XQuery 1.0 Formal Semantics

W3C Working Draft 07 June 2001

This version:
http://www.w3.org/TR/2001/WD-query-semantics-20010607
Latest version:
http://www.w3.org/TR/query-semantics/
Previous versions:
http://www.w3.org/TR/2001/WD-query-algebra-20010215/
http://www.w3.org/TR/2000/WD-query-algebra-20001204/
Editors:
Peter Fankhauser (GMD-IPSI) <fankhaus@darmstadt.gmd.de>
Mary Fernández (AT&T Labs - Research) <mff@research.att.com>
Ashok Malhotra (Microsoft) <ashokma@microsoft.com>
Michael Rys (Microsoft) <mrys@microsoft.com>
Jérôme Siméon (Bell Labs, Lucent Technologies) <simeon@research.bell-labs.com>
Philip Wadler (Avaya) <wadler@avaya.com>

Abstract

This document presents the formal semantics of [XQuery 1.0: A Query Language for XML], an XML query language. This document replaces the [XML Query Algebra].

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This is a First Public Working Draft for review by W3C Members and other interested parties. This document replaces the [XML Query Algebra]. It is a draft document and may be updated, replaced or made obsolete by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". This is work in progress and does not imply endorsement by the W3C membership.

This document has been produced as part of the W3C XML Activity, following the procedures set out for the W3C Process. The document has been written by the XML Query Working Group.

The purpose of this document is to present the current state of the formal semantics of [XQuery 1.0: A Query Language for XML] and to elicit feedback on its current state. The XML Query Working Group feels that it has made good progress on this document but that it is subject to change in future versions. Comments on this document should be sent to the W3C mailing list www-xml-query-comments@w3.org (archived at http://lists.w3.org/Archives/Public/www-xml-query-comments/). Important issues remain open - see **[B.3.1 Open Issues]**. In particular, the reader should note the following issues related to compatibility of the XQuery formal semantics with related XML activities.

- **[Issue-0089: Syntax for types in XQuery]**: The XQuery formal semantics's is based on a subset of the XQuery surface syntax, but some misalignments exist. The XQuery formal semantics presents a syntax for type expressions that is not supported in the XQuery surface syntax. It also has a static type-assertion expression (see **[Issue-0090: Static type-assertion expression]**), an attribute constructor expression (see **[Issue-0091: Attribute expression]**), and an error expression (see **[Issue-0092: Error expression]**) that are not in the XQuery surface syntax

- **[Issue-0088: Align XQuery types with XML Schema : Formal Description.]**: The XQuery formal semantics's type system is based on [XML Schema : Formal Description] (XSFD), but some misalignments exist. A related issue is **[Issue-0018: Align algebra types with schema]**. We assume that the XQuery formal semantics will be based on XSFD and leave alignment of XSFD and XML Schema for others to resolve.

- **[Issue-0056: Operators on Simple Types]**: A joint XSLT/Schema/Query task force is chartered to define the operators on Schema simple types. XQuery will adopt the operators defined by that group.

The XML Query Working Group is working diligently to achieve compatibility with these XML activities.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR/.

Table of contents

Appendices

1 Introduction

This document defines the formal semantics of XQuery, an XML query language. The formal semantics of XQuery is defined with respect to a ``core syntax'' of XQuery. XQuery's core syntax is based on a subset of the complete syntax that is available to users, and every expression in the user-level syntax can be rewritten as an expression in the core syntax. Although the intent is for the core syntax to be a proper subset of the complete XQuery syntax, some misalignments exist. The XQuery formal semantics presents a syntax for type expressions that is not supported in the XQuery surface syntax. It also has a static type-assertion expression (see **[Issue-0090: Static type-assertion expression]**), an attribute constructor expression (see **[Issue-0091: Attribute expression]**), and an error expression (see **[Issue-0092: Error expression]**) that are not in the XQuery surface syntax

A forthcoming document defines operators and a library of built-in functions for XQuery and XPath 2.0. In this document, functions in this library have the namespace prefix `xfo`.

In this document, ``query-analysis time'' refers to when an XQuery expression is parsed and type checked, that is before the value of the expression is computed, and ``query-evaluation time'' refers to when an XQuery expression is evaluated, that is, when it is reduced to a value. We sometimes use the phrases query-analysis time and ``compile time'' interchangeably as well as the phrases query-evaluation time and ``run time''.

This work builds on long-standing traditions in the database community. In particular, we have been inspired by systems such as SQL, OQL, and nested relational algebra (NRA). We have also been inspired by systems such as Quilt, UnQL, XDuce, XML-QL, XPath, XQL, XSLT, and YaTL. We give citations for all these systems below.

In the database world, it is common to translate a query language into an algebra; this happens in SQL, OQL, and NRA, among others. The purpose of the algebra is twofold. First, an algebra is used to give a semantics for the query language, so the operations of an algebra should be well-defined. Second, an algebra is used to support query optimization, so it should possess a rich set of laws. The core syntax of XQuery serves as an algebra for XQuery. The laws we give include analogues of most of the laws of relational algebra.

It is also common for a query language to exploit schemas or types; this happens in SQL, OQL, and NRA, among others. The purpose of types is twofold. Types can be used to detect certain kinds of errors at query-analysis time and to support query optimization. Given the type of input values, a query applied to those values, and the expected type of the query's output value, the XQuery type system can detect at query-analysis time if the query's output value has the expected output type.

DTDs and XML Schema can be thought of as providing something like types for XML. The XQuery formal semantics uses a a type system based on the formalism in [XML Schema : Formal Description]. On this basis, XQuery is statically typed. This allows an implementation of XQuery to determine and check at query-analysis time the output type of a query on documents conforming to an input type. Compare this to an untyped or dynamically typed query language, where each individual output has to be validated against a schema at query-evaluation time, and there is no guarantee that this check will always succeed.

To define the XQuery completely, we present a *static* semantics and a *dynamic* semantics. The static semantics is presented as type inference rules, which relate XQuery expressions to types and and specify under what conditions an expression is well typed. The semantics is static, because ill-typed expressions are identified at query-analysis time, i.e., before the query is evaluated. The dynamic, or operational, semantics is presented as value inference rules, which relate XQuery expressions to values. XQuery's values are defined in the [XQuery 1.0 and XPath 2.0 Data Model]; for example, they include XML simple values, attributes, and elements, as well as other values. A dynamic semantics guarantees that every expression can be reduced to a value and may serve as the basis for a query interpreter or compiler.

The document is organized as follows. A tutorial introduction is presented in [**2 XQuery Semantics by Example**]. The primary purpose of this tutorial to present various features of XQuery and to show how a type is computed for each XQuery expression. The reader is referred to [XQuery 1.0: A Query Language for XML] for a complete tutorial on XQuery's features. The grammar of XQuery's core syntax and the grammar for types are given in [**3 XQuery Core Syntax**]. We give the static typing rules for XQuery in [**4 Static Semantics : Type-Inference Rules**] and then the dynamic semantics for XQuery in [**5 Dynamic Semantics : Value-Inference Rules**]. These sections formalize the information presented informally in [**2 XQuery Semantics by Example**]. Although these two sections contain the most challenging material, we have tried to make the content as accessible as possible. Readers only interested in learning about XQuery's features need not read these sections, however, we expect that implementors of XQuery will read them. Finally, in [**6 XQuery Mapping to Core**], we present the mapping from complete syntax of XQuery to the core syntax. We note here that this section is still preliminary and contains inconsistencies (see [**Issue-0099: Incomplete/inconsistent mapping from XQuery to core**]).

In [**B.2 Issues list**], we discuss open issues and problems. We present some equivalence and optimization laws of XQuery in [**A Equivalences**].

Cited literature includes: monads [Mog89], [Mog91], [Wad92], [Wad93], [Wad95], NRA [BNTW95], [Col90], [LW97], [LMW96], OQL [BK93], [BKD90], [CM93], Quilt [Quilt], SQL [Date97], UnQL [BFS00], XDuce [HP2000], XMSL-QL [XMLQL99], XPath [XPath], XQL [XQL99], XSLT [XSLT 99], and YaTL [YAT99].

2 XQuery Semantics by Example

For a complete introduction to XQuery, see [XQuery 1.0: A Query Language for XML]. This document focuses on the static type and dynamic operational semantics of XQuery. This section introduces the static and dynamic semantics of XQuery, using examples based on accessing a database of books.

2.1 Data and types

Consider the following sample data:

```
<bib>
  <book year="1999" isbn="1-55860-622-X">
    <title>Data on the Web</title>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
  </book>
  <book year="2001" isbn="1-XXXXX-YYY-Z">
    <title>XML Query</title>
    <author>Fernandez</author>
    <author>Suciu</author>
  </book>
</bib>
```

Here is a fragment of an XML Schema for such data:

```
<xs:group name="Bib">
```

```
  <xs:element name="bib">
    <xs:complexType>
      <xs:group ref="Book"
                 minOccurs="0" maxOccurs="unbounded"/>
    </xs:complexType>
  </xs:element>
</xs:group>

<xs:group name="Book">
  <xs:element name="book">
    <xs:complexType>
      <xs:attribute name="year" type="xs:integer"/>
      <xs:attribute name="isbn" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author"type="xs:string" maxOccurs="unbounded"/>
    </xs:complexType>
  </xs:element>
</xs:group>
```

In the XQuery formal semantics, we present a syntax for type expressions that allows us to explain how the type of an XQuery expression is inferred. This type syntax is not in the XQuery surface syntax (see **[Issue-0089: Syntax for types in XQuery]**). In addition, the XQuery formal semantics includes an expression that asserts statically the type of an expression (see **[Issue-0090: Static type-assertion expression]**), an attribute constructor expression (see **[Issue-0091: Attribute expression]**), and an error expression (see **[Issue-0092: Error expression]**) that are not in the XQuery surface syntax. With the exception of type expressions and the static type-assertion expression, all other XQuery expressions in this document are in the XQuery surface syntax. The data and schema above is represented as follows:

```
TYPE Bib = ELEMENT bib (Book*)
TYPE Book =
  ELEMENT book
    ( ATTRIBUTE year  (xs:integer) &
      ATTRIBUTE isbn  (xs:string)
      ELEMENT   title (xs:string),
      (ELEMENT   author(xs:string))+
    )
LET $bib0 :=
<bib>
  <book year="1999" isbn="1-55860-622-X">
    <title>Data on the Web</title>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
  </book>
  <book year="2001" isbn="1-XXXXX-YYY-Z">
    <title>XML Query</title>
    <author>Fernandez</author>
    <author>Suciu</author>
  </book>),
</bib> : Bib
RETURN ...
```

The expression above defines two types, `Bib` and `Book`, and defines one variable, `$bib0`.

The `Bib` type corresponds to a single `bib` element, which contains a *sequence* of zero or more `Book` elements. Every attribute or element can be viewed as a sequence of length one.

The `Book` type corresponds to a single `book` element, which also contains a sequence of zero or more attributes and elements. It contains one `year` attribute and one `isbn` attribute, followed by one `title` element, followed by one or more `author` elements. The `&` operator, called the *interleave operator*, indicates that the `year` and `isbn` attributes may occur in any order. A `isbn` attribute and a `title` or `author` element contains a string value, and a `year` attribute contains an integer.

The `let` expression above binds the variable `$bib0` to a literal XML value. The variable `$bib0` is in scope for all expressions in the body of the `RETURN` clause. For convenience, the `RETURN ...` indicates that the expressions in the rest of this document are contained within the scope of this `LET` expression. The value of a variable is immutable, that is, once a variable is defined, its value does not change. The value of `$bib0` is a `bib` element that contains two `book` elements.

XQuery is a strongly typed language, therefore the value of `$bib0` must be an instance of its declared type, or the expression is ill-typed. Here the value of `$bib0` is an instance of the `Bib` type, because it contains one `bib` element, which contains two `book` elements, each of which contain an integer-valued `year` attribute, a string-valued `isbn` attribute, a string-valued `title` element, and one or more string-valued `author` elements.

For convenience, we define a second global variable `$book0` also bound to a literal value, which is equal to the first book in `bib0`.

```
LET $book0 :=
  <book year="1999" isbn="1-55860-622-X">
    <title>Data on the Web</title>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
  </book> : Book
RETURN ...
```

2.2 Projection

One of XQuery's most basic operations is projection. The following expression returns all `author` elements contained in `book` elements contained in `$bib0`:

```
    $bib0/book/author
==>(<author>Abiteboul</author>,
    <author>Buneman</author>,
    <author>Suciu</author>,
    <author>Fernandez</author>,
    <author>Suciu</author>)
:   (ELEMENT author (xs:string))*
```

Note that in the result, the document order of `author` elements is preserved.

The above example and the ones that follow have three parts. First is an expression in XQuery. Second, following the ==> is the value of this expression. Third, following the `:` is the type of the expression, which is (of course) also a legal type for the value.

It may be unclear why the type of `$bib0/book/author` contains *zero* or more authors, even though the type of a `book` element contains *one* or more authors. Let's look at the derivation of the result type by looking at the type of each sub-expression:

```
$bib0             : Bib
$bib0/book        : Book*
$bib0/book/author : (ELEMENT author (xs:string))*
```

Recall that `Bib`, the type of `bib0`, may contain *zero* or more `Book` elements, therefore the expression `bib0/book` might contain zero `book` elements, in which case, `bib0/book/author` would contain no authors.

This illustrates an important feature of the type system: the type of an expression depends only on the type of its sub-expressions. It also illustrates the difference between an expression's value at query-evaluation time and its type at query-analysis time. Since the type of `$bib0` is `Bib`, the best type for `$bib0/book/author` is one listing zero or more authors, even though for the given value of `$bib0`, the expression will always contain exactly five authors.

Its also possible to project on attributes. This expression produces the `year` attribute of `$book0` whose type is `ATTRIBUTE year (xs:string)`.

```
    $book0/@year
==> ATTRIBUTE year "1999"
:   ATTRIBUTE year (xs:string)
```

2.3 Simple data

One may access simple data (strings, integers, or booleans) using the keyword `data()`. For instance, if we wish to select all author names in a book, rather than all author elements, we could write the following.

```
    $book0/author/data()
==> ("Abiteboul",
     "Buneman",
     "Suciu")
:   xs:string+
```

Similarly, it is possible to project the simple values of attributes. The following returns the year the book was published.

```
    $book0/@year/data()
==> 1999
:   xs:integer
```

The `data()` operator has a similar purpose to the the `text()` node test in XPath 1.0, in that they both project the atomic values in a document. In XPath 1.0, `text` selects the text node children of an element node, where as in XQuery, `data` returns the simple-typed value of the element node. We chose the keyword `data()` because, as the second example shows, not all data items are strings.

2.4 Iteration

Another common operation is to iterate over elements in a document so that their content can be transformed into new content. Here is an example of how to process each book to list the author before the title, and remove the year and isbn.

```
    FOR $b IN $bib0/book RETURN
```

```
        <book> { $b/author, $b/title } </book>
==> (<book>
         <author>Abiteboul</author>
         <author>Buneman</author>
         <author>Suciu</author>
         <title>Data on the Web</author>
       </book>,
       <book>
         <author>Fernandez</author>
         <author>Suciu</author>
         <title>XML Query</author>
       </book>)
:    (ELEMENT book(
        (ELEMENT author(xs:string))+,
        ELEMENT title(xs:string))
     )*
```

The `for` expression iterates over all `book` elements in `$bib0`, and binds the variable `$b` to each such element. For each element bound to `$b`, the inner expression constructs a new `book` element containing the book's authors followed by its title. The transformed elements appear in the same order as they occur in `$bib0`.

In the result type, a `book` element is guaranteed to contain one or more authors followed by one title. Let's look at the derivation of the result type to see why:

```
$bib0/book          : Book*
$b                  : Book
$b/author           : (ELEMENT author(xs:string))+
$b/title            : ELEMENT title (xs:string)
```

The type system can determine that `$b` is always `Book`, therefore the type of `$b/author` is `(ELEMENT author(xs:string))+`, and the type of `$b/title` is `ELEMENT title (xs:string)`.

In general, the value of a `for` expression is a sequence of zero or more data-model values as defined in [XQuery 1.0 and XPath 2.0 Data Model]. If the body of the `for` expression itself yields a sequence, then all of the sequences are concatenated together. For instance, the expression:

```
FOR $b IN $bib0/book RETURN
  $b/author
```

is exactly equivalent to the expression `$bib0/book/author`.

2.5 Selection

To select values that satisfy some predicate, we use the `where` expression. For example, the following expression selects all `book` elements in `$bib0` that were published before 2000.

```
    FOR $b IN $bib0/book
    WHERE $b/@year/data() <= 2000 RETURN
      $b
==> <book year="1999" isbn="1-55860-622-X">
     <title>Data on the Web</title>
     <author>Abiteboul</author>
     <author>Buneman</author>
     <author>Suciu</author>
    </book>
:   Book*
```

In general, an expression of the form:

`where` e_1 return e_2

is converted to the form

`if` e_1 `then` e_2 `else` ()

WHERE e_1 and e_2 are expressions. Here `()` is an expression that stands for the empty sequence, a sequence that contains no attributes or elements. We also write `()` for the type of the empty sequence.

According to this rule, the expression above translates to

```
FOR $b IN $bib0/book RETURN
  IF $b/@year/data() <= 2000 THEN $b ELSE ()
```

and this has the same value and the same type as the preceding expression.

2.6 Quantification

The following expression selects all `book` elements in `$bib0` that have *some* author named "Buneman".

```
    FOR $b IN $bib0/book
    WHERE SOME $a IN $b/author SATISFIES $a/data() = "Buneman" RETURN
      $b
==> <book year="1999" isbn="1-55860-622-X">
     <title>Data on the Web</title>
     <author>Abiteboul</author>
     <author>Buneman</author>
     <author>Suciu</author>
    </book>
:   Book*
```

We can use the `every` expression to find all books where all the authors are Buneman:

```
    FOR $b IN $bib0/book
    WHERE EVERY $a IN $b/author SATISFIES $a/data() = "Buneman" RETURN
      $b
==> ()
:   Book*
```

There are no such books, so the result is the empty sequence.

2.7 Join

Another common operation is to *join* values from one or more documents. To illustrate joins, we give a second data source that defines book reviews:

```
    TYPE Reviews  =
      ELEMENT reviews (
        (ELEMENT book (
```

```
          ELEMENT title  (xs:string),
          ELEMENT review (xs:string))
        )*
      )
    LET $review0 :=
      <reviews>
        <book>
          <title>XML Query</title>
          <review>A darn fine book.</review>
        </book>,
        <book>
          <title>Data on the Web</title>
          <review>This is great!</review>
        </book>
      </review> : Reviews
    RETURN ...
```

The Reviews type contains one reviews element, which contains zero or more book elements; each book contains a title and a review.

We can use nested for expressions to join the two sources $review0 and $bib0 on title values. The result combines the title, authors, and reviews for each book.

```
    FOR $b IN $bib0/book, $r IN $review0/book
    WHERE $b/title/data() = $r/title/data() RETURN
      <book>{ $b/title, $b/author, $r/review }</book>
==> (<book>
       <title>Data on the Web</title>
       <author>Abiteboul</author>
       <author>Buneman</author>
       <author>Suciu</author>
       <review>A darn fine book.</review>
     </book>,
     <book>
       <title>XML Query</title>
       <author>Fernandez</author>
       <author>Suciu</author>
       <review>This is great!</review>
     </book>)
:   (ELEMENT book(
      ELEMENT title  (xs:string),
      (ELEMENT author (xs:string))+,
      ELEMENT review (xs:string))
    )*
```

Note that the outer-most for expression determines the order of the result. Readers familiar with optimization of relational join queries know that relational joins commute, i.e., they can be evaluated in any order. This is not true for XQuery: changing the order of the first two for expressions would produce different output. In **[2.8 Unordered built-in function]**, we introduce support for unordered sequences, which permits commutable joins.

It is beyond the scope of this document to describe algorithms for evaluating nested loop joins. See [Graefe93] for a survey.

2.8 Unordered built-in function

As discussed in **[2.7 Join]** joins do not commute on ordered forests. In databases, ordering often does not matter. To permit commutable joins, and to allow for other query optimization techniques, XQuery also allows to explicitly disregard the order of a sequence. This is accomplished by the built-in function `UNORDERED`. The expression `UNORDERED`(*Expr*) may return any permutation of the sequence returned by *Expr*. For example, when applying `UNORDERED` to the join-query **[2.7 Join]**, the result may be either ordered as in **[2.7 Join]** or as below:

```
       UNORDERED(
         FOR $b IN $bib0/book, $r IN $review0/book
         WHERE $b/title/data() = $r/title/data() RETURN
           <book> { $b/title, $b/author, $r/review } </book>
       )
   ==> (<book>
          <title>XML Query</title>
          <author>Fernandez</author>
          <author>Suciu</author>
          <review>This is great!</review>
        </book>,
        <book>
          <title>Data on the Web</title>
          <author>Abiteboul</author>
          <author>Buneman</author>
          <author>Suciu</author>
          <review>A darn fine book.</review>
        </book>)
   :   ELEMENT book (
         ELEMENT title  (xs:string),
         ELEMENT author (xs:string)+,
         ELEMENT review (xs:string)
       )*
```

The expression `UNORDERED`(*Expr*) satisfies some useful laws that can be used for optimization. E.g., `UNORDERED` distributes over `FOR`, and nested `FOR` expressions on `UNORDERED` expressions are commutative; see also Rules 12--18 in **[A.2 Laws]**. On this basis joins can be commuted, i.e., switching the inner `FOR` expression with the outer `FOR` expression, does not change the semantics of the above query:

```
       UNORDERED(
         FOR $r IN $review0/book, $b IN $bib0/book
         WHERE $b/title/data() = $r/title/data() RETURN
           <book> { $b/title, $b/author, $r/review } </book>
       )
```

Note that unordered sequences are currently not distinguished from ordered sequences at type level. This is mainly for two reasons: (1) XML Schema does not distinguish between unordered sequences and ordered sequences and (2) the distinction requires to overload all built-in operators for sequences, such as `FOR`, `DISTINCT`, as well as user defined functions on sequences.

2.9 Parent and treat operators

Many of the previous queries select values and return them or use them in the construction of new values. Once a value is selected, however, the previous queries do not access the original source of the selected value, i.e., the document or hierarchy of elements in which the selected value is contained. It is sometimes useful, however, to access or preserve the original context of selected nodes.

Consider the following example, which contains a new bibliography of articles in `bib1`:

```
TYPE Bib1 = <bib>Article*</bib>
TYPE Article =
  ELEMENT article(
    ATTRIBUTE year (xs:integer),
    ELEMENT title  (xs:string),
    ELEMENT journal(xs:string),
    (ELEMENT author (xs:string))+
  )
LET $bib1 :=
  <bib>
    <article year="2000">
      <title>Queries and computation on the web</title>
      <journal>Theoretical Computer Science</journal>
      <author>Abiteboul</author>
      <author>Vianu</author>
    </article>
  </bib> : Bib1
RETURN ...
```

Assume there exists a full-text search function, `contains`, which given a set of documents, selects elements that contain a particular keyword. (This function is not defined in XQuery, but is used here to illustrate a point.) The details of function application and declaration are given in [**2.18 Functions**].

The following expression returns those elements in `bib0` and `bib1` that contain the keyword "Abiteboul". Note that the result type of the expression is `AnyTree*`. This is because the `contains` function cannot know apriori which elements, if any, contain a given keyword.

```
    FOR $a IN contains(($bib0, $bib1), "Abiteboul") RETURN
      $a
==> (<author>Abiteboul<author>, <author>Abiteboul</author>)
:   AnyTree*
```

The result above does not provide the *context* in which the two `author` elements occur. Even if the `contains` function did return more context, it might be useful to *browse* the context in which they occurred, for example, by accessing their parent and/or sibling elements. The built-in function `parent` accesses the parent of an attribute or element. For example, this expression returns more useful information than the previous one:

```
    FOR $a IN contains(($bib0, $bib1),  "Abiteboul") RETURN
      $a/..
==> (<book year="1999" isbn="1-55860-622-X">
       <title>Data on the Web</title>
       <author>Abiteboul</author>
       <author>Buneman</author>
       <author>Suciu</author>
     </book>,
     <article year="2000">
       <title>Queries and computation on the web</title>
       <journal>Theoretical Computer Science</journal>
       <author>Abiteboul</author>
       <author>Vianu</author>
     </article>)
:   AnyElement*
```

Note that the result type of the expression is `AnyElement*`. When applied to an attribute or element value, the `parent` function always has return type `AnyElement?`, i.e., zero or one `AnyElement`. This is because XQuery's type system only preserves type information about an attribute or element's content, not about its containing parent.

It is possible to recover more precise type information with the *dynamic* or *run-time* `treat` operator, which attempts to cast at run time an expression to a given type. If the expression does not have the given type, a run-time error is raised. Dynamic casts are necessary when it not possible to determine at query-analysis time the most precise type of a value; they are sometimes called ``down casts''.

For example, the use of `parent` in the following expression loses some useful type information, that is that `p` is a `Book`. We can recover more precise information by casting `p` to the `Book` type:

```
    FOR $p IN $book0/title/.. RETURN
      TREAT AS Book ($p)
==> <book year="1999" isbn="1-55860-622-X">
      <title>Data on the Web</title>
      <author>Abiteboul</author>
      <author>Buneman</author>
      <author>Suciu</author>
    </book>
:   Book?
```

The result type is `Book?` because the `parent` function has type `AnyElement?`. If we try erroneously to cast `p` to an `Article`, the error value is returned. Its type is again `Article?`, because AnyElement could be an Article:

```
    FOR $p IN $book0/title/.. RETURN
      TREAT AS Article ($p)
==> error
:   Article?
```

However, if we try to cast `$book0` to an `Article`, the result type becomes Ø, the empty choice, because we can statically determine that a `Book` is not an `Article`.

```
    FOR $p IN $book0 RETURN
      TREAT AS Article ($p)
==> error
:   0
```

We have already seen many examples of *static* or *compile-time* casting. A static cast permits the type of an expression to be changed and checked at query-analysis time; they are sometimes called ``up casts''. For example, consider the type `Book0`, which permits a book to have zero or more authors.

```
    TYPE Book0 =
      ELEMENT book(
        ATTRIBUTE year (xs:integer) &
        ATTRIBUTE isbn (xs:string),
        ELEMENT title  (xs:string),
        (ELEMENT author (xs:string))*
      )
```

The explicit-type expression *e : t* statically casts a value to the given type. For example, the expression below statically casts `$book0` to `Book0`; this is permissible because the type of `$book0` at query-analysis time is a sub-type of `Book0`.

```
    $book0 : Book0
==> <book year"1999" isbn="1-55860-622-X">
```

```
      <title>Data on the Web</title>
      <author>Abiteboul</author>
      <author>Buneman</author>
      <author>Suciu</author>
    </book>
:   Book0
```

If we try erroneously to cast `$book0` to a more precise type (e.g., a book with 4 or more authors), a type error will occur at query-analysis time.

2.10 References and node identity

The uses of the `parent` operator in [**2.9 Parent and treat operators**] show that it is possible to access the original context of nodes. This is possible because the XML Query Data Model supports *node identity*, that is, every instance of a node (e.g., element, attribute, processing instruction, and comment) in the data model has a unique identity. We can compare the identity of two nodes for equality using the == operator. For example, in the following expression, two distinct element nodes are created and bound to variables `a1` and `a2`. Although the two nodes are structurally equal, their identities are not equal:

```
    LET $a1 := <author>Suciu</author>,
        $a2 := <author>Suciu</author>
    RETURN
      $a1 == $a2
==> false
:   xs:boolean
```

In general, all XQuery's operators preserve node identity. There is one exception: the element constructor, which given a tag name and a sequence of children nodes, constructs a new element. A new element's content does not refer directly to the given children nodes, but to *copies* of these nodes. For example, the following expression constructs an element with name `newbook` and content `$book0/author`, `$book0/title`:

```
    LET $book1 :=
      <newbook>
          { $book0/author, $book0/title }
      </newbook>
    RETURN $book1
==> <newbook>
      <author>Abiteboul</author>
      <author>Buneman</author>
      <author>Suciu</author>
      <title>Data on the Web</title>
    </newbook>
:   ELEMENT newbook (
      (ELEMENT author (xs:string))+,
      ELEMENT title  (xs:string)
    )
```

The `newbook` element contains copies of the nodes in the sequence `$book0/author`, `$book0/title`, not the original nodes in `$book0`. Copying guarantees that a node is always the parent of its child nodes and a node is always the child of its parent; these constraints are invariants of [XQuery 1.0 and XPath 2.0 Data Model]. For example, we would expect that the following expression is always true:

```
    $book1/author/.. == $book1
```

If the element constructor did not copy its arguments, anomalies such as the following could occur:

```
$book1/author/.. == $book0
```

that is, the parent of `book1`'s child node is not `book1`, and this would violate the XML Query Data Model's parent-child invariant.

