On the Expressive Power of XQuery-based Update Languages

Jan Hidders, Jan Paredaens, and Roel Vercammen*

University of Antwerp

Abstract. XQuery 1.0, the XML query language which is about to become a W3C Recommendation, lacks the ability to make persistent changes to instances of its data model. A number of proposals to extend XQuery with update facilities have been made lately, including a W3C Working Draft. In order to investigate some of the different constructs that are introduced in these proposals, we define an XQuery-based update language that combines them. By doing so, we show that it is possible to give a concise, complete and formal definition of such a language. We define subsets of this language to examine the relative expressive power of the different constructs, and we establish the relationships between these subsets in terms of queries and updates that can be expressed. Finally, we discuss the relationships between these subsets and existing XQuery-based update languages.

1 Introduction

With the growing acceptance of XQuery as the main query language for XML data, there has also been a growing need for an extension that allows updates. This has lead to several proposals such as [11], [9], UpdateX [10, 1], XQuery! [3] and the XQuery Update Facility [2]. Next to introducing operations for manipulating nodes such as inserting and deleting, these proposals often also introduce special operations such as the snap operation (in XQuery!) and the transform operation (in XQuery Update Facility) to extend the expressive power of the language, sometimes for queries as well as updates. For example, the snap operation allows us write queries in XQuery that use side effects and bounded iteration. Another example is the transform operation that allows us to concisely express a transformation that copies an entire tree and makes a few minor changes to it. In this paper we investigate the relative expressive power of such constructs for expressing queries as well as updates. In addition we examine the strict separation of expressions in updating and non-updating expressions, and determine whether this influences the ability to express certain queries and updates.

To investigate the mentioned questions we define LiXQuery$^+$ by taking LiXQuery[7], a concise and formally defined subset of XQuery, and extending it with all the mentioned constructs.

* Roel Vercammen is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant number 33581.

The remainder of this paper is organized as follows. Section 2 presents the syntax of LiXQuery$^+$ and discusses its semantics informally. Section 3 presents the formal framework necessary for defining the semantics. Section 4 defines the semantics of expressions in LiXQuery$^+$. Section 5 presents the results on the expressive power of the different constructs [1] . Section 6 relates these results to existing proposals in the literature and finally Section 7 contains the conclusion.

2 Syntax and Informal Semantics

Due to space limitations, we do not give the complete LiXQuery$^+$ syntax, but only show how to extend the LiXQuery grammar to obtain LiXQuery$^+$. We start from the grammar as given in [4], remove the start symbol ⟨*Query*⟩ and introduce a new start symbol ⟨*Program*⟩, which is a sequence of variable and function declarations followed by an expression. The syntax of LiXQuery$^+$ programs is given in Fig. 1 as an abstract syntax, i.e., it assumes that extra brackets and precedence rules are added for disambiguation. The ellipsis in the non-terminal ⟨*Expr*⟩ refer to the right-hand side of this non-terminal in the LiXQuery grammar. The XQuery features that we can express in non-recursive LiXQuery include FLWOR-expressions, path expressions, typeswitches, node and value comparisons, sequence generations (using the "`to`"-operation), sequence concatenation, and some simple arithmetic.

⟨*Program*⟩ ::= ((⟨*VarDecl*⟩ | ⟨*FunDecl*⟩) "`;`")* ⟨*Expr*⟩
⟨*VarDecl*⟩ ::= "`declare`" "`variable`" ⟨*Var*⟩ "`:=`" ⟨*Expr*⟩
⟨*FunDecl*⟩ ::= "`declare`" "`function`" ⟨*Name*⟩ "`(`" ((⟨*Var*⟩ ("`,`" ⟨*Var*⟩)*)? "`)`" "`{`" ⟨*Expr*⟩ "`}`"
⟨*Expr*⟩ ::= ... | ⟨*Insert*⟩ | ⟨*Rename*⟩ | ⟨*Replace*⟩ | ⟨*Delete*⟩ | ⟨*Snap*⟩ | ⟨*Transform*⟩
⟨*Insert*⟩ ::= "`insert`" ⟨*Expr*⟩ ("`into`" | "`before`" | "`after`") ⟨*Expr*⟩
⟨*Rename*⟩ ::= "`rename`" ⟨*Expr*⟩ "`as`" ⟨*Expr*⟩
⟨*Replace*⟩ ::= "`replace`" "`value`" "`of`" ⟨*Expr*⟩ "`with`" ⟨*Expr*⟩
⟨*Delete*⟩ ::= "`delete`" ⟨*Expr*⟩
⟨*Snap*⟩ ::= "`snap`" (("`unordered`" ("`nondeterministic`" | "`deterministic`")) | "`ordered`") "`{`" ⟨*Expr*⟩ "`}`"
⟨*Transform*⟩ ::= "`transform`" "`copy`" ⟨*Var*⟩ "`:=`" ⟨*Expr*⟩ "`modify`" ⟨*Expr*⟩ "`return`" ⟨*Expr*⟩

Fig. 1. Syntax of LiXQuery$^+$

We assume the reader is already familiar with XQuery. We therefore only describe the semantics of the new expressions and sketch the modifications to the semantics of the other expressions.

