On the Semantics of Updates in a Functional Language *

© Pavel Loupal

Department of Computer Science,
Faculty of Electrical Engineering, Czech Technical University in Prague,
Prague, Karlovo nám. 13, 121 35

loupalp@fel.cvut.cz

Abstract

Issues related to updating data in native XML database systems are studied extensively nowadays. In this work we consider a problem of updating typed XML documents having their schema described by a Document Type Definition (DTD) without breaking their validity and with ensured transaction consistency. We present a way how to express constructs available in DTD by using a functional framework and propose algorithms for performing insert, replace and delete operations. This solution is an intermediate step we need for our ongoing research – formal comparison of XQuery and XML-λ.

1 Motivation and Problem Statement

Fundamental work we continue to work on is Pokorný's proposal of a functional framework for modeling and querying XML – XML-λ [15, 16]. The main idea therein is to use simply typed λ-calculus adherent to a DTD-based type system for querying XML data. Over time 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. to have an ability both to query and update underlying data within one formal apparatus.

This work directly continues in the topic that we have opened in [13]; in this text we clarify more the concept of the framework by showing its relationship to the W3C data model, reformulate proposed algorithms and we also add some improvements in formal description of the solution.

Nevertheless, our *primary motivation* is not to develop a totally new sort of an XML update language but rather to propose an update extension that allows us to go on with our planned research in the future – comparison of properties of XQuery and XML-λ and evaluation of potential mutual transformations of queries written in respective languages. We see the benefit of this paper particularly in clarification of proposed update algorithms and in specification of a link to transaction management.

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 show an example of query evaluation with detailed description. Then, we discuss the problem of updates in Section 4 in general and show our solution in Section 5. In Sections 6 and 7 we conclude with feasible ideas for future work.

2 Languages for Updating XML

By the term updating XML we mean the ability of a language to perform modifications (insert, replace and delete operations) over an XML document or a collection of XML documents.

Since the creation of the XML in 1998, there have been many efforts to develop various data models and query languages for databases of XML data. Multiple approaches for indexing and query optimizations have been invented. 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 [3]. Lehti [11] proposes an extension to XQuery that allows all update operations but does not care about the validity of the documents. Tatarinov et al. [18] also extend XQuery syntax with insert, update and delete operations and show the implementation of storage in a relational database system. Benedikt et al. [1] and Sur et al. [17] deal in deep with the semantics of updates in XQuery. In the W3C XML Query Working Group is the need for having updates in the language also considered as one of the most important topics in its further development [5]. As a result, the XQuery Update Facility has been proposed [6].

For the sake of completeness we should not omit XUpdate [10] – a relatively old proposal that takes a different way. It uses XML-based syntax for describing update operations. This specification is less formal than those previous but it is often used in practice.

Another research field is represented by XDuce [9] and its successor CDuce [2] that use also a type system based approach for pattern matching and manipulation of XML data.

Considering previous works we can deduce that there

* I would like to thank to Prof. Pokorný for his patience and provisioning of many helpful hints for my research.

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems, Saint-Petersburg, Russia, 2008

are common types of operations for performing modifications that are to be embedded in a language – delete, replace, insert-before, insert-after or insert-as-child. This seems to be a sufficient base for ongoing work. None of those proposals but deals in detail with the problem of updating typed data and hence it makes sense to put effort and study this problem.

3 XML-λ Framework

XML-λ is a proposal published by Pokorný [15, 16]. In contrast to W3C specifications it uses a functional data model instead of tree- or graph-oriented model. The primary motivation was to see XML documents as a database that conforms to an XML schema (defined, for example, by DTD) and to gain a possibility to use a functional language, particularly a simply typed λ-calculus, as a query language for such database.

Except of the original proposal, that defines its formal base and shows its usage primarily as a query language for XML, there is a consecutive work that introduces updates into the language available in [13].

Here, we focus primarily on extending and improving the update part of the framework. Basic facts about the framework are repeated in following sections rather for convenience.

3.1 Basic Terms

In XML-λ there are three important components related to its type system: *element types*, *element objects* and *elements*. We can imagine these components as the data dictionary in relational database systems. Note also Figure 1 for relationships of basic terms between W3C standards and the XML-λ Framework.

