let $eps:res := ϵ(e)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
```

Function 4.28 eps:copyAsChildren

```
declare function eps:copyAsChildren($eps:parentid,$eps:nodeid,
    $eps:chEnc,$eps:child,$tau:V,$tau:E,$tau:delta){
  if (fn:count($eps:chEnc) >= $eps:child) then
    let $eps:childEnc :=
      eps:getNode($eps:chEnc[$eps:child],$tau:V)
    let $eps:res2 := eps:addNodeAfter($eps:nodeid,$eps:parentid,
      $eps:childEnc[2],$eps:childEnc[3],$eps:childEnc[4],
        $tau:V,$tau:E,$tau:delta)
    let $tau:V := eps:V($eps:res2)
    let $tau:E := eps:E($eps:res2)
    let $tau:delta := eps:delta($eps:res2)
    let $eps:val := eps:val($eps:res2)
    let $eps:desc := eps:decendants-o-s($tau:E, $eps:childEnc[1])
    let $eps:res3 := eps:deep-copy($eps:desc,4,$eps:val[2],
      ($eps:val[2]),$tau:V,$tau:E,$tau:delta)
    let $tau:V := eps:V($eps:res3)
    let $tau:E := eps:E($eps:res3)
    let $tau:delta := eps:delta($eps:res3)
    let $eps:val3 := eps:val($eps:res3)
    return
    eps:copyAsChildren($eps:parentid,$eps:val3[2],$eps:chEnc,
      $eps:child+2,$tau:V,$tau:E,$tau:delta)
  else eps:stValEnc($tau:V,$tau:E,$tau:delta,(0,0))
};
```

Function 4.29 eps:deep-copy

```
declare function eps:deep-copy($eps:seq,$eps:item,$eps:nodeId,
    $eps:nodeIdSeq,$tau:V,$tau:E,$tau:delta) {
  if (fn:count($eps:seq) >= $eps:item) then
    let $eps:nodeId := $eps:seq[$eps:item]
    let $eps:nodeEnc := eps:getNode($eps:nodeId,$tau:V)
    let $eps:res := eps:addNodeAfter($eps:nodeId,
      eps:copiedParentId($eps:nodeId,$eps:seq,
        $eps:nodeIdSeq,$tau:E),
      $eps:nodeEnc[2],$eps:nodeEnc[3],$eps:nodeEnc[4],
        $tau:V,$tau:E,$tau:delta)
    let $tau:V := eps:V($eps:res)
    let $tau:E := eps:E($eps:res)
    let $tau:delta := eps:delta($eps:res)
    let $eps:val := eps:val($eps:res)
    return
    eps:deep-copy($eps:seq,$eps:item+2,$eps:val[2],
      ($eps:nodeIdSeq,$eps:val[2]),$tau:V, $tau:E,$tau:delta)
  else eps:stValEnc($tau:V,$tau:E,$tau:delta,(1,$eps:nodeId))
};
```

Function 4.30 eps:copiedParentId

```
declare function eps:copiedParentId($eps:nodeId, $eps:ppSeq,
    $eps:idSeq, $tau:E) {
  let $eps:parent := (
    for $n at $npos in $tau:E
    where (($npos mod 2 = 0) and ($n = $eps:nodeId))
    return
    $tau:E[$npos - 1]
  )
  for $pp at $pos in $eps:ppSeq
  where (($pos mod 2 = 0) and ($pp = $eps:parent))
  return $eps:idSeq[$pos idiv 2]
};
```

```
  return eps:addNode("text","", $eps:val,$tau:V,$tau:E,$tau:delta)
```

Definition 4.40 (document node constructor).

```
ϵ(document {e}) =
  let $eps:res := ϵ(e)
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  let $eps:res1 := eps:addNode("document","","",
    $tau:V,$tau:E,$tau:delta)
  let $tau:V := eps:V($eps:res1)
  let $tau:E := eps:E($eps:res1)
  let $tau:delta := eps:delta($eps:res1)
  let $eps:val1 := eps:val($eps:res1)
  let $eps:res2 := eps:addChildren($eps:val1,$eps:val,
    $tau:V,$tau:E,$tau:delta)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $tau:delta := eps:delta($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  return $eps:stValEnc($tau:V, $tau:E, $tau:delta,
      $eps:val1)
```

4.4.17 Typeswitch-expression

The typeswitch-expression allows us to execute different expressions based on the type of a variable. This variable can be a node or an atomic value. The transformed typeswitch-expression will need to work on encoded nodes and atomic values, and we therefore need to decode them. For the atomic values the cases in the body typeswitch can remain unchanged. We just need to change the switch variable into the decoded atomic value (the second component of the encoding). For a typeswitch on nodes we need to extract the type information from the store first an can then translate the typeswitch into an `if-else` using the type information.

Definition 4.41 (Typeswitch-expression).

```
ϵ(typeswitch (e) case t_1
    return e_1 ...  case t_m return e_m default return e_{m+1}) =
  let $eps:res := ϵ(e)
  let $tau:V := eps:V($eps:res)
let $tau:E := eps:E($eps:res)
let $tau:delta := eps:delta($eps:res)
let $eps:val := eps:val($eps:res)
return
  if ($eps:val[1] = 0) then
    typeswitch $eps:val[2]
    case $t_1$
    return
      $\epsilon(e_1)$
    ...
    case $t_m$return
      $\epsilon(e_m)$
    default
    return
      $\epsilon(e_{m+1})$
  else
    let $eps:nodeId := $eps:val[2]
    return
      if $\epsilon(t_1)$ then
      return
        $\epsilon(e_1)$
      else
      ...
      if $\epsilon(t_m)$ then
      return
        $\epsilon(e_m)$
      else
      return
        $\epsilon(e_{m+1})$
```

where $\epsilon(t_i)$ is defined as follows:

ϵ(document-node()) = (eps:getNode($eps:nodeId)[2] = "document")
ϵ(attribute()) = (eps:getNode($eps:nodeId)[2] = "attribute")
ϵ(text()) = (eps:getNode($eps:nodeId)[2] = "text")
ϵ(element()) = (eps:getNode($eps:nodeId)[2] = "element")
and for all other t_i
$\epsilon(t_i)$ = (fn:false())

4.4.18 Functions

The ability to define functions in LiXQuery (and XQuery) increases the usability. The fact that these functions can be recursive adds a lot of possibilities. The transformation of the function declaration adds the variables encoding the store to the formal parameters and transforms the body.

Definition 4.42 (Function Declaration)**.**

