Integrating XQuery and Logic Programming*

Jesús M. Almendros-Jiménez, Antonio Becerra-Terón, and Francisco J. Enciso-Baños

Dpto. Lenguajes y Computación
Universidad de Almería
{jalmen,abecerra,fjenciso}@ual.es

Abstract. In this paper we investigate how to integrate the XQuery language and logic programming. With this aim, we represent XML documents by means of a logic program. This logic program represents the document schema by means of rules and the document itself by means of facts. Now, XQuery expressions can be integrated into logic programming by considering a translation (i.e. encoding) of *for-let-where-return* expressions by means of logic rules and a goal.

1 Introduction

The *eXtensible Markup Language (XML)* is a simple, very flexible text format derived from SGML. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. In this context, *XQuery* [W3C07b, CDF+04, Wad02, Cha02] is a typed functional language devoted to express queries against XML documents. It contains *XPath* [W3C07a] as a sublanguage which supports navigation, selection and extraction of fragments from XML documents. *XQuery* also includes expressions (i.e. *for-let-where-return* expressions) to construct new XML values and to join multiple documents. The design of *XQuery* has been influenced by group members with expertise in the design and implementation of other high-level languages. *XQuery* has static typed semantics and a formal semantics which is part of the *W3C* standard [CDF+04, W3C07b].

The integration of *logic programming languages* and *web technologies*, in particular, XML data processing is interesting from the point of view of the applicability of logic programming. On one hand, XML documents are the standard format of exchanging information between applications. Therefore, logic languages should be able to handle and query such documents. On the other hand, logic languages could be used for extracting and inferring semantic information from XML, RDF (*Resource Description Framework*) and OWL (*Ontology Web Language*) documents, in the line of *"Semantic Web"* requirements [BHL01]. Therefore, logic languages can find a natural and interesting application field in this area. The integration of *declarative programming* and *XML data processing*

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

D. Seipel, M. Hanus, and A. Wolf (Eds.): INAP/WLP 2007, LNAI 5437, pp. 117–135, 2009.

is a research field of increasing interest in the last years (see [BBFS05] for a survey). There are proposals of new languages for XML data processing based on functional, and logic programming.

The most relevant contribution is the *Galax* project [MS03, CDF$^+$04], which is an implementation of *XQuery* in functional programming, using *OCAML* as host language. There are also proposals for new languages based on functional programming rather than implementing *XPath* and *XQuery*. This is the case of *XDuce* [HP03] and *CDuce* [BCF05, BCM05], which are languages for XML data processing, using regular expression pattern matching over XML trees, subtyping as basic mechanism, and *OCAML* as host language. The *CDuce* language does fully statically-typed transformation of XML documents, thus guaranteeing correctness. In addition, there are proposals around *Haskell* for the handling of XML documents, such as *HaXML* [Thi02, ACJ04] and [WR99].

In the field of logic programming there are also contributions for the handling of XML documents. For instance, the *Xcerpt* project [SB02, BS02a] 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. For this new language, a specialized unification algorithm for query terms has been studied in [BS02b]. Another contribution of a new language is *XPathLog* (integrated in the the *Lopix* system) [May04] which is a *Datalog*-style extension for *XPath* with variable bindings. *Elog* [BFG01] is also a logic-based XML data manipulation language, which has been used for representing Web documents by means of logic programming. This is also the case of *XCentric* [CF07, CF03, CF04], which can represent XML documents by means of logic programming, and handles XML documents by considering terms with functions of flexible arity and regular types. *FNPath* [Sei02] is also a proposal for using *Prolog* as a query language for XML documents. It maps XML documents to a Prolog Document Object Model (DOM), which can consist of facts (graph notation) or a term structure (field notation). *FnPath* can evaluate XPath expressions based on that DOM. The *Rule Markup Language* (*RuleML*) [Bol01, Bol00b, Bol00a] is a different kind of proposal in this research area. The aim of *RuleML* is the representation of *Prolog* facts and rules in XML documents, and thus, the introduction of *rule systems* into the *Web*. Finally, some well-known *Prolog* implementations include libraries for loading and querying XML documents, such as *SWI-Prolog* [Wie05] and *CIAO* [CH01].

In this paper, we investigate how to integrate the *XQuery* language and logic programming. With this aim:

1. Following our previous proposal [ABE08, ABE06], an XML document can be seen as a logic program (a Prolog program), by considering *facts* and *rules* for expressing both the XML schema and document.
2. Now, our proposal is that an *XQuery* expression can be translated (i.e. encoded) into logic programming (i.e. into a Prolog program) by introducing *new rules* for the *join* of documents, and for the translation of *for-let-where-return* expressions. Such rules are combined with the rules and facts representing the input XML documents.

3. Finally, a *specific goal* is generated for obtaining the answer of the given *XQuery* expression. From the set of answers of the generated goal, we can rebuild an XML document representing the answer of the given XQuery expression.

In summary, our technique allows the handling of XML documents as follows. Firstly, the input XML documents are loaded. It involves the translation of the XML documents into a logic program. For efficiency reasons, the rules, which correspond to the XML document structure, are loaded in *main memory*, but facts, which represent the values of the XML document, are stored in *secondary memory*, whenever they do not fit in main memory and using appropriate *indexing techniques* [ABE06, ABE08]. Secondly, the user can now write queries against the loaded documents. Each given *XQuery* query is translated into a logic program and a specific goal. The evaluation of such goal takes advantage of the indexing technique to improve the efficiency of query solving. Finally, from the set of answers of the goal, an output XML document can be built. Let us remark that our proposal uses as basis the implementation of *XPath* in logic programming studied in our previous work [ABE08] (for which a bottom-up approach has been also studied in [ABE06]).

The structure of the paper is as follows. Section 2 will present the translation of XML documents into Prolog; Section 3 will review the translation of *XPath* into logic programming; Section 4 will provide the new translation of *XQuery* expressions into logic programming; and finally, Section 5 will conclude and present future work.

2 Translating XML Documents into Logic Programming

In order to define our translation, we need to number the nodes of the XML documents. Similar kinds of node numbering have been studied in some works about XML processing in relational databases [BGvK$^+$05, OOP$^+$04, TVB$^+$02]. Our goal is similar to these approaches: to identify each inner node and leaf of the tree represented by the XML document.

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**[1] where each j-th child of a tagged element is numbered with the sequence of natural numbers $i_1.\ldots.i_t.j$ whenever the parent is numbered as $i_1.\ldots.i_t$: $< tag\ att_1 = v_1, \ldots, att_n = v_n,$ **nodenumber**= $\mathbf{i_1.\ldots.\ i_t.j} >$ $elem_1, \ldots, elem_s < /tag >$. This is the case of tagged elements. If the j-th child is of a basic type (non tagged) and the parent is an inner node, then the element is labeled and numbered as follows: $<$ *unlabeled* **nodenumber** $= \mathbf{i_1.\ldots.i_t.j} >$ $elem < /unlabeled >$; otherwise the element is not numbered. It gives to us a *hierarchical and left-to-right numbering* of the nodes of an XML document.

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

An element in an XML document is further left in the XML tree than another when the node number is smaller w.r.t. the lexicographic order of sequences of natural numbers. Any numbering that identifies each inner node and leaf could be adapted to our translation.

