XQuery: a typed functional language for querying XML

Philip Wadler

Avaya Labs, wadler@avaya.com

Abstract. XQuery is a typed, functional language for querying XML, currently being designed by the XML Query Working Group of the World-Wide Web Consortium. Here are examples of XQuery queries on a suitable XML document describing books. To list titles of all books published before 2000 you might write:

```
/BOOKS/BOOK[@YEAR < 2000]/TITLE
```

To list the year and title of all books published before 2000 you might write:

```
for $book in /BOOKS/BOOK
where $book/@YEAR < 2000
return <BOOK>{ $book/@YEAR, $book/TITLE }</BOOK>
```

And to list for each author the titles of all books by that author you might write:

```
for $author in distinct(/BOOKS/BOOK/AUTHOR) return
  <AUTHOR NAME="{ $author }">{
    /BOOKS/BOOK[AUTHOR = $author]/TITLE
  }</AUTHOR>
```

1 Introduction

XQuery is a typed, functional language for querying XML. These notes provide an introduction to XQuery and related XML standards.

XQuery is currently being designed by the XML Query Working Group of the World-Wide Web Consortium (W3C). The design is currently in flux. The design is expressed in a number of documents, including a prose specification [1], a formal specification [2], and a library of functions and operators [3].

XQuery is closely related to other standards for XML. These include XML itself [4, 5], XML Namespaces [6], XML Schema [7], XML Stylesheet Transformations (XSLT) [8, 9], and XPath [10, 11]. XPath includes a common core of material included in XQuery, XSLT, and another standard, XPointer [12], and

the continued development of XPath is under the joint management of the XML Query and XML Stylesheet working groups.

Since the design is in flux, consult the current version of the standard for the latest version. All opinions expressed are my own. Other members of the XML Query working group may hold different opinions. (Some certainly do!)

2 XQuery by example

To get started, here are three examples of XQuery queries. Assume you have an XML document describing books — the format of this document is discussed further below. To list titles of all books published before 2000 you might write:

```
/BOOKS/BOOK[@YEAR < 2000]/TITLE
```

To list the year and title of all books published before 2000 you might write:

```
for $book in /BOOKS/BOOK
where $book/@YEAR < 2000
return <BOOK>{ $book/@YEAR, $book/TITLE }</BOOK>
```

And to list for each author the titles of all books by that author you might write:

```
for $author in distinct(/BOOKS/BOOK/AUTHOR) return
  <AUTHOR NAME="{ $author }">{
    /BOOKS/BOOK[AUTHOR = $author]/TITLE
  }</AUTHOR>
```

3 XQuery data model

Here is a sample XML document describing books, suitable as input to the above queries.

```
<BOOKS>
  <BOOK YEAR="1999 2003">
    <AUTHOR>Abiteboul</AUTHOR>
    <AUTHOR>Buneman</AUTHOR>
    <AUTHOR>Suciu</AUTHOR>
    <TITLE>Data on the Web</TITLE>
    <REVIEW>A <EM>fine</EM> book.</REVIEW>
  </BOOK>
  <BOOK YEAR="2002">
    <AUTHOR>Buneman</AUTHOR>
    <TITLE>XML in Scotland</TITLE>
    <REVIEW><EM>The <EM>best</EM> ever!</EM></REVIEW>
  </BOOK>
</BOOKS>
```

XML data tends to come in two styles, database-like and document-like. The above has aspects of both. This dual nature of XML is one of its more interesting aspects. (There is an old Saturday Night Live routine: "It's a floor wax! It's a dessert topping! It's both!" XML is similar. "It's a database! It's a document! It's both!")

XML data often resembles a database. Listing a year, an author, a title, and a review for each book is reminiscent of the columns in a relational database. However, the use of multiple author elements for a single book differs from the traditional relational approach, which would either have a single author entry with an array of authors, or a separate table relating books to authors.

XML data often resembles a document. Using markup to indicate emphasis in a review is typical of documents such as HTML. Note the recursive use of emphasis in the second review, where an enthusiastic reviewer has marked the entire review as emphasized, then further emphasized one word of it.

