Eliminating Dead-Code from XQuery Programs

Pierre Genevès
CNRS
pierre.geneves@inria.fr

Nabil Layaïda
INRIA
nabil.layaida@inria.fr

ABSTRACT

One of the challenges in web software development is to help achieving a good level of quality in terms of code size and runtime performance, for increasingly popular domain specific languages such as XQuery. We present an IDE equipped with static analysis features for assisting the programmer. These features are capable of identifying and eliminating dead code automatically. The tool is based on newly developed formal programming language verification techniques [4, 3], which are now mature enough to be introduced in the process of software development.

1. INTRODUCTION

One major difficulty for performing dead-code analysis for XQuery [1] actually comes from XPath expressions, for which analysis techniques are known to be very complex from a computational point of view. We build on our previous work on static analysis techniques for XPath expressions [4, 2], and propose a technique for performing basic dead-code analysis and elimination from an XQuery program. Removing such dead code has two benefits: first, it shrinks program size, which is an important consideration from a software engineering perspective, and second, it lets the running program avoid executing irrelevant operations, which reduces its running time.

2. XQUERY PROGRAMS

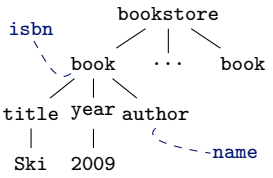
XQuery programs operate on XML documents that are considered as trees of element and attribute nodes. An XQuery program is usually written with respect to a *schema* that defines constraints that a particular set of documents should verify (as, e.g., XHTML for web pages). A schema defines the set of admissible elements and attributes in a XML document, as well as how they can be assembled together. This definition is usually done with regular expressions. For example, a simplistic schema for a bookstore (using DTD notation) follows:

ICSE '10, May 2-8 2010, Cape Town, South Africa


```
<!ELEMENT bookstore (book*)>
<!ELEMENT book (title, year, author+)
<!ATTLIST book isbn CDATA #REQUIRED>
 ...
```

This states that a `bookstore` element has any number of `book` elements as children. In turn, each `book` element must have a `title` child, followed by a `year` element and one or more `author` elements. Finally, each `book` element must carry an `isbn` attribute. The sample document shown below is valid with respect to this (partial) schema definition.

```
<bookstore>
 <book isbn="␣">
  <title>Ski</title>
  <year>2009</year>
  <author name="␣"/>
 </book>
 ...
 <book/>
</bookstore>
```



An XQuery program basically takes one (or possibly several) XML document as input, performs some computation based on its tree view, and finally outputs a result in the form of another XML document. The core of the XQuery language is composed of XPath expressions that make it possible to navigate in the document tree and extract nodes that satisfy some conditions. For instance, a simplistic XQuery program is:

```
<ul>
 {
  for $x in /descendant::book return
    if $x/year>2008 then <li>$x/title<li> else ()
 }
</ul>
```

where the `for` loop uses the XPath expression `/descendant::book` that traverses the whole input XML document looking for `book` elements. The `for` loop iterates over all these elements, and for each of them, returns the value of the `title` subelement, provided the year is greater than 2008. Executing this program produces an XML tree as output, whose root element is named "`ul`", and whose content is populated by the execution of the loop, that creates an XHTML-like list of book titles published after 2008.

3. DEAD-CODE ANALYSIS

We present a static analysis of XQuery programs in order to automatically detect and eliminate dead code. Our

analysis is sound and complete over the XPath navigational fragment of [4]. In order for the analysis to scale to programs with more complex features, we make several conservative approximations. First, we abstract over XPath features that make satisfiability undecidable (such as data value comparisons). Second, we consider that XPath expressions return sets of nodes (as in XPath 1.0) instead of node sequences (as in XPath 2.0 and XQuery).

These approximations preserve soundness of our approach (if dead code is detected, it can be safely eliminated as this is really dead code). However, the analysis may be incomplete due to undecidable features, that may prevent from finding some evil dead code.

