A Rule-based Implementation of XQuery

J. M. Almendros-Jiménez
Dpto. Lenguajes y Computación.
Universidad de Almería, Spain

Abstract
In this paper we describe the implementation of (a subset of) the XQuery language using logic programming (in particular, by means of Prolog). Such implementation has been developed using the Prolog interpreter SWI-Prolog. XML files are handled by means of the XML Library of SWI-Prolog. XPath/XQuery are encoded by means of Prolog rules. Such Prolog rules are executed in order to obtain the answer of the query.

Keywords: Database Query Languages, Logic Programming, XQuery Language.

1 Introduction

The W3C (World Wide Web Consortium) provides a suitable standard language to express XML document transformations and to query data, the XQuery language [14,11,15,10]. XQuery is a typed functional language containing XPath [13] as a sublanguage. XPath supports navigation, selection and extraction of fragments from XML documents. XQuery also includes the so-called flwor expressions (i.e. for-let-where-orderby-return expressions) to construct new XML values and to join multiple documents. XQuery has statically typed semantics and a formal semantics which is part of the W3C standard [11,14].

In this paper we investigate how to implement (a subset of) the XQuery language using logic programming (in particular, by means of Prolog). With this aim:

(i) XML documents can be handled in Prolog by means of the XML library available in most Prolog interpreters (this is the case, for instance, of SWI-Prolog [16] and CIAO [9]). Such library allows to load and parse XML files, representing them in Prolog by means of a Prolog term.

(ii) We have to study how to implement XPath and XQuery by means of logic programming. In other words, we have to study how to encode XPath and XQuery queries by means of Prolog rules.

1 This work has been partially supported by the Spanish MICINN under grant TIN2008-06622-C03-03
2 Email: jalmen@ual.es
(iii) Such rules are executed in order to obtain the output of the query. The XML library of Prolog is also used for generating the output.

In previous works we have already studied how to use logic programming for processing XML data. This work continues this research line in the following sense. In [4] we have studied how to encode XPath by means of rules and how to define a Magic Set Transformation [6] in order to execute XPath queries by means of Datalog following a Bottom-Up approach. In [5] we have described how to encode XPath by means of rules but in this case, the execution model is Top-Down, and therefore XPath can be executed by means of Prolog. In [3], we have described how to encode XQuery by means of rules, following a Top-Down approach, but with the aim to be integrated with the XPath encoding studied in [5].

Now, in this paper, we have studied a different approach to the same problem: how to encode XPath and XQuery by means of rules, but with the aim to integrate the encoding with the XML library available in most Prolog interpreters. Usually, Prolog libraries for XML allow to load a XML document from a file, storing the XML document by means of a Prolog term representing the XML tree. In our previous works [5,3], XML documents are represented by means of rules and facts. The current proposal uses the encoding of XML documents by means of a Prolog term. The difference of encoding of XML documents has as a consequence that XPath and XQuery languages have now to be re-encoded for admitting XML documents represented by means of a Prolog term. In order to test our proposal we have developed a prototype which can be downloaded from our Web page http://indalog.ual.es/XQuery.

With respect to existing XQuery implementations, our proposal uses as host language a logic language based on rules like Prolog. As far as we know our proposal is the first approach for implementing XQuery in logic programming. The existing XQuery implementations either use functional programming (with Objective Caml as host language) or Relational Database Management Systems (RDBMS).

In the first case, the Galax implementation [12] encodes XQuery into Objective Caml, in particular, encodes XPath. Since XQuery is a functional language (with some extensions) the main encoding is related with the type system for allowing XML documents and XPath expressions to occur in a functional expression. With this aim an specific type system for handling XML tags, the hierarchical structure of XML, and sequences of XML items is required. In addition, XPath expressions can be implemented from this representation. The XQuery expressions which do not correspond to pure functional syntax can be also encoded in the host language thanks to the type system.

In our case, SWI-Prolog lacks on a type system, however Prolog is able to handle trees and the hierarchical structure of XML documents by means of Prolog terms. The XML library of SWI-Prolog loads XML documents from a file and represents them by means of a Prolog term of hierarchical structure. XPath is implemented in our approach by traversing the hierarchical structure of the Prolog term. XQuery is implemented by encoding the flwor expressions by means of Prolog rules.

