Updating Typed XML Documents Using a Functional Data Model

Pavel Loupal

Dept. of Computer Science and Engineering
Faculty of Electrical Engineering, Czech Technical University
Karlovo nám. 13, 121 35 Praha 2
Czech Republic
loupalp@fel.cvut.cz

Abstract. We address a problem of updating XML documents having their XML schema described by a Document Type Definition (DTD) without breaking their validity. We present a way how to express constructs available in DTD in a functional data model and propose algorithms for performing insert, update and delete operations. After that we embed the update capability into an existing query language for XML. This paper thus outlines the whole "life cycle" of the approach from the problem analysis to its implementation.

1 Motivation and Problem Statement

During our work on a functional framework for querying XML – XML-λ – we identified a need for extending the language with support of data modification operations. Our aim is to develop an approach similar to the SQL language for relational databases, i.e. have an ability both to query and update underlying data.

With respect to our aim we set up basic requirements for our approach. First, we always consider *typed* data (this is a natural requirement because of the fact that our framework is based on a type system). At this stage we use DTD for constraining document validity. Second, we have already a query language designed. It makes sense to extend this language in a "logical" way with update operations. By the term "logical" we mean the utilization of existing constructs as sets, existing type system and the idea of functional approach in general.

The paper is structured as follows: Section 2 lists existing approaches for updating XML data and discusses their contribution. In Section 3 we briefly outline the concept of the functional framework we use, its data model and the query language that is used for implementing the proposal. We discuss the problem in Section 4 where we show our solution. Section 5 deals with enriching the syntax of our query language with update operations. In Section 6 we conclude with ideas for future work.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2007, pp. 11–20, ISBN 80-7378-002-X.

2 Languages for Updating XML

By the term *updating XML* we mean the ability of a language to perform modifications (i.e. insert, update and delete operations, etc.) over a set of XML documents.

Since the creation of the XML in 1998 there have been many efforts to develop various data models and query languages. A lot of time has also been spent on indexing and query optimization. On the other hand the problem of updating XML gains more interest in few past years. Yet there seems to be not a complete solution for this problem.

Existing papers dealing with updating XML are mostly related to XQuery [2] (and the need for having updates in XQuery is also considered as one of the most important topics in the further development of the language [4]). Lehti [7] proposes an extension to XQuery that allows all update operations but does not care about the validity of the documents. Tatarinov, et al. [11] also extends XQuery syntax with insert, update and delete operations and shows the implementation of storage in a relational database system. Benedikt, et al. [1, 10] deals in deep with the semantics of updates in XQuery.

For the sake of completeness we should not omit XUpdate [6] – a relatively old proposal that takes a different way. It uses XML-based syntax for describing update operations. This specification is probably less formal than those previous but it is often used in praxis.

Considering previous works we can deduce that there are common types of operations for performing modifications that are embedded in a language – delete, update, insert before or insert after. This seems to be a sufficient base for ongoing work. None of those proposals but deals with the problem of updating *typed data* and thus it makes sense to put some effort into studying of this problem. The evolution process around XML leads to use of types so it makes sense to work on this problem in the world of typed XML documents.

3 XML-λ Framework

XML-λ is a proposal published in 2001 by Pokorný [8, 9]. In contrast to W3C languages it uses *functional* data model instead of tree- or graph-oriented model. The primary motivation was to see XML documents as a database that conforms to some XML schema (defined, for example, by DTD).

The framework is based on type system theory – it can be informally said that first a "base" type system T_{base} is defined then a regular type system T_{reg} that extends T_{base} with regular types is induced. Upon this the T_{reg} is enriched with types corresponding to an XML schema and the T_E type system is defined. Over such type system we define a query (and update) language based on simply typed lambda calculus.

3.1 Type System Introduction

Type system is built on base $\mathcal{B}$ – a set containing finite number of base types $S_1, \ldots, S_k$ $(k \geq 1)$. Type hierarchy is then created by following inductive definition:

Definition 1. *Let $\mathcal{B}$ is a set of primitive types $S_1 \ldots S_n, k \geq 1$.* Type System *T_{base} over base $\mathcal{B}$ is the least set containing types given by 1.-4.*