```
ϵ(f($s_1$,...,$s_m$) { e };) =
  declare function eps:f($s_1$,...,$s_m$,$tau:V,$tau:E,$tau:delta) {
    ϵ(e)
  }
```

The transformation of the function call evaluates all actual parameters, each one in the context (changed store) of the preceding. All the resulting values are passed as parameters to the transformed function together with the variables encoding the result store of the last parameter evaluation.

Definition 4.43 (Function Call)**.**

```
ϵ(f($e_1$,...,$e_m$)) =
  let $eps:res$_1$ := ϵ($e_1$)
  let $tau:V := eps:V($eps:res$_1$)
  let $tau:E := eps:E($eps:res$_1$)
  let $tau:delta := eps:delta($eps:res$_1$)
  let $eps:val$_1$ := eps:val($eps:res$_1$)
  ...
  let $eps:res$_m$ := ϵ($e_m$)
  let $tau:V := eps:V($eps:res$_m$)
  let $tau:E := eps:E($eps:res$_m$)
  let $tau:delta := eps:delta($eps:res$_m$)
  let $eps:val$_m$ := eps:val($eps:res$_m$)
  return
    eps:f($eps:val$_1$,...,$eps:val$_m$,$tau:V,$tau:E,$tau:delta)
```

with `eps:f(...)` is the translated version of `f(...)`.

4.5 An Illustrative Example

In this section we provide an example of what a simulated expression looks like. The simple query in Example 4.3 is simulated by the expression in Exapmle 4.4. This is only the expression itself, it is preceded by function declarations specificly generated for this expression, given in Example 4.5 to 4.8, that provide the semantics for the path and for expressions. In Example 4.4 we see that the `doc()` call is simulated and the simulation of the path expression that follows it is called (`eps:path_789456()`). The result of this is passed onto a function that simulates the for expression (`eps:for_3423973`). Example 4.6 simulates the path expression on the doc function and applies the simulated `table` axis step and the following path expression (`eps:path_123654`). This path expression is implemented in Example 4.5 which in turn uses the simulated `row` axis step. The function that implements the for expression is given in Example 4.8 and contains a call to the `eps:path_456789()` function which simulates the path expression on `$row` together with the `b` axis step, given in Example 4.7.

Example 4.3 A Simple LiXQuery Query

```
for $row at $pos in doc("table.xml")/table/row
return $row/b
```

Example 4.4 A Simulated LiXQuery Query

```
let $eps:res := (
    let $eps:res := (
      let $eps:res := eps:stValEnc($tau:V, $tau:E, $tau:delta,(0,"table.xml"))
      return eps:doc($eps:res)
    )
    let $tau:V := eps:V($eps:res)
    let $tau:E := eps:E($eps:res)
    let $tau:delta := eps:delta($eps:res)
    let $eps:val := eps:val($eps:res)
    let $eps:res2 := eps:path_789456(1, $eps:val, $tau:V, $tau:E,$tau:delta)
    let $tau:V := eps:V($eps:res2)
    let $tau:E := eps:E($eps:res2)
    let $tau:delta := eps:delta($eps:res2)
    let $eps:val2 := eps:val($eps:res2)
    return eps:stValEnc($tau:V, $tau:E, $tau:delta,
      eps:docord($eps:val2, $tau:V))
  )
  let $tau:V := eps:V($eps:res)
  let $tau:E := eps:E($eps:res)
  let $tau:delta := eps:delta($eps:res)
  let $eps:val := eps:val($eps:res)
  return eps:for_3423973(1, $eps:val, $tau:V, $tau:E,$tau:delta)
```

Example 4.5 Generated path_x expression

```
declare function eps:path_123654($tau:k, $eps:seq, $tau:V, $tau:E, $tau:delta) {
  let $tau:m := fn:count($eps:seq) idiv 2
  return
    if ( $tau:k <= $tau:m ) then
      let $tau:x := ($eps:seq[$tau:k*2-1], $eps:seq[$tau:k*2])
      let $eps:res1 := (
        eps:stValEnc($tau:V,$tau:E,$tau:delta,
        eps:docord(eps:children("row", "element", $tau:x,
          $tau:V,$tau:E), $tau:V))
      )
      let $tau:V := eps:V($eps:res1)
      let $tau:E := eps:E($eps:res1)
      let $tau:delta := eps:delta($eps:res1)
      let $eps:val1 := eps:val($eps:res1)
      let $eps:res2 := eps:path_123654($tau:k + 1, $eps:seq, $tau:V, $tau:E, $tau:del
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $tau:delta := eps:delta($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      return eps:stValEnc($tau:V, $tau:E, $tau:delta,
      ( $eps:val1, $eps:val2 ))
    else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
};
```

Example 4.6 Generated path$_x$ expression

