Growing XQuery

Mary Fernández[1] and Jérôme Siméon[2]

[1] AT&T Labs – Research
180 Park Ave., Florham Park, NJ 07932, USA
mff@research.att.com
http://research.att.com/info/mff
[2] Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
simeon@research.bell-labs.com
http://www-db.research.bell-labs.com/user/simeon/

Abstract. XQuery is a typed, functional language for querying XML data sources. XQuery has features of both traditional query languages and modern functional languages. In this paper, we introduce XQuery from both a "programming language" and a "query language" perspective and consider how these features impact the implementation and the evolution of XQuery. We conclude with a discussion of features currently missing from XQuery, but that we expect users will soon demand.

1 Introduction

XML [42] is a flexible format that can represent many classes of data: structured documents with large fragments of marked-up text; homogeneous records such as those in relational databases; and heterogeneous records with varied structure and content such as those in object-oriented and hierarchical databases. XML makes it possible for applications to handle all these classes of data simultaneously and to exchange such data in a simple, extensible, and standard format. One measure of XML's impact is the proliferation of industry-specific XML vocabularies [14]. Numerous industry groups, including automotive, health care, and telecommunications, publish document type definitions (DTDs) and XML Schemata [43], which specify the format of the XML data to be exchanged between their applications. Ultimately, the goal is for XML to be the "lingua franca" of data exchange, making it possible for data to be exchanged regardless of where it is stored or how it is processed.

For the past (almost) four years, we have been actively involved in defining XQuery 1.0 [48], a query language for XML designed to meet the diverse needs of applications that query and exchange XML. XQuery 1.0 and its sister language XPath 2.0 are designed jointly by members of the World-wide Web Consortium's XSLT and XML Query working groups. Group members are from software vendors, large user communities, and industrial research labs. Broadly speaking, they represent two major software industries and user communities, each of which significantly influence XQuery's design and definition. The "document-processing" community contributes their experience in designing languages and

L. Cardelli (Ed.): ECOOP 2003, LNCS 2743, pp. 405–430, 2003.

tools (e.g., editors, formatters, browsers, and text-search engines) for processing structured documents. In particular, several members helped define the Standard Generalized Markup Language (SGML), from which XML is descended. The "database" community contributes their experience in designing query languages, storage systems, and query engines for data-intensive applications. In particular, several members helped define SQL [15], the standard query language for relational database systems. Each community has also unique and sometimes conflicting requirements for XQuery. Document-processing applications typically require a rich set of text-processing operators and the abilities to search for text that spans XML markup, to query and preserve the relative order of XML document fragments, and to rank approximate search results. Database applications typically require a rich set of operators on atomic types (e.g., numbers, dates, strings), the ability to compare, extract, and transform values in large XML databases, and the ability to construct new XML values that conform to a given schema. Chamberlin gives an excellent overview on these and other influences on the design of XQuery [9].

XQuery, the result of the collaboration of these two communities, is a typed, functional language that supports user-defined functions and modules for structuring large queries. It contains XPath 2.0 [47] as a sublanguage. XPath 2.0 supports navigation, selection, and extraction of fragments of XML documents, and is also an embedded sublanguage of XSLT 2.0 [51]. XQuery also includes expressions to construct new XML values, and to integrate or join values from multiple documents.

Interestingly, XQuery has as much in common with modern programming languages as it does with traditional query languages. User-defined functions and modules, for example, are not features typical of query languages. XQuery's design is also due to the influence of group members with expertise in the design and implementation of other high-level languages. This smaller "programming language" community advocated that XQuery have a static type semantics and that a formal semantics of XQuery be part of the W3C standard. As a result, XQuery has a complete formal semantics [49], which contains the only complete definition of XQuery's static typing rules.

Even though not yet completely specified, XQuery has generated an astounding level of interest from software vendors, potential users, and computer-science researchers. The XML Query working group Web page[1] lists twenty-three publicly announced implementations, many of which are embedded in products that integrate data from legacy databases. The interest of the database-research community in XML, and XQuery in particular, is also overwhelming. Every major database research conference has at least one track on XML and related technologies, and demonstration sessions are rife with XQuery applications. Numerous workshops accommodate the overflow of research papers.

One reason for this flood of activity is that *semi-structured data*, of which XML is one example, is substantially different than relational data, which has been the focus of database research for the past twenty years. These differ-

[1] `http://www.w3.org/XML/Query`

ences challenge most of what database researchers know about storing data and processing queries. Vianu provides a thorough survey of the theoretical issues related to semi-structured data, including schema and constraint languages; type checking of queries; and complexity of query evaluation and checking query containment [40].

If the response to XML by the database community is a flood, the response by the programming-language community is more like a babbling brook. Influential contributions focus on language expressiveness and type checking. XDuce [24] is a statically typed functional language for XML whose key feature is regular expression pattern matching over XML trees. The XQuery type system incorporates some of the structural features of XDuce's type system, as well as the named typing features of XML Schema. Siméon and Wadler formalize the semantics of named typing and establish the relationship between document validation and type matching in XQuery [35]. Cardelli and Ghelli have proposed a tree-based logic [7] as a foundation for expressing the semantics of query languages and schemata for semistructured data. Such a logic can be used to establish the complexity of problems such as query containment and type checking and thus influence development of practical algorithms, much as the the first-order logic serves a foundation for relational query languages. Hosoya and Pierce survey a variety of languages that process XML, focusing on the expressiveness of their type systems and the complexity of type checking [24]. Other contributions address efficient implementation of document validation [10] and XML parsing [25].

Query languages do not exist in isolation of their evaluation environment, which is often a high-level programming language, but the barrier between the query language and its host language is usually high and wide. An impedance mismatch typically exists between the query language's data model and its representation in the programming language's data types, making it difficult (or impossible) to guarantee the type safety of operations in the host language. There are several approaches to addressing this problem. One strategy is identifying a type-safe "embedding" of a data model (and possibly query language) in the host language (e.g., SQL tables in Haskell [30], OQL in Java [3], XML in Java [13], and XML in Haskell [41]). Another strategy is to ignore the barrier entirely by incorporating more programming language features into the query language [31], or to identify a programming language that can serve as a query language [39]. Implementing "hybrid" languages requires understanding implementation techniques for both programming and query languages.

1.1 Growing a Language

> "If we add just a few things – generic types, operator overloading, and user-defined types of light weight ... — that are designed to let users make and add things of their own use, I think we can go a long way, and much faster. We need to put tools for language growth in the hands of the users."
>
> — Guy Steele, "Growing a Language", 1999 [38]

In other work, we have focused on XQuery's static type system [17], on XQuery's formal semantics [18], and on the relationship between XQuery's core language and monads [19]. In this paper, we consider how XQuery may grow from an already powerful *query* language for XML into a *programming language* for XML-aware applications. History shows that successful query languages *do* grow, but often inelegantly. SQL-99 [28] is so large that no implementation supports the complete standard. As Guy Steele envisioned for Java, our vision is that XQuery grow elegantly with the addition of several flexible language features instead of numerous ad-hoc ones. An important open question is what these features should be.

