PROLE 2006

XQuery within Logic Programming [1]

J. M. Almendros-Jiménez[2], A. Becerra-Terón[3], F. J. Enciso-Baños[4]

Dpto. Lenguajes y Computación.
Universidad de Almería, Spain

Abstract

XQuery is a typed functional language for Web Databases. It is a query language against XML documents. In this paper we study how to translate XQuery programs into logic programming. With this aim, each XQuery functional expression is translated into logic rules and goals. The answers for the goals correspond with answers for the query against XML documents.

Key words: Database Query Languages, Logic Programming, XQuery Language.

1 Introduction

XQuery [12,5] is a typed functional language devoted to express queries against of XML documents. It contains XPath 2.0 as a sub-language which supports navigation, selection and extraction of fragments of XML documents. XQuery also includes expressions to construct new XML values, and to integrate or join values from multiple documents. In this framework many attempts for implementing XQuery have been accomplished. For instance, XQuery language has been implemented using a functional language as host language. This is the case of XDuce [7] and CDuce [3], which use regular expression pattern matching over XML trees and sub-typing as basic mechanism. Moreover, as host language, both use OCAML. The language XGalax [8,5] is also a relevant contribution to the implementation of XQuery in functional programming, using also OCAML as host language. There are also several proposals for introducing the definition and handling of XML documents rather than to

[1] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02.

[2] Email: `jalmen@ual.es`

[3] Email: `abecerra@ual.es`

[4] Email: `fjenciso@ual.es`

implement XQuery. This is the case of the proposal around Haskell, such as HaXML [11].

In addition, there are attempts to use logic programming for handling XML documents. For instance, the Xcerpt project [10] proposes a pattern and rule-based query language for XML documents using the so-called query terms including logic variables for the retrieval of XML elements. The same can be said for XPathLog (LOPIX system) [9] which is a Datalog-style extension of XPath with variable bindings. Let us also remark the case of XCentric [6], which can represent XML documents into logic programming and handle XML documents by considering terms with functions of flexible arity and regular types. Finally, some Prolog implementations include libraries for loading and querying XML documents, such as SWI-Prolog [13] and CIAO [4].

In this paper, we are interested in the use of logic programming for the handling of XQuery queries:

(i) A XML document can be seen as a logic program, by considering *facts* and *rules* for expressing both the XML schema and document. This was already studied in our previous work [1].

(ii) A *XQuery expression can be translated into logic programming* by considering a set of rules and goals. Some of the rules specialize the schema rules of each XML document involved in the XQuery expression and others are introduced in order to achieve the join operations between two or more documents. In addition, for each XQuery expression, a specific goal (or goals) is called, where *appropriate arguments can be instantiated.*

In our proposal XQuery is implemented into logic programming, by means of a translation of XQuery expressions into logic rules. As far as we know this is the first time that XQuery is implemented in logic programming. In this implementation, we use as basis the implementation of XPath studied in our previous works [1,2]. The advantages of such proposal is that XQuery is embedded into logic programming, and therefore XQuery can be combined with logic programs. For instance, logic programming can be used as inference engine, one of the requirements of the so-called Semantic Web (`http://www.w3.org/2001/sw/`). It opens a promising line of research of our work.

The structure of the paper is as follows. Section 2 will show the needed preliminaries for this paper; section 3 will study the translation of XML documents into Prolog; section 4 will discuss the translation of XQuery into logic programming; and finally, section 5 will conclude and present future work.

2 Preliminaries

An *XML document* basically is a *labelled tree* with nodes representing *composed or non-terminal items* and leaves representing *values or terminal items.* For instance, let us consider the following XML document:

```
<books>
<book year="2003">
<author>Abiteboul</author>
```

```
<author>Buneman</author>
<author>Suciu</author>
<title>Data on the Web</title>
<review>A <em>fine</em> book.</review>
</book>
<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
<review><em>The <em>best</em> ever!</em></review>
</book>
</books>
```

representing a set of books, where each record stores the authors, titles and reviews for each book; and each record has an attribute representing the publishing year. Assuming that the previous document is stored in a file called "books.xml", to list titles of all books published before 2003, you might write: `document("books.xml")/books/book[@year < 2003]/title`. Here, a XPath expression is used for describing the path in the XML tree to be retrieved. In each tree node a boolean condition can be required. In the example, the attribute `year` should be smaller than 2003. In this case, the result would be `<title>XML in Scotland</title>`.