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: $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i_1.\ldots, i_t.j, \mathbf{typenumber} = \mathbf{k} > elem_1, \ldots, elem_s < /tag >$. The type number k of the tag is equal to $l + n + 1$ whenever the type number of the parent is l, and n is the number of tagged elements weakly distinct [2] occurring in leftmost positions at the same level of the XML tree [3].

Now, the translation of the XML document into a logic program is as follows. For each inner node in the type and node numbered XML document $< tag\ att_1 = v_1, \ldots, att_n = v_n, nodenumber = i, typenumber = k > elem_1, \ldots, elem_s < /tag >$ we consider the following rule, called *schema rule*:

$$
\begin{array}{l}
tag(tagtype(Tag_{i_1}, \ldots, Tag_{i_t}, [Att_1, \ldots, Att_n]), NTag, k, Doc) \text{:-} \\
\qquad tag_{i_1}(Tag_{i_1}, [NTag_{i_1}|NTag], r, Doc), \\
\qquad \ldots, \\
\qquad tag_{i_t}(Tag_{i_t}, [NTag_{i_t}|NTag], r, Doc), \\
\qquad att_1(Att_1, NTag, r, Doc), \\
\qquad \ldots, \\
\qquad att_n(Att_n, NTag, r, Doc).
\end{array}
$$

where *tagtype* is a new function symbol used for building a Prolog term containing the XML document; $\{tag_{i_1}, \ldots, tag_{i_t}\}$, $i_j \in \{1, \ldots, s\}$, $1 \leq j \leq t$, is the *set of tags* of the tagged elements $elem_1, \ldots, elem_s$; $Tag_{i_1}, \ldots, Tag_{i_t}$ are variables; $att_1, \ldots, att_n$ are the attribute names; $Att_1, \ldots, Att_n$ are variables, one for each attribute name; $NTag_{i_1}, \ldots, NTag_{i_t}$ are variables (used for representing the last number of the node number of the children); $NTag$ is a variable (used for representing the node number of *tag*); k is the type number of *tag*; and finally, r is the type number of the tagged elements $elem_1, \ldots, elem_s$ [4].

In addition, we consider facts of the form: $att_j(v_j, i, k, doc)$ $(1 \leq j \leq n)$, where *doc* is the name of the document. Finally, for each leaf in the type and node numbered XML document: $< tag\ nodenumber = i, typenumber = k > value < /tag >$, we consider the *fact*: $tag(value, i, k, doc)$, where *doc* is the name of the document. For instance, let us consider the following XML document called "books.xml":

[2] Two elements are weakly distinct whenever they have the same tag but not the same structure.

[3] In other words, type numbering is done by levels and in left-to-right order, but each occurrence of weakly distinct elements increases the numbering in one unit.

[4] Let us remark that since *tag* is a tagged element, then $elem_1, \ldots, elem_s$ have been tagged with "unlabeled" labels in the type and node numbered XML document when they were not labeled; thus they must have a type number.

```
<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>
```

Now, the previous XML document can be represented by means of a logic program as follows:

```
Rules (Schema):
———————————————
books(bookstype(Book, []), NBooks,1,Doc) :-
        book(Book, [NBook|NBooks],2,Doc).
book(booktype(Author, Title, Review, [Year]),
         NBook ,2,Doc) :-
        author(Author, [NAu|NBook],3,Doc),
        title(Title, [NTitle|NBook],3,Doc),
        review(Review, [NRe|NBook],3,Doc),
        year(Year, NBook,3,Doc).
review(reviewtype(Un,Em,[]),NReview,3,Doc):-
    unlabeled(Un,[NUn|NReview],4,Doc),
    em(Em,[NEm|NReview],4,Doc).
review(reviewtype(Em,[]),NReview,3,Doc):-
    em(Em,[NEm|NReview],5,Doc).
em(emtype(Unlabeled,Em,[]),NEms,5,Doc) :-
    unlabeled(Unlabeled,[NUn|NEms],6,Doc),
    em(Em, [NEm|NEms],6,Doc).
```

```
Facts (Document):
—————————————————
year('2003', [1, 1], 3, "books.xml").
author('Abiteboul', [1, 1, 1], 3, "books.xml").
author('Buneman', [2,1, 1], 3, "books.xml").
author('Suciu', [3,1,1], 3, "books.xml").
title('Data on the Web', [4, 1, 1], 3, "books.xml").
unlabeled('A', [1, 5, 1, 1], 4, "books.xml").
em('fine', [2, 5, 1, 1], 4, "books.xml").
unlabeled('book.', [3, 5, 1, 1], 4, "books.xml").
year('2002', [2, 1], 3, "books.xml").
author('Buneman', [1, 2, 1], 3, "books.xml").
title('XML in Scotland', [2, 2, 1], 3, "books.xml").
unlabeled('The', [1, 1, 3, 2, 1], 6, "books.xml").
em('best', [2, 1, 3, 2, 1], 6, "books.xml").
unlabeled('ever!', [3, 1, 3, 2, 1], 6, "books.xml").
```

Here we can see the translation of each tag into a predicate name: *books*, *book*, etc. Each predicate has four arguments, the first one, used for representing the XML document structure, is encapsulated into a function symbol with the same name as the tag adding the suffix *type*. Therefore, we have *bookstype*, *booktype*, etc. The second argument is used for numbering each node; the third argument of the predicates is used for numbering each type; and the last argument represents the document name. The key element of our translation is to be able to recover the original XML document from the set of rules and facts.

3 Translating XPath into Logic Programming

In this section, we present how *XPath* expressions can be translated into a logic program. Here we present the basic ideas, a more detailed description can be found in [ABE08].

We restrict ourselves to *XPath* expressions of the form $xpathexpr = /expr_1 \ldots /expr_n$ where each $expr_i$ $(1 \leq i \leq n)$ can be a tag or a tag with a *boolean condition* of the form $[xpathexpr = value]$, where *value* has a basic type. More complex *XPath* queries [W3C07a] can be expressed in *XQuery*, and therefore this restriction does not reduce the expressivity power of our query language.

With the previous assumption, each *XPath* expression $xpathexpr = /expr_1 \ldots /expr_n$ defines a *free of equalities XPath expression*, denoted by $FE(xpathexpr)$. Basically, boolean conditions $[xpathexpr = value]$ are replaced by $[xpathexpr]$ in free of equalities *XPath* expressions. These free of equalities *XPath* expressions define a subtree of the XML document, in which is required that some paths exist (occurrences of boolean conditions $[xpathexpr]$).

For instance, with respect to the *XPath* expression $/books/book\ [author = Suciu]/title$, the free of equalities *XPath* expression is $/books/book\ [author]\ /title$ and the subtree of the type and node numbered XML document which corresponds with the expression $/books/book\ [author]/title$ is as follows:

```
<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>
</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>
</book>
</books>
```

Now, given a type and node numbered XML document $\mathcal{D}$, a program $\mathcal{P}$ representing $\mathcal{D}$, and an *XPath* expression *xpathexpr* then the *logic program representing xpathexpr* is $\mathcal{P}^{xpathexpr}$, obtained from $\mathcal{P}$ taking the schema rules for the subtree of $\mathcal{D}$ defined by $FE(xpathexpr)$, and the facts of $\mathcal{P}$. For instance, with respect to the above example, the schema rules defined by $/books/book\ [author]/title$ are:

```
books(bookstype(Book, []), NBooks, 1,Doc):-
        book(Book, [NBook|NBooks], 2,Doc).
book(booktype(Author,Title,Review,[Year]),NBook,2,Doc) :-
        author(Author,[NAuthor|NBook],3,Doc),
        title(Title,[NTitle|NBook],3,Doc).
```

and the facts are the same as the original program. Let us remark that in practice, these rules can be obtained from the schema rules by removing the predicates which do not occur as tags in the free of equalities *XPath* expression. Now, given a type and node numbered XML document, and an *XPath* expression *xpathexpr*, the set of *goals obtained from xpathexpr* are defined as follows.