We begin in Section 2 with the basics of XML and XQuery and present an example query that integrates data from two XML sources. We focus on the "query language" characteristics of XQuery in Section 3 and on the "programming language" characteristics of XQuery in Section 4. A critical barrier to XQuery's growth is identifying efficient evaluation strategies for queries on large XML data sources. XQuery's programming-language features make evaluation even more challenging. To familiarize the reader with these issues, we outline the stages of compilation and optimization in an "archetypal" XQuery implementation in Section 5. In Section 6, we look forward to XQuery 2.0 and consider some new features including update statements, exception handling, higher-order functions, and parametric polymorphism – features that require the knowledge and creativity of the programming language community. Our hope is that this tour will encourage readers to take a closer look at XQuery.

2 XML and XQuery Basics

XML often serves as an exchange format for data that is stored in other representations (e.g., relational databases, Excel spreadsheets, files with ad-hoc formats, etc.) or that is generated by application programs (e.g., stock-quote service or on-line weather service). An application may publish the data it wants to exchange as an XML document, or it may provide a query interface that produces XML. In our examples, we assume the data is published in an XML document. The example document in Figure 1 contains a book catalog represented in XML. The document has one top-level catalog element, which contains book elements.

An XML *element* has a name and may contain zero or more *attributes* and a sequence of zero or more properly nested *children* elements, possibly interleaved with character data. An attribute has a name and contains a *simple value*, i.e., character data only. In Figure 1, the book element contains two attributes: an isbn number and a year. All of an element's attributes must have distinct names, but their order is insignificant – so changing the attributes to year followed by isbn does not change the element's value. By contrast, an element's children may share the same names, and their relative order is significant. The book element contains a title element followed by an author, a publisher, a retail_price, and a list_price. The review element is an example of *mixed* content in which character data is interleaved with elements: The title element is embedded in the text of the

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<catalog>
  <book isbn="156352578X" year="2000">
    <title>No Such Thing as a~Bad Day</title>
    <author>Hamilton Jordan</author>
    <publisher>Longstreet Press, Inc.</publisher>
    <retail_price currency="USD">15.40</retail_price>
    <list_price currency="USD">22.10</list_price>
    <review reviewer="Library Journal">
      This book is the moving account of one man's successful battles
      against three cancers...<title>No Such Thing as a~Bad Day</title>
      is warmly recommended.
    </review>
  </book>
  <!-- More books here -->
</catalog>
```

Fig. 1. A book catalog represented in XML

review. This document is *well-formed*, because its elements are properly nested and the attributes of each element have unique names.[2]

XQuery expressions operate on *values* in the XML data model [45], not directly on the character data in XML documents. A value is a sequence of individual *atomic values* or *nodes*. Sequences are central to XQuery, so much so that one atomic value or node and a sequence containing that item are indistinguishable. An atomic value is an instance of one of the the twenty-three XML Schema primitive types (e.g., xs:string, xs:decimal, xs:date, et al) [44]. A node is either a document, element, attribute or text [3]. A document node has a value; an attribute or element node has a name, a value, and a *type annotation*; and a text node has a string value. A node's type annotation specifies that the node is valid with respect to a type defined in a schema.

Although XQuery only requires that input documents be well formed, data-exchange applications often require that some structure be imposed on documents. There are a number of standard schema languages for XML, including: DTDs, part of the original W3C recommendation defining XML [42]; XML Schema, a W3C recommendation which supersedes DTDs [43, 44]; and Relax NG, an Oasis standard [12]. XML Schema features both named and structural types [35], with structure based on tree grammars, whereas all other XML schema languages only express structural constraints. XQuery's type system is based on XML Schema, so it supports both named and structural types. In this paper, we describe only essential features of XML Schema, including named simple types and complex types, global attributes and elements, and atomic simple

[2] The XML specification defines several other constraints for well-formedness.

[3] For simplicity, we omit comment and processing-instruction nodes.

types. We omit anonymous types, local elements and attributes, and derivation of new types by restriction and by extension.

XML Schema's syntax is XML, making it difficult to read, and the same type can be modeled using different constructs, making it a poor notation for types. Instead, we use XQuery's internal type notation, which is concise and orthogonal. Figure 2 defines a schema for book catalogs in XQuery type notation. A schema is a collection of mutually referential declarations of *simple*, *complex*, *element* and *attribute* types.

A simple-type declaration associates a name with an atomic type, a list of atomic types, or a union of atomic types. Atomic types include XML Schema's twenty-three primitive types. The simple-type declaration on line 13 in Figure 2 specifies that the simple-type name ISBN is associated with the atomic type xs:string.

A complex-type declaration associates a name with a model of *node types*. A node type is a document type, a named element or attribute type, or the text type. The complex type declaration on line 2 associates the name Catalog with the model containing one or more book elements, and the declaration on lines 4–12 associate the name Book with the model containing one isbn and one year attribute, one title element followed by one-or-more author elements or one-or-more editor elements, followed by one publisher element, one retail_price element, an optional list_price element, and zero-or-more review elements. In general, atomic and node types can be combined with the infix operators for sequence (,), union (|), and interleave (&), and the post-fix operators zero-or-one (?), one-or-more (+), or zero-or-more (*).

An attribute declaration associates a name with a simple type (lines 14,18, 24, and 29 contain examples), and an element declaration associates an element name with a simple or complex type (lines 1, 3, 16, 17, 19–22, and 28 contain examples).

XQuery expressions operate on data-model values, not directly on documents. Given an (external) document and a type (from a schema), *validation* produces an (internal) data-model value in which every element and attribute node is annotated with a simple or complex type, or it fails. Validation guarantees that a node's content matches the node's type annotation.

2.1 An Example Query

A common application of XQuery is to integrate information from multiple XML data sources. Our example query in Figure 3 integrates information from the Barnes and Ignoble book catalog with information about book sales from the Publisher's Weekly trade magazine. For each author in the catalog, the query produces the total number of and the total sales receipts for books published by the author since 2000. The query illustrates most of XQuery's key features: path expressions for navigating, selecting, and extracting XML values; constructors for creating new XML values; let expressions for binding variables to intermediate results; for expressions for iterating over sequences and for constructing new sequences; and functions for modularizing queries.