We first describe the semantics of the update expressions, i.e., the "`insert`", "`rename`", "`replace`" and "`delete`" operations. The "`insert`" operation makes a copy of the nodes in the result of the first expression and adds these (afterwards) at the position that is indicated by either "`into`", "`before`", or "`after`" and which is relative to the singleton result node of the second expression. The "`rename`" operation renames an element or an attribute, and the "`replace`" operation replaces the value of a text or an attribute node with a new atomic value. Both operations are node-identity preserving, i.e., the identity of the updates node is not changed. The "`delete`" expression removes the incoming edges

[1] We only give sketches of the proofs, for the full proofs we refer to [5]

for a set of nodes, which can then be garbage collected iff they are not accessible anymore through variable bindings or the result sequence.

For most expressions we assume a snapshot semantics, which intuitively means that a snapshot of the store is being made before the evaluation of the expression and the resulting updates are not yet performed, but instead they are added to a list of pending updates. There are four exceptions to this: the "`snap`" operation, expressions at the end of a program, expressions at the right-hand side of a variable declaration and the "`transform`" expression. We discuss these four cases in the following.

A "`snap`" operation applies the list of pending updates that is generated by the subexpression to the store and returns an empty update list. If the snap expression contains the keyword "`ordered`", then the pending updates are applied in the same order as they were generated. Else the order of application is undefined and the keywords "`deterministic`" and "`nondeterministic`" specify whether the order of the application of pending updates is allowed to affect the set of possible result stores. As an illustration of the "`snap`" expression consider:

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

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

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

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

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

This expression retrieves all information about the security department except the subtrees which have a security level higher than three. Note that the transform operation cannot update an existing fragment in the XML store.

Finally, all other operations were already in LiXQuery and their semantics is now extended in such a way that the result is not only the result sequence, but also the concatenation of all lists of pending updates that were generated during the evaluations of subexpressions.

3 Formal Framework

We now proceed with the formal semantics of LiXQuery$^+$. Due to space limitations, we will not fully introduce all concepts of LiXQuery here, but refer to [7] for some examples and a more elaborated introduction. We assume a set of *strings* $\mathcal{S}$ and a set of *names* $\mathcal{N} \subseteq \mathcal{S}$, which contains those strings that may be used as tag names. The set of all atomic values is denotes by $\mathcal{A}$ and is a superset of $\mathcal{S}$. We also assume four countably infinite sets of nodes $\mathcal{V}^d$, $\mathcal{V}^e$, $\mathcal{V}^a$ and $\mathcal{V}^t$ which respectively represent the set of *document*, *element*, *attribute* and *text nodes*. These sets are pairwise disjoint with each other and the set of atomic values. The set of all nodes is denoted as $\mathcal{V}$, i.e., $\mathcal{V} = \mathcal{V}^d \cup \mathcal{V}^e \cup \mathcal{V}^a \cup \mathcal{V}^t$. In the rest of this paper, we use the following notation: 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, e for expressions and p for programs. We denote the empty sequence as $\langle\rangle$, non-empty sequences as for example $\langle 1, 2, 3\rangle$ and the concatenation of two sequences l_1 and l_2 as $l_1 \circ l_2$. Finally, if l is a list or sequence, then the set of items in l is denoted as $\mathbf{Set}(l)$ and the bag (unordered list) representation of l is denoted by $\mathbf{Bag}(l)$.

3.1 XML Store

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

Definition 1 (XML Store). *An XML store is a 6-tuple* $St = (V, E, \ll, \nu, \sigma, \delta)$*:*

- V *is a finite subset of* $\mathcal{V}$[2]*;*
- (V, E) *is a directed acyclic graph where each node has an in-degree of at most one, and hence it is composed of trees; if* $(m, n) \in E$ *then we say that* n *is a child of* m*; we denote by* E^* *the reflexive transitive closure of* E*;*
- $\ll$ *is a total order on the nodes of* V*;*
- $\nu : V^e \cup V^a \to \mathcal{N}$ *labels element and attribute nodes with their* node name*;*
- $\sigma : V^a \cup V^t \to \mathcal{S}$ *labels attribute and text nodes with their* string value*;*
- $\delta : \mathcal{S} \to \mathcal{V}^d$ *a partial function that associates a URI with a document node.*

Moreover, some additional properties must hold for such a tuple in order to be a valid XML store. We refer to the technical report [6] on LiXQuery$^+$ for these properties.

