Translation of XML documents into logic programs

Martin Zima; Karel Jezek

Department of Computer Science & Engineering
University of West Bohemia in Pilsen
Universitni 8, 306 14 Pilsen, Czech Republic
{zima, jezek_ka}@kiv.zcu.cz

Abstract

The semantic web is supposed to become a characteristic phenomenon of the worldwide web in the next decade. One of the basic semantic web tools is the XML language. The aim of this paper is to provide information on how web documents written in the XML language can be rewritten into logic forms expressed as Prolog/Datalog programs. The XML language constitutes the basis of many semantic web languages and information in XML documents is usually retrieved with the help of procedural language called XQuery. Retrieving based on logic formulas gives us the chance to take advantage of deduction and this way to gain new, originally hidden information.

Keywords: semantic web; logic programming; XML; Datalog language.

1. Introduction

An interesting and topical research issue is the use of logic rules (logic program) to evaluate a query about XML documents [1]. It provides an option to combine XML technology with the inference capabilities of logic programming. Logic programming allows us to evaluate the queries that require computation of a transitive closure of relations. This means that we can query such information that is not explicitly included in the documents. In other words, we are able to draw deductive conclusions concerning the facts contained in the documents and in this way to find new, but originally hidden facts. Suppose, for example, that we have an XML collection of scientific articles. The logic program allows a query such as: *find the names of all co-authors of the given author, including co-authors of found co-authors, etc.* It means this query evaluates the transitive closure of the relation co-author. There are many similar tasks having something to do with transitivity of relations, e.g. looking for a path from one place to some destination, searching for owners of a given company (as frequently real final owners are hidden behind the companies that transitively own other companies), etc. All compound queries have the form of logic formulas. Therefore it is natural to use the logic query languages and, consequently, transform XML documents into expressions written in the logic language. This task is particularly interesting for us, as a few years ago we implemented an experimental deductive system [2], translating the Datalog language into the PL/SQL procedural language [3], which is used in the DBMS Oracle.

The use of Datalog instead of Prolog has some pragmatic reasons too. A database-oriented system with a professional database management system is able to process large data collections within a reasonable time. This is particularly important in the case of web documents processing.

2. Logic programming and Datalog

To be able to explain the method of translation, let us briefly introduce some principles of logic programming and its form used in the language Datalog [4]. Datalog is a slightly modified version of the primary logic language Prolog [5] and is tailored to database processing.

The logic program consists of a set of facts, a set of logic rules and a query. On the basis of the logic program execution, a set of new facts can be inferred and delivered as the query result. Every logic program can contain constants and variables. The names of variables begin with upper case letters. The exceptions are anonymous variables, i.e. variables whose values we are not interested in. They are marked with an underscore. Names of predicates begin with lower case letters and predicates are distinguished by the number of their arguments as well. Facts have the common form

```
predicate_name(list of constant arguments).
```

Rules have the common form `head :- body`. The symbol "`:-`" means "if" and expresses an implication between the truth of the body and head predicates. The head is a predicate name, the arguments of which are mostly variables. Such variables are evaluated during the program execution and in case the head predicate is TRUE, they are returned as the result of the rule processing. The rule body consists of predicates whose arguments have to contain all variables from the rule head. The head predicate becomes TRUE if there exist such values of variables in the logic program that the values of all predicates in the rule body are TRUE too. If such values of variables do not exist, the rule is evaluated as FALSE. The query consists of a predicate whose arguments are variables or constants. The deductive system tries to find values of query variables which are derivable from existing program facts with the possible use of program rules. The query succeeds if such values exist, otherwise, the query predicate has the value FALSE and the query answer is NO.

Our experimental deductive system implements an extended version of the Datalog language. The current extensions include relational operators, assignment and not operation for negation.

3. Translation of XML documents

Advantage of logic programming for XML documents querying and processing has inspired other researchers too. To our knowledge, a similar issue was described by Jesús M. Almendros-Jiménez [6]. He proposed a possible solution to the problem of converting XML documents to a logic program written in Prolog. His solution uses a list structure which describes data of the XML document and represents the result of the XPath query [7]. The rules of the logic program define the structure of the XML document (the way the elements are nested within other elements). He introduces specific functions with a different number of arguments to specify the XML documents structure, but the process of functions evaluations is missing. The resulting logic program contains all data of the input XML document in a set of facts. The structure of the XML document is fixed in logical rules. This means that each XML document is transformed into a different set of rules.

On the contrary the method proposed by us generates a logic program which consists of universal rules. That is, two different logic programs (results of the transformation of two different XML documents) contain the same logic rules. Differences between the structures of various XML documents are captured by the facts. This technique also eliminates the need to work with lists that our implementation of Datalog still lacks.

