LUX: A lightweight, statically typed XML update language

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

Abstract

Several proposals for updating XML have been introduced. Many of them have a rather complicated semantics due to the interaction of side-effects and updates, and some proposals also complicate the semantics of XQuery because arbitrary side-effecting update statements are allowed inside queries. Moreover, static typechecking has not been studied for any proposed XML update language.

In this paper, we survey prior work on XML update languages and motivate an alternative approach to updating XML. We introduce LUX, which stands for Lightweight Updates for XML. LUX can perform relational database-style updates, has a simple, deterministic operational semantics, and has a sound static type system based on regular expression types and structural subtyping.

1. Introduction

Query and transformation languages for tree-structured data, such as XML, have been extensively investigated. XML, in particular, is now widely used for streaming, exchanging and storing data, yet while standards for validating (DTDs, XML Schema [17]), querying (XQuery [5]), and transforming (XSLT [13]) XML data have reached an advanced stage; formal semantics and type systems for such languages are well-understood, and mature, efficient implementations exist. However, dedicated update languages for XML have received much less attention.

Several languages for updating XML have been proposed over the last six years [24, 32, 6, 33, 31, 8, 20, 10, 22, 9], with a trend towards increasing expressiveness and semantic complexity. The proposals can be characterized by the answers to a number of questions shown in Figure 1, many of which correspond directly to **MUST**- or **SHOULD**-requirements in the current draft of the W3C's XQuery Update Facility Requirements [11]. Early XML update language proposals considered only sequences of atomic operations without iteration, but more recent proposals include sophisticated forms of iteration and control of side-effects so are much more expressive. Expressiveness is certainly a reasonable design goal, but it is important to weigh the benefits of added expressiveness against their costs in increased semantic and implementation complexity: sophisticated update operations are more difficult to optimize and analyze.

Atomic: What atomic updates are allowed?

- **I:** Inserting data before or after a node
- **A:** Appending or prepending data to a node's child list
- **D:** Deleting a node (and its subtree)
- **R:** Replacing a node's value
- **N:** Renaming a node's label

Selection: How can parts of a document be selected for updating?

- **XPath:** Using XPath expressions
- **XQuery:** Using XQuery expressions
- **Ad hoc:** Using ad hoc formalism

Style: What style of programming is used for update statements, and how do such statements interact with XQuery expressions?

- **Ad hoc:** Update statements use a separate, ad hoc syntax unrelated to XQuery (e.g. sequences of atomic updates)
- **Separate:** Update statements use an XQuery-like syntax and contain XQuery expressions but are not allowed as XQuery expressions themselves. but are separate from XQuery
- **Combined:** Update statements are allowed as XQuery expressions of type ()

Nondeterminism: How is nondeterminism arising from the interaction of side-effects and evaluation order handled?

- **Unspecified:** issue not discussed; no formal semantics
- **Checked:** pending update lists with determinism enforced by dynamic checks
- **Fixed order:** pending update lists with determinism enforced by fixing an evaluation order
- **User control:** nondeterminism controlled by user annotations

Variables: Are variables mutable?

Transforms: Can XQuery values be constructed by "transformation" (applying an update to a copy of some data)?

Formal: How formal is the description of the semantics of updates?

Types: Is there a static type system?

Figure 1. Design space for XML update languages.

The situation in the relational/SQL world is very different. In SQL, side-effecting update statements cannot be embedded within arbitrary queries; moreover, the update language is considerably weaker than the query language. For example, SQL's `INSERT-DELETE-UPDATE` statements cannot simulate joins. This simplicity has a number of advantages. First, update optimization appears much easier than query optimization. Second, since updates cannot occur inside queries, query optimization is unburdened by the obligation to prove the absence (or irrelevance) of side-effects. A third benefit of SQL's simplistic approach to updates is that it is possible to statically typecheck them before execution. None of this is to say that SQL's update facility is perfect. But understanding what made it successful despite its flaws may be helpful when designing an update language for XML, especially given that updating relational data stored as XML is an important use case for XML update languages [11, 26].

In recent XML update proposals, however, updates introduce considerable complexity over XQuery. In some proposals, updates are allowed in XQuery expressions, which means that many of XQuery's advantages as a purely-functional language disappear due to the presence of side-effects. For example, determining whether two XQuery[1] expressions commute requires a subtle analysis [21]. In other proposals, queries and updates are separated, but nodes may be selected using arbitrary XPath or XQuery expressions. As a result, static analysis is needed to determine whether updates can be reordered or applied eagerly [2, 3].

Static typechecking appears to pose significant challenges for most of the extant update languages. This may seem surprising because type systems for XML transformation and query languages have received considerable attention [16, 28, 19, 23]. However, all of these languages treat XML values as immutable. In contrast, most XML update languages include mutable variables. Since arbitrary XPath or XQuery expressions can be used to select nodes to be updated, updates may be nonlocal and involve aliasing among mutable variables. This makes it difficult to determine which part of the context has actually been updated and how to modify the types of variables to reflect the change. To avoid nondeterminism arising from updates and iteration, most proposals use a two-phase "snapshot" semantics. Statically predicting the behavior of an update based on this semantics seems difficult. To the best of our knowledge, static typing has not been investigated for any XML update language.

The purpose of this paper is to investigate an XML update language which is less expressive but has a simpler semantics and is more amenable to static analysis. Our language, LUX (Lightweight Updates for XML), has the following properties:

- Support for standard atomic update operations including insertion, appending, deletion, renaming, and replacement.
- Nodes can be selected using only restricted XPath expressions; only horizontal iteration is allowed.
- Side-effecting update statements are not allowed as queries.
- Variables are immutable.
- The semantics is one-pass and deterministic.
- Updates can be statically typechecked using XDuce-style regular expression types with structural subtyping.

LUX is work in progress. Currently, we have developed a core language that consists of a small number of simple, orthogonal features, with a deterministic operational semantics and sound type system. We have also implemented a proof-of-concept interpreter for the core language. In this paper, we will also present a motivating high-level language which we believe can be translated to the core language; developing such a translation is left for future work.

The structure of this paper is as follows. Section 2 reviews previous database and XML update languages and work on static type systems for XML. Section 3 introduces the high-level version of our proposed language, LUX, via a sequence of examples. Section 4 defines the operational semantics of the core LUX languages, and gives translations of the high-level queries. Section 5 defines a type system for LUX updates, proves its soundness with respect to the operational semantics, and discusses the typechecking decision problem. Section 6 discusses extensions and future directions and Section 7concludes.

2. Background and related work

2.1 Updates in other data models

SQL's update facility has been standard for a long time; the `INSERT`, `DELETE` and `UPDATE` commands have remained essentially unchanged since their introduction in early relational database systems such as INGRES [30]. In the absence of aggregation operations, the semantics of such operations is obvious. Aggregations such as summation or averaging motivate a snapshot-based semantics, which ensures that the result of an update which reads and modifies the same data is deterministic. Research concerning updates for the relational model has focused on efficiently maintaining or safely updating views of a database or checking that integrity constraints are preserved, not on increasing the expressiveness of update languages.

As object-oriented and semistructured data models have been investigated over the last 20 years, little attention appears to have been paid to the problem of high-level update languages to match high-level query languages. This is partly because providing structured updates is a much harder problem for a general graph data model due to the problem of aliasing. Object-oriented databases generally provide for updates via general-purpose programming in an object-oriented language such as C++ or Java. The Lore semistructured database [27] provides a more sophisticated update language interface which permits bulk creation and deletion of edges based on subqueries.

Update languages have also not been studied extensively for complex-object or nested relational data models. A notable exception is Liefke and Davidson's update language CPL+ [25]. This language extends CPL [7], a typed language for querying data consisting of arbitrary combinations of record and monadic collection types, with path-based insert, update, and delete operations; high-level CPL+ updates were formalized by translation to a simpler core language with orthogonal operations for iteration, navigation, insertion/deletion, and replacement.

2.2 Static typing for XML processing

XML processing systems can be characterized as domain-specific transformation languages (e.g. XSLT [13], XDuce [23]) or query languages (e.g. XQuery, XPath [16]), or general-purpose programming languages with native XML support (e.g. Xtatic [19]). We will focus on only the most closely related work; Møller and Schwartzbach [28] provide a much more complete survey of type systems for XML transformation languages.