Firstly, each *XPath* expression *xpathexpr* can be mapped into a set of Prolog terms, denoted by $PT(xpathexpr)$, representing the *patterns* of the query. Due to XML records can have different structure, one pattern is generated for each kind of record. To each pattern t we can associate a set of type numbers, denoted by $TN(t)$.

Now, the *goals* are defined as: $\{tag(Pattern, Node, Type, doc)\{Pattern \rightarrow t, Type \rightarrow r\} \mid t \in PT(xpathexpr), r \in TN(t)\}$ where *tag* is the leftmost tag in *xpathexpr* with a boolean condition; r is a type number associated to each pattern (i.e. $r \in TN(t)$); $Pattern$, $Node$ and $Type$ are variables; and *doc* is the document name of the input XML document. In the case of *xpathexpr* without boolean conditions we have that *tag* is the rightmost one.

For instance, with respect to $/books/book\ [author = Suciu]/title$, then $PT($ $/books/\ book\ [author = Suciu]/title) = \{booktype('Suciu', Title, Review, [Year])\}$, $TN(booktype('Suciu', Title, Review, [Year])) = \{2\}$, and therefore the (unique) goal is : $-book(booktype('Suciu', Title, Review, Year), Node, 2,''books.xml'')$.

We will call *head tag* of *xpathexpr* to the leftmost tag with a boolean condition, and it will be denoted by $htag(xpathexpr)$. In the case of *xpathexpr* without boolean conditions then the head tag is the rightmost one. In the previous example, $htag(/books/book[author = Suciu]/title) = book$.

In summary, the handling of an *XPath* query involves the *"specialization"* of the schema rules of the XML document (removing predicates) and the *generation* of one or more goals. The goals are obtained from the patterns and the leftmost tag with a boolean condition on the *XPath* expression. Obviously, instead of a set of goals for each *XPath* expression, a unique goal can be considered by adding a new rule. In such a case, the head tag would be the name of the predicate of the added rule.

4 Translating XQuery into Logic Programming

Similarly to *XPath*, *XQuery* expressions can be translated into a logic program generating the corresponding goal. We will focus on a subset of *XQuery*, called *XQuery* core language, whose grammar can be defined as follows.

Core XQuery

xquery:= $dxpfree \mid < tag >' \{'xquery, \ldots, xquery'\}' < /tag > \mid flwr$.
dxpfree:= *document(doc)* '/' *xpfree*.
flwr:= **for** *\$var* **in** *vxpfree* [**where** *constraint*] **return** *xqvar* |
  let *\$var* **:=** *vxpfree* [**where** *constraint*] **return** *xqvar*.
xqvar:= $vxpfree \mid < tag >' \{'xqvar, \ldots, xqvar'\}' < /tag > \mid flwr$.
vxpfree:= *\$var* | *\$var* '/' *xpfree* | *dxpfree*.
Op:= $<= \mid >= \mid < \mid > \mid =$.
constraint := *vxpfree Op value* | *vxpfree Op vxpfree*
  | *constraint* '**or**' *constraint* | *constraint* '**and**' *constraint*.

where *value* is an XML document, *doc* is a document name, and *xpfree* is a free of equalities *XPath* expression. Let us remark that *XQuery* expressions use free of equalities *XPath* expressions, given that equalities can be always introduced in *where* expressions. We will say that an *XQuery* expression *ends with attribute name* whenever the *XQuery* expression has the form of an *vxpfree* expression, and the rightmost element has the form *@att*, where *att* is an attribute name. The translation of an *XQuery* expression involves the following steps:

- Firstly, for each *XQuery* expression *xquery*, we can define a logic program $\mathcal{P}^{xquery}$ and a goal.
- Secondly, analogously to *XPath* expressions, for each *XQuery* expression *xquery*, we can define the so-called *head tag*, denoted by $htag(xquery)$, denoting the *predicate name* used for the building of the goal (or subgoal whether the expression *xquery* is nested).

- Finally, for each *XQuery* expression *xquery*, we can define the so-called *tag position*, denoted by *tagpos*(*xquery*), representing the argument of the head tag (i.e. the argument of the predicate) in which the answer is retrieved.

In other words, in the translation each *XQuery* expression can be mapped into a program $\mathcal{P}^{xquery}$ and into a goal of the form $: -tag(\overline{Tag}, Node, Type, Docs)$, where *tag* is the head tag, $\overline{Tag} \equiv Tag_1, \ldots, Tag_n$ are variables, and Tag_{pos} represents the answer of the query, where $pos = tagpos(xquery)$. In addition, $Node$ and $Type$ are variables representing the node and type numbering of the output document, and *Docs* is a variable representing the documents involved in the query. As a particular case of *XQuery* expressions, *XPath* expressions *xpathexpr* hold that $tagpos(xpathexpr) = 1$.
As running example, let us suppose a query requesting the year and title of the books published before 2003.

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

For this query, the translation is as follows:

```
P^xquery = {
(1) mybook(mybooktype(Title, [Year]), [Node], [Type], [Doc]) : −
        join(Title, Year, Node, Type, Doc).

(2) join(Title, Year, [Node], [Type], [Doc]) : −
        vbook(Title, Year, Node, Type, Doc),
        constraints(vbook(Title, Year)).

(3) constraints(Vbook) : −lc(Vbook).
    lc(Vbook) : −c(Vbook).
    c(vbook(Title, Year)) : −le(Year, 2003).

(4) vbook(Title, Year, [Node, Node], [TTitle, TYear], "books.xml") : −
        title(Title, [NTitle|Node], TTitle, "books.xml"),
        year(Year, Node, TYear, "books.xml").
}
```

Basically, the translation of *XQuery* expressions needs to consider the following elements:

- The so-called *document variables* which are *XQuery* variables associated to XML documents by means of *for* or *let* expressions.
- Variables which are not document variables. Each one of these variables can be associated to a document variable. The value of these variables depends on the value of the associated document variable. Such dependence is expressed by means of a *for* or *let* expression. In this case, we say that the associated document variable is the *root* of the given variable.
- *XPath* expressions associated to a document variable. Such *XPath* expressions are those ones such that: (a) the document variable occurs in the *XPath* expression or (b) a variable whose root is the document variable occurs in the *XPath* expression.
- Constraints associated to a document variable. Such constraints are those including *XPath* expressions associated to the given document variable.