Sometimes it is useful to construct elements that do preserve the identity of its child nodes, for example, when constructing a *view* of one or more XML documents. In this case, we want the new element to contain *references* to, not copies of, the original nodes. The `ref` operator constructs a reference to a node. For example, `book2` below contains references to the nodes in `$book0`:

```
    LET $book2 :=
      <newbook>
        { (FOR $a IN $book0/author RETURN ref($a)),
          ref($book0/title)
        }
      </newbook>
    RETURN $book2
==> <newbook>
      <q:ref><author>Abiteboul</author></q:ref>
      <q:ref><author>Buneman</author></q:ref>
      <q:ref><author>Suciu</author></q:ref>
      <q:ref><title>Data on the Web</title></q:ref>
    </newbook>
:   ELEMENT newbook (
      (REFERENCE (ELEMENT author (xs:string)))+,
      REFERENCE (ELEMENT title  (xs:string))
    )
```

Ed. Note: MF : Issue - serialized representation of Data Model reference nodes.

Note that the type of the expression contains reference types. The `deref` operator dereferences a reference value. In the following, it returns the elements in `$book0`.

```
    FOR $v IN $book2/* RETURN deref($v)
==> (<author>Abiteboul</author>,
     <author>Buneman</author>,
     <author>Suciu</author>,
     <title>Data on the Web</title>)
:   (ELEMENT author (xs:string))+,
    ELEMENT title  (xs:string)
```

For convenience, the expression above can be also be written as `$book2/deref()`.

2.11 Restructuring and grouping

Often it is useful to regroup elements in an XML document. For example, each `book` element in `$bib0` groups one title with multiple authors. This expression groups each author with the titles of his/her publications.

```
    FOR $a IN distinct-value($bib0/book/author/data()) RETURN
      <biblio>
        <author>{ $a }</author>
        { FOR $b IN $bib0/book, $a2 IN $b/author/data()
```

```
            WHERE $a = $a2 RETURN
              $b/title
          }
        </biblio>
==> (<biblio>
        <author>Abiteboul</author>
        <title>Data on the Web</title>
      </biblio>,
      <biblio>
        <author>Buneman</author>
        <title>Data on the Web</title>
      </biblio>,
      <biblio>
        <author>Suciu</author>
        <title>Data on the Web</title>
        <title>XML Query</title>
      </biblio>,
      <biblio>
        <author>Fernandez</author>
        <title>XML Query</title>
      </biblio>)
:   (ELEMENT biblio (
       ELEMENT author (xs:string),
       (ELEMENT title  (xs:string))*)
     )*
```

Readers may recognize this expression as a self-join of books on authors. The expression `distinct-value($bib0/book/author/data())` produces a sequence of author names whose values are all distinct. The outer `for` expression binds `$a` to the name of each author element, and the inner `for` expression selects the title of each book that has some author whose name equals `$a`.

Here `distinct-value` is an example of a built-in function. It produces a sequence of nodes whose values are all distinct, i.e., there are no duplicate values; the order of the resulting sequence is not defined. The builtin function `distinct-node` produces a sequenc of nodes whose *identities* are all distinct.

The type of the result expression may seem surprising: each `biblio` element may contain *zero* or more `title` elements, even though in `$bib0` every `author` co-occurs with a `title`. Recognizing such a constraint is outside the scope of the type system, so the resulting type is not as precise as we would like.

2.12 Querying order

> **Ed. Note:** MF: Clearly, the following example of index is not appropriate for a tutorial -- it is only used to define RANGE.

Often it is useful to query the order of elements in an sequence or a document. There are two kinds of order among elements: *local* order and *document* (or global) order. XQuery supports querying of local and global order.

Local order refers to the order among sibling elements in an sequence. To query local order, the `index` function pairs an integer index with each element in an sequence:

```
    index($book0/author)
==> (<q:pair><q:fst>1</q:fst>
              <q:snd><q:ref><author>Abiteboul</author></q:ref></q:snd></q:pair>,
     <q:pair><q:fst>2</q:fst>
```

```
              <q:snd><q:ref><author>Buneman</author></q:ref></q:snd></q:pair>
       <q:pair><q:fst>3</q:fst>
              <q:snd><q:ref><author>Suciu</author></q:ref></q:snd></q:pair>)
:   (ELEMENT q:pair(
      ELEMENT q:fst (xs:integer),
      ELEMENT q:snd (REFERENCE (ELEMENT author (xs:string))))
    )+
```

The `index` function uses reference in order to preserve node identity when accessing local order. Note that the result type takes into account that at least one pair exists in the result, as `$book0/author` always contains one or more authors.

Once we have paired authors with an integer index, we can select the first two authors:

```
    FOR $p IN index($book0/author)
    WHERE ($p/q:fst/data() <= 2) RETURN
      $p/q:snd/deref()
==> (<author>Abiteboul</author>,
     <author>Buneman</author>)
:   (ELEMENT author (xs:string))*
```

The `for` expression iterates over all pair elements produced by the `index` expression. It selects elements whose index value in the `q:fst` element is between one and two inclusive, and it returns the original content by dereferencing the content of the `q:snd` element.

The result type may be surprising, because the `Book` type guarantees that each book has at least one author. However, the type system cannot determine that the conditional `where` expression will always succeed, so the inner expression may produce zero results. (A sophisticated analysis might improve type precision, but is likely to require much work for little benefit.)

Document (or global) order refers to the total order among all elements in a document. Global order is defined as the order of appearance of the element nodes when performing a pre-order, depth-first traveral of a tree. This corresponds to the order of appearance of their opening tags in the XML serialization. This is equivalent to the definition used in [XPath].

To query global order, the `xfo:node-before` function can be applied to two nodes. It returns true if the first node is before (and different from) the second node in document order. It returns false if the first node is equal to or after the second node in document order. It raises an error if the nodes are in different documents. For example, the nodes `bib0` and `review0` are unrelated therefore comparing their order raises an error:

```
    xfo:node-before($bib0, $review0)
==> ERROR
:   0
```

The `xfo:node-after` function is defined similiarly.

Using global order, the following expression returns all author nodes appearing after a book written in 2001:

```
    FOR $b IN $bib0/book
    WHERE $b/@year/data() = 2001 RETURN
      (FOR $a IN $bib0/book/author
       WHERE $b before $a RETURN $a)
==> (<author>Fernandez</author>,
     <author>Suciu</author>)
```

```
    :   (ELEMENT author (xs:string))*
```

Note that the root element of a document is before any other element. More generally, an element is before all of its children. For example, the set of elements that are before `$bib0` is empty:

```
        empty(FOR $b IN $bib0/book
              WHERE $b before $bib0 $b RETURN
                $b)
    ==> true
    :   xs:boolean
```

XQuery supports global order only for elements within the same document. Support for global order among elements in distinct documents is discussed in **[Issue-0003: Global Order]**.

2.13 Sorting

To sort a sequence, XQuery provides a sort expression, whose form is: *Expr*$_1$ `sortby` *Expr*$_2$. The built-in variable '.', called dot, ranges over the items in the sequence *Expr*$_1$ and sorts those items using the key value defined by *Expr*$_2$. For example, this expression sorts `book` elements in `$review0` by their titles.

```
        $review0/book SORTBY ./title/data()
    ==> (<book>
           <title>Data on the Web</title>
           <review>This is great!</review>
         </book>,
         <book>
           <title>XML Query</title>
           <review>This is pretty good too!</review>
         </book>)
    :   (ELEMENT book (
          ELEMENT title (xs:string),
          ELEMENT review (xs:string))
        )*
```

The `sort` expression is a restricted form of *higher-order* function, i.e., it takes a function as an argument. In this case, `sort` takes a single function, which extracts the key value from each element. The `sort` expression requires that the less-than inequality, <, be defined for the type of *Expr*$_2$.

2.14 Aggregation

We have already seen two built-in functions: `index` and `distinct-value`. In addition to these functions, XQuery has five built-in aggregation functions: `avg`, `count`, `max`, `min`, and `sum`.

This expression selects books that have more than two authors:

```
        FOR $b IN $bib0/book
        WHERE count($b/author) > 2 RETURN
          $b
    ==> <book year="1999" isbn="1-55860-622-X">
          <title>Data on the Web</title>
          <author>Abiteboul</author>
          <author>Buneman</author>
          <author>Suciu</author>
```

```
    </book>
:   Book*
```

All the aggregation functions take a sequence with repetition type and return an integer value; `count` returns the number of elements in the sequence.

2.15 Expanded names

So far, all our examples of attributes and elements use unqualified *local names*, i.e., names that do not include an explicit namespace URI. It is also possible to specify and match on the *expanded name* of an attribute or element. The expanded name *Namespace*:*LocalName* consists of a namespace URI *Namespace* and a local name *LocalName*.

Consider an inventory of books that contains data from both `http://www.BooksRUs.com` and `http://www.cheapBooks.com`. In this example, the first element contains values whose names are defined in the `BooksRUs.com` namespace, and the second element contains values whose names are defined in the `cheapBooks.com` namespace:

```
NAMESPACE booksRus = "http://www.BooksRUs.com/books.xsd"
NAMESPACE cheapBooks = "http://www.cheapBooks.com/ourschema.xsd"
TYPE Inventory = <inv> InvBook* </inv>
LET $inventory :=
  <inv>
    <booksRus:book year="1999" isbn="1-55860-622-X">
      <booksRus:title>Data on the Web</booksRus:title>
      <booksRus:author>Abiteboul</booksRus:author>
      <booksRus:author>Buneman</booksRus:author>
      <booksRus:author>Suciu</booksRus:author>
    </booksRus:book>
    <cheapBooks:book year="2001">
      <cheapBooks:title>XML Query</cheapBooks:title>
      <cheapBooks:author>Fernandez</cheapBooks:author>
      <cheapBooks:author>Suciu</cheapBooks:author>
      <cheapBooks:isbn>1-XXXXX-YYY-Z</cheapBooks:isbn>
    </cheapBooks:book>
  </inv> : Inventory
RETURN ...
```

In this example, elements imported from existing schemas each refer to a single namespace, thus the definition of `InvBook` is:

```
TYPE BooksRUBook =
  ELEMENT booksRus:book (
    ATTRIBUTE year (xs:integer) &
    ATTRIBUTE isbn (xs:string),
    ELEMENT booksRus:title(xs:string)
    (ELEMENT booksRus:author(xs:string))+
  )
TYPE CheapBooksBook =
  ELEMENT cheapBooks:book (
    ATTRIBUTE year (xs:integer),
    ELEMENT cheapBooks:title (xs:string),
    (ELEMENT cheapBooks:author(xs:string))+
    ELEMENT cheapBooks:isbn  (xs:string)
  )
TYPE InvBook = BooksRUBook | CheapBooksBook
```

Here vertical bar (|) is used to indicate a choice between types: each `InvBook` is either a `BooksRUBook` or a `CheapBooksBook`.

We have already seen how to project on the constant name of an attribute or element. It is also useful to project on *wildcards*, which are used to match names with any namespace and/or any local name. For example, this expression matches elements with any local name and with namespace URI `http://www.BooksRUs.com/books.xsd`:

```
    $inventory/booksRus:*
==> <booksRus:book year="1999" isbn="1-55860-622-X">
      <booksRus:title>Data on the Web</booksRus:title>
      <booksRus:author>Abiteboul</booksRus:author>
      <booksRus:author>Buneman</booksRus:author>
      <booksRus:author>Suciu</booksRus:author>
    </booksRus:book>
: ELEMENT booksRus:book (
    ATTRIBUTE year (xs:integer) &
    ATTRIBUTE isbn (xs:string),
    ELEMENT booksRus:title  (xs:string)
    (ELEMENT booksRus:author (xs:string))+
  )
```

Similarly, this expression first projects elements in any namespace whose local name is `book` and then projects on their `year` attributes:

```
    $inventory/*:book/@year
==> (ATTRIBUTE year (1999), ATTRIBUTE year (2001))
:   (ATTRIBUTE year (xs:integer))*
```

> **Ed. Note:** MF: Open issue whether *:localname will be supported.

The expression *Expr*/*a* is shorthand for *Expr*/*ns*:*a*, where *ns* is the default namespace. Similarly, `*` is shorthand for *ns*:`*`, i.e., any name in the default namespace.

2.16 Comments and processing instructions

In an XML document, comments and processing instructions may appear anywhere outside other markup[XML]. Processing instructions permit documents to contain instructions for applications. Comments and processing instructions are not part of the document's character data. An XML processor may, but need not, make the text of comments available to an application, but it must pass processing instructions to the application. The processing instruction begins with a target used to identify the application to which the instruction is directed.

XQuery supports comments and processing instructions in types and expressions. The type expression `PIC`(*t*) denotes a value in which zero or more processing instructions and comments may be interleaved arbitrarily with the nodes in type *t*. For example, the two element types `BibPIC` and `BookPIC` permit PIs and comments to be interleaved with their content:

```
TYPE BibPIC  = ELEMENT bib (pic(BookPIC*))
TYPE BookPIC =
  ELEMENT book (
    ATTRIBUTE year (xs:integer) &
    ATTRIBUTE isbn (xs:string),
```

```
      PIC (ELEMENT title (xs:string), (ELEMENT author(xs:string))+)
    )
```

Note that in the `book` element, the `PIC` operator is only applied to its element content, not its attribute content, because comments and processing instructions may not occur in attributes.

We can construct processing instruction and comment values using the built-in constructors `processing_instruction` and `comment`:

```
 LET $bibpc0 :=
  <bib>
     { comment("Canonical XQuery  example.") }
     <book year="1999" isbn="1-55860-622-X">
        { comment("First book example"),
          processing_instruction("Publisher.asp",
            "publisher=http://www.mkp.com") }
        <title>Data on the Web</title>
        <author>Abiteboul</author>
        <author>Buneman</author>
        <author>Suciu</author>
     </book>
     <book year="2001" isbn="1-XXXXX-YYY-Z">
        <title>XML Query"</title>
        { comment("Second book example") }
        <author>Fernandez</author>
        <author>Suciu</author>
     </book>
  </bib> : BibPIC
RETURN ...
```

Finally, we can project on processing instructions and comments, in the same way we project on children, attributes, and simple content:

```
    $bibpc0/book/comment()
==> (comment("First book example"), comment("Second book example"))
:   Comment*

    $bibpc0/book/processing_instruction()
==> processing_instruction("Publisher.asp", "publisher=http://www.mkp.com")
:   ProcessingInstruction*
```

Comments and processing instructions may be ignored by an XML processor, in which case they would not even be accessible to a query processor. If they are not ignored, however, comments and processing instructions are typed values and are treated like any other value in an XQuery expression.

2.17 Mixed Content

An element type has mixed content when elements of that type may contain character data, optionally interspersed with child elements[XML]. The type expression `mixed`(*t*) denotes a value in which zero or more `xs:string` values may be interleaved arbitrarily with the nodes in type *t*. For example, the content of the `review` element contains a `reviewer` element, which may be interleaved with string values:

```
TYPE ReviewsMixed  =
  ELEMENT reviews (
    (ELEMENT book (
```

```
        ELEMENT title (xs:string),
        ELEMENT review (MIXED (ELEMENT reviewer(xs:string)))))*
    )
```

Here are two examples of mixed content, in which the text of the book review may be interleaved with the name of the reviewer:

```
LET $reviewmix0 :=
  <reviews>
    <book>
      <title>XML Query</title>
      <review>A darn fine book: <reviewer>XML On-line</reviewer></review>
    </book>
    <book>
      <title>Data on the Web</title>,
      <review>The <reviewer>publisher</reviewer> says 'This is great!'</review>
    <book>
  </reviews> : ReviewsMixed
RETURN ...
```

It is often useful to concatenate all the string values of a mixed-content element to recover its complete text value. We use the builtin function `string_value`; as defined in XPath [XPath], the string value of a node is determined by its kind, e.g., element, attribute, etc.

```
    FOR $b IN $reviewmix0/book RETURN
      string_value($b/review)
==> ("A darn fine book : XML On-line",
     "The publisher says 'This is great!'")
:   xs:string*
```

2.18 Functions

Functions can make queries more modular and concise. Recall that we used the following query to find all books that do not have "Buneman" as an author.

```
    FOR $b IN $bib0/book
    WHERE EVERY $a IN $b/author SATISFIES NOT($a/data() = "Buneman") RETURN
      $b
==> <book year="2001" isbn="1-XXXXX-YYY-Z">
      <title>XML Query</title>
      <author>Fernandez</author>
      <author>Suciu</author>
    </book>
:   Book*
```

A different way to formulate this query is to first define a function that takes a string `s` and a book `b` as arguments, and returns true if book `b` does not have an author with name `s`.

```
DEFINE FUNCTION notauthor (xs:string $s, Book $b) RETURNS xs:boolean {
  EVERY $a IN $b/author SATISFIES NOT($a/data() = $s)
}
```

The query can then be re-expressed as follows.

```
FOR $b IN bib0/book
WHERE notauthor("Buneman", $b) RETURN
  $b
```

```
==> <book year="2001" isbn="1-XXXXX-YYY-Z">
      <title>XML Query</title>
      <author>Fernandez</author>
      <author>Suciu</author>
    </book>
:   Book*
```

Note that a function declaration includes the types of all its arguments and the type of its result. This is necessary for the type system to guarantee that applications of functions are type correct.

In general, any number of functions may be declared at the top-level. The order of function declarations does not matter, and each function may refer to any other function. Among other things, this allows functions to be recursive (or mutually recursive), which supports structural recursion, the subject of the next section.

Functions make XQuery extensible. We have seen examples of built-in functions (`sort` and `distinct-value`) and examples of user-defined functions (`notauthor`). In addition to built-in and user-defined functions, XQuery could support externally defined functions, i.e., functions that are not defined in XQuery itself, but in some external language. This would make special-purpose implementations of, for example, full-text search functions available in XQuery. We discuss support for externally defined functions in **[Issue-0009: Externally defined functions]**.

2.19 Structural recursion

XML documents can be recursive in structure, for example, it is possible to define a `part` element that directly or indirectly contains other `part` elements. In XQuery, we use recursive types to define documents with a recursive structure, and we use recursive functions to process such documents. (We can also use mutually recursive functions for more complex recursive structures.)

For instance, here is a recursive type defining a part hierarchy.

```
TYPE Part = Basic | Composite
TYPE Basic =
  ELEMENT basic (
    ELEMENT cost (xs:integer)
  )
TYPE Composite =
  ELEMENT composite (
    ELEMENT assembly_cost(xs:integer)
    ELEMENT subparts (Part+)
  )
```

And here is some sample data.

```
LET $part0 :=
  <composite>
    <assembly_cost>12</assembly_cost>
    <subparts>
      <composite>
        <assembly_cost>22</assembly_cost>
        <subparts>
          <basic><cost>33</cost</basic>
        </subparts>
      </composite>
      <basic><cost>7</cost</basic>
    </subparts>
  </composite> : Part
```

```
RETURN ...
```

Here vertical bar (|) is used to indicate a choice between types: each part is either basic (no subparts), and has a cost, or is composite, and includes an assembly cost and subparts.

We might want to translate to a second form, WHERE every part has a total cost and a list of subparts (for a basic part, the list of subparts is empty).

```
TYPE Part2 =
  ELEMENT part (
    ELEMENT total_cost (xs:integer),
    ELEMENT subparts (Part2*)
  )
```

Here is a recursive function that performs the desired transformation. It uses a new construct, the `typeswitch` expression.

```
DEFINE FUNCTION convert(Part $p) RETURNS Part2 {
  TYPESWITCH ($p) AS $x
    CASE Basic RETURN
      <part>
        <total_cost> { $x/cost/data() } </total_cost>
        <subparts/>
      </part>
    CASE Composite RETURN
      LET $s := (FOR $y IN $x/subparts/* RETURN convert($y))
      RETURN
      <part>
          <total_cost>
            { $q/assembly_cost/data() +
              sum($s/total_cost/data()) }
          <total_cost>
          <subparts> { $s } </subparts>
      </part>
    DEFAULT RETURN ERROR
}
```

Each branch of the typeswitch expression is labeled with a type, `Basic` or `Composite`. The evaluator checks the type of the value of `$p` at query-evaluation time, i.e., run time, and evaluates the corresponding branch. If the first branch is taken then `$x` is bound to the value of `$p`, and the branch returns a new part with total cost the same as the cost of `$x`, and with no subparts. If the second branch is taken, then `$x` is bound to the value of `$p`. The function is recursively applied to each of the subparts of `$x`, giving a sequence of new subparts `$s`. The branch returns a new part with total cost computed by adding the assembly cost of `$x` to the sum of the total cost of each subpart in `$s`, and with subparts `$s`.

One might wonder why `$x` is required, since it has the same value as `$p`. The reason why is that `$p` and `$x` have different types.

```
$p : Part
$x : Basic       -- in the first branch
$x : Composite   -- in the second branch
```

The type of `$x` is more precise than the type of `$p`, because which branch is taken depends upon the type of value in `$p`.

Applying the query to the given data gives the following result.

```
    convert($part0)
==> <part>
      <total_cost>74</total_cost>
      <subparts>
        <part>
          <total_cost>55</total_cost>
          <subparts>
            <part>
              <total_cost>33</total_cost>
              <subparts/>
            </part>
          </subparts>
        </part>
        <part>
          <total_cost>7</total_cost>
          <subparts/>
        </part>
      </subparts>
    </part>
: Part2
```

Of course, a `typeswitch` expression may be used in any query, not just in a recursive one.

2.20 Functions for all well-formed documents

[XML Schema : Formal Description] defines a ``root'', or most general type, for the four kinds of schema components: elements, attributes, simple types, and complex types. They are named `xs:AnyElement`, `xs:AnyAttribute`, `xs:AnySimpleType`, and `xs:AnyComplexType`, respectively. The type `xs:AnySimpleType` stands for the most general simple type. All built-in primitive types (like `xs:integer` or `xs:string`) and lists of simple types are subtypes of it. The built-in simple types are listed in [**3.4 Atomic simple types**]. The remaining schema components are defined as follows:

```
TYPE xs:AnyTree        = xs:AnySimpleType
                        | xs:AnyElement
                        | xs:AnyAttribute
TYPE xs:AnyAttribute   = ATTRIBUTE *:* (xs:AnySimpleType)
TYPE xs:AnyElement     = ELEMENT *:* (xs:AnyComplexType)
TYPE xs:AnyComplexType = xs:AnyAttribute*,
                         ((xs:AnyElement | xs:string)* | xs:AnySimpleType)
TYPE xs:AnyType        = (xs:AnyTree)*
```

The type `xs:AnyTree` denotes any simple type, element, or attribute. The type `xs:AnyAttribute` stands for the most general attribute type, which may have any name, and its content must have type `xs:AnySimpleType`, i.e., it may contain simple values, but no elements. The type `xs:AnyElement` stands for the most general element type, which may have any name, and its content must be a complex type. The type `xs:AnyComplexType` stands for the most general complex type, which is any sequence of attributes followed by any sequence of elements or strings or by any simple type. Strings are permissible in a complex type because an element may contain *mixed* content, i.e., character data interleaved with other elements. Finally, `xs:AnyType` is a sequence of any tree.

In particular, our earlier data also has type `xs:AnyElement`.

```
    $book0 : xs:AnyElement
==> <book year="1999" isbn="1-55860-622-X">
```

```
      <title>Data on the Web</title>
      <author>Abiteboul</author>
      <author>Buneman</author>
      <author>Suciu</author>
    </book>
:   xs:AnyElement
```

A specific type can be indicated for any expression in the query language, by writing a colon and the type after the expression.

As an example of a function that can be applied to all well-formed documents, we define a recursive function that converts any XML data into HTML. We first give a simplified definition of HTML.

```
    TYPE HTML_body =
      ( xs:AnySimpleType
      | ELEMENT b(HTML_body)
      | ELEMENT ul ((ELEMENT li (HTML_body))*)
      ) *
```

An HTML body consists of a sequence of zero or more items, each of which is either a simple value, or a `b` element, where the content is an HTML body, or an `ul` element, where the children are `li` elements, each of which has as content an HTML body.

Now, here is the function that performs the conversion.

```
DEFINE FUNCTION html_of_xml( xs:AnyTree $x ) RETURNS HTML_Body {
  TYPESWITCH ($x) AS $z
    CASE xs:AnySimpleType RETURN $z
    CASE xs:AnyAttribute RETURN
      <b> { name($z) } </b>,
      <ul> { FOR $y IN $z/data() RETURN <li>{ html_of_xml($y) }</li> } </ul>
    CASE xs:AnyElement RETURN
      <b> { name($z) } </b>,
      <ul>{ FOR y IN $z/@* RETURN <li>{ html_of_xml($y) }<li> } </ul>,
      <ul>{ FOR y IN $z/* RETURN <li>{ html_of_xml($y) }</li> } </ul>
    DEFAULT RETURN ERROR
}
```

The first branch of the `typeswitch` expression checks whether the value of `$x` is a subtype of `xs:AnySimpleType`, and if so then `$z` is bound to that value, so if this branch is taken then `$z` is the same as `$x`, but with a more precise type (it must be a simple type, not an element). This branch returns the scalar.

The second branch checks whether the value of `$x` is a subtype of `xs:AnyAttribute`. As before, `$z` is the same as `$x` but with a more precise type (it must be an attribute, not a scalar). This branch returns a `b` element containing the name of the attribute, and a `ul` element containing one `li` element for each value of the attribute. The function is recursively applied to get the content of the `li` element. The last branch is analogous to the second, but it matches an element instead of an attribute, and it applies `html_of_xml` to each of the element's attributes and children.

Applying the query to the book element above gives the following result.

```
    html_of_xml($book0)
==> <b>book</b>
    <ul>
```

```
    <li><b>year</b><ul><li>1999</li></ul></li>
    <li><b>isbn</b><ul><li>1-55860-622-X</li></ul></li>
    <li><b>title</b><ul><li>Data on the Web</li></ul></li>
    <li><b>author</b><ul><li>Abiteboul</li></ul></li>
    <li><b>author</b><ul><li>Buneman</li></ul></li>
    <li><b>author</b><ul><li>Suciu</li></ul></li>
  </ul>
: HTML_Body
```

2.21 Top-level queries and XML results

A XQuery *module* consists of a sequence of top-level declarations, i.e., a namespace declaration, function declaration, or type declaration, followed by a query expression. The order of top-level declarations is immaterial; all namespace, function, and type declarations may be mutually recursive.

The query expression is evaluated in the environment specified by all of the declarations. We have already seen examples of type, function, and namespace declarations. An example of a top-level query is:

```
html_of_xml(book0)
```

The result of a top-level query can be serialized into an XML document by the application in which XQuery is used.

3 XQuery Core Syntax

In this section, we present the grammar for XQuery's core expressions and types. Literals are in `typewriter` font. Terminal classes of literals are italicized and have the suffix 'literal', e.g., StringLiteral. Non-terminal symbols are italicized, e.g., *Expr*. In the grammar, the '|' operator denotes an alternative of two symbols; '*' denotes zero or more repetition of a symbol; and '?' denotes an optional symbol.

3.1 Expressions

[Figure 1] contains the grammar for XQuery's core expressions. We define XQuery's typing rules on these core expressions in [4 Static Semantics : Type-Inference Rules].

NCName		
Variable	::=	`$u, $v, $w, ...`
StringLiteral	::=	`"", "a"`, ...
NumericLiteral	::=	`0, 1, 2`, ...
BooleanLiteral	::=	`true` | `false`
Literal	::=	*StringLiteral*
	|	*NumericLiteral*
	|	*BooleanLiteral*
QName	::=	*NCName*
	|	*NCName*:*NCName*
Op_{eq}	::=	`eq` | `node-equal` | `ne` | `lt` | `lteq` | `gt` | `gteq`
Op_{arith}	::=	`+` | `-` | `*` | `mod` | `div`
Op_{coll}	::=	`union` | `except` | `intersect`
Op_{bool}	::=	`and` | `or`

```
          InfixOp ::= Op_eq | Op_arith | Op_coll | Op_bool
         PrefixOp ::= + | - | not
             Expr ::= Literal
                    | Variable
                    | QName ( ExpSequence? )
                    | Expr InfixOp Expr
                    | PrefixOp Expr
                    | attribute QName ( Expr )
                    | ElementConstructor
                    | ExprSequence
                    | if (Expr) then Expr else Expr
                    | let Variable := Expr return Expr
                    | Expr : Type
                    | error
                    | for Variable in Expr return Expr
                    | Expr sortby Expr ascending | descending
                    | cast as Type ( Expr )
                    | typeswitch (Expr) as Variable CaseRules
        CaseRules ::= case Type return Expr CaseRules
                    | default return Expr
     ExprSequence ::= Expr (, ExprSequence)*
ElementConstructor ::= <NameSpec/>
                    | <NameSpec> EnclosedExpression </NameSpec>
EnclosedExpression ::= { ExprSequence }
         NameSpec ::= QName
                    | { Expr }
         TypeDecl ::= type NCName = Type
      ContextDecl ::= namespace NCName = StringLiteral
                    | default namespace = StringLiteral
      FunctionDefn ::= define function QName ( ParamList? )
                      returns Type { Expr }
        ParamList ::= Type Variable (, Type Variable)*
      QueryModule ::= ContextDecl*
                      TypeDecl*
                      FunctionDefn* ExprSequence?
  QueryModuleList ::= QueryModule ( ; QueryModule )*
```

Figure 1: XQuery Core Expression Syntax

Many of the expressions that appear in the examples, for example *Expr*`/*`, do not appear in **[Figure 1]**, because they are reducible to expressions in the core syntax. **[6 XQuery Mapping to Core]** defines the mapping for every XQuery expression into an equivalent expression in the core syntax.