XML is a notation for writing trees. Below is the representation we use to describe the tree corresponding to the XML document above.

```
document {
  element BOOKS of type BOOKS-TYPE {
    element BOOK of type BOOK-TYPE {
      attribute YEAR of type INTEGER-LIST { 1999, 2003 },
      element AUTHOR of type xs:string { "Abiteboul" },
      element AUTHOR of type xs:string { "Buneman" },
      element AUTHOR of type xs:string  { "Suciu" },
      element TITLE of type xs:string { "Data on the Web" },
      element REVIEW of type INLINE {
        text { "A " },
        element EM of type INLINE { text { "fine" } },
        text { " book." }
      }
    }
    element BOOK {
      attribute YEAR of type INTEGER-LIST { 2002 },
      element AUTHOR of type xs:string { "Buneman" },
      element TITLE of type xs:string { "XML in Scotland" },
      element REVIEW of type INLINE {
        element EM of type INLINE {
          text { "The " },
          element EM of type INLINE { text { "best" } },
          text { " ever!" }
        }
      }
    }
  }
}
```

Here the leaves of the tree are either strings (enclosed in quotes) or integers (not in quotes), and the nodes of the tree are labeled as document, element, attribute, or text nodes. Each element and attribute node is labeled with a type; the source of these types is the validating XML Schema, which is defined below.

Since the purpose of XML is as a notation for data interchange, one would expect the mapping from XML into the corresponding tree to be trivial. Alas, it is not. In the above, we ignored whitespace between elements. How can one tell where whitespace is and is not significant? We mapped the first `YEAR` attribute to a list of two integers (`1999, 2003`). How can one know that this is the correct interpretation, rather than a string (`"1999 2003"`), or a list of two strings (`"1999", "2003"`)? This information, too, comes from the validating XML Schema, which we describe next.

4 XML Schema

The expected format of an XML document can be described with an XML Schema. From the schema one can determine the expected structure of the markup, and what datatype (if any) is associated with character data.

Typically, XML input consists of both an document and a schema that should be used to *validate* that document. Validation does three things. First, it checks that the document has the format indicated by the schema. Second, it labels the internal representation of the document with the type specified by the schema. Among other things, this resolves the questions of determing when whitespace is significant, and of distinguishing between strings and integers. Third, it may supply default values for omitted attributes. (We don't deal with the third point further here.)

In many contexts, the intended XML Schema is known, and only the document is supplied. There are also conventions by which an XML document can indicate an associated XML Schema. It may also be that there is no Schema, this is discussed further below.

Here is a Schema for the document described in the preceding section.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="BOOKS" type="BOOKS-TYPE"/>
  <xs:complexType name="BOOKS-TYPE">
    <xs:sequence>
      <xs:element name="BOOK" type="BOOKS-TYPE"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="BOOK-TYPE">
    <xs:sequence>
      <xs:element name="AUTHOR" type="xs:string"
         minOccurs="1" maxOccurs="unbounded"/>
      <xs:element name="TITLE" type="xs:string"/>
```

```
      <xs:element name="REVIEW" type="INLINE"
         minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="YEAR" type="INTEGER-LIST"
      use="optional"/>
  </xs:complexType>
  <xs:complexType name="INLINE" mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="EM" type="INLINE"/>
      <xs:element name="BOLD" type="INLINE"/>
    </xs:choice>
  </xs:complexType>
  <xs:simpleType name="INTEGER-LIST">
    <xs:list itemType="xs:integer"/>
  </xs:simpleType>
</xs:schema>
```

Validating the document of the previous section against the above schema yields the data model presented in the previous section.

The above schema contains one element declaration, three complex type declarations, and one simple type declaration.

- The `BOOKS` element has type `BOOKS-TYPE`.
- The `BOOKS-TYPE` type contains a sequence of zero or more elements with name `BOOK` of type `BOOK-TYPE`.
- The `BOOK-TYPE` type contains a sequence consisting of:
  - A `TITLE` element of type string.
  - One or more `AUTHOR` elements of type string.
  - An optional `REVIEW` element of type `INLINE`.
  - An optional `YEAR` attribute of type `INTEGER-LIST`.
- The `INLINE` type contains text nodes and any number of either `EM` or `BOLD` elements, both of which themselves have type `INLINE`.
- The `INTEGER-LIST` type contains a sequence of (zero or more) integers.

The XQuery formal specification includes an alternative notation for representing schemas, which is more readable, more compact, and more uniform. The above schema is written in this notation as follows.

