Information Systems

WS 2005, JKU Linz
Course 12: XQuery

Gábor Bodnár

URL: `http://www.risc.uni-linz.ac.at/education/courses/ws2005/is/`

Overview

- Data models and Types
- Expressions
- Query prolog

Generalities on XQuery

The purpose of XQuery is to provide a language for extracting data from XML documents.

Queries can operate on more than one documents at once. Subsets of nodes can be selected using XPath expressions.

The query language is functional (but it also includes universal and existential quantifiers), supports simple and complex data types defined in XML Schema.

Just as in XSLT, the expressions play the central role in XQuery.

A The value of an expression is always a sequence, having some sequence type.

Data Model

The data model of XQuery is an extended version of the data model of XPath.

Roots of documents are called 'document nodes' (cf. root node in XPath).

The data describing each node can be retrieved by functions implementing prescribed interfaces, called *accessors*. For instance

- `dm:node-kind($n as node()) as xs:string` prescribes an interface to obtain the node type (e.g. 'element', 'attribute', etc).

- `dm:string-value($n as node()) as xs:string` prescribes an interface to obtain the string value of the node.

Type Annotation

XQuery is a strongly typed language, using all types defined in XSchema.

Type information can be assigned to element or attribute nodes by

- using the post schema validation infoset (PSVI),
- using the `xdt:untypedAny` type for element nodes and the `xdt:untypedAtomic` type for attribute nodes

The typed value of a node can be extracted by applying the `fn:data` function on the node, which corresponds to the `dm:typed-value` accessor.

Typed Value Computation

There is a well-defined protocol that allows the XQuery processor to determine the typed value of an object. Just an excerpt:

- The typed value of an attribute: If the type annotation `xdt:untypedAtomic`, its string value as an instance of this type. For other type annotation it is derived from the string value in a way consistent with the schema validation.
- The typed value of a comment or processing instruction node is the same as its string value, as an instance of the type `xs:string`.
- For text, document, and namespace nodes, the typed value of the node is the same as its string value, as an instance of the type `xdt:untypedAtomic`.

Typed Value Computation

- If the element has a type of `xdt:untypedAtomic` or a complex type with mixed content, the typed value is the string value of the node as an instance of `xdt:untypedAtomic`.
- For elements with simple type or a complex type with simple content: the typed value is a sequence of zero or more atomic values derived from the string value of the node and its type in a way that is consistent with the schema validation.
- If the node has a complex type with empty content, the typed value is the empty sequence: ().
- If the node has a complex type with element only complex content, its typed value is undefined.

Expression Type Annotation

Expressions are evaluated in two phases, called *static analysis* and *dynamic evaluation*.

Since expressions always have sequence values, we have to deal with *sequence types*. A few examples are:

- `xs:date` refers to the built-in Schema type `date`,
- `attribute()?` refers to an optional attribute,
- `element()` refers to any element,
- `node()*` refers to a sequence of zero or more nodes of any type.

Basic Expressions

- *Primary expressions* include literals (`"3.14"`), variable references (`$x`), function calls (`fn:count(book/author)`) and constructors.
- *Path expressions* were discussed last time in context of XPath (`/faculty/*[@type="institute"]/head/email`).
- *Sequence expressions* can be formed via the comma operator (`((1,(2,3))` results `(1,2,3)`).
- *Arithmetic expressions* are built from numerical atomic values using `+, -, *, div, idiv, mod`.

Basic Expressions

- *Comparison expressions* can be value comparisons (`eq`, `ne`, `lt`, `le`, `gt`, `ge`), general comparisons (`=`, `!=`, `<`, `<=`, `>`, `>=`) and node comparisons (`is`, `<<`, `>>`).

- *Logical expressions* are built from boolean atomic values using `and`, `or`.

Constructors

Constructors are provided for every kind of node types to create XML structures in a query.