3.2 Operators

In addition to the core syntax, XQuery has a set of operators and built-in functions. The binary and unary operators are enumerated in **[Figure 1]**. They include two equality operators, `eq` and `nodeeq`, defined in the [XQuery 1.0 and XPath 2.0 Data Model], and five inequality operators, `lteq`, `lt`, `gteq`, gt, and ne. We have not defined the semantics of all the binary operators in XQuery. In particular, it might be useful to define more than one type of equality over scalar and element values, or to define implicit coercions between values of related types. A joint task force on operators with members from the [XSLT 99], XML Schema, and XML Query working groups is chartered to define operators. XQuery will adopt the decisions of that group (See **[Issue-0056: Operators on Simple Types]**).

3.3 Built-in functions

XQuery's built-in functions are either defined in [XQuery 1.0 and XPath 2.0 Data Model] or in a forthcoming document that defines operators and a library of functions for XQuery and XPath 2.0. In this document, the data model functions have no namespace prefix and the library functions have the namespace prefix xfo. Some of these functions require special static type rules; these are listed in **[Figure 2]** and **[Figure 3]**. **[Figure 2]** contains the constructor and accessor functions defined in the [XQuery 1.0 and XPath 2.0 Data Model]. The remaining built-in functions are listed in **[Figure 3]**. One benefit of having built-in functions is that more precise types can be given to these functions than to user-defined functions. The type rules for these functions appear in **[4 Static Semantics : Type-Inference Rules]**.

attributes(*Expr*)	Returns attributes of element.
children(*Expr*)	Returns children of element.
comment(*Expr*)	Constructs a comment.
dereference(*Expr*)	Dereferences a node reference.
local-name(*Expr*)	Extracts local NCName of a node.
name(*Expr*)	Returns element or attribute's tag name.
namespace-uri(*Expr*)	Extracts URI namespace from a node.
parent(*Expr*)	Returns the parent of a node.
pcdata(*Expr*)	Constructs parsable character data from string argument.
processing-instruction(*Expr*, *Expr*)	Constructs a processing instruction.
ref(*Expr*)	Constructs a node reference.
string-value(*Expr*)	Returns string value of given node, as defined in [XPath].
typed-value(*Expr*)	Returns the simple typed value of an element or attribute.

Figure 2: Data Model Constructor and Accessor Functions

agg(*Expr*)	Aggregation functions, where agg is avg, count, min, max, sum.
descendent-or-self(*Expr*)	Returns given node and all its descendents in document order.
distinct-node(*Expr*)	Removes duplicate nodes from a sequence.
distinct-value(*Expr*)	Removes duplicate values from a sequence.
eop(*Expr*)	Equality/inequality functions, where *eop* is one of eq, neq, lt, lteq, gt, gteq.
index(*Expr*)	Pairs each element of an sequence with integer index.
xfo:node-before(*Expr*, *Expr*)	True if first argument is before second in document order.
xfo:node-equal(*Expr*, *Expr*)	Returns true if both expressions denote the same node.

Figure 3: Built-In Functions

3.4 Atomic simple types

The XQuery type system takes as given the *built-in simple datatypes* from XML Schema Part 2 [XML Schema Part 2]. We let *b* range over all built-in datatypes.

built-in datatype *b* ::= `xs:AnySimpleType`
| `xs:string`
| `xs:boolean`
| `xs:float`
| `xs:double`
| `xs:decimal`
| `xs:timeDuration`
| `xs:recurringDuration`
| `xs:binary`
| `xs:uriReference`
| `xs:ID`
| `xs:IDREF`
| `xs:ENTITY`
| `xs:QName`
| `xs:CDATA`
| `xs:token`
| `xs:language`
| `xs:IDREFS`
| `xs:ENTITIES`
| `xs:NMTOKEN`
| `xs:NMTOKENS`
| `xs:Name`
| `xs:NCName`
| `xs:NOTATION`
| `xs:integer`
| `xs:nonPositiveInteger`
| `xs:negativeInteger`
| `xs:long`
| `xs:int`
| `xs:short`
| `xs:byte`
| `xs:nonNegativeInteger`
| `xs:unsignedLong`
| `xs:unsignedInt`
| `xs:unsignedShort`
| `xs:unsignedByte`
| `xs:positiveInteger`
| `xs:timeInstant`
| `xs:time`
| `xs:timePeriod`
| `xs:date`

| `xs:month`
| `xs:year`
| `xs:century`
| `xs:recurringDate`
| `xs:recurringDay`

The built-in simple datatype `AnySimpleType` stands for the most general simple type, and all other primitive and simple types (like `xs:integer` or `xs:string`) are subtypes of it.

In [XML Schema Part 2], a *simple datatype* is either a primitive datatype, or is derived from another simple datatype by specifying a set of facets. A type hierarchy is induced between simple types by containment of facets. Note that lists of simple datatypes are specified using repetition and unions are specified using alternation, as defined in **[3.5 Types]**

For simplicity, the type syntax in this document does not provide any way to define datatypes with facets. Such types can be imported from XML Schema and may be referenced by a qualified name *QName*. We let *p* range over all built-in datatypes, lists of built-in datatypes, and imported simple types.

p ::= *b*
| *b* `min` *m* `max` *n*
| *QName*

3.5 Types

[Figure 4] contains the abstract syntax for XQuery's type system. This type syntax appears in the typing rules in **[4 Static Semantics : Type-Inference Rules]**. An XQuery type corresponds to a *content group* as defined in [XML Schema : Formal Description]. See **[Issue-0088: Align XQuery types with XML Schema : Formal Description.]** for alignment issues between XQuery type syntax and [XML Schema : Formal Description].

---

type variable	*y*	
unit type	*u* ::= *p*	simple type
	| `ATTRIBUTE` *NameSet* (*t*)	
	| `ELEMENT` *NameSet* (*t*)	
	| `PROCESSING-INSTRUCTION`	
	| `COMMENT`	
	| `REFERENCE` (*t*)	
type	*t* ::= *y*	type variable
	| ()	empty sequence
	| Ø	empty choice
	| *u*	unit type
	| t_1 , t_2	sequence, t_1 followed by t_2
	| t_1 & t_2	interleaved product
	| t_1 | t_2	choice, t_1 or t_2
	| *t* `min` *m* `max` *n*	repetition of *t* *m* to *n* times
	| *t* *	repetition of *t* 0 to * times
	| *t* +	repetition of *t* one to * times
	| *t* ?	repetition of *t* 0 to 1 times
	| `PIC` (*t*)	

		|	MIXED (*t*)
bound	*m*, *n*	::=	natural number or *
prime type	*q*	::=	*u*
		|	*q* | *q*
expanded QName	*expQName*	::=	{*anyURI*}*NCName*
names	*NameSet*	::=	*QName*
		|	*expQName*
		|	*
		|	*NCName*:*
		|	*:*
		|	*NameSet* OR *NameSet*
		|	*NameSet* DIFF *NameSet*
		|	(*NameSet*)

Figure 4: Abstract Syntax for Types

*, +, ?, min *m* max *n*, |, &, ,,

Figure 5: Precedence of Type Operators (highest to lowest)

A *unit* type is is either a simple type, an attribute or element type with name in *NameSet* and content in *t*, a comment type, a processing-instruction type, or a node-reference type.

The empty sequence matches only the empty document; it is an identity for sequence and all. The empty choice matches no document; it is an identity for choice.

An interleaved product *t1* & *t2* is nodes in t_1 interleaved with nodes in t_2 The interleaved product (also known as the shuffle product) is a generalization of XML Schema's [XML Schema Part 1] allgroups. t_1 & t_2 matches any sequence that is an interleaving of a sequence that matches t_1 and a sequence that matches t_2. For example,

```
(ELEMENT a(), ELEMENT b()) & ELEMENT c() =
          ELEMENT a(), ELEMENT b(), ELEMENT c()
      |   ELEMENT a(), ELEMENT c(), ELEMENT b()
      |   ELEMENT c(), ELEMENT a(), ELEMENT b()
```

As another example, `ELEMENT a()* & ELEMENT b()` matches any sequence of `ELEMENT a()` and `ELEMENT b()` that has exactly one `ELEMENT b()`.

Allgroups in XML Schema may only consist of global or local element declarations with lower bound 0 or 1, and upper bound 1. With these restrictions, an allgroup in XML Schema is equivalent to p_1 min m_1 max 1 & ... & p_n min m_n max 1 where p_i are element declarations and 0 <= m_i <= 1.

The repetition type *t* min *m* max *n* denotes a sequence of at least *m* and at most *n* *t*. The bounds on a repetition type will be either a natural number (that is, either a positive integer or zero) or the special value *, meaning unbounded. We extend arithmetic to include * in the obvious way:

m + * = * + *m* = *
0 · * = * · 0 = 0
m · * = * · *m* = * if *m* ? 0
m min * = * min *m* = *m*
m max * = * max *m* = *
m <= * = true
* < *m* = false

For technical reasons, we allow the lower bound of a repetition to be `*`. A repetition *t* `min` *m* `max` *n* is equivalent to the empty choice Ø if *m* > *n* or if *m* is `*`.

The type expression `PIC`(*t*) is a convenient shorthand for the type `(PROCESSING-INSTRUCTION | COMMENT)* &` *t*, that is, a type in which processing instructions and comments may be interleaved with nodes in type *t*. Similarly, the type expression `MIXED`(*t*) is a convenient shorthand for the type `xs:AnySimpleType* &` *t*.

A *prime* type is a unit type or a disjunction of prime types. Unit and prime types appear in several typing rules in [**4 Static Semantics : Type-Inference Rules**].

A *wildcard expression* ([XML Schema : Formal Description]) denotes a set of names. A *name set* is either a *QName* denoting the name with the given namespace prefix and local name; an *expanded QName* denoting the name with the given namespace uri and local name; `*:*` denoting any name in any namespace; *NCName*:`*` denoting any local name in namespace *NCName*; or `*` denoting any local name in the default namespace. A name set also consists of union of wildcards or difference of wildcards. We let *NameSet* range over name sets.

The abstract syntax for content groups in [XML Schema : Formal Description] and in the corresponding abstract syntax above is more permissive than XML Schema. For example, these grammars permit an attribute to contain arbitrarily complex content and for attributes and elements to be combined in arbitrary ways. In [XML Schema : Formal Description], syntactic categories are used to specify various subsets of types and to restrict how types may be combined. Syntactic categories can indicate, for example, that the content of an attribute should be a simple type and that the content of an element should consist of attributes followed by elements. These restrictions guarantee that errors in type construction can be caught during parsing of a query.

We also use syntactic categories to restrict how types may be constructed. In the concrete syntax for types, we capitalize non-terminal symbols. We first distinguish between *type variables* used to name attribute groups, element groups, and the content types of elements. Type variables correspond to component names in [XML Schema : Formal Description].

TypeVar	::=	*AttrGroupVar*
	|	*ElemGroupVar*
	|	*ContentTypeVar*

A *simple type* is either an atomic type, a choice of atomic types, or a list of atomic or choice types:

UnionType	::=	*p*	atomic simple type
	|	*UnionType* **|** *UnionType*	choice of simple atomic types
SimpleType	::=	*UnionType*	
	|	*UnionType* `min` *m* `max` *n*	list of union type

An *attribute group* defines the syntactic form of attributes: their content may only be a simple type and they are combined only with the interleaving operator, but not the sequence operator.

AttrGroup	::=	ATTRIBUTE *NameSet* (*SimpleType*)	
	|	ATTRIBUTE *NameSet* (*SimpleType*) min 0 max 1	optional attribute
	|	*AttrGroup* & *AttrGroup*	
	|	*AttrGroupVar*	
	|	(*AttrGroup*)	
	|	()	

An *element group* contains elements with constant or wildcard names and they are combined in sequences, choices, interleavings, and repetitions.

ElemGroup	::=	ELEMENT *NameSet* (*ContentType*)
	|	*ElemGroup* , *ElemGroup*
	|	*ElemGroup* | *ElemGroup*
	|	*ElemGroup* & *ElemGroup*
	|	*ElemGroup* min *m* max *n*
	|	*ElemGroupVar*
	|	PIC (*ElemGroup*)
	|	MIXED (*ElemGroup*)
	|	(*ElemGroup*)
	|	()
	|	Ø

The *content type* of an element is either a simple type, an attribute group, an element group, a sequence of an attribute group followed by an element group, or a content-type variable.

ContentType	::=	*SimpleType*	
	|	*AttrGroup*	
	|	*ElemGroup*	
	|	*AttrGroup* , *ElemGroup*	
	|	*ContentTypeVar*	content-type variable

Finally, a *type* in an XQuery expression may be an attribute group, an element group, or a content type:

Type ::= *AttrGroup* | *ElemGroup* | *ContentType*

4 Static Semantics : Type-Inference Rules

XQuery's static semantics is presented as type inference rules, which relate XQuery expressions to types and and specify under what conditions an expression is well typed. The rules for typing expressions are described with an inference rule notation, which is used for describing type systems and semantics of programming languages. For a textbook introduction to type systems, see, for example, [Mitchell].

In inference notation, when all judgements above the line hold, then the judgement below the line holds as well. Here is an example of a rule used later on:

$$\frac{|\text{-}\ Data_1 : t_1 \qquad |\text{-}\ Data_2 : t_2}{|\text{-}\ Data_1, Data_2 : t_1, t_2}$$

Data is the subset of expressions that consists only of literal constant, attribute, element, sequence, and empty-sequence expressions.

Data::=*Literal*		literal constant
	`attribute` *NCName Data*	attribute
	<*NCName*/>	empty element
	<*NCName*> { *Data* } <*NCName*>	element
	Data, *Data*	sequence
	( )	empty sequence

The judgement "**|-** *Data* : *t*" is read as "in the empty environment, the value *Data* has type *t*." The symbol **|-** is called a *turnstyle*, and is usually preceded by an environment symbol, which represents a mapping from variables to values. In this example, there is no environment symbol, which means the judgement holds in the empty environment. In [4.4 Expressions], we will see examples of rules that have non-empty environments. The rule states that if both $Data_1$: t_1 and $Data_2$: t_2 hold, then $Data_1$, $Data_2$: t_1, t_2 holds as well. For instance, take

- $Data_1$ = `<a>{ 1 }</a>`
- $Data_2$ = `<b>{ "two" }</b>`
- t_1 = `ELEMENT a (xs:integer)`
- t_2 = `ELEMENT b (xs:string)`.

Then since both these hold:

```
<a>{ 1 }</a>     : ELEMENT a (xs:integer)
<b>{ "two" }</b> : ELEMENT b (xs:string)
```

we may conclude that the following holds:

```
  <a>{ 1 }</a>, <b>{ "two" }</b>
: ELEMENT a (xs:integer), ELEMENT b (xs:string)
```

4.1 Relating data to types

The following type rules relate *Data* values to types. The next rule rules states that a literal *NCName* has the type `xs:NCName`. The following three rules are analogous for strings and double precision numbers.

$$\frac{}{\vdash NCName : \texttt{xs:NCName}}$$

$$\frac{}{\vdash StringLiteral : \texttt{xs:string}}$$

$$\frac{}{\vdash NumericLiteral : \texttt{xs:double}}$$

> **Ed. Note:** MF: Analogous rules are necessary for all the functions that construct Schema simple-typed values.

The next three rules are for attribute and element construction. The first rule states that if *Data* has type *t*, then the attribute expression `attribute` *NCName Data* has type `ATTRIBUTE` *NCName* (*t*). The subsequent rules are analogous for element expressions.

$$\frac{\vdash Data : t}{\vdash \texttt{attribute}\ NCName\ Data : \texttt{ATTRIBUTE}\ NCName\ (t)}$$

$$\frac{}{\vdash <NCName/> : \texttt{ELEMENT}\ NCName\ ()}$$

$$\frac{\vdash Data : t}{\vdash <NCName>\ \{\ Data\ \}\ </NCName> : \texttt{ELEMENT}\ NCName\ (t)}$$

The next rule states that the empty sequence value always has the empty sequence type:

$$\frac{}{\vdash () : ()}$$

This rule was described above:

$$\frac{\vdash Data_1 : t \qquad \vdash Data_2 : t}{\vdash Data_1, Data_2 : t_1, t_2}$$

The rules above associate the most specific type possible with a *Data* value. The remaining rules in this section associate more general types with a *Data* value, which are necessary when the type system must determine whether a *Data* value with most specific type *t* is permissible when a value of type *t'* is expected. This occurs during type checking of a query.

The next two rules are also for attribute and element construction, but these rules specify more general types. The first rule states that if *Data* has type *t*, and *NCName* is in the set of names defined by the wildcard expression *NameSet*, then the given attribute expression also has type `ATTRIBUTE` *NameSet* (*t*). The subsequent rules are analogous for element expressions.

$$\frac{\vdash Data : t \qquad NCName <:_{Wild} NameSet}{\vdash \texttt{attribute}\ NCName\ Data : \texttt{ATTRIBUTE}\ NameSet\ (t)}$$

$$\frac{NCName <:_{Wild} NameSet}{\vdash <NCName/> : \texttt{ELEMENT}\ NameSet\ ()}$$

$$\frac{NCName <:_{Wild} NameSet \qquad \vdash Data : t}{\vdash <NCName>\ \{\ Data\ \}\ </NCName> : \texttt{ELEMENT}\ NameSet\ (t)}$$

The next rule states that the sequence of one value with type *t* followed by a value with repetition type $t\ \texttt{min}\ m\ \texttt{max}\ n$ has repetition type $t\ \texttt{min}\ m{+}1\ \texttt{max}\ n{+}1$.

$$\frac{\vdash Data_1 : t \qquad \vdash Data_2 : t\ \texttt{min}\ m\ \texttt{max}\ n}{\vdash Data_1, Data_2 : t\ \texttt{min}\ (m+1)\ \texttt{max}\ (n+1)}$$

The next rule states that the empty sequence is a repetition of any type with lower bound 0:

$$\frac{}{\vdash () : t\ \texttt{min}\ 0\ \texttt{max}\ n}$$

4.2 Equivalences and subtyping

The symbol <: denotes the subtype relation. We write $t_1 <: t_2$ if for every data *Data* such that |- *Data* : t_1, it is also the case that |- *Data* : t_2, i.e., is t_1 is a subtype of t_2. The subtyping relation is used in many of the type rules that follow. It is easy to see that the subtype relation <: is a partial order, i.e., it is reflexive, $t <: t$, and it is transitive, if $t_1 <: t_2$ and $t_2 <: t_3$ then $t_1 <: t_3$. Here are some of the inequalities that hold:

$$\varnothing <: t$$
$$t <: \texttt{xs:AnyType}$$
$$p <: \texttt{xs:AnySimpleType}$$
$$t_1 <: t_1 \mid t_2$$
$$t_2 <: t_1 \mid t_2$$

Further, if *QName* $<:_{\text{wild}}$ *NameSet* and $t <: t'$, then

$$\texttt{ELEMENT}\ \mathit{QName}\ (t) <: \texttt{ELEMENT}\ \mathit{NameSet}\ (t')$$

If $t <: t'$ and $m >= m'$ and $n <= n'$ then

$$t\ \texttt{min}\ m\ \texttt{max}\ n <: t'\ \texttt{min}\ m'\ \texttt{max}\ n'$$

If $t_1 <: t_1'$ and $t_2 <: t_2'$ then

$$t_1, t_2 <: t_1', t_2'$$
$$t_1, t_2 <: t_1' \ \&\ t_2'$$
$$t_1 \ \&\ t_2 <: t_1' \ \&\ t_2'$$

We write $t_1 = t_2$ if $t_1 <: t_2$ and $t_2 <: t_1$. Here are some of the equalities that hold:

$t\ \texttt{min}\ 1\ \texttt{max}\ 1$	$= t$
$(t_1, t_2), t_3$	$= t_1, (t_2, t_3)$
$t, ()$	$= t$
$(), t$	$= t$
$t, \varnothing$	$= \varnothing$
$\varnothing, t$	$= \varnothing$
$t_1 \mid t_2$	$= t_2 \mid t_1$
$(t_1 \mid t_2) \mid t_3$	$= t_1 \mid (t_2 \mid t_3)$
$t \mid \varnothing$	$= t$
$\varnothing \mid t$	$= t$
$t_1, (t_2 \mid t_3)$	$= (t_1, t_2) \mid (t_1, t_3)$
$(t_1 \mid t_2), t_3$	$= (t_1, t_3) \mid (t_2, t_3)$
$(t_1 \ \&\ t_2) \ \&\ t_3$	$= t_1 \ \&\ (t_2 \ \&\ t_3)$
$t_1 \ \&\ t_2$	$= t_2 \ \&\ t_1$
$t \ \&\ ()$	$= t$
$() \ \&\ t$	$= t$
$t \ \&\ \varnothing$	$= \varnothing$
$\varnothing \ \&\ t$	$= \varnothing$
$\texttt{ELEMENT}\ \mathit{NameSet}\ (t_1 \mid t_2)$	$= \texttt{ELEMENT}\ \mathit{NameSet}(t_1) \mid \texttt{ELEMENT}\ \mathit{NameSet}(t_2)$
$\texttt{ELEMENT}\ \mathit{NameSet}\ (t) \mid \texttt{ELEMENT}\ \texttt{*:*}\ (t)$	$= \texttt{ELEMENT}\ \texttt{*:*}\ (t)$
$t\ \texttt{min}\ 1\ \texttt{max}\ n{+}1$	$= t, (t\ \texttt{min}\ 0\ \texttt{max}\ n)$
$t\ \texttt{min}\ 0\ \texttt{max}\ n + 1$	$= () \mid t, (t\ \texttt{min}\ 0\ \texttt{max}\ n)$
$t\ \texttt{min}\ 0\ \texttt{max}\ 0$	$= ()$

t `min` 0 `max` 0	= ()
t `min` *m* `max` *n*	= Ø, if *m* > *n* or *m* = *

We also have that t_1<: t_2 if and only if t_1| t_2= t_2.

We define the *intersection* t_1 /\ t_2 of two types t_1 and t_2 to be the largest type *t* that is smaller than both t_1 and t_2. That is, *t* = t_1 /\ t_2 if *t* <: t_1 and *t* <: t_2 and if for any *t'* such that *t'* <: t_1 and *t'* <: t_2, we have *t'* <: *t*.

4.3 Environments

The type rules use an environment comprised of a type environment and a namespace environment, defined in [Figure 6].

T inTypeEnv = (*Variable* -> *Type*) ∪ (*QName* -> *QName*(*Type*$_1$, ..., *Type*$_n$) `returns` *Type*)	Type environment
NSinNamespaceEnv = *NCName* -> *anyURI*	Namespace environment
? inEnv = TypeEnv x NamespaceEnv	Static environment

Figure 6: Environments used in type-inference rules

The type environment T is a finite map from variables and function names to types.

Part of an attribute or element's type is its tag name, which is a *QName*. A *QName* contains a namespace prefix, whose meaning depends on the namespace environment NS, which is a finite map from namespace prefixes (NCNames) to namespace URIs. The namespace environment is modified by namespace declarations and by element constructors that contain namespace declarations.

To select the type-environment component of an environment, we use the notation T of ?; similarly, we use NS of ? to select the namespace environment.

We retrieve type information from the environment by writing T (*Variable*)= *t* to look up a variable, or by writing T(*QName*) = (*QName*(t_1, ..., t_n) `returns` *t*) to look up a function type. We write NS(*NCName*) = *anyURI* to look up a namespace prefix.

It is often necessary to modify an environment, for example, when defining variables or functions. The modification of one environment ? by another ?' is written ?, ?' and it denotes:
?, ?' = ((T of ?), (T' of ?'); (NS of ?), (NS' of ?'))

The finite map *f*, *g* is the finite map with domain Dom *f* ∪ Dom *g*, and values

```
(f, g)(a) = if a in Dom g then g(a) else f(a)
```

This definition means that when ``looking up'' a variable in the environment ?, ?', the environment ?' will always be searched first.

The type checking starts with an empty type environment. As type checking proceeds, new variables are added to the type environment and new namespace prefixes are added to the namespace environment. For instance, when typing the query of [2.4 Iteration], variable `$b` will be typed with `Book`, and added in the environment. This will result in a new environment:

```
G' = G, $b : Book
```

4.4 Expressions

We write ? |- *Expr* : *t* if in environment ? the expression *Expr* has type *t*. Below are all the rules except those for `for` and `typeswitch` expressions, which are discussed in later subsections.

The following rule states that in any environment ?, the literal *NCName* has the type `xs:NCName`. The two subsequent rules are analogous for strings and numerical values. For example, ? |- 1 : `xs:integer` and ? |- "two" : `xs:string`.

? |- *NCName*: `xs:NCName`

? |- *StringLiteral*: `xs:string`

? |- *NumericLiteral*: `xs:double`

The next rule simply states the variable *Variable* has type *t* in environment ?:

(T of ?)(*Variable*) = *t*

? |- *Variable* : *t*

Part of an attribute or element's type is its tag name, which is a *QName*. Before computing the type of an attribute or element, we convert the attribute or element's *QName* into an *expanded QName*. The inference rule below converts a *QName* into an *expanded QName* by mapping its namespace prefix to a namespace URI. Later rules rely on this judgement to resolve the namespace prefixes that occur in the *QNames* of elements and attributes.

QName = $NCName_1$:$NCName_2$ (NS of ?)($NCName_1$) = *anyURI*

? |- *QName* => {*anyURI*}$NCName_2$

The next two rules are for attribute and element construction with a constant tag name. The first rule states that if *Expr* has type *t*, and that *QName* resolves to the given *expanded QName*, then the attribute expression `ATTRIBUTE` *QName* (*Data*) has type `ATTRIBUTE` *expQName* (*t*). The subsequent rules are analogous for element expressions.

? |- *Expr* : *t* ? |- *QName* => *expQName*

? |- `ATTRIBUTE` *QName* (*Expr*): `ATTRIBUTE` *expQName* (*t*)

? |- *QName* => *expQName*

? |- <*QName*/> : `ELEMENT` *expQName* ()

? |- *Expr* : *t* ? |- *QName* => *expQName*

? |- <*QName*> { *Expr* } </*QName*> : `ELEMENT` *expQName* (*t*)

> **Ed. Note:** MF: (08-May-2001) The above rules do not handle namespace attributes in element constructors, e.g., `<foo:bar xmlns:foo="http://www.foo.com/foo.xsd">`. These need to be added. In mapping from XQuery to core, all namespace attributes should occur before other attributes.

The next rule is for element construction in which the tag name is computed. The built-in function `make-attribute` is used to construct an attribute with a computed tag; its typing rule appears in [4.6 Built-in functions]. The rule states that if $Expr_2$ has type *t*, then the element expression with computed tag $Expr_1$ has type `ELEMENT` *:* (*t*).

Ed. Note: MF: Note that the types are less precise than in Algebra, which supported name expressions and wildcard types. This probably makes the 80-20 cut.

$$\frac{? \vdash Expr_1 : \texttt{xs:QName} \qquad ? \vdash Expr_2 : t}{? \vdash \texttt{<\{} Expr_1 \texttt{\}>\{} Expr_2 \texttt{\}</>} : \texttt{ELEMENT *:*}(t)}$$

The following two rules are analogous to the sequence and empty sequence rules in [4.1 Relating data to types].

$$\frac{? \vdash Expr_1 : t_1 \qquad ? \vdash Expr_n : t_2}{? \vdash Expr_1, Expr_2 : t_1, t_2}$$

$$\frac{}{? \vdash () : ()}$$

Note that in the type rule for a conditional expression, the result type is the choice ($t_2|t_3$).

$$\frac{? \vdash Expr_1 : \texttt{xs:boolean} \qquad ? \vdash Expr_2 : t_2 \qquad ? \vdash Expr_3 : t_3}{? \vdash \texttt{if } (Expr_1) \texttt{ then } Expr_2 \texttt{ else } Expr_3 : (t_2 \mid t_3)}$$

A `let` expression extends the type environment ? by mapping *Variable* to type *t*. Note that $Expr_2$, the body of the `let` expression, is typed in the extended environment, and the type of the entire `let` expression is t_2.

$$\frac{? \vdash Expr_1 : t_1 \qquad ?, (Variable : t_1) \vdash Expr_2 : t_2}{? \vdash \texttt{let } Variable := Expr_1 \texttt{ return } Expr_2 : t_2}$$

The next rule is for function application. In a function application, the type of each actual argument to the function must be a *subtype* of the corresponding formal argument to the function, i.e., it is not necessary for the actual and formal types to be equal.

$$\frac{\begin{array}{c}(\text{T of } ?)(QName) = QName(t_1, \cdots, t_n) \texttt{ returns } t \\ ? \vdash Expr_1 : t'_1 \qquad t'_1 <: t_1 \\ \cdots \\ ? \vdash Expr_n : t'_n \qquad t'_n <: t_n\end{array}}{? \vdash QName\ (Expr_1, \cdots, Expr_n) : t}$$

The next rule states that it is always permissible to explicitly type an expression with a type *t'* that is a supertype of the expression's type *t*. In programming-language terminology, this operation is sometimes called an ``upcast''.

$$\frac{? \vdash Expr : t' \qquad t' <: t}{? \vdash (Expr : t) : t}$$

The `cast` operator converts the type of an expression to the given atomic simple type.