```
define element BOOKS of type BOOKS-TYPE
define type BOOKS-TYPE {
  element BOOK of type BOOK-TYPE *
}
define type BOOK-TYPE {
  attribute YEAR of type INTEGER-LIST ? ,
  element AUTHOR of type xs:string + ,
  element TITLE of type xs:string ,
  element REVIEW of type INLINE ?
```

```
}
define type INLINE mixed {
  ( element EM of type INLINE |
    element BOLD of type INLINE ) *
}
define type INTEGER-LIST {
  xs:integer *
}
```

The formal semantics notation utilizes a number of conventions familiar from regular expressions: comma (`,`) for sequencing, bar (`|`) for alternation, query (`?`) for optional, plus (`+`) for one or more, star (`*`) for zero or more. In the above, the line `element BOOK of type BOOK-TYPE *` is parsed as `(element BOOK of type BOOK-TYPE) *`.

The formal semantics notation is more uniform than Schema notation. Schema uses `minOccurs` and `maxOccurs` to indicate whether an element is optional or repeated, uses `optional` or `required` to indicated whether an attribute is optional, and use `list` to indicate that a value of simple type is repeated. The formal semantics notation uses regular expression occurrence indicators (`?`, `+`, `*`) for all these purposes.

There is an older way of specifying the structure of XML documents, called a Document Type Definition (DTD), which is part of the original XML specification [4, 5]. There are also a number of alternative proposals for specifying the structure and datatypes of XML documents, notably Relax NG [13, 14]. Both DTDs and the compact syntax for Relax NG use a regular expression notation similar to that in the XQuery formal specification.

5 Projection

Here is a query that lists all authors of all books.

```
  /BOOKS/BOOK/AUTHOR
⇒
  <AUTHOR>Abiteboul</AUTHOR>,
  <AUTHOR>Buneman</AUTHOR>,
  <AUTHOR>Suciu</AUTHOR>,
  <AUTHOR>Buneman</AUTHOR>
∈
  element AUTHOR of type xsd:string *
```

This follows the style we will use to list a query, it's result, and the static type inferred for its result.

There is a second way to express the same query using explicit iteration.

```
  /BOOKS/BOOK/AUTHOR
=
  let $root := / return
```

```
    for $dot1 in $root/BOOKS return
      for $dot2 in $dot1/BOOK return
        $dot2/AUTHOR
```

Note that an associate law applies to both notations. For the XPath slash notation we have:

```
BOOKS/(BOOK/AUTHOR)  =  (BOOKS/BOOK)/AUTHOR
```

And for the `for` notation we have:

```
  for $dot1 in $root/BOOKS return
    for $dot2 in $dot1/BOOK return
      $dot2/AUTHOR
=
  for $dot2 in (
    for $dot1 in $root/BOOKS return
      $dot1/BOOK
  ) return
    $dot2/AUTHOR
```

6 Selection

Here is a query that lists titles of all books published before 2000.

```
  /BOOKS/BOOK[@YEAR < 2000]/TITLE
⇒
  <TITLE>Data on the Web</TITLE>
∈
  element TITLE of type xs:string *
```

Note that the `@YEAR` attribute is bound to a sequence of integers, and that the expression `@YEAR < 2000` returns true if some integer in the sequence is smaller that 2000.

Again, there is a second way to express the same query.

```
  /BOOKS/BOOK[@YEAR < 2000]/TITLE
=
  for $book in /BOOKS/BOOK
  where $book/@YEAR < 2000
  return $book/TITLE
```

The `where` clause in the above may be re-expressed as a conditional.

```
  for $book in /BOOKS/BOOK
  where $book/@YEAR < 2000
  returns $book/TITLE
=
  for $book in /BOOKS/BOOK returns
    if $book/@YEAR < 2000 then $book/TITLE else ()
```

There is also a second way to express the comparison, which makes the existential explicit.

```
  $book/@YEAR < 2000
=
  some $year in $book/@YEAR satisfies $year < 2000
```

The existential can itself be expressed in terms of iteration and selection.

```
  some $year in $book/@YEAR satisfies $year < 2000
=
  not(empty(
    for $year in $book/@YEAR where $year < 2000 returns $year
  ))
```

Combining all the previous laws allows one to expand the original expression into a larger expression in a smaller language.