```
declare function eps:path_789456($tau:k, $eps:seq, $tau:V, $tau:E, $tau:delta) {
  let $tau:m := count($eps:seq) idiv 2
  return
    if ( $tau:k <= $tau:m ) then
      let $tau:x := ($eps:seq[$tau:k*2-1], $eps:seq[$tau:k*2])
      let $eps:res1 := (
        let $eps:res := (
          (: table :)
          eps:stValEnc($tau:V,$tau:E,$tau:delta,
          eps:docord(eps:children("table", "element", $tau:x,
          $tau:V,$tau:E), $tau:V))
        )
        let $tau:V := eps:V($eps:res)
        let $tau:E := eps:E($eps:res)
        let $tau:delta := eps:delta($eps:res)
        let $eps:val := eps:val($eps:res)
        let $eps:res2 := eps:path_123654(1, $eps:val, $tau:V, $tau:E,$tau:delta)
        let $tau:V := eps:V($eps:res2)
        let $tau:E := eps:E($eps:res2)
        let $tau:delta := eps:delta($eps:res2)
        let $eps:val2 := eps:val($eps:res2)
        return eps:stValEnc($tau:V, $tau:E, $tau:delta,
        eps:docord($eps:val2, $tau:V))
      )
      let $tau:V := eps:V($eps:res1)
      let $tau:E := eps:E($eps:res1)
      let $tau:delta := eps:delta($eps:res1)
      let $eps:val1 := eps:val($eps:res1)
      let $eps:res2 := eps:path_789456($tau:k+1, $eps:seq,
      $tau:V, $tau:E, $tau:delta)
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $tau:delta := eps:delta($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      return eps:stValEnc($tau:V, $tau:E, $tau:delta,
      ( $eps:val1, $eps:val2 ))
    else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
};
```

Example 4.7 Generated path$_x$ expression

```
declare function eps:path_456789($tau:k, $eps:seq,
  $tau:V, $tau:E, $tau:delta) {
  let $tau:m := fn:count($eps:seq) idiv 2
  return
    if ($tau:k <= $tau:m) then
      let $tau:x := ($eps:seq[$tau:k*2-1], $eps:seq[$tau:k*2])
      let $eps:res1 := (
        eps:stValEnc($tau:V,$tau:E,$tau:delta,
        eps:docord(eps:children("b", "element", $tau:x,
        $tau:V,$tau:E), $tau:V))
      )
      let $tau:V := eps:V($eps:res1)
      let $tau:E := eps:E($eps:res1)
      let $tau:delta := eps:delta($eps:res1)
      let $eps:val1 := eps:val($eps:res1)
      let $eps:res2 := eps:path_456789($tau:k+1, $eps:seq,
      $tau:V, $tau:E, $tau:delta)
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $tau:delta := eps:delta($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      return eps:stValEnc($tau:V, $tau:E, $tau:delta,
      ( $eps:val1, $eps:val2 ))
    else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
};
```

Example 4.8 Generated for$_x$ expression

```
declare function eps:for_3423973($eps:pos, $eps:seq, $tau:V, $tau:E, $tau:delta) {
  if ($eps:pos <= (fn:count($eps:seq) idiv 2)) then
    let $pos := $eps:pos
    let $row := ($eps:seq[($eps:pos * 2) - 1], $eps:seq[($eps:pos) *2])
    let $eps:res1 := (
      let $eps:res := eps:stValEnc($tau:V, $tau:E, $tau:delta,$row)
      let $tau:V := eps:V($eps:res)
      let $tau:E := eps:E($eps:res)
      let $tau:delta := eps:delta($eps:res)
      let $eps:val := eps:val($eps:res)
      let $eps:res2 := eps:path_456789(1, $eps:val, $tau:V, $tau:E,
        $tau:delta)
      let $tau:V := eps:V($eps:res2)
      let $tau:E := eps:E($eps:res2)
      let $tau:delta := eps:delta($eps:res2)
      let $eps:val2 := eps:val($eps:res2)
      return eps:stValEnc($tau:V, $tau:E, $tau:delta,
      eps:docord($eps:val2, $tau:V))
    )
    let $tau:V := eps:V($eps:res1)
    let $tau:E := eps:E($eps:res1)
    let $tau:delta := eps:delta($eps:res1)
    let $eps:val1 := eps:val($eps:res1)
    let $eps:res2 := eps:for_3423973($eps:pos + 1, $eps:seq,
    $tau:V, $tau:E, $tau:delta)
    let $tau:V := eps:V($eps:res2)
    let $tau:E := eps:E($eps:res2)
    let $tau:delta := eps:delta($eps:res2)
    let $eps:val2 := eps:val($eps:res2)
    return eps:stValEnc($tau:V, $tau:E, $tau:delta,
    ( $eps:val1, $eps:val2 ))
  else eps:stValEnc($tau:V, $tau:E, $tau:delta,())
};
```

4.6 Creating a Constructor-Free Expression

We now sketch how to create constructor-free semi-equivalent expressions for deterministic (node-conservative) ones, i.e., how to generate the expression e' of Theorem 4.4, based on $\epsilon(e)$, which is working on an encoding of (St, Env). We do so by showing how encoding (St, Env) and afterwards decoding results St', v can be done for node-conservative expressions.

The expression $\epsilon(e)$ will be evaluated against $(St, \widehat{Env})$, where $\widehat{Env}$ contains the encoded store and environment. We construct $St_{\widehat{Env}}$ in such a way that it contains exactly all trees of St for which a node occurs in the variable bindings of Env. Assuming that we can have a sequence that is the concatenation of all variable bindings in Env, we can write an expression to create a new sequence `$roots` that, starting from the former sequence, filters out the nodes, applies the `root` function to each node and finally sorts this result by document order by applying a self-axis step. Since the roots of all trees that have to be in $St_{\widehat{Env}}$ are now in document order in `$roots`, we can write another expression that creates the encoded store $St_{\widehat{Env}}$ starting from an empty encoded store, by simply traversing through the trees under the nodes in `$roots` and extending $St_{\widehat{Env}} = (\widehat{V}, \widehat{E}, \widehat{\delta})$, represented by the variables `$tau:V`, `$tau:E` and `$tau:delta` in the environment $\widehat{Env}$). If this traversal is done in depth-first, left-to-right manner, we visit all nodes of St that will be encoded in $St_{\widehat{Env}}$ in document order. Node identifiers can then be chosen in such a way that they correspond to the position in $St_{\widehat{Env}}$. If we suppose the sequence of varbable bindings is bound to `$varBindings` the code could look like this