```
define element catalog of type Catalog
define type Catalog { element book + }

define element book of type Book
define type Book {
 ( attribute isbn & attribute year ) ,
  element title ,
  ( element author + | element editor + ),
  element publisher ,
  element retail_price ,
  element list_price ? ,
  element review *
}

define type ISBN restricts xs:string
define attribute isbn of type ISBN

define type Name restricts xs:string
define element author of type Name
define element editor of type Name

define attribute year of type xs:integer
define element title  of type xs:string
define element publisher of type xs:string

define element retail_price of type Price
define element list_price of type Price

define type Currency restricts xs:string
define attribute currency of type Currency
define type Price {
  attribute currency , xs:decimal
}

define element review of type Review
define attribute reviewer of type xs:string
define type Review {
  attribute reviewer , ( text | element )*
}

define type Vendor {
  attribute type of type xs:string ,
  element name of type xs:string ,
  element total-sales *
}
define element vendor of type Vendor

define type Sales {
  element author ,
  element count of type xs:integer ,
  element total of type xs:decimal
}
define element total-sales of type Sales
```

Fig. 2. Schema in XQuery type notation for book catalog in Figure 1

Figure 3 contains an XQuery *main module*. A main module consists of imported schemas, user-defined functions, and one main expression, whose value is the result of evaluating the module. The schema imported on line 1 corresponds to the book catalog schema in Figure 2 and is imported as the default schema, which means unprefixed names of nodes and types refer to definitions in the given schema. The schema imported on line 2 corresponds to a schema for book

```
 1. import schema default element namespace = "http://book-vendors.com/catalog.xsd"
 2. import schema namespace sls = "http://book-trade.com/sales.xsd"

 3. define function sales-by-author ($cat as element catalog,
 4.                 $sales as element sls:sales) as element total-sales *
 5. {
 6.  for $name in fn:distinct-values($cat/book/author)
 7.  let $books := $cat/book[@year >= 2000 and author = $name],
 8.      $receipts := $sales/sls:book[@isbn = $books/@isbn]/sls:receipts
 9   order-by $name
10.  return
11.     <total-sales>
12.       <author> { $name } </author>
13.       <count>  { fn:count($books) } </count>
14.       <total>  { fn:sum($receipts) } </total>
15.     </total-sales>
16. }

17. let $bi := fn:doc("http://www.bni.com/catalog.xml"),
18.     $pw := fn:doc("http://www.publishersweekly.com/sales.xml")
19. return
20.   <vendor type="retail" name="Barns and Ignoble">
21.     { sales-by-author($bi/catalog, $pw/sls:sales) }
22.   </vendor>
```

Fig. 3. An XQuery main module

sales and is associated with the prefix sls, which means all elements and types prefixed with sls refer to this schema. We will discuss schemas and typing more in the Section 4.

The function sales-by-author (lines 3–16) is the work-horse of this module. It takes a catalog element and a sls:sales element, and for each author in the catalog, returns a total-sales element containing the author's name, the total number of and the total sales of books that the author published since January, 2000.

This function has several examples of *path expressions*, so we describe those first. The path expression $cat/book/author on line 6 extracts all the author children of book children of the catalog element bound to the variable $cat. Path expressions may conditionally select nodes. The path expression on line 7:

```
$cat/book[@year >= 2000 and author = $name]
```

extracts all book children of the catalog element that have a year attribute with value greater-or-equal to 2000 and that have at least one author child whose content equals the value bound to the variable $name. In database parlance, this path expression *self-joins* the authors and books in the catalog source and *selects* those books published since 2000. Similarly, the path expression on line 8:

```
$sales/sls:book[@isbn = $books/@isbn]/sls:receipts
```

joins the books selected by the path expression on line 7 with the sls:books from the sales source. The nodes are joined on their isbn attribute values. The path expression then extracts or *projects* the sls:receipts elements.

The let expression on lines 7–8 is a classic functional let: It binds the variable on the left-hand-side of := to the value on the right-hand side, then evaluates its body (lines 9–15) given the new variable binding.

Returning to line 6, the function fn:distinct-values[4] takes a sequence of atomic values, possibly containing duplicates, and returns a sequence with no duplicates. When applied to the sequence of author nodes, it returns their string contents with duplicates eliminated. Given this sequence of author names, the for expression on line 4 binds the variable $name to each string in the sequence of author names, evaluates the let expression on lines 5–12 once for each binding of $name, and concatenates the resulting values into one sequence. The for expression corresponds to a monad over sequences of atomic values and nodes [19].

The order-by expression on line 7 guarantees that the sequence produced by the return expression is in sorted order by the authors' names. The return expression on lines 8–13 is evaluated once for each binding of $name. The element constructor on lines 9–13 constructs one total-sales element, and in turn its subexpressions construct one author, one count, and one total element, which contain the author's name, the total number of books published in 2000, and the sum of all book receipts, respectively.

The main expression on lines 17–22 applies the function sales-by-author to the book catalog published by Barns and Ignoble and to the book sales data published by Publisher's Weekly magazine – of course, the function could be applied to any pair of elements that are valid instances of the catalog and sls:sales elements. The function fn:doc accesses the XML document at the given URL, validates it, and maps it into a document-node value. Documents typically contain references to the schemas against which they should be validated. The book catalog is validated against the schema book-catalog.xsd, and the sales document is validated against the schema book-sales.xsd. This correspondence is not explicit in the query, but instead is established by the environment in which the query is evaluated. For example, the fn:doc function might be implemented by a database in which pre-validated documents are stored.

A document node represents an entire XML document and therefore does not correspond to any data in the document itself. The path expression $bi/catalog selects all catalog elements that are children of the document node. The path expression $pw/sls:sales is similar. Lastly, the element constructor on lines 20–22 constructs a new vendor element, which contains the result of applying the function sales-by-author to the values of the path expressions $bi/catalog and $pw/sls:sales.

This quick introduction should give the reader a sense of XQuery's expressiveness and capabilities. For the reader interested in more details, we recommend Robie's XQuery tutorial [33].

3 XQuery as a Query Language

XQuery has many characteristics of traditional query languages, such as SQL, Datalog [2], and OQL [8]. First, Its data model is restricted to those values that XML can represent, that is XQuery's data model includes sequences of nodes

[4] The fn is the namespace prefix that denotes XQuery's built-in functions [46].

and atomic values, but excludes, for example, sets, bags, and nested sequences, because they are not intrinsic to XML.

Second, almost all XQuery operators and expressions either construct or access values in the data model, and common idioms for constructing and accessing values are built-in to the language to improve ease of use. For example, XQuery's equality and inequality operators have a fairly complex implicit semantics. This equality expression evaluates to true if the book bound to `$book` contains at least one `author` child whose content equals the string "Hamilton Jordan":

```
$book/author = "Hamilton Jordan"
```

Thus, the (in)equality operators are existentially quantified over sequences of items: The operators are applied to pairs of items drawn from their operands. If any one item evaluates to a node, the node's (atomic-valued) content is extracted and then compared to the other operand. This implicit semantics improves ease-of-use for the query writer, especially when writing queries over XML documents with irregular structure. The query writer writes the same expression whether a `book` has zero, one, or multiple `author` children. Other expressions in XQuery's user-level syntax also have rich implicit semantics. Although convenient for a user, this rich semantics can complicate typing and evaluation, so the semantics of user-level expressions is made explicit by *normalization* into a smaller core language. Typing, optimization, and evaluation operate on this smaller core language. We discuss normalization and other compilation steps in Section 5.

Third, XQuery is strongly typed, meaning that the types of values and expressions must be compatible with the context in which the value or expression is used. For example, this expression raises a type error because an `isbn` attribute contains a string, which cannot be compared to an integer:

```
$book[@isbn = 156352578]
```

All implementations of XQuery must support dynamic typing, which checks during query evaluation that the type of a value is compatible with the context in which it is used and raises a type error if an incompatibility is detected. Static typing is an optional feature of XQuery implementations and more common in programming languages than in query languages. We discuss static typing in the next section.

Lastly, XQuery is declarative, thus its semantics permits a variety of evaluation strategies. Recall that the function `sales-by-author` *self-joins* the authors and books in the catalog source, *selects* those books published since 2000, *joins* those books with the sales receipts, *projects* the books' receipts, *groups* the resulting books and receipts by author, *aggregates* the total number of books and total receipts, and *orders* the results by the author's name. From a query-language perspective, this function is very expressive and consequently may be difficult to evaluate efficiently. Although it is easy to produce a naive evaluation strategy for this query – simply interpret the query on an in-memory representation of the documents – for all but the smallest input documents, the naive strategy will be prohibitively slow. Because XQuery is declarative, its semantics does not enforce an order of evaluation, and this flexibility permits implementations to use a variety of evaluation strategies. For example, the following expression, in which i1 ... ik are integer values:

```
for $i in (i1, i2,..., ik) return 100 div $i
```

is equivalent to the following sequence expression, which evaluates the body of the for expression once for each value in the integer sequence:

```
(100 idiv i1), (100 idiv i2), ... (100 idiv ik)
```

Because each integer-division expression is independent of all others, they can be evaluated in any order, or even in parallel. We discuss evaluation strategies in Section 5.

Flexible evaluation order permits some expressions to be non-deterministic. For example, the following expression may raise a divide-by-zero error or evaluate to true, depending on which disjunct is evaluated first:

```
(1 idiv 0 < 2) or (3 < 4)
```

The if-then-else conditional expression, however, enforces an evaluation order, thus the or expression above is not equivalent to the following if-then-else, because the else branch is only evaluated if the conditional expression evaluates to false:

```
if (1 idiv 0 < 2) then fn:true()
else if (3 < 4) then fn:true()
else fn:false()
```

XQuery's formal semantics [49] specifies formally where an evaluation order must be enforced and where it is flexible, and it also guarantees that an expression either raises an error or evaluates to a unique value.

4 XQuery as a Programming Language

Despite its similarity to other query languages, XQuery has two significant characteristics more common in programming languages: it is statically typed and it is Turing complete. We consider the impact of these features next.

Static typing, in general, refers to both *type checking* and *type inference*. For each expression in a query, type checking determines if the type of the expression is compatible with the context in which it is used, and type inference computes the type of the expression based on the types of its subexpressions. Neither type checking nor type inference are difficult for languages like SQL and Datalog. The types include only atomic types and tuples of atomic types, and simple inspection of the query determines the type of each expression. As a compositional, object-based language, static typing of OQL is more difficult than static typing of SQL, but less so than static typing of XQuery. OQL types include records, objects and collection (sets, lists, and bags), but does not include regular expressions over types, construction of objects with new types, or user-defined functions or modules. These features make static typing of XQuery like static typing of high-level programming languages.

We expect the reader is familiar with static typing's numerous benefits. Most modern compiled languages (Java, C++, C#, ML, Haskell, etc.) provide static typing to help build large, reliable applications. Static typing can help by detecting common type errors in a program during static analysis instead of the developer discovering those errors when the program is run. Static typing in XQuery serves the same purpose and can detect numerous common errors. For example, it detects the type error in this expression from Section 3 in which a string is compared to an integer:

```
$book[@isbn = 156352578]
```

It can also detect the misspelling of isbn as ibsn in the path expression $book/@ibsn. Assuming that $book has type element book, the static type inferred for $book/@ibsn is the empty sequence, because a book element contains no ibsn attributes. A static type error is raised whenever the type of an expression (other than the literal empty sequence ()) is empty. XQuery's static typing rules also detect when a newly constructed element will never validate against the expected type for that element. In the query Figure 3, the vendor element constructed contains a name *attribute*, but the vendor element type in the schema in Figure 2 expects a name *element* – static type checking detects this error. In addition, static type analysis can help yield more efficient evaluation strategies. We discuss those benefits in the next section.

Given these benefits, it may come as a surprise that static typing is an optional feature of XQuery. One reason is that there is a tension between writing queries that operate on well-formed documents and that are also statically well-typed. In Section 2, we stated that XQuery only requires input documents to be well formed, but also stated that all data-model values be labeled with a type. To represent well-formed documents in the data model, all nodes are labeled with types indicating that no additional type information is known – well-formed elements are labeled with xdt:untypedAny and well-formed attributes are labeled with xdt:untypedAtomic. Assuming that $book has static type element book of type xdt:anyType, the following expression is ill-typed:

```
$book/list_price - $book/retail_price
```

The reason is that the static typing rules for arithmetic operators require that each operand be zero-or-one atomic value. A well-formed book element may have an arbitrary number of list_price and retail_price children, and therefore contain an arbitrary number of atomic values. Because static typing examines a query's expressions, not the values that those expressions produce, static typing a *conservative* analysis. Even though during evaluation every well-formed book element may contain exactly one list_price and one retail_price, static analysis must assume otherwise. To write statically well-typed queries over well-formed data, the query writer must explicitly assert the expected structure of the document. The following expression asserts *statically* that *dynamically* the book element will contain one list_price and one retail_price:

```
fn:one($book/list_price) - fn:one($book/retail_price)
```

This permits static typing to proceed assuming the correct types. If during evaluation, the book does not have the expected structure, a dynamic error is raised. XQuery is designed to be easy to use on both well-formed and validated documents. Because these assertions make writing queries over well-formed documents burdensome, static typing is optional.

XQuery is Turing complete, because it does not restrict recursion in user-defined functions. XML documents support recursive structure and therefore some form of recursion is necessary. For example, here is a schema that describes a parts manifest, in which a part element may contain other part elements.

```
define element part {
  attribute name of type xs:string ,
  attribute quantity of type xs:integer ,
  element parts *
}
```

And here is a parts manifest conforming to the above schema:

```
<element part name="widget" quantity="1">
  <element part name="nut" quantity="100"/>
  <element part name="bolt" quantity="100"/>
</element>
```

Any query that must preserve the recursive structure of the document can only be expressed by a recursive function.

From a database-theory perspective, Turing completeness is heresy. Many optimizations for relational queries require solving the *query containment* problem: Given two queries Q_1 and Q_2 and a schema S, for all databases D such that D is an instance of S, is $Q_1(D)$ contained in $Q_2(D)$, i.e., $\forall D$ s.t. $D : S, Q_1(D) \subset Q_2(D)$? Numerous results from database theory characterize the complexity of query containment based on the expressiveness of the query language. Answering the question has practical implications. Query optimizers use containment to determine whether the result of a new query is contained in a pre-computed *view* – thus potentially reducing the cost of evaluating the new query. Evaluation strategies also use containment to rewrite queries so that they may better utilize physical indices.

Answering the containment question for a Turing-complete program is equivalent to solving the halting problem(!), so to establish containment results for XQuery, we must consider subsets of the language. Most results are restricted to containment of path expressions, which express a very limited form of recursion (i.e., navigation via the descendant and ancestor axes) and do not construct new values. The UnQL [6] query language supports mutually recursive functions over trees, but requires that recursion alway proceed down the tree, thus guaranteeing termination. Considering such a subset of XQuery may help make some queries more amenable to analysis.

5 Implementing XQuery

Most implementations of XQuery are not generic, stand-alone processors, but are designed with particular applications or goals in mind. Examples include processors that operate on streams of XML data [**?**, 22] and ones that query data stored in relational databases and publish it in XML[16]. These implementations are designed for speed and/or scalability, but not necessarily completeness. Our own implementation, called Galax[23] and the IPSI XQuery processor [20] aim for completeness and are the only implementations to date that support static typing. All processors, regardless of how they work, must preserve the XQuery semantics as described in the language and formal semantics documents [48, 49], otherwise they do not implement XQuery! But how they achieve this result is an open and highly competitive area. In this section, we describe an "archetypal" XQuery architecture, which loosely corresponds to the Galax architecture.

5.1 Archetypal Architecture

Figure 4 depicts the query-processing stages of archetypal architecture. The first four query-processing stages (top of diagram) are common in compilers for high-level languages. The later stages (bottom of diagram) are common in interpreters for query languages. *Parsing* takes an expression in XQuery's user-level syntax and yields an abstract syntax tree (AST). We do not discuss this stage further. *Normalization* takes the AST of the user-level expression and maps it into an AST of XQuery's smaller *core* language, which is a proper subset of the user-level language. This stage makes the implicit semantics of user-level expressions explicit in the core language. The optional *static typing* stage takes the core AST and yields the same AST in which every expression node is annotated with its static type. *Logical optimization* takes the core AST (with or without static type annotations) and applies logical rewriting rules, such as common-subexpression elimination, constant folding, hoisting of loop-invariant expressions, etc., and if static types are known, type-specific simplifications.

The first four stages typically are independent of the physical representation of documents, whereas the last three depend on the representation. A fast-path to a complete implementation is building a simple interpreter for the typed XQuery core. We initially took this path in Galax, but are now extending Galax to include the later stages. *Compilation* takes a (typed) core AST and compiles it into an algebraic query plan that depends on the physical operators available for accessing the document. Whereas the core AST is a "top-down" representation of the original expression, the algebraic query plan is a "bottom up" representation. *Physical optimization* takes a query plan and improves it by utilizing any available indices. Lastly, the *evaluation* stage interprets the optimized query

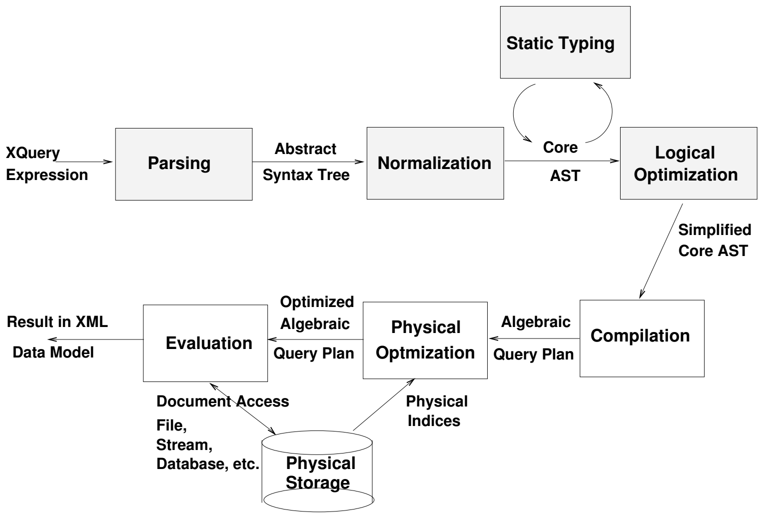

Fig. 4. Archetypal Architecture

plan and yields an XML value in the data model, which is returned to the environment in which the query was evaluated. We illustrate the normalization, logical optimization, and physical optimization stages on a simplified version of the query in Figure 3.

Our example architecture figure excludes the document-processing stages, which are highly implementation dependent. An implementation typically will provide a few methods for accessing documents, for example, in the file system, on the network [1, 22], in native XML databases with specialized indices [11, 5, 29], or in relational databases [4, 34], but most do not provide all possible methods. Our example architecture assumes that documents are stored in a relational database.

5.2 Normalization

To illustrate normalization, we consider a variant of the query in Figure 3 that computes the number of books published by each author since 2000:

```
for $name in distinct-values($cat/book/author),
let $books := $cat/book[@year >= 2000 and author = $name]
return
  <total-sales>
    <author> { $author } </author>
    <count> { count($books) } </count>
  </total-sales>
```

Recall from Section 3 that many user-level expressions have a complex implicit semantics. This semantics improves ease-of-use when writing queries over

XML documents whose structure may not be known, but complicates static typing and compilation into algebraic query plans. Normalization rewrites each user-level expression into an expression in the core syntax that has the same semantics but in which each subexpression has a very simple semantics. By necessity, normalization precedes static typing, so the rewritings are independent of typing information. For example, the path expression $cat/book[@year >= 2000] in the query above is is normalized into the following core expression:

```
for $_c in $cat return
  for $_b in $_c/child::book return
  if (some $v1 in fn:data($_b/attribute::year) satisfies
        some $v2 in fn:data(2000) satisfies
          let $u1 := fs:promote-operand($v1,$v2) return
          let $u2 := fs:promote-operand($v2,$v1) return
            op:ge($u1, $u2))
  then $_b
  else ()
```

The implicit iteration in each step of the path expression is made explicit in the nested for expressions. The axis (or direction) in which path navigation proceeds is also made explicit – in this case, it is the child axis. The implicit existential quantification of the predicate expression is made explicit in the nested some expressions, and the automatic extraction of atomic values from a sequence of atomic values or nodes is handled by the fn:data function. Before applying the overloaded greater-than-or-equal operator, the pair of atomic values are promoted to comparable types, if possible. For example, promoting a float and decimal yields two floats, and promoting a decimal and date would raise a type error, because they are incomparable. If the book element bound to $_b satisfies the conditional expression, the conditional evaluates to the book element, otherwise it evaluates to the empty sequence (). The for expressions yield a single sequence of all the book elements.

From the very small example above, we can see that normalization yields large core expressions in which each sub-expression has a simple semantics. For simplicity, we have omitted other explicit operations, e.g., that guarantee the result of every path expression is in document order. After normalization and static typing, logical optimizations can further simplify the core expression.

5.3 Logical Optimization

Many standard optimizations for functional languages, such as elimination of common subexpressions, constant propagation and folding, function inlining, and elimination of unused variables, are applicable to XQuery. For example, he normalization of $cat/book[@year >= 2000] in the last section can be simplified to:

```
for $_c in $cat return
  for $_b in $_c/child::book return
  if (some $v1 in fn:data($_b/attribute::year) satisfies
        let $u1 := fs:promote-operand($v1,2000) return
        let $u2 := fs:promote-operand(2000,$v1) return
          op:ge($u1, $u2))
  then $_b
  else ()
```

The application of fn:data to the constant integer 2000 simplifies to the constant itself, the existential quantification over the constant is eliminated, and the constant is propagated to its uses. Without additional type information, the above expression cannot be further simplified, because we do not know, for example, the type of values contained in an year attribute.

Static type information can be used to further simplify expressions. Assuming that $cat has type element catalog of type Catalog, the static types of the other expressions are as follows:

$cat :	element catalog of type Catalog
$cat/child::book :	element book +
$_b :	element book of type Book
$_b/attribute::year :	attribute year of type xs:integer
fn:data($_b/attribute::year) :	xs:integer
$v1 :	xs:integer
$u1 :	xs:integer
$u2 :	xs:integer

Given this information, the above expression is simplified to:

```
for $_b in $cat/child::book return
if (op:integer-ge(fn:data($_b/attribute::year), 2000))
then $_b
else ()
```

The first for expression is eliminated (because its input sequence is a single element). Similarly, the existential quantification over the year attribute's single xs:integer value is eliminated. Because both arguments to fs:promote-operand are integers, the promotions are eliminated, and the overloaded op:ge operator is replaced by the monomorphic op:integer-ge. Even these basic simplifications can substantially reduce the size and complexity of query plans. Although not illustrated by this example, another important logical optimization is determining when the order of values produced by an expression is insignificant. Knowing that order is insignificant can yield more efficient evaluation plans – we return to this issue in the next section.

Returning to the example query that computes the number of books published by each author since 2000, the simplified core expression assuming static typing is:

```
for $name in
  distinct-values(for $_b in $cat/child::book return
                    fn:data($_b/child::author))
return
let $books :=
  for $_b in $cat/child::book return
  if (op:integer-ge(fn:data($_b/attribute::year), 2000)
      and
      some $_a in $_b/child::author
      satisfies fn:data($_a) = $name)
  then $_b
  else ()
return
  <total-sales>
    <author> { $name } </author>
    <count>  { fn:count($books) } </count>
  </total-sales>
```

Although this expression is substantially simpler than the core expression that is dynamically typed, a naive evaluation strategy is quadratic in the number of distinct author names and books (i.e., each book in the catalog is accessed once for each author in the catalog). Clearly, this is impractical for any document in which the set of books and authors exceed main memory.

5.4 Physical Optimization

Efficient evaluation strategies are possible if the physical representation of the XML documents is taken into account. Typically, an evaluation plan is composed of algebraic operators specialized to the access methods provided by the storage system. A common technique is to store XML documents in relational tables and use operators that process streams of tuples. This strategy can take advantage of high-performance relational query engines. Native XML databases with custom indices over trees and algebras for utilizing these structures also exist [26, 29], but are yet to be shown effective on very large-scale data.

For our small example, we assume the book catalog document is stored in a relational database containing the two tables:

```
BookTable(bid, title, year)
AuthorTable(bid, name, idx)
```

The BookTable table contains one tuple for each book; each tuple contains the book's year, its title, and a key field (bid) that uniquely identifies the book in the catalog document. The AuthorTable table contains one tuple for each author in each book. The bid field is the unique identifier of the book, name is the name of the author, and idx is the ordinal index of the given author in the book's sequence of authors. We chose this representation, because it is simple. There are numerous techniques for "shredding" XML document into relational tables [34], and with each technique, there are corresponding trade-offs in query performance [4].

Given the relational representation above, the compilation stage rewrites the core expression into a tree (or graph) of operations that consume and produce streams of tuples. We assume the following standard operations:

– **Scan** produces each tuple in a table.
– **Select** takes a stream of tuples and a predicate and produces a stream of tuples that satisfy the predicate.
– **Project** takes a stream of tuples and a set of field names and produces a stream of tuples that contain the specified fields.
– **Join** takes two tuple streams and a join predicate, and produces tuples from the Cartesian product of the two input streams that satisfy the join predicate.
– **Map** takes an input tuple stream and a variable, binds the variable to each tuple in the stream, and evaluates an expression given the variable binding, which yields a result tuple. The operator yields the result tuple extended with the variable bound to the input tuple.
– **Group-by** takes a stream of tuples and a grouping expression and produces one tuple for each distinct value in the grouping expression. Each such tuple contains the group-by value and the partition of (nested) tuples for which the grouping expression has the given value.

A naive compilation strategy takes the core expression and mechanically produces a query-evaluation plan composed of the above operators. The query plan for our example query is:

```
Map(
  AUTHOR-BOOKS ;
    Map(
      AUTHOR ;
      distinct(
        Project(A1.name,
          Join(Scan(A1 in AuthorTable),
               Scan(B1 in BookTable),
               A1.bid = B1.bid)
        )
      ),
      Select(Join(Scan(A2 in AuthorTable),
                  Scan(B2 in BookTable),
                  A2.bid = B2.bid),
             B2.year >= 2003 AND A2.author = AUTHOR)
      As BOOKS
    ),
    <total-sales>
       <author> { AUTHOR-BOOKS.AUTHOR } <author>
       <count> { count(AUTHOR-BOOKS.BOOK) } </count>
    </total-sales>
)
```

The variables A1, A2, B1, and B2 denote *tuples* in the AuthorTable and BookTable tables, respectively. Path navigation is compiled in to scans over tables and joins over the author and book tuple streams. The predicate in the path expression is compiled into a select operation. The outer for expression over distinct author names is compiled into a map operation and the the inner let expression into the outer map operation. Obviously, this query plan is not better than naive evaluation of the core expression, but given this representation, we can now apply database optimization techniques. For example, query unnesting techniques can be applied, and the nested query can be converted into a group-by operation. One possible query plan is as follows:

```
Map(
  AUTHOR-BOOKS ;
  GroupBy(
     Select(Join(Scan(A1 in AuthorTable),
                 Select(Scan(B1 in BookTable), B1.year >= 2000)
                 A1.bid = B1.bid)) As BOOKS,
     A1.name As AUTHOR
  ),
  <total-sales>
    <author> { AUTHOR-BOOKS.AUTHOR } </author>
    <count>  { count(AUTHOR-BOOKS.BOOKS) } </count>
  </total-sales>
)
```

This plan applies the selection predicate early and performs only one join over the author and book tables.

This example illustrates that there are many strategies for evaluating even the simplest XQuery expressions and that choices made in each stage can influence the result of later stages. Although we have described a processing model in which early stages are independent of later stages, the physical operators available may influence decisions made in earlier stages, for example, when or if to apply function inlining or unrolling of tail-recursive function calls. Understanding the interaction between earlier "programming language" stages and later "query processing" stages is an important and open problem when implementing XQuery.

6 Growing XQuery

We cannot predict what XQuery 2.0 will be, but we observe that XQuery 1.0 is growing already. Requirements for fulltext operators already exist [50], and we expect more special-purpose operators will follow. XL [21], a programming language for web services, is based on XQuery. Xduce, a cousin of XQuery, is becoming Xtatic, a programming language for XML [24]. Our own experiences with Galax constantly reveal opportunities in which a richer XQuery semantics would permit our users to build more XML applications faster and more reliably.

We expect that some (many?) of our working-group colleagues will object to our suggestions that XQuery evolve into a programming language for XML. But we believe it is prudent to consider version 2.0 features now, before many incompatible feature sets emerge.

We focus on features already in demand and those that we believe will help XQuery grow in a disciplined way: updates, exception handling, higher-order functions, and parametric polymorphism. Even if XQuery were to have all these features, it still has to co-exist within a variety of environments. We conclude with a discussion of XQuery's interface to other host languages.

Update statements are conspicuously absent from XQuery 1.0, and are the most frequently requested feature. Database programmers rightly expect the ability to query *and* update XML. Updates were excluded from XQuery 1.0, because they require substantial study to get right, and thus would delay delivery of XQuery 1.0. Lehti has proposed an update language for XQuery [27] in which an insert, delete, or replace statement specifies how to update a node or location, and a path expression denotes the node or location to update. This statement updates our example book catalog by inserting a new book element:

```
insert
  <book isbn="0399127380" year="1982">
    <title>Crisis: The Last Year of the Carter Presidency</title>
    <author>Hamilton Jordan</author>
    <publisher>Putname Pub Group</publisher>
    <retail_price currency="USD">1.94</retail_price>
  </book>
before $cat/catalog/book[@isbn="156352578X"]
```

Update statements are an imperative feature, but more restricted than pointers in imperative languages or reference values in functional languages, making it possible to retain some benefits of declarativeness, such as flexible evaluation order. To permit reordering of update and query statements, it must be possible to determine "non-interference" between statements. A formal semantics of updates would help establish criteria for non-interference, and thus should be specified before officially adding updates to XQuery.

Another feature missing from XQuery is exception handling. XQuery's built-in functions may raise errors, and user-defined errors can be raised by calling the function fn:error, which takes any atomic value or node as an argument. For example, this expression raises an error containing a myerror element:

```
fn:error(<myerror>An error in my query</myerror>)
```

There is no expression, however, for catching and handling errors – errors are propagated to the environment in which the expression is evaluated. As more libraries of XQuery functions are created and used, the ability to detect and recover from errors becomes an important usability issue. The working group debated a proposal for an exception-handling expression. A try expression takes an expression and zero or more catch branches labeled with types, and conditionally evaluates a branch if the expression raises an error value that matches the branch's type. For example, the expression below either evaluates to the value of *Expr* or if *Expr* raises an error, and the error value matches element myerror, the try evaluates to the string "My error", otherwise any other error is re-raised.

```
try (Expr)
catch $err as element myerror return  "My error"
default $err return fn:error($err) {-- Re-raise the error --}
```

One reason the try-catch expression was excluded from XQuery 1.0 is its potential interaction with updates. It was not immediately clear what the semantics of updates should be in the presence of exceptions and exception handling. For example, should exception handling enforce a transactional semantics (i.e., the ability to rollback or commit) on update statements? Because we cannot yet answer these questions, we decided to study updates and exception handling together in XQuery 2.0.

Although XQuery is a functional language, it does not support higher-order functions or parametric polymorphism – two of the most powerful programming constructs in languages like O'Caml, Standard ML, and Haskell. In higher-order languages, functions are first-class values and, for example, can be bound to variables and passed as arguments to other functions. Higher-order functions promote code reuse, much as method overriding promotes code reuse in object-oriented languages. Parametric polymorphism permits a function to have one definition but to operate on values of different types. Higher-order functions and parametric polymorphism are most powerful when combined. For example, this O'Caml signature for the function quicksort takes a list of values of any type 'a, a comparison function that takes two 'a values and returns an integer, and returns a list of 'a values in sorted order.