In the second case, XQuery has been implemented by using a RDBMS. It evolves in most of cases the encoding of XML documents by means of relational tables and the encoding of XPath and XQuery. The most relevant contribution in this research
Fig. 1. XML document

```xml
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>

<book year="1992">
  <title>Advanced Programming in the Unix environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>

<book year="2000">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <author><last>Suciu</last><first>Dan</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
</book>

<book year="1999">
  <title>The Economics of Technology and Content for Digital TV</title>
  <editor><last>Gerbarg</last><first>Darcy</first></editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
</book>
```

line is MonetDB/XQuery [7]. It consists of the Pathfinder XQuery compiler [8] on top of the MonetDB RDBMS, although Pathfinder can be deployed on top of any RDBMS. MonetDB/XQuery encodes the XML tree structure in a relational table following a pre/post order traversal of the tree (with some variant). XPath can be implemented from such table-based representation. XQuery can be implemented by encoding `flwor` expressions into the relational algebra, extended with the so-called `loop-lifted staircase join`.

Our implementation of XQuery use as host language Prolog instead of a RDBMS. The advantage of using Prolog is that Prolog can handle the hierarchical structure of a XML document and does not need to encode the tree structure of XML documents. However RDBMS implementations take advantage from optimization techniques for RDBMSs. Since our implementation is based on the encoding into Prolog we are limited in efficiency by the interpreter.

However, our approach has the following advantages. Our aim is the development of a query language for the Semantic Web. In this context, XML documents can be handled by means of XQuery, however, other kinds of Web documents could be handled in our framework. More concretely, RDF and OWL documents. A suitable query language for such documents should include reasoning and inference capabilities. Logic programming can be used for Web reasoning. Therefore, a logic programming based implementation of XQuery would be easier integrated with rules for Web reasoning. In this line, we have already [1,2] designed extensions of our framework for representing RDF and OWL by means of rules, which can be integrated with our rule based implementation of XQuery.

The structure of the paper is as follows. Section 2 will show the representation of XML documents by means of SWI-Prolog; Section 3 will describe the implementation of XPath into Prolog; Section 4 will define the translation of XQuery expressions into Prolog rules; finally, Section 5 will conclude and present future work.
2 Loading XML Documents by means of the Prolog Library

The SWI-Prolog library for loading XML documents stores the XML documents by means of a Prolog term representing a tree. The representation of XML documents is as follows. Each tag is represented as a Prolog term of the form \texttt{element(Tag, Attributes, Subelements)} where \texttt{Tag} is the name of the XML tag, \texttt{Attributes} is a Prolog list containing the attributes, and \texttt{Subelements} is a list containing the subelements (i.e. subtrees) of the tag. For instance, let us consider the XML document called “ex.xml” of Figure 1, represented in SWI-Prolog like in the Figure 2.

For loading XML documents in our prototype we can use the predicate \texttt{load_xml(+File,-Term)} defined as follows:

\begin{verbatim}
load_xml(File,Term):-load_structure(File,Term,[dialect(sgml)]).
\end{verbatim}

where \texttt{load_structure(+File,-Term,+Options)} is the SWI-Prolog predicate of the XML library for loading \texttt{SGML} documents. Similarly, we have implemented a predicate called \texttt{write_xml(+File,+Term)} for writing Prolog terms representing a XML document into a file.

3 Implementing XPath by means of Prolog

Now, we will present how \texttt{XPath} can be implemented by means of Prolog. We restrict ourselves to \texttt{XPath} expressions of the form \texttt{/tag1 …/tagn (/text())}. More complex \texttt{XPath} queries [13] can be expressed in \texttt{XQuery}, and therefore this restriction does not reduce the expressivity power of our proposal. In Prolog, XPath expressions will be represented by means of lists of the form \texttt{[tag1, …, tagn, (text)]} in such a way that we have a predicate \texttt{load_xpath(+XPath,-ListXPath)} to transform Path expressions into the Prolog representation.