Ed. Note: MF: the relationship between *p* and *p'* depends on operator work.

$$\frac{? \ |\!\!- \ Expr : p \qquad p' <: p}{? \ |\!\!- \ \texttt{cast as} \ p' \ (Expr) : p'}$$

The `error` expression always has the empty choice type.

$$\frac{}{? \ |\!\!- \ \texttt{error} : \varnothing}$$

4.5 Operators

A joint task force on operators with members from the XSLT, XML Schema, and XML Query working groups is chartered to define the operators for XQuery and XPath 2.0. The complete set of operators will be defined in a forthcoming document by that group, so it is not possible to give the typing rules for all operators here. (See [Issue-0056: Operators on Simple Types]). In general, however, arithmetic operators will have a type rule such as the following, in which t_1 and t_2 are numeric types and appropriate type conversions exist between the two:

$$\frac{? \ |\!\!- \ Expr_1 : p_1 \qquad ? \ |\!\!- \ Expr_2 : p_2 \qquad t = \texttt{result-type}(p_1, p_2)}{? \ |\!\!- \ Expr_1 \ Op_{\text{arith}} \ Expr_2 : t}$$

Equality and inequality operators are typed similarly.

$$\frac{? \ |\!\!- \ Expr_1 : t_1 \qquad ? \ |\!\!- \ Expr_2 : t_2}{? \ |\!\!- \ Expr_1 \ Op_{\text{eq}} \ Expr_2 : \texttt{xs:boolean}}$$

Boolean operators require that their subexpressions be boolean expressions.

$$\frac{? \ |\!\!- \ Expr_1 : \texttt{xs:boolean} \qquad ? \ |\!\!- \ Expr_2 : \texttt{xs:boolean}}{? \ |\!\!- \ Expr_1 \ Op_{\text{bool}} \ Expr_2 : \texttt{xs:boolean}}$$

The type rules for the collection operators, Op_{coll}, are given in [4.7 Typing unordered expressions].

4.6 Built-in functions

The following type rules define the input and output types for built-in functions.

$$\frac{? \ |\!\!- \ Expr : \texttt{t}}{? \ |\!\!- \ \texttt{count}(Expr) : \texttt{xs:integer}}$$

The next two rules are for comment and processing instruction constructors.

$$\frac{? \ |\!\!- \ Expr : \texttt{xs:string}}{? \ |\!\!- \ \texttt{comment}(Expr) : \texttt{COMMENT}}$$

$$\frac{? \ |\!\!- \ Expr_1 : \texttt{xs:NCName} \qquad ? \ |\!\!- \ Expr_2 : \texttt{xs:string}}{? \ |\!\!- \ \texttt{processing-instruction}(Expr_1, Expr_2) : \texttt{PROCESSING-INSTRUCTION}}$$

In the core syntax, all character-data literals are expressed using the built-in `pcdata` (parsable character data) function. The type of PCDATA is always `xs:AnySimpleType`.

$$\frac{}{\textbf{|-}\ \texttt{pcdata(}\mathit{StringLiteral}\texttt{)} : \texttt{xs:AnySimpleType}}$$

Attributes with computed tag names are constructed by the `make-attribute`. This rule states that if $Expr_2$ has type t, then the attribute expression with computed tag $Expr_1$ has type `ATTRIBUTE` *:* (t).

$$\frac{?\ \textbf{|-}\ Expr_1 : \texttt{xs:QName} \qquad ?\ \textbf{|-}\ Expr_2 : t}{?\ \textbf{|-}\ \texttt{make-attribute}(Expr_1, Expr_2) : \texttt{ATTRIBUTE *:*}(t)}$$

The name of an attribute or element always has type `xs:QName`:

$$\frac{?\ \textbf{|-}\ Expr : \texttt{ATTRIBUTE}\ NameSet\ (t)}{?\ \textbf{|-}\ \texttt{name}(Expr) : \texttt{xs:QName}}$$

$$\frac{?\ \textbf{|-}\ Expr : \texttt{ELEMENT}\ NameSet(t)}{?\ \textbf{|-}\ \texttt{name}(Expr) : \texttt{xs:QName}}$$

The `attributes` and `children` accessors only apply to values with element type.

$$\frac{\begin{array}{c}?\ \textbf{|-}\ Expr : \texttt{ELEMENT}\ NameSet\ (t) \\ t <: t_1, t_2 \qquad t_1 < \texttt{xs:AnyAttribute*} \qquad t_2 < (\texttt{xs:AnyElement}\ \textbf{|}\ \texttt{xs:string})^*\end{array}}{?\ \textbf{|-}\ \texttt{attributes}(Expr) : t_1}$$

$$\frac{\begin{array}{c}?\ \textbf{|-}\ Expr : \texttt{ELEMENT}\ NameSet(t) \\ t <: t_1, t_2 \qquad t_1 < \texttt{xs:AnyAttribute*} \qquad t_2 < (\texttt{xs:AnyElement}\ \textbf{|}\ \texttt{xs:string})^*\end{array}}{?\ \textbf{|-}\ \texttt{children}(Expr) : t_2}$$

The next two rules are for name expressions: they extract the constituent parts of a name.

$$\frac{?\ \textbf{|-}\ Expr : t \qquad ?\ \textbf{|-}\ t <: \texttt{xs:AnyElement}\ \textbf{|}\ \texttt{xs:AnyAttribute}\ \textbf{|}\ \texttt{PROCESSING-INSTRUCTION}\ \textbf{|}\ \texttt{COMMENT}}{?\ \textbf{|-}\ \texttt{namespace-uri}(Expr) : \texttt{xs:uriReference}}$$

$$\frac{?\ \textbf{|-}\ Expr : t \qquad ?\ \textbf{|-}\ t <: \texttt{xs:AnyElement}\ \textbf{|}\ \texttt{xs:AnyAttribute}\ \textbf{|}\ \texttt{PROCESSING-INSTRUCTION}\ \textbf{|}\ \texttt{COMMENT}}{?\ \textbf{|-}\ \texttt{local-name}(Expr) : \texttt{xs:NCName}}$$

The next rule is for the `parent` function, which may be applied to any unit type:

$$\frac{?\ \textbf{|-}\ Expr : u}{?\ \textbf{|-}\ \texttt{parent}(Expr) : \texttt{xs:AnyElement?}}$$

The next two rules are for the reference constructor and dereference function. Note that `ref` requires that its argument have a unit type.

$$\frac{?\ \textbf{|-}\ Expr : u}{?\ \textbf{|-}\ \texttt{ref}(Expr) : \texttt{REFERENCE}(u)}$$

$$\frac{?\ \textbf{|-}\ Expr : \texttt{REFERENCE}(t)}{?\ \textbf{|-}\ \texttt{deref}(Expr) : t}$$

The next rule is for the `string-value` of a unit value.

? |- *Expr* : t

? |- `string-value`(*Expr*) : `xs:string`

The accessor `typed-value` returns the simple typed value of an attribute or element. An attribute always contains a simple typed value, therefore `typed-value` simply returns the attribute's typed content. An element may contain a simple-typed value or a complex-typed value. The second and third rules below distinguish between these two cases.

? |- *Expr* : `ATTRIBUTE` *NameSet* (*t*)

? |- `typed-value`(*Expr*) : *t*

? |- *Expr* : `ELEMENT` *NameSet* (*t*)   *t* <: `xs:AnySimpleType`

? |- `typed-value`(*Expr*) : *t*

? |- *Expr* : `ELEMENT` *NameSet* (*t*)   *t* <: `xs:AnyComplexType`

? |- `typed-value`(*Expr*) : `()`

4.7 Typing unordered expressions

The built-in functions `index` and `unordered` and the operator `sortby` disregard the relative order of components in their input type.

For example, consider the following.

```
    index (children (book0))
==> (<q:pair><q:fst>1</q:fst>
            <q:snd><q:ref><title>Data on the Web</title></q:ref></q:snd></q:pair>,
     <q:pair><q:fst>2</q:fst>
            <q:snd><q:ref><author>Abiteboul</author></q:ref></q:snd></q:pair>,
     <q:pair><q:fst>3</q:fst>
            <q:snd><q:ref><author>Buneman</author></q:ref></q:snd></q:pair>,
     <q:pair><q:fst>4</q:fst>
            <q:snd><q:ref><author>Suciu</author></q:ref></q:snd></q:pair>,
     <q:pair><q:fst>5</q:fst>
            <q:snd><q:ref><year>1999<year></q:ref></q:snd></q:pair>)
```

By nesting the children of `book0` under `snd` the original sequence of `title, author+, year` gets lost. The `snd` element can contain either a `title`, an `author`, or a `year`. To compute a meaningful type for this expression, we need to find a type *q*, *m*, and *n* such that

```
children(book0) : q min m max n
```

and then the type is given by

```
index(children(book0)) :
ELEMENT q:pair(
  ELEMENT q:fst (xs:integer),
  ELEMENT q:snd (REFERENCE (ELEMENT author (q)))
) min m max n
```

In the case of books, the values of *q* are:

```
ELEMENT title (xs:string)  |
ELEMENT author (xs:string)
```

the value of m is three (because there will be one `title`, at least one `author`, and one `year` element) and the value of n is * (because there may be any number of author elements).

We call a type like q a *prime* type. In general, it may contain scalar, attribute, element, choice, and empty choice types, but it will not contain repetition, sequence, or empty sequence types (except, perhaps, within an element or attribute type). The definition of prime types appears in [Figure 4].

factor (p)	= p `min` 1 `max` 1
factor (`ELEMENT` *NameSet* (t))	= `ELEMENT` *NameSet* (t) `min` 1 `max` 1
factor (`ATTRIBUTE` *NameSet* (t))	= `ATTRIBUTE` *NameSet* (t) `min` 1 `max` 1
factor (t_1 , t_2)	= (q_1| q_2) `min` m_1+ m_2 `max` n_1+ n_2 where q_i `min` m_i `max` n_i = *factor* (t_i)
factor (t_1 & t_2)	= (q_1| q_2) `min` m_1+ m_2 `max` n_1+ n_2 where q_i `min` m_i `max` n_i = *factor* (t_i)
factor (t_1| t_2)	= (q_1| q_2) `min` (min m_1 m_2) `max` (max n_1 n_2) where q_i `min` m_i `max` n_i = *factor* (t_i)
factor (t `min` m `max` n)	= q `min` $m \cdot m'$ `max` $n' \cdot n$ where q `min` m' `max` n' = *factor* (t)
factor (())	= Ø `min` 0 `max` 0
factor (Ø)	= Ø `min` * `max` 0

Figure 7: Definition of factoring

The *factor* function, as shown in [Figure 7], converts any type t to a type of the form q `min` > `max` n, where t <: q `min` > `max` n, so that any value that has type t also has type q `min` m `max` n. For example,

```
factor (ELEMENT title (xs:String),
        ELEMENT author (xs:String) min 1 max *,
        ELEMENT year (xs:integer)) =

  (ELEMENT title (xs:string) |
   ELEMENT author (xs:string) |
   ELEMENT year(xs:integer)) min 3 max *
```

We can see here that the factored type is less specific than the unfactored type. For convenience we write q `min` m `max` n = *factor* (t), but one should actually think of the function as returning a triple consisting of a prime type q and two bounds m and n.

Just as factoring a number yields a product of prime numbers, factoring a type yields a repetition of prime types. Further, the result yielded by factoring is in some sense optimal. If q `min` m `max` n = *factor* (t) then t <: q `min` m `max` n and for any q', m', and n' such that t <: q'`min` m `max` n' we have that q <: q' and m >= m' and n <= n'. Also, if t = t', then *factor* (t) = *factor* (t'). In particular, the choice of the lower bound `*` for *factor* (Ø) guarantees that *factor* (t) = *factor* (t | Ø), since m min `*` = m.

Using *factor*, we can type `index`, `sortby`, `unordered` and the `union`, `difference`, and `intersection` operations. Note that *factor* is only used by the type inference rules; it is not part of XQuery expressions.

The type rule for `index` requires that its argument be a factored type. The second expression above the judgement line converts t into a factored type.

$$\frac{? \vdash Expr : t \qquad q \texttt{ min } m \texttt{ max } n = factor\ (t)}{? \vdash \texttt{index}(Expr) : \texttt{ELEMENT q:pair(ELEMENT q:fst (xs:integer), ELEMENT q:snd (REFERENCE (}q\texttt{))) min m max } n}$$

The types of aggregated expressions must be factored, and their prime type must be a numeric type.

$$\frac{? \vdash Expr : t \quad q \texttt{ min } m \texttt{ max } n = \texttt{factor}\ (t) \quad q <: \texttt{xs:decimal} \mid \texttt{xs:double}}{? \vdash \texttt{agg } Expr : q}$$

`agg` is one of `avg`, `max`, `min`, `sum`.

The `sortby` operator returns the `factor` of its input type.

$$\frac{? \vdash Expr_1 : t_1 \quad ? \vdash Expr_1 : t_2 \quad q \texttt{ min } m \texttt{ max } n = \texttt{factor}(t_1)}{? \vdash Expr_1 \texttt{ sortby } Expr_2 \texttt{ (ascending|descending)}: q \texttt{ min } m \texttt{ max } n}$$

> **Ed. Note:** MF, Oct 23/2000: This definition assumes that the equality operator on t_2 is defined. An alternative is requiring $Expr_2$ to have `xs:AnySimpleType`, but that seems too restrictive.

The `distinct-node` function removes all duplicate nodes from its input.

$$\frac{? \vdash Expr : t \qquad q \texttt{ min } m \texttt{ max } n = \texttt{factor}(t)}{? \vdash \texttt{distinct-node}(Expr) : q \texttt{ min } (1 \text{ min } m) \texttt{ max } n}$$

> **Ed. Note:** PF (Jan 29, 2000): a more accurate lower bound may be derived by counting the disjoint constituent types of q. But this is probably too complex.

The `distinct-value` function removes all duplicate values from its input. The typing rule is analogous to that of `distinct-node`, except the lower bound m can be at most one.

$$\frac{? \vdash Expr : t \qquad q \texttt{ min } m \texttt{ max } n = \texttt{factor}(t) \qquad m' = 1 \text{ min } m}{? \vdash \texttt{distinct-value}(Expr) : q \texttt{ min } m' \texttt{ max } n}$$

Similar to `distinct-value` and `distinct-node`, the `intersect` and `except` operators need to set the lower bound of the cardinality of the input type to 0. The upper bound of the cardinality of the intersection of two types can be at most the minimum of upper bounds of their individual cardinalities.

$$\frac{? \vdash Expr_1 : t_1 \qquad ? \vdash Expr_2 : t_2 \qquad q \texttt{ min } m \texttt{ max } n = factor\ (t_1, t_2)}{? \vdash Expr_1 \texttt{ union } Expr_2 : q \texttt{ min } m \texttt{ max } n}$$

$$\frac{? \;\texttt{|-}\; Expr_1 : t_1 \qquad ? \;\texttt{|-}\; Expr_2 : t_2 \qquad q_1 \;\texttt{min}\; m \;\texttt{max}\; n = factor\,(t_1) \qquad q_2 \;\texttt{min}\; m \;\texttt{max}\; n = factor\,(t_2)}{? \;\texttt{|-}\; Expr_1 \;\texttt{intersect}\; Expr_2 : (q_1 \;/\backslash\; q_2) \;\texttt{min}\; 0 \;\texttt{max}\; (m_2 \;\mathrm{min}\; n_2)}$$

$$\frac{? \;\texttt{|-}\; Expr_1 : t_1 \qquad ? \;\texttt{|-}\; Expr_2 : t_2 \qquad q_1 \;\texttt{min}\; m_1 \;\texttt{max}\; n_1 = factor\,(t_1) \qquad q_2 \;\texttt{min}\; m_2 \;\texttt{max}\; n_2 = factor\,(t_2)}{? \;\texttt{|-}\; Expr_1 \;\texttt{except}\; Expr_2 : q_1 \;\texttt{min}\; 0 \;\texttt{max}\; n_1}$$

The `unordered` function ignores the order of the items in its sequence argument, therefore its type is also a factored type.

$$\frac{? \;\texttt{|-}\; Expr : t \qquad q \;\texttt{min}\; m \;\texttt{max}\; n = factor\,(t)}{? \;\texttt{|-}\; \texttt{unordered}(Expr) : q \;\texttt{min}\; m \;\texttt{max}\; n}$$

4.8 Iteration expressions

The typing of `for` expressions is rather subtle. We give an intuitive explanation first and then the detailed typing rules below.

A *unit* type is defined in **[Figure 4]**; it is either an simple type, an attribute or element type with a constant or wildcard name, a comment, or a processing instruction. A `for` expression

`for` *Variable* `in` $Expr_1$ `return` $Expr_2$

is typed as follows. First, one finds the type of expression $Expr_1$. Next, for each unit type in this type one assumes the variable *Variable* has the unit type and one types the body $Expr_2$. Note that this means we may type the body of $Expr_2$ several times, once for each unit type in the type of $Expr_1$. Finally, the types of the body $Expr_2$ are combined, according to how the types were combined in $Expr_1$. That is, if the type of $Expr_1$ is formed with sequencing, then sequencing is used to combine the types of $Expr_2$, and similarly for choice or repetition.

For example, consider the following expression, which selects all `author` elements from a book.

```
for $c in children($book0) return
  typeswitch ($c)
    case $a : ELEMENT author(xs:AnyType) return $a
    default return ()
```

The type of `children(book0)` is

```
ELEMENT title(xs:String), ELEMENT year(xs:integer), ELEMENT author(xs:string)+
```

This is composed of three unit types, and so the body is typed three times.

assuming $c has type	`ELEMENT title(xs:string)`	the body has type	`()`
"	`ELEMENT year(xs:integer)`	"	`()`
"	`ELEMENT author(xs:string)`	"	`ELEMENT author(xs:String)`

The three result types are then combined in the same way the original unit types were, using sequencing and iteration.

```
(), (), ELEMENT author(xs:string)+
```

as the type of the iteration, and simplifying yields

```
ELEMENT author(xs:string)+
```

as the final type.

As a second example, consider the following expression, which selects all `title` and `author` elements from a book, and renames them.

```
for $c in $book0/* return
  typeswitch ($c)
    case ELEMENT title (xs:string)  return
         <titl> { t/data() } </titl>
    case ELEMENT year (xs:integer)  return
         ()
    case ELEMENT author (xs:string) return
         <auth> { a/data() } </auth>
    default return ERROR
```

Again, the type of `book0/*` is

```
ELEMENT title(xs:string), ELEMENT year(xs:integer), ELEMENT author(xs:string)+
```

This is composed of three unit types, and so the body is typed three times.

assuming $c has type	`ELEMENT title(xs:string)`	the body has type	`ELEMENT titl(xs:string)`
"	`ELEMENT year(xs:integer)`	"	`()`
"	`ELEMENT author(xs:string)`	"	`ELEMENT auth(xs:string)`

The three result types are then combined in the same way the original unit types were, using sequencing and iteration. This yields

```
ELEMENT titl(xs:string), (), ELEMENT auth(xs:string)+
```

as the type of the iteration, and simplifying yields

```
ELEMENT titl(xs:string), ELEMENT auth(xs:string)+
```

as the final type. Note that the title occurs just once and the author occurs one or more times, as one would expect.

As a third example, consider the following expression, which selects all basic parts from a sequence of parts.

```
for $p in $part0/subparts/* return
  typeswitch ($p)
    case Basic     return $b
    case Composite return ()
    default return ERROR
```

The type of `part0/subparts/*` is

```
(Basic | Composite)+
```

This is composed of two unit types, and so the body is typed two times.

if `$p` has type	`Basic`	the body has type	`Basic`
"	`Composite`	"	`()`

The two result types are then combined in the same way the original unit types were, using sequencing and iteration. This yields

```
(Basic | ())+
```

as the type of the iteration, and simplifying yields

```
Basic*
```

as the final type. Note that although the original type involves repetition one or more times, the final result is a repetition zero or more times. This is what one would expect, since if all the parts are composite the final result will be an empty sequence.

In this way, we see that `for` expressions can be combined with `typeswitch` expressions to select and rename elements from a sequence, and that the result is given a sensible type.

In order for this approach to typing to be sensible, it is necessary that the unit types can be uniquely identified. However, the type system given here satisfies the following law.

```
ELEMENT a (t1 | t2) = ELEMENT a (t1) | ELEMENT a (t2)
```

This has one unit type on the left, but two distinct unit types on the right, and so might cause trouble. Fortunately, our type system inherits an additional restriction from XML Schema: we insist that the regular expressions can be recognized by a top-down deterministic automaton. In that case, the regular expression must have the form on the left, the form on the right is outlawed because it requires a non-deterministic recognizer. With this additional restriction, there is no problem.

The method of translating projection to iteration described in the previous section combined with the typing rules given here yield optimal types for projections, in the following sense. Say that variable *x* has type *t*, and the projection *x / a* has type *t'*. The type assignment is *sound* if for every value of type *t*, the value of *x / a* has type *t'*. The type assignment is *complete* if for every value *y* of type *t'* there is a value *x* of type *t* such that *x / a = y*. In symbols, we can see that these conditions are complementary.

sound:	`forall` *x* in *t*. `exist` *y* in *t'*. *x / a = y*
complete:	`forall` *y* in *t'*. `exist` *x* in *t*. *x / a = y*

Any sensible type system must be sound, but it is rare for a type system to be complete. But, remarkably, the type assignment given by the above approach is both sound and complete.

The type rule for for expressions uses the following auxiliary judgement. We write ? **|-** `for` *Variable* : *t* `return` *Expr* : *t'*, if in environment ? when the bound variable *Variable* of an iteration has type *t* then the body *Expr* of the iteration has type *t'*.

? , *Variable* : *t* **|-** *Expr* : *t'*

? **|-** `for` *Variable* : *t* `return` *Expr* : *t'*

? **|-** `for` *Variable* : `()` `return` *Expr* : `()`

$$\frac{? \vdash \texttt{for}\ \textit{Variable} : t_1\ \textit{Expr} : t'_1 \qquad ? \vdash \texttt{for}\ \textit{Variable} : t_2\ \textit{Expr} : t'_2}{? \vdash \texttt{for}\ \textit{Variable} : t_1, t_2\ \texttt{return}\ \textit{Expr} : t'_1, t'_2}$$

$$\frac{? \vdash \texttt{for}\ \textit{Variable} : t_1\ \textit{Expr} : t'_1 \qquad ? \vdash \texttt{for}\ \textit{Variable} : t_2\ \textit{Expr} : t'_2}{? \vdash \texttt{for}\ \textit{Variable} : t_1\ \&\ t_2\ \texttt{return}\ \textit{Expr} : t'_1\ \&\ t'_2}$$

$$\frac{}{? \vdash \texttt{for}\ \textit{Variable} : \varnothing\ \texttt{return}\ \textit{Expr} : \varnothing}$$

$$\frac{? \vdash \texttt{for}\ \textit{Variable} : t_1\ \textit{Expr} : t'_1 \qquad ? \vdash \texttt{for}\ \textit{Variable} : t_2\ \textit{Expr} : t'_2}{? \vdash \texttt{for}\ \textit{Variable} : t_1 \mid t_2\ \texttt{return}\ \textit{Expr} : t'_1 \mid t'_2}$$

$$\frac{? \vdash \texttt{for}\ \textit{Variable} : t\ \textit{Expr} : t'}{? \vdash \texttt{for}\ \textit{Variable} : (t\ \texttt{min}\ m\ \texttt{max}\ n)\ \texttt{return}\ \textit{Expr} : (t'\ \texttt{min}\ m\ \texttt{max}\ n)}$$

Given the above rules, the type rules for `for` expressions are immediate.

$$\frac{? \vdash \textit{Expr}_1 : t_1 \qquad ? \vdash \texttt{for}\ \textit{Variable} : t_1\ \texttt{return}\ \textit{Expr}_2 : t_2}{? \vdash \texttt{for}\ \textit{Variable}\ \texttt{in}\ \textit{Expr}_1\ \texttt{return}\ \textit{Expr}_2 : t_2}$$

4.9 Typeswitch expressions

The typing of `typeswitch` expressions is closely related to the typing of `for` expressions. Due to the typing rules of `for` expressions, it is possible that the body of an iteration is checked many times. Thus, when a `typeswitch` expression is checked, it is possible that quite a lot is known about the type of the expression being matched, and one can determine that only some of the clauses of the `typeswitch` apply. The definition of `typeswitch` uses the auxiliary judgements to check whether a given clause is applicable.

We write $? \vdash$ `case` *Variable* : *t* `return` *Expr* : *t'*, if in environment ?, the *Variable* of the `typeswitch` has type *t*, then the body *Expr* of `case` has type *t'*. Note the type of the body is irrelevant if $t = \varnothing$.

$$\frac{\neg(t = \varnothing) \qquad ?, \textit{Variable} : t \vdash \textit{Expr} : t'}{? \vdash \texttt{case}\ \textit{Variable} : t\ \texttt{return}\ \textit{Expr} : t'}$$

$$\frac{}{? \vdash \texttt{case}\ \textit{Variable} : \varnothing\ \texttt{return}\ \textit{Expr} : \varnothing}$$

We write $? \vdash t <: t'$ `default return` *Expr*: *t''* if in environment ? when $t <: t'$ does not hold, then the body *Expr* of the `typeswitch` expression's `default return` clause has type *t''*. Note that the type of the body is irrelevant if $t < t'$.

$$\frac{t <: t'}{? \vdash t <: t'\ \texttt{else}\ \textit{Expr} : \varnothing}$$

$$\frac{\neg\ t <: t' \qquad ? \vdash \textit{Expr} : t''}{? \vdash t <: t'\ \texttt{else}\ \textit{Expr} : t''}$$

Given the above, it is straightforward to construct the typing rule for a `typeswitch` expression. Recall that we write *t* /\ *t'* for the intersection of two types.

$$\frac{\begin{array}{c} ? \mid\!\!- Expr_0 : t_0 \\ ? \mid\!\!- \texttt{case}\ Variable : t_0 \ /\backslash\ t_1\ \texttt{return}\ Expr_1 : t'_1 \\ \ldots \\ ? \mid\!\!- \texttt{case}\ Variable : t_0 \ /\backslash\ t_n\ \texttt{return}\ Expr_n : t'_n \\ ? \mid\!\!- t_0 < t_1 \mid \ldots \mid t_n\ \texttt{default return}\ Expr_{n+1} : t'_{n+1} \end{array}}{\begin{array}{l} ? \mid\!\!-\ (\texttt{typeswitch}\ (Expr_0)\ \texttt{as}\ Variable \\ \qquad \texttt{case}\ t_1\ \texttt{return}\ Expr_1 \\ \qquad \ldots \\ \qquad \texttt{case}\ t_n\ \texttt{return}\ Expr_n \\ \qquad \texttt{default return}\ Expr_{n+1}) \qquad : t'_1 \mid \ldots \mid t'_{n+1} \end{array}}$$

4.10 Typing descendent-or-self

The built-in function `DESCENDENT-OR-SELF(expr)` implements the descendent-or-self axis of XPath, and is used to express the sematics of the `//` operator. For example, consider the following type declaration:

```
TYPE Part = Basic | Composite
TYPE Basic =
  ELEMENT basic (
    ELEMENT cost (xs:integer)
  )
TYPE Composite =
  ELEMENT composite (
    ELEMENT assembly_cost (xs:integer)
    ELEMENT subparts (Part+)
  )
```

If `$v` has type `Part`, the XPath expression `$v//basic` would retrieve all basic parts. We would rewrite this query into the core language as follows:

```
for $x in descendent-or-self($v) return
  typeswitch ($x) as $z
    case ELEMENT basic(xs:AnyType) return $z
    default return ()
```

The type of `descendent-or-self($v)` is

```
(ELEMENT basic (
     ELEMENT cost (xs:integer)
   )
 | ELEMENT cost (xs:integer)
 | ELEMENT composite (
     ELEMENT assembly_cost (xs:integer)
     ELEMENT subparts (Part+)
   )
 | ELEMENT assembly_cost ( xs:integer )
 | ELEMENT subparts ( Part+)
       | xs:integer
 ) min 3 max *
```

The lower bound of 3 indicates that `descendent-or-self($v)` returns at least three nodes, `ELEMENT basic (...)`, `ELEMENT cost (...)`, and `xs:integer`, which is what happens when `$v` is bound to a single element of type `Basic`. Based on this type, the type of `$v//basic` is

```
ELEMENT basic (
        ELEMENT cost (xs:integer)) min 0 max *
```

On the other hand, the type of `$v//part` is `()`, the empty sequence, because no `part` element appears in the type `Part`. (An expression with type `()` typically indicates a programming error, unless the expression is `()` itself.)

The typing of `descendent-or-self` loses all information about the relative ordering of the subparts. For instance, in `descendent-or-self($v)` it is always the case that a `basic` element is immediately followed (in document order) by a `cost` element, but this is not reflected in the type above.

There are two reasons why information about order is not retained.

First, there may exist two locally declared elements, e.g., `ELEMENT foo(type1)` and `ELEMENT foo (type2)`, with the same name but different content-type. Maintaining their relative order at type level would generate the invalid type `ELEMENT foo(type1) ... ELEMENT foo(type2)`, whereas `ELEMENT foo(type1) | ELEMENT foo(type2)` can be transformed to the valid type `ELEMENT foo( type1 | type2)`.