Element types are derived from a particular DTD and in our scenario they cannot be changed – we do not allow any schema changes but only data modifications. For each element defined in the DTD there exists exactly one element type in the set of all available element types (called T_E).

Consequently, we denote E as a set of *abstract elements*. Set members are of element types. Note that (from definition) E is an infinite set.

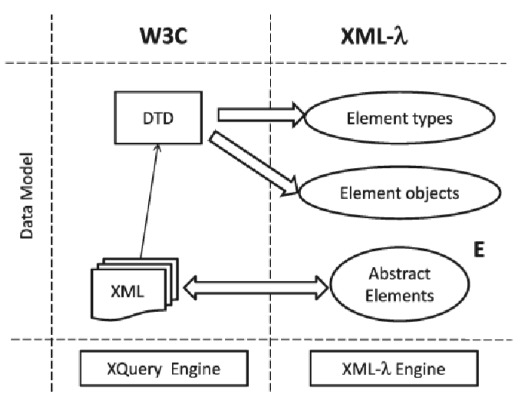

Figure 1: The Relationship Between W3C and XML-λ Data Models

Element objects[1] are basically functions of type either $E \rightarrow String$ or $E \rightarrow (E \times \ldots \times E)$. Application of these functions to an *abstract element* allows access to element's content. *Elements* are, informally, values of *element objects*, i.e. of functions. For each $t \in T_E$ there exists a corresponding t-object.

For convenience, we add a "nullary function" (also known as 0-ary function) into our model. This function returns a set of all abstract elements of a given element type from an XML document.

Finally, we can say that in XML-λ the instance of an XML document is represented by a subset of E and set of respective t-objects.

For readers familiar with W3C terminology, there is a comparison of related terms in both environments shown in Table 1.

3.2 XML-λ Example

This section shows an example of using the XML-λ Framework in a real example with detailed description. Let us consider an example DTD shown in Figure 2.

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  author+, price )>
<!ELEMENT author  (last, first )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

Figure 2: Example DTD

For given schema we obtain element types as follows:

$BIB : BOOK*$,
$BOOK : (TITLE, AUTHOR+, PRICE)$,
$AUTHOR : (LAST, FIRST)$,
$LAST : String$,
$FIRST : String$,
$TITLE : String$,
$PRICE : String$.

Then, we define functional types – designated as t-objects:

$BIB : E \rightarrow 2^E$,
$BOOK : E \rightarrow (E \times 2^E \times E)$,
$AUTHOR : E \rightarrow (E \times E)$,
$TITLE : E \rightarrow String$,
$LAST : E \rightarrow String$,
$FIRST : E \rightarrow String$,
$PRICE : E \rightarrow String$.

These types are the cornerstone for manipulation with typed data from XML documents as shown in the list of semantic functions (see Table 2).

Having look at DTD in Figure 2 and sample data in Figure 3 we can obviously see that there are 7 abstract elements (members of $E' \subset E$). Now, for instance, the $title$-object is defined exactly for one abstract element (the one gained from `<title>TCP/IP Illustrated</title>` element and for this abstract element it returns a string value "TCP/IP Illustrated".

[1] We denote the element object of type $t \in T_E$ as t-object

	W3C	XML-λ
Data Format	XML 1.0	XML 1.0
Data Model Constraints	Document Type Definition (DTD)	Types in T_E derived from DTD
XML Data Instance	DOM - A tree instance	Set of abstract elements – E, definition of t-objects
Query Languages	XPath, XQuery, XSLT	Simply typed lambda calculus

Table 1: The Relationship Between W3C and XML-λ Terms

```
<bib>
   <book>
      <title>TCP/IP Illustrated</title>
      <author>
        <last>Stevens</last>
        <first>W.</first>
      </author>
      <price>65.95</price>
    </book>
    ...
</bib>
```

Figure 3: Fragment of a Valid XML Instance

Following example query returns all books with specified price

```
lambda b (/book(b) and b/price = "65.95" )
```

Evaluation of this query with respect to semantics described in [19] takes place in following way:

1. First, the binding of free variable b is evaluated (`/book(b)`), i.e. nullary function returns a set of all abstract elements of element type $BOOK$).