In the example, there is only one document variable, that is $\$book$, associated to "books.xml" by means of a *for* expression, and $\$year$ can be associated to $\$book$ whose dependence is expressed by means of the *let* expression. Therefore, $\$book$ is the *root* of $\$year$. In addition, there are two *XPath* expressions associated to $\$book$: $\$year$ and $\$book/title$. Finally, the constraint *"\$year<2003"* is associated to the document variable *\$book*. Now, the translation of *XQuery* expressions can be summarized as follows:

- The *return* expression generates one or more rules for describing the structure of the output XML document.
- Such structure is generated by means of a special predicate called *join*, defined by means of one rule, whose role is to make the join of multiple documents.
- The predicate *join* calls to predicates called *vvar*'s, one for each *\$var*, where *\$var* is a document variable.
- Each *vvar* predicate calls to the predicates of the head tags of the *XPath* expressions associated to *\$var*.
- The predicate *join* also calls to a special predicate called *constraints*, defined by one rule, whose role is to check the constraints of the *where* expressions included in the *XQuery* expression. The *constraints* predicate calls to $lc^1, \ldots, lc^n$, one for each document variable i $(1 \leq i \leq n)$, and each one of them checks a list of constraints for the given document variable i. $c_1^i, \ldots, c_m^i$ check each constraint k $(1 \leq k \leq m)$ of a document variable i.

In the example, rule **(1)** defines the structure of the output XML document according to the *return* expression in which a *mybook* record is built including *title* and *year* as attribute. Rule **(2)** of the predicate *join* generates such structure by calling the predicate *vbook* which represents the document variable *\$book*. In addition, *join* also calls to the predicate *constraints* by checking the *where* expression. The *vbook* predicate (rule **(4)**) calls to the head tags of the *XPath* expressions associated to the document variable *\$book*. In this case, the head tags of $\$year$ and $\$book/title$ are *title* and *year*, respectively. Finally, rule **(3)** declares the special predicate *constraints* which checks the constraint *"\$year<2003"* associated to *\$book*. Since there is only one document variable and one constraint, the transformation only generates predicates called *lc* and *c*, in order to check the given constraint. The predicate *le* represents the operator "<".

With respect to node and type numbering, we adopt the following convention. The output XML document can be built from several input XML documents. Therefore it is possible that the original numbering is not valid for numbering the output document. However, we can still number the output XML documents by considering as identifier (node and type number) of each record of the output document the list of identifiers (node and type numbers) of the input documents. Such numbering allow to identify each record of the output XML document. This is the reason why rules **(1)**, **(2)** and **(4)** collect in a Prolog list the type and node number of the called predicates (in this case, there is only one input document).

With respect to the goal, the head tag of each $\mathcal{P}^{xquery}$ has to be computed in each case (see next section for more details). In the example, the head tag is *mybook*, that is, $htag(xquery) = mybook$, and the *tagpos* is *1*, that is, $tagpos(xquery) = 1$. Therefore, the goal is : $-mybook(MyBook, Node, Type, Doc)$ and the answer is:

$$MyBook = mybooktype(''XML\ in\ Scottland'', [''2002'']),\ Node = [[[[2,1],[2,1]]]]$$
$$Type = [[[3,3]]],\ Doc = [[''books.xml'']]$$

This answer represents the following XML document:

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

In order to build the output XML document from the set of answers, we have to consider some *auxiliary rules* for expressing the schema of the XML output documents. In the example, the schema rules are the following:

```
mybook(mybooktype(Title, [Year]), [[[Node1, Node2]]], [[[Type1, Type2]]], [[Doc]]) :-
        title(Title, [NTitle|Node1], Type1, Doc),
        year(Year, Node2, Type2, Doc).
```

Similarly to input documents, in output XML documents the children are numbered with a larger number than parents. In the example the *mybook* element is numbered as $[[[[2,1],[2,1]]]]$ and the child *title* is numbered as $[2,2,1]$.

4.1 Formalizing the Transformation

In this section, we show an algorithm for encoding XQuery in logic programming. This algorithm will be illustrated with an example. Assuming the notation of Table 1, the algorithm is shown in Tables 2 and 3. The algorithm has the following elements:

(1) It distinguishes cases for each type of *XQuery* expression;
(2) It defines the values for $\mathcal{P}^{xquery}$, $htag(xquery)$ and $tagpos(xquery)$ in each case;
(3) It uses the notation $\mathcal{P}_{\Gamma}^{\mathcal{X}}$ in order to denote the encoding of a set $\mathcal{X}$ of *XQuery* expressions w.r.t. a context Γ;
(4) The context Γ includes assertions of the form $(\$var, let, xpathexpr, C)$ and $(\$var, for, xpathexpr, C)$ whose meaning is the following: the *XQuery* variable $\$var$ has been assigned to *xpathexpr* by means of a *let* (resp. a *for*) expression with the list of constraints C.

The most relevant cases of the algorithm are cases **(2)** of Table 2, and **(8)** of Table 3.

Case **(2)** introduces the rule for providing structure to the output document. The set $\{tag_1, \ldots, tag_k, att_1, \ldots, att_s\}$ contains the head tags of the expressions $xquery_1, \ldots, xquery_n$, and for each one of them, the type and node numbers are collected in a Prolog list. In addition, the tag position allows to know which arguments have to be selected from the call to the head tags (it is expressed in the conditions of case **(2)**).

Table 1. Notation

$Vars(\Gamma) =_{def} \{\$var \mid (\$var, let, vxpfree, C) \in \Gamma\ or\ (\$var, for, vxpfree, C) \in \Gamma\}$; Denotes the variables of a context Γ;
$DocVars(\Gamma) =_{def} \{\$var \mid (\$var, let, dxpfree, C) \in \Gamma\ or\ (\$var, for, dxpfree, C) \in \Gamma\}$; Denotes the document variables of a context Γ;
$Doc(\$var, \Gamma) =_{def} doc$ whenever $\overline{\Gamma}_{\$var} = document(doc)/xpfree$; Denotes the document associated to a document variable $\$var$ in a context Γ;
$\Gamma_{\$var} =_{def} vxpfree$ whenever $(\$var, let, vxpfree, C)$ or $(\$var, for, vxpfree, C) \in \Gamma$; Denotes the *XPath* expression associated to a variable $\$var$ in a context Γ;
$\overline{\Gamma}_{\$var} =_{def} vxpfree[\lambda_1 \cdot \ldots \cdot \lambda_n]$ where $\lambda_i = \{\$var_i \rightarrow \Gamma_{\$var_i}\}$ and $\{\$var_1, \ldots, \$var_n\} = Vars(\Gamma)$; Denotes the free of variables *XPath* expression associated to a variable $\$var$ in a context Γ; Variables are replaced by the associated *XPath* expression;
$Root(\$var) =_{def} \$var'$ whenever $\$var \in DocVars(\Gamma)$ and $\$var = \$var'$ or $((\$var, let, \$var''/xpfree, C) \in \Gamma\ or\ (\$var, for, \$var''/xpfree, C) \in \Gamma$ and $Root(\$var'') = \$var')$; Denotes the root of a given variable $\$var$;
$Rootedby(\$var, \mathcal{X}) =_{def} \{xpfree \mid \$var/xpfree \in \mathcal{X}\}$; Denotes the *XPath* expression associated to $\$var$ in $\mathcal{X}$;
$Rootedby(\$var, \Gamma) =_{def} \{xpfree \mid \$var/xpfree\ Op\ \ vxpfree \in C$ or $\$var/xpfree\ \ Op\ value \in C,\ C \in Constraints(\$var, \Gamma)\}$; Denotes the *XPath* expression associated to $\$var$ in a context Γ;
$Constraints(\$var, \Gamma) =_{def} \{C_i \mid 1 \leq i \leq n,\ C \equiv C_1\ Op\ \ldots\ Op\ C_n,$ $(\$var, let, vxpfree, C) \in \Gamma\ or\ (\$var, for, vxpfree, C) \in \Gamma\}$ Denotes the list of constraints associated to $\$var$ in a context Γ;

Case **(8)** properly introduces the rule of *join*, which calls *vvar* predicates for each document variable $\$var$. In addition, the *join* predicate calls to the *constraints* predicate. The tag position indicates the argument to be selected from the call to the *vvar* predicate (condition **(b)**). Each *vvar* predicate calls to the head tags of the *XPath* expressions associated to the document variable $\$var$ (condition **(c)**). Finally, the *constraints* predicate calls to predicates $lc^1, \ldots, lc^n$ which check each constraint in a sequential way if the connective is **and**, and otherwise, the algorithm introduces alternative rules for each **or** connective (conditions **(e)** and **(f)**).