1. base type: *each member of $\mathcal{B}$ is type over $\mathcal{B}$*
2. functional type: *if T_1 and T_2 are types over $\mathcal{B}$, then $(T_1 \rightarrow T_2)$ is also a type over $\mathcal{B}$*
3. n-tuple type*: if $T_1, \ldots, T_n$ $(n \geq 1)$ are types over $\mathcal{B}$, then $(T_1, \ldots, T_n)$ is type over $\mathcal{B}$*
4. union type: *if $T_1, \ldots, T_n$ $(n \geq 1)$ are types over $\mathcal{B}$, then $(T_1 + \ldots + T_n)$ is type over $\mathcal{B}$*

Subsequently we define a regular type system T_{reg} that extends type system T_{base} with regular constructs:

Definition 2. *Let $\mathcal{B}$= $\{String, Bool\}$, let $NAME$ be a set of names. Type System T_{reg} is the least set containing types given by 1.-6.*

1. *Every member of the base $\mathcal{B}$ is an (*primitive*) type over $\mathcal{B}$.*
2. named character data: *Let $tag \in NAME$. Then tag : $String$ is an (*elementary*) type over $\mathcal{B}$,*
   *tag : is an (*empty elementary*) type over $\mathcal{B}$.*
3. *Let T be a group type or named character data. Then*
   - zero or more: *$T*$ is a type over $\mathcal{B}$.*
   - one or more: *$T+$ is a type over $\mathcal{B}$.*
   - zero or one: *$T?$ is a type over $\mathcal{B}$.*
4. alternative: *Let T_1 and T_2 be types. Then $(T_1|T_2)$ is a type over $\mathcal{B}$.*
5. sequence: *Let $T_1, \ldots, T_n$ be types. Then $(T_1, \ldots, T_n)$ is a type over $\mathcal{B}$.*
6. named type: *Let T be a type given by a step from 3.-5. Let $tag \in NAME$. Then $tag : T$ is a type over $\mathcal{B}$.*

3.2 Binding Types to XML

Having the T_{reg} type system we have to extend it to be able to work with XML data. We build the type system T_E induced by T_{reg}. Key idea is to define *abstract items* that are particular XML elements or attributes with some content and also define a set containing all abstract items within an XML instance – **E**.

Definition 3. *Let T_{reg} over base $\mathcal{B}$ be a type system from definition 2 and E is the set of abstract items. Then* type system *T_E induced by T_{reg} is the least set containing type given by this rule:*

Let $tag : T \in T_{reg}$. Then $TAG : T$ is a member of T_E. (Replacement of all tags in $tag : T$ by uppercase version)

With types from T_E we can consider functional types for extracting data values from elements (via abstractions and projections) with two ways

1. for *simple element:* if $tag : String \in T_{reg}$, then $(\mathbf{E} \rightarrow tag : String) \in T_E$
2. for *compound element:* if $tag : T \in T_{reg}$, then $(\mathbf{E} \rightarrow T') \in T_E$

Note also that in T_E we can express attributes in the same way as XML elements – as functions.

3.3 Query Language Construction

Typical query has the query part – an expression to be evaluated over data – and the constructor part that wraps query result and forms the XML output. XML-λ's query language is based on λ-terms defined over the type system T_E as shown in Definition 4.

Main constructs of the language are *variables*, *constants*, *tuples*, use of *projections* and λ-calculus operations – *applications* and *abstractions*. Tagged terms might be used for declaring functions. Syntax of this language is similar to λ-calculus expression i.e. $\lambda \ldots (\lambda \ldots (expression) \ldots)$. In addition, there are also typical constructs such as logical connectives, constants or comparison predicates.

Language of terms is inductively defined as the least set containing all terms created by application of following rules:

Definition 4. *Let $T, T_1, \ldots, T_n, n \geq 1$ be members of T_{base}. Then*

1. variable: *each variable of type T is a term of type T*
2. constant: *each constant (member of $\mathcal{F}$) of type T is a term of type T*
3. application: *if M is a term of type $((T_1, \ldots, T_n) \rightarrow T)$ and $N_1, \ldots, N_n$ are (in the same order) types $T_1, \ldots, T_n$, then $M(N_1, \ldots, N_n)$ is a term of type T*
4. λ-abstraction: *if $x_1, \ldots, x_n$ are distinct variables of types $T_1, \ldots, T_n$ and M is a term of type T, then $\lambda x_1, \ldots, x_n(M)$ is a term of type $((T_1, \ldots, T_n) \rightarrow T)$*
5. n-tuple: *if $N_1, \ldots, N_n$ are terms of types $T_1, \ldots, T_n$, then $(N_1, \ldots, N_n)$ is a term of type $(T_1, \ldots, T_n)$*
6. projection: *if $(N_1, \ldots, N_n)$ is a term of type $(T_1, \ldots, T_n)$, then $N_1, \ldots, N_n$ are terms of types $T_1, \ldots, T_n$*
7. tagged term: *if N is a term of type $NAME$ and M is a term of type T then $N : T$ is a term of type $(E \rightarrow T)$.*

3.4 Query Example

For our purposes we use the notoriously known bibliography example DTD from the XML Query Use Cases [5] document. We also consider XML data provided in the same document.