```
let $tau:V := ()
let $tau:E := ()
let $tau:delta := ()
let $roots := (
  for $var in $varBindings
  return root($var)
  )/.
return eps:encodeRoots($roots,1,$tau:V,$tau:E,$tau:delta)
```

Here `eps:encodeRoots()` is given in Function 4.31.

Starting from Env, we can now create the encoded environment $\widehat{Env}$ by replacing all expressions in b by the simulations $\epsilon(b)$, adding the variables for the encoded store and environment to the function signatures in a, replacing all sequences in the variable bindings with their encoded sequences, and finally, adding the variables `$tau:V`,`$tau:E` and `$tau:delta` to v. Since all nodes that occur in Env are encoded in $St_{\widehat{Env}}$ and node identifiers were assigned based on the position of nodes within the forest under `$roots`, we can easily obtain the encoded sequences for the variable bindings.

For all bounded variables `$b` we could write

```
let $b := eps:replaceBindings($b,$eps:roots,$tau:E)
```

Function 4.31 eps:encodeRoots

```
declare function eps:encodeRoots($eps:rootSequence,$eps:item,
    $tau:V,$tau:E,$tau:delta){
  if (fn:count($eps:rootSequence) >= $eps:item) then
    let $root := $eps:rootSequence[$eps:item]
    let $eps:res := eps:encodeRoot($root,$tau:V,$tau:E,$tau:delta)
    let $tau:V := eps:V($eps:res)
    let $tau:E := eps:E($eps:res)
    let $tau:delta := eps:delta($eps:res)
    return eps:encodeRoots($eps:rootSequence,$eps:item+1,
      $tau:V,$tau:E,$tau:delta)
  else
    eps:stValEnc($tau:V,$tau:E,$tau:delta,(0,0))

};
```

Here `eps:replaceBindings()` is given in Function 4.33.

The result of the evaluation of $\epsilon(e)$ is the store St and a store-value encoding $St_{\widehat{v}}$. Based on this we can create the result sequence the original expression returned if it was a node-conservative expression. In that case the encoded result sequence will only contain encoded nodes of which the real counterparts were available in the initial XML store St. Therefore we can loop over encoded items in $St_{\widehat{v}}$. Encodings of atomic values are simply replaced by the atomic values itself. For every encoded node we first determine whether it was originally in $St_{\widehat{Env}}$. This can be done by storing (during the encoding phase) all nodes and their chosen node identifiers as pairs in a variable. If the node identifier occurs in this variable then it is an original node and we can easily return the corresponding node. If the root of the encoded node is an encoded document node that is associated to a URI in the variable `$tau:delta` then we can obtain the original document root node by a simple `doc` function call, else it is a newly created node and hence this expression is not a node-conservative expression. By using the position of the encoded node relative to the encoded root node, we can determine the position of the corresponding real node in the document tree and hence we replace the encoded node by the real node in the result sequence.

If we assume that `$eps:finRes` is the final result of the expression this could be coded as

```
let $tau:V := eps:V($eps:finRes)
let $tau:E := eps:E($eps:finRes)
let $tau:delta := eps:delta($eps:finRes)
let $eps:finVal := eps:val($eps:finRes)
return eps:reverseReplaceBindings($eps:finVal,$eps:roots,
```

Function 4.32 eps:encodeRoot

```
declare function eps:encodeRoot($eps:node,
    $tau:V,$tau:E,$tau:delta){
  (: add node itself :)
  let $eps:v := (
    typeswitch($eps:node)
    case document-node()
      return ("document","","")
    case element()
      return ("element",fn:name($eps:node),"")
    case text()
      return ("text","", fn:data($eps:node))
    case attribute()
      return ("attribute",fn:name($eps:node), fn:data($eps:node))
    default return ()
    )
  let $eps:res2 := eps:addNode($eps:v[1],$eps:v[2],$eps:v[3],
    $tau:V,$tau:E,$tau:delta)
  let $tau:V := eps:V($eps:res2)
  let $tau:E := eps:E($eps:res2)
  let $eps:val2 := eps:val($eps:res2)
  (: if document-node add to delta :)
  let $tau:delta := (
    if ($eps:v[1] = "document") then
      (eps:delta($eps:res2),
      fn:document-uri($eps:node),
      $eps:val2[2])
    else
      eps:delta($eps:res2)
    )
  (: encode descendants :)
  let $eps:descendants :=
    ($eps:node/descendant-or-self::node()
    | $eps:node//@*)
  let $eps:res3 := eps:encodeDesc($eps:descendants,2,$eps:val2[2],
    ($eps:val2[2]),$tau:V,$tau:E,$tau:delta)
  let $tau:V := eps:V($eps:res3)
  let $tau:E := eps:E($eps:res3)
  let $tau:delta := eps:delta($eps:res3)
  return
    eps:stValEnc($tau:V,$tau:E,$tau:delta,(1,$eps:val2[2]))
};
```

Function 4.33 eps:replaceBindings

```
declare function eps:replaceBindings($eps:bindings,$eps:roots,
    $tau:E) {
  for $eps:binding in $eps:bindings
  return  eps:replaceBinding($eps:binding,$eps:roots, $tau:E)
};
```

Function 4.34 eps:replaceBinding

```
declare function eps:replaceBinding($eps:binding,$eps:roots,
    $tau:E) {
  let $eps:encRootIds := eps:getRootIds($tau:E)
  let $eps:root := root($eps:binding)
  let $eps:rootPos := (
    for $r at $pos in $eps:roots
    where $r is $eps:root
    return $pos
    )
  let $eps:bindPos := (
    for $b at $pos in ($eps:root/descendant-or-self::node()
      | $eps:root//@*)
    where $b is $eps:binding
    return $pos
    )
  let $eps:encRootId := (
    for $er at $pos in $eps:encRootIds
    where $pos = $eps:rootPos
    return $er
    )
  let $eps:encNodeId := (
    let $eps:desc := eps:decendants-o-s($tau:E, $eps:encRootId)
    for $en at $pos in $eps:desc
    where ($pos mod 2 = 0) and (($pos idiv 2) = $eps:bindPos)
    return $eps:desc[$pos]
    )
  return (1,$eps:encNodeId)
};
```