In order to introduce XQuery, let us start with the main feature; that is the `for-let-where-return` constructions (*"FLoWeR"* expressions). XQuery uses these expressions in order to build more complex queries. As second feature, we can remark the use of XPath as sub-language in XQuery in order to traverse over the structure of a XML document. For instance, to list the year and title of all books published before 2003, you might write in XQuery:

```
for $book in document("books.html")/books/book
where $book/@year<2003
return <mybook> {$book/@year, $book/title} </mybook>
```

Here, in a declarative way, for each `book` of the XML file published before 2003, a new record labelled with `mybook` will store the year and titles. In addition, the local definition $`book` involved in the `for` expression is used for traversing the sequence of books in the XML file; next, the condition included in the `where` expression allows us to filter those books with publishing year before than 2003, and finally, the `return` expression builds a new document as a sequence of labelled elements in which each element contains the `year` and `title` of the retrieved books enclosed in the label `mybook`. In this case the result would be:

```
<mybook year="2002">
<title>XML in Scotland</title>
</mybook>
```

Moreover, `let` expressions can be used for declaring local variables as in functional languages. The difference between `let` and `for` expressions is that `let` is used for local binding to one node of a XML document, whereas `for` allows us to local bind to each element of a sequence of nodes. Whenever the sequence is unitary, `let` and `for` have the same semantics. For instance, in order to list all books that are more expensive at Amazon than at Barnes and Noble, we might write in XQuery the following expression:

```
let $am:=document("http://www.amazon.com/books.xml")/books,
let $bn:=document("http://www.bn.com/books.xml")/books
for $a in $am/book
for $b in $bm/book
where $a/isbn= $b/isbn and $a/price > $b/price
return <book>{ $a/title, $a/price, $b/price}</book>
```

`For-let-where-return` expressions can be nested building complex queries involving two or more XML documents.

3 Translating XML Documents into Logic Programming

In this section, we will show how to translate an XML document into a logic program. We will use a set of rules for describing the XML schema and a set of facts for storing the XML document. In general, an XML document includes: (a) *tagged elements* which have the form: $<\mathtt{tag}\ \mathtt{att_1} = \mathtt{v_1}, \ldots, \mathtt{att_n} = \mathtt{v_n} > \mathtt{subelem_1}, \ldots, \mathtt{sublem_k} < /\mathtt{tag} >$ where $\mathtt{att_1}, \ldots, \mathtt{att_n}$ are the attributes names, $\mathtt{v_1}, \ldots, \mathtt{v_n}$ are the attribute values supposed to be of a `basic type` (*string,integer,list of integers, etc*), and $\mathtt{subelem_1}, \ldots, \mathtt{subelem_k}$ are subelements; in addition, it can include (b) *untagged elements* which are of a `basic type`.

Terminal tagged elements are those ones whose subelements are of basic types. Otherwise, they are called *non-terminal tagged elements.* Two tagged elements are *similar* whether they have the same structure; that is, they have the same tag and attributes names, and the subelements are similar. Untagged elements are always similar. Two tagged elements are *distinct* if they do not have the same tag and, finally, *weakly distinct* if they have the same tag but are not similar.

3.1 Numbering XML documents

In order to define our translation we need to number the nodes of the XML document. Therefore, given an XML document we can consider a new XML document called *node-numbered XML document* as follows. Starting from the root element numbered as 1, the node-numbered XML document is numbered using an attribute called **nodenumber**[5] where each `j`-th child of a tagged element is numbered with the sequence of natural numbers $\mathtt{i_1.\ldots.i_t.j}$ whenever the ancestor is numbered as $\mathtt{i_1.\ldots.i_t}$: $<\mathtt{tag}\ \mathtt{att_1} = \mathtt{v_1}, \ldots, \mathtt{att_n} = \mathtt{v_n}$, $\mathbf{nodenumber} = \mathbf{i_1.\ldots.i_t.j} > \mathtt{elem_1}, \ldots, \mathtt{elem_s} < /\mathtt{tag} >$. This is the case of tagged elements. If the j-th child is an string and the ancestor is a non-terminal tagged element then the element is labelled and numbered as follows: $<\mathtt{unlabeled}\ \mathbf{nodenumber} = \mathbf{i_1.\ldots.i_t.j} > \mathtt{elem} < /\mathtt{unlabeled} >$; otherwise

[5] It is supposed that "nodenumber" is not already used as attribute in the tags of the original XML document.

the element is not numbered. It gives to us a *left-to-right numbering* of the nodes of an XML document. An element in an XML document is leftmost in the XML tree than other whether the node number is smaller w.r.t. the lexicographic order of sequence of natural numbers.