A query returning all books published by "Addison-Wesley" is in XML-λ expressed as shown in Figure 1.

```
xmldata("bib.xml")
lambda b ( /book(b) and  b/publisher = "Addison-Wesley" )
```

Fig. 1. An example query written in XML-λ

3.5 Data Model Summary

Previous sections outline the definition of type system T_E that we use for modelling types in an XML schema. This means that for each DTD we can construct a particular type system of respective types. In the *Language of terms* we propose a mechanism based on lambda calculus operations (applications and abstractions) combined with projections to work with XML documents.

The most important idea in the framework is the fact that even the smallest piece of information in an XML document (e.g. an attribute of element containing just a PCDATA value) is modelled as a partial function that assigns a value for exactly one $e \in \mathbf{E}$. For example, having an XML element `<phone>+420-800123456</phone>` there is a function `phone(e)` that for exactly one $e \in \mathbf{E}$ returns value `+420-800123456`. For more complex types, e.g. `<!ELEMENT author (last, first)>` the result of the function is a Cartesian product $\mathbf{E} \times \mathbf{E}$.

In XML-λ we model each XML document by a *set of items* $\mathbf{E}$ where each $e \in \mathbf{E}$ is of type T_{ITEM}. T_{ITEM} is a type consisting of a couple $(t : TYPE, uid : INT)$; $TYPE \in T$ and *uid* is an integer value for maintaining order of items in the set. Note that some types in a particular type system can have related information attached (each item of type PCDATA has attached a value of the item – its content).

In following text we consider following semantic functions with informal meaning as summarized in following table:

Semantic Function	Behaviour
parent(e)	For an $e \in \mathbf{E}$ return its parent item
type(e)	For an $e \in \mathbf{E}$ return its type t ($t \in \mathbf{T}$)
application(e, t)	Executes an application (rule 3 in Definition 4) of t-object to the e item. In general it returns a Cartesian product of $\mathbf{E} \times \ldots \times \mathbf{E}$
projection(n-tuple, t)	Retrieves all items of type t from given n-tuple.
childTypes(t)	Retrieves an list of types (sorted by document order) that might be contained in the result of application of a t-object

For further usage we present an algorithm of traversing a fragment of XML data utilizing our functional framework. The algorithm *traverse(***E***, e, op)* takes three parameters, **E** – set of items (this represents an XML document in our

model) and e – start-up item for traversing, *op* – an operation to be performed on each node.

```
ALGORITHM traverse(E, e, op)
1: Initialize stack S;
2: Mark e as NEW; Push e to stack S;
3: while (any NEW or OPEN node in S)
4:   Pop i from S; Mark i as OPEN;
5:   Type t = type(i);
6:   n-tuple nt = application(i, t);
7:   List_of_types lt = childTypes(t);
8:   if lt is String
9:     op(i);
10:    Mark i as CLOSED;
     else
11:    For each type in lt
12:     n-tuple nt = application(i, type);
13:     Mark all items as NEW and push to S;
```

4 Updating Typed Documents

Document Type Definition (DTD) [3] is a syntactic way how to describe a *valid* XML instance. We can break all DTD features into disjoint categories:

1. Elements constraints – Specify the type of an element content. Is one of `EMPTY`, `ANY`, `MIXED` or `ELEMENT_CONTENT`,
2. Structure constraints – The occurrence of elements in a content model. Options are *exactly-one*, *zero-or-one*, *zero-or-more*, *one-or-more*
3. Attributes constraints – `#REQUIRED`, `#IMPLIED`, `#FIXED`, ID, IDREF(S)

Each update operation can or cannot affect any construct from the particular DTD. Considering a transactional behaviour we can see two violation scenarios:

1. *Fully consistent.* After each operation (insert, update or delete) the instance remains valid. This means that we have to define a complete set of operations that are strong enough to perform all possible updates.
2. *Partially consistent.* In this mode we allow partial inconsistency i.e. we consider the whole query as an atomic operation. Therefore we do not require to have atomic insert,update and delete operations but we have to ensure that at the end of the processing the instance is valid. In general it means revalidation of the document being updated.