```
  /BOOKS/BOOK[@YEAR < 2000]/TITLE
=
  let $root := / return
    for $books in $root/BOOKS return
      for $book in $books/BOOK return
        if (
          not(empty(
            for $year in $book/@YEAR returns
              if $year < 2000 then $year else ()
          ))
        ) then
          $book/TITLE
        else
          ()
```

7 Static typing issues

The static type associated with an expression may be too broad or too narrow.

Here is a query to list all books with the title "Data on the Web".

```
  /BOOKS/BOOK[TITLE = "Data on the Web"]
⇒
  <BOOK YEAR="1999 2003">
    <AUTHOR>Abiteboul</AUTHOR>
    <AUTHOR>Buneman</AUTHOR>
    <AUTHOR>Suciu</AUTHOR>
    <TITLE>Data on the Web</TITLE>
    <REVIEW>A <EM>fine</EM> book.</REVIEW>
  </BOOK>
∈
  element BOOK of type BOOK-TYPE *
```

Here the inferred type is too broad. It indicates that there will be zero or more books, when in fact one might expect that there should be at most one book with a given title; or even exactly one book, if we know we have supplied a valid title. Understanding how to exploit information about keys and foreign keys in the type system is an important open issue.

When the statically inferred type is too broad, it may be narrowed using a "treat as" expression.

```
  treat as element BOOK of type BOOK-TYPE (
    /BOOKS/BOOK[TITLE = "Data on the Web"]
  )
∈
  element BOOK of type BOOK-TYPE
```

The purpose served by "treat as" expressions in XQuery is similar to that served by casting in languages such as Java and C++.

For convenience, there is also a built-in function that indicates that a result sequence will have length one.

```
  one(/BOOKS/BOOK[TITLE = "Data on the Web"])
∈
  element BOOK of type BOOK-TYPE
```

This allows the type to be inferred, rather than requiring all the type information to be repeated. There are three similar convenience functions: `one()`, `zeroOrOne()`, `oneOrMore()`.

The type associated with an iteration may also be broader than you might expect. Say we define two different elements to represent books supplied by two different vendors, and a catalogue containing all books from the first vendor followed by all books from the second.

```
  define element AMAZON-BOOK of type BOOK-TYPE
  define element BN-BOOK of type BOOK-TYPE
  define element CATALOGUE of type CATALOGUE-TYPE
  define type CATALOGUE-TYPE {
    element AMAZON-BOOK * , element BN-BOOK*
  }
```

Here is a query to list all books in the catalogue with Buneman as an author.

```
  for $book in (/CATALOGUE/AMAZON-BOOK, /CATALOGUE/BN-BOOK)
  where $book/AUTHOR = "Buneman"
  return $book
∈
  ( element AMAZON-BOOK | element BN-BOOK )*
⊈
  element AMAZON-BOOK *, element BN-BOOK *
```

The typing rule for iteration assumes that the type of the bound variable is an alternation of elements. Here, the bound variable `$book` is given type

```
  element AMAZON-BOOK | element BN-BOOK
```

and hence the type of the iteration is as shown. This loses the information that all of the books from the first vendor will preceed books from the second vendor. If this information is important, it may be recovered by use of a suitable "treat as" expression, which will test at run-time that the value has the expected structure.

```
  treat as type CATALOGUE-TYPE (
    for $book in (/CATALOGUE/AMAZON-BOOK, /CATALOGUE/BN-BOOK)
    where $book/AUTHOR = "Buneman"
    return $book
  )
∈
  element AMAZON-BOOK * , element BN-BOOK *
```

The best trade-off between simplicity in the definition of iteration and accuracy of the inferred types is an important open issue.

8 Construction

Here is a query to list the year and title of all books published before 2000.

```
  for $book in /BOOKS/BOOK
  where $book/@YEAR < 2000
  return <BOOK>{ $book/@YEAR, $book/TITLE }</BOOK>
⇒
  <BOOK YEAR="1999 2003">
    <TITLE>Data on the Web</TITLE>
  </BOOK>
∈
  element BOOK {
    attribute YEAR { integer+ } ,
    element TITLE { string }
  } *
```

XQuery actually provides two notations for element and attribute construction. The "physical" notation looks like XML, the "logical" notation more accurately reflects the tree structure.

```
  <BOOK>{ $book/@YEAR , $book/TITLE }</BOOK>
=
  element BOOK { $book/@YEAR , $book/TITLE }
```

The XML-like notation nests arbitrarily deep, allowing brackets to splice-in values or nodes inside attributes or elements.