2. For each item in b the application of $BOOK$-object element (note, it is a function) is performed

   $BOOK : E \rightarrow (E \times \ldots \times E)$

   and this operation returns an n-tuple.

3. Projection by name `price` returns then item(s) of type `PRICE` (there is just one). Application of function $PRICE : E \rightarrow PRICE : String$ returns a string value of the price element that is compared with literal ”65.95”. Non-matching item is skipped, otherwise the content of b is serialized to output.

4. Steps 2.-3. are repeated for all items found in Step 1.

For readers familiar with XQuery, here is the same query expressed in XQuery syntax:

```
{
  for $b in doc("bib.xml")/bib/book
  where $b/price = "65.95"
  return {$b}
 }
```

Expected output is shown in Figure 4.

```
<book>
  <title>TCP/IP Illustrated</title>
  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>
  <price>65.95</price>
</book>
```

Figure 4: Expected Query Output

4 Updating XML Documents

This section covers the process of updating data in an existing XML data store. Thus, we do not update XML schema of these documents but their content only. It is a typical database life cycle – the database schema remains stable but the data is changing in time. In our (query and update) language there is no way how to construct new documents yet – sometimes this approach is called ”incremental update”. In other words we can change the structure of the input document (w.r.t the DTD) by a given XML-λ update statement but cannot e.g. create a set of new XML files.

4.1 Updates in General

We can describe the whole operation of updating an XML document rather on a physical level as (1) retrieving its content from database, (2) performing update, (3) storing document back to database. This paper deals with the second part of the process. Viewed from closer look in more detailed pieces it is (a) localization of point in data model where the change will take place, (b) validation of requested operation, (c) execution of the update operation. These steps are shown more from the semantical point of view, in implementation it is usually not necessary to retrieve complete XML document from database into memory but we can manipulate only with a part of its content needful for update.

There are two options when to perform data validation – before or after an update. The XQuery Update Facility proposal uses optional post-update revalidation; in our approach we focus more on doing pre-update checks. Our goal is to detect the maximum number of possible conflicts during compilation of the update statement and potentially raise a static error. Unfortunately, not in all cases is the information from data model enough for validation and, therefore, it is necessary to perform validation with respect to particular data stored in the data store. We discuss this issue later in Section 5. Regardless the scenario, the processed XML document is a valid instance in the type system both before and after update.

4.2 Validation Constraints in DTD

Document Type Definition (DTD) [4] 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 element content. The possible value 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 - Each attribute can have one of `#REQUIRED`, `#IMPLIED`, `#FIXED`, `ID`, `IDREF(S)` options assigned.

Each update operation can or cannot be affected by any construct from the particular DTD. Note that element content type `ANY` cannot be used in XML-λ, because of the framework's type system nature.

4.3 Concept of Updates in XML-λ

This section covers the basic concept of updates in the XML-λ Framework. It initially had not have any update facility. We had to extend it with features allowing us to check constraints available in DTD. The idea of updates has been opened in [13] but here we focus just on the main idea.

As already outlined in Section 3.1, there are three crucial components related to the type system - *element types*, *element objects* and *abstract elements*. Element types are derived from a particular DTD and in our scenario they cannot be changed.

Elements are, informally, values of element objects, i.e. of functions. Thus, by updating an XML document in XML-λ we modify the actual domains of these functions (subsets of E) and element objects affected by required update operation (insert, delete, replace).

Before of that, we have to validate requested operation. For now let us consider constraints described by a DTD but in the outlook there are more options which standards we plan to use as well (e.g. XML Schema). Therefore we design our solution keeping this possibility in mind.

Sections 5.3 - 5.5 discuss the semantics of delete, insert and replace operations in detail.

4.4 Concurrency Support

One disadvantage of the solution proposed in [13] is the lack of transaction support [7]. In this work we assume the existence of a transaction manager that can control (i.e. lock, unlock, suspend or abort) user activities. Currently we carry out a parallel research on using the taDOM locking protocol [8] together with XML-λ (there is a recent paper that introduces our first proposal of the transactional behavior for XML-λ in [14]).

For now, we can consider that a transaction manager locks the complete part of XML data that can be modified during the update operation (in the worst case even the whole XML document). It is a significant performance issue but for purpose of this paper it is not fundamental.