Hosoya, Vouillon, and Pierce [23] introduced XDuce, the first statically typed XML transformation language. They used a structural type system based on regular expressions in which types can be interpreted as regular tree languages and subtyping is language inclusion. Subtyping is EXPTIME-complete, but Hosoya et al. developed an algorithm that appears well-behaved in practice.

XQuery has a type system based on XML Schema, with regular expression types and nominal subtyping; its operational semantics and type system have been standardized in the W3C's XPath 2.0/XQuery 1.0 Formal Semantics [16]. Síméon and Wadler [29] investigated XML Schema's nominal subtyping and established some basic formal properties of validation. Their work helped clarify inconsistencies in the informal XQuery type system specification. Although typechecking for XQuery is well understood, we will consider a more flexible structural type system in this paper. Typechecking updates using nominal subtyping appears awkward, because of the need to ensure that updates preserve typability using named types.

Colazzo, Ghelli, Manghi and Sartiani [14] considered the problem of typechecking an XQuery core language called μXQ, from the point of view of both *result analysis* (the result of the query

	Atomic	Selection	Style	Nondeterminism	Transforms	Variables	Formal	Types
XML:DB XUpdate [24]	IADRN	XPath	Ad hoc	Unspecified	No	Immutable	No	No
Updating XML [32]	IDRN	XPath	Separate	Unspecified	No	Mutable	No	No
Bruno et al. [6]	IADRN	XPath	Ad hoc	Unspecified	Yes	Immutable	No	No
XML-RL Update [33]	IADR	Ad hoc	Ad hoc	Unspecified	No	Mutable	Yes	No
UpdateX [31]	IADR	XQuery	Separate	Fixed order	No	Mutable	Yes	No
Context logic [8]	IADR	XPath(child)	Ad hoc	Fixed order	No	Mutable	Yes	No
XQuery$^!$ [20]	IADRN	XQuery	Combined	User control	No	Mutable	Yes	No
XQuery Update [10]	IADRN	XQuery	Combined	Checked	Yes	Mutable	No	No
LiXQuery$^+$ [22]	IADRN	XQuery	Combined	User control	Yes	Mutable	Yes	No
XQueryP [9]	IADRN	XQuery	Combined	User control	Yes	Mutable	No	No

Table 1. Summary of XML update language characteristics. See Figure 1 for explanations of the column entries.

always matches a type) and *correctness analysis* (every subexpression which is statically guaranteed to evaluate to () is syntactically ()). We will use μXQ as a component in defining LUX, and use its type system's result analysis component to typecheck μXQ queries incorporated inside LUX updates.

Links [15] is a programming language being developed by Cooper, Lindley, Wadler, and Yallop. Links is a pure, statically typed programming language with features tailored for implementing Web applications which interact with relational or XML databases using queries and updates and with users browsing the Web using XML/HTML and JavaScript. Because Links is purely functional, database updates (especially schema-changing ones) pose potential problems even for simple SQL updates. Some of the motivation for this work came from discussions of update language semantics with colleagues working on Links.

2.3 XML updates

We will briefly describe each of the XML update languages of which we are aware. Since many features are common, we will focus on the distinctive features only; Table 1 summarizes the features of the languages in terms of the design space in Figure 1.

Laux and Martin [24] developed an early working draft of an XML update language called XUpdate. XUpdate provides only XPath-based node selection, conditionals, and sequences of atomic updates.

Tatarinov, Ives, Halevy and Weld [32] considered XQuery language extensions for XML updates. This work focused on evaluating the performance of XML updates translated to SQL updates applied to XML data stored in a relational database.

Bruno, Le Maitre, and Murisasco [6] proposed an extension to XQuery to include *transformation operators*. A transformation operator evaluates an XQuery expression, makes a copy of it, applies updates to the copy, and returns the result. They described a number of examples and compared their approach with equivalent transformations written in XSLT.

Wang, Liu and Lu [33] investigated adding XML update primitives to XML-RL, a rule-based query language which views XML documents as instances of a complex-object data model [1]. The main difference between their approach and others seems to be that it uses XML-RL pattern-matching queries to select nodes and bind them to variables, rather than using XPath/XQuery. While perhaps more convenient for some examples, this approach appears to have the same semantic complexity as most other approaches. Wang et al. described a complex-object schema language for XML-RL but did not consider the issue of how to typecheck updates.

UpdateX, due to Sur, Hammer and Siméon [31], is an XML update language based on XQuery. UpdateX permits arbitrary XQuery expressions as update content, but updates cannot occur within queries and the syntax of updates has some arbitrary-seeming restrictions. The original paper discussed semantics and optimization informally, but Benedikt, Bonifati, Flesca, and Vyas [2, 3] subsequently studied the semantics of UpdateX more rigorously, and developed analyses for ensuring the safety of optimizations such as eager evaluation and update reordering.

Calcagno, Gardner and Zarfaty [8] investigated XML updates in the setting of *context logic*, which includes formulas describing both trees and contexts (trees with holes). They developed a form of Hoare logic which can be used to reason about sequences of updates to deterministic, unordered trees using path addresses. This approach is very expressive, but also comparatively low-level; the basic update operations are comparable to those provided by the the DOM interface. Also, their approach currently applies only to unordered, deterministic trees, a simplification of full XML.

Ghelli, Re and Siméon [20] have developed XQuery$^!$, an XML query language with side-effects. In XQuery$^!$, update statements are considered to be query expressions returning (). When an update expression is evaluated, it generates updates which are added to a pending update list, and applied at the end of the query. In order to provide finer-grained control of update application, XQuery$^!$ provides a `snap` operator that takes a snapshot of the database, evaluates an expression, and applies its pending updates to the snapshot. Ghelli et al. presented an XQuery-style formal operational semantics for XQuery$^!$ and discussed implementation and optimization issues. More recently, Ghelli, Rose, and Símeôn [21] have developed an analysis for determining whether two expressions in a simplified form of XQuery$^!$ commute.

The W3C XQuery Update Facility [10] has been under development for several years. The current version is most closely related to XQuery$^!$; the two main differences are the absence of the `snap` operator and the presence of transformations.

LiXQuery$^+$ is an update language introduced by Hidders, Paredaens, and Vercammen [22] in order to study the expressive power of XQuery-based update languages, such as XQuery$^!$ and the XQuery Update Facility. LiXQuery$^+$ includes all of the features considered above, including both transformations and explicit snapshots, and possesses an operational semantics similar to those for XQuery and XQuery$^!$.

XQueryP, due to Chamberlin, Carey, Florescu, Kossman, and Robie [9], is an extension to XQuery that includes both updates and traditional imperative programming features, including block structure, assignment, an `updating` keyword for declaring impure functions, and `while`-loops. XQueryP also includes an `atomic` keyword which groups operations into atomic blocks (transactions). The motivation for adding these features, as with procedural extensions to SQL, is to allow the programmer to move more processing into the database, thus avoiding the "impedance mismatch" between XML and traditional type systems and improving prospects for optimization.

```
UpdStmt ::= CREATE DOCUMENT Name
          | DROP DOCUMENT Name
          | INSERT (AS (FIRST|LAST))? INTO Path
              VALUE Expr
          | INSERT (BEFORE|AFTER) Path VALUE Expr
          | UPDATE (IN?) Path BY UpdStmt
          | LET $var := Expr IN UpdStmt
          | DELETE (FROM)? Path
          | RENAME Path TO Name
          | REPLACE Path (CONTENT?) WITH Expr
          | UpdStmt WHERE Cond
          | UpdStmt ; UpdStmt
Path    ::= . | Path/Name | Path/*
Expr    ::= XQuery expressions
Cond    ::= XQuery/XPath conditional expressions
```

Figure 2. Concrete syntax of LUX updates.

3. Overview and examples

3.1 Full language syntax

As with CPL+, XQuery and many other languages, we introduce a high-level, readable syntax with complex, overlapping operations which can be translated to a much simpler core language. Update statements allow us to create and drop (delete) top-level documents, insert XQuery values into trees based on paths (at the beginning or end of a child list or before or after a node), update subtrees at a given path, delete from a path all subtrees satisfying a condition, rename subtrees at a given path, or replace subtrees. Statements can also be sequenced and restricted using conditional expressions, and let-expressions can be used to bind variables to the values of expressions.