In the running example, case **(2)** is applied to $< mybook > \$year, \$book/title < /mybook >$, and the head tags of $\$year$ and $\$book/title$ are *join*. For this reason the rule **(1)** of the running example has the form $mybook(\ldots) : - join(\ldots)$. Case **(8)** is applied to *vbook*, calling the predicates *title* and *year* which are the head tags of the associated *XPath* expressions $\$year$ and $\$book/title$. Finally, the *constraints* predicate calls to *lc*, which at the same time calls to *c* for checking the constraint $\$year < 2003$.

As an example of application of the algorithm, let us suppose the following *XQuery* expression:

Table 2. Translation of XQuery into Logic Programming

(1) $\mathcal{P}^{document(doc)/xpfree} =_{def} \mathcal{P}^{xpfree}$ $htag(document(doc)/xpfree) =_{def} htag(xpfree)$ $tagpos(document(doc)/xpfree) =_{def} tagpos(xpfree)$	
(2) $\mathcal{P}^{<tag>\{xquery_1,\ldots,xquery_n\}</tag>} =_{def}$ $\{\mathcal{R}\} \cup_{1\leq i\leq n} \mathcal{P}^{xquery_i}$ *and* $\mathcal{R} \equiv$ $tag(tagtype(Tag^{1}_{p_1},\ldots,Tag^{k}_{p_k},[Att^{1}_{q_1},\ldots,Att^{s}_{q_s}]),$ $[NTag_1,\ldots,NTag_k,NAtt_1,\ldots,NAtt_s],$ $[TTag_1,\ldots,TTag_k,TAtt_1,\ldots,TAtt_s],$ $[DTag_1,\ldots,DTag_k,DAtt_1,\ldots,DAtt_s]) :-$ $tag_1(\overline{Tag^1},NTag_1,TTag_1,DTag_1),$ $\ldots$ $tag_k(\overline{Tag^k},NTag_k,TTag_k,DTag_k),$ $att_1(\overline{Att^1},NAtt_1,TAtt_1,DAtt_1),$ $\ldots$ $att_s(\overline{Att^s},NAtt_s,TAtt_s,DAtt_s).$ $htag(xquery) =_{def} tag,\ tagpos(xquery) =_{def} 1$	– $\overline{Tag^t}$ $1 \leq t \leq k$, *denotes* $Tag^t_1,\ldots,Tag^t_r$ *where* r *is the arity of* tag_t; – $\overline{Att^j}$ $1 \leq j \leq s$, *denotes* $Att^j_1,\ldots,Att^j_s$ *where* s *is the arity of* att_j; – *for every* $j \in \{1,\ldots,n\}$ $htag(xquery_j) = att_i$, $tagpos(xquery_j) = q_i$, $1 \leq i \leq s$, *whenever* $xquery_j$ *ends with attribute names,* *and* $htag(xquery_j) = tag_t$, $tagpos(xquery_j) = p_t$ $1 \leq t \leq k$, *otherwise*
(3) $\mathcal{P}^{\textbf{for}\ \$var\ \textbf{in}\ vxpfree\ [\textbf{where}\ C]\ \textbf{return}\ xqvar} =_{def}$ $\mathcal{P}^{xqvar}_{\{(\$var,for,vxpfree,C)\}}$ $htag\ (xquery) =_{def} htag(xqvar)$ $tagpos(xquery) =_{def} tagpos(xqvar)$	
(4) $\mathcal{P}^{\textbf{let}\ \$var\ :=\ vxpfree\ [\textbf{where}\ C]\ \textbf{return}\ xqvar} =_{def}$ $\mathcal{P}^{xqvar}_{\{(\$var,let,vxpfree,C)\}}$ $htag\ (xquery) =_{def} htag(xqvar)$ $tagpos(xquery) =_{def} tagpos(xqvar)$	
(5) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def}$ $\{\mathcal{R}\} \cup \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\}\cup_{1\leq i\leq n}\{xqvar_i/xpfree_0\}}_{\Gamma}$ *and* $\mathcal{R} \equiv$ $tag(tagtype(Tag_1,\ldots,Tag_r,[Att^1,\ldots,Att^m]),$ $[Node_1,\ldots,Node_s],$ $[Type_1,\ldots,Type_s],$ $[Doc_1,\ldots,Doc_s]) :-$ $tag_1(\overline{Tag^1},Node_1,Type_1,Doc_1),$ $\ldots$ $tag_s(\overline{Tag^s},Node_s,Type_s,Doc_s).$ $htag(xquery/xpfree) =_{def} tag$ $tagpos(xquery/xpfree) =_{def} 1$	– $\overline{Tag}^t$ $(1 \leq t \leq s)$ *denotes* $Tag^t_1,\ldots,Tag^t_a$ *where* a *is the arity of* tag_t; – $xquery/xpfree \in \mathcal{X}$ *and* $xpfree \equiv /tag/xpfree_0$; – $xquery \equiv <tag>\{xqvar_1,\ldots,$ $xqvar_n\}</tag>$; – $\{tag_1,\ldots,tag_s\} =$ $\{htag\ (xqvar_i\ /xpfree_0) \mid$ $1 \leq i \leq n\}$; – *for every* $p \in \{1,\ldots,n\}$ $Tag_i = Tag^j_{p_j}$, $1 \leq i \leq r$, *whenever* $tagpos(xqvar_p\ /xpfree_0) = p_j$, $htag(xqvar_p\ /xpfree_0) = tag_j$, *and* $Att^l = Tag^j_{p_j}$, $1 \leq l \leq m$, *whenever* $tagpos(xqvar_p\ /xpfree_0) = p_j$, $htag(xqvar_p\ /xpfree_0) = tag_j$ *and* $xqvar_p\ /xpfree_0$ *ends with attribute names*
(6) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\}\cup\{xqvar/xpfree\}}_{\Gamma\cup\{(\$var,for,vxpfree,C)\}}$ $htag(xquery/xpfree) =_{def} htag(xqvar/xpfree)$ $tagpos(xquery/xpfree) =_{def} tagpos(xqvar/xpfree)$	– $xquery/xpfree \in \mathcal{X}$, – $xquery \equiv$ **for** $\$var$ **in** $vxpfree$ [**where** C] **return** $xqvar$
(7) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \mathcal{P}^{\mathcal{X}-\{xquery/xpfree\}\cup\{xqvar/xpfree\}}_{\Gamma\cup\{(\$var,let,vxpfree,C)\}}$ $htag(xquery/xpfree) =_{def} htag(xqvar/xpfree)$ $tagpos(xquery/xpfree) =_{def} tagpos(xqvar/xpfree)$	– $xquery/xpfree \in \mathcal{X}$, – $xquery \equiv$ **let** $\$var := vxpfree$ [**where** C] **return** $xqvar$