Thus, at the beginning of suggested algorithms we only ask for locking of a specific part of processed XML document and keep all concurency-related worries and issues on the ”virtual” transaction manager.

5 Analysis and Design of Updates in XML-λ

In this section we describe two parts of the update process – general concept of validation we use in XML-λ and then semantics of all supported update operations – insert, delete and replace.

5.1 Validation Approach

In this paper we base our work on constraints available in DTDs. The goal here is to describe these limitations in general as much as possible for eventual future extensions. Validating update operations is a problem very closely related to the problem of validating a complete XML instance. This process, however, can be for extensive documents very time consuming.

In our approach we propose two sets of types and algorithms for validation for each update operation. Mentioned sets are constructed and initiated during analysis of given DTD and contain element types from T_E. Due to the fact that we do not allow schema changes, they are stable in time.

1. $T_{immutable}$. Abstract elements of types from this set and respective t-objects are not changeable in our data model. In terms of DTD these types are associated with DTD types which content cannot be modified, i.e. attributes declared as `#FIXED` and element types with `EMPTY` content model.

2. $T_{mandatory}$. Abstract elements of types from this set and respective t-objects are not modifiable (must not be removed) in our data model. In terms of DTD this set contains types associated with attribute types with `#REQUIRED` declaration and element types for those types T_i iff all occurrences of T_i in given DTD are exactly-one.

These sets we use in our semantics for particular update operations. For future work we can also consider sets $T_{referencing}$ and $T_{referenced}$ of types associated with attributes declared in given DTD as `IDREF` or `IDREFS` and for attributes declared as `ID` respectively. In following text we use number of functions with informal meaning as summarized in Table 2.

5.2 General Notes to Proposed Algorithms

Following sections contain particular algorithms for data modification shown in detail. The most important of them – *Delete* and *Insert* – follow the same structure. First, they check (optionally) the validity of the operation and if there is no conflict with the type system definition they break down the modification into a list of primitive operations (stored in a structure also known as the ”Pending Update List”). This list represents hence the result of these algorithms. Note that the *Replace* algorithm combines aforesaid *Delete* and *Insert* with within. The items inside the list are pairs (e, op), where $e \in E$ is an abstract element and $op \in \{DELETE, INSERT\}$ is the operation to be carried out.

The output pending update list, that represents the result of each update algorithm, is then passed to the *ProcessPendingList* algorithm. This algorithm then executes all primitive changes requested.

5.3 Delete

Formally, we decompose the process into two parts – a function *checkDelete* that is used for checking whether

Semantic Function	Behavior
parent(e)	For an $e \in E$ returns its parent abstract element. An abstract element can have at most one associated "parent" element. When considering E as infinite set of abstract elements, most of them have no parent associated.
typeOf(e)	For an $e \in E$ returns its element type (see Section 3.1).
cardMin(e),cardMax(e)	Return minimal (or maximal, respectively) cardinality of an abstract element's type in a particular data model instance.
alterTObjectDel(e, t), alterTObjectIns(e, t)	Alters the t-object for given $e \in E$. Regarding the fact that t-objects are functions these semantic functions change the domain of given t-object and thus associations among abstract elements. Basically, alterTObjectDel removes the abstract element e from domain of the t-object and alterTObjectIns adds the abstract element e into the domain.
isSubtype(t_1, t_2)	Describes a relation between element types t_1 and t_2. Returns $true$ iff the result of application(e, t_2) for an $e \in E$ can return an n-tuple containing an abstract element of type t_1 (at any position).
canSubstitute(t_1, t_2)	Returns $true$ iff an abstract element e_1 of type t_1 can replace an element e_2 of type t_2 without breaking document's validity. It is utilized in the Replace algorithm.
isElementary(t)	Returns $true$ iff t is an elementary element type.
application(e, t)	Executes an application of t-object to the e element. In general it returns an n-tuple from Cartesian product of $(E \times \ldots \times E)$. Note that the application function serves for diving in the "content" of an element.
projection(n-tuple, t)	Retrieves all elements of type t from given n-tuple.
count(n-tuple)	Returns number of elements in an n-tuple.