Note that this definition slightly differs from our original definition of an XML Store [7], since we now have included the document order in the store instead of the sibling order. In the rest of this paper we will write V_{St} to denote the set of nodes of the store St, and similarly we write E_{St}, $\ll_{St}$, ν_{St}, σ_{St} and δ_{St} to denote respectively the second to the sixth component of the 6-tuple St.

[2] We write V^d to denote $V \cap \mathcal{V}^d$, and use a similar notation for V^e, V^a, and V^t

3.2 Evaluation Environment

Expressions are evaluated against an environment. Assuming that $\mathcal{X}$ is the set of LiXQuery$^+$-expressions this environment is defined as follows.

Definition 2 (Environment). *An environment of an XML store St is a tuple* $En = (\mathbf{a}, \mathbf{b}, \mathbf{v}, \mathbf{x})$ *with a partial function* $\mathbf{a} : \mathcal{N} \to \mathcal{N}^*$ *that maps a function name to its formal arguments, a partial function* $\mathbf{b} : \mathcal{N} \to \mathcal{X}$ *that maps a function name to the body of the function, a partial function* $\mathbf{v} : \mathcal{N} \to (\mathcal{V} \cup \mathcal{A})^*$ *that maps variable names to their values, and* $\mathbf{x}$ *which is either undefined* ($\perp$) *or an item of St and indicates the context item.*

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

3.3 List of Pending Updates

A new concept in the LiXQuery$^+$ semantics, when compared to LiXQuery, is the list of pending updates. This list contains a number of primitive update operations which have to be performed after the evaluation of the entire expression.

Definition 3 (Primitive Update Operations). *Let* $n, n_1, \ldots, n_m$ *be nodes in a store St, and* $s \in \mathcal{S}$. *A primitive update operation on the store St is one of following operations:* $insBef(n, \langle n_1, \ldots, n_m \rangle)$, $insAft(n, \langle n_1, \ldots, n_m \rangle)$, $insInto(n, \langle n_1, \ldots, n_m \rangle)$, $ren(n, s)$, $repVal(n, s)$, $del(n)$.

Before proceeding with the formal semantics, we first give some intuition about these primitive update operations. The operation $insBef$ ($insAft$, $insInto$) moves nodes n_1 to n_m before (after, into) the node n. In the formal semantics of LiXQuery$^+$, we will see that the nodes n_1 to n_m are always copies of other nodes. Note that the operation $insInto$ can have several result stores, since the list of nodes can be inserted in an arbitrary position among the children. The operations ren and $repVal$ change respectively the name and the value of n to s. Finally, the operation del removes the incoming edge from n and hence detaches the subtree rooted at n. Note that del can, similar to $insInto$, have more than one result store, due to the resulting document order. More precisely, the subtree that is detached by a del operation has to be given another place in document order, since otherwise this tree would be mixed in document order with the tree from which we deleted the edge, which is a violation of one of the additional properties of Definition 1. The exact location in document order of the detached subtree is chosen in a non-deterministic manner.

We write $St \vdash o \Rightarrow^{\mathtt{U}} St'$ to denote that applying the primitive update operation o to St can result in the store St'. The definition of $\Rightarrow^{\mathtt{U}}$ is given in Fig. 2 by means of inference rules. Each rule consists of a set of premises and a conclusion

$$\frac{St' = St[\nu(n) \mapsto s]}{St \vdash ren(n,s) \Rightarrow^{\mathtt{U}} St'}$$

$$\frac{St' = St[\sigma(n) \mapsto s]}{St \vdash repVal(n,s) \Rightarrow^{\mathtt{U}} St''}$$

$$\frac{St' = (St \setminus n) \cup St[n]}{St \vdash del(n) \Rightarrow^{\mathtt{U}} St'}$$

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

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

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

Fig. 2. Semantics of the Primitive Update Operations.

of the form $St \vdash o \Rightarrow^{\mathtt{U}} St'$. The free variables in the rules are always assumed to be universally quantified. In these rules we use some additional notations, which we will now explain.