In our approach we use the first scenario and declare all operations as *fully consistent*.

With respect to abilities of the existing XML-λ framework we have to extend this framework with features allowing us to check constraints available in DTD.

The cornerstone of the framework is its type system (it is the basis of types we can use). For modelling DTD constraints we propose four *sets of types*, where all types come from the type system, i.e. $T \in T_E$.

1. $T_{unmodifable}$ is a set of types that cannot be modified. This set contains types for attributes declared as `#FIXED` and element types with `EMPTY` content model.
2. $T_{mandatory}$ is a set of types that must not be removed from a document instance because it would break the DTD constraints. This set contains attribute types with `#REQUIRED` declaration and element types for those T iff all occurrences of T are exactly-one.
3. $T_{referencing}$ is a set of types that may reference another type, for DTDs those are attributes declared as `IDREF` or `IDREFS`.
4. $T_{referenced}$ is a set of types that may be referenced by another type, for DTDs those are attributes declared as `ID`.

These sets we use in our semantics for particular update operations. We will use access functions *isUnmodifable(e)*, *isMandatory(e)*, *isReferencing(e)* and *isReferenced(e)* that check the containment of item's type in respective sets.

In general the semantics of all operations consists of two parts: (1) Check if the operation is permitted regarding the DTD constraints and (2) Execution of given update operation. Following sections discuss the semantics of delete, insert and update operations in detail.

4.1 Delete

Deletion is a operation of removing given part of XML data (i.e. element or attribute) with its potential subelement(s). We can see the function with a signature `DELETE(`e`)` where $e : \mathbf{t} \in \mathbf{E}$, $\mathbf{t} \in T$.

With respect to validity issues there are two scenarios where this operation is denied:

1. `e` is an *attribute* and is declared as `#REQUIRED` or `#FIXED`
2. `e` is an *element* with *exactly-one* or *one-or-more* occurrence

In our framework it means checking whether the type of item being deleted is a member of $T_{mandatory}$ or $T_{unmodifiable}$ sets. After that we have to ensure that by deleting of the item we do not delete the last remaining item with *exactly-one* or *one-or-more* occurrence. Following algorithm outlines conceptually the operation.

Algorithm *delete(***E***, e)* takes two parameters, **E**– set of items (this represents an XML document in our model) and e – the item to be deleted. It returns *true* if the item has been deleted or *false* if the delete has been denied.

```
ALGORITHM delete(E, e):
1: if (isMandatory(e) or isUnmodifieble(e))
2:    return false;
```

```
   else
3:    p = parent(e);
4:    List_of_types pt = childrenTypes(p);
5:    if (type(e) in pt) is exactly-one
6:       return false;
7:    if (type(e) in pt) is one-or-more and
8:        data contains at least 1 occurrence of item with type(e)
9:        return false;
      else
10       traverse(E, e, delete);
11:      return true;
```

Informally we can imagine the operation as a subset subtraction $\mathbf{E_{result}} = \mathbf{E} \setminus e$ with ongoing renumbering of remaining items to keep document order. Maintenance of potential references in document (attributes of type ID, IDREF and IDREFS) that should take place in the `traverse` function. For now we consider it as being out scope of this paper.

4.2 Insert

By the insert operation we mean adding an XML fragment into the target document. By the fragment we understand an element, an attribute or an XML subtree. Insert is more complicated operation than delete because there are more conditions and restrictions to be checked. We write the statement as

`INSERT` e_1 (`AFTER` | `BEFORE` | `AS CHILD`) e_2`;`

where $e_1 : T_1, e_2 : T_2 \in \mathbf{E}$; T_1, T_2 are types. Note that e_1 must be a valid expression in T, i.e. it must be of a type from T_E. We can see the operations as $f_{insert}(F, \mathbf{E_1}, \mathbf{E_i}) = \mathbf{E_{result}}$ where $F \in T_E$ and $\mathbf{E_{result}} = \mathbf{E_1} \cup \mathbf{E_i}$

Algorithm *insert_after(***E***,* e_1*,* e_2*)* takes three parameters, $\mathbf{E}$ – set of items, e_1 – the item to be inserted ($e_1 \notin \mathbf{E}$) and e_2 – the context item. It returns *true* if the item has been inserted or *false* if the insert has been denied.

```
ALGORITHM insert_after (E, e1, e2):
1: p = parent(e1);
2: List_of_types pt = childrenTypes(p);
3: if (type(e1) not in pt)
4:    return false;
5: if (following type of e2) is exactly-one and
6:     item with the same type already exists in n-tuple
7:    return false;
8: Put item into E and perform uid renumbering;
9: return true;
```

4.3 Update

Update operation means replacing one item by another with the same type. We can write the signature of this operation as `UPDATE` e_1 `WITH` e_2`;` where $e_1 : T_1, e_2 : T_2 \in \mathbf{E}$; $T_1 = T_2$ are types.