Table 2: List of Semantic Functions and their Informal Meaning

an abstract element can be deleted and a complete algorithm *Delete* that accomplishes the operation completely:

Function: *checkDelete*;
Input: E - set of abstract elements
    e - an abstract element to be deleted
Output: returns **true** - deletion is allowed,
    false - deletion is denied
begin
  let $t = typeOf(e)$;
  if $((t \in T_{mandatory})$ **or** $(t \in T_{immutable}))$ **then**
    return false;
  if $((cardMin(e) = 0)$ **and** $(cardMax(e) = \infty))$ **then**
    return true;
  if $((cardMin(e) \geq 1)$ **and**
    $(count(application(parent(e), t)) > 1))$ **then**
    return false;
  return true;
end

Algorithm: *Delete*;
Input: E - set of abstract elements
    e - an abstract element to be deleted
    $checkValidity$ - a boolean flag. Enables or disables validity check. Default is $true$.
    $trans$ - a new transaction
    $pList$ - a list of currently pending update operations
Output: returns **true** - delete is allowed,
    false - delete failed
    pList - updated list of pending operations
begin
  /* Lock the data being deleted */
  $trans.lockRequest(DELETE\_NODE, e)$;

  /* Check type constraints - if requested */
  if $(checkValidity)$ **then**
    if (**not** $checkDelete(E, e)$) **then return false**;

  let $S =$ **new** $Stack()$; $S.push(e)$;

  while $(tmp = S.pop())$ **do**
    let $t = typeOf(tmp)$;
    let $nt = application(tmp, t)$;
    For $i = 1$ **to** $count(nt)$
      let $e_{tmp} = nt[i]$;
      let $t_{tmp} = typeOf(e_{tmp})$;

      /* Elementary element types are added into the pending delete list */
      if $(isElementary(t_{tmp}))$ **then**
        $pList.add(e_{tmp}, DELETE))$
      else
        /* Complex element types are stored for next iterations */
        $S.push(e_{tmp})$;
    next;
  end

  /* Add the initial abstract element to pending list */
  $pList.add(tmp, DELETE)$;

  /* Deletion is finished */
  return true;
end

5.4 Insert

As for the *Delete* algorithm, we propose two parts of the insert process – function *checkInsert* that validates insertion of given abstract element and *Insert* algorithm that implements the operation in whole.

Function: *checkInsert*;
Input: E - set of abstract elements
    e_1 - an abstract element to be inserted,
    e_2 - an abstract element to be associated with e_1 as its parent abstract element,

Output: returns **true** - insertion is allowed,
false - insertion is denied
begin
let $t = typeOf(e_2)$;
if $(t \in T_{immutable})$ **then return false**;

/*Traversing through all "sibling" abstract elements*/
let $nt = application(tmp, t)$;
for $i = 1$ **to** $count(nt)$
let $e_{tmp} = nt[i]$;
let $t_{tmp} = typeOf(e_{tmp})$;
if $(isSubtype(typeOf(e_1), t_{tmp}))$ **then**
if $(cardMax(e_{tmp}) > 1)$ **then return true**;
if $((cardMin(e) = 0)$ **and**
$(cardMax(e_{tmp} = \infty))$ **and**
$(count(application(e_{tmp}, t_{tmp})) = 0))$ **then**
return true;
next;
return false;
end

Structure of the *Insert* algorithm is similar to the *Delete* algorithm. It is generally a traversal of given data model instance with modification of currently processed abstract element of elementary type.

Algorithm: *Insert*;
Input: E - set of abstract elements
e_1 - an abstract element to be inserted,
e_2 - an abstract element to be associated with e_1 as its parent abstract element,
$checkValidity$ - a boolean flag. Enables or disables validity check. Default is $true$.
$trans$ - a new transaction
$pList$ - a list of currently pending update operations
Output: returns **true** - insert is allowed,
false - insert failed
pList - updated list of pending operations
begin
/* Lock the data being inserted */
$trans.lockRequest(INSERT\_NODE, e_1)$;

if $(checkValidity)$ **then**
if not $checkInsert(E, e_1, e_2)$ **then return false**;

let $S =$ **new** $Stack()$; $S.push(e)$;

while $(tmp = S.pop())$ **do**
let $t = typeOf(tmp)$;
let $nt = application(tmp, t)$;
For $i = 1$ **to** $count(nt)$
let $e_{tmp} = nt[i]$;
let $t_{tmp} = typeOf(e_{tmp})$;