Let St be a store and n an element of V_{St}. We define V^n_{St} as $\{n' | (n,n') \in E^*_{St}\}$, i.e., the set of nodes in the subtree rooted at n in St. The projection of St to a set of nodes N is denoted by $\Pi_N(St)$ and is the restriction of all components of St to N instead of V_{St}. The restriction of St to n is defined as $\Pi_{V^n_{St}}(St)$ and is denoted by $St[n]$. The exclusion of n from St is defined as $\Pi_{V_{St}-V^n_{St}}(St)$ and is denoted by $St \setminus n$. For both restriction and exclusion it is not hard to see that the projection always results in a store. Finally, if St is a store, n a node in St, and s a string, then we let $St[\delta(n) \mapsto s]$ ($St[\nu(n) \mapsto s]$) denote the store that is equal to St except that $\delta_{St'}(n) = s$ ($\nu_{St'}(n) = s$).

We now define *a list l of pending updates over a store St* as a list of primitive update operations on St. The set of affected nodes of l is denoted by $\mathbf{Targets}(l)$ and defined as the set of nodes that occur as the first argument in a primitive update operation appearing in l.

The notation $St \vdash o \Rightarrow^{\mathtt{U}} St'$, used to specify the semantics of primitive update operations, is overloaded for sequences of primitive update operations. For such a sequence $l = \langle o_1, \ldots, o_m \rangle$ we define $St \vdash l \Rightarrow^{\mathtt{U}} St'$ by induction on m such that (1) $St \vdash \langle\rangle \Rightarrow^{\mathtt{U}} St$ and (2) if $St \vdash \langle o_1, \ldots, o_{m-1} \rangle \Rightarrow^{\mathtt{U}} St'$ and $St' \vdash o_m \Rightarrow^{\mathtt{U}} St''$ then $St \vdash \langle o_1, \ldots, o_m \rangle \Rightarrow^{\mathtt{U}} St''$.

For some lists of pending updates, we can reorder the application of these primitive update operations without changing the semantics. Therefore we say that l is *execution-order independent* if for every sequences l' such that $\mathbf{Bag}(l) = \mathbf{Bag}(l')$ and store St' it holds that $St \vdash l \Rightarrow^{\mathtt{U}} St'$ iff $St \vdash l' \Rightarrow^{\mathtt{U}} St'$.

Finally, the following lemma gives an algorithm to decide execution-order independence of a list of pending updates:

Lemma 1. *A list of pending updates $l = \langle o_1, \ldots, o_m \rangle$ over a store St is execution-order dependent iff there are two primitive update operations o_i and o_j in l such that $i \neq j$, and there are $n, n_1, \ldots, n_m, n'_1, \ldots, n'_l \in V_{St}$ and $s, s' \in \mathcal{S}$, such that $s \neq s'$, $\langle n_1, \ldots, n_m \rangle \neq \langle n'_1, \ldots, n'_l \rangle$ and one of the following holds:*
- $o_i = ren(n, s) \wedge o_j = ren(n, s')$
- $o_i = repVal(n, s) \wedge o_j = repVal(n, s')$
- $o_i = insBef(n, \langle n_1, \ldots, n_m \rangle) \wedge o_j = insBef(n, \langle n'_1, \ldots, n'_l \rangle)$
- $o_i = insAft(n, \langle n_1, \ldots, n_m \rangle) \wedge o_j = insAft(n, \langle n'_1, \ldots, n'_l \rangle)$

3.4 Program Semantics

We now define the semantics of programs. We write $(St, En) \vdash p \Rightarrow (St', v)$ to denote that the *program* p, evaluated against the XML store St and environment En of St, can result in the new XML store St' and value v of St'. Similarly, $(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l)$ means that the evaluation of *expression* e against St and En may result in St', v, and the list of pending updates l over St'. The semantics of expressions is given in Section 4. Finally, the semantics of a program is defined by following reasoning rules:

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

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

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

Note that in the last rule, v is a value of St'', since $V_{St'} = V_{St''}$.

3.5 Auxiliary Notions

We conclude this section by giving some notational tools for the rest of this paper. First, we define some auxiliary operations on stores.

Two stores St and St' are disjoint, denoted as $St \cap St' = \emptyset$, iff $V_{St} \cap V_{St'} = \emptyset$. The definition of the *union* of two disjoint stores St and St', denoted as $St \cup St'$, is straightforward. The resulting document order is extended to a total order in a nondeterministic way.

An *item* of an XML store St is an atomic value in $\mathcal{A}$ or a node in St. Given a sequence of nodes l in an XML store St we let $\mathbf{Ord}_{St}(l)$ denote the unique sequence $l' = \langle y_1, \ldots, y_m \rangle$ such that $\mathbf{Set}(l) = \mathbf{Set}(l')$ and $y_1 \ll_{St} \ldots \ll_{St} y_m$.

Two trees defined by two nodes n_1 and n_2 in a store St can be equal up to node identity, in which case we say that they are *deep equal* and denote this as $\mathbf{DpEq}_{St}(n_1, n_2)$.