Direct constructors resemble to generating literal elements in XSLT with dynamic substitution using $\{\}$. For example, if `$b` is

```
<book isbn="0-812909191-5">
  <title>Codenotes for XML</title>
  <author>G. Brill</author>
</book>

<p id="{$b/@isbn}">Book:<br/>
{string($b/title)}, {string($b/author)} [{string($b/@isbn)}]</p>

<p id="0-812909191-5">Book:<br/>
Codenotes for XML, G. Brill [0-812909191-5]</p>
```

Constructors

Computed constructors allow to set the element names dynamically via XQuery expressions. The computed version of the previous direct constructor is

```
element p {
  attribute id {$b/@isbn},
  "Book:", element br {},
  string($b/title), ",", string($b/author), ", [",
  string($b/@isbn), "]"
}
```

Exercise

Let $s be bound to the msgs element in

```
<system>
 <stamp>12-03-02 23:13</stamp>
 <msgs>
  <msg type="info">System started</msg>
  <msg type="info">Logging in user 'maryk'</msg>
  <msg type="warn">User 'bobm' not found</msg>
 </msgs>
</system>
```

What element does the following constructor create?

```
element {name($s/..)} {
  attribute stamp { $s/../stamp },
  <second>{string($s/msg[position()=2])}</second>
}
```

FLWOR Expressions

The acronym is an abbreviation of `for`, `let`, `where`, `order by`, `return`.

The expression constructs a "tuple stream" (list of tuples of variable bindings) filters it by the given condition and evaluates the returning expression for each of them in the order requested.

The variables in the `for` clause will iterate on the values in the sequences specified for them.

In usual imperative languages (C,Perl) an n-fold nested loop would be analogous to n given variables in a FLWOR expression.

The variables in the `let` clause will be bound to the given sequences.

Example

```
for $d in fn:doc("depts.xml")//deptno
let $e := fn:doc("emps.xml")//emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
  <big-dept>
    {$d,
    <headcount>{fn:count($e)}</headcount>,
    <avgsal>{fn:avg($e/salary)}</avgsal>}
  </big-dept>
```

Conditional Expressions

```
<result>
  { for $u in doc("users.xml")//user_tuple
    let $b := doc("bids.xml")//bid_tuple[userid =
      $u/userid]
    order by $u/userid
    return <user>
            { $u/userid }
            { $u/name }
            { if (empty($b))
                 then <status>inactive</status>
                 else <status>active</status>}
        </user>}
</result>
```

Quantified Expressions

General structure

every | some variable in expr
      [, variable in expr]* satisfies expr

Example

```
<frequent_bidder>
 {for $u in doc("users.xml")//user_tuple
  where
   every $item in doc("items.xml")//item_tuple satisfies
    some $b in doc("bids.xml")//bid_tuple satisfies
     ($item/itemno = $b/itemno and $u/userid = $b/userid)
  return $u/name}
</frequent_bidder>
```

Query Prolog

The prolog is a semicolon separated series of declarations (of variables, namespaces, functions, etc.) and imports (of schemas, modules) that create the environment for query processing.

A *namespace declaration* binds a namespace prefix to a namespace URI:

```
declare namespace x = "http://www.my-company.com/";
```

A *schema import* adds a named schema to the in-scope schema definitions:

```
import schema namespace
   xhtml="http://www.w3.org/1999/xhtml"
   at "http://example.org/xhtml/xhtml.xsd";
```

Query Prolog

A *variable declaration* adds a variable binding to the in-scope variables:

```
declare variable $x as xs:integer {7};
declare variable $x as xs:integer external;
```

A *function declaration* adds a user defined or external function to the available in-scope functions.

XQuery predefines the prefix `local` to the namespace `http://www.w3.org/2003/11/xquery-local-functions`.

Example

```
declare function
   local:summary($emps as element(employee)*)
   as element(dept)*
{
   for $d in fn:distinct-values($emps/deptno)
   let $e := $emps[deptno = $d]
   return
      <dept>
         <deptno>{$d}</deptno>
         <headcount> {fn:count($e)} </headcount>
         <payroll> {fn:sum($e/salary)} </payroll>
      </dept>
};
```

Example

This application of the previous function computes a summary of employees in Denver.

```
local:summary(fn:doc("acme_corp.xml")//employee[location
  = "Denver"])
```

Summary

- Data model and Types
- Typed values and Expression type annotation
- Basic expressions
- Constructors and FLWOR expressions
- Conditional and Quantified expressions
- Query prolog