```
    $tau:E,$tau:delta)
```

Here `eps:reverseReplaceBindings()` is given in Function 4.35.

Function 4.35 eps:reverseReplaceBindings

```
declare function eps:reverseReplaceBindings($eps:bindings,$eps:roots,
    $tau:E,$tau:delta) {
  for $eps:binding in $eps:bindings
  return eps:reverseReplaceBinding($eps:binding,$eps:roots,
    $tau:E,$tau:delta)
};
```

Function 4.36 eps:reverseReplaceBinding

```
declare function eps:reverseReplaceBinding($eps:binding,
    $eps:roots, $tau:E,$tau:delta) {
  let $eps:encRootIds := eps:getRootIds($tau:E)
  let $eps:encRoot := eps:root($eps:binding[2],$tau:E)
  let $eps:encRootPos := (
    for $r at $pos in $eps:encRootIds
    where ($r = $eps:encRoot[2])
    return $pos
    )
  let $eps:encBindPos := (
    for $b at $pos in eps:decendants-o-s($tau:E, $eps:encRoot[2])
    where ($pos mod 2 = 0) and ($b = $eps:binding[2])
    return $pos idiv 2
    )
  return
    let $eps:root := (
      if ($eps:encRootPos <= count($eps:roots)) then
        (: the node was originally in the store :)
        for $er at $pos in $eps:roots
        where $pos = $eps:encRootPos
        return $er

      else
        (: the node was not in the original store, it must be
        a node from a loaded document, otherwise the expr
        is not node concervative :)
        let $eps:rootDocURI := eps:doc-uri($eps:encRoot[2],
          $tau:delta)
        return doc($eps:rootDocURI)
      )
    let $eps:node := (
      let $eps:desc := ($eps:root/descendant-or-self::node()
        | $eps:root//@*)
      for $en at $pos in $eps:desc
      where ($pos = ($eps:encBindPos))
      return $eps:desc[$pos]
      )
    return ($eps:node)