```
  <BOOK YEAR="{ data($book/@YEAR) }">
    <TITLE>{ data($book/TITLE) }</TITLE>
```

```
  </BOOK>
=
  element BOOK {
    attribute YEAR { data($book/@YEAR) },
    element TITLE { data($book/TITLE) }
  }
```

The logical notation provides a way to construct an attribute in isolation, which is not possible in the physical notation.

```
  for $book in /BOOKS/BOOK
  return
   <BOOK>
      if empty($book/@YEAR) then
        attribute YEAR  2000
      else
        $book/@YEAR ,
      $book/title
   </BOOK>
```

The logical notation also provides a way to compute the name of an element or attribute, which will be demonstrated in Section 14.

9 Grouping

A common operation for databases is grouping. In the relational world, this often requires special support, such as the "group by" clause in SQL. The nested structure of XQuery supports grouping naturally.

Here is a query that lists for each author the titles of all books by that author.

```
  for $author in distinct(/BOOKS/BOOK/AUTHOR) return
    <AUTHOR NAME="{ $author }">{
      /BOOKS/BOOK[AUTHOR = $author]/TITLE
    }</AUTHOR>
⇒
  <AUTHOR NAME="Abiteboul">
    <TITLE>Data on the Web</TITLE>
  </AUTHOR>,
  <AUTHOR NAME="Buneman">
    <TITLE>Data on the Web</TITLE>
    <TITLE>XML in Scotland</TITLE>
  </AUTHOR>,
  <AUTHOR NAME="Suciu">
    <TITLE>Data on the Web</TITLE>
  </AUTHOR>
```

Grouping provides another example where the inferred type may be too broad.

```
  for $author in distinct(/BOOKS/BOOK/AUTHOR) return
    <AUTHOR NAME="{ $author }">{
      /BOOKS/BOOK[AUTHOR = $author]/TITLE
    }</AUTHOR>
∈
  element AUTHOR {
    attribute NAME { string },
    element TITLE { string } *
  }
⊈
  element AUTHOR {
    attribute NAME { string },
    element TITLE { string } +
  }
```

As before, this may be fixed using a "treat as" expression, or using the convenience function `oneOrMore()`.

10 Join

Another common operation for databases is to join data from two relations. Indeed, efficient expression and optimization of joins is central to the popularity and power of databases.

Here is a revised type declaration for books.

```
  define element BOOKS {
    element BOOK *
  }
  define element BOOK {
    element TITLE of type xs:string ,
    element PRICE of type xs:decimal ,
    element ISBN of type xs:string
  }
```

Assume that Amazon and Barnes and Noble make available data in this format. Here is a query that lists all books that are more expensive at Amazon than at Barnes and Noble.

```
  let $amazon := document("http://www.amazon.com/books.xml"),
      $bn := document("http://www.BN.com/books.xml")
  for $a in $amazon/BOOKS/BOOK,
      $b in $bn/BOOKS/BOOK
  where $a/ISBN = $b/ISBN
    and $a/PRICE > $b/PRICE
  return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
```

(Because it will be easy to formulate such queries, it may be a while before vendors make data available in such formats.)

If a similar query was formulated for a relational database, it might be implemented by sorting the Amazon books and the Barnes and Noble books in order of ISBN, then merging the resulting lists and checking the prices. It is difficult to apply this optimization to the query above because order is significant in XQuery. The way in which the query is written specifies that the books should be presented in the same order that they appear in the Amazon database. Reversing the two "for" clauses would specify that they should be in the same order as in the Barnes and Noble database.

In fact, the user may not care about the order in which the results are computed, and may wish to give the XQuery implementation flexibility to choose an order that can be computed efficiently. This may be specified by using the `unordered` expression.

```
unordered (
  for $a in $amazon/BOOKS/BOOK,
      $b in $bn/BOOKS/BOOK
  where $a/ISBN = $b/ISBN
    and $a/PRICE > $b/PRICE
  return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
)
```

In general, the expression `unordered` *Expr* may return any permutation of the sequence returned by *Expr*.

Often, the user wants the result to be sorted in a particular order. In the query above, one may want the answer to be sorted with the titles in alphabetic order.