```
quicksort : 'a list -> ('a * 'a -> int) -> 'a list
```

XQuery 1.0 has ad-hoc polymorphism. All the infix operators and many built-in functions are overloaded, e.g., the arithmetic operators can be applied to any numeric type. Users can simulate polymorphism by constructing a new type that is the union of a fixed set of types and then define a function that expects the union type. But this requires that the input types be known in advance of writing the function, which defeats much of the usefulness of polymorphism. Like exception handling, higher-order functions and parametric polymorphism become more important as users write more libraries. For example, we can imagine an XQuery library that constructs and processes SOAP messages [37], which consist of generic headers and application-specific payloads. An XQuery library for SOAP could take as arguments functions that construct and process the application-specific payloads. Not surprisingly, as higher-order functions and parametric polymorphism increase expressiveness, they also increase the complexity of static typing and evaluation. But because XQuery is a functional language, they are natural features to consider.

In Section 1, we noted that query languages, no matter how expressive or complete, are usually evaluated within a host programming language. In general, the host language provides the application's interface to the query language, and vice versa. For example, a host language might convert a user's request posted in a form into a query expression and convey the query results to the user through a GUI. In our experience with Galax, the application programming interface (API) is often the first experience a user has with XQuery. The API must be lightweight enough for new users to understand, but complete enough so that experienced users can exercise all the features of the underlying implementation. We have designed a traditional functional API to Galax, but an important open question is what the boundary should be between XQuery and its host languages. Despite attempts to tightly couple SQL's and OQL's data models with host type systems and to embed SQL in a host language, in practice, the barrier between query and host language remains solid.

Designing XQuery 1.0 has been both an invigorating and exhausting experience. The requirements of vendors, expectations of users, and scrutiny of academics has added equal amounts of challenge and frustration. We believe the resulting language will be a success, and that with success, users will demand that XQuery grow to meet their XML programming needs. We hope to influence that growth by adding a small number of powerful language features. In that way, we hope to put the tools for XQuery's growth in the hands of its users.

Acknowlegments We are privileged to work with the talented and dedicated members of the W3C XML Query Working Group. In particular, we thank Philip Wadler, friend and colleague, for many years of fruitful collaboration on XQuery.

References

[1] S. Abiteboul, O. Benjelloun, I. Manolescu et al. "Active XML: Peer-to-Peer Data and Web Services Integration", Proceedings of Conference on Very Large Databases (VLDB) 2002, pp 1087–1090. 419

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*, Addison-Wesley, 1995. 413

[3] S. Alagic. "Type-Checking OQL Queries In the ODMG Type Systems", Transactions on Database Systems, 24(3), 1999, pp 319–360. 407

[4] P. Bohannon, J. Freire, P. Roy, and J. Siméon. "From XML Schema to Relations: A Cost-Based Approach to XML Storage", International Conference on Data Engineering, 2002, pp 209-218. 419, 423

[5] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. "Covering indexes for branching path queries" Proceedings of ACM SIGMOD Conference 2002, pp 133–144. 419

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. "A Query Language and Optimization Techniques for Unstructured Data", Proceedings of ACM SIGMOD Conference, 1996, pp 505–516. 417

[7] L. Cardelli and G. Ghelli. "A Query Language based on the Ambient Logic", Mathematical Structures in Computer Science, 2003. 407

[8] R. G. Cattell et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997. 413

[9] D. Chamberlin, "Influences on the Design of XQuery", in *XQuery from the Experts: A Guide to the W3C XML Query Language*, edited by H. Katz, Addison-Wesley, 2003. 406
[10] T. Chuang, "Generic Validation of Structural Content with Parametric Modules", Proceedings of ACM SIGPLAN International Conference on Functional Programming, 2001, pp 98–109. 407
[11] C. Chung, J. Min, and K. Shim, "APEX: an adaptive path index for XML data", Proceedings of ACM SIGMOD Conference 2002, pp 121–132. 419
[12] J. Clarke and M. Makoto. RELAX NG specification, Oasis, 2001, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html. 409
[13] R. Connor et al. "Extracting Typed Values from XML Data 407
, OOPSLA Workshop on Objects, XML, and Databases, 2001.
[14] R. Cover. The XML Cover Pages: XML Industry Sectors,
http://www.xml.org/xml/industry_industrysectors.jsp. 405
[15] H. Darwen (Contributor) and C. J. Date. *Guide to the SQL Standard : A User's Guide to the Standard Database Language SQL*, Addison-Wesley, 1997. 406
[16] M. Fernandez, Y. Kadiyska, D. Suciu et al. "SilkRoute: A framework for publishing relational data in XML", Transactions on Database Systems 27(4), 2002, pp 438-493. 418
[17] M. Fernández, J. Siméon and P. Wadler. "Static Typing in XQuery", in *XQuery from the Experts: A Guide to the W3C XML Query Language*, edited by H. Katz, Addison-Wesley, 2003. 408
[18] M. Fernández, J. Siméon and P. Wadler. "Introduction to the Formal Semantics", in *XQuery from the Experts: A Guide to the W3C XML Query Language*, edited by H. Katz, Addison-Wesley, 2003. 408
[19] M. Fernández, J. Siméon and P. Wadler. "A Semi-monad for Semi-structured Data", International Conference on Database Theory, 2001, pp. 263–300. 408, 413
[20] P. Fankhauser, T. Groh and S. Overhage. "XQuery by the Book: The IPSI XQuery Demonstrator", International Conference on Extending Database Technology (EDBT), 2002, pp 742–744. 418
[21] D. Florescu, A. Grünhagen and D. Kossmann. "XL: a platform for Web Services", Conference on Innovative Data Systems Research (CIDR) 2003, Online Proceedings. 424
[22] D. Florescu, C. Hillary, D. Kossmann, et al. "A Complete and High-performance XQuery Engine for Streaming Data", Proceedings of Conference on Very Large Databases (VLDB) 2003, to appear. 418, 419
[23] M. Fernández, J. Siméon et al. Implementing XQuery 1.0: The Galax Experience Demonstration track, Proceedings of Conference on Very Large Databases (VLDB) to appear, 2003, `http://db.bell-labs.com/galax/`. 418
[24] H. Hosoya and B. Pierce. "XDuce: A Statically Typed XML Processing Language", ACM Transactions on Internet Technology, to appear, 2003. 407, 424
[25] Q. Jackson. "Efficient Formalism-Only Parsing of XML/HTML Using the S-Calculus", ACM SIGPLAN Notices, 38(2), Feb. 2003. 407
[26] H. V. Jagadish et al. "TAX: A Tree Algebra for XML", Workshop on Databases and Programming Languages, LNCS 2397, 2002, pp. 149–164. 422
[27] P. Lehti. "Design and Implementation of a Data Manipulation Processor for an XML Query Language", Technische Universität Darmstadt Technical Report No. KOM-D-149, `http://www.ipsi.fhg.de/ lehti/diplomarbeit.pdf`, August, 2001. 425

[28] SQL, Parts 1 – 13, International Organization for Standards (ISO), Technical reports ISO/IEC 9075-1:1999 through ISO/IEC 9075-13:1999. 408
[29] H. V. Jagadish, S. Al-Khalifa, A. Chapman, et al. "TIMBER: A Native XML Database", The VLDB Journal, 11(4), 2002, pp 274–291. 419, 422
[30] D. Leijen and E. Meijer. "Domain Specific Embedded Compilers", USENIX 2nd Conference on Domain-Specific Languages, 1999, pp 109-122. 407
[31] P. Manghi et al. "Hybrid Applications over XML: Integrating the Procedural and Declarative Approaches", ACM CIKM International Workshop on Web Information and Data Management (WIDM'02), 2002. 407
[32] F. Neven. "Automata Theory for XML Researchers", ACM SIGMOD Record, 31(3), Sept. 2003.
[33] J. Robie. "An Introduction to XQuery", in *XQuery from the Experts: A Guide to the W3C XML Query Language* edited by Howard Katz, Addison-Wesley, 2003. 413
[34] J. Shanmugasundaram, K. Tufte, G. He, et al. "Relational Databases for Querying XML Documents: Limitations and Opportunities", Proceedings of Conference on Very Large Databases (VLDB) 1999, pp 302–314. 419, 423
[35] J. Siméon and P. Wadler. "The Essence of XML", ACM Symposium on Principles of Programming Languages, 2003, pp 1–13. 407, 409
[36] F. Simeoni et al. "Language Bindings to XML", IEEE Internet Computing, 7(1), Jan./Feb. 2003.
[37] SOAP 1.2 Part 1: Messaging Framework, W3C Proposed Recommendation, May 2003. http://www.w3.org/TR/2003/PR-soap12-part1-20030507/. 426
[38] G. Steele. "Growing a Language", Journal of Higher-Order and Symbolic Computation, 12(3), Oct 1999, pp 221–236. 407
[39] V. Tannen, P. Buneman and S. Naqvi. "Structural Recursion as a Query Language", Workshop on Databases and Programming Languages, 1991, Morgan Kaufmann, pp 9–19. 407
[40] V. Vianu. "A Web Odyssey: from Codd to XML", Proceedings of ACM Symposium on Principles of Database Systems, pp 1–16, 2001. 407
[41] M. Wallace and C. Runciman. "Haskell and XML: generic combinators or type-based translation?", Proceedings of ACM SIGPLAN International Conference on Functional Programming, 1999, pp 148–159. 407
[42] Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. http://www.w3.org/TR/REC-xml/. 405, 409
[43] XML Schema Part 1: Structures. W3C Recommendation, May 2001. 405, 409
[44] XML Schema Part 2: Datatypes. W3C Recommendation, May 2001. 409
[45] XQuery 1.0 and XPath 2.0 data model. W3C Working Draft, May 2003. http://www.w3.org/TR/query-datamodel/. 409
[46] Xquery 1.0 and xpath 2.0 functions and operators version 1.0. W3C Working Draft, May 2003. http://www.w3.org/TR/xpath-operators/. 413
[47] XPath 2.0. W3C Working Draft, May 2003. http://www.w3.org/TR/xquery/. 406
[48] XQuery 1.0: An XML Query Language. W3C Working Draft, May 2003. http://www.w3.org/TR/xquery/. 405, 418
[49] XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, May 2003. http://www.w3.org/TR/query-semantics/. 406, 415, 418
[50] XQuery and XPath Full-text Requirements W3C Working Draft, May 2003. http://www.w3.org/TR/xmlquery-full-text-requirements/. 424

[51] XSL Transformations (XSLT) Version 2.0. W3C Working Draft, May 2003. http://www.w3.org/TR/xslt20/. 406