};
```

Chapter 5

Conclusion

In this work, we showed that deterministic XQuery expressions, always yielding a result with only nodes from the initial store, can be rewritten to equivalent expressions that do not contain node constructors.

In our approach we first restricted ourselves to an expressive fragment of XQuery called LiXQuery. LiXQuery is a fully downwards compatible sublanguage of XQuery and has almost the same expressive power as XQuery. It also has a compact and well defined syntax and formal semantics, which allows us to make precise an formal statments. It semantics included a store, which contained all the documents from the web (initial store) and fragments of xml which were created during the expression. With this formal semantics we defined the notion of a *node-conservative expression*, this type of expression always yields a result with only nodes from the initial store. Because our goal is to eliminate construction we had to restrict our types of expressions even further. A consequence of construction in LiXQuery is non-determinism, as the fragment that is created by a constructor can be placed at an arbitrary position in document order between the already existing trees in the store. Because this is not an fundamental XQuery feature we excluded non-deterministic expression. We then introduced the notion of a *simulation*. The semantics of our type of simulation differs in two ways from the semantics of the original expression. First our simulation may have a result when the original does not, and second, the result store of our simulation and that of the original are only the same up to *garbage collection*.

We then proved the following theorem: *For every deterministic node-conservative expression $e \in LQE$ there exists a simulation $e' \in LQE$ that does not contain constructors.* We did this by defining such a simulation based on a transformation function that transforms all LiXQuery expressions, and additional pre- and post-processing steps which encoded/decoded the store and environment into/from their simulated form.

5.1 Beyond node-conservatism

Because it is clear we cannot simulate an expression which returns new nodes by an expression that has no node construction, we restricted ourselves to the *node-conservative* *expressions*. But what if we changed our notion of a simulation a little. We could demand that a simulation must return a result sequence that is deep-equal to the result sequence of the original expression. We therefore formally define the notion of deep equallity similar to the one given in Definition 2.21 but now defined with different stores for the deep equal nodes:

Definition 5.1 (Different Store Deep Equal)**.** Given the XML stores $St_1 = (V_1, E_1, \ll_1, \nu_1, \sigma_1, \delta_1)$ and $St_2 = (V_2, E_2, \ll_2, \nu_2, \sigma_2, \delta_2)$ and the two nodes n_1 and n_2 in St_1 and St_2 respectively. n_1 and n_2 are said to be *deep equal*, denoted as $\mathbf{DpEq}_{St_1,St_2}(n_1, n_2)$, if n_1 and n_2 refer to two isomorphic trees, i.e., there is a one-to-one function $h : C_{n_1} \to C_{n_2}$ with $C_{n_i} = \{n|(n_i, n) \in E_i^*\}$ for $i = 1, 2$, such that for each $n, n' \in C_{n_1}$ it holds that (1) if $n \in \mathcal{V}_1^d$ ($\mathcal{V}_1^e$, $\mathcal{V}_1^a$, $\mathcal{V}_1^t$) then $h(n) \in \mathcal{V}_2^d$ ($\mathcal{V}_2^e$, $\mathcal{V}_2^a$, $\mathcal{V}_2^t$), (2) if $\nu_1(n) = s$ then $\nu_2(h(n)) = s$, (3) if $\sigma_1(n) = s'$ then $\sigma_2(h(n)) = s'$, (4) $(n, n') \in E_1$ iff $(h(n), h(n')) \in E_2$ and (5) if $n, n' \notin \mathcal{V}_1^a$ then $n \ll_1 n'$ iff $h(n) \ll_2 h(n')$.

With this new deep equality notion we can now define the new kind of simulation formally:

Definition 5.2. Given two expression $e, e' \in LQE$ we say that e' is a *semi-simulation* of e if for all stores St and environments Env with undefined x, k and m it holds that if $St, Env \vdash St', v$ then there exists a store St'' and sequence w such that $St, Env \vdash St'', w$ and $\mathbf{DpEq}_{St',St''}(v, w)$.

This creates some interesting possibilities. Let us first look into a slightly more broader class of expressions, namely those expressions that do not only have original nodes in their result sequence but can have copied nodes, as defined in Section 2.4, too. These expressions mainly will be using construction for intermediate restructuring (e.g. joins), we will call these *node-restructuring expressions* (NREs).

We will want to prove that for node-restructuring expression there also exists a equivalent expression that does not contain constructors. Again we will have to restrict ourselves to deterministic expressions, as non-determinism will not be possible in LiXQuery without the constructors. The definition of determinism introduced in Section 4.1 is much to strict, as it only allows for node-conservative expressions. We therefore introduce a looser definition of determinism:

Definition 5.3. An expression $e \in LQE$ is said to be *semi-deterministic* if for every store St and environment Env it holds that if $St, Env \vdash e \Rightarrow St', v$ and $St, Env \vdash e \Rightarrow St'', w$ then (St'', w) is isomorphic to (St', v), i.e., there exists a one-to-one function $f : V' \to V''$, with V' and V'' the nodes of St' and St'' respectively, such that if each occurrence of node $n \in St'$ in (St', v) is replaced by $f(n)$ we obtain (St'', w).

With this we can now formally define the theorem we will want to prove:

Theorem 5.4. *For every semi-deterministic node-restructuring expression $e \in LQE$, there exists a semi-simulation $e' \in LQE$ that does not contain constructors.*

To be able to prove this we construct such a simulation. We will not define formally how this simulation can be constructed. We will instaid informally discus the modifications that have to be made to the simulation used for the node-conservative expressions to be able to use it for node-restructuring expressions.

The first thing we have to do is add an extra special variable to the *store encoding*. This special variable, which we will refer to as $\widehat{C}$, will be structured like $\widehat{E}$ (which encodes pairs of node id's), but instead of encoding the parent-child relationship, it will encode the original-copy relationship. It must be passed onto and from every expression, like the other elements encoding the store, so we must also modify the *store-value encoding* to incorporate it.

Secondly, the transformation of the element and document constructor must be extended to add the appropriate information to this encoding, this comes down to an addition to the *eps:copyAsChildren()* function which they both use to copy the nodes.

Thirdly, the post-processing step discussed in Section 4.6 must be extended. With node-conservative expressions it was the case that only original nodes where to be found in the result sequence, and so the decoding was fairly straightforward. Garbage collection applied to the result store of the original expression would have removed all the newly created nodes from that store. All the nodes in our result sequence could be linked to a real node in a document immediately or after the loading of the associated document. With node-restructuring expressions the result sequence can also contain copied nodes, and as a consequence there will also still be copied nodes in the result store of the original expression after garbage collection. Our simulation on the other hand has not touched the store and it is still the initial store of our original expression. But as our original expression was node-restructuring all the nodes in the result sequence are original or copied nodes. What we are going to do is replace the copied nodes in the result sequence by the original node from which they were copied. An encoded node can be recognized as an copied node if its node id appears in $\widehat{C}$. We can then obtain the node id of its originating node using the information in $\widehat{C}$. We must be careful here however, because this originating node is not necessarily an original node, it can be a copy too. In generally we apply on this node the same test as if we would have found it in the result sequence. If it is an original node id, it will be replaced (possibly after loading it) by the original node, if it is again a copy, its originating node will be searched for and the process repeats itself until it stops with a final original node (if this is not the case it would have been a newly created node and the expression would not have been node-restructuring). As a result we obtain a result sequence with only original nodes which will be deep-equal to the result sequence of the original expression. It is clear hover that the result stores of the two expressions are not isomorphic as the result store of the original expression contains the created copies and the result store of the semi-simulation does not.

We must however stress the difference between a simulation and a semi-simulation. Any further expression evaluation on the result sequence returned by the semi-simulation will possibly return different results than the same expression evaluation applied to the

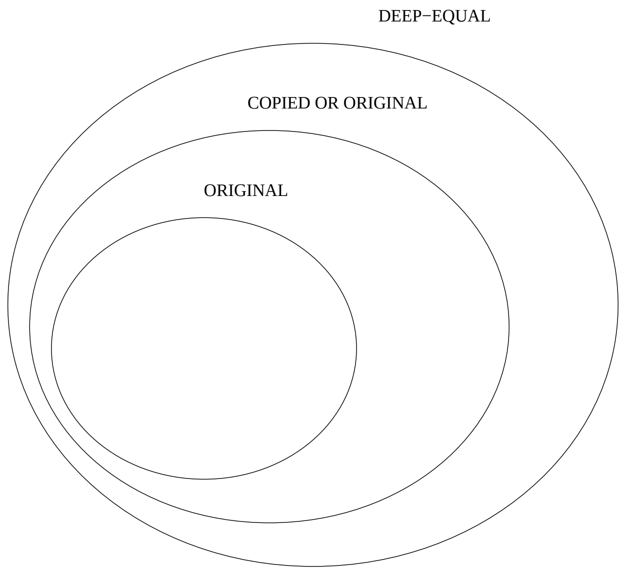

Figure 5.1: Subdivision of deep equal result nodes

result of the original expression. A node in the result sequence of the semi-simulation is only guaranteed to be deep-equal to the node at the same position in the original result sequence. For example this means that they may differ in their parent. So any application of the a parent axis step on this sequence possibly returns different results. If we want to use this extended result of our theorem we must be aware of this limitation.

An other use of this technique however could be the postponing of node construction. Instead of replacing the nodes in the result with the original nodes in the post-processing step we could as yet perform a construction and copy the nodes (taking into account the context). This would then yield an identical result (under the condition of deterministic expressions).

Now the question rises if we can extend our results even further. Node-restructuring expressions only allow for original and copied nodes to occur in their result sequence. Original and copied nodes are both part of a broader class of nodes in the result sequence of a LiXQuery expression we defined as deep equal nodes in Section 2.4. This is illustrated in Figure 5.1.

Nodes that are deep equal but were not copied, are created with the use of another mechanism. A simple example of such a mechanism is given in Example 5.1. Here a deep-equal copy of the node e is made without e itself being copied, only its attributes, element and text children.

This is just one of many ways to create such non-copied deep equal nodes, and many of them will not have such direct links with the node to which they are deep equal. This is where the problem lies if we want to replace them in a post-processing step with their deep equal original variants like we did with node-restructuring expressions. If we cannot

Example 5.1 A Deep Equally Constructed node