Second, for some recursive types, maintaining the relative order of elements leads to a context-free type. For example, with

```
TYPE type1 = ELEMENT a (type1 min 0 max 1), ELEMENT b ()
```

the type of descendents would need to be the context-free type

```
TYPE type1' = ELEMENT a (type1 min 0 max 1), type1', ELEMENT b ()
```

but there is no equivalent regular type expressible in our system.

In principle, `DESCENDENT-OR-SELF(expr)` could be expressed as a recursive user defined function:

```
DEFINE FUNCTION descendent-or-self (xs:AnyTree $x)
       RETURNS (xs:AnyElement | xs:AnyScalar) min 0 max * {
  TYPESWITCH ($x)
     CASE (Comment|PI|AnyAttribute) RETURN ()
     DEFAULT RETURN
       ($x, FOR $z IN children($x) RETURN descendent-or-self($z))
}
```

However, this loses far too much type information. With this definition and the translation above the type of `$v//basic` is

```
ELEMENT basic(xs:AnyType) min 0 max *
```

rather than

```
ELEMENT basic(ELEMENT cost(xs:integer)) min 0 max *
```

Similarly, the type of `$v//part` is (`ELEMENT part (AnyType) min 0 max *`) rather than `()`, losing the chance to uncover a type error.

The rule for typing `descendent-or-self(expr)` is as follows.

$$\frac{? \vdash \mathit{Expr} : \mathit{type}}{? \vdash \texttt{descendent-or-self}(\mathit{Expr})\ \texttt{:}\ \mathit{recfactor}(\mathit{type};\ \varnothing)}$$

This uses the function recfactor defined in [Figure 8]. The function *recfactor*() is similar to *factor*(), but it applies to all the descendents of a node.

To be precise, *recfactor*(*t*; *E*) takes a type *t* and a choice of type variables *E*. The argument *E* is a factoring environment: initially the empty choice Ø, it collects all the type variables encountered so far in a recursive traversal of the type. If *t* is a type in which all elements have empty content models, then *recfactor*(*t*; Ø) = *factor*(*t*).

recfactor (*p*; *E*)	= *p* `min` 1 `max` 1
recfactor (`ELEMENT` *NameSet* (*t*); *E*)	= (`ELEMENT` *NameSet* (*t*) | *q*) `min` *m* + 1 `max` *n* + 1) where *q* `min` *m* + 1 n + 1 = *recfactor* (*t*; *E*)
recfactor (`ATTRIBUTE` *NameSet* (*t*); *E*)	= Ø `min` 0 `max` 0
recfactor (t_1 , t_2; *E*)	= (q_1| q_2) `min` m_1+ m_2 `max` n_1+ n_2 where q_i `min` m_i `max` n_i = *recfactor* (t_i; *E*)
recfactor (t_1 & t_2; *E*)	= (q_1| q_2) `min` m_1+ m_2 `max` n_1+ n_2 where q_i `min` m_i `max` n_i = *recfactor* (t_i; *E*)
recfactor (t_1| t_2; *E*)	= (q_1| q_2) `min` (min m_1 m_2) `max` (max n_1 n_2) where q_i `min` m_i `max` n_i = *recfactor* (t_i; *E*)
recfactor (*t* `min` *m* `max` *n*)	= *q* `min` *m* · *m'* `max` *n*' · *n* where *q* `min` *m*' `max` *n*' = *recfactor* (*t*; *E*)
recfactor ((); *E*)	= Ø `min` 0 `max` 0
recfactor (Ø; *E*)	= Ø `min` * `max` 0
recfactor (*X*; *E*)	= *recfactor* (`def`(*X*);*E* | *X*) if `not` *X* `<:` *E*
recfactor (*X*; *E*)	= *factor* (`def`(*X*)) `min` * `max` * if *X* `<:` *E*

Figure 8: Definition of recursive factoring

4.11 Top-level declarations and query expressions

We write ? |- *QueryModule* if in environment ?, the query module *QueryModule* is well-typed.

Context declarations are always well typed. Below, they have the effect of updating the namespace environment.

$$\frac{}{? \vdash \texttt{namespace}\ \mathit{NCName} = \mathit{StringLiteral}}$$

$$\frac{}{? \vdash \texttt{default namespace} = \mathit{StringLiteral}}$$

A type declaration is always well typed:

$$\frac{}{? \ |\!\!- \ \texttt{type}\ NCName = t}$$

A function declaration is well-typed if in the environment extended with the type assignments for its formal variables, its body is well-typed.

$$\frac{?\,,(Variable_1 : t_1), \cdots (Variable_n : t_n)\ |\!\!-\ Expr : t' \qquad t' <: t}{? \ |\!\!- \ \texttt{define function}\ QName\ (\,t_1\ Variable_1, \cdots t_n\ Variable_n)\ \texttt{returns}\ t\ \{\ Expr\ \}}$$

The function *static-env* constructs an environment by extracting type assertions and namespace mappings from top-level declarations. It constructs a two part environment: the first part is the type environment, T, and the second part is the namespace environment, NS.

static-env (`type` *x* = *t*) =
(); ())

static-env (`define function` *QName* ($Type_1$ $Variable_1$, ···, $Type_n$ $Variable_n$) `returns` *t* { *Expr* } =
((*QName* |=> *QName*(t_1, ···, t_n) `returns` *t*); ())

static-env (`namespace` *NCName* = *StringLiteral*) =
((); *NCName* |-> *StringLiteral*)

static-env (`default namespace` = *StringLiteral*) =
((); '' |-> *StringLiteral*)

Note that the default namespace is represented in the namespace environment by the empty string.

We write |- *QueryModule* if each declaration and expression in the query module is well typed.

$$\frac{\begin{array}{c} ?\ |\!\!-\ static\text{-}env\ (ContextDecl_1),\ldots,\ static\text{-}env\ (ContextDecl_k), \\ static\text{-}env\ (TypeDecl_1),\ldots,\ static\text{-}env\ (TypeDecl_l), \\ static\text{-}env\ (FunctionDecl_1),\ldots,\ static\text{-}env\ (FunctionDecl_m) \\ ?\ |\!\!-\ ContextDecl_1 \quad \ldots \quad ?\ |\!\!-\ ContextDecl_k \\ ?\ |\!\!-\ TypeDecl_1 \quad \ldots \quad ?\ |\!\!-\ TypeDecl_l \\ ?\ |\!\!-\ FunctionDecl_1 \quad \ldots \quad ?\ |\!\!-\ FunctionDecl_m \\ ?\ |\!\!-\ Expr_1 \quad \ldots \quad ?\ |\!\!-\ Expr_n \end{array}}{\begin{array}{c} |\!\!-\ ContextDecl_1 \ldots\ ContextDecl_k \\ TypeDecl_1 \ldots\ TypeDecl_l \\ FunctionDecl_1 \ldots\ FunctionDecl_m \\ Expr_1 \ldots\ Expr_n \end{array}}$$

5 Dynamic Semantics : Value-Inference Rules

XQuery's dynamic, or operational, semantics is presented as value inference rules. The value inference rules are similar to the type inference rules in **[4 Static Semantics : Type-Inference Rules]**, but they relate expressions to values or *semantic objects*. XQuery's semantic objects are defined in the [XQuery 1.0 and XPath 2.0 Data Model]. An operational semantics specifies the order in which an XQuery expression is evaluated and guarantees that every expression can be reduced to a simple semantic object. XQuery's dynamic semantics is modeled on the dynamic semantics presented in [Milner].

5.1 Semantic objects

	ERROR
a	in SimpleValue
n	in Node
u	in UnitValue = SimpleValue ∪ Node
s	in Sequence<SimpleValue | Node>
v	in Values = Sequence<SimpleValue | Node> ∪ {ERROR}
sc	in SchemaComponent

Figure 9: Semantic objects

There are five categories of values in [XQuery 1.0 and XPath 2.0 Data Model]: *error*, *simple values*, *nodes*, *sequences*, and *schema components*. There is a single error value, named ERROR. Simple values are the union of all the value spaces of XML Schema simple types. A node is one of eight node kinds: element, attribute, namespace, comment, processing-instruction, text, and reference. A unit value is the union of all simple values and nodes; a unit value is an instance of the unit type [**Figure 4**].

A sequence is an ordered collection of nodes, simple values, or any mixture of nodes and simple values. A sequence cannot be a member of a sequence. An important characteristic of the data model is there is no distinction between a unit value (i.e., a node or a simple value) and a singleton sequence containing that value, i.e., a unit value is equivalent to a singleton sequence containing that value and vice versa.

Value is the class of values that includes all sequences and ERROR.

A schema component represents the type of element nodes, attribute nodes, and simple values.

All the above classes, except ERROR, are infinite sets.

[**Figure 10**] lists several basic functions from [XQuery 1.0 and XPath 2.0 Data Model] used in the dynamic semantics.

`append`	Construct a sequence from two or more sequences
`empty-sequence`	Constructs the empty sequence.
`empty`	Returns true if argument is empty sequence, otherwise false.
`head`	Returns the first node in a non-empty sequence
`tail`	Returns items in a sequence excluding its first item.
`schema-component`	Returns the schema component representing its type argument

Figure 10: Functions used in dynamic semantics

Ed. Note: MF : (Jan-15-2001) To define the sort expression completely, we need to specify what basic sort operators are available.

5.2 Environments

The value inference rules use an environment, defined in [Figure 11], comprised of a static environment, a value environment, and a function environment.

?	Static environment
VE in ValueEnv = *Variable* -> Value	Value environment
FE in FuncEnv = *QName* -> (*Expr* x *Variable**)	Function environment
? in Env = StaticEnv x ValueEnv x FuncEnv	

Figure 11: Environments used in value-inference rules

The static environment ? is defined by the static type rules [4 Static Semantics : Type-Inference Rules]. The value environment VE is a finite map from variables to values. The function environment FE is a finite map from function names to pairs containing an expression that is the body of the function and a list of free variables that are the function's formal arguments. An environment ? is a tuple containing a static enviroment, a value environment, and a function environment.

In the value-inference rules, we use the same notation to select an environment and to lookup values in an environment as used in [4 Static Semantics : Type-Inference Rules].

To select the static-environment component of an environment, we use the notation ? of ?; similarly, we use VE of ? to select the value environment and FE of ? for the function environment.

We look up the type of a variable by writing (? of ?)(*Variable*) = *t*, and we write (VE of ?)(*Variable*) = *v* to look up the value of a variable or (FE of ?)(*QName*) = *f* to look up a function.

It is often necessary to modify an environment, for example, when defining variables or functions. The modification of one environment ? by another ?' is written ?, ?' and it denotes:
?, ?' = ((? of ?), (?' of ?'); (VE of ?), (VE' of ?'); (FE of ?), (FE' of ?'))

The finite map *f*, *g* is the finite map with domain Dom *f* ∪ Dom *g*, and values

```
(f, g)(a) = if a in Dom g then g(a) else f(a)
```

This definition means that when ``looking up'' a variable in the environment ?, ?', the environment ?' will always be searched first.

5.3 Expressions

Each rule of the dynamic semantics are inferences among sentences of the form: ? |- *phrase* => *v*, where ? is an environment, *phrase* is a phrase in the core syntax [Figure 1], and *v* is a semantic object.

As in [4 Static Semantics : Type-Inference Rules], when all judgements above the line of an inference rule hold, then the judgement below the line holds as well. The following three rules state that in any environment, a string, numeric, or boolean literal reduces to the value it denotes. The XQuery/XPath 2.0 function library provides constructors for all simple-typed values. Those constructors are used here:

$$\frac{}{?\ \text{|-}\ \mathit{StringLiteral} \Rightarrow \text{xfo:string}(\mathit{StringLiteral})}$$

$$\frac{}{?\ \text{|-}\ \mathit{NumericLiteral} \Rightarrow \text{xfo:double}(\mathit{NumericLiteral})}$$

$$\frac{}{?\ \text{|-}\ \mathit{BooleanLiteral} \Rightarrow \text{xfo:boolean}(\mathit{BooleanLiteral})}$$

The next rule determines the value of a variable, which is simply the variable's value in the current value environment VE:

$$\frac{(\text{VE of } ?)(\mathit{Variable}) = v}{?\ \text{|-}\ \mathit{Variable} \Rightarrow v}$$

$$\frac{(\text{NS of } ?)(\mathit{NCName}_1) \Rightarrow v}{?\ \text{|-}\ \mathit{NCName}_1 : \mathit{NCName}_2 \Rightarrow \text{xfo:expanded-QName}(v, \mathit{NCName}_2)}$$

$$\frac{?\ \text{|-}\ \mathit{Expr} \Rightarrow v}{?\ \text{|-}\ \{\ \mathit{Expr}\ \} \Rightarrow v}$$

The next rule constructs an attribute node from its name and value sub-expressions. This is the first rule that uses the static type environment: the run-time type `schema-component(t)` of the attribute is obtained from the type environment ?.

$$\frac{?\ \text{|-}\ \mathit{NameSpec} \Rightarrow v_1 \qquad ?\ \text{|-}\ \mathit{Expr} \Rightarrow v_2 \qquad (?\ \text{of}\ ?)\ \text{|-}\ \texttt{ATTRIBUTE}\ \mathit{NameSpec}\ (\mathit{Expr}) : t}{?\ \text{|-}\ \texttt{ATTRIBUTE}\ \mathit{NameSpec}\ (\mathit{Expr}) \Rightarrow \text{attribute-node}(v_1, v_2, \text{schema-component}(t))}$$

The next rule constructs an empty element.

$$\frac{?\ \text{|-}\ \mathit{NameSpec} \Rightarrow v_1 \qquad ?\ \text{of}\ ?\ \text{|-}\ <\mathit{NameSpec}/> : t}{?\ \text{|-}\ <\mathit{NameSpec}/> \Rightarrow \text{element-node}(v_1, \text{empty-sequence}(), \text{empty-sequence}(), \text{empty-sequence}(), \text{schema-component}(t))}$$

The following rule constructs an element from its name and children sub-expressions. The element constructor in [XQuery 1.0 and XPath 2.0 Data Model] requires that all children be nodes, which means that any simple-typed value in the element's sequence of child expressions must be converted to a text node before applying the constructor. The auxilliary function simple-to-text-node defined below performs this conversion.

$$\frac{?\ \text{|-}\ \mathit{NameSpec} \Rightarrow v_1 \qquad ?\ \text{|-}\ \mathit{Expr} \Rightarrow v_2 \qquad ?\ \text{of}\ ?\ \text{|-}\ <\mathit{NameSpec}>\ \{\ \mathit{Expr}\ \}\ </\mathit{NameSpec}> : t}{?\ \text{|-}\ <\mathit{NameSpec}>\ \{\ \mathit{Expr}\ \}\ </\mathit{NameSpec}> \Rightarrow \text{element-node}(v_1, \text{empty-sequence}(), \text{empty-sequence}(), \text{simple-to-text-node}(v_2), \text{schema-component}(t))}$$

```
define function simple-to-text-node(xs:AnyTree* $s) returns xs:AnyTree* {
  for $v in $s return
    typeswitch ($v)
    case xs:AnySimpleType return text-node(string-value($v))
    default return $v
}
```

> **Ed. Note:** PF: Do we need to concatenate consecutive text-nodes ? MF: The element constructor should do this.

The next rule constructs a sequence.

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow v_1 \qquad ? \mathrel{|\!-} Expr_2 \Rightarrow v_2}{? \mathrel{|\!-} Expr_1, Expr_2 \Rightarrow \mathrm{append}(v_1, v_2)}$$

We note here that in the [XQuery 1.0 and XPath 2.0 Data Model], sequences are always ``flat'', i.e., they do not contain other sequences.

The next three rules construct the union, difference, and intersection of two sequences. The order of items in the resulting sequence is non-deterministic.

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow v_1 \qquad ? \mathrel{|\!-} Expr_2 \Rightarrow v_2}{? \mathrel{|\!-} Expr_1 \;\texttt{union}\; Expr_2 \Rightarrow \mathrm{xfo{:}union}(v_1, v_2)}$$

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow v_1 \qquad ? \mathrel{|\!-} Expr_2 \Rightarrow v_2}{? \mathrel{|\!-} Expr_1 \;\texttt{except}\; Expr_2 \Rightarrow \mathrm{xfo{:}difference}(v_1, v_2)}$$

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow v_1 \qquad ? \mathrel{|\!-} Expr_2 \Rightarrow v_2}{? \mathrel{|\!-} Expr_1 \;\texttt{intersect}\; Expr_2 \Rightarrow \mathrm{xfo{:}intersect}(v_1, v_2)}$$

The next rule constructs the empty sequence.

$$\frac{}{? \mathrel{|\!-} () \Rightarrow \mathrm{empty\text{-}sequence}()}$$

The next rule evaluates a conditional expression. If the conditional's boolean expression $Expr_1$ evaluates to true, $Expr_2$ is evaluated and its value is produced. If the conditional's boolean expression evaluates to false, $Expr_3$ is evaluated and its value is produced.

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow \mathrm{true} \qquad ? \mathrel{|\!-} Expr_2 \Rightarrow v_2}{? \mathrel{|\!-} \mathrm{if}\; Expr_1 \;\mathrm{then}\; Expr_2 \;\mathrm{else}\; Expr_3 \Rightarrow v_2}$$

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow \mathrm{false} \qquad ? \mathrel{|\!-} Expr_3 \Rightarrow v_3}{? \mathrel{|\!-} \texttt{if}\; Expr_1 \;\texttt{then}\; Expr_2 \;\texttt{else}\; Expr_3 \Rightarrow v_3}$$

The next rule evaluates a `let` expression: it evaluates the body of the expression $Expr_2$ in the environment ? extended with the variable *Variable* bound to the value *v*.

$$\frac{? \mathrel{|\!-} Expr_1 \Rightarrow v \qquad ?, \{\, Variable \mathrel{|\!\!\to} v \,\} \mathrel{|\!-} Expr_2 \Rightarrow v_2}{? \mathrel{|\!-} \texttt{let}\; Variable := Expr_1 \;\texttt{return}\; Expr_2 \Rightarrow v_2}$$

The next rule evaluates a function application. It evaluates all the function's arguments in the current environment, then extracts the definition of the function's body from the function environment. It constructs a new environment that contains the static and function environments of ? and a new value environment that maps the function's formal arguments $Variable_1 \cdots Variable_n$ to the function's actual arguments, and then it evaluates the body of the function in this new environment.

$$\frac{\begin{array}{c}? \vdash Expr_1 \Rightarrow v_1 \quad \cdots \quad ? \vdash Expr_n \Rightarrow v_n \\ (\text{FE of } ?)(QName) = (Expr;\ Variable_1 \cdots Variable_n) \\ (? \text{ of } ?;\ \{ Variable_1 \mapsto v_1 \}, \cdots \{ Variable_n \mapsto v_n \};\ \text{FE of } ?) \vdash Expr \Rightarrow v\end{array}}{? \vdash QName(Expr_1; \cdots ;Expr_n) \Rightarrow v}$$

The next rule states that in any environment, the `error` expression evalutes to the ERROR value.

$$\frac{}{? \vdash \texttt{error} \Rightarrow \text{ERROR}}$$

5.4 Operators

The [XQuery 1.0 and XPath 2.0 Data Model] defines two equality functions, xfo:node-equal and xfo:value-equal, which correspond to XQuery's equality operators, '==' and '=', respectively. These functions operate only on unit values, i.e., a singleton sequence containing a simple value or node.

$$\frac{? \vdash Expr_1 \Rightarrow u_1 \quad ? \vdash Expr_2 \Rightarrow u_2}{? \vdash Expr_1 == Expr_2 \Rightarrow \text{xfo:node-equal}(u_1, u_2)}$$

$$\frac{? \vdash Expr_1 \Rightarrow u_1 \quad ? \vdash Expr_2 \Rightarrow u_2}{? \vdash Expr_1 = Expr_2 \Rightarrow \text{xfo:value-equal}(u_1, u_2)}$$

The boolean operators `and` and `or`, and the boolean function `not` and defined in terms of `if-then-else` expressions.

```
E1 and E2  ==> if (E1) then (if (E2) then true else false) else false
E1 or E2   ==> if (E1) then true else (if (E2) then true else false)
not (E)    ==> if (E)  then false else true
```

We have not defined the semantics of all the arithmetic operators in XQuery. A joint task force on operators with members from the [XSLT 99], XML Schema, and XML Query working groups is chartered to define arithmetic operators. XQuery will adopt the decisions of that group (See **[Issue-0056: Operators on Simple Types]**). In the following two rules, we assume that the binary and unary operators are implemented by the *apply* function whose semantics are defined by the operator task force and that this function only operates on simple values.

$$\frac{? \vdash Expr_1 \Rightarrow a_1 \quad ? \vdash Expr_2 \Rightarrow a_2}{? \vdash Expr_1\ Op_{arith}\ Expr_2 \Rightarrow \text{apply}(Op_{arith}, a_1, a_2)}$$

$$\frac{? \vdash Expr_1 \Rightarrow v_1 \quad a_2 = \text{apply}(Op, a_1)}{? \vdash Op\ Expr_1 \Rightarrow a_2}$$

5.5 Built-in functions

There is a near one-to-one correspondence between XQuery expressions for constructing and accessing data model values **[Figure 2]** and the data model constructors and accessors.

The next three rules construct comment, processing instruction, and reference nodes, respectively.

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{comment}(\mathit{Expr}) \Rightarrow \text{comment-node}(v)}$$

$$\frac{? \vdash \mathit{Expr}_1 \Rightarrow v_1 \qquad ? \vdash \mathit{Expr}_2 \Rightarrow v_2}{? \vdash \texttt{processing-instruction}(\mathit{Expr}_1, \mathit{Expr}_2) \Rightarrow \text{processing-instruction-node}(v_1, v_2)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{ref}(\mathit{Expr}) \Rightarrow \text{reference-node}(v)}$$

The following rules define all of the accessor and other data-model functions. In the following rule, the function name *f* denotes any one of the following accessors: `nodes`,

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{deref}(\mathit{Expr}) \Rightarrow \text{dereference}(v)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{local-name}(\mathit{Expr}) \Rightarrow \text{local-name}(v)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{namespace-uri}(\mathit{Expr}) \Rightarrow \text{namespace-uri}(v)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{name}(\mathit{Expr}) \Rightarrow \text{name}(v)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{parent}(\mathit{Expr}) \Rightarrow \text{parent}(v)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{string-value}(\mathit{Expr}) \Rightarrow \text{string-value}(v)}$$

$$\frac{? \vdash \mathit{Expr} \Rightarrow v}{? \vdash \texttt{typed-value}(\mathit{Expr}) \Rightarrow \text{typed-value}(v)}$$

The `sortby` expression may return any permutation of its input. The easiest way to explain the dynamic semantics of `sortby` is to assume the existence of a function `stable-sort` that takes a sequence of `pair` elements, each of which contains a key value in its `fst` subelement and the sort value in its `snd` subelement The pairs are sorted in ascending or descending order based on the value in the `fst` subelement. The purpose of is `stable-sort` is to explain the semantics of the `sortby` expression. An implementation may choose to implement the `sortby` expression as it chooses, as long as it guarantees stability on sequences. Given a `stable-sort` function, the semantics of `sortby` is defined in terms of a `for` expression, which constructs the key, value pairs, and the `stable-sort` function.

```
E1 sortby E2 ascending  ==>
  let sorted-pairs :=
    stable-sort(
      for $dot in E1 return
```

```
          <q:pair><q:fst>E2</q:fst><q:snd>$dot</q:snd></q:pair>,
        "ascending")
    return sorted-pairs/q:snd/node()

E1 sortby E2 descending  ==>
    let sorted-pairs :=
      stable-sort(
        for $dot in E1 return
          <q:pair><q:fst>E2</q:fst><q:snd>$dot</q:snd></q:pair>,
        "descending")
    return sorted-pairs/q:snd/node()
```

5.6 Iteration expressions

The next two rules evaluate iteration expressions. If the iteration expression evaluates to the empty sequence, then the entire expression evalutes to the empty sequence.

$$\frac{? \mid\!\!- Expr_1 \Rightarrow v \qquad \text{empty}(v)}{? \mid\!\!- \text{for } Variable \text{ in } Expr_1 \text{ return } Expr_2 \Rightarrow \text{empty-sequence}()}$$

The next rule first evaluates the iteration expression $Expr_1$, which produces the sequence $u_1, \ldots, u_n$. For each unit value u_i in the sequence, it evaluates the body of the iteration expression in the environment ? extended with the variable *Variable* bound to u_i, which produces the value v_i. All the v_i values are appended into the result sequence.

$$\frac{\begin{array}{c} ? \mid\!\!- Expr_1 \Rightarrow u_1, \cdots, u_n \\ ?, \{ Variable \mid\!\!\to u_1 \} \mid\!\!- Expr_2 \Rightarrow v_1 \\ \cdots \\ ?, \{ Variable \mid\!\!\to u_n \} \mid\!\!- Expr_2 \Rightarrow v_n \end{array}}{? \mid\!\!- \text{for } Variable \text{ in } Expr_1 \text{ return } Expr_2 \Rightarrow \text{append}(v_1, \cdots, v_n)}$$

5.7 Typeswitch expressions

When evaluating a typeswitch expression, the variable *Variable* and the value *v* to match against occurs on the left of the turnstile |-. Each case rule of a typeswitch expression is always evaluated *against* this value. Alternative case rules are tried from left to right. The rule for the typeswitch expression evaluates its expression and sets up the appropriate environment for the case rules:

$$\frac{? \mid\!\!- Expr \Rightarrow v \qquad ?; (Variable, v) \mid\!\!- CaseRules \Rightarrow v_1}{? \mid\!\!- \texttt{typeswitch}\ (Expr)\ \texttt{as}\ Variable\ CaseRules \Rightarrow v_1}$$

If the value *v* is in the domain of the type expression *Type*, the next rule extends the environment by binding the variable *Variable* to *v* and evaluates the body of the case rule. The *domain* of a type is the possibly infinite set containing all values that are instances of that type. We can check whether a value *v* is in the domain of a type by checking whether *v*'s run-time type is a subsumed by *Type*. For an element or attribute, the *declaration* accessor in [XQuery 1.0 and XPath 2.0 Data Model] returns its run-time type. For a simple value, the *type* accessor returns its run-time type.

$$\frac{v \text{ in Dom}(Type) \qquad ?, \{ Variable \mid\!\!\to v \} \mid\!\!- Expr \Rightarrow v_2}{?; (Variable, v) \} \mid\!\!- \texttt{case}\ Type\ \texttt{return}\ Expr\ CaseRules \Rightarrow v_2}$$

If the value *v* is not in the domain of the type expression *Type*, the next rule evaluates the case rules following the current one. The body of the given case rule is not evaluated if *v* is not in the domain of the given type.

not (*v* in Dom(*Type*)) ? ; (*Variable*, *v*) |- *CaseRules* => v_2

? ; (*Variable*, *v*) |- `case` *Type* `return` *Expr* *CaseRules* => v_2

The next rule states that the else branch of a typeswitch expression always evaluates to its given expression.

? |- *Expr* => v_1

? ; (*Variable*, *v*) |- `default return` *Expr* => v_1

5.8 Top-level declarations and query expressions

The value-inference rules for top-level declarations and query expressions return a value. All top-level declarations return the empty-sequence value.

Context declarations return the empty-sequence value.

? |- `namespace` *NCName* = *StringLiteral* => ()

? |- `default namespace` = *StringLiteral* => ()

? |- `type` *NCName* = *t* => ()

? |- `define function` *QName* (*$Type_1$* *$Variable_1$*, · · ·, *$Type_n$* *$Variable_n$*) `returns` *Type* { *Expr* } => ()

We use the function *dynamic-env* to compute the initial environment for evaluating a query expression.
dynamic-env (`type` *x* = *t*) = (*static-env*(`type` *x* = *t*); (); ())
dynamic-env (`define function` *QName* (*$Type_1$* *$Variable_1$*, ···, *$Type_n$* *$Variable_n$*) `returns` *Type* { *Expr* } =
(*static-env*(`define function` *QName* (*$Type_1$* *$Variable_1$*, ···, *$Type_n$* *$Variable_n$*) `returns` *Type* { *Expr* }); (); (
dynamic-env (`namespace` *NCName* = *StringLiteral*) = (*static-env*(`namespace` *NCName* = *StringLiteral*); ();
dynamic-env (`default namespace` = *StringLiteral*) = (*static-env*(`default namespace` = *StringLiteral*); ();

A function declaration extends the top-level environment ? by adding the mapping from the function's *QName* to the expression that is the function's body, and the function's formal arguments, i.e., a sequence of variables that are free in the function's body.

A top-level expression sequence has no effect on the environment and returns a value that may be returned to the programming environment in which the XQuery expression is evaluated.

