XQuery Formal Semantics State and Challenges

Peter Fankhauser
Integrated Publication and Information Systems Institute
Dolivostraße 15
D-64293 Darmstadt, Germany
fankhaus@darmstadt.gmd.de

ABSTRACT

The XQuery formalization is an ongoing effort of the W3C XML Query working group to define a precise formal semantics for XQuery. This paper briefly introduces the current state of the formalization and discusses some of the more demanding remaining challenges in formally describing an expressive query language for XML.

General Terms

Standardization, Languages

Keywords

XML, databases, query languages, formal semantics

1. INTRODUCTION

XML has become the widely adopted standard to represent semi-structured data. Building on the wealth of approaches to querying semi-structured data [1], the W3C XML Query Working Group has started in September 99 to design a standardized query language for XML, now coined *XQuery* [2]. Part of this effort is the XQuery Formal Semantics document [4]. This document sets out *XQuery Core* as a relatively small but fully expressive sub language of XQuery [3], and provides a *static semantics* by means of type inference rules and a *dynamic semantics* by means of value inference rules, which map expressions of the core language to simple operations on the XQuery Data Model [6]. In addition, it defines a mapping of XQuery to the core language and thus a precise dynamic and static semantics for the complete language.

This paper briefly introduces the overall approach of XQuery Core, the underlying processing model, and its relationship to XQuery. It then discusses some of the more demanding remaining challenges for querying XML.

2. XQUERY CORE

2.1 Overall Approach

XQuery Core is a functional language based on the algebra for XML Query introduced in [8,7]. Since its initial release, it has been syntactically adapted to XQuery and better aligned with the XML family of standards. Furthermore, it has been extended with some additional features such as support for unordered sequences and more precisely typed recursive navigation.

Four main design principles have guided its design:

Closure: Both input and output of a query expression are fragments of XML documents, which are represented as sequences of nodes and/or values in the XQuery Data Model. This has some subtle consequences. Because the primary structure of XML is a tree, elements may not have multiple parents and need to be copied to become children of a constructed result element. Furthermore, node sequences cannot be nested, because XML only supports nesting via explicit markup.

Compositionality: Operators can be arbitrarily composed both semantically and syntactically. There are no side effects; all operators are exclusively defined by their input and output. For example, there exists no primitive operator that hides edges or paths of an input tree, because the input and output of such an operator cannot be described in terms of node sequences.

Correctness: XQuery Core is a statically typed language, which supports both, inferring an output schema from a query and its input schema, and statically checking the output type of a query against a given output schema. XQuery Core is also rather rigid with respect to types. It does not perform any implicit coercions or iterations, which are heavily used in XPath 1.0, the now widely adopted filtering and addressing language for XML. In XQuery Core it is a static error to compare an element node with a value, or to apply an operator that expects a single node to a node sequence. This rigidity keeps the formal specification of static and dynamic semantics at a manageable size, and

provides a framework for static optimizations, such as avoiding costly iteration when not necessary. However, as exemplified in Section 2.3, convenience features of the XQuery surface language and its sub-language XPath, e.g., implicit existential quantification of predicates, can be realized by appropriate mappings to XQuery Core.

Completeness: XQuery Core supports the equivalent of selection, projection, and set operators, and thus is arguably relationally complete. Furthermore, for every constructor (element, attribute, and sequence) there exists some form of deconstruction. Together this allows expressing all query classes that have been identified in Dave Maier's database desiderata for querying XML [13] and in the functional requirements of the XML Query Working Group [3]. Completeness is certainly a mixed blessing for a query language. In particular, XQuery Core also supports user defined recursive functions, which are difficult to optimize and to type generically. Nevertheless, specific recursive functions, such as the widely used recursive XPath-axis `descendants-or-self`, are supported as a built-in function with specific type rules.

2.2 XQuery Processing Model

The three main constituents of the XQuery formalization are static semantics, dynamic semantics, and a mapping between XQuery and XQuery Core. In this section, their role for XQuery is described along the (slightly idealized) processing model in Figure 1.