Path expressions in updates are used in two distinct senses, “tree-oriented” and “sequence-oriented”. In tree-oriented updates, the update expects a singleton tree and is executed once for each subtree selected by the path expression; in sequence-oriented updates, the update operates on an arbitrary sequence and is executed on the child-list of each node selected by the path expression.

Tree-oriented insertions (`INSERT BEFORE/AFTER`) insert a value before or after each node selected by the path expression. Similarly, tree-oriented deletes (`DELETE`) assume that we have selected a single node and want to delete it. Plain replacement `REPLACE Path WITH` is tree-oriented by default.

Sequence-oriented insertions (`INSERT INTO`) insert a value into the child-list of each selected node, at the beginning (`AS FIRST`) or end (`AS LAST`). Sequence-oriented deletes (`DELETE FROM`) delete nodes from the child-list of each selected node. The `REPLACE Path CONTENT WITH` operator is sequence-oriented; it replaces the content of a path with new content.

The “dot” path expression `.` refers to the currently selected part of the tree; its value depends on context. Path expressions always begin with `./`, but in the examples we usually omit this prefix. In what follows, we assume familiarity with XQuery and XPath syntax, and with XDuce-style regular expression types for XML data.

3.2 Creating a database and loading data

Suppose we start an XML database with no pre-loaded data; its type is the empty sequence (). We want to create a small database listing books and authors. The following LUX updates accomplish this:

```
U1 : CREATE DOCUMENT books[];
     CREATE DOCUMENT authors[]
```

After this update, the database has type

```
books[],authors[]
```

Now we want to load some XML data into the database. Since XML text is included in XQuery’s expression language, we can just do the following:

```
U2 : INSERT INTO books VALUE
     <book><author>Charls Dickens</author>
           <title>A Tale of Two Cities</title>
           <year>1858</year></book>
     <book><author>Lewis Carroll</author>
           <title>Through the Looking-Glass</title>
           <year>??</year></book>;
     INSERT INTO authors VALUE
     <author><name>Charles Dickens</name>
             <born>1812</born>
             <died>1870</died></author>
     <author><name>Lewis Carroll</name>
             <born>1832</born>
             <died>1898</died></author>
```

This results in a database with type

```
books[ book[author[string],name[string],
            year[string]]* ],
authors[ author[name[string],born[string],
                died[string]]* ]
```

3.3 Updating data

The data we initially inserted was incomplete and incorrect. Now we want to fill in some of the missing data and correct an error.

```
U3 : UPDATE books/book BY
       REPLACE CONTENT author WITH "Charles Dickens";
       REPLACE CONTENT year WITH "1859"
     WHERE title = "A Tale of Two Cities"
U4 : UPDATE books/book BY
     REPLACE CONTENT year WITH "1872";
     WHERE title = "Through the Looking-Glass"
```

This update leaves the structure of the database unchanged.

3.4 Restructuring the database

SQL updates include `ALTER TABLE` statements that make it possible to add or remove columns. Similarly, we can add an element to each `book` in `books` as follows:

```
U5 : INSERT AS LAST INTO books/book
     VALUE publisher["Grinch"]
```

After `U5`, the books database has type

```
books[ book[author[string],name[string],
            year[string],publisher[string]]* ]
```

Now perhaps we want to add a co-author; for example, perhaps Lewis Carroll collaborated on “Through the Looking-Glass” with Charles Dickens. This is not as easy as adding the publisher field to the end because we need to select a particular node to insert before or after. In this case we happen to know that there is only one author, so we can insert after that; however, this would be incorrect if there were multiple authors, and we would have to do something else (such as inserting before the title).

```
U6 : UPDATE books/book BY
       INSERT AFTER author
       VALUE <author>Charles Dickens</author>
     WHERE name = "Through the Looking-Glass"
```

Now the database has the type:

```
books[ book[author[string]*,title[string],
            year[string],publisher[string]]* ]
```

Now that some books have multiple authors, we might want to change the flat author lists to nested lists:

```
U7 : REPLACE books/book WITH
     <book><authors>{author}</authors>
           {title}{year}{publisher}</book>
```

This visits each book and changes its structure so that the authors are grouped into an authors element. The resulting database has type:

```
books[ book[authors[author[string]* ],title[string],
            year[string],publisher[string]]* ]
```

3.5 Deleting data

Suppose we later decide that the publisher field is unnecessary after all. We can get rid of it using the following update:

```
U8 : DELETE books/book/publisher
```

This results in schema

```
books[ book[authors[author[string]* ],
            title[string],year[string]]* ]
```

Now suppose Lewis Carroll no longer interests us and we wish to remove all of his books from the database.

```
U9 : DELETE FROM books
     WHERE book/authors/author = "Lewis Carroll"
```

This update does not modify the type of the database. Finally, we can delete a top-level document as follows:

```
U10 : DROP DOCUMENT authors
```

3.6 Execution model

Informally, the execution model used by Lux is as follows. Each update executes with respect to a *context node*. Moreover, *an update can only modify data at or beneath its current context node*. We call this the *side-effect isolation property*. For example, navigating to the context node's parent and then modifying another sibling is not allowed. In addition, whenever an iterative update occurs (that is, one involving a loop traversing a number of nodes), we require that the iteration is deterministic; that is, *the result of an iterative update is independent of the order in which the nodes are updated*. We call this the *traversal-order independence property*.

In general, an update operation with a path expression in it is evaluated as follows: The path expression is evaluated, yielding a set of nodes. Then the corresponding basic update operation (insert, delete, etc.) is applied to each node in this set in turn, using it as the context node.

To ensure isolation of side effects and traversal-order independence, we restrict the XPath expressions that can be used to select a context node. Specifically, only the child axis and path composition are allowed. This ensures that only descendants of a given context node can be selected as the new context node and that a selection of new context nodes contains no pairs of nodes in an ancestor-descendant relationship. Consequently, the side effects of an update are confined to the subtree of its context node, and the result of an iteration is independent of the traversal order. This keeps the semantics deterministic and helps make typechecking feasible.

In Section 4, we will define a core language that satisfies the side-effect isolation and traversal-order independence properties. We believe that a high-level language such as the one outlined in this section can be compiled down to this core language, but doing this is beyond the scope of this paper.

3.7 Non-design goals

We are not attempting to make or defend a claim that this is a candidate for "the" update language for XML to the exclusion of other proposals. Although we believe it works well for database-style applications (especially for SQL-like operations over XML data), there are several things that other proposals do that we make no attempt to do. Whether this is an advantage or disadvantage depends on the application.

Pattern matching: Many transformation/query languages and some update languages (e.g. XML-RL update and CPL+) allow defining transformations by *pattern matching*, that is, matching tree patterns against the data. Pattern matching is crucial for transforming XML, but we believe it is not as important for updates. We have not considered general pattern matching in Lux, in order to keep the type system and operational semantics as simple as possible.

Side-effects in queries: Several motivating examples for XQuery[1] depend on the ability to perform side-effects within queries. Examples include logging accesses to particular data or profiling or debugging an XQuery program. Lux cannot be used for these applications because side-effects are not allowed within queries.

Deep updates/recursion: Lux is only capable of horizontal iteration: iteration over all of the children of a node. Lux does not include recursively defined updates or recursive types, and cannot meaningfully update data arbitrarily deeply in the tree, because selection paths are restricted to the child axis only, so cannot, for example, update all $person$ elements everywhere in the tree in-place. Consequently, Lux by itself is not suitable for performing XML transformations such as rendering XML data to XHTML, or for updating genuinely recursive data. Static typechecking for recursive updates appears to be a challenging future direction.

Joins: Lux updates cannot, by themselves, iterate superlinearly over the data to perform a join. However, updates can use embedded XQuery expressions to restructure the database, for example:

```
INSERT INTO tmp VALUE
  FOR $x in books,$y in authors
  WHERE $x/author = $y/name
  RETURN <pair>{$x}{$y}</pair>
```

4. Formalization

4.1 Core query language

Because Lux uses queries to construct values, we need to introduce a query language and define its semantics before doing the same for Lux. We will use a small fragment of XQuery called μXQ, introduced by Colazzo, Ghelli, Manghi and Sartiani [14].