```
for $a in $amazon/BOOKS/BOOK,
    $b in $bn/BOOKS/BOOK
where $a/ISBN = $a/ISBN
  and $b/PRICE > $b/PRICE
order by $a/TITLE
return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
```

Whenever a sequence is sorted, the order of the original sequence is irrelevant (unless the sort is required to be stable). Opportunities for optimization can be expressed by introducing unordered expressions, and pushing such expressions into the computation.

```
for $a in $amazon/BOOKS/BOOK,
    $b in $bn/BOOKS/BOOK
where $a/ISBN = $a/ISBN
  and $b/PRICE > $b/PRICE
order by $a/TITLE
return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
```

=

```
  for $x in
    unordered(
      for $a in $amazon/BOOKS/BOOK,
          $b in $bn/BOOKS/BOOK
      where $a/ISBN = $a/ISBN
        and $b/PRICE > $b/PRICE
      return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
    )
  order by $x/TITLE
  return $x
=
  for $x in
    unordered (
      for $a in unordered( $amazon/BOOKS/BOOK ),
          $b in unordered( $bn/BOOKS/BOOK )
      where $a/ISBN = $a/ISBN
        and $b/PRICE > $b/PRICE
      return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
    )
  order by $x/TITLE
  return $x
```

For some queries that compute a join over a database it is desirable to include some data that does not appear in both relations. In SQL this is called a "left outer join", and SQL includes special statements to support computing such joins. In XQuery, this may be specified using operations that we have already discussed.

Here is a query that lists all books available from Amazon and from Barnes and Noble, followed by all books available from Amazon only.

```
  for $a in $amazon/BOOKS/BOOK, $b in $bn/BOOKS/BOOK
  where $a/ISBN = $b/ISBN
  return <BOOK>{ $a/TITLE, $a/PRICE, $b/PRICE }</BOOK>
  ,
  for $a in $amazon/BOOKS/BOOK
  where not($a/ISBN = $bn/BOOKS/BOOK/ISBN)
  return <BOOK>{ $a/TITLE, $a/PRICE }</BOOK>
∈
  element BOOK { TITLE, PRICE, PRICE } *
  ,
  element BOOK { TITLE, PRICE } *
```

11 Nulls and three-valued logic

We don't always know everything: sometimes data is missing. In recognition of this SQL supports a special "null" value. In XML, one may support missing data

by simply making the associated element or attribute optional. (XML Schema also supports a special `xsi:nil` attribute, but we won't go into that here.)

The arithmetic operations of XQuery are designed to make it easy to operate on potentially missing data. In XQuery the arithmetic operations expect each argument to be either a number or the empty sequence, and if either argument is the empty sequence then the result is the empty sequence. The design is motivated, in part, by a desire to mimic the behaviour of arithmetic operators in SQL when passed null data.

Here is yet another set of declarations for books.

```
define element BOOKS { element BOOK * }
define element BOOK {
  element TITLE of type xs:string ,
  element PRICE of type xs:decimal ,
  element SHIPPING of type xs:decimal ?
}
```

Here is some data matching the above.

```
<BOOKS>
  <BOOK>
    <TITLE>Data on the Web</TITLE>
    <PRICE>40.00</PRICE>
    <SHIPPING>10.00</PRICE>
  </BOOK>
  <BOOK>
    <TITLE>XML in Scotland</TITLE>
    <PRICE>45.00</PRICE>
  </BOOK>
</BOOKS>
```

Here is a query that lists all books with total cost $50.00.

```
for $book in /BOOKS/BOOK
where $book/PRICE + $book/SHIPPING = 50.00
return $book/TITLE
⇒
<TITLE>Data on the Web</TITLE>
```

If the shipping is missing, then the total cost is unknown, and hence cannot be equal to $50.00. That is, we have `45.00 + ()` ⇒`()` and `() = 50.00` ⇒`false()`.

For convenience, there is a function `ifAbsent(`x`,`y`)` that makes it easy to supply a default value. This function returns the value of x, unless x is the empty sequence, in which case it returns y.

Here is a query that lists all books with total cost $50.00, where a missing shipping cost is assumed to be $5.00.

```
for $book in /BOOKS/BOOK
```

```
  where $book/PRICE + ifAbsent($book/SHIPPING, 5.00) = 50.00
  return $book/TITLE
⇒
  <TITLE>Data on the Web</TITLE>,
  <TITLE>XML in Scotland</TITLE>
```

12 Type errors

When evaluating a type system, it is instructive to examine not only those programs that pass the type checker but also those that fail it. What errors does the type checker catch?

For this section, it will be helpful to consider a series of similar type declarations. All of the definitions presume an element that contains a sequence of books.