In addition, we have to consider a new document called *type and node-numbered XML document* numbered using an attribute called **typenumber** as follows. Starting the numbering from 1 in the root of the node-numbered XML document, each tagged element is numbered as: $<\mathtt{tag}\ \mathtt{att}_1 = \mathtt{v}_1, \ldots, \mathtt{att}_n = \mathtt{v}_n, \mathtt{nodenumber} = \mathtt{i}_1.\ldots, \mathtt{i}_t.\mathtt{j}, \mathbf{typenumber} = \mathbf{k} > \mathtt{elem}_1, \ldots, \mathtt{elem}_s < /\mathtt{tag} >$ and $< \mathtt{unlabeled}\ \mathtt{nodenumber} = \mathtt{i}_1.\ldots.\mathtt{i}_t.\mathtt{j}, \mathbf{typenumber} = \mathbf{k} > \mathtt{elem}_j < /\mathtt{unlabeled} >$ for ”unlabelled” nodes. In both cases, the type number of the tag is $k = l + n + 1$ whenever the type number of the ancestor is l, and n is the number of tagged elements weakly distinct to the ancestor, occurring in leftmost positions at the same level of the XML tree. Therefore, all the children of a tag have the same type number. For instance, with respect to the running example, next we show the type and node numbering of the XML document.

Example of type and node numbered XML Document

```
<books nodenumber=1, typenumber=1>
<book year=”2003”, nodenumber=1.1, typenumber=2>
<author nodenumber=1.1.1 typenumber=3>Abiteboul</author>
<author nodenumber=1.1.2 typenumber=3>Buneman</author>
<author nodenumber=1.1.3 typenumber=3>Suciu</author>
<title nodenumber=1.1.4 typenumber=3>Data on the Web</title>
<review nodenumber=1.1.5 typenumber=3>
<unlabeled nodenumber=1.1.5.1 typenumber=4> A </unlabeled>
<em nodenumber=1.1.5.2 typenumber=4>fine</em>
<unlabeled nodenumber=1.1.5.3 typenumber=4> book.  </unlabeled>
</review>
</book>
<book year=”2002” nodenumber=1.2, typenumber=2>
<author nodenumber=1.2.1 typenumber=3>Buneman</author>
<title nodenumber=1.2.2 typenumber=3>XML in Scotland</title>
<review nodenumber=1.2.3 typenumber=3>
<em nodenumber=1.2.3.1 typenumber=5>
<unlabeled nodenumber=1.2.3.1.1, typenumber=6> The </unlabeled>
<em nodenumber=1.2.3.1.2, typenumber=6>best</em>
<unlabeled nodenumber=1.2.3.1.3,typenumber=6> ever!  </unlabeled>
</em></review></book></books>
```

Let us remark that in practice the type and node numbering of XML documents can be simultaneously generated during the translation of the XML document into a logic program. In fact, the type and node numbered version of the original XML document is not generated as an XML file. Here, we have shown the type and node numbered version of the XML document only to define the translation into logic programming.

3.2 Translation of XML documents

Now, the translation of the XML document into a logic program is as follows. For each non-terminal tagged element in the type and node numbered XML document:

$$< \mathtt{tag}\ \mathtt{att_1} = \mathtt{v_1}, \ldots, \mathtt{att_n} = \mathtt{v_n}, \mathtt{nodenumber} = \mathtt{i}, \mathtt{typenumber} = \mathtt{k} >$$
$$\mathtt{elem_1}, \ldots, \mathtt{elem_s} < /\mathtt{tag} >$$

we consider the following rule, called *schema rule*:

$$\begin{aligned}
&\mathtt{tag(tagtype(X_{i_1}, \ldots, X_{i_t}, [A_1, \ldots, A_n]), L, k)\text{:-}}\\
&\qquad\mathtt{tag_{i_1}(X_{i_1}, [B_{i_1}|L], r),}\\
&\qquad\ldots,\\
&\qquad\mathtt{tag_{i_t}(X_{i_t}, [B_{i_t}|L], r),}\\
&\qquad\mathtt{att_1(A_1, L, r),}\\
&\qquad\ldots,\\
&\qquad\mathtt{att_n(A_n, L, r).}
\end{aligned}$$

where *tagtype* is a new function symbol used for building a Prolog term containing the XML document; $tag_{i_1}, \ldots, tag_{i_t}$, $i_j \in \{1, \ldots, s\}$, $1 \leq j \leq t$, are the *set of tags* of the tagged elements $elem_1, \ldots, elem_s$ of the type and node-numbered XML document (including unlabelled elements); $X_{i_1}, \ldots, X_{i_t}$ are variables; $att_1, \ldots, att_n$ are the attribute names; $A_1, \ldots, A_n$ are variables, one for each attribute name; $B_{i_1}, \ldots, B_{i_t}$ are variables (used for representing the node number of the children); L is a variable (used for representing the node number of the tag). k is the type number of *tag*. r is the type number of the tagged elements $elem_1, \ldots, elem_s$ [6]. In addition, we consider facts of the form: $\mathtt{att_j(v_j, i, k)}$ ($1 \leq j \leq n$). Finally, for each terminal tagged element in the type and node numbered XML document: $< \mathtt{tag}\ \mathtt{nodenumber} = \mathtt{i}$, $\mathtt{typenumber} = \mathtt{k} > \mathtt{value} < /\mathtt{tag} >$, we consider the *fact*: $\mathtt{tag(value, i, k)}$. For instance, the running example can be represented by means of a logic program as follows:

Prolog program of an XML document

```
Rules (Schema):
---------------------------------------

books(bookstype(A, []), L,1) :-
      book(A, [B|L],2).
book(booktype(A, B, C, [D]), L ,2) :-
      author(A, [E|L],3),
      title(B, [F|L],3),
      review(C, [G|L],3),
      year(D, L,3).

    review(reviewtype(A,B,[]),L,3):-
          unlabeled(A,[J|L],4),
          em(B,[K|L],4).
    review(reviewtype(A,[]),L,3):-
          em(A,[J|L],5).
    em(emtype(A,B,[]),L,5) :-
          unlabeled(A,[G|L],6),
          em(B, [H|L],6).
```

```
Facts (Document):
-----------------------------------------

year('2003', [1, 1], 3).
author('Abiteboul', [1, 1, 1], 3).
author('Buneman', [2,1, 1], 3).
author('Suciu', [3,1,1], 3).
title('Data on the Web', [4, 1, 1], 3).
unlabeled('A', [1, 5, 1, 1], 4).
em('fine', [2, 5, 1, 1], 4).

unlabeled('book.', [3, 5, 1, 1], 4).
year('2002', [2, 1], 3).
author('Buneman', [1, 2, 1], 3).
title('XML in Scotland', [2, 2, 1], 3).
unlabeled('The', [1, 1, 3, 2, 1], 6).
em('best', [2, 1, 3, 2, 1], 6).
unlabeled('ever!', [3, 1, 3, 2, 1], 6).
```

[6] Let us remark that since *tag* is a tagged element, then $elem_1, \ldots, elem_s$ have been tagged with "unlabelled" labels in the type and node numbered XML document; thus they have a type number.

4 Translating XQuery into Logic Programming

In this section, we will present the technique for translating XQuery expressions against an XML document into a logic program. With this aim, we will show the general criteria for the transformation technique.

(i) The handling of an XQuery query involves the *specialization of the schema rules* of the XML documents involved in the XQuery expression, together with the *inclusion of new rules*, and the *generation of one or more goals.*

(ii) Each XPath expression included in the XQuery expression is associated to an XML document. Thus, the translation of *each XPath expression* involves the *specialization of the schema rules* of the associated XML document.

(iii) Each `return` *expression* included in the XQuery expression is translated into *a new schema rule*, whose role is to describe the structure of the new XML document build from the `return` expression.