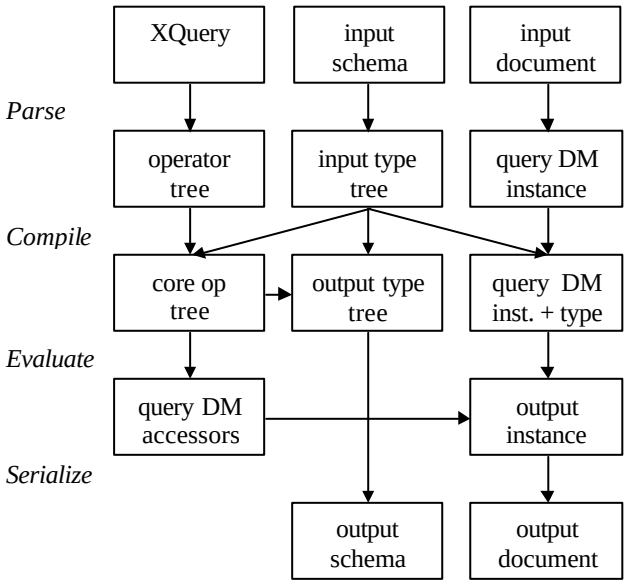

Figure 1: XQuery Processing Model

Query processing involves four steps; the XQuery formalization is mainly concerned about the second and the third step.

Parse checks the query, schema, and document syntactically, and generates an operator tree, a type tree, and an instance of the query data model. The type tree may be represented by means of the component model of XML Schema [14]; the data model instance may be implemented by means of an augmented DOM (Document Object Model) or some (object-) relational instance.

Compile validates the data model instance against the input schema and annotates it with type information. Furthermore, using the *mapping rules* it translates the operator tree to an operator tree of XQuery Core. As exemplified in Section 2.3, input type information may be used to simplify the resulting operator tree. The *static semantics* describes how the output type is inferred from the core operator tree and the input type.
Of course an actual implementation will not necessarily use the core operator tree for evaluation, but further optimize it using standard relational optimization techniques. Furthermore, e.g. for path expressions, it will not use XQuery Core at all, but translate them directly from XQuery to specialized operators.

Evaluate processes the query by using the accessors and constructors defined in the XQuery Data Model, and produces a typed output instance. This is described by the *dynamic semantics*. Of course this is also a massive idealization; any actual implementation will recur to the operators and data structures of an efficient physical algebra instead. But also there, the idealized mapping described by the dynamic semantics can provide a guideline to implement more complex mappings to a physical algebra.

Finally, *Serialize* generates an XML document or fragment from the output instance, and an output schema from the output type tree.

2.3 XQuery vs. XQuery Core

The documents of the XML Query Working Group [1,3,4] introduce numerous example queries, which cannot be reasonably sampled with the given space limitations. Thus, a different track is pursued here: A simple join query expressed in XQuery's surface syntax is mapped step by step to XQuery Core, emphasizing the role of XQuery Core's type system.

Here are two simple elements. The type of the first element, `<bib>`, which is declared after ":", describes a sequence of `<book>`s, consisting of an attribute `year`, exactly one `<title>`, and one or more `<author>`s.

```
<bib>
 <book year="1999">
   <title>Data on the Web</title>
   <author>Abiteboul</author>
   <author>Buneman</author>
```

```
   <author>Suciu</author>
  </book>
  <book>
   <book year="2001">
   <title>XML Query</title>
   <author>Fernandez</title>
   <author>Suciu</title>
  </book>
</bib>:
ELEMENT bib {
 ELEMENT book {
  ATTRIBUTE year {Integer},
  ELEMENT title {String},
  ELEMENT author {String}+
 }*}
```

The type of the second element, <reviews>, describes a sequence of <book>s containing exactly one <title> and exactly one <review>.

```
<reviews>
 <book>
   <title>XML Query</title>
   <review>A darn fine book</review>
 </book>
 <book>
 <book>
  <title>Data on the Web</title>
  <review>This is great!</review>
 </book>
</reviews>:
ELEMENT reviews {
 ELEMENT book {
  ELEMENT title {string},
  ELEMENT review {string}
 }*}
```