4 Semantics of Expressions

Similar to LiXQuery, the semantics of LiXQuery$^+$ expressions is specified by means of inference rules. Each rule consists of a set of premises and a conclusion

of the form $(St, En) \vdash e \Rightarrow^{\mathtt{E}} (St', v, l)$. The free variables in the rules are assumed to be universally quantified. Due to the lack of space we only give the rules for the expressions that are new in LiXQuery$^+$ and illustrate how the other rules can be obtained.

4.1 Basic Update Expressions

The `delete` results into a set of pending updates which will delete the incoming edges of the selected nodes.

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

The `rename` and `replace value` expressions evaluate two subexpressions which have to result in respectively one node and one string value. Similar to the `delete` expression we add new primitive operation to the list of pending updates. For the exact inference rules we refer to the technical report[6]. An `insert` expression makes a copy of the nodes that are selected by the first subexpression and puts these copies at a certain place w.r.t. the node that is returned by the second expression. The position is indicated by either "`before`", "`after`", or "`into`". In case of insertion into a node n, the relative place of the copied nodes among the children of n is chosen arbitrarily, but the relative order of the copies has to be preserved. We show the semantics for the "`insert` ... `into` ..." expression.

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

4.2 Snap Expression

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

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

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

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

4.3 Transform Expression

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

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

4.4 Other Expressions

To illustrate the semantics of expressions already in LiXQuery we present the reasoning rules for the concatenation and the `for`-expression. The other rules can be obtained from those in [7] in a similar way by extending them such that the lists of pending updates of the subexpressions are concatenated.

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

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

5 Expressive Power of Fragments of LiXQuery$^+$

In this section we compare the relative expressive power of a number of constructs of LiXQuery$^+$ by looking at different fragments of the language that do or do not contain these constructs.

5.1 LiXQuery$^+$ Fragments

The motivation for these fragments follows by their correspondence to existing query and update languages for XML, based on XQuery. In Section 6 we discuss the relation between these fragments and the existing update languages. First, we define the following four fragments of LiXQuery$^+$:

– The fragment XQ corresponds intuitively to non-recursive XQuery. More precisely, the non-terminals $\langle Insert \rangle$, $\langle Rename \rangle$, $\langle Replace \rangle$, $\langle Delete \rangle$, $\langle Snap \rangle$, and $\langle Transform \rangle$ are removed from LiXQuery$^+$, as well as $\langle FunDecl \rangle$.
– The fragment XQ_t corresponds to non-recursive XQuery extended with transformations. It is defined as LiXQuery$^+$ without $\langle Snap \rangle$ and $\langle FunDecl \rangle$ and where $\langle Insert \rangle$, $\langle Rename \rangle$, $\langle Replace \rangle$, and $\langle Delete \rangle$ only occurs within the body of the "`modify`" clause of a $\langle Transform \rangle$ expression.
– The fragment XQ_+ corresponds to non-recursive XQuery extended with the update operations, but without the $\langle Snap \rangle$ operation. It is defined as LiXQuery$^+$without $\langle Snap \rangle$ and $\langle FunDecl \rangle$.
– The fragment $XQ_!$ corresponds to non-recursive XQuery extended with updates and snap operations. It is defined as LiXQuery$^+$ without $\langle FunDecl \rangle$.

We can add (recursive) function definitions to all these fragments, which we denote by adding a superscript R to the name of the fragments.

5.2 Expressiveness Relations between Fragments

It seems intuitive to say that two programs express the same update function if they map the same input stores to the same output stores. However, a program can make changes to the store that cannot be observed, since the modified nodes are not reachable through the result sequence of the program or through document calls. Therefore, we assume that the result store of a program does not contain nodes that are no longer reachable, since such nodes can be safely garbage collected. More precisely, the garbage collection is defined by the function Γ_v that, given a sequence v, maps a store St to a new store St' by removing all trees from St for which the root node is not in the range of δ_{St} and for which no node of the tree is in v.

We now define the query and update relations that correspond to LiXQuery$^+$ programs. Since programs can return sequences over another store than the input store, we only consider mappings from a store to a sequence of *atomic values* in this paper, i.e., we only look at queries that do not return nodes. The *query relation* of a LiXQuery$^+$ program p is the relation $\mathcal{R}_p^Q$ between stores St and sequences of *atomic values* v such that $(St, v) \in \mathcal{R}_p^Q \Leftrightarrow \exists St' : (St, (\emptyset, \emptyset, \emptyset, \bot)) \vdash p \Rightarrow (St', v)$. The *update relation* of a LiXQuery$^+$ program p is the relation $\mathcal{R}_p^U$ between stores St and St' such that $(St, St') \in \mathcal{R}_p^U \Leftrightarrow \exists St'', v : (St, (\emptyset, \emptyset, \emptyset, \bot)) \vdash p \Rightarrow (St'', v) \wedge \Gamma_v(St'') = St'$.