```
element {name($e)}{$e/@*, $e/(* | text())
```

make the link when they are created, an exhaustive search must be performed in the post-processing step. Of course this search can be limited to the nodes already in the store when the evaluation started and the documents loaded during the evaluation, making the it not impossible (looking at all the document in the web would be impossible), but rather highly impractical.

If we call these type of expression node-crafting expressions (NCrE), after their ability to craft new nodes without copying them directly, we can state the following theorem:

Theorem 5.5. *For every semi-deterministic node-crafting expression $e \in LQE$, there exists a semi-simulation $e' \in LQE$ that does not contain constructors.*

As said this semi-simulation can be created by starting of from the simulated created for node-restructuring expressions and extending its post-processing step. If eventually we come to a node in the result sequence which is not a copy of another node we perform an exhaustive search trough the encoded store for another node to which this node is deep equal (if such a node was not found the original expression was not node-crafting). We then replace our node with this node and perform on it the normal routine to see if its is an original or copied node. If it again seems to be deep equal we must take care not to consider the replaced node in our search otherwise we could go in a loop if two deep equal non-copied nodes existed.

Again this technique could also be used to postpone the node construction.

5.2 Future Work

In our approach these generated equivalent expression make extensive use of recursive functions. First of all to evaluate path-, for- and filter-expressions but also during the document call, where a document is encoded into our encoded store by traversing it recursively. For a practically usable approach this may not be feasible. The many recursive function calls make the execution of the simulated expression very slow. It is thus important from a practical but also from a theoretical perspective that further research investigates whether a similar result can also be obtained for non-recursive XQuery.

Furthermore, as mentioned in Section 5.1 our result can possibly be used for removing or postponing node creations in query evaluation plans. This can be a form of query optimization and further research can investigate what the applications are in this area.

Appendix A

Blixem LiXQuery BNF

This appendix contains a BNF for LiXQuery based on the abstract syntax of LiXQuery provided in [11] and that of XQuery provided in [4]. Please note that this BNF contains some special constructs due to limitations in the JavaCC tokenizer.

⟨mainModule⟩ ::= (⟨funDef⟩ ';')* ⟨expr⟩ 'eof'

⟨funDef⟩ ::= 'declare' 'function' ⟨qname⟩ '(' (⟨var⟩ (',' ⟨var⟩)*)? ')' '{' ⟨expr⟩ '}'

⟨expr⟩ ::= ⟨singleExpr⟩ (',' ⟨expr⟩)?

⟨singleExpr⟩ ::= ⟨ifExpr⟩
  | ⟨forExpr⟩
  | ⟨letExpr⟩
  | ⟨typeSw⟩
  | ⟨orExpr⟩

⟨builtIn⟩ ::= ('doc(' ⟨singleExpr⟩ ')' | 'name(' ⟨singleExpr⟩ ')' | 'string(' ⟨singleExpr⟩ ')' | 'integer(' ⟨singleExpr⟩ ')' | 'root(' ⟨singleExpr⟩ ')' | 'concat(' ⟨singleExpr⟩ ',' ⟨singleExpr⟩ ')' | 'true()' | 'false()' | 'not(' ⟨singleExpr⟩ ')' | 'count(' ⟨singleExpr⟩ ')' | 'position()' | 'last()')

⟨ifExpr⟩ ::= 'if' '(' ⟨singleExpr⟩ ')' 'then' ⟨singleExpr⟩ 'else' ⟨singleExpr⟩

⟨forExpr⟩ ::= 'for' ⟨var⟩ ('at' ⟨var⟩)? 'in' ⟨singleExpr⟩ ('return' ⟨singleExpr⟩ | ⟨forExpr⟩ | ⟨letExpr⟩)

⟨letExpr⟩ ::= 'let' ⟨var⟩ 'assign' ⟨singleExpr⟩ ('return' ⟨singleExpr⟩ | ⟨forExpr⟩ | ⟨letExpr⟩)

⟨orExpr⟩ ::= ⟨andExpr⟩ ('or' ⟨andExpr⟩)*

⟨andExpr⟩ ::= ⟨equalExpr⟩ ('and' ⟨equalExpr⟩)*

$\langle \mathit{equalExpr} \rangle$ ::= $\langle \mathit{lessExpr} \rangle$ (‘`equal`’ $\langle \mathit{lessExpr} \rangle$)*

$\langle \mathit{lessExpr} \rangle$ ::= $\langle \mathit{isExpr} \rangle$ (‘`less`’ $\langle \mathit{isExpr} \rangle$)*

$\langle \mathit{isExpr} \rangle$ ::= $\langle \mathit{nodeCmpExpr} \rangle$ (‘`is`’ $\langle \mathit{nodeCmpExpr} \rangle$)*

$\langle \mathit{nodeCmpExpr} \rangle$ ::= $\langle \mathit{addExpr} \rangle$ (‘`nodecmp`’ $\langle \mathit{addExpr} \rangle$)*

$\langle \mathit{addExpr} \rangle$ ::= $\langle \mathit{subExpr} \rangle$ (‘`plus`’ $\langle \mathit{subExpr} \rangle$)*