Following [14], we distinguish between *tree values* $t \in Tree$, which include strings $w \in \Sigma^*$ (for some alphabet Σ), boolean values, and singleton trees $n[v]$; and *(forest) values* $v \in Val$, which are sequences of tree values:

$$\begin{array}{lrcl} \text{(Forest) values} & v & ::= & () \mid t, v \\ \text{Tree values} & t & ::= & n[v] \mid w \mid \texttt{true} \mid \texttt{false} \end{array}$$

Two values can be concatenated by concatenating them as lists; abusing notation, we identify trees t with singleton forests $t, ()$ and write v, v' for forest concatenation. We define a comprehension operation on forest values as follows:

$$\begin{array}{rcl} [f(x) \mid x \in ()] & = & () \\ [f(x) \mid x \in t, v] & = & f(t), [f(x) \mid x \in v] \end{array}$$

This operation takes a forest $(t_1, \ldots, t_n)$ and a function $f(x)$ from trees to forests and applies f to each tree t_i, concatenating the resulting forests in order. Comprehensions satisfy basic monad laws as well as some additional equations (see [18]).

$$
\begin{array}{rcl@{\qquad}rcl}
\llbracket \texttt{true} \rrbracket\sigma & = & \texttt{true} & \llbracket \texttt{false} \rrbracket\sigma & = & \texttt{false} \\
\llbracket () \rrbracket\sigma & = & () & \llbracket e, e' \rrbracket\sigma & = & \llbracket e \rrbracket\sigma, \llbracket e' \rrbracket\sigma \\
\llbracket n[e] \rrbracket\sigma & = & n[\llbracket e \rrbracket\sigma] & \llbracket w \rrbracket\sigma & = & s \\
\llbracket x \rrbracket\sigma & = & \sigma(x) & \llbracket \bar{x} \rrbracket\sigma & = & \sigma(\bar{x})
\end{array}
$$

$$
\begin{array}{rcl}
\llbracket \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rrbracket\sigma & = & \llbracket e_2 \rrbracket\sigma[x := \llbracket e_1 \rrbracket\sigma] \\
\llbracket \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rrbracket\sigma & = & \begin{cases} \llbracket e_1 \rrbracket\sigma & \llbracket c \rrbracket\sigma = \texttt{true} \\ \llbracket e_2 \rrbracket\sigma & \llbracket c \rrbracket\sigma = \texttt{false} \end{cases} \\
\llbracket e = e' \rrbracket\sigma & = & \begin{cases} \texttt{true} & \llbracket e \rrbracket\sigma = \llbracket e' \rrbracket\sigma \\ \texttt{false} & \llbracket e \rrbracket\sigma \neq \llbracket e' \rrbracket\sigma \end{cases} \\
\llbracket e :: n \rrbracket\sigma & = & [n[v] \mid n[v] \in \llbracket e \rrbracket\sigma] \\
\llbracket \bar{x}/\texttt{child} \rrbracket\sigma & = & \mathit{children}(\sigma(\bar{x})) \\
\llbracket \texttt{for}\ \bar{x} \in e_1\ \texttt{return}\ e_2 \rrbracket\sigma & = & [\llbracket e_2 \rrbracket\sigma[\bar{x} := t] \mid t \in \llbracket e_1 \rrbracket\sigma]
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{children}(n[f]) & = & f \\
\mathit{children}(v) & = & () \quad (v \neq n[v'])
\end{array}
$$

Figure 3. Semantics of query expressions.

Again following [14], we distinguish between *tree variables* $\bar{x} \in \mathit{TVar}$, introduced by `for`, and *forest variables*, $x \in \mathit{Var}$, introduced by `let`. We write $X \in \mathit{Var} \cup \mathit{TVar}$ for an arbitrary tree or forest variable. The other syntactic classes of our variant of μXQ include labels $l \in \mathit{Lab}$ and expressions $e \in \mathit{Exp}$; expressions are defined by the following BNF grammar:

$$
\begin{array}{rcl}
e & ::= & () \mid e, e' \mid n[e] \mid w \mid x \mid \texttt{let}\ x = e\ \texttt{in}\ e' \\
& \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}\ c\ \texttt{then}\ e\ \texttt{else}\ e' \mid e = e' \\
& \mid & \bar{x} \mid \bar{x}/\texttt{child} \mid e :: n \mid \texttt{for}\ \bar{x} \in e\ \texttt{return}\ e'
\end{array}
$$

The distinguished variables $\bar{x}$ in $\texttt{for}\ \bar{x} \in e\ \texttt{return}\ e'(x)$ and x in $\texttt{let}\ x = e\ \texttt{in}\ e'(x)$ are bound in $e'(x)$. Here and elsewhere, we make the convention that expressions containing bound variables are equivalent modulo α-renaming.

To simplify the presentation, we split μXQ's projection operation $\bar{x}\ \texttt{child} :: l$ into two expressions: child projection ($\bar{x}/\texttt{child}$) which returns the children of $\bar{x}$, and node name filtering ($e :: n$) which evaluates e to an arbitrary sequence and selects the nodes labeled n. We also define the children of a value other than $n[v]$ to be the empty sequence rather than leaving it undefined. Thus, the ordinary child axis expression $\bar{x}\ \texttt{child} :: n$ is syntactic sugar for $(\bar{x}/\texttt{child}) :: n$ and the "wildcard" child axis is definable as $x\ \texttt{child} :: * = x/\texttt{child}$. We also consider only one built-in operation, equality.

An environment is a pair of functions $\sigma : (\mathit{Var} \to \mathit{Val}) \times (\mathit{TVar} \to \mathit{Tree})$. Abusing notation, we write $\sigma(x)$ for $\pi_1(\sigma)(x)$ and $\sigma(\bar{x})$ for $\pi_2(\sigma)(\bar{x})$. The semantics of expressions is defined as shown in Figure 3. The function $\mathit{children} : \mathit{Val} \to \mathit{Val}$ maps a singleton tree $n[f]$ to its child list and maps other values to the empty sequence. We write $\sigma \vdash e \Rightarrow v$ when $v = \llbracket e \rrbracket\sigma$.

4.2 Core update language

We now introduce the core LUX update language, which includes statements $s \in \mathit{Stmt}$, tests $\phi \in \mathit{Test}$, and directions $d \in \mathit{Dir}$:

$$
\begin{array}{rcl}
s & ::= & \texttt{skip} \mid s; s' \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s' \mid \texttt{let}\ x = e\ \texttt{in}\ s \\
& \mid & \texttt{insert}\ e \mid \texttt{delete} \mid \texttt{snapshot}\ x\ \texttt{in}\ s \mid \phi?s \mid d[s] \\
\phi & ::= & n \mid * \mid \texttt{bool} \mid \texttt{string} \\
d & ::= & \texttt{left} \mid \texttt{right} \mid \texttt{children} \mid \texttt{iter}
\end{array}
$$

Updates include standard constructs such as the no-op `skip`, sequential composition, conditionals, and `let`-binding. The basic update operations include insertion $\texttt{insert}\ e$, which inserts an expression into an empty sequence; deletion `delete`, which deletes the selected value; and the "snapshot" operation $\texttt{snapshot}\ x\ \texttt{in}\ s(x)$, which binds x to the current value of the selected part of the tree and then applies an update $s(x)$. Also, `snapshot` is *not* equivalent to XQuery!'s `snap` operator; `snapshot` binds x to an immutable value which can be used in s, whereas `snap` delimits a "snapshot semantics" execution of a side-effecting query. The snapshot operator is also not equivalent to the `let` statement, since there is no way to refer to the database within a μXQ query.