(iv) The *outermost* `return` *expression* in the XQuery expression is translated into *one or more goals.* The answers of such goals allow the reconstruction of the XML document representing the answer of the XQuery expression.

(v) For *each local definition* of `for` and `let` expressions, *a specialized rule* is introduced corresponding to the XPath expression involved in the local definition.

(vi) Each `where` *expression* included in the XQuery expression *specializes the rules* obtained from XPath and `for`, `let` and `return` expressions.

Now, we can consider the following examples:

- Let us suppose a query requesting the year and title of the books published before 2003. For this query the translation is as follows:

```
for $book in document (books.xml)/books/book
let $year = $book/@year
where $year<2003
return <mybook>{$year, $book/title}</mybook>
```

```
mybook(mybooktype(A,[B]),[C],1):-
                query(B,[C],1),
                year2(B,[C],2),
                title2(A,[[D|C]],2).
year2(A,[C],2):-year(A,C,3).
title2(B,[C],2):-title(B,C,3).
query(D, [E], 1) :-
          book(booktype(A,B,C,[D]), E, 2).
book(booktype(A,B,C,[D]), E, 2) :-
                year(D, E, 3),
                le(D, 2003).
```

where `mybook` is a new schema rule introduced by the `return` expression. In the `return` expression, a new XML document is defined containing records labelled with `mybook` and including the `year` as attribute and `title` as value. In order to distinguish them from the original `year` and `title` predicates, they are numbered as `year2` and `title2`. This kind of schema rules have a little difference w.r.t. the schema rules representing the original document. They have introduced a predicate called `query` whose role is to retrieve the query w.r.t. the source document. In other words, the `query` predicate is used for computing:

```
for $book in document (books.xml)/books/book
let $year = $book/@year
where $year<2003
```

That is, it allows us to retrieve the books published before than 2003. Once computed, the new schema rule defines the new structure, as follows:

```
mybook(mybooktype(A,[B]),[C],1):-
            query(B,[C],1),
            year2(B,[C],2),
            title2(A,[[D|C]],2).
```

This rule is defined from the return expression: `"return <mybook>{$year, $book/title}</mybook>"`. In this new structure, `year` and `title` are taken from the original document. This fact is expressed by means of the following rules:

```
year2(A,[C],2):-year(A,C,3).
title2(B,[C],2):-title(B,C,3).
```

Let us remark an important aspect in these rules. The use of node numbering in the variable `C` is the key for retrieving only those years before than 2003 and, of course, the titles of the books for such years. `C` has been previously instantiated by means of the rule `query`, retrieving those node numbers which satisfy the proposed condition.

There is an additional difference of this kind of schema rules w.r.t. the schema rules of the original document. The node numbering consists on lists of sequences of natural numbers. This is due to XQuery expressions can involve multiple XML documents. Therefore, the output document is numbered in each node as $seq_1, \ldots, seq_n$, whenever the XQuery expression involves n documents wherein each document is numbered with sequences of natural numbers seq_i. In the previous example, we have only considered one document as input and it is numbered with a unitary list (of sequences of natural numbers). Finally, the type numbering is new and defined according to the structure defined in the `return` expression. With respect to the `query` predicate:

```
query(D, [E], 1) :- book(booktype(A,B,C,[D]), E, 2).
book(booktype(A,B,C,[D]), E, 2) :-
            year(D, E, 3),
            le(D, 2003).
```

remark that the rule head has a variable `D` taken from the `book` call in the rule body. It represents the `year` of each `book`. The `where` expression forces to include `year(D,E,3),le(D,2003)` in the `book` body rule, which requires that the year of the books is before than 2003. Such rule *specializes* the schema rule of the original document w.r.t. the XQuery expression. In this specialized version, `author` and `title` predicates have been removed and `le` predicate (less than) has been added. Finally, the `return` expression generates the goal `:-mybook(A,B,1).` The top-down evaluation of such query computes the answer: `{A/mybooktype('XML in Scotland',[2002]), B/[[2,1]]}` which represents the following instance of the goal: `mybook(mybooktype('XML in Scotland', [2002]), [[2, 1]], 1),` and represents, according to the new schema rule, the following output document:

```
<mybook year="2002">
<title>XML in Scotland</title>
</mybook>
```