The following two partial orders are defined on LiXQuery$^+$ fragments:
- $XF_1 \succeq^Q XF_2$ iff $\forall p \in XF_2 : \exists p' \in XF_1 : \mathcal{R}_p^Q = \mathcal{R}_{p'}^Q$.
- $XF_1 \succeq^U XF_2$ iff $\forall p \in XF_2 : \exists p' \in XF_1 : \mathcal{R}_p^U = \mathcal{R}_{p'}^U$.

Based on these partial orders $\succeq^Q$ and $\succeq^U$ we define in the usual way the strict partial orders $\succ^Q$ and $\succ^U$, and the equivalence relations $\equiv^Q$ and $\equiv^U$ which are called *query-equivalence* and *update-equivalence*, respectively. Note that $XF_1 \not\equiv^Q XF_2 \Rightarrow XF_1 \not\equiv^U XF_2$, since we can translate in all fragments a result sequence of atomic values to a node containing a sequence of nodes that each contain one of the atomic values, and vice versa.

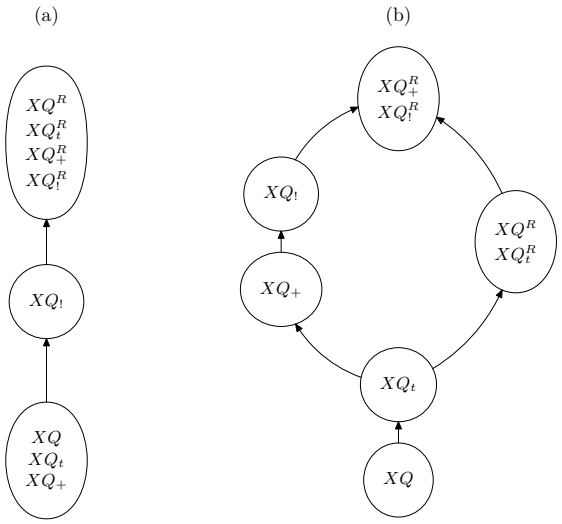

Fig. 3. Relations between the fragments in terms of expressive power of (a) mappings from stores to sequences of atomic values and (b) mappings from stores to stores.

Theorem 1. *For the graph in part (a) of Fig. 3 and for all fragments* XF_1, XF_2 *it holds that* $XF_1 \equiv^Q XF_2 \iff XF_1$ *and* XF_2 *are within the same node, and* $XF_1 \succ^Q XF_2 \iff$ *there is a directed path from the node containing* XF_2 *to the node containing* XF_1*.*

Proof. (Sketch) This theorem will be proven in the remainder of this section. We now sketch which lemmas are needed to complete this proof. From Lemma 4 it follows that $XQ \equiv^Q XQ_t \equiv^Q XQ_+$, and from Lemma 2, Lemma 3, and Lemma 5 it follows that $XQ^R \equiv^Q XQ_t^R \equiv^Q XQ_+^R \equiv^Q XQ_!^R$. From Lemma 7 and Lemma 8 follows that $XQ_! \succ^Q XQ_+$ and from Lemma 9 follows that $XQ_!^R \succ^Q XQ_!$.

Theorem 2. *For the graph in part (b) of Fig. 3 and for all fragments* XF_1, XF_2 *it holds that* $XF_1 \equiv^U XF_2 \iff XF_1$ *and* XF_2 *are within the same node, and* $XF_1 \succ^U XF_2 \iff$ *there is a directed path from the node containing* XF_2 *to the node containing* XF_1*.*

Proof. (Sketch) This theorem will be proven in the remainder of this section. We now sketch which lemmas are needed to complete this proof. From Lemma 3 it follows that $XQ^R \equiv^U XQ_t^R$ and from Lemma 2 follows that $XQ_+^R \equiv^U XQ_!^R$. From Lemma 7 and Lemma 8 follows that $XQ_! \succ^U XQ_+$ and from Lemma 9 follows that $XQ_!^R \succ^U XQ_!$ and $XQ_t^R \succ^U XQ_t$. Moreover, we know by Lemma 6 that $XQ_+ \succ^U XQ_t$ and $XQ_+^R \succ^U XQ_t^R$. By Lemma 10 and Lemma 11 we get that $XQ_t \succ^U XQ$. Finally, it follows from Lemma 6 and Lemma 9 that $XQ_!$ and XQ_+ are incomparable with XQ^R.