/* Elementary element types are inserted into pending list */
if $(isElementary(t_{tmp}))$ **then**
$pList.add(e_{tmp}, INSERT)$
else
/* Complex element types are stored for next iterations */
$S.push(e_{tmp})$;
next;
end
/* Add the initial abstract element to pending list */
$pList.add(e_1, INSERT)$

/* Insert is finished */
return true;
end

5.5 Replace

The replace operation can be logically separated into two parts – first, the removal of old data and then insertion of new data. To ensure that the XML instance remains valid we have to check the relation between types of deleted and inserted data. For this reason we introduce the $canSubstitute(t_{old}, t_{new})$ semantic function. This function returns $true$ if and only if we can replace an abstract element e_1 of type t_{old} with e_2 of type t_{new} (for example, for $t_1 = (a|b), t_2 = a \Rightarrow canSubstitute(t_1, t_2) = true$).

Note that we turn off the type validation for particular $Delete$ and $Insert$ calls. Type validity is already checked at the beginning of the algorithm.

Algorithm: *Replace*;
Input: E - set of abstract elements
e_1 - an abstract element to be replaced,
e_2 - an abstract element used as the substitution of e_1
$trans$ - a new transaction,
$pList$ - a list of currently pending update operations
Output: returns **true** - replace is allowed,
false - replace failed
pList - list of pending update operations
begin
/* It must be allowed to replace e_1 with e_2 */
if not $(canSubstitute(typeOf(e_1), typeOf(e_2))$ **then**
return false;

let $e_{tmp} = parent(e_1)$;
if not $Delete(E, e_1, trans, false, pList)$ **then**
return false;
if not $Insert(E, e_2, e_{tmp}, trans, false, pList)$ **then**
return false;

/* Replace is finished */
return true;
end

5.6 Pending Update List Processing

The *Delete*, *Insert* and *Replace* algorithms introduced in previous sections transform high-level manipulation operations into a sequential list of primitives that is stored in the structure called Pending Update List – here it is denoted as variable $pList$. This list is to be processed by the database engine at the end of each high-level operation in cooperation with the transaction manager.

Following algorithm describes the operation more formally.

Algorithm: *ProcessPendingList*;
Input: E - set of abstract elements
t-objects associated with affected abstract elements
$pList$ - a list of currently pending update operations
Output: $pList$ - an empty pending list,

E - (potentially modified) set of abstract elements,
t-objects - (potentially modified) t-objects

begin
  while $(pList.hasNext())$ **do**
    let $tmp = pList.next(); pList.remove()$;
    let $e = tmp.getItem()$;
    let $t = typeOf(parent(e))$;
    let $op = tmp.getOperation()$;

    if $(op == INSERT)$ **then**
      let $E = E \cup e$;
      $alterTObjectIns(e, t)$;
    else $/ * DELETE * /$
      let $E = E \setminus e$;
      $alterTObjectDel(e, t)$;
    next;
  end

  /* Pending List is now empty */
end

5.7 Query Language Impact

Considering the XML-λ Query Language as specified in [13], we have changed and extended the semantics of all update operations. The syntax of the language remains the same.

6 XML-λ's Future Exploitation

By the extensions proposed in this paper we obtain a framework suitable for both querying and updating XML data. With respect to its original idea there is a number of potential applications of the framework. Let us sketch three possible ways how to continue with its development:

1. further expand its query and update capabilities,
2. use it for integration of heterogeneous data sources,
3. use the XML-λ's formal apparatus for description of XQuery semantics.

For each option there is still a lot of work ahead. To get a complete query framework we have to finalize an issue with references within documents (`IDs` and `IDREFS`). This is only a technical problem of introducing new types and formalizing the algorithm to be executed to keep the documents consistent and valid. Let us also note another questionable area that is not covered in this paper and thus the dependencies of multiple update operations in one "query" statement. This issue deals with transactional processing and optimizing multiple update primitives' execution.