Because of the fact that we require both expressions to be of the same type there cannot occur any validation conflict (both of them are valid before the operation). The algorithm for performing update is then straightforward.

5 Implanting Updates into the Language

First implementation of the XML-λ language was developed in [12]. It is basically a query language without any updating capability. We will extend this language with operations as shown above.

With respect to the concept of the language we declare all operations as *tagged terms*. This means that we consider each function as a term of functional type $(\mathbf{E} \rightarrow T)$. Consequently we define the semantics for all operations.

Following fragment of EBNF shows a concept of including all operations into the language

[1] *Query* ::= *Options* (*OpUpdateList* | *OpQuery*)

[2] *OpQuery* ::= *ConstructorPart QueryBody* Eof

[3] *OpUpdateList* ::= { *OpUpdate* }+

[4] *OpUpdate* ::= { `Delete` *SubQuery* |
`Insert` *Expr* (`after|before|as child of`) *SubQuery* |
`Update` *SubQuery* `With` *Expr* }

Note that the non-terminal *SubQuery* (rule `[18]` in [12, p.55]) presents a lambda term that may return set of items. There is also a significant advantage of using non-terminals *SubQuery* and *Expr*. In general these can be function calls (even user-defined). This is a difference with XUpdate, where only XPath expressions are used.

An example of delete operation removing all books published by "Addison-Wesley" is then written as follows

```
xmldata("bib.xml")
delete( lambda b ( /book(b) and  b/publisher = "Addison-Wesley"))
```

The insert operation that adds a new element `author` after the first author of a book specified by its name we write as

```
xmldata("bib.xml")
insert-after( lambda a (
  /book(b) and  b/title="TCP/IP Illustrated" and a=b/author[1]),
  "<author><last>Richta</last><first>Karel</first></author>"))
```

An example of the delete operation is obvious and is omitted.

6 Conclusion and Future Work

We have shown a proposal for updating XML data constrained by a Document Type Definition. We present a functional framework for querying XML that is extended by structures for expressing DTD semantics. By enriching the query language with modification operations – inserts, deletes and updates – we obtain a language suitable both for querying and updating XML documents.

There is still a lot of future work ahead. To get a complete framework we have to finalize the issue with references within documents (IDs and IDREFS). This is only a technical problem how to formalize the algorithm to be executed to keep the documents consistent and valid. Another questionable area are the dependencies of multiple update operations in one "query" statement. In this paper we do not solve any potential conflicts.

Probably the biggest challenge for future work is replacement of DTD by XML Schema. This means restructuring the type systems T_{reg} and T_E and re-developing the idea of constraint sets.

References

1. M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P 2005*, 2005.
2. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language, September 2005. http://www.w3.org/TR/xquery/.
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (third edition), February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.
4. D. Chamberlin. XQuery: Where do we go from here? In *XIME-P 2006*, 2006.
5. D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases, September 2005. http://www.w3.org/TR/2005/WD-xquery-use-cases-20050915/.
6. A. Laux and L. Martin. XUpdate – XML Update Language, 2000. available online at http://xmldb-org.sourceforge.net/xupdate/index.html.
7. P. Lehti. Design and implementation of a data manipulation processor for an XML query language. Master's thesis, Technische Universitaet Darmstadt, 2001.
8. J. Pokorný. XML functionally. In B. C. Desai, Y. Kioki, and M. Toyama, editors, *Proceedings of IDEAS2000*, pages 266–274. IEEE Comp. Society, 2000.
9. J. Pokorný. XML-λ: an extendible framework for manipulating XML data. In *Proceedings of BIS 2002*, pages 160–168, Poznan, 2002.
10. G. M. Sur, J. Hammer, and J. Siméon. An XQuery-Based Language for Processing Updates in XML. In *PLAN-X 2004*, 2004.
11. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD 2001*, 2001.
12. P. Šárek. Implementation of the XML lambda language. Master's thesis, Dept. of Software Engineering, Charles University, Prague, 2002.