- Now, let us suppose the following XQuery expression, requesting the reviews of books published before than 2003 occurring in two documents: the first one is the running example, and the second one is as follows:

Second XML Document

```
<books>
<book year="2003">
<author>Abiteboul</author>
<author>Buneman</author>
<author>Suciu</author>
<title>Data on the Web</title>
<review>very good</review>
</book>
<book year="2002">
<author>Buneman</author>
<title>XML in Scotland</title>
<review>It is not bad</review>
</book> </books>
```

XQuery Expression

```
Let $store1 in document (books1.xml)/books,
$store2 in document(books2.xml)/books
for $book1 in $store1,
for $book2 in $store2
let $title= $book1/title
where $book1/@year<2003 and $title=$book2/title
return <mybook>{$title, $book1/review,
                   $book2/review}</mybook>
```

Here, we consider the following rules:

```
mybook(mybooktype(B,R,[]),[E,J],1):-
              query(B,[E,J],1),
              title3(B,[[G|E],[H|J]],2),
              review3(R,[[I|E],[K|J]],2).
title3(B,[E,J],2):-title1(B,E,3).
review3(R,[E,J],2):-review1(R,E,3).
review3(R,[E,J],2):-review2(R,J,3).
```

```
query(B, [E,J], 1) :-
       book1(book1type(A, B, C, [D]), E, 2),
       book2(book2type(F, B, G, [I]), J, 2).
book1(book1type(A, B, C, [D]), E, 2) :-
       year1(D, E, 3),
       le(D, 2003),
       title1(B, [F|E], 3).
book2(book2type(A, B, C, [D]), E, 2) :-
       title2(B, [G|D], 3).
```

In this case, the `return` expression generates a new schema rule `mybook` in which `title` is generated from the first document and `review` from both documents. In addition, the `query` predicate computes the join of both documents w.r.t. the title demanding those books with publishing year previous than 2003. With this aim, the `book1` rule –input schema rules are numbered as 1 and 2 and output as 3– is specialized in order to retrieve books previous than 2003 and titles. That is, in the original rule, `author` and `review` predicates have been removed. In addition, the `book2` rule has been specialized in order to retrieve the book titles. The specialization consists on the removing of `author`, `review` and `year` in the original rule. Finally, the `query` predicate joins both documents by sharing the variable `B` in the call of `book1` and `book2`. In this case, the `return` expression generates the goal: `:-mybook(A, B, 1)`.

5 Conclusions and Future Work

In this paper, we have studied how to translate XQuery expressions into logic programming. It allow us to evaluate XQuery expressions against XML documents using logic rules. Our work is still in development. We believe that the core of the XQuery language can be expressed in logic programming using the presented technique. As future work, firstly, we would like to formally define our translation; and secondly, we would like to implement our technique.

References

[1] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Magic sets for the XPath language. In *Procs. PROLE'05*, 2005.

[2] J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Querying xml documents in logic programming. Technical report, Universidad de Almería, available at `http://www.ual.es/~jalmen`, 2006.

[3] Veronique Benzaken, Giuseppe Castagna, and Alain Frish. CDuce: an XML-centric general-purpose language. In *Procs of ICFP'05*, pages 51–63. ACM Press, 2005.

[4] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *TPLP*, 1(3):251–282, 2001.

[5] D. Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, and Philip Wadler. *XQuery from the Experts*. Addison Wesley, 2004.

[6] Jorge Coelho and Mario Florido. Type-based xml processing in logic programming. In *Procs of the PADL'03*, pages 273–285. LNCS 2562, 2003.

[7] H. Hosoya and B. C. Pierce. XDuce: A Statically Typed XML Processing Language. *TOIT*, 3(2):117–148, 2003.

[8] A. Marian and J. Simeon. Projecting XML Documents. In *Procs. of VLDB'03*, pages 213–224. Morgan Kaufmann, 2003.

[9] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *TPLP*, 4(3):239–287, 2004.

[10] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Procs. of RuleML'02*, 2002.

[11] Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *JFP*, 12(4&5):435–468, 2002.

[12] W3C. Xml query working group and xsl working group, XQuery 1.0: An XML Query Language. Technical report, W3C Consortium, 2004.

[13] J. Wielemaker. SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam, March 2005.