3.1. Construction of the set of facts

To show our method of generating a set of logic facts, we need to choose some example of an input XML document. Such suitable candidate is e.g. books.xml, a modified XML document describing a library content, which was adopted from [6]. The text of the XML document, accompanied by a number of lines, is shown in Fig. 1.

```
 1 <?xml version="1.0" ?>
 2 <bookshelf>
 3   <book year="2003">
 4     <author>Abiteboul</author>
 5     <author>Buneman</author>
 6     <author>Suciu</author>
 7     <title>Data on the Web</title>
 8     <review>A fine book.</review>
 9   </book>
10   <book year="2002">
11     <author>Buneman</author>
12     <title>XML in Scotland</title>
13     <review>The best ever!</review>
14   </book>
15 </bookshelf>
```

Figure 1: XML document books.xml

As [6] shows, the information *"Buneman is the author of the first book"* describe this fact:

```
author(•Buneman•, [2, 1, 1], 3, •books.xml•).
```

- The first argument is the value of the element `<author>`.
- The second argument defines the XML document structure: Number 2 means: the element is the second one inside another element. The first 1 means that the element `<book>` is the first one inside another element `<bookshelf>`. The second 1 stands for the element `<bookshelf>` which is the root element of the document.
- The remaining arguments do not require further explanation.

Before explaining the above-presented transformations, let us modify the sample XML document so it is consistent with the standards of the Semantic Web. An adapted version of the XML document is shown in Fig. 2. The above-mentioned information *"Buneman is the author of the first book"*, will be written into a triple of auxiliary logic facts that define the predicate `xml`. In this way we simultaneously eliminate the need to use the data structure list.

```
xml(5, •dc:creator•, 2, 3).
xml(5, •bk:book•, 1, 2).
xml(5, •bk:bookshelf•, 1, 1).
```

The predicate `xml` mostly shows only the structure of the document. The first argument of the `xml` predicate holds the row number of the input XML document. This value will be the same for all logic facts defining the predicate `xml`, i.e. facts which describe a specific occurrence of the element written in a given row of the XML document. The other three arguments define the path in the XML document, i.e. the path from the given element to the root element of the document. To be specific, on line 5 there is recorded the element `<dc:creator>`, whose parental element is `<bk:book>`. The element `<bk:book>` contains a total of 5 children, of which the second child element is `<dc:creator>` located on line 5. The element `<bk:book>` has a parental element `<bk:bookshelf>`. This element contains 2 children (books). The element `<dc:creator>` from line 5 is contained in the first element `<bk:book>`. The last of the three facts says that the described `<bk:bookshelf>` element is the root element of the document.

```
 1 <?xml version="1.0" ?>
 2 <bk:bookshelf
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:bk="http://example.org/books/">
 3   <bk:book year="2003">
 4     <dc:creator>Abiteboul</dc:creator>
 5     <dc:creator>Buneman</dc:creator>
 6     <dc:creator>Suciu</dc:creator>
 7     <dc:title>Data on the Web</dc:title>
 8     <bk:review>A fine book.</bk:review>
 9   </bk:book>
10   <bk:book year="2002">
11     <dc:creator>Buneman</dc:creator>
12     <dc:title>XML in Scotland</dc:title>
13     <bk:review>The best ever!</bk:review>
14   </bk:book>
15 </bk:bookshelf>
```

Figure 2: Adapted XML document

```
xml(2, 'bk:bookshelf', 1, 1).
xml(3, 'bk:book', 1, 2).
xml(3, 'bk:bookshelf', 1, 1).
xml(4, 'dc:creator', 1, 3).
xml(4, 'bk:book', 1, 2).
xml(4, 'bk:bookshelf', 1, 1).
xml(5, 'dc:creator', 2, 3).
xml(5, 'bk:book', 1, 2).
xml(5, 'bk:bookshelf', 1, 1).
xml(6, 'dc:creator', 3, 3).
xml(6, 'bk:book', 1, 2).
xml(6, 'bk:bookshelf', 1, 1).
xml(7, 'dc:title', 4, 3).
xml(7, 'bk:book', 1, 2).
xml(7, 'bk:bookshelf', 1, 1).
xml(8, 'bk:review', 5, 3).
xml(8, 'bk:book', 1, 2).
xml(8, 'bk:bookshelf', 1, 1).
xml(10, 'bk:book', 2, 2).
xml(10, 'bk:bookshelf', 1, 1).
xml(11, 'dc:creator', 1, 3).
xml(11, 'bk:book', 2, 2).
xml(11, 'bk:bookshelf', 1, 1).
xml(12, 'dc:title', 4, 3).
xml(12, 'bk:book', 2, 2).
xml(12, 'bk:bookshelf', 1, 1).
xml(13, 'bk:review', 5, 3).
xml(13, 'bk:book', 2, 2).
xml(13, 'bk:bookshelf', 1, 1).

data(4, 'dc:creator', 'Abiteboul').
data(5, 'dc:creator', 'Buneman').
data(6, 'dc:creator', 'Suciu').
data(7, 'dc:title', 'Data on the Web').
data(8, 'bk:review', 'A fine book.').
data(11, 'dc:creator', 'Buneman').
data(12, 'dc:title', 'XML in Scotland').
data(13, 'bk:review', 'The best ever!').

attribute(3, 'bk:book', 'year', '2003').
attribute(10, 'bk:book', 'year', '2002').
```