$$
\boxed{\sigma; v \vdash s \Rightarrow^{\texttt{U}} v'}
$$

$$
\frac{}{\sigma; v \vdash \texttt{skip} \Rightarrow^{\texttt{U}} v}
\qquad
\frac{\sigma; v \vdash s \Rightarrow^{\texttt{U}} v_1 \quad \sigma; v_1 \vdash s' \Rightarrow^{\texttt{U}} v_2}{\sigma; v \vdash s; s' \Rightarrow^{\texttt{U}} v_2}
$$

$$
\frac{\sigma \vdash e \Rightarrow v \quad \sigma[x := v] \vdash v_1 \Rightarrow sv_2}{\sigma; v_1 \vdash \texttt{let}\ x = e\ \texttt{in}\ s \Rightarrow^{\texttt{U}} v_2}
$$

$$
\frac{\sigma \vdash e \Rightarrow \texttt{true} \quad \sigma; v \vdash s_1 \Rightarrow^{\texttt{U}} v'}{\sigma; v \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow^{\texttt{U}} v'}
$$

$$
\frac{\sigma \vdash e \Rightarrow \texttt{false} \quad \sigma; v \vdash s_2 \Rightarrow^{\texttt{U}} v'}{\sigma; v \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow^{\texttt{U}} v'}
$$

$$
\frac{\sigma \vdash e \Rightarrow v}{\sigma; () \vdash \texttt{insert}\ e \Rightarrow^{\texttt{U}} v}
\qquad
\frac{}{\sigma; v \vdash \texttt{delete} \Rightarrow^{\texttt{U}} ()}
$$

$$
\frac{\sigma[x := v]; v \vdash s \Rightarrow^{\texttt{U}} v'}{\sigma; v \vdash \texttt{snapshot}\ x\ \texttt{in}\ s \Rightarrow^{\texttt{U}} v'}
\qquad
\frac{t \in \llbracket \phi \rrbracket}{\sigma; t \vdash \phi?s \Rightarrow^{\texttt{U}} t}
$$

$$
\frac{t \notin \llbracket \phi \rrbracket}{\sigma; w \vdash \phi?s \Rightarrow^{\texttt{U}} w}
\qquad
\frac{\sigma; v \vdash s \Rightarrow^{\texttt{U}} v'}{\sigma; n[v] \vdash \texttt{children}[s] \Rightarrow^{\texttt{U}} n[v']}
$$

$$
\frac{\sigma; () \vdash s \Rightarrow^{\texttt{U}} v'}{\sigma; v \vdash \texttt{left}[s] \Rightarrow^{\texttt{U}} v', v}
\qquad
\frac{\sigma; () \vdash s \Rightarrow^{\texttt{U}} v'}{\sigma; v \vdash \texttt{right}[s] \Rightarrow^{\texttt{U}} v, v'}
$$

$$
\frac{\sigma; t_1 \vdash S \Rightarrow^{\texttt{U}} v_1' \quad \sigma; v_2 \vdash \texttt{iter}[s] \Rightarrow^{\texttt{U}} v_2'}{\sigma; t_1, v_2 \vdash \texttt{iter}[s] \Rightarrow^{\texttt{U}} v_1', v_2'}
\qquad
\frac{}{\sigma; () \vdash \texttt{iter}[s] \Rightarrow^{\texttt{U}} ()}
$$

Figure 4. Operational semantics of updates.

Updates also include *tests* $\phi?s$ which allow us to examine the local structure of a tree value and perform an update if the structure matches. The node label test $n?s$ checks whether the tree is of the form $n[v]$, and if so executes s, otherwise is a no-op; the wildcard test only checks that the value is a singleton tree. Similarly, $\texttt{bool}?s$ and $\texttt{string}?s$ test whether a tree value is a boolean or string value.

Finally, updates include *navigation* operators that change the selected part of the tree, and perform an update on the sub-selection. The `left` and `right` operators perform an update on the empty sequence located to the left or right of a value. The `children` operator applies an update to the child list of a tree value. Conversely, the `iter` operator applies an update to each tree value in a forest.

We distinguish between *singular* (unary) updates which apply only when the context is a tree value and *plural* (multi-ary) updates which apply to an entire value. Tests $\phi?s$ are always singular. The `children` operator takes a plural update and returns a singular one; the `iter` operator takes a singular update to a plural one. Other updates can be either singular or plural in different situations. Our type system (presented in Section 5) tracks arity as well as input and output types in order to ensure that updates are well-behaved.

Figure 4 shows the operational semantics of Core LUX. We write $\sigma; v \vdash s \Rightarrow^{\texttt{U}} v'$ to indicate that given environment σ and context value v, statement s evaluates to value v'. The rules for tests are defined in terms of the following semantic interpretation

$$
\begin{array}{lcl}
\texttt{repl}\ e & = & \texttt{delete};\texttt{insert}\ e\\
\texttt{delete}\ n & = & \texttt{iter}[n?\texttt{delete}]\\
\texttt{if}\ c\ \texttt{then}\ s & = & \texttt{if}\ c\ \texttt{then}\ s\ \texttt{else skip}\\
\texttt{update}\ n\ \texttt{by}\ [s] & = & \texttt{iter}[n?s]\\
\texttt{update in}\ n\ \texttt{by}\ [s] & = & \texttt{update}\ n\ \texttt{by}\ [\texttt{children}[s]]\\
\texttt{update}\ x\ \texttt{in}\ n\ \texttt{by}\ [s] & = & \texttt{update}\ n\\
 & & \texttt{by}\ [\texttt{snapshot}\ x\ \texttt{in children}[s]]\\
\texttt{delete}\ x\ \texttt{from}\ n & = & \texttt{update}\ x\ \texttt{in}\ n\\
\texttt{where}\ c & & \texttt{by}\ [\texttt{if}\ c\ \texttt{then delete}]\\
\texttt{insert}^{\leftarrow}\ e & = & \texttt{left}[\texttt{insert}\ e]\\
\texttt{insert}^{\rightarrow}\ e & = & \texttt{right}[\texttt{insert}\ e]\\
\texttt{insert}^{\swarrow}\ e & = & \texttt{children}[\texttt{insert}^{\leftarrow}\ e]\\
\texttt{insert}^{\searrow}\ e & = & \texttt{children}[\texttt{insert}^{\rightarrow}\ e]
\end{array}
$$

Figure 5. Abbreviations for derived update expressions.

of tests:

$$
\begin{array}{rcl}
[\![\texttt{bool}]\!] & = & \{\texttt{true},\texttt{false}\}\\
[\![\texttt{string}]\!] & = & \Sigma^*\\
[\![n]\!] & = & \{n[v] \mid v \in Val\}\\
[\![*]\!] & = & \{n[v] \mid n \in Lab, v \in Val\}
\end{array}
$$

Note that while we described the evaluation of LUX updates informally in terms of nodes, we can define the semantics entirely in terms of forest and tree values, without needing to define an explicit store. This would not be the case if we considered full XQuery, which includes node identity comparison operations.

It is straightforward to show that an update run against a given environment and input value has most one output value:

THEOREM 1. *Let* σ, v, s, v_1, v_2 *be given such that* $\sigma; v \vdash s \Rightarrow^{\mathtt{U}} v_1$ *and* $\sigma; v \vdash s \Rightarrow^{\mathtt{U}} v_2$. *Then* $v_1 = v_2$.

PROOF. Straightforward by induction on the structures of the two derivations. The interesting cases are those for conditionals and iteration, since they are the only statements that have more than one applicable rule. However, in each case, only matching pairs of rules are applicable. □

Figure 6 shows the example queries from Section 3 translated to core LUX. To simplify the presentation, we use full XQuery syntax for subqueries, rather than more verbose core μXQ queries, and use several abbreviations introduced in Figure 5.

5. Type system

As noted earlier, some updates expect that the input value is a singleton (for example, `children`, $n?s$, etc.) while others work for an arbitrary sequence of trees. Singular (tree-oriented) updates may fail if applied to a sequence. This makes programming and debugging updates tricky, so our type system should prevents all such runtime errors. Moreover, as with all XML transformation languages, we often would like to ensure that when given an input tree of some type τ, an update is guaranteed to produce an output tree of some other type τ'. For example, updates made by non-privileged users are usually required to preserve the database schema.