$\langle \mathit{subExpr} \rangle$ ::= $\langle \mathit{multExpr} \rangle$ (‘`min`’ $\langle \mathit{multExpr} \rangle$)*

$\langle \mathit{multExpr} \rangle$ ::= $\langle \mathit{divExpr} \rangle$ (‘`star`’ $\langle \mathit{divExpr} \rangle$)*

$\langle \mathit{divExpr} \rangle$ ::= $\langle \mathit{union} \rangle$ (‘`div`’ $\langle \mathit{union} \rangle$)*

$\langle \mathit{union} \rangle$ ::= $\langle \mathit{path} \rangle$ (‘`union`’ $\langle \mathit{union} \rangle$)*

$\langle \mathit{path} \rangle$ ::= $\langle \mathit{filter} \rangle$ ((‘`/`’ $\langle \mathit{path} \rangle$ | ‘`//`’ $\langle \mathit{path} \rangle$))?

$\langle \mathit{filter} \rangle$ ::= $\langle \mathit{step} \rangle$ (‘`[`’ $\langle \mathit{singleExpr} \rangle$ ‘`]`’)*

$\langle \mathit{step} \rangle$ ::= (‘`.`’ | ‘`..`’ | ‘`@`’ $\langle \mathit{qname} \rangle$ | ‘`*`’ | ‘`@*`’ | ‘`text`’ | $\langle \mathit{primaryExpr} \rangle$)

$\langle \mathit{primaryExpr} \rangle$ ::= $\langle \mathit{builtIn} \rangle$
  | $\langle \mathit{funCall} \rangle$
  | $\langle \mathit{qname} \rangle$
  | $\langle \mathit{constr} \rangle$
  | $\langle \mathit{var} \rangle$
  | $\langle \mathit{literal} \rangle$
  | $\langle \mathit{empSeq} \rangle$
  | ‘`(`’ $\langle \mathit{expr} \rangle$ ‘`)`’

$\langle \mathit{literal} \rangle$ ::= $\langle \mathit{string} \rangle$
  | $\langle \mathit{integer} \rangle$

$\langle \mathit{string} \rangle$ ::= ‘`string`’

$\langle \mathit{integer} \rangle$ ::= ((‘`digits`’ | ‘`plus`’ ‘`digits`’) | (‘`min`’ ‘`digits`’))

$\langle \mathit{var} \rangle$ ::= ‘`dollar`’ $\langle \mathit{qname} \rangle$

$\langle \mathit{empSeq} \rangle$ ::= ‘`(`’ ‘`)`’

$\langle \mathit{constr} \rangle$ ::= (‘`element`’ ‘`{`’ $\langle \mathit{singleExpr} \rangle$ ‘`}`’ ‘`{`’ $\langle \mathit{expr} \rangle$ ‘`}`’ | ‘`attribute`’ ‘`{`’ $\langle \mathit{singleExpr} \rangle$ ‘`}`’ ‘`{`’ $\langle \mathit{expr} \rangle$ ‘`}`’ | ‘`text`’ ‘`{`’ $\langle \mathit{singleExpr} \rangle$ ‘`}`’ | ‘`document`’ ‘`{`’ $\langle \mathit{singleExpr} \rangle$ ‘`}`’)

$\langle$*typeSw*$\rangle$ ::= ‘`typeswitch`’ ‘`(`’ $\langle$*singleExpr*$\rangle$ ‘`)`’ (‘`case`’ $\langle$*type*$\rangle$ ‘`return`’ $\langle$*singleExpr*$\rangle$)+
‘`default`’ ‘`return`’ $\langle$*singleExpr*$\rangle$

$\langle$*type*$\rangle$ ::= (‘`typebool`’ | ‘`typeint`’ | ‘`typestring`’ | ‘`elementf`’ | ‘`attributef`’ | ‘`textf`’ |
‘`documentnodef`’)

$\langle$*funCall*$\rangle$ ::= $\langle$*qname*$\rangle$ ‘`(`’ ($\langle$*singleExpr*$\rangle$ (‘`,`’ $\langle$*singleExpr*$\rangle$)*)? ‘`)`’

$\langle$*qname*$\rangle$ ::= ‘`ncname`’

Bibliography

[1] Extensible markup language (XML). `http://www.w3.org/XML/`.

[2] Extensible markup language (XML) 1.1. `http://www.w3.org/TR/2004/REC-xml11-20040204/`.

[3] Namespaces in XML 1.1. `http://www.w3.org/TR/2004/REC-xml-names11-20040204/`.

[4] XML query (XQuery). `http://www.w3.org/TR/2005/WD-xquery-20050404/`.

[5] Xquery 1.0 and xpath 2.0 data model, w3c working draft 23 july 2004. http://www.w3.org/tr/2004/wd-xpath-datamodel-20040723/.

[6] Xquery 1.0 and xpath 2.0 formal semantics, w3c working draft 20 february 2004. http://www.w3.org/tr/2004/wd-xquery-semantics-20040220/.

[7] Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45:798–842, September 1998.

[8] Michael Brundage. *XQuery: The XML Query Language.* Addison-Wesley, 2004.

[9] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL hosts. In *Proceedings of the 30th Int'l Conference on Very Large Databases (VLDB 2004)*, August/September 2004 2004.

[10] Jan Hidders, Jan Paredaens, Philippe Michiels, and Roel Vercammen. LiXQuery: A formal foundation for XQuery research. *SIGMOD Record*, September 2005.

[11] Jan Hidders, Jan Paredaens, Roel Vercammen, and Serge Demeyer. A light but formal introduction to XQuery. In *Proceedings of the Second International XML Database Symposium (XSym 2004)*, Toronto, Canada, 2004. Springer.

[12] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler, editors. *XQuery from the Experts: A Guide to the W3C XML Query Language.* Addison-Wesley, 2004.

[13] Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems (TODS)*, 17:65–93, 1992.

[14] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254:363–377, 2001.

[15] Jan Van den Bussche, Dirk Van Gucht, Marc Andries, and Marc Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44:272–319, March 1997.