Figure 3: Generated facts

The data itself (names and attributes of elements, text content, etc.) has to be saved through other predicates. Let us introduce for this purpose a piece of predicate `data` which will contain the line number, the element name that contains the relevant text information and the value, i.e. the text content of this element. The fact that *"Buneman is the author of the first book"* is expressed as follows:

```
data(5, •dc:creator•, •Buneman•).
```

Attributes and values will also be recorded in the form of facts. We can introduce the predicate attribute containing the line number, the element name by which an attribute is defined, the attribute name and finally its value. For example, as our document defines only one attribute year within the element <bk:book>, the corresponding fact will have the form:

```
attribute(3, •bk:book•, •year•, •2003•).
```

Fig. 3 shows the full set of facts defining the predicates xml, data and attribute, which were obtained by translation from the XML document in Fig. 2.

3.2. Universal rules

The logic facts defining the predicate xml hold the structure of the input XML document. Therefore, it is possible to determine which element is a part (descendant) of another element. As this process is recursive we can define an auxiliary predicate intersection. This predicate is defined by two rules, which look for the intersection of sets of facts determining the predicate xml. Each set describes an element from one line of the XML document. The set of logic facts from Fig. 3 defines 11 various sets in total. To demonstrate, we selected two sets which describe the elements listed in lines 3 and 6.

```
xml(3, 'bk:book', 1, 2).
xml(3, 'bk:bookshelf', 1, 1).

xml(6, 'dc:creator', 3, 3).
xml(6, 'bk:book', 1, 2).
xml(6, 'bk:bookshelf', 1, 1).
```

At first sight it is clear that both sets have three common arguments: 'bk:bookshelf', 1, 1. They describe the root element of the XML document, which must be specified in any set of facts defining the predicate xml. The form of rules defining the predicate intersection is as follows:

```
intersection(Line1, Line2, Element, N, 1) :-
  xml(Line1, Element, N, 1),
  xml(Line2, Element, N, 1),
  Line1 < Line2.
```

The above-mentioned sets also have another three common arguments: `'bk:book', 1, 2`. The value 2 defines the level of nesting of the element, the value 1 indicates the root element. To ensure that these two facts will also be included in the intersection, the intersection must include the fact defined at a lower level. This results in the following recursive rule:

```
intersection(Line1, Line2, Element, N, P2) :-
  xml(Line1, Element, N, P2),
  xml(Line2, Element, N, P2),
  P1 := P2 - 1,
  intersection(Line1, Line2, _, _, P1).
```

The pairs of line numbers, which are the results of an evaluation of the predicate `intersection`, do not guarantee so far that on these XML lines there are written two immediately nesting elements, e.g., that on `Line2` there is written such element, whose parent is written on `Line1`. This condition applies only in the following case. With `Line1` there is associated such a set which is identical to the intersection of both sets and the second set, associated with `Line2`, contains one more fact. This condition is true for elements on lines 3 and 6 of the XML document, but it is false in the case of elements on line 4 and 7. The following rule describes this condition.

```
child_lines(Line1, Line2) :-
  intersection(Line1, Line2, _, _, P1),
  P2 := P1 + 1,
  not xml(Line1, _, _, P2),
  xml(Line2, _, _, P2),
  P3 := P2 + 1,
  not xml(Line2, _, _, P3).
```

XML documents often contain deeply nested elements. We call nested elements the descendants of a surrounding element. The nesting has to be verified. The following recursive predicate will do this activity.

```
descendant_lines(Line1, Line2) :-
  child_lines(Line1, Line2).

descendant_lines(Line1, Line3) :-
  child_lines(Line1, Line2),
  descendant(Line2, Line3).
```

To make our list of universal rules complete, we have to add a rule detecting which element is on the specified line. This rule must be used in case of elements without any attributes, e.g. the element `<bookshelf>` (see Fig. 1). The rule `element_line` looks for the specified number of such line element (in the set of predicates `xml`) which has the greatest value of the last argument, i.e. the level of nesting.

```
element_line(Line, Element) :-
  xml(Line, Element, _, P1),
  P2 := P1 + 1,
  not xml(Line, _, _, P2).
```

All listed and described logic rules are universal. If we use the proposed transformation on two different XML documents, the resulting logic programs will contain different set of facts, but the logic rules will be the same.