With $bib0 bound to <bib> and $rev0 bound to <reviews> the following XQuery expression joins <book>s with <review>s and returns <book>s with their <title>, <author>s and <review>.

```
FOR $b IN $bib0/book, $r IN $rev0/book
WHERE $b/title = $r/title
RETURN
 <book>
  {$b/title, $b/author, $r/review}
 </book>
=>
<book>
 <title>Data on the Web</title>
 <author>Abiteboul</author>
 <author>Buneman</author>
 <author>Suciu</author>
 <review>A darn fine book</review>
<book>
<book>
 <title>XML Query</title>
 <author>Fernandez</author>
 <author>Suciu</author>
 <review>This is great!</review>
</book>:
ELEMENT book {
 ELEMENT title {string},
 ELEMENT author {string}+,
 ELEMENT review {string}
 }*
```

The inferred result type tells us that there are zero or more <book>s, each with exactly one <title>, one or more <author>s, and exactly one <review>.

This simple query translates to a quite complex expression in XQuery Core. First, consider the translation of the path expression $bib0/book, denoted by [[$bib0/book]].

```
FOR $v1 IN $bib0 RETURN
 FOR $v2 IN NODES($v1) RETURN
  TYPESWITCH ($v2) AS $v3
   CASE ELEMENT book {ANYTYPE}
           RETURN $v3
   DEFAULT RETURN ()
```

XQuery Core does not directly support path expressions of the form $v/name1/…/namen. Instead, each $bib0/book is translated into a nested for-loop. This loop first iterates over all nodes $v1 in $bib0, then over all attribute and child nodes $v2 in $v1 to match their type against the type ELEMENT book {AnyType}. This is expressed by the TYPESWITCH operator, which returns the node in $v2 if its dynamic type is subsumed by the type declared in the CASE statement, and returns the empty list "()" otherwise. With the given input type of $bib0, however, this complex expression can be significantly simplified *statically* using standard techniques: First, $bib0 is known to contain only one <bib>-element, therefore the outer for-loop can be discarded. This leads to:

```
FOR $v2 IN NODES($bib0) RETURN (…)
```

Second, $bib0 is known to only contain <book>-elements, thus the DEFAULT-case in TYPESWITCH can be discarded as well. Because only one CASE remains, the entire TYPESWITCH can be simplified, leading to

```
FOR $v2 IN NODES($bib0) RETURN $v2
```

which in turn simplifies to NODES($bib0). Similarly, with the given input type $rev0/book translates to NODES($rev0).

From XPath 1.0, XQuery inherits the implicit existential quantification of predicates. For example, the simple comparison `$b/title = $r/title` evaluates to `true` if there exists some title in `$b` whose content is equal to the content of some title in `$r`. Consequently, the comparison translates to a rather baroque expression, where `[[$b/title]]` is translated along the lines described above.

```
NOT(EMPTY(
     FOR $v1 IN [[$b/title]] RETURN
     FOR $v2 IN [[$r/review]] RETURN
       IF EQ($v1,$v2) THEN $v1 ELSE ()))
```

Because one can statically determine that each `$b` and each `$r` has exactly one `<title>`-element, the existential quantification can be removed, leading to:

```
NOT(EMPTY(
  IF EQ([[$b/title]],[[$r/review]])
  THEN [[$b/title]] ELSE ())
```

This can be further simplified to:

```
EQ([[$b/title]], [[$r/review]])
```

Finally, the FOR-clause which binds both `$b` and `$r` needs to be translated into a nested loop, binding first `$b` and then `$r`, and the WHERE-clause needs to be translated into an IF-THEN-ELSE expression. Together this leads to the final translation:

```
FOR $b IN NODES($bib0) RETURN
FOR $r IN NODES($review0) RETURN
IF (EQ[[$b/title]], [[$r/review]]) THEN
ELEMENT book {
  [[$b/title, $b/author, $r/review]]}
```

Both, the translation of XQuery to XQuery Core, and the type inference discussed above heavily use the given input schema. However, XQuery Core does not require an input schema; it can also deal with well-formed documents, which are typed with `AnyType`. In this case, the translated XQuery Core expression cannot be simplified, and the inferred type would be as follows:

```
ELEMENT book {
 ELEMENT title {AnyType}*,
 ELEMENT author {AnyType}*,
 ELEMENT review {AnyType}*
 }*
```

This type is still more specific than `AnyType`; it guarantees that the result consists of nothing but `<book>`s, which contain nothing but a sequence of `<title>`, `<author>`, and `<review>` elements.