Now, the XPath language can be implemented in Prolog by means of a predicate \texttt{xpath(+ListXPath,+Tree,-Subtrees)}, where \texttt{ListXPath} is the Prolog representation of an XPath expression, \texttt{Tree} is an input XML document and \texttt{Subtrees} is a list of subtrees of the input document. Basically, the \texttt{xpath} predicate traverses the Prolog tree representing a XML document and extracts in a Prolog list the subtrees occurring in the given path. The predicate includes the following rules, distinguishing cases in the form of the input document and the XPath expression:

\begin{verbatim}
3 From now on, we will show the main rules of each predicate, a full version can be downloaded from http://indalog.ual.es/XQuery.
\end{verbatim}
XPath(
[Text],
[Tree|Trees],
[Tree|Trees2]:-atomic(Tree),!,xpath([Text],Trees,Trees2).

xpath([Text],[],Trees2):-!,xpath([Text],Trees,Trees2).

xpath([Tag],
[element(Tag,Attr,SubTrees)|Trees],
[element(Tag,Attr,SubTrees)|Trees2]):-!

xpath([Tag],Trees,Trees2).

xpath([Tag],[],Trees2):-!,xpath([Tag],Trees,Trees2).

For instance, the following goal extracts the subtrees in the path 'bib/book/title' from the document 'ex.xml', and writes them into the file 'output.xml':

?-load_xml('ex.xml',Term), load_xpath('bib/book/title',LXPath), xpath(LXPath,Term,OutputTerm), write_xml('output.xml',OutputTerm).

The previous goal generates the following sequence of items:

\[
<\text{title}>TCP/IP \text{ Illustrated} </\text{title}> \\
<\text{title}>Advanced Programming in the \text{ Unix environment} </\text{title}> \\
<\text{title}>Data on the \text{ Web} </\text{title}> \\
<\text{title}>The \text{ Economics of Technology and Content for \text{ Digital TV}} </\text{title}>
\]

4 Implementing XQuery by means of Prolog

Now, we will show how to encode XQuery in Prolog using the representation of XML documents and the previous XPath implementation. We will focus on a subset of XQuery, called XQuery core language, whose grammar can be defined as follows.

Core XQuery

\[
xquery := \text{xpath} | \text{tag} | \{xquery,...,xquery\} | \text{flwr}.
\]

\[
dxpath := \text{doc}(\text{Doc}) | '/' xpath .
\]

\[
\text{flwr} := \text{for} \$\text{var} \text{in} \text{xpath} \text{where} \text{constraint} \text{return} \text{xvar} | \text{let} \$\text{var} := \text{xpath} \text{where} \text{constraint} \text{return} \text{xvar}.
\]

\[
xvar := \text{xpath} | \text{tag} | \{xvar,...,xvar\} | \text{flwr}.
\]

\[
\text{xpath} := \text{text}() | \text{tag} | \text{xpath} | \text{xpath}.'tag'. \text{Op}:=\text{=}|\text{=}|\text{>}|\text{<} |\text{=}.
\]

\[
\text{constraint} := \text{xpath} \text{Op} \text{value} | \text{xpath} \text{Op} \text{xpath} | \text{constraint} \text{Op} \text{constraint}.
\]

In the previous definition value is an \text{string}, \text{integer}, etc, \text{Doc} is a document name, and \text{Op} is a boolean operator. The previous subset of the language allows to express the following query:

\[
\text{for} \$\text{Book} \text{in} \text{doc('ex.xml')/bib/book \text{return}} \\
\text{let} \$\text{Year} := \$\text{Book/book/@year} \text{where} \$\text{Year} < 1995 \text{return} \\
<\text{mybook}> \{\text{year} \$\text{Book/book/title}\} </\text{mybook}>
\]

Such query requests the year and title of books published before than 1995. It represents the result as a sequence of XML items whose tag is \text{mybook}. The answer of the query is:

\[
<\text{result}> \\
<\text{mybook}>1994<\text{title}>TCP/IP \text{ Illustrated} </\text{title}> </\text{mybook}> \\
<\text{mybook}>1992<\text{title}>Advanced Programming in the \text{ Unix environment} </\text{title}> </\text{mybook}> \\
</\text{result}>
\]

In order to encode XQuery into Prolog rules we have to take into account the following elements:

5
The main element of the encoding is a predicate called \texttt{xquery/1} such that \texttt{xquery} returns the XML tree representing the result of a query. For instance, the previous query is executed by means of the goal \texttt{?- xquery(Tree)} and the Prolog answer is \texttt{Tree=[element(result, [], [1994, ...])].}