? |- $Expr_1$ => v_1
...
? |- $Expr_n$ => v_n

? |- $Expr_1$ ··· $Expr_n$ => append(v_1, ···, v_2)

The expression sequence in a *QueryModule* is evaluated in the dynamic environment computed by *dynamic-env*.

$$\frac{\begin{array}{c} ? \;\vdash dynamic\text{-}env\ (ContextDecl_1),\ldots,\ dynamic\text{-}env\ (ContextDecl_k),\\ dynamic\text{-}env\ (TypeDecl_1),\ldots,\ dynamic\text{-}env\ (TypeDecl_l),\\ dynamic\text{-}env\ (FunctionDecl_1),\ldots,\ dynamic\text{-}env\ (FunctionDecl_m)\\ ? \;\vdash Expr_1 \Rightarrow v_1 \quad \cdots, \quad ? \;\vdash Expr_n \Rightarrow v_n \end{array}}{\begin{array}{c} \vdash ContextDecl_1,\ \cdots,\ ContextDecl_k,\\ TypeDecl_1,\ \cdots,\ TypeDecl_l,\\ FunctionDecl_1,\ \cdots,\ FunctionDecl_m,\\ Expr_1,\ \cdots,\ Expr_n \Rightarrow \mathrm{append}(v_1,\ \cdots,\ v_n) \end{array}}$$

6 XQuery Mapping to Core

As explained in the introduction, the semantics of the full XQuery language is obtained by mapping full XQuery expression (In the full XQuery grammar, see [XQuery 1.0: A Query Language for XML]) into the core syntax (In the Core Xquery grammar, see **[3 XQuery Core Syntax]**). In **[4 Static Semantics : Type-Inference Rules]** and **[5 Dynamic Semantics : Value-Inference Rules]** we have given the complete static and dynamic semantics for the core. We now explain how the full XQuery syntax is mapped into this core.

Please note: this mapping is still preliminary and contains inconsistencies. See **[Issue-0099: Incomplete/inconsistent mapping from XQuery to core]**

In XQuery, a query is composed of a preamble (containing schema, namespace and function declarations) and a body (containing a single XQuery expression). First, we give a mapping for XQuery expressions into Core XQuery expressions, then we give a mapping for XQuery declarations into Core XQuery declarations. This mapping is based on the XQuery grammar given in appendix B.

6.1 Notations

We will use the following notations.

```
E                     Expression
l                     Local Name (NCName)
n:l                   QName with namespace n and local name l
a                     Qualified name

$v                    variables
$dot                  Distinguished variable used to contain the current
                    node.
$roots                Distinguished variable that contains a sequence of
                    document root nodes.

T                     Type

[[ E ]]_c ==> E'    Given the context c, the XQuery expression E is mapped
                      to the Core Xquery expression E'.
```

When the context is clear, we will write: [[E]] ==> E'.

6.2 Mapping for XQuery expressions

We give a mapping for each class of XQuery expression. Each of these classes corresponds to some specific productions in the full XQuery Grammar.

6.2.1 Path expressions

XQuery uses a fragment of XPath, plus two additional features: dereference and range predicates. XQuery and the Core XQuery do not currently support all of the axis of XPath.

The mapping assumes that path steps are always given an explicit input expression. I.e., XPath abreviations are resolved, for instance, name is already in the form ./name, /person[name = "John"] is already in the form /person[./name = "John"], etc.

6.2.1.1 Current Node

In XQuery, '.' denotes the current node. The current node is a special node, whose value depends on the context of the expression within which it is evaluated. Inside the XQuery core, we will use a distinguished variable, called dot, to contain the value of the current node. Variable dot will always be bound explicitly in the core (see for instance, navigation and SORTBY expressions).

```
[[ . ]]   ==>   $dot
```

6.2.1.2 Root node

In XQuery, '/' denotes a fixed set of input document roots, the mapping is:

```
[[ / ]]   ==>   $roots
```

Note that the value for variable $roots is defined in the environment, before compilation and evaluation of the query. Therefore, it is fixed for the duration of query compilation and evaluation. There is no operation in the language that can modify that value.

6.2.1.3 Simple Navigation

The XQuery Core does not contain path expressions. Instead, it uses a combination of access to children, iteration and typeswitch in order to capture navigation in path expressions. The following mapping applies when 'a' is a *NameSet*.

```
[[ E/a ]]    ==>   for $v1 in [[ E ]] return
                     for $v2 in children($v1) return
                       typeswitch ($v2) as $v3
                         case ELEMENT a (AnyComplexType) return $v3
                         default return ()

[[ E/@a ]]   ==>   for $v1 in [[ E ]] return
                     for $v2 in attributes($v1) return
                       typeswitch ($v2) as $v3
                         case ATTRIBUTE a (AnyComplexType) return $v3
                         default return ()

[[ /a ]]     ==>   [[ $root/a ]]
```

Projecting the simple-typed value of an element or attribute is equivalent to applying the typed-value accessor in [XQuery 1.0 and XPath 2.0 Data Model].

```
[[ E/DATA() ]]  ==> for $v1 in [[ E ]] return
                      typed-value([[ E ]])
```

6.2.1.4 Recursive Navigation

Recursive navigation uses the built-in function descendent-or-self that returns the current node as well as all its descendents.

```
[[ E//a ]]   ==>   [[ descendent-or-self([[E]])/a ]]

[[ //a ]]    ==>   [[ descendent-or-self($root)/a ]]
```

6.2.1.5 Access to COMMENT, NODES, PIs, and TEXT

Again, access to specific nodes in path expressions is mapped to iteration and type matching.

```
[[ E/COMMENT() ]]                ==>    for $v1 in [[ E ]] return
                                          for $v2 in children($v1) return
                                            typeswitch ($v2) as $v3
                                              case Comment return $v3
                                              default return ()

[[ E/PROCESSING-INSTRUCTION() ]] ==> for $v1 in [[ E ]] return
                                       for $v2 in children($v1) return
                                         typeswitch ($v2) as $v3
                                           case PI return $v3
                                           default return ()

[[ E/NODE() ]]                   ==> for $v1 in [[ E ]] return
                                       children($v1)

[[ E/TEXT() ]]                   ==> string([[ E ]])
```

6.2.1.6 Dereference

This uses the support for dereference in the XQuery core. This assumes an agreement on what is a reference and what is not (e.g., that ID/IDREF are converted to ref() in the data model).

```
[[E => a]]   ==>   for $v1 in [[ E ]] return
                     for $v2 in deref($v1) return
                       typeswitch ($v2) as $v3
                         case ELEMENT a (AnyComplexType) return $v3
                         default return ()

[[E => @a]]  ==>   for $v1 in [[ E ]] return
                     for $v2 in deref($v1) return
                       typeswitch ($v2) as $v3
                         case ATTRIBUTE a (AnyComplexType) return $v3
                         default return ()
```

6.2.1.7 Predicates

Local XPath predicates iterate over the nodes in a sequence, and selects only the nodes that verify the predicate.

There are three kinds of predicates. Predicates which take a boolean expression as a parameters return only the nodes which for which that expression returns true. Predicates which take an integer as a parameter return the nodes whose index in the sequence is equal to that integer. Finally, range predicates select nodes whose index is between the bounds of the range expression.

```
[[ E1[i] ]]           ==>   [[ E1[i to i] ]]

[[ E1[i1 to i2] ]]    ==>   for $v in index([[ E1 ]]) return
                              if ([[ $v/fst/data() >= i1 ]] and [[ $v/fst/data() <
                              then [[ $v/snd/deref() ]] else ()

[[ E1[E2] ]]          ==>   for $v in [[ E1 ]] return
                              let $dot := $v return
                              if [[ E2 ]] then $v else ()
```

6.2.1.8 Parent

Parent navigation is mapped to the corresponding parent built-in function in the core.

```
[[ E/.. ]] ==> parent([[ E ]])
```

6.2.2 Element and attribute constructors

Element constructors in XQuery can support full XML syntax. As a result the mapping give here is more involved and requires collaboration with the XML parser. The content of elements is always mapped to enclosed expressions with '{' '}' in the core. The mapping then takes content of element constructors and generate appropriate node creation statements using the data model constructors.

In order to map element constructors, we will use the following additional mapping notations.

```
Ci's               are allowed characters or entity references
Ni'                  are elementary components within the element content
[[ ]]_elemcontent  is a specific mapping rule for the content of elements
[[ ]]_elem           is a specific mapping rule for elementary
                   components within the element content
```

The first rules assume that the parser can indicate 'breaking points' in the content of element constructors in order to generate corresponding text nodes, subelements, etc

```
//text node creation
[[ C1C2C3... ]]_elem      ==>  text("C1C2C3...")

//{} are always produced from the embedding rule for the element
[[ { E } ]]_elem    ==>  [[ E ]]

// cdata node creation
[[ "&lt;![CDATA[" C1C2C3... "]]&gt;" ]]_elem  ==>  cdata("C1C2C3...")

// nested element are kept unchanged
[[ "&lt;" X ">" ]]     ==>  "&lt;" X ">"

[[ N1N2N3... ]]_elemcontent  ==>  [[N1]]_elem[[N2]]_elem[[N3]]_elem...
```

Then the next rules convert the attribute specifications separately by embedding them within the content of the element constructors.

```
[[ <a a1=E1 ... an=En> E </a> ]] ==>
   <a> {[ ATTRIBUTE a1 [[E1]], ..., ATTRIBUTE an [[En]], [[E]]_elemcontent ]} </>

[[ <a a1=E1 ... an=En> E </> ]] ==>
   <a> {[ ATTRIBUTE a1 [[E1]], ..., ATTRIBUTE an [[En]], [[E]] ]} </>

[[ <{Et} a1=E1 ... an=En> E </> ]] ==>
   <{ [[ Et ]] }> {[ ATTRIBUTE a1 [[E1]], ..., ATTRIBUTE an [[En]], [[E]] ]} </>

[[ <a a1=E1 ... an=En /> ]] ==>
   <a> {[ ATTRIBUTE a1 [[E1]], ..., ATTRIBUTE an [[En]], [[E]] ]} </>
```

Note that if the Ei's do not return an attribute value, this will be detected by the Core XQuery type system at compile time. Note also, that the expressions E or might return attributes, in which case the element constructor will append them at the beginning along with the explicitely declared attributes (see corresponding dynamic semantics).

6.2.3 FLWR expressions

6.2.3.1 FLWR normalization

In the full XQuery grammar, Flower expressions are defined using the following production:

```
FlwrExpr     ::= (ForClause | LetClause)+ WhereClause? ReturnClause
```

This allows complex forms of FLWR expressions in the full XQuery syntax, where LET and FOR clauses may have several variables. On the other hand, the XQuery core only allows elementary FOR and LET clauses independant from each other and terminated by a RETURN.

The mapping from full XQuery to the Core XQuery performs the corresponding transformation by separating multiple FOR and LET statements and adding RETURN clauses whenever necessary.

Let C be a FlwrClause. The mapping from XQuery to the core is defined using an auxiliary mapping function noted [[]]_clause which processes FLWR clauses one at a time.

The first two rules are used to map each clause one at a time.

```
[[ C return E ]]
==>
[[ C ]]_clause([[ E ]])

[[ C1 C2  ... Cn return E ]]
==>
[[ C1 ]]_clause([[ C2  ... Cn return E ]])
```

Then, the following rules are used for each individual clause in the full XQuery grammar, adding appropriate return keywords.

```
[[ for $v1 in E1
       ...,
       $vn in En ]]_clause(E)
==>
for $v1 in [[ E1 ]] return
...
for $vn in [[ En ]] return E
```

```
[[ let $v1 := E1
       ...,
       $vn := En ]]_clause(E)
==>
let $v1 := [[ E1 ]] return
...
let $vn := [[ En ]] return E

[[ where E1 ]]_clause(E)
==>
if [[ E1 ]] then E else ()
```

6.2.3.2 Sort by

The mapping for sortby transform the n-ary XQuery sort into a series of binary sorts in the Core. Like for predicates, this mapping supposes that path expressions in the sort criterias are all given an explicit input. This also assumes a stable sort in the Algebra.

```
[[ E sortby E1 ASCENDING, ..., En ASCENDING ]]

==>

(...((([[ E ]]) sortby [[ En ]] ASCENDING)
                sortby [[ En-1 ]] ASCENDING)
                ...)
                sortby [[ E1 ]] ASCENDING
```

6.2.4 Operators

Operators in XQuery are mapped to corresponding operators in the XQuery core.

6.2.4.1 Boolean operators

Mapping predicates from XQuery to the Core is not completely straightforward, as XQuery uses implicit existential quantification (like XPath), while the Core XQuery does not. As a consequence, existential quantification needs to be explicitly introduced by the mapping.

```
[[ E1 = E2 ]]   ==> [[ not(empty(for $v1 in [[ E1 ]] return
                                 for $v2 in [[ E2 ]] return
                                 if eq($v1,$v2) then $v1 else ())) ]]

[[ E1 == E2 ]]  ==> [[ not(empty(for $v1 in [[ E1 ]] return
                                 for $v2 in [[ E2 ]] return
                                 if nodeeq($v1 = $v2) then $v1 else ())) ]]

[[ E1 < E2 ]]   ==> [[ not(empty(for $v1 in [[E1]] return
                                 for $v2 in [[E2]] return
                                 if lt($v1, $v2) then $v1 else ())) ]] ]]
[[ E1 <= E2 ]]  ==> [[ not(empty(for $v1 in [[E1]] return
                                 for $v2 in [[E2]] return
                                 if lteq($v1, $v2) then $v1 else ())) ]]
[[ E1 >= E2 ]]  ==> [[ not(empty(for $v1 in [[E1]] return
                                 for $v2 in [[E2]] return
                                 if gteq($v1, $v2) then $v1 else ())) ]]
[[ E1 > E2 ]]   ==> [[ not(empty(for $v1 in [[E1]] return
                                 for $v2 in [[E2]] return
                                 if gt($v1, $v2) then $v1 else ())) ]]
```

```
[[ E1 != E2 ]]  ==> [[ not(empty(for $v1 in [[E1]] return
                                  for $v2 in [[E2]] return
                                  if neq($v1, $v2) then $v1 else ())) ]]
```

Note that these mapping use elementary built-in comparison operators in the Core (e.g., eq for equality). Note also that as opposed to XPath 1.0, the current semantics does not perform any implicit value conversions

6.2.4.2 Collection operators

The empty function can be mapped to value equality:

```
[[ empty(E) ]]        ==> [[ E ]] = ()
```

All other collection operations in XQuery are based on node identity.

Union in XQuery corresponds to the Union over unordered collections and removes duplicates.

```
[[ E1 UNION E2 ]]     ==> distinct-node([[E1]], [[E2]])
```

Intersection and exception can be written in the algebra using existential quantification.

```
[[ E1 INTERSECT E2 ]]
==>
for $v1 in [[ E1 ]] return
  if not((for $v2 in [[ E2 ]] return
            if node_equal($v2,$v1) then $v2 else ()) = ())
  then $v1
  else ()

[[ E1 EXCEPT E2 ]]
==>
for $v1 in [[ E1 ]] return
  if ((for $v2 in [[ E2 ]] return
         if node_equal($v2,$v1) then $v2 else ()) = ())
  then $v1
  else ()
```

BEFORE and AFTER operators returns the subset of the element in the first expression which are before or after one of the element in the second expression in document order. The mapping to the algebra uses both existential quantification and built-in document order functions 'before()' and 'after()'.

```
[[ E1 BEFORE E2 ]]
==>
for $v1 in [[ E1 ]] return
  if not((for $v2 in [[ E2 ]] return
            if before($v2,$v1) then $v2 else ()) = ())
  then $v1
  else ()

[[ E1 AFTER E2 ]]
==>
for $v1 in [[ E1 ]] return
  if not((for $v2 in [[ E2 ]] return
            if after($v1,$v2) then $v2 else ()) = ())
  then $v1
  else ()
```

6.2.5 Function application

Function application is mapped to equivalent function calls. According to the current XQuery specification, this can imply implicit iteration over the parameters of the function.

```
if FunctionName(T1 $v1,..., Tn $vn) and Ti <: AnyTree for each i,
then:

[[ FunctionName(E1,..., En) ]]
     ==>
for $x1 in [[ E1 ]] return
...
for $xn in [[ En ]] return
FunctionName($x1, ..., $xn)

Otherwise:

[[ FunctionName(E1,..., En) ]]
     ==> FunctionName( [[ E1 ]], ..., [[ E2 ]] )
```

The FILTER function will be treated in the next section

6.2.6 Quantification

Quantifiers are simply mapped to existential predicate and iteration in the Algebra.

```
[[ some $v in E1 satisfies E2 ]]
==>
not((for $v in [[ E1 ]] return
       if [[ E2 ]] then $v else ()) = ())

[[ every $v in E1 satisfies E2 ]]
==>
(for $v in [[ E1 ]] return
   if not([[ E2 ]]) then $v else ()) = ()
```

6.2.7 Type related operations

6.2.7.1 Typeswitch

Here are mappings for typeswitch short-cuts.

One might want to omit the else clause.

```
[[ typeswitch (E0) as $v
     case $v1 : T1 return E1
     ...
     case $vn : Tn return En ]]
==>

typeswitch ([[ E0 ]]) as $v
  case T1 return [[ E1 ]]
  ...
  case Tn return [[ En ]]
  default return ERROR
```

6.2.7.2 TREAT

The TREAT operation changes the type of an expression and might raise an error at run-time.

```
[[ (TREAT AS Type) E ]]
==>
typeswitch ([[ E ]]) as $v
  case Type return $v
  default return ERROR
```

6.2.7.3 CAST

The CAST operation is left intact:

```
[[ CAST AS Type (E) ]]
==>
CAST AS Type ([[ E ]])
```

6.2.7.4 INSTANCEOF

The INSTANCEOF operation checks whether an expression is of a given type.

```
[[ E INSTANCEOF Type ]]
==>
typeswitch ([[ E ]]) as $v
  case Type return true
  default return false
```

The INSTANCEOF operations allows an optional 'ONLY' parameter. This is not clear what is its semantics and how it can be supported.

6.3 Mapping of XQuery declarations to Algebra declarations

6.3.1 Mapping of type declarations

6.3.2 Mapping of function declarations

Each user defined function in XQuery is mapped to a corresponding function in the XQuery Core by mapping the expression which defines that function.

```
[[ DEFINE FUNCTION f(T1 v1, ..., Tn vn) RETURNS T' { E } ]]
==>
DEFINE FUNCTION f(T1 v1, ..., Tn vn) RETURNS T' { [[ E ]] }
```

6.3.3 Predefined functions

In the above mapping, we used a number of functions, which are not part of the Core XQuery, but can be expressed using the core. These functions need to be declared accordingly as part of the preamble. Note that this means some of the preamble is extended on behalf of the user each time he runs a query. We could think of adding these functions to XQuery itself as part of the native XQuery function library (like filter).

6.3.3.1 root()

The following function computes the root of the input node.

```
function root(AnyElement $x) returns AnyElement
 {
   let $p := parent($x) return
   if $p = () then $x else root($p)
 }
```

6.3.3.2 filter()

The following algebra functions implement the corresponding XQuery filter function. Note that this definition lose all type information, see [XQuery Issue MAP-005: Typing for Filter].

```
function member(AnyNode $x, AnyType $y) returns Boolean =
  {
    not((for $v in $y return
           if $v == $y then $v else ()) = ())
  }

function filter1(AnyElement $x, AnyForest $y) returns AnyForest =
  {
    if (member($x,$y))
    then
      typeswitch ($x) as $x'
        case (Comment|PI|String|AnyAttribute) return $x
        default return
         let $tag := name($x) return
             <{$tag$}>
                 { $x/@*,
                     filter( [[ $x/node() ]], $y ) }
             </>
    else
      filter( [[ $x/node() ]], $y )
   }

function filter(AnyType $x, AnyType $y) returns AnyType
  {
    for $x' in $x return filter1($x',$y)
  }
```

6.3.3.3 id()

The following function computes the node corresponding to a given IDREF. I assume that one can compare ID and IDREF values using equality.

```
function contains_id(AnyElement $x, IDREF $i) returns Boolean
  {
     let $attr_val := (for $v1 in [[ $x/@* ]] return value($v1))
     return not((for $v2 in $attr_val return
                   if eq($v2, $i) then $i else ()) = ())
  }

function find_id(AnyElement $x, IDREF $i) returns AnyElement
  {
     if contains_id($x, $i)
     then $x
     else for $v1 in children($x) return find_id($v1,$i)
```

```
  }

function id(IDREF $i) returns AnyElement
  {
     let $r := root($i) return
       [[ (TREAT AS AnyElement) find_id($r,$i) ]]
//  the treat should never fail as the schema validator enforces existence
//  and uniqueness of the ID for a given IDREF
  }
```

6.3.3.4 Aggregation functions

The following functions compute aggregation by using head, tail and recursion. Note that these functions rely on arithmetic and will be revisited by the operators task force.

```
function sum((Integer | Float)* $x)
  {
     if ($x = ())
     then 0
     else head($x) + sum(tail($x))
  }

function count(AnyType $x) returns Integer
  {
     if ($x = ())
     then 0
     else 1 + count(tail($x))
  }

function avg((Integer | Float)* $x) returns (Integer | Float)
  {
     sum($x) div count($x)
  }

function get_max((Integer | Float)* $x, (Integer | Float) $y) returns (Integer | Fl
  {
     if $x = ()
     then $y
     else
       let $first := tail($x) return
       if ($first > $y)
       then get_max(tail($x),$first)
       else get_max(tail($x),$y)
  }

function max((Integer | Float)* $x) returns (Integer | Float)
  {
     get_max($x, -Infinite)
  }

function get_min((Integer | Float)* $x, (Integer | Float) $y) returns (Integer | Fl
  {
     if $x = ()
     then $y
     else
       let $first := tail($x) return
       if ($first < $y)
       then get_min(tail($x),$first)
       else get_min(tail($x),$y)
```

```
    }

  function min((Integer | Float)* $x) returns (Integer | Float)
    {
       get_min($x, +Infinite)
    }
```

7 References

BFS00
: P. Buneman, M. Fernandez, D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, April, 2000, Vol 9, Number 1.

BK93
: Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. *Theoretical Computer Science* 116(1&2):59--94, August 1993.

BKD90
: Francois Bancilhon, Paris Kanellakis, Claude Delobel. *Building an Object-Oriented Database System*. Morgan Kaufmann, 1990.

BNTW95
: Peter Buneman, Shamim Naqvi, Val Tannen, Limsoon Wong. Principles of programming with complex object and collection types. *Theoretical Computer Science* 149(1):3--48, 1995.

Quilt
: Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.

CM93
: S. Cluet and G. Moerkotte. Nested queries in object bases. *Workshop on Database Programming Languages*, pages 226--242, New York, August 1993.

YAT99
: S. Cluet, S. Jacqmin and J. Siméon *The New YaTL: Design and Specifications. Technical Report*, INRIA, 1999.

Col90
: L. S. Colby. A recursive algebra for nested relations. *Information Systems* 15(5):567-582, 1990.

Date97
: Hugh Darwen (Contributor) and Chris J. Date. *Guide to the SQL Standard : A User's Guide to the Standard Database Language SQL* Addison-Wesley, 1997.

XMLQL99
: A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. *A query language for XML.* In *International World Wide Web Conference*, 1999.

Graefe93
: Goetz Graefe, *Query Evaluation Techniques for Large Databases*. In *ACM Computing Surveys*, 25(2):73--170, 1993.

LW97
: Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *Journal of Computer and Systems Sciences*, 55(2):241--272, October 1997.

HP2000
: Haruio Hosoya, Benjamin Pierce, XDuce : A Typed XML Processing Language (Preliminary Report) *WebDB Workshop* 2000.

LMW96
: Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multi-dimensional arrays: Design, implementation, and optimization techniques. *SIGMOD* 1996.

Milner
: R. Milner, M. Tofte, R. Harper, D. MacQueen *The Definition of Standard ML (Revise)*. MIT Press, 1997.

Mitchell
: John C. Mitchell *Foundations for Programming Languages*. MIT Press, 1998.

Mog89
: E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science* Asilomar, California, IEEE, June 1989.

Mog91
: E. Moggi, Notions of computation and monads. *Information and Computation*, 93(1), 1991.

OCaml
: The Caml Language. See http://pauillac.inria.fr/caml/.

XQL99
: J. Robie, editor. XQL '99 Proposal, 1999. See http://www.ibiblio.org/xql/xql-proposal.html.

Wad92
: Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461-493, 1992.

Wad93
: P. Wadler, Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, Springer Verlag, 1993. Also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

Wad95
: P. Wadler, How to declare an imperative. *ACM Computing Surveys*, 29(3):240--263, September 1997.

XML Names
: World Wide Web Consortium. Namespaces in XML. W3C Recommendation. See http://www.w3.org/TR/REC-xml-names/ .

XML
: World-Wide Web Consortium Extensible Markup Language (XML) 1.0 (Second edition), October, 2000 See http://www.w3.org/TR/REC-xml.

XML Query Algebra
: World-Wide Web Consortium The XML Query Algebra, Working Draft, Feb 2001. See http://www.w3.org/TR/query-algebra/.

XQuery 1.0: A Query Language for XML
: World Wide Web Consortium, *XQuery 1.0: A Query Language for XML*, Working Draft, June 2001. Available at: http://www.w3.org/TR/xquery/

XQuery 1.0 and XPath 2.0 Data Model
: World-Wide Web Consortium XQuery 1.0 and XPath 2.0 Data Model, Working Draft, June 2001. See http://www.w3.org/TR/query-datamodel/.

XPath
: World-Wide Web Consortium XML Path Language (XPath) : Version 1.0. November, 1999. See http://www.w3.org/TR/xpath.html.

XML Schema Part 1
: World-Wide Web Consortium XML Schema Part 1 : Structures, Working Draft. April 2000. See http://www.w3.org/TR/xmlschema-1/

XSLT 99
: World-Wide Web Consortium XSL Transformations (XSLT), Version 1.0. W3C Recommendation, November 1999. See http://www.w3.org/TR/xslt.

XML Schema Part 2
: World-Wide Web Consortium XML Schema Part 2 : Datatypes, Working Draft, April 2000. See http://www.w3.org/TR/xmlschema-2/.

XML Schema : Formal Description
: World-Wide Web Consortium XML Schema: Formal Description, Working Draft, March 2001. See http://www.w3.org/TR/xmlschema-formal/

A Equivalences

A.1 Relating projection to iteration

The examples use the `/` operator liberally, but in fact we use `/` as a convenient abbreviation for expressions built from lower-level operators: `for` expressions, the `nodes` function, and `typeswitch` expressions.

For example, the expression:

```
$book0/author
```

is equivalent to the expression:

```
for $v in children($book0) return
  typeswitch ($v)
    case author[AnyComplexType] return $v
    default return ()
```

Here the `children()` function returns a sequence consisting of the content of the element `book0`, namely, a title element and three author elements (the order is preserved). The `for` expression binds the variable `v` successively to each of these elements. Then the `typeswitch` expression selects a branch based on the type of `v`. If it is an `author` element then the first branch is evaluated, otherwise the second branch. If the first branch is evaluated, it returns `a`. The variable `a` contains the same value as `v`, but the type of `a` is `author[xs:string]`, which is the intersection of the type of `v` and the type `author[AnyComplexType]`. If the second branch is evaluated, then the branch returns `()`, the empty sequence.

To compose several expressions using `/`, we again use `for` expressions. For example, the expression:

```
$bib0/book/author
```

is equivalent to the expression:

```
for $b in children($bib0) return
  typeswitch ($b)
    case book[AnyComplexType] return
      (for $d in children($b) return
         typeswitch ($d)
           case author[AnyComplexType] return $d
           default return ())
    default return ()
```

The `for` expression iterates over all `book` elements in `bib0` and binds the variable `b` to each such element. For each element bound to `b`, the inner expression returns all the `author` elements in `b`, and the resulting sequences are concatenated together in order.

In general, an expression of the form *e* / *a* is converted to the form

```
for $v1 in e return
  for $v2 in children($v1) return
    typeswitch ($v2)
      case a[AnyComplexType] return $v2
      default return ()
```

where *e* is an expression, *a* is an element name, and v_1 and v_2 are fresh variables (ones that do not appear in the expression being converted).

According to this rule, the expression `bib0/book` translates to

```
for $v1 in $bib0 return
  for $v2 in children($v1) return
    typeswitch ($v2)
      case book[AnyComplexType] return $v2
      default return ()
```

In [A Equivalences], we discuss laws which allow us to simplify this to the previous expression:

```
for $v2 in children($bib0) return
  typeswitch ($v2)
    case book[AnyComplexType] return $v2
    default return ()
```

Similarly, the expression `bib0/book/author` translates to

```
for $v4 in (for $v2 in children($bib0) return
              typeswitch ($v2)
                case book[AnyComplexType] return $v2
                default return ()) return
  for $v5 in children($v4) return
    typeswitch ($v5)
      case author[AnyComplexType] return $v5
      default return ()
```

Again, the laws will allow us to simplify this to the previous expression

```
for $v2 in children($bib0) return
  typeswitch ($v2)
    case book[AnyComplexType] return
      (for $v5 in children($v2) return
         typeswitch ($v5)
           case author[AnyComplexType] return $v5
           default return ())
    default return ()
```

These examples illustrate an important feature of the Algebra: high-level operators may be defined in terms of low-level operators, and the low-level operators may be subject to algebraic laws that can be used to further simplify the expression.

A.2 Laws

In this section, we describe some laws that hold for the Algebra. These laws are important for defining rules that simplify algebraic expressions, such as eliminating unnecessary `for` or `typeswitch` expressions.

The iteration construct of the Algebra is closely related to an important mathematical object called a *monad*. A monad, among other things, generalizes set, bag, and list types. In functional languages, the *comprehension* construct is used to express iteration over set, bag, and list types [BNTW95], [LW97]. A comprehension corresponds directly to a monad [Wad92], [Wad93], [Wad95].