In order to illustrate the practical relevance of our approach, consider the following XQuery program:

```
<para>
  {
    for $x in //body//switch
    where $x/animateMotion
    return $x/*
  }
</para>
```

It is intended to be evaluated over SMIL[1] documents. Specifically, it has been written against the schema defining SMIL 1.0 documents. When applied to such a document, it returns all children of `switch` elements that have at least one `animateMotion` child, wrapped in a `para` element.

This code portion may be reused in the context of SMIL 2.0 documents. However, in contrast to SMIL 1.0, the occurrence of `animateMotion` is not permitted as a chid of `switch` in SMIL 2.0. In this case, the XPath expression in the `where` clause is unsatisfiable, and therefore the whole `for` loop is dead code. We explain how we make this static analysis automatic for a given XQuery program and a given schema in the next subsections.

3.1 Path-Error Detection

We consider a given XQuery program P and a schema S that describes constraints over the set of documents that can serve as input to P. For each XPath expression occurring in P, we are interested in checking whether it is meaningful or not with respect to the constraints described in S. It may happen that the navigational information contained in a given path contradicts the constraints described in S. In that case, the path will always return an empty sequence of nodes no matter what the actual document instance valid for S is. In that case, we know statically that there is no need to evaluate the path at runtime. Furthermore, we also know that all XQuery instructions that depend on this path (dead code) may be removed.

For performing this path-error analysis, taking into account the schema S, we use the logical solver developed in [4]. The solver takes a given path and a schema, translates them into a logical representation and uses a logical satisfiability-checking algorithm that determines the existence (or inexistence) of a tree (a document) that satisfies both the constraints expressed by the schema and the structural requirements assumed by the path.

[1]SMIL is the standard language for expressing synchronized multimedia documents as found in e.g., MMS mobile phone messages, and more generally on the web.

3.2 Static Code Refactoring and Highlighting

Each path which is found unsatisfiable indicates dead code. We perform a code dependency analysis that propagates this information in order to detect and eliminate dead code from an XQuery program.

The typical integrated development environment allows one to open an XQuery program and to associate with it a schema. A variety of schema languages are actually supported including DTDs, XML Schemas and Relax NG definitions (see [4] for details). The code analysis process is made of several steps. First, the program is parsed to build an abstract syntax tree. The abstract syntax tree (AST) analysis phase consists in extracting all the path expressions from the program and checking their satisfiability individually. Then, in a second step, these paths are combined with the schema, and checked again for satisfiability. Indeed, there are two kinds of unsatisfiable paths: self-contradicting paths, and unsatisfiable paths due to the schema. A trivial example of a self-contradicting path is the following:

```
child::a/child::b[parent::c]
```

Beyond this trivial example, XPath errors are often introduced due to the expressive power of XPath for expressing forward, backward and recursive navigation in trees. Each kind of unsatisfiable path is marked differently in the AST. This makes it possible to inform the programmer, by underlining the empty path expressions in a different color depending on the origin of the unsatisfibility (self-contradictory or unsatisfiable with the given schema). More specifically, each path is considered as a sequence of basic navigation steps possibly with qualifiers. The first step is analyzed. Then each additional step is successively appended to this initial step and the resulting path is analyzed in turn. This makes it possible to identify precisely where the error has been introduced in the path. For instance, in the previous example, this step by step subpath analysis identifies the qualifier `parent::c` as causing the error.

Whenever an unsatisfiable path is found, a refactoring command is provided to the IDE user. When this command is triggered, the AST is pruned using the rules presented earlier, and the new XQuery program is provided to the user.

4. CONCLUSION

We have presented a feature that allows an IDE to automatically identify and eliminate of dead code from XQuery programs. The tool integrates, for the first time, the support of formally proven properties on types and path expressions in order to assist programmers writing and updating XQuery code against complex XML schema evolutions.

5. REFERENCES

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C recommendation, January 2007.

[2] P. Genevès. *Logics for XML: Reasoning with Trees.* ISBN 3639193717, VDM Verlag, September 2009.

[3] P. Genevès, N. Layaïda, and V. Quint. Identifying query incompatibilities with evolving XML schemas. In *ICFP'09*.

[4] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI'07*.