5.3 Expressibility Results

In this subsection, we present the lemmas that are used to prove the query- or update-equivalence of LiXQuery$^+$ fragments.

Lemma 2. *For all $XQ^R_!$ programs p it holds that there is a XQ^R_+ program p' that has the same update relation and the same query relation.*

Proof. (Sketch) In [8] we have shown that node construction in XQ^R does not add expressive power for "node-conservative deterministic queries". This was shown by encoding the store into a sequence of atomic values and simulating expressions with node construction to manipulate this encoded store. Using a similar simulation technique we can encode the output store, output sequence and list of pending updates in one sequence. Note that we have to use recursive functions to simulate the behavior of for-loops, since the encoded result store of one iteration has to be the input encoded store of the next iteration. Moreover, to ensure a correct computation, we have to apply updates on the encoded store as soon as they are applied in the $XQ^R_!$ expression. Note also that in the simulation we have to use node construction as source of non-determinism in order to have the same update relations. This can be done by expressions like `(element {"a"} {()}) << (element {"a"} {()})`. Finally, we obtain the result store by performing the (encoded) lists of pending updates and performed updates in the correct order.

Lemma 3. *For all XQ^R_t programs p it holds that there is a XQ^R program p' that has the same update relation and the same query relation.*

Proof. (Sketch) We use the same simulation as sketched in the proof of Lemma 2. Since the nodes of the input store are not modified by transform-expressions, we only extend the input store. The result store can be obtained at the end of the simulation by using a recursive function that creates new nodes for nodes that are in the encoded output store but not in the input store.

Lemma 4. *For all XQ_+ programs p it holds that there is a XQ program p' that has the same query relation.*

Proof. (Sketch) Similar to the proof of Lemma 2 and Lemma 3 we can simulate all expressions to work on an encoded store. However, since we now do not have recursive functions to do the simulation, we have to keep track of the transitive closure of E, which we can obtain by using the descendant axis. It can be shown that all updates that are expressible in XQ_+ can be simulated, since we can express the corresponding updates on the encoded descendant relation in XQ.

Lemma 5. *For all $XQ^R_!$ programs p it holds that there is a XQ^R program p' that has the same query relation.*

Proof. (Sketch) The proof of this lemma is similar to that of Lemma 2 and Lemma 4. However, at the end we do not have to create the result store, but it suffices to return the result sequence, which only contains atomic values.

5.4 Inexpressibility Results

We now present the lemmas that are used to show that two LiXQuery$^+$ fragments are not query- or update-equivalent.

Lemma 6. *There are XQ_+ programs which have an update relation that we cannot express by a program in XQ_t^R.*

Proof. (Sketch) This can easily be seen by the fact that in XQ_+ we can modify nodes from the input store, while we cannot do this in XQ_t^R.

Lemma 7. *For all XQ_+ programs p it holds that the largest number (atomic value) in the output sequence is polynomially bounded by the number of nodes in the input store, the length of the longest sequence in the environment and the largest number (atomic value) in both the store and the environment.*

Proof. (Sketch) This can be shown by induction on the structure of the program. In [4] this was shown for the fragment that we refer to as XQ in this paper. From the simulation used to prove Lemma 4 it holds that the same polynomial upper bounds also holds for XQ_+ expressions, because the size of the simulating expression is polynomially bounded by the size of the simulated expression.

Lemma 8. *$XQ_!$ can express all primitive recursive functions over integers.*

Proof. (Sketch) We can give a translation that maps primitive recursive functions to $XQ_!$ expressions with one free variable, which models the arguments of such functions, i.e., tuples of natural numbers. It can easily be seen that the zero function, the succesor function, the projection and the composition can already be expressed in XQ. Primitive recursion can be simulated in $XQ_!$ by using the for-expression and the snap operation, which allows us to do bounded iteration.

Lemma 9. *There are programs in XQ^R which have a query relation that we cannot express by a program in $XQ_!$.*

Proof. (Sketch) It can be shown that all $XQ_!$ programs can be simulated by Turing machines that always halt, while XQ^R is Turing-complete.

Lemma 10. *For all XQ programs there is a depth d such that all nodes that are in the result store, but not in the input store and that have at least d ancestors are deep-equal to nodes in the input store.*

Proof. (Sketch) This property can be shown by induction on the structure of the program. Only node construction can create new nodes and the result is a new tree in the store, where all nodes except for the root are deep-equal to nodes that already existed, i.e., that are in the result store of the subexpression.

Lemma 11. *The property of Lemma 10 does not hold for XQ_t programs.*

Proof. (Sketch) This XQ_t program does not satisfy the property of Lemma 10:

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

6 Relation to Other XQuery-based Update Languages

In this section we briefly discuss the relationship of various LiXQuery$^+$ fragments and a number of existing proposals that extend XQuery with updates.