5.1 Types, subtyping and typing rules

$$
\begin{array}{lcl}
U_1 & : & \texttt{insert}\ (books[], authors[])\\
U_2 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by insert}\ (...);\\
\texttt{update in}\ authors\ \texttt{by insert}\ (...);
\end{array}\right.\\
U_3 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by}\ [\\
\quad\texttt{update}\ x\ \texttt{in}\ book\ \texttt{by}\ [\\
\quad\quad\texttt{if}\ x/title = \texttt{"AToTC"}\ \texttt{then}\ [\\
\quad\quad\texttt{update in}\ author\ \texttt{by}\ [\texttt{repl "CD"}];\\
\quad\quad\texttt{update in}\ year\ \texttt{by}\ [\texttt{repl "1859"}]]]]
\end{array}\right.\\
U_4 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by}\ [\\
\quad\texttt{update}\ x\ \texttt{in}\ book\ \texttt{by}\ [\\
\quad\quad\texttt{if}\ x/title = \texttt{"TtLG"}\ \texttt{then}\ [\\
\quad\quad\quad\texttt{update in}\ year\ \texttt{by}\ [\texttt{repl("1872")}]]]]
\end{array}\right.\\
U_5 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by}\ [\\
\quad\texttt{update in}\ book\ \texttt{by}\ [\\
\quad\quad\texttt{insert}^{\rightarrow}\ (publisher[])]]
\end{array}\right.\\
U_6 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by}\ [\\
\quad\texttt{update}\ x\ \texttt{in}\ book\ \texttt{by}\ [\\
\quad\quad\texttt{if}\ x/title = \texttt{"TtLG"}\ \texttt{then}\ [\\
\quad\quad\quad\texttt{update}\ author\ \texttt{by}\ [\\
\quad\quad\quad\quad\texttt{insert}^{\rightarrow}\ (author[\texttt{"CD"}])]]]]
\end{array}\right.\\
U_7 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by}\ [\\
\quad\texttt{update}\ x\ \texttt{in}\ book\ \texttt{by}\ [\\
\quad\quad\texttt{repl}\ (book[authors[x/authors],\\
\quad\quad\quad x/title, x/year, x/publisher])]]
\end{array}\right.\\
U_8 & : & \left\{\begin{array}{l}
\texttt{update in}\ books\ \texttt{by}\ [\\
\quad\texttt{update in}\ book\ \texttt{by}\ [\texttt{delete}\ publisher]]
\end{array}\right.\\
U_9 & : & \left\{\begin{array}{l}
\texttt{update}\ books\ \texttt{by}\ [\\
\quad\texttt{delete}\ x\ \texttt{from}\ book\\
\quad\texttt{where}\ x/authors/author = \texttt{"LC"}]
\end{array}\right.\\
U_{10} & : & \texttt{delete}\ authors
\end{array}
$$

Figure 6. Translations of example queries into core LUX.

We consider a regular expression type system with structural subtyping, similar to those considered in several transformation and query languages for XML [23, 14, 18].

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid () \mid \tau|\tau' \mid \tau, \tau' \mid \tau^*\\
\alpha & ::= & \texttt{bool} \mid \texttt{string} \mid n[\tau]
\end{array}
$$

We call types of the form α *singular* types (or tree types), and types of all other forms *plural* (or forest types). It should be obvious that a value of singular type must always be a sequence of length one (that is, a tree); plural types may have values of any length. There exist plural types with only values of length one, but which are not syntactically singular (for example `int|bool`). As usual, the $+$ and ? quantifiers can be defined as follows: $\tau^+ = \tau, \tau^*$ and $\tau^? = \tau|()$. Type variables and recursive type definitions are not included.

As in XDuce, a type denotes a set of matching values:

$$
\begin{array}{rclrcl}
[\![\texttt{string}]\!] & = & \Sigma^* & [\![\texttt{bool}]\!] & = & \{\texttt{true},\texttt{false}\}\\
[\![()]\!] & = & \{()\} & [\![n[\tau]]\!] & = & \{n[v] \mid v \in [\![\tau]\!]\}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\tau, \tau']\!] & = & \{v, v' \mid v \in [\![\tau]\!], v' \in [\![\tau']\!]\}\\
[\![\tau|\tau']\!] & = & [\![\tau]\!] \cup [\![\tau']\!]\\
[\![\tau^*]\!] & = & \{()\} \cup \{v_1, \ldots, v_n \mid v_1 \in [\![\tau]\!], \ldots, v_n \in [\![\tau]\!]\}
\end{array}
$$

In addition, we define a binary *subtyping* relation on types. A type τ_1 is a subtype of τ_2 ($\tau_1 <: \tau_2$), by definition, if $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. Since our system is a special case of XDuce's type system, we know that subtyping is decidable; although XDuce subtyping is EXPTIME-complete in general, the algorithm of Hosoya, Vouillon and Pierce is well-behaved in practice [23]. Therefore, we shall not give explicit inference rules for checking or deciding subtyping,

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{}{\Gamma \vdash () : ()} \quad \frac{}{\Gamma \vdash w : \texttt{string}}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e, e' : \tau, \tau'} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash n[e] : n[\tau]}$$

$$\frac{\bar{x}{:}\alpha \in \Gamma}{\Gamma \vdash \bar{x} : \alpha} \quad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\sigma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash c : \texttt{bool} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1|\tau_2}$$

$$\frac{\bar{x}{:}n[\tau] \in \Gamma}{\Gamma \vdash \bar{x}/\texttt{child} : \tau} \quad \frac{\Gamma \vdash e : \tau \quad \tau :: n \Rightarrow \tau'}{\Gamma \vdash e :: n : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \rightarrow e_2 : \tau_2}{\Gamma \vdash \texttt{for}\ \bar{x} \in e_1\ \texttt{return}\ e_2 : \tau_2} \quad \frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

$$\boxed{\tau :: n \Rightarrow \tau'}$$

$$\frac{}{n[\tau] :: n \Rightarrow n[\tau]} \quad \frac{\alpha \neq n[\tau]}{\alpha :: n \Rightarrow ()} \quad \frac{}{() :: n \Rightarrow ()} \quad \frac{\tau_1 :: n \Rightarrow \tau_2}{\tau_1^* :: n \Rightarrow \tau_2^*}$$

$$\frac{\tau_1 :: n \Rightarrow \tau_1' \quad \tau_2 :: n \Rightarrow \tau_2'}{\tau_1, \tau_2 :: n \Rightarrow \tau_1', \tau_2'} \quad \frac{\tau_1 :: n \Rightarrow \tau_1' \quad \tau_2 :: n \Rightarrow \tau_2'}{\tau_1|\tau_2 :: n \Rightarrow \tau_1'|\tau_2'}$$

$$\boxed{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau \rightarrow e : \tau'}$$

$$\frac{}{\Gamma \vdash \bar{x}\ \texttt{in}\ () \rightarrow e : ()} \quad \frac{\Gamma, \bar{x}{:}\alpha \vdash e : \tau}{\Gamma \vdash \bar{x}\ \texttt{in}\ \alpha \rightarrow e : \tau} \quad \frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \rightarrow e : \tau_2}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1^* \rightarrow e : \tau_2^*}$$

$$\frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \rightarrow e : \tau_1' \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_2 \rightarrow e : \tau_2'}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1, \tau_2 \rightarrow e : \tau_1', \tau_2'}$$

$$\frac{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1 \rightarrow e : \tau_1' \quad \Gamma \vdash \bar{x}\ \texttt{in}\ \tau_2 \rightarrow e : \tau_2'}{\Gamma \vdash \bar{x}\ \texttt{in}\ \tau_1|\tau_2 \rightarrow e : \tau_1'|\tau_2'}$$

Figure 7. Query well-formedness.

but treat it as a "black box". We also consider subtyping relations between tree types and tests: we say that $\alpha <: \phi$ if $[\![\alpha]\!] \subseteq [\![\phi]\!]$. This is decidable using the following rules:

$$\frac{}{\texttt{bool} <: \texttt{bool}} \quad \frac{}{\texttt{string} <: \texttt{string}} \quad \frac{}{n[\tau] <: n} \quad \frac{}{n[\tau] <: *}$$

We consider contexts Γ of the form

$$\Gamma ::= \cdot \mid x{:}\tau \mid \bar{x}{:}\alpha$$

that is, tree variables may only be bound to tree types. As usual, we assume that variables in contexts are distinct; this convention implicitly constrains the inference rules. We write $[\![\Gamma]\!]$ for the set of all environments σ such that $\sigma(X) \in \Gamma(X)$ for all $X \in dom(\Gamma)$.