Table 3. Translation of XQuery into Logic Programming (cont'd)

(8) $\mathcal{P}^{\mathcal{X}}_{\Gamma} =_{def} \bigcup_{\substack{\$var \in DocVars(\Gamma), \\ \$var = Root(\$var'), \\ xpfree \in Rootedby(\$var', \mathcal{X}) \cup Rootedby(\$var', \Gamma)}} \{\mathcal{J}^{\Gamma}\} \cup \mathcal{C}^{\Gamma} \cup \{\mathcal{R}^{\$var} \mid \$var \in DocVars(\Gamma)\} \quad \mathcal{P}^{\bar{\Gamma}_{\$var'}/xpfree}$ **(a)**	**(a)** – $\mathcal{X}$ *does not include tagged elements and flwr expressions*
$\mathcal{J}^{\Gamma} \equiv$ $join(Tag_1, \ldots, Tag_m, [Node_1, \ldots, Node_n],$ $[Type_1, \ldots, Type_n], [Doc_1, \ldots, Doc_n]) :-$ $vvar_1(\overline{Tag^1}, Node_1, Type_1, Doc_1),$ $\ldots$ $vvar_n(\overline{Tag^n}, Node_n, Type_n, Doc_n),$ $constraints(vvar_1(\overline{Tag^1}), \ldots, vvar_n(\overline{Tag^n})).$ **(b)**	**(b)** – $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$; – *for each* $\$var'/xpfree_j \in \mathcal{X}$ *such that* $Root(\$var') = \$var_i$ *and* $tagpos(\overline{\Gamma}_{\$var'}/xpfree_j) = p_j$ *then* $Tag_j = Tag^i_{p_j}$ – $\overline{Tag^i} = Tag^i_1 \ldots Tag^i_s$ *one* Tag^i_r, $1 \leq r \leq s$ *for each* $\$var'/xpfree_r \in \mathcal{X} \cup \Gamma$ *such that* $Root(\$var') = \$var_i$
$\mathcal{R}^{\$var} \equiv$ $vvar(Tag_1, \ldots, Tag_n, Node, [Type_1, \ldots, Type_n], doc) :-$ $tag_1(Tag_1, [Node_{11}, \ldots, Node_{1k_1} \mid NTag], Type_1, doc),$ $\ldots,$ $tag_n(Tag_n, [Node_{n1}, \ldots, Node_{nk_n} \mid NTag], Type_n, doc).$ **(c)**	**(c)** – $doc = Doc(\$var, \Gamma)$ – $tag_i = htag(\overline{\Gamma}_{\$var'}/xpfree)$ $\$var'\ /xpfree \in \mathcal{X}$ $\$var = Root(\$var')$ – $Node = [N_1, \ldots, N_n]$ *and* $N_i = [Node_{ik_i} \mid NTag]$ *if* $(\$var', for, vxpfree, C) \in \Gamma$, *and* $N_i = NTag$, *otherwise*
$\mathcal{C}^{\Gamma} \equiv \{$ $constraints(Vvar_1, \ldots, Vvar_n) :-$ $lc^1_1(Vvar_1, \ldots, Vvar_n),$ $\ldots$ $lc^n_1(Vvar_1, \ldots, Vvar_n).$ $\} \cup_{\$var \in Vars(\Gamma), C^j \in constraints(\$var, \Gamma)} \mathcal{C}^j$ **(d)**	**(d)** $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$
$\mathcal{C}^j \equiv$ $\{lc^j_i(Vvar_1, \ldots, Vvar_n) :-$ $c^j_i(Vvar_1, \ldots, Vvar_n), lc^j_{i+1}(Vvar_1, \ldots, Vvar_n).$ $\mid 1 \leq i \leq n, Op_i = \mathbf{and}\}$ $\cup$ $\{lc^j_i(Vvar_1, \ldots, Vvar_n) :- c^j_i(Vvar_1, \ldots, Vvar_n).$ $lc^j_i(Vvar_1, \ldots, Vvar_n) :- lc^j_{i+1}(Vvar_1, \ldots, Vvar_n).$ $\mid 1 \leq i \leq n, Op_i = \mathbf{or}\}$ $\cup_{\{c^j_i \mid 1 \leq i \leq n\}} \{\mathcal{C}^j_i\}$ **(e)**	**(e)** – $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$, – $C^j \equiv c^j_1 Op_1 \ldots, Op_n c^j_n$
$\mathcal{C}^j_i \equiv c^j_i(vvar_1(\overline{Tag^1}), \ldots, vvar_n(\overline{Tag^n})) :- Op(Tag^k_j, value).$ **(*)** $\mathcal{C}^j_i \equiv c^j_i(vvar(\overline{Tag^1}), \ldots, vvar(\overline{Tag^n})) :- Op(Tag^k_j, Tag^m_r).$ **(**)** **(f)**	**(f)** – $\{\$var_1, \ldots, \$var_n\} = DocVars(\Gamma)$ – **(*)** $c^j_i \equiv \$var'/xpfree_j$ *Op value and* $Root(\$var') = \$var_k$ – **(**)** $c^j_i \equiv \$var'/xpfree_j$ *Op* $\$var'/xpfree_r$, $Root(\$var') = \$var_k$ *and* $Root(\$var') = \$var_m$
$htag(\$var/xpfree_j) =_{def} join$ $tagpos(\$var/xpfree_j) =_{def} j$ **(g)**	**(g)** *for every* $\$var \in Vars(\Gamma)$, $xpfree_j \in Rootedby(\$var, \mathcal{X}) \cup Rootedby(\$var, \Gamma)$

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

requesting the reviews of books (published before 2003) occurring in two documents: the first one is the running example and the second one is:

```
<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>Good reference!</review>
  </book>
</books>
```

In this case, the *return* expression generates a new rule *mybook* in which the *title* is obtained from the first document and *review*'s are obtained from both documents. The application of the algorithm is as follows:

$$\mathcal{P}^{xquery} =_{(Rule(\mathbf{4}))} \mathcal{P}^{xquery_1}_{(\$store1, let, document(''books1.xml'')/books, \emptyset)} =_{(Rule(\mathbf{4}))}$$
$$\mathcal{P}^{xquery_2}_{\Gamma_1} =_{(Rule(\mathbf{3}))} \mathcal{P}^{xquery_3}_{\Gamma_1 \cup \{(\$book1, for, \$store1, \emptyset)\}} =_{(Rule(\mathbf{3}))}$$
$$\mathcal{P}^{xquery_4}_{\Gamma_1 \cup \{(\$book1, for, \$store1, \emptyset), (\$book2, for, \$store2, \emptyset)\}} =_{(Rule(\mathbf{4}))}$$
$$\mathcal{P}^{xquery_5}_{\Gamma_2} =_{(Rule(\mathbf{2}))} \{\mathcal{R}\} \cup \mathcal{P}^{\$title, \$book1/review, \$book2/review}_{\Gamma_2}$$