The correspondence between the Algebra's iteration construct and a monad is close, but not exact. Each monad is based on a unary type constructor, such as *Set{t}* or *List{t}*, representing a homogenous set or list where all elements are of type *t*. In the Algebra, we have more complex and heterogenous types, such as a sequence consisting of a title, a year, and a sequence of one or more authors. Also, one important component of a monad is the unit operator, which converts an element to a set or list. If *x* has type *t*, then *set{x}* is a unit set of type *Set{t}* or *[x]* is a unit list of type *List{t}*. In the Algebra, we simply write, say, `author["Buneman"]`, which stands for both a tree and for the unit sequence containing that tree.

Monads satisfy three laws, and three corresponding laws are satisfied by the the Algebra's *for* expression.

First, iteration over a unit sequence can be replaced by substition. This is called the *left unit* law.

```
for v in e1 return e2 = e2{v := e1}
```

provided that e_1 is a unit type (e.g., is an element or a scalar constant). We write $e_2\{v := e_1\}$ to denote the result of taking expression e_2 and replacing occurrences of the variable *v* by the expression e_1. For example,

```
for v in author["Buneman"] return auth[v/data()] = auth["Buneman"]
```

The iteration over a sequence of one item can always be eliminated using variable substitution.

Second, an iteration that returns the iteration variable is equivalent to the identity. This is called the *right unit* law.

```
for v in e return v = e
```

For example

```
for $v in $book0 return $v == $book0
```

An important feature of the type system described here is that the left side of the above equation always has the same type as the right side.

Third, there are two ways of writing an iteration over an iteration, both of which are equivalent. This is called the *associative* law.

```
for $v2 in (for $v1 in e1 return e2) return e3
= for $v1 in e1 return (for $v2 in e2 return e3)
```

For example, a projection over a sequence includes an implicit iteration, so *e* / *a* = `for` *v* `in` *e* `return` *v* / *a*. Say we define a sequence of bibliographies, `bib1 = bib0, bib0`. Then `bib1/book/author` is equivalent to the first expression below, which in turn is equivalent to the second.

```
for $b in (for $a in bib1 return $a/book) return $b/author
= for $a in $bib1 return (for $b in $a/book return $b/author)
```

With nested relational algebra, the monad laws play a key role in optimizing queries. Similarly, the monad laws can also be exploited for optimization in this context.

For example, if `$b` is a book, the following finds all authors of the book that are not Buneman:

```
for $a in $b return
where $a/data() != Buneman return
```

```
    $a
```

If `$l` is a list of authors, the following renames all `author` elements to `auth` elements:

```
for $a' in $l return
  auth[ $a'/data() ]
```

Combining these, we select all authors that are not Buneman, and rename the elements:

```
for $a' in (for $a in $b
            where $a/data() != Buneman return
               $a) return
  auth[ $a'/data() ]
```

Applying the associative law for a monad, we get:

```
for $a in $b,
    $a' in (where $a/data() != Buneman return $a) return
  auth[ $a'/data() ]
```

Expanding the `where` clause to a conditional, we get:

```
for $a in $b
    $a' in (if $a/data() != Buneman then $a else ()) return
  auth[ $a'/data() ]
```

Applying a standard law for distributing loops over conditionals gives:

```
for $a in $b return
  if $a/data() != Buneman then
    for $a' in $a return
      auth[ $a'/data() ]
  else ()
```

Applying the left unit law for a monad, we get:

```
for $a in $b return
  if $a/data() != Buneman then
    auth[ $a/data() ]
  else ()
```

And replacing the conditional by a `where` clause, we get:

```
for $a in $b return
  where $a/data() != Buneman do
    auth[ $a/data() ]
```

Thus, simple manipulations, including the monad laws, fuse the two loops.

[**A.1 Relating projection to iteration**] ended with two examples of simplification. Returning to these, we can now see that the simplifications are achieved by application of the left unit and associative monad laws.

[**Figure 12**] contains a dozen algebraic simplification laws. In a relational query engine, algebraic simplifications are often applied by a query optimizer before a physical execution plan is generated;

algebraic simplification can often reduce the size of the intermediate results computed by a query evaluator. The purpose of our laws is similar -- they eliminate unnecessary `for` or `typeswitch` expressions, or they enable other optimizations by reordering or distributing computations. The set of laws given is suggestive, rather than complete.

$$\begin{array}{rl} E ::= & \texttt{if } [] \texttt{ then } e_1 \texttt{ else } e_2 \\ | & \texttt{let } v = [] \texttt{ in } e \\ | & \texttt{for } v \texttt{ in } [] \texttt{ return } e \\ | & \texttt{typeswitch } [] \\ & \texttt{case } v : t \texttt{ return } e \\ & \texttt{...} \\ & \texttt{case } v : t \texttt{ return } e \\ & \texttt{else } e \end{array}$$

$$\texttt{for } v \texttt{ in } () \texttt{ return } e \Rightarrow () \qquad (8$$

$$\texttt{for } v \texttt{ in } (e_1, e_2) \texttt{ return } e_3 \Rightarrow (\texttt{for } v \texttt{ in } e_1 \texttt{ return } e_3), (\texttt{for } v \texttt{ in } e_2 \texttt{ return } e_3) \qquad (9$$

$$\texttt{for } v \texttt{ in } e_1 \texttt{ return } e_2 \Rightarrow e_2\{e_1 / v\}, \text{ if } e_1 : u \qquad (1$$

$$\texttt{for } v \texttt{ in } e \texttt{ return } v \Rightarrow e \qquad (1$$

$$\begin{array}{l} \text{if } e \text{ has a prime type} \\ \texttt{unordered(} e \texttt{)} \end{array} \Rightarrow e \qquad (1$$

$$\texttt{unordered(unordered(} e \texttt{))} \Rightarrow \texttt{unordered(} e \texttt{)} \qquad (1$$

$$\texttt{unordered(for \$x in } e_1 \texttt{ return } e_2 \texttt{)} \Rightarrow \texttt{for \$x in unordered(} e_1 \texttt{) return unordered(} e_2 \texttt{)} \qquad (1$$

$$\texttt{unordered(} e_1 \texttt{ sortby } e_2 \texttt{)} \Rightarrow \texttt{unordered(} e_1 \texttt{) sortby } e_2 \qquad (1$$

$$\texttt{unordered(distinct(} e \texttt{))} \Rightarrow \texttt{distinct(unordered(} e \texttt{))} \qquad (1$$

$$\texttt{unordered(if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{)} \Rightarrow \texttt{if } e_1 \texttt{ then unordered(} e_2 \texttt{) else unordered(} e_3 \texttt{)} \qquad (1$$

$$\begin{array}{l} \text{if \$x free in } e_1 \text{ and \$y free in } e_2 \\ \texttt{for \$x in unordered(} e_1 \texttt{) return} \\ \texttt{for \$y in unordered(} e_2 \texttt{) return } e_3 \end{array} \Rightarrow \begin{array}{l} \texttt{for \$y in unordered(} e_2 \texttt{) return} \\ \texttt{for \$x in unordered(} e_1 \texttt{) return } e_3 \end{array} \qquad (1$$

$$E[\texttt{ if } e_1 \texttt{ then } e_2 \texttt{ else } e_3] \Rightarrow \texttt{if } e_1 \texttt{ then } E[e_2] \texttt{ else } E[e_3] \qquad (1$$

$$E[\texttt{ let } v = e_1 \texttt{ return } e_2] \Rightarrow \texttt{let } v = e_1 \texttt{ return } E[e_2] \qquad (2$$

$$E[\texttt{ for } v \texttt{ in } e_1 \texttt{ return } e_2] \Rightarrow \texttt{for } v \texttt{ in } e_1 \texttt{ return } E[e_2] \qquad (2$$

$$\begin{array}{l} E[\texttt{typeswitch } e_1 \\ \quad \texttt{case } v : t \texttt{ return } e_2 \end{array} \qquad \begin{array}{l} \texttt{typeswitch } e_1 \\ \texttt{case } v : t \texttt{ return } E[\ e_2\ ] \end{array}$$

```
case v : t return e_{n-1}          => ...                              (2
...                                  case v : t return E[ e_{n-1} ]
else e_n ]                           else E[e_n ]
```

Figure 12: Optimization Laws

Rules 8, 9, and 10 simplify iterations. Rule 8 rewrites an iteration over the empty sequence as the empty sequence. Rule 9 distributes iteration through sequence: iterating over the sequence (e_1, e_2) is equivalent to the sequence of two iterations, one over e_1 and one over e_2. Rule 10 eliminates an iteration over a single element or scalar. If e_1 is a unit type, then e_1 can be substituted for occurrences of v in e_2.

> **Ed. Note:** MF (Oct-18-2000) The rules for eliminating trivial `typeswitch` expressions need to be written. They are more complex than those for the old `case` expressions.

Rule 11 eliminates an iteration when the result expression is simply the iteration variable *v*.

Rule 12 eliminates `unordered` for expressions with a primetype, i.e., exressions that return a singleton sequence. Rule 13 shows that `unordered` is idempotent. Rules 14--17 show that `unordered` distributes with `for`, `sortby`, `distinct`, and `if`. Rule 18 commutes a nested iteration over unordered sequences. This equivalence does not hold for so called dependent joins, where the outer iteration variable `$x` is bound in the inner expression `$y`.

Rules 19--22 are used to commute expressions. Each rule actually abbreviates a number of other rules, since the *context variable E* stands for a number of different expressions. The notation *E*[*e*] stands for one of the four expressions given with expression *e* replacing the hole [] that appears in each of the alternatives. For instance, one of the expansions of Rule 21 is the following, when *e* is taken to be `for` *v* `in` [] `return` *e* :

`for` v_2 `in` (`for` v_1 `in` e_1 `return` e_2) `return` e_3 => `for` v_1 `in` e_1 `return` (`for` v_2 `in` e_2 `return` e_3)

B Issues

B.1 Introduction

The issues in [B.2 Issues list] serve as a design history for this document. The ordering of issues is irrelevant. Each issue has a unique id of the form Issue-<dddd> (where d is a digit). This can be used for referring to the issue by <url-of-this-document>#Issue-<dddd>. Furthermore, each issue has a mnemonic header, a date, an optional description, and an optional resolution. For convenience, resolved issues are displayed in green. Some of the issues contain references to W3C internal archives. These are marked with "W3C-members only". Some of the descriptions of the resolved issues are obsolete w.r.t. to the current version of the document.

> **Ed. Note:** PF (Aug-05-2000): For the sake of archival, there are some duplicate issues raised in multiple instances. Duplicate issues are marked as "resolved" with reference to the representative issue.

B.2 Issues list

Unless stated explicitly otherwise, **[Issue-0001: Attributes]** through **[Issue-0039: Dereferencing**

semantics] have been raised in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Jul/0142.html (W3C-members only).

Issue-0001: Attributes

Date: Jul-26-2000

Description: One example of the need for support of [**Issue-0049: Unordered Collections**], but also: Attributes need to be constrained to contain white space separated lists of simple types only.

Resolution Attributes are represented by @attribute-name[content]. See [**3.5 Types**] for the constraint on white space seperated lists of simple types, and [**3.1 Expressions**] for selecting and constructing attributes.

Issue-0002: Namespaces

Date: Jul-26-2000

Resolution Namespaces are represented by {uri-of-namespace}localname. See [**3.5 Types**].

Issue-0003: Global Order

Date: Jul-26-2000

Description: The data model and algebra do not define a global order on documents. Querying global order is often required in document-oriented queries.

See the thread starting at http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0179.html (W3C-members only).

Resolution Resolved by adding < operator defined on nodes in same document. See [**2.12 Querying order**]. See [**Issue-0079: Global order between nodes in different documents**] for order between nodes in different documents.

Issue-0004: References vs containment

Date: Jul-26-2000

Description: The query-algebra datamodel currently does not explicitly model children-elements by references (other than the XML-Query Datamodel. This facilitates presentation, but may be an oversimplification with regard to [**Issue-0005: Element identity**].

Resolution This issue is resolved by subsumption as follows: (1) As [**5 Dynamic Semantics : Value-Inference Rules**] points out, all child-elements are (implicit) references to nodes. (2) Thus, having resolved [**Issue-0005: Element identity**] this issue is resolved too.

Issue-0005: Element identity

Date: Jul-26-2000

Description: Do expressions preserve element identity or don't they? And does "=" and distinct use comparison by reference or comparison by value?

Resolution The first part of the question has been resolved by resolution of [**Issue-0010: Construct values by copy**]. The second part raises a more specific issue [**Issue-0066: Shallow or Deep Equality?**].

Issue-0006: Source and join syntax instead of "for"

Date: Jul-26-2000

Description: Another term for "source and join syntax" is "comprehension". See [**A Equivalences**] for a discussion of the relationship between iteration by for and comprehension syntax.

Resolution This issue is resolved by subsumption under [**Issue-0021: Syntax**]. List comprehension is a syntactic alternative to "for v in e1 do e2", which has been favored by the WG in the resolution of [**Issue-0021: Syntax**].

Issue-0007: References: IDREFS, Keyrefs, Joins

Date: Jul-26-2000

Description: Currently, the Algebra does not support reference values, such as IDREF, or Keyref (not to be mixed up with "node-references" - see [**Issue-0005: Element identity**], which are defined in the XML Query Data Model. The Algebra's type system should be extended to support reference types and the data model operators `ref`, and `deref` should be supported (similar to id() in XPath).

Resolution Delegated to XPath 2.0. Algebra should adopt solutions (e.g., id()/keyref() functions) provided in XPath 2.0. There may be an interaction between IDREFs and RefNodes, but we're not going to cover that now.

Issue-0008: Fixed point operator or recursive functions

Date: Jul-26-2000

Description: It may be useful to add a fixed-point operator, which can be used in lieu of recursive functions to compute, for example, the transitive closure of a collection.

Currently, the Algebra does not guarantee termination of recursive expressions. In order to ensure termination, we might require that a recursive function take one argument that is a singleton element, and any recursive invocation should be on a descendent of that element; since any element has a finite number of descendents, this avoids infinite regress. (Ideally, we should have a simple syntactic rule that enforces this restriction, but we have not yet devised such a rule.)

Impacts optimization; hard to do static type inference; current algebra is first-order

See for the subproblem of typing "//" or desc() http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0187.html (W3C-members only).

Issue-0009: Externally defined functions

Date: Jul-26-2000

Description: There is no explicit support for externally defined functions.

The set of built-in functions may be extended to support other important operators.

See also http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0003.html (W3C-members only).

Resolution Algebra editors endorse a solution that uses XP for specifying signatures of external functions. Algebra will adopt solution provided by XQuery.

Issue-0010: Construct values by copy

Date: Jul-26-2000

Description: Need to be able to construct new types from bits of old types by reference and by copy. Related to [**Issue-0005: Element identity**].

Resolution The WG wishes to support both: construction of values by copy, as well as references to original nodes (http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only)) See also http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0155.html (W3C-members only) (W3C-members only). This needs some further investigation to sort out all technical difficulties (see [**Issue-0062: Open questions for constructing elements by reference**]) so the change has not yet been reflected in the Algebra document.

Issue-0011: XPath tumbler syntax instead of index?

Date: Jul-26-2000

Description: XPath provides as a shorthand syntax [integer] to select child-elements by their position on the sibling axes, whereas the xml-query algebra uses a combination of a built-in function index() and iteration. See http://lists.w3.org/Archives/Member/w3c-archive/2000Sep/0168.html (W3C-members only) for a suggestion to to support indexed iteration in the form "for v sub i in e1 do e2", and to express index() as a function (or macro).

Addendum by JS (submitted by MF) Dec 19/2000: The typing of index is lossy : it produces a factored type. Jerome suggests the more precise `range` operator:

```
e : q min m  max n   n' - (m'-1) = r  m' >= m   n' <= n
-----------------------------------------------
            range(e;m';n') : q min r  max r

nth(e;n) == range(e;n;n)
```

The `range` operator takes a repetition of prime types and those values in the range `m'` to `n'`; if the repetition does not include that range, a run-time error is raised. The `range` and `nth` operators could also be defined in terms of `head` and `tail` and polymorphic recursive functions. In the absence of parameteric polymorphism, it is not possible to define `range` and `nth` with precise types.

Here are Peter's rules:

```
e : p min m  max n     n!=*
-------------------------------------------------
range(e;m';n') : p{n'-max(m,m')+1,min(n',n)-m'+1}

For example:
let v1 = a[] min 2  max 4

range(v1;3;3): a[] min 1  max 1
range(v1;1;3): a[] min 2  max 3
range(v1;3;5): a[] min 1  max 2
range(v1;1;5): a[] min 2  max 4

e : p min m  max *
-----------------------------
range(e;m';n') : p min 0  max n'-m'+1

let v2 = a[] min 0  max *

range(v2;1;3): a[] min 0  max 2

this follows the typical semantics for head() and tail():
head(()) = tail(()) = ()
and the semantics behind
range(e;m',n') = tail o ...(m' times)   ... o tail o head,
                 tail o ...(m'+1 times) ... o tail o head,
                 ...
                 tail o ...(n' times)   ... o tail o head
```

I would have no troubles in restricting ourselves to nth() instead of range() in the algebra (range can always be enumerated by nth()). Furthermore, we should consider whether m',n' can be computed numbers.

Issue-0012: GroupBy - needs second order functions?

Date: Jul-26-2000

Description: The type system is currently first order: it does not support function types nor higher-order functions. Higher-order functions are useful for specifying, for example, sorting and grouping operators, which take other functions as arguments.

Resolution The WG has decided to express groupBy by a combination of `for` and `distinct` (see also http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only) and [**Issue-0042: GroupBy**]):. Thus w.r.t. to GroupBy this Issue is resolved. Because GroupBy is not the only use case for higher order functions, a new issue [**Issue-0063: Do we need (user defined) higher order functions?**] is raised.

Issue-0013: Collations

Date: Jul-26-2000

Description: Collations identify the ordering to be applied for sorting strings. Currently, it is considered to have an (optional parameter) collation "name" as follows: "SORT variable IN exp BY +(expression {ASCENDING|DESCENDING} {COLLATION name}) (see http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only)). An alternative would be to model a collation as a simple type derived from string, and use type-level casting, i.e. expression :collationtype (which is already supported in the XML Query Algebra), for specifying the collation. That would make: "SORT variable IN exp BY +(expression:collationname {ASCENDING|DESCENDING}). But that requires some support from XML-Schema.

More generally, collations are important for any operator in the Algebra that involves string comparison, among them: sort, distinct, "=" and "<".

Resolution Formal semantics will adopt solution provided by Operators.

Issue-0014: Polymorphic types

Date: Jul-26-2000

Description: The type system is currently monomorphic: it does not permit the definition of a function over generalized types. Polymorphic functions are useful for factoring equivalent functions, each of which operate on a fixed type.

The current type system has already a built-in polymorphic type (lists) and is likely to have more (unordered collections). The question is, whether to allow for user-defined polymorphic types and user defined polymorphic functions.

See also thread around http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0111.html (W3C-members only) (W3C-members only).

Issue-0015: 3-valued logic to support NULLs

Date: Jul-26-2000

Issue-0016: Mixed content

Date: Jul-26-2000

Description: The XML-Query Algebra allows to generate elements with an arbitrary mixture of data (of simple type) and elements. XML-Schema only allows for a combination of strings interspersed with elements (aka mixed content). We need to figure out whether and how to constrain the XML-Query Algebra accordingly (e.g. by typing rules?)

Resolution The type system has been extended to support the interleaving operator & - see [**3.5 Types**]. Mixed content is defined in terms of &.

Issue-0017: Unordered content

Date: Jul-26-2000

Description: All-groups in XML-Schema, not to be mixed up with [**Issue-0049: Unordered Collections**]

Resolution The type system has been extended with the support of all-groups - see

[3.5 Types].

Issue-0018: Align algebra types with schema

Date: Jul-26-2000

Description: The Algebra's internal type system is the type system of XDuce. A potentially significant problem is that the Algebra's types may lose information when converted into XML Schema types, for example, when a result is serialized into an XML document and XML Schema.

James Clark points out : "The definition of AnyComplexType doesn't match the concrete syntax for types since it applies unbounded repetition to AnyTree and one alternative for AnyTree is AnyAttribute." This is another example of an alignment issue.

This issue comprises also issues **[Issue-0016: Mixed content]**, **[Issue-0017: Unordered content]**, **[Issue-0053: Global vs. local elements]**, **[Issue-0054: Global vs. local complex types]**, **[Issue-0019: Support derived types]**, substitution groups.

Issue-0019: Support derived types

Date: Jul-26-2000

Description: The current type system does not support user defined type hierarchies (by extension or by restriction).

Issue-0020: Structural vs. name equivalence

Date: Jul-26-2000

Description: The subtyping rules in **[4.1 Relating data to types]** only define structural subtyping. We need to extend this with support for subtyping via user defined type hierarchies - this is related to **[Issue-0019: Support derived types]**.

Issue-0021: Syntax

Date: Jul-26-2000

Description: (e.g. for.<-.in vs for.in.do)

Resolution The WG has voted for several syntax changes (see also http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt(W3C-members only) (W3C-members only), **[3.1 Expressions]**: "for v in e do e", "let v = e do", "sort v in e by e ...", "distinct", "match case v:t e ... else e".

Issue-0022: Indentation, Whitespaces

Date: Jul-26-2000

Description: Is indentation significant?

Resolution The WG has consensus that indentation is not significant (see http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Aug/0043.html (W3C-members only) (W3C-members only)), i.e., all documents are white space normalized.

Issue-0023: Catch exceptions and process in algebra?

Date: Jul-26-2000

Description: Does the Algebra give explicit support for catching exceptions and processing them?

Resolution Subsumed by new issue [**Issue-0064: Error code handling in Query Algebra**].

Issue-0024: Value for empty sequences

Date: Jul-26-2000

Description: What does "value" do with empty sequences?

Resolution The definition of value(e) has changed to:

```
value(e) = typeswitch children(e)
                     case v: AnyScalar do v
                     else()
```

Furthermore, the typing rules for "for v in e1 do e2" have been changed such that the variable v is typed-checked seperately for each unit-type occuring in expression e1.

Consequently the example in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Jul/0138.html (W3C-members only) (W3C-members only) would be typed as follows:

```
query for b in b0/book do
      value(b/year): xs:integer min 0 max *
```

rather than leading to an error.

Issue-0025: Treatment of empty results at type level

Date: Jul-26-2000

Description: According to http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Jul/0138.html(W3C-members only) (W3C-members only) this is closely related to [**Issue-0024: Value for empty sequences**].

Resolution Resolved by resolution of [**Issue-0025: Treatment of empty results at type level**].

Issue-0026: Project - one tag only

Date: Jul-26-2000

Description: Project is only parameterized by one tag. How can we translate a0/(b | c)?

Resolution With the new syntax (and type system) a0/(b **|** c) can be translated to "for v in a0 do typeswitch case v1:b[AnyType] do v1 case v2:c[AnyType] do c else ()" - see also [**A.1 Relating projection to iteration**].

Issue-0027: Case syntax

Date: Jul-26-2000

Description: N-ary case can be realized by nested binary cases. For design alternatives of case see: http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Aug/0017.html (W3C-members only) (W3C-members only)

Resolution New (n-ary) case syntax is introduced in [**3.1 Expressions**].

Issue-0028: Fusion

Date: Jul-26-2000

Description: Does the Algebra support fusion as introduced by query languages such as LOREL? This is related to [**Issue-0005: Element identity**], because fusion only makes sense with support of element identity.

Resolution Fusion is equivalent to 'natural full-outer join'. XQuery can reraise issue if desired. If added, the Algebra editors should review any solution w.r.t typing.

Issue-0029: Views

Date: Jul-26-2000

Description: One of the problems in views: Can we undeclare/hide things in environment? For example, if we support element-identity, can we explicitly discard a parent, and/or children from an element in the result-set? Related to [**Issue-0005: Element identity**]. See also description in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0047.html (W3C-members only) (W3C-members only).

Resolution XQuery can reraise issue if desired. If added, the Algebra editors should review any solution w.r.t typing.

Issue-0030: Automatic type coercion

Date: Jul-26-2000

Description: What do we do if a value does not have a type or a different type from what is required? See also thread around http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0071.html (W3C-members only) (W3C-members only). This link also contains a recommendation, which has been agreed as the general direction to go in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0196.html (W3C-members only) (W3C-members only):

Suggested Resolution: We believe that the XML Query Language should specify default type coercions for mixed mode arithmetic should be performed according to a fixed precedence hierarchy of types, specifically integer to fixed decimal, fixed decimal to float, float to double. This policy has the advantage of simplicity, tradition, and static type inference. Programmers could explicitly specify alternative type coercions when desirable.

Resolution Delegation to XPath 2.0, XQuery, and/or Operators.

Issue-0031: Recursive functions

Date: Jul-26-2000

Resolution subsumed by [**Issue-0008: Fixed point operator or recursive functions**]

Issue-0032: Full regular path expressions

Date: Jul-26-2000

Description: Full regular path expressions allow to constrain recursive navigation along paths by means of regular expressions, e.g. a/b*/c denotes all paths starting with an a, proceeding with arbitrarily many b's and ending in a c. Currently the XML-Query Algebra can express this by means of (structurally) recursive functions. An alternative may be the introduction of a fixpoint operator [**Issue-0008: Fixed point operator or recursive functions**].

Resolution XPath 2.0 can raise issue if desired. The Algebra editors should review any solution w.r.t typing.

Issue-0033: Metadata Queries

Date: Jul-26-2000

Description: Metadata queries are queries that require runtime access to type information. See also discussion starting at http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0087.html (W3C-members only) (W3C-members only).

Issue-0034: Fusion

Date: Jul-26-2000

Resolution Identical with [**Issue-0028: Fusion**]

Issue-0035: Exception handling

Date: Jul-26-2000

Resolution Subsumed by [**Issue-0023: Catch exceptions and process in algebra?**] and [**Issue-0064: Error code handling in Query Algebra**].

Issue-0036: Global-order based operators

Date: Jul-26-2000

Resolution Subsumed by [**Issue-0003: Global Order**]

Issue-0037: Copy vs identity semantics

Date: Jul-26-2000

Resolution subsumed by [**Issue-0005: Element identity**]

Issue-0038: Copy by reachability

Date: Jul-26-2000

Description: Is it possible to copy children as well as IDREFs, Links, etc.? Related to [**Issue-0005: Element identity**] and [**Issue-0008: Fixed point operator or recursive functions**]

Resolution Resolved by addition of "deep" copy operator in [XQuery 1.0 and XPath 2.0 Data Model].

Issue-0039: Dereferencing semantics

Date: Jul-26-2000

Resolution subsumed by [**Issue-0005: Element identity**]

[**Issue-0040: Case Syntax**] through [**Issue-0047: Attributes**] are raised in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Aug/0010.html (W3C-members only) (W3C-members only)

Issue-0040: Case Syntax

Date: Aug-01-2000

Description: We suggest that the syntax for "case" be made more regular. At present, it takes only two branches, the first labelled with a tag-name and the second labelled with a variable. A more traditional syntax for "case" would have multiple branches and label them in a uniform way. If the algebra is intended only for semantic specification, "case" may not even be necessary.

Resolution subsumed by [**Issue-0027: Case syntax**]

Issue-0041: Sorting

Date: Aug-01-2000

Description: We are not happy about the three-step sorting process in the Algebra. We would prefer a one-step sorting operator such as the one illustrated below, which handles multiple sort keys and mixed sorting directions: SORT emp <- employees BY emp/deptno ASCENDING emp/salary DESCENDING

Resolution The WG has decided to go for the above syntax, with an (optional) indication of COLLATION. (see http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only), [**2.13 Sorting**]).

Issue-0042: GroupBy

Date: Aug-01-2000

Description: We do not think the algebra needs an explicit grouping operator. Quilt and other high-level languages perform grouping by nested iteration. The algebra can do the same.

related to [**Issue-0012: GroupBy - needs second order functions?**]

Resolution The WG has decided (see http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only)) to skip groupBy for the time being (see also revised [**2.11 Restructuring and grouping**] and raise [**Issue-0069: Organization of Document**] for a possible future revision of this resolution.

Issue-0043: Recursive Descent for XPath

Date: Aug-01-2000

Description: The very important XPath operator "//" is supported in the Algebra only by writing a recursive function. This is adequate for a semantic specification, but if the Algebra is intended as an optimizable target language it will need better support for "//" (possibly in the form of a fix-point operator.)

Resolution Resolved by subsumption under [**Issue-0043: Recursive Descent for XPath**] (see http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt(W3C-members only) (W3C-members only)).

Issue-0044: Keys and IDREF

Date: Aug-01-2000

Description: We think the algebra needs some facility for dereferencing keys and IDREFs (exploiting information in the schema.)

Resolution Subsumed by [**Issue-0007: References: IDREFS, Keyrefs, Joins**]

Issue-0045: Global Order

Date: Aug-01-2000

Description: We are concerned about absence of support for operators based on global document ordering such as BEFORE and AFTER.

Resolution subsumed by [**Issue-0003: Global Order**]

Issue-0046: FOR Syntax

Date: Aug-01-2000

Description: We agree with comments made in the face-to-face meeting about the aesthetics of the Algebra's syntax for iteration. For example, the following syntax is relatively easy to understand: FOR x IN some_expr EVAL f(x) whereas we find the current algebra equivalent to be confusing and misleading: FOR x <- some_expr IN f(x) This syntax appears to assign the result of some_expr to variable x, and uses the word IN in a non-intuitive way.