The typing judgment for queries is $\Gamma \vdash e : \tau$ (that is, in context Γ, expression e has type τ); following [14], there are two auxiliary judgments, $\Gamma \vdash \bar{x}\ \texttt{in}\ \tau \rightarrow s : \tau'$, used for typechecking `for`-expressions, and $\tau :: n \Rightarrow \tau'$, used for typechecking label matching expressions $e :: n$. The rules for these judgment are shown in Figure 7. There are two typing judgments for updates: singular well-formedness $\Gamma \vdash^1 \{\alpha\}\ s\ \{\tau'\}$ (that is, in context Γ, update u maps tree type α to type τ'), and plural well-formedness $\Gamma \vdash^* \{\tau\}\ s\ \{\tau'\}$ (that is, in context Γ, update u maps type τ to type τ'). Several of the rules are polymorphic with respect to the arity $a \in \{1, *\}$. In addition, there is an auxiliary judgment $\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}$ for typechecking iterations. The rules for update well-formedness are shown in Figure 8.

$$\boxed{\Gamma \vdash^a \{\tau\}\ s\ \{\tau'\}}$$

$$\frac{}{\Gamma \vdash^a \{\tau\}\ \texttt{skip}\ \{\tau\}} \quad \frac{\Gamma \vdash^a \{\tau\}\ s\ \{\tau'\} \quad \Gamma \vdash^a \{\tau'\}\ s'\ \{\tau''\}}{\Gamma \vdash^a \{\tau\}\ s; s'\ \{\tau''\}}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash^a \{\tau\}\ s\ \{\tau_1\} \quad \Gamma \vdash^a \{\tau\}\ s'\ \{\tau_2\}}{\Gamma \vdash^a \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s'\ \{\tau_1|\tau_2\}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash^a \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash^a \{\tau_1\}\ \texttt{let}\ x = e\ \texttt{in}\ s\ \{\tau_2\}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash^* \{()\}\ \texttt{insert}\ e\ \{\tau\}} \quad \frac{}{\Gamma \vdash^a \{\tau\}\ \texttt{delete}\ \{()\}}$$

$$\frac{\Gamma, x{:}\tau \vdash^a \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^a \{\tau\}\ \texttt{snapshot}\ x\ \texttt{in}\ s\ \{\tau'\}} \quad \frac{\alpha <: \phi}{\Gamma \vdash^1 \{\alpha\}\ \phi?s\ \{\tau\}}$$

$$\frac{\alpha \not<: \phi}{\Gamma \vdash^1 \{\alpha\}\ \phi?s\ \{\alpha\}} \quad \frac{\Gamma \vdash^* \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^1 \{n[\tau]\}\ \texttt{children}[s]\ \{n[\tau']\}}$$

$$\frac{\Gamma \vdash^* \{()\}\ s\ \{\tau'\}}{\Gamma \vdash^a \{\tau\}\ \texttt{left}[s]\ \{\tau', \tau\}} \quad \frac{\Gamma \vdash^* \{()\}\ s\ \{\tau'\}}{\Gamma \vdash^a \{\tau\}\ \texttt{right}[s]\ \{\tau, \tau'\}}$$

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}{\Gamma \vdash^* \{\tau\}\ \texttt{iter}[s]\ \{\tau'\}} \quad \frac{\tau_1 <: \tau_1' \quad \Gamma \vdash^* \{\tau_1'\}\ s\ \{\tau_2'\} \quad \tau_2' <: \tau_2}{\Gamma \vdash^* \{\tau_1\}\ s\ \{\tau_2\}}$$

$$\boxed{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ s\ \{\tau'\}}$$

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\} \quad \Gamma \vdash_{\texttt{iter}} \{\tau_2\}\ s\ \{\tau_2'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1, \tau_2\}\ s\ \{\tau_1', \tau_2'\}}$$

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\} \quad \Gamma \vdash_{\texttt{iter}} \{\tau_2\}\ s\ \{\tau_2'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1|\tau_2\}\ s\ \{\tau_1'|\tau_2'\}}$$

$$\frac{\Gamma \vdash_{\texttt{iter}} \{\tau_1\}\ s\ \{\tau_1'\}}{\Gamma \vdash_{\texttt{iter}} \{\tau_1^*\}\ s\ \{\tau_2^*\}} \quad \frac{\Gamma \vdash^1 \{\alpha\}\ s\ \{\tau\}}{\Gamma \vdash_{\texttt{iter}} \{\alpha\}\ s\ \{\tau\}} \quad \frac{}{\Gamma \vdash_{\texttt{iter}} \{()\}\ s\ \{()\}}$$

Figure 8. Update well-formedness.

5.2 Type soundness and decidability of typechecking

We take for granted the following type soundness property for queries (a similar property is shown for μXQ in Colazzo et al. [14]).

THEOREM 2 (Query soundness). *If* $\Gamma \vdash e : \tau$, $\sigma \in [\![\Gamma]\!]$ *and* $\sigma \vdash e \Rightarrow v$ *then* $v \in [\![\tau]\!]$.

The corresponding result also holds for updates, by a straightforward structural induction argument:

THEOREM 3 (Update soundness).

1. *If* $\Gamma \vdash^1 \{\alpha\}\ e\ \{\tau'\}$, $t \in [\![\alpha]\!]$, $\sigma \in [\![\Gamma]\!]$, *and* $\sigma; t \vdash e \Rightarrow^{\mathrm{U}} v'$ *then* $v' \in [\![\tau']\!]$.
2. *If* $\Gamma \vdash^* \{\tau\}\ e\ \{\tau'\}$, $v \in [\![\tau]\!]$, $\sigma \in [\![\Gamma]\!]$, *and* $\sigma; v \vdash e \Rightarrow^{\mathrm{U}} v'$ *then* $v' \in [\![\tau']\!]$.

We now consider the decision problem of checking types for queries and updates. The subsumption rule makes typechecking non-syntax-directed. However, given a context and query, we can calculate a unique syntax-directed result type, if it can be typed:

LEMMA 1. *Given* Γ *and* e*, there exists at most one* τ *such that* $\Gamma \vdash e : \tau$ *has a subsumption-free derivation.*

Similarly, given a context, update, and initial type, we can calculate a unique syntax-directed output type:

LEMMA 2. *Given* Γ*,* a*,* τ*, and* s*, there exists at most one* τ' *such that* $\Gamma \vdash^a \{\tau\}\ e\ \{\tau'\}$ *has a subsumption-free derivation.*

Since there are no functions or function types in the query and update languages, we believe it is possible to show that all uses of the subsumption rules can be permuted down in a derivation tree:

CONJECTURE 1 (Subsumption). *If a query or update typing judgment is derivable, it is derivable using at most one instance of the subsumption rule, at the root of the tree.*

Conjecture 1 would imply the completeness of the following algorithm for typechecking a query or update against a fixed context and input/output types: first, calculate the result type using only the syntax-directed rules (without subsumption), then test whether this is a subtype of the desired output type. This algorithm is sound whether or not Conjecture 1 holds.

Type inference appears trickier, but is not crucial in a database update setting, because of the absence of function types and because we always know the schema of an XML database statically (assuming the database has a schema). Nevertheless, the ability to typecheck a query or update in isolation from a database schema is potentially useful, and type inference and principal types should be investigated.

6. Extensions and future work

6.1 Recursive types and nominal subtyping

Our type system is weaker than other XML type systems in several respects. It does not include type variables or recursive type definitions, so although it permits *horizontal* iteration using Kleene star, *vertical* recursion is not supported. The type system also employs structural subtyping on unannotated XML data, in contrast to the XML Schema/XQuery type system which employs nominal subtyping on validated XML documents in which each element is tagged with a type name according to the XML Schema validation algorithm.

Although these are significant restrictions relative to full DTDs, XML Schemas, or XDuce regular expression types, we believe they are a reasonable given that LUX is meant to be used for database-style updates, not queries, stylesheet transformations, or general-purpose programming with XML. Many DTDs and XML Schemas encountered in database applications of XML are nonrecursive and "shallow" [12, 4].

On the other hand, there are use cases for XML updates involving recursion and XML Schemas [26] which we would like to handle. Extending the system to include recursive queries and recursive regular expression types appears to be straightforward; adding recursive *updates* may not be. Adapting our approach to updates to be compatible with XML Schema-validated documents and the XQuery nominal subtyping system seems difficult.