4. Queries

The logic program is complete if it contains a query we want to evaluate. Datalog has the basic form of a query:

```
?- predicate(list of arguments).
```

For the formulation of a query, it is usually necessary to define additional predicates in the form of one or more logic rules. This approach is used for all queries listed below.

The first query looks for the names of all authors participating in books published in 2003. The rule defining the predicate `authors_2003` and its corresponding query are as follows:

```
authors_2003(Author) :-
  attribute(Line1, 'bk:book', 'bk:year', '2003'),
  data(Line2, 'dc:creator', Author),
  descendant_lines(Line1, Line2).

?- authors(Author).
```

The rule looks for all lines (see variable `Line1`) where books issued in 2003 are recorded and all lines (see variable `Line2`) where all authors existing in the relevant facts of the predicate `data` are recorded. The predicate `descendant_lines` searches only pairs values of variables `Line1` and `Line2`, which satisfy the condition that the element written on `Line2` is a descendant of the element written on `Line1`. So the predicate searches only for the names of the authors of those books that were issued in 2003.

The last query is more complicated, working with several auxiliary rules. It searches for all co-authors of a given author (e.g. Abiteboul), including all co-authors of the searched co-authors, etc. This means it is looking for the transitive closure of a co-authorship relation (for the connected component of the co-authorship graph). The core of the recursive evaluation is given in a simplified form:

```
coauthors(New_coauthor) :-
  coauthors(Coauthor),
  search_book(Coauthor, Book),
  serarch_new_coautor(Book, New_coauthor).

?- coauthors(Coauthor).
```

The predicate `coauthors` will bind the variable `Coauthor` to the name of the previously found co-author. The predicate `search_books` finds out such books in which the `Coauthor` participated with other co-authors. The predicate `search_new_coauthor` evaluates the names of these co-authors (the value of the variable `New_coauthor`). This recursive evaluation stops when no other co-authors are found.

5. Conclusions

This paper shows one possible transformation of an arbitrary XML document into a logic program. The advantage of our transformation is the use of universal rules. These rules are the same in all generated logic programs. The programs differ only in facts.

The shortcoming of the proposed procedure is the assumption that the elements in XML documents will not have mixed content. This means that the element will contain either text or other nested elements. For example, the element `<review>A <em>fine</em> book.</review>` has a mixed content. It contains both text as well as a nested element `<em>`. Such information cannot be recorded into facts. If the proposed procedure transforms the RDF, RDFS or OWL document written in XML syntax, the elements with mixed content are not occurred.

In the future, we suppose to extend the set of universal rules and add rules which simplify the queries formulation.

Acknowledgements

This work was partly supported by Ministry of Education, Youth and Sports of the Czech Republic – the project 2C06009 *Knowledge base tools for natural language communication with semantic web*.

Notes and References

[1] W3 CONSORTIUM. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, November 2008, `http://www.w3.org/TR/xml/`.

[2] ZIMA, M. *Experimental Deductive Database System with Uncertainty [in Czech]*, Ph.D. Thesis, University of West Bohemia in Pilsen, 2002.

[3] ORACLE CORPORATION. *Oracle Database PL/SQL User's Guide and Reference 10g Release 2 (10.2)*, June 2005, Available at `http://www.oracle.com/`.

[4] CERI, S; GOTTLOB, G; TANCA, T. *What you always wanted to know about Datalog (and never dared to ask)*, IEEE Transactions on Knowledge and Data Engineering 1(1), March 1989, p. 146-66.

[5] STERLING, L; SAPIRO, E. *The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming)*, The MIT Press, March 1994.

[6] ALMENDROS-JIMENEZ, J.M. *An RDF Query Language based on Logic Programming*, Electronic Notes in Theoretical Science, 200, 2008, p. 67-85.

[7] W3 CONSORTIUM. *XML Path Language (XPath) 2.0*, January 2007, `http://www.w3.org/TR/xpath20/`.

[8] W3 CONSORTIUM. *RDF/XML Syntax Specification (Revisited)*, February 2004, `http://www.w3.org/TR/rdf-syntax-grammar/`.