```
define element BOOKS { element BOOK * }
```

For each example we will define the type of book elements, and also possibly a type of answer elements.

One common kind of error is to select an element or attribute that is not present. This can happen through misunderstanding or misspelling.

Say we define a book to contain a title and an optional price.

```
define element BOOK {
  element TITLE of type xs:string ,
  element PRICE of type xs:decimal ?
}
```

Here is a query that lists the title and ISBN number of each book.

```
for $book in /BOOKS/BOOK return
  <ANSWER>{ $book/TITLE, $book/ISBN }</ANSWER>
∈
element ANSWER {
  element TITLE of type xs:string
} *
```

This is not a sensible query, because book is defined to contain a title and a price, not an ISBN element.

The usual reason for reporting a type error is that evaluation of the expression may go wrong, for instance, by adding an integer to a string. Here "wrong" is used in the technical sense of Milner's motto: "Well-typed programs do not go wrong". Achieving this requires a careful definition of "wrong": we do not define queries that divide by zero or fail in a "treat as" expression as wrong.

But the expression above is not wrong in this sense! The semantics of the XPath expression `$book/ISBN` is perfectly well-defined, it returns the empty sequence. Similarly, `$book/PRICE` returns the empty sequence whenever the optional price is absent.

Nonetheless, it is possible to issue a warning. The computation may not be wrong, but it is wrong-headed. Type inference shows that in this case the type of the expression `$book/ISBN` is `()`, the type of the empty sequence. Why would one ever want to write an expression that always evaluates to the empty sequence? There is one such expression that is useful, namely the expression `()` itself. But any other expression that has type `()` is likely to be an error. Note that the expression `$book/PRICE` does not have type `()`, because the type indicates that a price may be present, and so there would be no warning in that case. The idea of issuing warnings for expressions with empty type appears to be new with XQuery.

(Why make it a warning rather than an error? Because there are circumstances where it might be reasonable to write such an expression. For instance, a single query may be used against several different data sources with different schemas. For some schemas an expression might have the empty type, while for other schemas the same expression might have a non-empty type.)

In many circumstances, types will be declared for both the input and the output of a query. In this case, the error will be caught even if the mechanism of issuing warnings is not in effect. Here are input and output type declarations.

```
define element BOOK {
  element TITLE of type xs:string ,
  element PRICE of type xs:decimal
}
define element ANSWER {
  element TITLE of type xs:string ,
  element ISBN of type xs:string
}
```

Here is the same query as before, modified to explicitly validate its output.

```
for $book in /BOOKS/BOOK return
  validate {
    <ANSWER>{ $book/TITLE, $book/ISBN }</ANSWER>
  }
```

This will report a static error, because the type declared for an answer element requires it to have both a title and an ISBN number, while the type infrerred shows that it has only a title.

Of course, the type system also catches errors when an expression does go wrong, for instance, by adding a boolean to a number. Say that the type for books is declared as follows.

```
define element BOOK {
  element TITLE of type xs:string ,
  element PRICE of type xs:decimal ,
  element SHIPPING of type xs:boolean ,
  element SHIPCOST of type xs:decimal ?
}
```

Here is a query that lists the total cost of a book by adding the price and the shipping.

```
for $book in /BOOKS/BOOK return
  <ANSWER>{
     $book/TITLE,
     <TOTAL>{ $book/PRICE + $book/SHIPPING }</TOTAL>
  }</ANSWER>
```

Here the author of the query has gotten confused: the `SHIPPING` element contains not the cost of shipping (that is in `SHIPCOST`), but a boolean indicating whether shipping charges apply. In this case, the expression may go wrong, and a static type error occurs in the usual way.

As explained in the previous section, arithmetic operators are specially designed to accommodate null data. If the query writer has forgotten that some element or attribute is optional, this may again yield a wrong-headed result without the query actually going wrong. Such errors can often be detected if a declaration is also provided for the output.

Say that the type of input and output is as follows.

```
define element BOOK {
  element TITLE of type xs:string ,
  element PRICE of type xs:decimal ,
  element SHIPPING of type xs:decimal ?
}
define element ANSWER {
  element TITLE of type xs:string ,
  element TOTAL of type xs:decimal
}
```

Here is a query that lists the title and total cost of each book.