3. CHALLENGES

Since their first release, XQuery and XQuery Core have converged and matured. First implementations have been presented at XML DevCon 2001 Spring in April 2001. A lot of the current work is devoted to polishing the designs at hand and to fully aligning XQuery with the existing family of XML standards. Apart from these rather tedious tasks, there also exist some challenges that probably go beyond what a standardization committee can and should achieve, at least for Version 1. Support for updates is a very obvious challenge, but discussing their implication goes beyond the scope of this paper. Here is a highly subjective and certainly not exhaustive selection of some other, maybe not so obvious challenges.

3.1 Expression Level

While XQuery and XQuery Core are good at selecting and recombining portions of documents, queries that introduce nesting structure on a flat node sequence are not that easy to express:

One such class of queries are group-by queries that partition a flat set, typically the 1:1 XML representation of a flat relational table, into a set of equivalence classes, typically a nested XML document. Such queries can be expressed by nested queries. However, arbitrary nested queries are difficult to optimize; they often require more than one pass through the document, whereas explicit group-by queries can be processed in one pass. An early version of XQuery Core contained an explicit group-by operator; other approaches are being discussed.

A similar, more challenging class of queries are queries that segment a flat *sequence* according to some pattern, typically a combination of a regular expression with some additional predicates. Such queries are important for linguistic applications, where the patterns describe, e.g., some phrase structure to be matched in a flat sequence of tokens, in bio-informatics, where patterns may describe subsequences of, e.g., genetic sequences, but also for extracting structure from semi structured sources. An early version of XQuery Core had included regular expression comprehension as a possible basis for such query classes, other promising approaches are described in [11,12].

Another rather blind spot of XQuery (Core) is ranked information retrieval. Conventional full-text predicates including proximity operators can be supported via user-defined functions. But queries that need to combine rankings obtained from structural similarity and content-related similarity cannot be expressed generically. Some initial approaches to add information retrieval capabilities to query languages for XML can be found in [15, 9].

3.2 Type Level

Inspired by the type system of XDuce [10], XQuery Core's type system has succeeded in transferring much of the state-of-the art of typing functional programming languages to the regular tree types underlying XML-DTDs and XML-Schema. A few, rather subtle technical challenges remain:

The first one is very much an implementation problem: For static type checking, XQuery Core's type system needs to determine whether an inferred type T_1 is a subtype of a given type T_2, i.e., whether T_2 accepts a superset of the instances T_1 accepts. If the given type T_2 is an arbitrary regular tree (or forest) this is exponential. However, if the given type adheres to the 1-unambiguity constraint imposed by XML-DTDs and XML-Schema, there exists an efficient polynomial algorithm to decide about subtyping [12]. It therefore may be prudent to impose XML-Schema's 1-unambiguity constraint on explicitly declared types.

The second challenge arises from the rather rich subtyping mechanisms supported by XML-Schema. XML-Schema allows deriving a type T_1 from another type T_2 by extending T_2 with arbitrary elements, including elements that are already in T_2. This makes it difficult to view an instance of T_1 as an instance of T_2, which should hide all additional elements of T_2.

The third challenge comes from typing parameter polymorphic, recursive functions. XQuery Core requires that the input- and output type of a function is declared explicitly. For polymorphic functions that accept any well-formed document, the most specific output-type that can be declared explicitly is often `AnyType`. For such functions it would be useful to infer a more specific output type from the concrete type of their actual parameter. For general recursive functions this is difficult, but there may exist function classes, such as some forms of structural recursion, for which a generic approach can be accomplished.

3.3 Non Issues

Some of the qualms that have been raised with regard to XQuery Core do not really qualify as genuine challenges.

One qualm is that due to its strong reliance on static typing XQuery Core cannot deal with irregularly structured, well-formed documents. This is not the case. XML Query Core's type system consistently supports `AnyType` as an upper bound, and thus certainly does not require the existence of a DTD or a Schema.

Another qualm is that inferred types do not adhere to all the constraints of XML-Schema, such as 1-unambiguity or the consistent element restriction, which requires that any element has exactly one content-model in a particular context. While certainly useful and appropriate for efficient parsing and subtyping, these constraints destroy closure of the type system, and thus can not be reasonably obeyed for the inferred type of a query that selects and recombines arbitrary portions of a document. However, for every inferred type a possibly more general valid XML-Schema can be given by means of an explicit type declaration.