This option is also questionable because of wide acceptance of XQuery as the de-facto theoretical and industrial standard in the area of query languages for XML. At least, the research here will require extensive enthusiasm and sufficient resources.

Integration of heterogeneous data sources (as outlined in [15]) is a practical application of the solution we have presented. With respect to the universal type system construction it is possible to use various data models (not only DTD or XML Schema for XML) but for instance the relational or object data model as well.

The third option for ongoing research is using the framework for description of XQuery's semantics. This is probably the most interesting research branch from the theoretical point of view. It generally means that we will be able to express any XQuery statement with a corresponding XML-λ alternative. This idea represents a theoretical research related to formal methods and compilers. In this case the framework is going to be used as a tool for transformation of queries between various query and update languages. In addition, we can use this tool for evaluation of XQuery queries within our prototype of a native XML database system ExDB [12] based on XML-λ.

Another feasible challenge for future work is redefinition of the type system by replacement of DTD types by types available in XML Schema or in RELAX NG. This means restructuralization of the type systems T_{reg} and T_E and redevelopment of the idea of constraint sets. This research would demonstrate that the concept of functional framework is not strictly bound to DTD but is more general as we declare.

7 Conclusion

We have shown a proposal for updating typed XML data constrained by a Document Type Definition. We build on a functional framework for querying XML that can utilize concept of DTD constraints in its type system T_E. Main part of the paper discusses the idea of extending the framework with update operations with accent to keep the documents always valid. By enriching the XML-λ query language with modification operations - inserts, deletes and replacements - we obtain a language suitable both for querying and updating XML documents.

Further work and research directions outlined in Section 6 lead to onward framework extensions – either improving its query capabilities, using it for integration of heterogeneous data or utilizing the framework for description of semantics of various query languages.

In any case the work presented in this paper creates sufficient base for extensive future work.

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] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general purpose language. In *Proceedings of ICFP 2003*, Uppsala, Sweden, August 2003.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language, January 2007. http://www.w3.org/TR/xquery/.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fourth edition), August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[5] D. Chamberlin. XQuery: Where do we go from here? In *XIME-P 2006*, 2006.

[6] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Siméon. XQuery Update Facility 1.0, March 2008. http://www.w3.org/TR/2008/CR-xquery-update-10-20080314/.

[7] C. J. Date. *An Introduction to Database Systems, 6th Edition*. Addison-Wesley, 1995.

[8] M. P. Haustein and T. Härder. A synchronization concept for the DOM API. In H. Höpfner, G. Saake, and E. Schallehn, editors, *Grundlagen von Datenbanken*, pages 80–84. Fakultät für Informatik, Universität Magdeburg, 2003.

[9] H. Hosoya and B. Pierce. Xduce: A statically typed XML processing language, 2002.

[10] A. Laux and L. Martin. XUpdate – XML Update Language, 2000. available online at http://xmldb-org.sourceforge.net/xupdate/index.html.

[11] P. Lehti. Design and implementation of a data manipulation processor for an XML query language. Master's thesis, Technische Universitaet Darmstadt, 2001.

[12] P. Loupal. Experimental DataBase (ExDB) Project Homepage. http://swing.felk.cvut.cz/˜loupalp.

[13] P. Loupal. Updating typed XML documents using a functional data model. In J. Pokorný, V. Snášel, and K. Richta, editors, *DATESO*, volume 235 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[14] P. Loupal. Using taDOM Locking Protocol in a Functional XML Update Language. In *DATESO*, 2008.

[15] J. Pokorný. XML functionally. In B. C. Desai, Y. Kioki, and M. Toyama, editors, *Proceedings of IDEAS2000*, pages 266–274. IEEE Comp. Society, 2000.

[16] J. Pokorný. XML-λ: an extendible framework for manipulating XML data. In *Proceedings of BIS 2002*, pages 160–168, Poznan, 2002.

[17] G. M. Sur, J. Hammer, and J. Siméon. An XQuery-Based Language for Processing Updates in XML. In *PLAN-X 2004*, 2004.

[18] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD 2001*, 2001.

[19] P. Šárek. Implementation of the XML lambda language. Master's thesis, Dept. of Software Engineering, Charles University, Prague, 2002.