```
for $book in /BOOKS/BOOK return
  validate {
    <ANSWER>{
      $book/TITLE,
      <TOTAL>{ $book/PRICE + $book/SHIPPING }</TOTAL>
     }</ANSWER>
  }
```

This time the shipping cost is kept in an element called `SHIPPING`, but the cost is optional. If it is not present, the sum yields an empty sequence. However, the error can be detected because a type has been declared for the answer, and this type requires that the `TOTAL` element contains a decimal, and that this is required not optional.

13 Functions

Here is a function that takes a book and returns the shipping cost.

```
define element BOOK {
  element TITLE of type xs:string ,
  element PRICE of type xs:decimal ,
  element SHIPPING of type xs:decimal ?
}
define function cost (element BOOK $book)
returns xs:decimal ? {
  $book/PRICE + $book/SHIPPING
}
```

14 Recursion

XML data may have a recursive structure. Such structures may be processed using recursive functions.

Here are declarations for a recursive part hierarchy.

```
define element PART {
  attribute NAME of type xs:string &
  attribute COST of type xs:decimal ,
  element PART *
}
```

Here is some data. The costs are incremental, that is, the cost of assembling the subparts to yield the part.

```
<PART NAME="system" COST="500.00">
  <PART NAME="monitor" COST="1000.00"/>
  <PART NAME="keyboard" COST="500.00"/>
  <PART NAME="pc" COST="500.00">
    <PART NAME="processor" COST="2000.00"/>
    <PART NAME="dvd" COST="1000.00"/>
  </PART>
</PART>
```

Here is a function that computes new values for the cost attribute of each part. The new value is the total cost.

```
define function total (element PART $part)
returns element PART {
  let $subparts := $part/PART/total(.)
  return
    <PART NAME="$part/@NAME"
          COST="$part/@COST + sum($subparts/@COST)">{
       $subparts
     }</PART>
}
```

Here is the result of applying the function to the given data.

```
  total(/PART)
⇒
  <PART NAME="system" COST="5000.00">
    <PART NAME="monitor" COST="1000.00"/>
    <PART NAME="keyboard" COST="500.00"/>
    <PART NAME="pc" COST="3500.00">
      <PART NAME="processor" COST="2000.00"/>
      <PART NAME="dvd" COST="1000.00"/>
    </PART>
  </PART>
```

15 Wildcards and computed names

Here is a function that swaps all attributes and elements within an element.

```
  define function swap(element $e) returns element {
    element { name($e) } {
      for $x in $e/* return
        attribute { name($x) } { data($x) } ,
      for $x in $e/@* return
        element { name($x) } { data($x) } ,
    }
  }
```

This function uses the XPath wildcards `$e/*` to select all elements in `$e` and `$e/@*` to select all attributes in `$e`, and the wildcard type `element` which denotes the type of all elements. For example,

```
  swap( <BOOK YEAR="2003"><TITLE>XQuery</TITLE></BOOK> )
⇒
  <BOOK TITLE="XQuery"><YEAR>2003</YEAR></BOOK>
∈
  element
```

References

1. XQuery 1.0: An XML Query Language. W3C Working Draft, 30 April 2002. (Next draft expected August 2002.)
2. XQuery 1.0 Formal Semantics. W3C Working Draft, 26 March 2002. (Next draft expected August 2002.)
3. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft, 30 April 2002. (Next draft expected August 2002.)
4. Extensible Markup Language (XML) 1.0. W3C Recommendation, 10 February 1998.
5. Extensible Markup Language (XML) 1.0 (Second edition). W3C Recommendation, 6 October 2000.

6. Namespaces in XML. W3C Recommendation, 14 January 1999.
7. XML Schema Part 0: Primer, Part 1: Structures, Part 2: Datatypes. W3C Recommendation, 2 May 2001.
8. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 16 November 1999.
9. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, 30 April 2002.
10. XML Path Language (XPath) 1.0. W3C Recommendation, 16 November 1999.
11. XML Path Language (XPath) 2.0. W3C Working Draft, 30 April 2002.
12. XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation, 11 September 2001.
13. RELAX NG Specification. Oasis Committee Specification, 3 December 2001.
14. RELAX NG Compact Syntax. Oasis Working Draft, 7 June 2002.