In order to define \texttt{xquery/1}, we have defined a predicate \texttt{xquery/2} of the form \texttt{xquery(-Tree,+Number)} where \texttt{Tree} is a Prolog term representing a XML tree and \texttt{Number} is an identifier of the XML tree \texttt{Tree}. In order to build the hierarchical structure of the output tree, each subtree is computed by means to a call \texttt{xquery(-Subtree,+Number)}, assuming subtrees are numbered by levels. Therefore, the structure of the \texttt{xquery} rules is as follows:

\begin{verbatim}
xquery([element(tag,[],Subtrees)],Number):-xquery(Subtree,Number+1),
  xquery(SubtreeList,Number+2),
  combine([Subtree,SubtreeList],Combination),
  member(Subtrees,Combination).
\end{verbatim}

whenever the element \texttt{tag} has as subtrees in the output document the elements \texttt{Subtree} and \texttt{SubtreeList}. The root of the output tree is numbered as 1. Therefore \texttt{xquery/1} is defined as \texttt{xquery(Tree):-xquery(Tree,1)}. When a subtree is a sequence of elements the call to \texttt{xquery} is combined with a call to the Prolog predicate \texttt{findall}. For instance, \texttt{findall(Eachelement, xquery(Eachelement,Number+1),Subtree)}.

The predicates \texttt{xquery} might call to the predicates \texttt{flwr(-Tree,+Number)} which compute a \texttt{flwr} expression. \texttt{Tree} is the output of such expression, and \texttt{Number} is the identifier of the \texttt{Tree}. In general, the structure of \texttt{flwr} predicates is:

\begin{verbatim}
flwr(Tree,Number):-for_exp(Tree,path(xqueryterm,xpath)).
flwr(Tree,Number):-let_exp(Tree,path(xqueryterm,xpath)).
\end{verbatim}

The predicates \texttt{flwr} call to the predicates \texttt{for_exp} (and \texttt{let_exp}), whenever the \texttt{flwr} expression is a \texttt{for} expression and \texttt{let} expression, respectively. \texttt{xqueryterm} is (i) either a variable of the form \texttt{'$X'} or (ii) a document name of the form \texttt{doc(docname)}. \texttt{xpath} is an XPath expression. The meaning of \texttt{flwr(Tree,Number)} is that \texttt{Tree} (whose number is \texttt{Number}) is a Prolog term whose value is the result of evaluating the “pseudo-expression”:

(i) “for Tree in xqueryterm/xpath” (and “let Tree := xqueryterm/xpath”), whenever \texttt{xqueryterm} is a document name.

(ii) “for Tree in doc(docname)/pathtodoc” (and “let Tree := doc(docname) / pathtodoc”), whenever \texttt{xqueryterm} has the form \texttt{'$X'}. Where the document name associated to \texttt{'$X'} is \texttt{doc(docname)}, and the path from \texttt{'$X'} to the document name is \texttt{pathtodoc}.

Let us remark that in our core language, each variable has an associated document name, that is, each variable is used for traversing a given input document. In addition, in case (ii), analysing the \texttt{XQuery} expression, a path from the root of the document to the variable can be rebuilt.

\texttt{XQuery} expressions of the form “\texttt{doc(Doc)/xpath}” are represented in Prolog as \texttt{xpath(doc(docname)/xpath)} and \texttt{XQuery} boolean conditions “\texttt{$X$/xpath1 Op $Y$/xpath2}” are represented in Prolog as \texttt{varpath($X$,xpath1) Op varpath($Y$,xpath2)}.

Our encoding makes a previous program transformation in which \texttt{XQuery} expres-
sions including a return expression involving XPath expressions, are transformed into the so-called XPath-free return XQuery expressions, which are equivalent. It will be explained in the following example.

For instance, the previous query can be encoded as follows. Firstly, our encoding transforms the query into an equivalent XPath-free return XQuery expression:

```xml
<result>
  for $Book in doc('ex.xml')/bib/book return
  let $Year := $Book/book/@year
  where $Year < 1995 return
  let $Book1 := $Book/book/title return
  <mybook> { $Year $Book1 } </mybook>
</result>
```