where $\Gamma_1 = \{(\$store1, let, document(''books1.xml'')/books, \emptyset), (\$store2, let, document\ (''books2.xml'')/\ books, \emptyset)\}$ and also $\Gamma_2 = \Gamma_1 \cup \{(\$book1, for, \$store1, \emptyset), (\$book2, for, \$store2, \emptyset), (\$title, let,\ \$book1/\ title,\ \$book1/\ @year < 2003\ and\ \$title = \$book2/title)\}$. In addition, $\mathcal{R}$ is defined as follows:

$$\mathcal{R} = \begin{array}{l} mybook(mybooktype(Title, Review1, Review2, []), [Node], [Type], [Doc]) :- \\ \quad join(Title, Review1, Review2, Node, Type, Doc). \end{array}$$

where $join = htag(\$title)$, $join = htag(\$book1/review)$, $join = htag(\$book2/review)$, $tagpos(\$title) = 1$, $tagpos\ (\$book1/review) = 2$, $tagpos(\$book2/review) = 3$.
Now, $\mathcal{P}^{\$title, \$book1/review, \$book2/review}_{\Gamma_2}$ is defined as:

$$\begin{array}{l} \mathcal{P}^{\$title, \$book1/review, \$book2/review}_{\Gamma_2} =_{Rule(\mathbf{8})} \\ \quad \{\mathcal{J}^{\Gamma}\} \cup \mathcal{C}^{\Gamma} \cup \{\mathcal{R}^{\$store1}, \mathcal{R}^{\$store2}\} \cup \\ \quad \mathcal{P}^{document(books1.xml)/books/book/title} \cup \\ \quad \mathcal{P}^{document(books1.xml)/books/book/@year} \cup \\ \quad \mathcal{P}^{document(books1.xml)/books/book/review} \cup \\ \quad \mathcal{P}^{document(books2.xml)/books/book/title} \cup \\ \quad \mathcal{P}^{document(books2.xml)/books/book/review} \end{array}$$

where $\mathcal{J}^{\Gamma}$ and $\mathcal{C}^{\Gamma}$ are defined as follows:

$$\mathcal{J}^{\Gamma} = \begin{array}{l} join(Title1, Review1, Review2, [Node1, Node2], [Type1, Type2], [Doc1, Doc2]) :- \\ \quad vstore1(Title1, Year1, Review1, Node1, Type1, Doc1), \\ \quad vstore2(Title2, Review2, Node2, Type2, Doc2), \\ \quad constraints(vstore1(Title1, Year1, Review1), vstore2(Title2, Review2)). \end{array}$$

$\mathcal{C}^{\Gamma} = \{$
$constraints(Vstore1, Vstore2) :-$
$\quad lc^{1}(Vstore1, Vstore2).$
$lc^{1}_{1}(Vstore1, Vstore2) :- c^{1}_{1}(Vstore1, Vstore2),$
$\quad c^{1}_{2}(Vstore1, Vstore2).$
$c^{1}_{1}(vstore1(Title1, Year1, Review1), vstore2(Title2, Review2)) :-$
$\quad le(Year1, 2003).$
$c^{1}_{2}(vstore1(Title1, Year1, Review1), vstore2(Title2, Review2)) :-$
$\quad eq(Title1, Title2).$
$\}$

where

- $DocVars(\Gamma) = \{\$vstore1, \$vstore2\}$,
- $\$title$, $\$book1\ /review$, $\$book2\ /review \in \mathcal{X}$,
- $Root(\$title) = \$vstore1$, $Root(\$book1) = \$vstore1$ and $Root(\$book2) = \$vstore2$,
- $\$book1/@year$ and $\$book2/title$ occur in Γ,
- $Root(\$book1) = \$vstore1$ and $Root(\$book2) = \$vstore2$,
- $C^{1} \equiv c^{1}_{1}$ **and** $c^{1}_{2} \in \Gamma$, $c^{1}_{1} \equiv \$book1/@year < 2003$, $c^{1}_{2} \equiv \$title = \$book2/title$,
- $Root(\$book1) = \$vstore1$, $Root(\$title) = \$vstore1$ and $Root(\$book2) = \$vstore2$

Finally, $\mathcal{R}^{\$store1}$ and $\mathcal{R}^{\$store2}$ are defined as:

$\mathcal{R}^{\$store1} =$
$vstore1(Title, Year, Review, [Node, Node, Node], [Type_1, Type_2, Type_3],$
$\quad ''books1.xml'') :-$
$\quad title(Title, [Node_1, Node_2|Node], Type_1, ''books1.xml''),$
$\quad year(Year, [Node_2|Node], Type_2, ''books1.xml''),$
$\quad review(Review, [Node_1, Node_2|Node], Type_3, ''books1.xml'').$
$\mathcal{R}^{\$store2} =$
$vstore2(Title, Review, [Node, Node], [Type_1, Type_2], ''books2.xml'') :-$
$\quad title(Title, [Node_1, Node_2|Node], Type_1, ''books2.xml''),$
$\quad review(Review, [Node_1, Node_2|Node], Type_2, ''books2.xml'').$

and

$\mathcal{P}^{document(books1.xml)/books/book/title} = Facts\ of\ \mathcal{P}$
$\mathcal{P}^{document(books1.xml)/books/book/@year} = Facts\ of\ \mathcal{P}$
$\mathcal{P}^{document(books2.xml)/books/book/title} = Facts\ of\ \mathcal{P}$
$\mathcal{P}^{document(books1.xml)/books/book/review} =$
$\mathcal{P}^{document(books2.xml)/books/book/review} =$
$\{$
$review(reviewtype(Unlabeled, Em, []), NReview, 3, Doc) :-$
$\quad unlabeled(Unlabeled, [NUnlabeled|NReview], 4, Doc),$
$\quad em(Em, [NEm|NReview], 4, Doc).$
$review(reviewtype(Em, []), NReview, 3, Doc) :-$
$\quad em(Em, [NEm|NReview], 5, Doc).$
$em(emtype(Unlabeled, Em, []), NEms, 5, Doc) :-$
$\quad unlabeled(Unlabeled, [NUnlabeled|NEms], 6, Doc),$
$\quad em(Em, [NEm|NEms], 6, Doc).$
$\} \cup Facts\ of\ \mathcal{P}$

where

- $''books1.xml'' = Doc(\$vstore1, \Gamma), ''books2.xml'' = Doc(\$vstore2, \Gamma)$
- $htag(document\ (''books1.xml'')\ /books/book\ /title) = title$
- $htag(document\ (''books1.xml'')\ /books/book\ /year) = year$
- $htag(document\ (''books1.xml'')\ /books/book\ /review) = review$
- $\overline{\Gamma}_{\$title} = document\ (''books1.xml'')\ /books/book/$
- $\overline{\Gamma}_{\$book1} = document\ (''books1.xml'')\ /books/book/$
- $\overline{\Gamma}_{\$book2} = document\ (''books2.xml'')\ /books/book/$

5 Conclusions and Future Work

In this paper, we have studied how to encode *XQuery* expressions into logic programming. It allows us to evaluate *XQuery* expressions against XML documents using logic rules.