Finally, XQuery Core is sometimes questioned as a basis for query optimization. This is only partially true. Much of the traditional relational optimization of logical query plans, such as shuffling joins on unordered sequences or pushing down selections, can be expressed within the framework of XQuery Core. However, indeed some of the useful optimization techniques for querying XML, such as deploying special index structures for path expressions do require specialized physical operators, which can be more easily generated from XQuery directly rather than via XQuery Core.

4. CONCLUSIONS

From its rather concise beginnings, the XQuery Formal Semantics document has grown into something quite big. This is probably the fate of any research result meeting standardization. However, its core constituents, static semantics, dynamic semantics and mapping from XQuery to XQuery core, have remained remarkably stable, and have certainly helped in polishing the design of XQuery (a bi-directional process), and in specifying a strong type system for XQuery. This paper has tried to illustrate that the role of the XQuery formalization goes beyond just an academic exercise. All its constituents can be used rather directly to arrive at interoperable and efficient implementations of XQuery. In addition, they can provide a framework to investigate some of the more taunting challenges ahead.

5. ACKNOWLEDGMENTS

I am indebted to the co-editors of the XQuery Formalization working draft, and to the rest of the XML Query working group for a great learning experience. I also want to thank Tobias Groh and Sven Overhage of TU Darmstadt for their valiant efforts in implementing the moving target of XQuery Core, and providing very valuable feedback on the way. The assessments given in this paper are my personal position and flaws are due to me.

6. REFERENCES

[1] Serge Abitboul. Querying Semi-Structured Data. In: Proceedings of the 6th International Conference on Database Theory (ICDT'97),1--18, Delphy, Greece, January 1997

[2] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, M. Stefanescu: XQuery 1.0: An XML Query Language. W3C Working Draft 2001. Available at: www.w3.org/TR/xquery/

[3] D. Chamberlin, P. Fankhauser, M. Marchiori, J. Robie. XML Query Requirements. W3C Working Draft 2001, Available at: www.w3.org/XML/Group/xmlquery/xmlquery-req

[4] D. Chamberlin, P. Fankhauser, M. Marchiori, J. Robie. XML Query Use Cases. W3C Working Draft 2001. Available at: www.w3.org/XML/Group/xmlquery/xmlquery-use-cases

[5] P. Fankhauser, M. Fernandez, J. Siméon, A. Malhotra, M. Rys, P. Wadler. XQuery 1.0 Formal Semantics. Available at: www.w3.org/TR/query-algebra/

[6] M. Fernandez, Jonathan Marsh. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 2001. Available at: www.w3.org/TR/query-datamodel/

[7] Mary Fernandez, Jérôme Siméon, and Philip Wadler, A semi-monad for semi-structured data. In: Proceedings of the International Conference on Database Theory (ICDT'2001), January 2001, London, UK.

[8] Mary Fernandez, Jérôme Siméon, and Philip Wadler, An Algebra for XML Query, FST TCS, Delhi, December 2000.

[9] N. Fuhr, K. Großjohann. XIRQL - An Extension of XQL for Information Retrieval. In: Proceedings of the ACM SIGIR 2000 Workshop on XML and Information Retrieval. (Athens, Greece, July 2000), ACM.

[10] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. In: Proceedings of the International Conference on Functional Programming (ICFP), 2000.

[11] Haruo Hosoya, Benjamin C. Pierce. Regular Expression Pattern Matching for XML. In: Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2001.

[12] Andreas Neumann. Parsing and Querying XML Documents in SML, PHD-Thesis, Trier, Germany, December 1999

[13] David Maier. Database Desiderata for an XML Query Language. W3C Query Language Workshop, Boston, December 1998. Available at: www.w3.org/TandS/QL/QL98/pp/maier.html.

[14] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelson (eds.). XML Schema Part 1: Structures. W3C Recommendation 2001, Available at: www.w3.org/TR/xmlschema-1/

[15] A. Theobald, G. Weikum. Adding Relevance to XML. In: Proc. of 3rd International Workshop on the Web and Databases (WebDB), (Dallas, USA, May 2000).