6.2 Transformations

Some of the existing update proposals include a facility for running an update operation within an XQuery expression in a side-effect-free way. Such a facility can easily be added using LUX updates:

$$e ::= \cdots \mid \texttt{transform}\ e\ \texttt{by}\ s$$

with static and dynamic semantics given by

$$\frac{\sigma \vdash e \Rightarrow v \quad \sigma; v \vdash s \Rightarrow^{\mathtt{U}} v'}{\sigma \vdash \texttt{transform}\ e\ \texttt{by}\ s \Rightarrow v} \qquad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash^{*} \{\tau_1\}\ s\ \{\tau_2\}}{\Gamma \vdash \texttt{transform}\ e\ \texttt{by}\ s : \tau_2}$$

6.3 Update optimization

Liefke and Davidson [25] investigated a number of optimization techniques for CPL+, including rewriting updates using equational laws that provably improve performance (relative to an appropriate cost model), and transforming query-based updates to more efficient in-place updates ("deltafication"). These techniques should also be applicable to LUX. For example, the equational laws shown in Figure 9 are clearly valid for LUX.

$$\begin{array}{rcl}
\phi?u_1;\phi?u_2 & \equiv & \phi?(u_1;u_2)\\
d[u_1];d[u_2] & \equiv & d[u_1;u_2]\\
\texttt{snapshot}\ x\ \texttt{in}\ \texttt{snapshot}\ y\ \texttt{in}\ e & \equiv & \texttt{snapshot}\ x\ \texttt{in}\ e[x/y]\\
\texttt{snapshot}\ x\ \texttt{in}\ e & \equiv & e \quad (x \notin FV(e))\\
s;\texttt{delete} & \equiv & \texttt{delete}\\
d[\texttt{if}\ c\ \texttt{then}\ s_1\ \texttt{else}\ s_2] & \equiv & \texttt{if}\ c\ \texttt{then}\ d[s_1]\ \texttt{else}\ d[s_2]
\end{array}$$

Figure 9. Some equational laws for updates.

6.4 Correctness analysis

Colazzo et al. [14] studied both "result analysis" (determining the result type of a query given types for its input variables) and "correctness analysis" (ensuring that no subexpression of the query is statically empty, except for ()). They introduced a type system that statically performs both result and correctness analysis; moreover, the correctness analysis is *complete*: it precisely characterizes queries as either correct or incorrect. This problem is decidable for μXQ since it lacks recursion.

Correctness analysis is potentially very useful in debugging queries, since often a bug manifests itself as an empty subquery; such bugs are generally not caught by result analysis since the empty sequence is a subtype of every type. Colazzo et al.'s type system for query correctness analysis can clearly also be used in LUX updates to help avoid bugs in subqueries. Moreover, it seems reasonable to extend their concept of correctness to updates by judging an update to be correct only if every sub-statement that is statically guaranteed to have no effect on the database is literally a no-op (`skip`). Updates can easily be incorrect: for example, if a test update `bool?`s is only run against data of type `string`, then s will never be executed; also s will never be typechecked, so might contain nonsense. Such "dead" code is likely a bug in the update, and should be flagged as a warning to the programmer. It would be very interesting to develop a type system for update correctness, especially a complete one.

6.5 Designing a high-level update language

In the first half of this paper we motivated and informally described a high-level update language that we believe can be systematically translated to the core LUX language, just as full XQuery is defined by translation to a much smaller core language. In addition we have shown how a number of examples can be so translated. However, the high-level language design we have outlined is just a strawman. Its SQL-style syntax and somewhat random collection of primitives leave a lot to be desired. We hope to gain the kind of practical experience with an implementation and large-scale examples that seems prerequisite to improving the design.

7. Conclusions

In this paper, we have introduced a simple approach to database-style updates for XML, called LUX (Lightweight Updates for XML). Our approach is inspired in large part by the update language CPL+ for nested relational data introduced by Liefke and Davidson. Our approach, like CPL+, factors complicated operations similar to SQL's `INSERT`, `DELETE`, and `UPDATE` statements into a small, orthogonal set of operations which perform basic updates, navigate, examine the local database structure, or combine updates. LUX is carefully designed so that each update's side-effects are confined to a particular subtree, so that the result of an iterative update is independent of the order in which the updates are performed. Variables are immutable; updates can only mutate the database state. Side-effecting updates are syntactically distinguished from purely functional queries.

These design decisions make it possible to give updates a simple, deterministic semantics. In contrast, the semantics of more sophisticated update languages involve two passes, one to collect atomic updates evaluated against a snapshot of the database, and another to perform the updates. In addition, it is possible to provide a type system for the core update language which can be used to check that an update has desired input-output behavior. We believe that this is the first type system for any XML update language.

Besides proving type soundness, we have described a simple, sound typechecking algorithm that computes an upper bound on the type of a query or update, then checks whether the bound is included in the desired result type. We believe this algorithm is complete, but have not fully verified this conjecture.

LUX is work in progress, and in the near future we plan to prove Conjecture 1, improve the design of the high-level interface, define the translation to the core language, and experiment with large-scale examples (including some of the W3C's use cases [26]) inside an XML database management system. Additional possible areas for future work include static typechecking relative to XQuery's type system; considering recursive types and updates; and developing a type system capturing correctness.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations Of Databases*. Addison-Wesley, 1995.

[2] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In Daniela Florescu and Hamid Pirahesh, editors, *XIME-P*, 2005.

[3] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Verification of tree updates for optimization. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2005.

[4] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML schema: a practical study. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM Press.

[5] Scott Boag, Don Chamberlin, Mary F. Fernndez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Candidate Recommendation, June 2006. `http://www.w3.org/TR/xquery`.

[6] Emmanuel Bruno, Jacques Le Maitre, and Elisabeth Murisasco. Extending XQuery with transformation operators. In *DocEng '03: Proceedings of the 2003 ACM Symposium on Document Engineering*, pages 1–8, New York, NY, USA, 2003. ACM Press.

[7] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

[8] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. *SIGPLAN Not.*, 40(1):271–282, 2005.

[9] Don Chamberlin, Mike Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robie. XQueryP: Programming with XQuery. In *XIME-P*, 2006.

[10] Don Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery update facility. W3C Working Draft, July 2006. `http://www.w3c.org/TR/xqupdate/`.

[11] Don Chamberlin and Jonathan Robie. XQuery update facility requirements. W3C Working Draft, June 2005. `http://www.w3.org/TR/xquery-update-requirements/`.

[12] Byron Choi. What are real DTDs like? In *WebDB*, pages 43–48, 2002.

[13] J. Clark. XSL transformations (XSLT). W3C Recommendation, November 1999. `http://www.w3.org/TR/xslt`.

[14] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Types for path correctness of XML queries. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 126–137, New York, NY, USA, 2004. ACM Press.

[15] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. Unpublished manuscript, June 2006.

[16] D. Chamberlin et al. XQuery 1.0 and XPath 2.0 formal semantics. W3C Candidate Recommendation, June 2006. `http://www.w3.org/TR/xquery-semantics/`.

[17] David C. Fallside (Ed). XML Schema Part 0: Primer. W3C Recommendation, October 2004. `http://www.w3.org/TR/xmlschema-0`.

[18] Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 263–300, London, UK, 2001. Springer-Verlag.

[19] Vladimir Gapeyev, François Garillot, and Benjamin C. Pierce. Statically typed document transformation: An Xtatic experience. In Giuseppe Castagna and Mukund Raghavachari, editors, *PLAN-X*, pages 2–13. BRICS, Department of Computer Science, University of Aarhus, 2006.

[20] G. Ghelli, C. Ré, , and J. Siméon. XQuery!: An XML query language with side effects. In *Proc. of Workshop on Database Technologies for Handling XML Information on the Web (DataX)*, 2006. To appear.

[21] G. Ghelli, K. Rose, , and J. Siméon. Commutativity analysis in XML update languages. In *ICDT*, 2007. To appear.

[22] Jan Hidders, Jan Paredaens, and Roel Vercammen. On the expressive power of XQuery-based update languages. In Sihem Amer-Yahia, Zohra Bellahsene, Ela Hunt, Rainer Unland, and Jeffrey Xu Yu, editors, *XSym*, volume 4156 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2006.

[23] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[24] Andreas Laux and Lars Martin. XUpdate - XML update language. `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`, September 2000. Work in progress.

[25] Hartmut Liefke and Susan B. Davidson. Specifying updates in biomedical databases. In *SSDBM*, pages 44–53, 1999.

[26] Ioana Manolescu and Jonathan Robie. XQuery update facility use cases. W3C Working Draft, May 2006. `http://www.w3.org/TR/xqupdateusecases`.

[27] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[28] Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In Thomas Eiter and Leonid Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[29] Jérôme Siméon and Philip Wadler. The essence of XML. *SIGPLAN Not.*, 38(1):1–13, 2003.

[30] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[31] Gargi Sur, Joachim Hammer, and Jérôme Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[32] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *SIGMOD Conference*, pages 413–424, 2001.

[33] Guoren Wang, Mengchi Liu, and Li Lu. Extending XML-RL with update. In *IDEAS*, pages 66–75, 2003.