where a new variable $Book1 is introduced by means of a let expression in such a way that now, the return expression does not include XPath expressions. Now, the encoding is as follows:

```
(1) xquery([element(result, [], A)], 1) :- xquery(A, 2).
(2) xquery(B, 2) :- findall(A, xquery([A], 3), B).
(3) xquery([element(mybook, [], A)], 3) :- xquery(A, 6).
(4) xquery(E, 6) :- findall(A, xquery([A], 7), C), findall(B, xquery(B, 8), D),
    combine([C, D], F), member(E, F).
(5) xquery([A], 7) :- flwr(B, 7), member(A, B).
(6) xquery([A], 8) :- flwr(B, 8), member(A, B).
(7) flwr(A, 7) :- for_var(A, path("$Year", "")).
(8) flwr(A, 8) :- for_var(A, path("$Book1", "")).
(9) for_var(B, path(A, C)) :- atomic(A), is_var(A, _), !, for_var(r, B, path(A, C)).
(10) for_var(r, A, path("$Year", C)) :- xquery(B, 5), for_exp(A, path(B, C)).
(11) for_var(r, A, path("$Book1", C)) :- xquery(B, 9), for_exp(A, path(B, C)).
(12) xquery([A], 5) :- flwr(B, 5), member(A, B).
(13) xquery([A], 9) :- flwr(B, 9), member(A, B).
(14) flwr(B, 5) :- let_exp(A, path("$Book", 'book/year')).
(15) flwr(A, 9) :- let_exp(A, path("$Book", 'book/title')).
(16) let_exp(B, path(A, C)) :- atomic(A), is_var(A, _), !, let_var(r, B, path(A, C)).
(17) let_var(r, B, path("$Book", C)) :- xquery(A, 4),
    let_exp(B, path(A, C)).
(18) xquery([A], 4) :- flwr(B, 4), member(A, B).
(19) flwr(A, 4) :- for_exp(A, path('doc('ex.xml')'), 'ib/book')).
(20) for_exp(C, path(A, D)) :- atomic(A), string_to_term(A, doc(B)), !,
    execute_term(B, C, D).
(21) execute_term(A, E, B) :- load_xml(A, D), load_xpath(B, C), xpath(C, D, E).
```

The previous encoding can be summarized as follows:

- Rule (1) is the root of the encoding. It defines the Prolog tree element(result, [], A) as the root of the output XML document, where the subtrees are computed in A by means of the rule (2).
- Rule (2) defines the subtrees of element(result, [], A). They are included in a Prolog list of trees and they are computed by means of the rule (3). The rule (3) computes the elements enclosed in the tag mybook.
- The rule (3) computes the elements element(mybook, [], A). The subelements A of element(mybook, [], A) are couples of elements (representing $Year, $Book1)
which are computed by means of the rule (4).

- The values of $\text{Year}$ and $\text{Book1}$ have to be computed by means of a \texttt{flwr} expression. Rules (5) and (6) call to the \texttt{flwr} predicate, defined by means of rules (7) and (8). Following case (ii), rules (7) and (8) represent “for $A$ in doc(‘ex.xml’)/bib/book/@year” and “for $A$ in doc(‘ex.xml’)/bib/book/title”, respectively, given that: the document name associated to $\text{Year}$ and $\text{Book1}$ is doc(‘ex.xml’); the path from $\text{Year}$ to doc(‘ex.xml’) is ’/bib/book/@year’, and the path from $\text{Book1}$ to doc(‘ex.xml’) is ’/bib/book/title’.

- With the aim to obtain the previous behaviour the \texttt{for_exp} predicate (in rules (7) and (8)) calls by means of the rules (9), (10), (11), (12) and (13) to \texttt{let_exp} predicate in rules (14) and (15). Rules (14) and (15) compute the value of the \texttt{let} expressions in which $\text{Year}$ and $\text{Book1}$ are involved.

- Rules (14) and (15) call to the rule (16), which in its turn calls to the rule (17). Rule (17) calls to rules (18) and (19) in order to compute the main \texttt{for} expression. Following (i) of previous description, rule (19) represents “for $A$ in doc(‘ex.xml’)/bib/book” which is the main \texttt{flwr} expression of the query. Moreover, rule (17) checks the boolean condition of the \texttt{XQuery} expression, that is, “$\text{Year} < 1995$”.

- Finally, rules (20) and (21) compute the main \texttt{for} expression by means of the \texttt{xpath} predicate, defined in previous section.

5 Conclusions and Future Work

In this paper, we have studied how to encode \texttt{XQuery} expressions into Prolog. It allows us to evaluate \texttt{XQuery} expressions against XML documents using logic rules. We have developed a prototype available in \texttt{http://indalog.ual.es/XQuery}. The distribution includes a package of examples of \texttt{XQuery} expressions which has been tested with our prototype. As future work we would like to extend our prototype for reasoning with RDF/OWL in \texttt{XQuery}. The theoretical background of such extension has been studied in [1,2].

References