Resolution subsumed by [**Issue-0021: Syntax**]

Issue-0047: Attributes

Date: Aug-01-2000

Description: See [**Issue-0001: Attributes**].

Resolution subsumed by [**Issue-0001: Attributes**]

[**Issue-0048: Explicit Type Declarations**] through [**Issue-0050: Recursive Descent for XPath**] are raised in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Jul/0148.html (W3C-members only) (W3C-members only)

Issue-0048: Explicit Type Declarations

Date: Jul-27-2000

Description: Type Declaration for the results of a query: The issue is whether to auto construct the result type from a query or to pre-declare the type of the result from a query and check for correct type on the return value. Suggestion: Support for pre-declared result data type and as well as to coerce the output to a new type is desirable. Runtime or compile time type checking is to be resolved? Once you attach a name to a type, it is preserved during the query processing.

Resolution W.r.t. compile time type casts this is already possible with e:t (see [**3.1 Expressions**]). For run-time casts an issue has been raised in [**Issue-0062: Open questions for constructing elements by reference**].

Issue-0049: Unordered Collections

Date: Jul-27-2000

Description: Currently, all sequences in the data model are ordered. It may be useful to have unordered forests. The `distinct-node` function, for example, produces an inherently unordered forest. Unordered forests can benefit from many optimizations for the relational algebra, such as commutable joins.

Handling of collection of attributes is easy but the collection of elements is complex due to complex type support for the elements. It makes sense to allow casting from unordered to ordered collection and vice versa. It is not clear whether the new ordered or unordered collection is a new type or not. It affects function resolution, optimization.

See also thread around http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0135.html (W3C-members only) (W3C-members only).

Our request to Schema to represent insignificance of ordering at schema level has not been fulfilled - see http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0136.html (W3C-members only) (W3C-members only). Thus we need to be aware that this information may get lost, when mapping to schema.

Resolution Unordered collections are described by {t} see [**3.5 Types**], some operators (sort, distinct-node, for, and sequence) are overloaded, and some operators (difference, intersection) are added). A new issue [**Issue-0076: Unordered types**] is raised.

Issue-0050: Recursive Descent for XPath

Date: Jul-27-2000

Description: Suggestion: The group likes to add a support for fixed-point operator in the query language that will allow us to express the semantics of the // operator in an xpath expression. A path expression of the form a//b may be represented by a fixed-point operator fp(a, "/.")/b.

Resolution subsumed by [**Issue-0043: Recursive Descent for XPath**]

Issue-0051: Project redundant?

Date: Aug-05-2000

Description: It appears that project a e could be reduced to sth. like

```
for v <- e in
  case  v of a[v1] => a[v1]
           | v2 => ()
```

... or would that generate a less precise type?

Resolution With the new type system and handling of the for operator, project is indeed redundant. See [**A.1 Relating projection to iteration**].

Issue-0052: Axes of XPath

Date: Aug-05-2000

Description: The current algebra makes navigation to parents difficult to impossible. With support of Element Identity [**Issue-0005: Element identity**] and recursive functions [**Issue-0008: Fixed point operator or recursive functions**] one can express parent() by a recursive function via the document root. More direct support needs to be investigated w.r.t its effect on the type system.

The WG wishes to support a built-in operator parent() (see http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only)). For the current state of affairs see thread around http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0074.html (W3C-members only) (W3C-members only). For some use-cases see http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0011.html (W3C-members only) (W3C-members only).

Resolution XPath 2.0 and XQuery can reraise issue if desired. Algebra should review any solution w.r.t typing. Question: whether namespace axis (i.e., access namespace nodes) will be included in XQuery. Algebra currently has issues related to typing of parent() and descendent(). If sibling axes are included in XQuery, then Algebra should review w.r.t. typing.

Issue-0053: Global vs. local elements

Date: Aug-05-2000

Description: The current type system cannot represent global element-declarations of XML-Schema. All element declarations are local.

Issue-0054: Global vs. local complex types

Date: Aug-05-2000

Description: The current type system does not distinguish between global and local types as XML-Schema does. All types appear to be fully nested (i.e. local types)

Issue-0055: Types with non-wellformed instances

Date: Aug-05-2000

Description: The type system and algebra allows for sequences of simple types, which can usually be not represented as a well-formed document. How shall we constrain this? Related to [**Issue-0016: Mixed content**].

Issue-0056: Operators on Simple Types

Date: Jul-15-2000

Description: We intentionally did not define equality or relational operators on element and simple type. These operators should be defined by consensus.

See also first designs for support of arithmetic operators http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0138.html (W3C-members only) (W3C-members only) and for support of operators for date/time http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0113.html (W3C-members only) (W3C-members only).

Ed. Note: MF, 15-Jan-2001 A joint task force on operators with members from the XSLT, XML Schema, and XML Query working groups is chartered to define arithmetic operators.

Resolution XQuery formal semantics adopts solution provided by Operators task force.

Issue-0057: More precise type system; choice in path

Date: Aug-07-2000

Description: (This subsumes [**Issue-0051: Project redundant?**]). If the type system were more precise, then (project a e) could be replaced by:

```
for v <- e in
    case v of
```

```
      a[v1] => a[v1]
    | v2 => ()
```

One could also represent (e/(a|b)) directly in a similar style.

```
for v <- e in
    case v of
      a[v1] => a[v1]
    | v2 => case v2 of
               b[v3] => b[v3]
            | v4 => ()
```

Currently, there is no way to represent (e/(a|b)) without loss of precision, so if we do not change the type system, we may need to have some way to represent (e/(a|b)) and similar terms without losing precision. (The LA team has a design for this more precise type system, but it is too large to fit in the margin of this web page!)

Resolution See resolution of [**Issue-0051: Project redundant?**]

Issue-0058: Downward Navigation only?

Date: Aug-07-2000

Description: Related to [**Issue-0052: Axes of XPath**]. The current type system (and the more precise system alluded to in [**Issue-0057: More precise type system; choice in path**]) seems well suited for handling XPath children and descendent axes, but not parent, ancestor, sibling, preceding, or following axes. Is this limitation one we can live with?

Resolution Subsumed by [**Issue-0052: Axes of XPath**]

Issue-0059: Testing Subtyping

Date: Aug-07-2000

Description: One operation required in the Algebra is to test whether XML type t1 is a subtype of XML type t2, indicated by writing t1 <: t2. There is a well-known algorithm for this, based on tree automata, which is a straightforward variant of the well-known algorithm for testing whether the language generated by one regular-expression is a subset of the language generated by another. (The algorithm involves generating deterministic automata for both regular expressions or types.)

However, the naive implementation of the algorithm for comparing XML types can be slow in practice, whereas the naive algorithm for regular expressions is tolerably fast. The only acceptably fast implementation of a comparison for XML types that the LA team knows of has been implemented by Haruo Hasoya, Jerome Voullion, and Benjamin Pierce at the University of Pennsylvania, for their implementation of Xduce. (Our implementation of the Algebra re-uses their code, with permission.)

So, should we adopt a simpler definition of subtyping which is easier to test? One possibility is to adopt the sibling restriction from Schema, which requires that any two elements which appear a siblings in the same content model must themselves have contents of the same type. Jerome Simeon and Philip Wadler discovered that adopting the sibling restriction reduces the problem of checking subtyping of XML types to that of checking regular languages for inclusion, so it may be worth adopting the restriction for that reason.

Issue-0060: Internationalization aspects for strings

Date: Jun-26-2000

Description: These issues are taken from the comments on the Requirements Document by I18N (http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Jun/0137.html (W3C-members only) (W3C-members only)).

Further information can be found at http://www.w3.org/TR/WD-charreq.

It is a goal of i18n that queries involving string matching ("select x where x='some_constant'") treat canonically equivalent strings (in the Unicode sense) as matching. If the query and the target are both XML, early normalization (as per the Character Model) is assumed and binary comparison ensures that the equivalence requirement is satisfied. However, if the target is originally a legacy database which logically has a layer that exports the data as XML, that XML must be exported in normalized form. The XML Query spec must impose the normalization requirement upon such layers.

Similarly, the query may come from a user-interface layer that creates the XML query. The XML Query spec must impose the normalization requirement upon such layers.

Provided that the query and the target are in normalized form C, the output of the query must itself be in normalized form C.

Queries involving string matching should support various kinds of loose matching (such as case-insensitivity, katakana-hiragana equivalence, accent-accentless equivalence, etc.)

If such features as case-insensitivity are present in queries involving string matching, these features must be properly internationalized (e.g. case folding works for accented letters) and language-dependence must be taken into account (e.g. Turkish dotless-i).

Queries involving character counting and indexing must take into account the Character Model. Specifically, they should follow Layer 3 (locale-independent graphemes). Additional details can be found in The Unicode Standard 3.0 and UTR#18. Queries involving word counting and indexing should similarly follow the recommendations in these references.

Resolution XQuery formal semantics adopts solution provided by Operators task force.

Issue-0061: Model for References

Date: Aug-16-2000

Description: Raised in: http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Aug/0063.html (W3C-members only) (W3C-members only). Related to a number of issues around [**Issue-0005: Element identity**].

- Use Cases

  Table of Contents

REF *could* do this well if it were restructured - it does not maintain unforeseen relationships or use them...

Bibliographies

Recursive parts

RDF assertions

Inversion of simple parent/child references (related to **[Issue-0058: Downward Navigation only?]**).

- What can we leave out?

  can we leave out transitive closure?

  can we limit recursion?

  can we leave out fixed point recursion?

  related to **[Issue-0008: Fixed point operator or recursive functions]**

- Do we need to be able to...

  a. Find the person with the maximum number of descendents?

  b. Airplane routes: how can I get from RDU to Raleigh? (fixed point: guaranteeing termination in reasonable time...)

  c. Given children and their mothers, can I get mothers and their children? (without respect to the form of the original reference...)

  related to **[Issue-0008: Fixed point operator or recursive functions]**.

- Should we abstract out the difference between different kinds of references? If so, should we be able to cast to a particular kind of reference in the output?

  a. abstracting out the differences is cheaper, which is kewl...

  b. the kind of reference gives me useful information about: locality (same document, same repository, big bad internet...) static vs. dynamic (xpointer *may* be resolved dynamically, or *may* be resolved at run time, ID/IDREF is static).

  related to **[Issue-0007: References: IDREFS, Keyrefs, Joins]**.

- do we need to be able to generate ids, e.g. using skolem functions?

  for a document in RAM, or in a persistent tree, identity may be present, implicit, system dependent, and cheap - it's nice to have an abstraction that requires no more than the implicit identity persistable ID is more expensive, may want to be able to serialize with ID/IDREF to instantiate references in the data model

can use XPath instead of generating ID/IDREF, but these references are fragile, and one reason for queries is to create data that may be processed further

persistable ID unique within a repository context

persistable ID that is globally unique

related to [**Issue-0005: Element identity**].

- copy vs. reference semantics

  "MUST not preclude updates..."

  in a pure query environment, sans update, we do not need to distinguish these

  if we have update, we may need to distinguish, perhaps in a manner similar to "updatable cursors" in SQL

  programs may do queries to get DOM nodes that can that be modified. It is essential to be able to distinguish copies of nodes from the nodes themselves.

  copy semantics - what does it mean?

  copy the descendent hierarchy?

  copy the reachability tree? (to avoid dangling references)

  related to [**Issue-0038: Copy by reachability**].

Resolution Handled in current data model and algebra

The following issues have been raised since Sep-25-2000.

Issue-0062: Open questions for constructing elements by reference

Date: Sep-25-2000

Description: (1) What is the value of parent() when constructing new elements with children refering to original nodes? See also discussion at http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0155.html (W3C-members only) (W3C-members only).

(2) Is an approach to either make copies for all children or provide references to all children, or should we allow for a more flexible combination of copies and references?

Resolution Operational semantics specifies that element node constructor creates copies of all its children. Addition of RefNode in [XQuery 1.0 and XPath 2.0 Data Model] supports explicit reference value.

Issue-0063: Do we need (user defined) higher order functions?

Date: Oct-16-2000

Description: The current XML-Query-Algebra does not allow functions to be parameters of another function - so called higher order functions. However, most of the Algebra operators are (built-in) higher functions, taking expressions as an argument ("sort", "for", "case" to name a few). Even a fixpoint operator, "fun f(x)=e, fix f(x) in e" (see also [**Issue-0008: Fixed point operator or recursive functions**]), would be a built-in higher order function.

Resolution As agreed in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0196.html (W3C-members only) (W3C-members only) the XML Query Algebra will not support user defined higher order functions. It does support a number of built-in higher order functions.

Issue-0064: Error code handling in Query Algebra

Date: Oct-04-2000

Description: How do we return an error code from a function defined in current Query algebra. Do we need to create an array (or a structure) to merge the return value and error code to do this. If that is true, it may be inefficient to implement. In order for cleaner and efficient implementation, it may be necessary to allow a function declaration to take a parameter of type "output" and allow it to return an error code as part of the function definition. See also thread starting with http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0051.html(W3C-members only) (W3C-members only).

Resolution One does not need to create a structure to combine return values with error codes, provided each operator or function /either/ returns a value /or/ raises an error. The XML-Query Algebra supports means to raise errors, but does not define standard means to catch errors. Raising errors is accomplished by the expression "error" of type Ø (empty choice). Because Ø **|** t = t, such runtype errors do not influence static typing. The surface syntax and/or detailed specification of operators on simple types (see [**Issue-0056: Operators on Simple Types**]) may choose to differentiate errors into several error-codes.

Issue-0065: Built-In GroupBy?

Date: Oct-16-2000

Description: As discussed in http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only), we may revisit the resolution of [**Issue-0042: GroupBy**] and reintroduce GroupBy along the lines of sort: "group v in e1 by [e2 {collation}]". One reason for this may be that this allows to use collation for deciding about the equality of strings.

Resolution In http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0196.html (W3C-members only) (W3C-members only) the WG has decided to close this issue, and for the time being not consider GroupBy as a built-in operator. Furthermore, [**Issue-0013: Collations**] is ammended to deal with collations for all operators involving a comparison of strings.

Issue-0066: Shallow or Deep Equality?

Date: Oct-16-2000

Description: What is the meaning of "=" and "distinct"? Equality of references to nodes or deep equality of data?

Resolution [XQuery 1.0 and XPath 2.0 Data Model] defines "=" (value equality) and "==" (identity equality) operators. Description of distinct states that it uses "==".

Issue-0067: Runtime Casts

Date: Sep-21-2000

Description: In some contexts it may be desirable to cast values at runtime. Such runtime casts lead to an error if a value cannot be cast to a given type. See also http://www.w3.org/XML/Group/2000/09/ql/unedited-minutes-day1.txt (W3C-members only) (W3C-members only), where the Algebra team has been put in charge of introducing run-time casts into the Algebra.

Resolution `cast` *e* `:` *t* has been introduced as a reducible operator expressed in terms of `typeswitch`.

Issue-0068: Document Collections

Date: Oct-16-2000

Description: Per our requirements document we are chartered to support document collections. The current XML-Query Algebra deals with single documents only. There are a number of subissues:

(a) Do we need a more elaborate notion of node-references? E.g. pair of (URI of root-node, local node-ref)

(b) Does the namespace mechanism suffice to type collections of nodes from different documents? Probably yes.

(c) Provided (a) and (b) can be settled, will the approach taken for [**Issue-0049: Unordered Collections**] do the rest?

Issue-0069: Organization of Document

Date: Oct-16-2000

Description: The current document belongs more to the genre (scientific) paper than to the genre specification. One may consider the following modifications: (a) reorganize intro to give a short overview and then state the purpose (strongly typed, neutral syntax with formal semantics as a basis for possibly multiple syntaxes, etc.) (compared to version Aug-23, this version has already gone a good deal in this direction). (b) Equip various definitions and type rules with id's. (c) Elaborate appendices on mapping XML-Query-Algebra Model vs. XML-Query-Datamodel, XML-Query-Type System vs. XML-Schema-Type System. (d) Maybe add an appendix on use-case-solutions. The problem is of course: Part of this is a lot of work, and we may not achieve all for the first release.

Resolution At http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0196.html (W3C-members only) (W3C-members only) the WG decided to dispose of this issue. The current overall organization of the document is quite adequate, but of course editorial decisions will have to made all the time.

Issue-0070: Stable vs. Unstable Sort/Distinct

Date: Oct-02-2000

Description: Should sort (and distinct) be stable on ordered collections, i.e. lists, and unstable on unordered collections (see [**Issue-0049: Unordered Collections**])? For more details see thread around http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0007.html (W3C-members only) (W3C-members only).

Resolution sort and distinct are stable on ordered collections, and unstable on unordered collections - see [**4.7 Typing unordered expressions**].

Issue-0071: Alignment with the XML Query Datamodel

Date: Sep-26-2000

Description: Currently, the XML Query Algebra Datamodel does not model PI's and comments. For more details see thread starting with http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0167.html (W3C-members only) (W3C-members only).

Resolution Addition of operational semantics defines relationship of Algebra to Data Model.

Issue-0072: Facet value access in Query Algebra

Date: Oct-04-2000

Description: Each of the date-time data types have facet values as defined by the schema data types draft spec. This problem is general enough to be applied to other simple data types.

The question is : Should we provide access to these facet values on an instance of a particular data types? If so, what type of access? My take is the facets are to be treated like read-only attributes of a data instance and one should have a read access to them. See also thread starting at http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0044.html (W3C-members only) (W3C-members only).

Issue-0073: Facets for simple types and their role for typechecking

Date: Oct-16-2000

Description: XML-Schema introduces a number of constraining facets http://www.w3.org/TR/xmlschema-2/ for simple types (among them: length, pattern, enumeration, ...). We need to figure out whether and how to use these constraining facets for type-checking. See also thread starting at http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Oct/0146.html(W3C-members only) (W3C-members only).

Issue-0074: Operational semantics for expressions

Date: Nov-16-2000

Description: It is necessary to add an operational semantics that formally defines each operator in the Algebra.

Resolution Added in [**5 Dynamic Semantics : Value-Inference Rules**]

Issue-0075: Overloading user defined functions

Date: Nov-17-2000

Description: User defined functions can not be overloaded in the XML Query Algebra, i.e., a function is exclusively identified by its name, and not by its signature. Should this restriction be relaxed and if so - to which extent?

Resolution No overloading in Query 1.0

Issue-0076: Unordered types

Date: Dec-11-2000

Description: Currently unorderedness is represented at type level by {t}, and some (built-in) operators are overloaded such they have different semantics (and potentially different return type) depending on their input type. An alternative is to not represent unorderedness at type level, but rather support unordered for, unordered (unstable) sort, unordered (unstable) distinct.

Resolution Removed unordered types from type system. Added support for unordered operator.

Issue-0077: Interleaved repetition and closure

Date: Dec-12-2000

Description: Regular Languages are closed w.r.t. to the interleaved product. However, they are not closed w.r.t. to interleaved repetition, which can (e.g) generate the 1 degree Dyck language D[1] = () | a D[1] b | D[1] D[1] = (a,b)^{0,*}, and more generally, any language that coordinates cardinalities of individual members from an alphabeth: E.g. (a ^ b)^ min 0 max * = all strings with equally many a's and b's. These are beyond regular languages. Should we thus try to do without interleaved repetition?

Resolution if we use interleaved repetition (which we will because it is in MSL), they will be restricted to prime types.

Issue-0078: Generation of ambiguous types

Date: Dec-12-2000

Description: Unambiguous content-models in XML 1.0 and XML Schema are not closed w.r.t. union. It appears that the XML Query-Algebra can generate result types which can not be transformed to an unambiguous content-model.

Issue-0079: Global order between nodes in different documents

Date: Dec-16-2000

Description: The global order operator < is defined on nodes in the same document, but not between nodes in different documents.

Issue-0080: Typing of parent

Date: Dec-16-2000

Description: Currently, the `parent` operator yields an imprecise type : `AnyElement min 0 max 1`. It might be possible to type `parent` more precisely, for example, by using the normalized names in MSL, which encode containment of types.

Issue-0081: Lexical representation of Schema simple types

Date: Jan-17-2001

Description: Schema simple types must be defined for the Algebra and XQuery.

Resolution Algebra will adopt lexical reps supported by XQuery.

Issue-0082: Type and expression operator precedence

Date: Jan-17-2001

Description: The precedence of expression operators and type operators is not defined.

Resolution For expression operators, Algebra adopts solution given in XQuery. For type operators, Algebra specifies precidence.

Issue-0083: Expressive power and complexity of typeswitch expression

Date: Jan-17-2001

Description: When processing an XML document without schema information, i.e., the type of the document is AnyComplexType, then match expressions may be very expensive to evaluate:

```
typeswitch x
case t1 : AnyTree do 1
case t2 : AnyTree min 0 max 2 do 2
case t3 : *[*[*[*[* ... [AnyAttribute] ]]]] do 3
else ERROR
```

typeswitch itself is not the issue. The real problem is having very liberal type patterns. We could restrict the kinds of type patterns that we permit.

Issue-0084: Execution model

Date: Jan-17-2001

Description: Need prose describing execution model scenarios : interpretor vs. compile/runtime vs. translation into another query language. Explain relationship between static and dynamic semantics.

Issue-0085: Semantics of Wildcard type

Date: Jan-17-2001

Description: Cite: wildcard types cannot be implemented (Section 2.12: Expanded names, paragraph 11 http://www.w3.org/XML/Group/2000/12/xmlquery-algebra20001204.html; critical, core) If x!y means any name in x except names in y, what does x!y!z mean? In general, how do ! and | operate (precedence, associativity)? Parentheses are required to force the desired grouping of these two operators. Also, what does x!* mean? (There's an infinite family of such examples.)

Issue-0086: Syntactic rules

Date: Jan-17-2001

Description: Need rules for specifying syntactic correctness of query: symbol spaces; variable def'ns precede uses; list of keywords, etc.

Resolution Syntactic rules should be dealt with in XQuery document

Issue-0087: More examples of Joins

Date: Jan-17-2001

Description: Cite: no join operator; wants example of many-to-many joins, inner join, left and full outer joins.

Issue-0088: Align XQuery types with XML Schema : Formal Description.

Date: 02-Apr-2001

Description: Sources of misalignment: XQuery types include comment and processing instruction; [XML Schema : Formal Description] does not. XQuery uses () for empty sequence; MSL uses the epsilon character. XQuery permits the names of attribute and element components to be wildcard expressions. MSL only permits literal names for attributes and elements, but permits stand-alone wildcard expressions. XQuery types call '&' interleaved repetition, but MSL says it means 'all g1 and g2 in either order'. Does MSL mean interleaved repetition?

Issue-0089: Syntax for types in XQuery

Date: 30-Apr-2001

Description: Formalism document gives a particular syntax for type expressions that is not supported in the XQuery surface syntax.

Issue-0090: Static type-assertion expression

Date: 30-Apr-2001

Description: Formalism document uses a static type-assertion expression that is not supported in the XQuery surface syntax. See http://lists.w3.org/Archives/Member/w3c-query-editors/2001Apr/0021.html.

Issue-0091: Attribute expression

Date: 30-Apr-2001

Description: XQuery formal semantics has stand-alone attribute constructor/expression `ATTRIBUTE QName (Exp)` that is not supported in XQuery surface syntax.

Issue-0092: Error expression

Date: 11-May-2001

Description: XQuery formal semantics has an error expression `Error` that is not supported in XQuery surface syntax.

Issue-0093: Representation of Text Nodes in type system

Date: 11-May-2001

Description: The data model distinguished between text nodes and strings, which are simple-typed values. Text nodes have identity, parents, and siblings. Strings do not. Text nodes are accessed by the children() accessor; strings and other simple-typed values are accessed by the typed-value() accessor. The distinction between text nodes and simple-typed values should exist in type system as well. well.

Issue-0094: Static type errors and warnings

Date: 31-May-2001

Description: Static type errors and warnings are not specified. We need to enumerate in both the XQuery and formal semantics documents what kinds of static type errors and warnings are produced by the type system. See thread beginning: http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2001May/0405.html See also [**Issue-0090: Static type-assertion expression**].

Issue-0095: Importing Schemas and DTDs into query

Date: 31-May-2001

Description: We do not specify how a Schema or DTD is 'imported' into a query so that its information is available during type checking. Schema and DTDs can either be named explicitly (e.g., by an 'IMPORT SCHEMA' clause in a query) or implicitly, by accessing documents that refer to a Schema or DTD. The mechanism for statically accessing a Schema or DTD is unspecified.

Issue-0096: Support for schema-less and incompletely validated documents

Date: 31-May-2001

Description: This is related to [**Issue-0095: Importing Schemas and DTDs into query**]. We do not specify what is the effect of type checking a query that is applied to a document without a DTD or Schema. In general, a schema-less document has type xs:AnyType and type checking can proceed under that assumption. A related issue is what is the effect of type checking a query that is applied to an incompletely validated document. As above, we can make *no* assumptions about the static type of an incompletely validated document and must assume its static type is xs:AnyType.

See also: http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2001May/0408.html

Issue-0097: Static type-checking vs. Schema validation

Date: 31-May-2001

Description: Static type checking and schema validation are not equivalent, but we might want to do both in a query. For example, we might want to assert statically that an expression has a particular type and also validate dynamically the value of an expression w.r.t a particular schema.

The differences between static type checking and schema validation must be enumerated clearly (the XSFD people should help us with this).

Issue-0098: Implementation of and conformance levels for static type checking

Date: 31-May-2001

Description: This issue is related to [**Issue-0059: Testing Subtyping**] Static type checking may be difficult and/or expensive to implement. Some discussion of algorithmic issues of type checking are needed. In addition, we may want to define "conformance levels" for XQuery, in which some processors (or some processing modes) are more permissive about types. This would allow XQuery implementations that do not understand all of Schema, and it would allow customers some control over the cost/benefit tradeoff of type checking.

Issue-0099: Incomplete/inconsistent mapping from XQuery to core

Date: 06-June-2001

Description: This mapping is still preliminary and contains inconsistencies. These inconsistencies will be addressed in detail in the next draft of the document.

B.3 Alphabetic list of issues

B.3.1 Open Issues

Number: 34

[**Issue-0015: 3-valued logic to support NULLs**]

B.3.2 Resolved (or redundant) Issues

Number: 65

[Issue-0035: Exception handling]
[Issue-0048: Explicit Type Declarations]
[Issue-0009: Externally defined functions]
[Issue-0046: FOR Syntax]
[Issue-0032: Full regular path expressions]
[Issue-0028: Fusion]
[Issue-0034: Fusion]
[Issue-0003: Global Order]
[Issue-0045: Global Order]
[Issue-0036: Global-order based operators]
[Issue-0042: GroupBy]
[Issue-0012: GroupBy - needs second order functions?]
[Issue-0022: Indentation, Whitespaces]
[Issue-0077: Interleaved repetition and closure]
[Issue-0060: Internationalization aspects for strings]
[Issue-0044: Keys and IDREF]
[Issue-0081: Lexical representation of Schema simple types]
[Issue-0016: Mixed content]
[Issue-0061: Model for References]
[Issue-0057: More precise type system; choice in path]
[Issue-0002: Namespaces]
[Issue-0062: Open questions for constructing elements by reference]
[Issue-0074: Operational semantics for expressions]
[Issue-0056: Operators on Simple Types]
[Issue-0069: Organization of Document]
[Issue-0075: Overloading user defined functions]
[Issue-0026: Project - one tag only]
[Issue-0051: Project redundant?]
[Issue-0043: Recursive Descent for XPath]
[Issue-0050: Recursive Descent for XPath]
[Issue-0031: Recursive functions]
[Issue-0007: References: IDREFS, Keyrefs, Joins]
[Issue-0004: References vs containment]
[Issue-0067: Runtime Casts]
[Issue-0066: Shallow or Deep Equality?]
[Issue-0041: Sorting]
[Issue-0006: Source and join syntax instead of "for"]
[Issue-0070: Stable vs. Unstable Sort/Distinct]
[Issue-0086: Syntactic rules]
[Issue-0021: Syntax]
[Issue-0025: Treatment of empty results at type level]
[Issue-0082: Type and expression operator precedence]
[Issue-0049: Unordered Collections]
[Issue-0017: Unordered content]
[Issue-0076: Unordered types]
[Issue-0024: Value for empty sequences]
[Issue-0029: Views]

B.4 Delegated Issues

B.4.1 XPath 2.0

The following issues are delegated to XPath 2.0: [Issue-0007: References: IDREFS, Keyrefs, Joins], [Issue-0030: Automatic type coercion], [Issue-0032: Full regular path expressions], [Issue-0052: Axes of XPath].

B.4.2 XQuery

The following issues are delegated to XQuery: **[Issue-0009: Externally defined functions]**, **[Issue-0028: Fusion]**, **[Issue-0029: Views]**, **[Issue-0030: Automatic type coercion]**, **[Issue-0052: Axes of XPath]**, **[Issue-0081: Lexical representation of Schema simple types]**, **[Issue-0082: Type and expression operator precedence]**, **[Issue-0086: Syntactic rules]**.

B.4.3 Operators

The following issues are delegated to XPath 2.0: **[Issue-0013: Collations]**, **[Issue-0030: Automatic type coercion]**, **[Issue-0056: Operators on Simple Types]**, **[Issue-0060: Internationalization aspects for strings]**.