As far as we know, this is the first time that *XQuery* is implemented in logic programming. Previous proposals in this research area are mainly focused on the definition of *new* query languages of logic style [SB02, CF07, May04, Sei02] and functional style [HP03, BCF05] for XML documents, and the only proposal for *XQuery* implementation takes as host language a functional language (i.e. *OCAML*). The proposals of new query languages in this framework have to adapt the unification in the case of logic languages [BS02b, May04, CF03], and the pattern matching in the case of functional languages [BCF05, HP03] in order to accommodate the handling of XML records. However, in our case, we can adopt standard term unification by encoding XML documents into logic programming, and therefore one of the advantages of our approach is that it can be integrated with any Prolog implementation. In addition, the advantage of a logic-based implementation of *XQuery* is that, *XQuery* can be combined with logic programs. Logic programs can be used, for instance, for representing RDF and OWL documents (see, for instance, [WSW03, Wol04]), and therefore XML querying and processing can be combined with RDF and OWL reasoning in our framework –in fact, we have been recently working in a proposal in this line [Alm08].

On the other hand, the proposal of this paper also contributes to the study of the representation and handling of XML documents in relational database systems. In our framework, logic programs represent XML documents by means of rules and a table of facts. In addition, the table of facts is indexed in secondary memory for improving the retrieval. Similar processing and storing can be found in the proposals of XML processing with relational databases (see [BGvK+05], [OOP+04] and [TVB+02]). In fact, we plan to implement the storing of facts in a relational database management system in order to improve fact storing and retrieval.

Therefore our proposal of a *logic-based query language for the Semantic Web* combines the advantages of efficient retrieval of facts in a relational database style together with reasoning capabilities of logic programming.

As future work we would like to implement our technique. We have already implemented *XPath* in logic programming (see `http://indalog.ual.es/Xindalog`). Taking as basis this implementation we would like to extend it to *XQuery* expressions.

References

[ABE06] Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Magic sets for the XPath language. Journal of Universal Computer Science 12(11), 1651–1678 (2006)

[ABE08] Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Querying XML documents in logic programming. Theory and Practice of Logic Programming 8(3), 323–361 (2008)

[ACJ04] Atanassow, F., Clarke, D., Jeuring, J.: UUXML: A Type-Preserving XML Schema Haskell Data Binding. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 71–85. Springer, Heidelberg (2004)

[Alm08] Almendros-Jiménez, J.M.: An RDF Query Language based on Logic Programming. In: Proceedings of the 3rd Int'l. Workshop on Automated Specification and Verification of Web Systems. Electronic Notes on Theoretical Computer Science, vol. 200, pp. 67–85 (2008)

[BBFS05] Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and Semantic Web Query Languages: A Survey. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 35–133. Springer, Heidelberg (2005)

[BCF05] Benzaken, V., Castagna, G., Frish, A.: CDuce: an XML-centric general-purpose language. In: Proc. of the ACM SIGPLAN International Conference on Functional Programming, pp. 51–63. ACM Press, New York (2005)

[BCM05] Benzaken, V., Castagna, G., Miachon, C.: A full pattern-based paradigm for XML query processing. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 235–252. Springer, Heidelberg (2005)

[BFG01] Baumgartner, R., Flesca, S., Gottlob, G.: The elog web extraction language. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 548–560. Springer, Heidelberg (2001)

[BGvK+05] Boncz, P.A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: Pathfinder: XQuery - The Relational Way. In: Proc. of the International Conference on Very Large Databases, pp. 1322–1325. ACM Press, New York (2005)

[BHL01] Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web – A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. In: Scientific American, 36 pages (May 2001)

[Bol00a] Boley, H.: Relationships Between Logic Programming and RDF. In: Kowalczyk, R., et al. (eds.) PRICAI-WS 2000. LNCS, vol. 2112, pp. 201–218. Springer, Heidelberg (2001)

[Bol00b] Boley, H.: Relationships between logic programming and XML. In: Proc. of the Workshop on Logic Programming, pp. 19–34, Würzburg, Germany (2000) (GMD Report 90)

[Bol01] Boley, H.: The rule markup language: RDF-XML data model, XML schema hierarchy, and XSL transformations. In: Bartenstein, O., Geske, U., Hannebauer, M., Yoshie, O. (eds.) INAP 2001. LNCS, vol. 2543, pp. 124–139. Springer, Heidelberg (2003)

[BS02a] Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In: Chaudhri, A.B., et al. (eds.) NODe-WS 2002. LNCS, vol. 2593, pp. 295–310. Springer, Heidelberg (2003)

[BS02b] Bry, F., Schaffert, S.: Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 255–270. Springer, Heidelberg (2002)

[CDF+04] Chamberlin, D., Draper, D., Fernández, M., Kay, M., Robie, J., Rys, M., Simeon, J., Tivy, J., Wadler, P.: XQuery from the Experts. Addison-Wesley, Reading (2004)

[CF03] Coelho, J., Florido, M.: Type-based XML Processing in Logic Programming. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, pp. 273–285. Springer, Heidelberg (2002)

[CF04] Coelho, J., Florido, M.: CLP(Flex): Constraint logic programming applied to XML processing. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3291, pp. 1098–1112. Springer, Heidelberg (2004)

[CF07] Coelho, J., Florido, M.: XCentric: logic programming for XML processing. In: WIDM 2007: Proceedings of the 9th annual ACM international workshop on Web information and data management, pp. 1–8. ACM Press, New York (2007)

[CH01] Cabeza, D., Hermenegildo, M.: Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. Theory and Practice of Logic Programming 1(3), 251–282 (2001)

[Cha02] Chamberlin, D.: XQuery: An XML Query Language. IBM Systems Journal 41(4), 597–615 (2002)

[HP03] Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)

[May04] May, W.: XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. Theory and Practice of Logic Programming 4(3), 239–287 (2004)

[MS03] Marian, A., Simeon, J.: Projecting XML Documents. In: Proc. of International Conference on Very Large Databases, Burlington, USA, pp. 213–224. Morgan Kaufmann, San Francisco (2003)

[OOP+04] O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: OrdPaths: Insert-friendly XML Node Labels. In: Proc. of the ACM SIGMOD Conference, pp. 903–908. ACM Press, New York (2004)

[SB02] Schaffert, S., Bry, F.: A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In: Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Aachen, Germany, CEUR Workshop Proceedings 60, 22 pages (2002)

[Sei02] Seipel, D.: Processing XML-Documents in Prolog. In: Procs. of the Workshop on Logic Programming, 15 pages, Dresden, Germany, Technische Universität Dresden (2002)

[Thi02] Thiemann, P.: A typed representation for HTML and XML documents in Haskell. Journal of Functional Programming 12(4&5), 435–468 (2002)

[TVB+02] Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and Querying Ordered XML using a Relational Database System. In: Proc. of the ACM SIGMOD Conference, pp. 204–215. ACM Press, New York (2002)

[W3C07a] W3C. XML Path Language (XPath) 2.0. Technical report (2007), `http://www.w3.org/TR/xpath`

[W3C07b] W3C. XML Query Working Group and XSL Working Group, XQuery 1.0: An XML Query Language. Technical report (2007), `http://www.w3.org`

[Wad02] Wadler, P.: XQuery: A Typed Functional Language for Querying XML. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 188–212. Springer, Heidelberg (2003)

[Wie05] Wielemaker, J.: SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam (March 2005)

[Wol04] Wolz, R.: Web Ontology Reasoning with Logic Databases. PhD thesis, Universität Fridericiana zu Karlsruhe (2004)

[WR99] Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proceedings of the International Conference on Functional Programming, pp. 148–159. ACM Press, New York (1999)

[WSW03] Wielemaker, J., Schreiber, G., Wielinga, B.J.: Prolog-Based Infrastructure for RDF: Scalability and Performance. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 644–658. Springer, Heidelberg (2003)