The first proposal that we consider is UpdateX [10, 1] which corresponds closely to XQ_+ and with XQ_+^R if recursive function definitions are allowed. They have a notion similar to a list of pending updates and the updates in this list are applied in the order that they are generated, as in XQ_+. The constructs of XQ_+ that are not in UpdateX include transform and rename operations. However, as can be seen from the proof of Lemma 4 it is possible to simulate programs that contain transform expressions with programs that do not.

The second proposal that we consider is XQuery! which is an extension of UpdateX with a snap operation. Hence it closely corresponds to $XQ_!$ and with $XQ_!^R$ if recursive function definitions are allowed. A small difference is that in UpdateX the semantics of the implicit top-level snap expressions is of the type *unordered deterministic*, a mode that they call *conflict-free*.

The final and third proposal that we consider is the XQuery Update Facility which corresponds closely to XQ_+ and with XQ_+^R if recursive function definitions are allowed. The semantics of this proposal differs in some details with the semantics of LiXQuery$^+$. For example, their semantics of the "`replace value of`" operation allows to change the content of element nodes, which can be simulated in LiXQuery$^+$. This proposal has an explicit "`transform`" operation, which the other two proposals lack but is included in XQ_+. An important difference is that at the time of writing the working draft does not allow to mix updating and non-updating expressions anymore, i.e., queries and updates are split. We propose that, as is demonstrated by the presented syntax and semantics of LiXQuery$^+$, it is straightforward to define the semantics of a language that does not have this restriction. Moreover, it can be shown that for all LiXQuery$^+$ programs p there exists an equivalent program p' where all queries and updates are split, i.e., there are no subexpressions that return both a non-empty result sequence and a non-empty list of pending updates. To prove this, we can use again a simulation of the store and list of pending updates, similar as used in the proof of Lemma 5. We can do this as follows. First we declare a variable as the encoded result of the simulating expression. Note that the simulating expression generates no real pending update list, only an encoded one. Then we extract and perform the encoded list of pending updates, and bind the resulting empty sequence to a variable. Finally, we can extract the result sequence from the encoded result sequence and return this as result of the program.

7 Conclusion

In order to investigate the relative expressive power of some special constructs that were introduced in XQuery-based update languages, we define LiXQuery$^+$ which combines these constructs. The syntax and semantics of this language is formally defined, demonstrating that this can be done in a concise and complete

fashion. We compare several subsets of LiXQuery$^+$ in terms of queries and updates that can be expressed. One observation that is made is that the "`snap`" operation adds expressive power, even for expressing queries, because it allows the simulation of primitive recursive functions without using recursive function definitions. Another observation is that the "`transform`" operation allows the construction of new trees that would require recursion in XQuery.

In future research we intend to look at the relative expressive power of the different types of "`snap`" operations. Another subject of interest is finding better characterizations of the expressive power of the presented fragments. For example, we suspect that $XQ_!$ can express exactly all primitive recursive functions over XML trees.

References

1. M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P*, 2005.
2. D. Chamberlin, D. Florescu, and J. Robie. XQuery update facility. W3C Working Draft, 2006. `http://www.w3.org/TR/xqupdate/`.
3. G. Ghelli, C. Ré, and J. Siméon. XQuery!: An XML query language with side effects. In *DataX 2006*, Munich, Germany, 2006.
4. J. Hidders, S. Marrara, J. Paredaens, and R. Vercammen. On the expressive power of XQuery fragments. In *DBPL 2005*, Trondheim, Norway, 2005.
5. J. Hidders, J. Paredaens, and R. Vercammen. Comparing the expressive power of XQuery-based update languages. Technical Report TR UA 2006-10, University of Antwerp, Dept. of Mathematics and Computer Science, 2006. `http://adrem.ua.ac.be/pub/TR2006-10.pdf`.
6. J. Hidders, J. Paredaens, and R. Vercammen. LiXQuery$^+$: an XQuery-based update language. Technical report, University of Antwerp, Dept. of Mathematics and Computer Science, 2006. `http://adrem.ua.ac.be/pub/lixqueryplus.pdf`.
7. J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *XSym 2004*, Toronto, Canada, 2004.
8. W. Le Page, J. Hidders, P. Michiels, J. Paredaens, and R. Vercammen. On the expressive power of node construction in XQuery. In *WebDB 2005*, 2005.
9. D. Obasanjo and S. B. Navathe. A proposal for an XML data definition and manipulation language. In *Proc. of EEXTT 2002*, London, UK, 2003.
10. G. M. Sur, J. Hammer, and J. Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.
11. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.