XQuery Tutorial

Peter Fankhauser, Fraunhofer IPSI

`Peter.Fankhauser@ipsi.fhg.de`

Philip Wadler, Avaya Labs

`wadler@avaya.com`

Acknowledgements

This tutorial is joint work with:

Mary Fernandez (AT&T)

Gerald Huck (IPSI/Infonyte)

Ingo Macherius (IPSI/Infonyte)

Thomas Tesch (IPSI/Infonyte)

Jerome Simeon (Lucent)

The W3C XML Query Working Group

Disclaimer: This tutorial touches on open issues of XQuery. Other members of the XML Query WG may disagree with our view.

Goals

After this tutorial, you should understand

- Part I XQuery expressions, types, and laws
- Part II XQuery laws and XQuery core
- Part III XQuery processing model
- Part IV XQuery type system and XML Schema
- Part V Type inference and type checking
- Part VI Where to go for more information

> “Where a mathematical reasoning can be had, it’s as great folly to make use of any other, as to grope for a thing in the dark, when you have a candle standing by you.”
>
> — Arbuthnot

Part I

XQuery by example

XQuery by example

Titles of all books published before 2000

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

Year and title of all books published before 2000

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

Books grouped by author

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

Part I.1

XQuery data model

Some XML data

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

Data model

XML

```
<BOOK YEAR="1999 2003">
  <AUTHOR>Abiteboul</AUTHOR>
  <AUTHOR>Buneman</AUTHOR>
  <AUTHOR>Suciu</AUTHOR>
  <TITLE>Data on the Web</TITLE>
  <REVIEW>A <EM>fine</EM> book.</REVIEW>
</BOOK>
```

XQuery

```
element BOOK {
  attribute YEAR { 1999, 2003 },
  element AUTHOR { "Abiteboul" },
  element AUTHOR { "Buneman" },
  element AUTHOR { "Suciu" },
  element TITLE { "Data on the Web" },
  element REVIEW { "A", element EM { "fine" }, "book." }
}
```

Part I.2

XQuery types

DTD (Document Type Definition)

```
<!ELEMENT BOOKS (BOOK*)>
<!ELEMENT BOOK (AUTHOR+, TITLE, REVIEW?)>
<!ATTLIST BOOK YEAR CDATA #OPTIONAL>
<!ELEMENT AUTHOR (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ENTITY % INLINE "( #PCDATA | EM | BOLD )*">
<!ELEMENT REVIEW %INLINE;>
<!ELEMENT EM %INLINE;>
<!ELEMENT BOLD %INLINE;>
```

Schema

```
<xsd:schema targetns="http://www.example.com/books"
            xmlns="http://www.example.com/books"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            attributeFormDefault="qualified"
            elementFormDefault="qualified">
  <xsd:element name="BOOKS">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="BOOK"
          minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

Schema, continued

```
<xsd:element name="BOOK">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="AUTHOR" type="xsd:string"
         minOccurs="1" maxOccurs="unbounded"/>
      <xsd:element name="TITLE" type="xsd:string"/>
      <xsd:element name="REVIEW" type="INLINE"
         minOccurs="0" maxOccurs="1"/>
    <xsd:sequence>
    <xsd:attribute name="YEAR" type="NONEMPTY-INTEGER-LIST"
      use="optional"/>
  </xsd:complexType>
</xsd:element>
```

Schema, continued^2

```
    <xsd:complexType name="INLINE" mixed="true">
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="EM" type="INLINE"/>
        <xsd:element name="BOLD" type="INLINE"/>
      </xsd:choice>
    </xsd:complexType>
    <xsd:simpleType name="INTEGER-LIST">
      <xsd:list itemType="xsd:integer"/>
    </xsd:simpleType>
    <xsd:simpleType name="NONEMPTY-INTEGER-LIST">
      <xsd:restriction base="INTEGER-LIST">
        <xsd:minLength value="1"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:schema>
```

XQuery types

```
define element BOOKS { BOOK* }
define element BOOK { @YEAR?, AUTHOR+, TITLE, REVIEW? }
define attribute YEAR { xsd:integer+ }
define element AUTHOR { xsd:string }
define element TITLE { xsd:string }
define type INLINE { ( xsd:string | EM | BOLD )* }
define element REVIEW { #INLINE }
define element EM { #INLINE }
define element BOLD { #INLINE }
```

Part I.3

XQuery and Schema

XQuery and Schema

Authors and title of books published before 2000

```
schema "http://www.example.com/books"
namespace default = "http://www.example.com/books"
validate
  <BOOKS>{
    for $book in /BOOKS/BOOK[@YEAR < 2000] return
      <BOOK>{ $book/AUTHOR, $book/TITLE }</BOOK>
  }</BOOKS>
```

$\in$

```
element BOOKS {
  element BOOK {
    element AUTHOR { xsd:string } +,
    element TITLE { xsd:string }
  } *
}
```

Another Schema

```
<xsd:schema targetns="http://www.example.com/answer"
            xmlns="http://www.example.com/answer"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            elementFormDefault="qualified">
  <xsd:element name="ANSWER">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="BOOK"
          minOccurs="0" maxOccurs="unbounded"/>
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="TITLE" type="xsd:string"/>
              <xsd:element name="AUTHOR" type="xsd:string"
                 minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
          </xsd:complexType>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Another XQuery type

```
element ANSWER { BOOK* }
element BOOK { TITLE, AUTHOR+ }
element AUTHOR { xsd:string }
element TITLE { xsd:string }
```

XQuery with multiple Schemas

Title and authors of books published before 2000

```
schema "http://www.example.com/books"
schema "http://www.example.com/answer"
namespace B = "http://www.example.com/books"
namespace A = "http://www.example.com/answer"
validate
  <A:ANSWER>{
    for $book in /B:BOOKS/B:BOOK[@YEAR < 2000] return
      <A:BOOK>{
        <A:TITLE>{ $book/B:TITLE/text() }</A:TITLE>,
        for $author in $book/B:AUTHOR return
          <A:AUTHOR>{ $author/text() }</A:AUTHOR>
      }<A:BOOK>
  }</A:ANSWER>
```

Part I.4

Projection

Projection

Return all authors of all books

```
/BOOKS/BOOK/AUTHOR
```

⇒

```
<AUTHOR>Abiteboul</AUTHOR>,
<AUTHOR>Buneman</AUTHOR>,
<AUTHOR>Suciu</AUTHOR>,
<AUTHOR>Buneman</AUTHOR>
```

∈

```
element AUTHOR { xsd:string } *
```

Laws — relating XPath to XQuery

Return all authors of all books

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

Laws — Associativity

Associativity in XPath

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

Associativity in XQuery

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

Part I.5

Selection

Selection

Return titles of all books published before 2000

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

$\Rightarrow$

```
<TITLE>Data on the Web</TITLE>
```

$\in$

```
element TITLE { xsd:string } *
```

Laws — relating XPath to XQuery

Return titles of all books published before 2000

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

=

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

Laws — mapping into XQuery core

Comparison defined by existential

```
  $book/@YEAR < 2000
```

=

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

Existential defined by iteration with selection

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

=

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

Selection defined by conditional

```
  for $year in $book/@YEAR where $year < 2000 returns $year
```

=

```
  for $year in $book/@YEAR returns
    if $year < 2000 then $year else ()
```

Laws — mapping into XQuery core

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

Selection — Type may be too broad

Return book with title "Data on the Web"

```
/BOOKS/BOOK[TITLE = "Data on the Web"]
```

$\Rightarrow$

```
<BOOK YEAR="1999 2003">
  <AUTHOR>Abiteboul</AUTHOR>
  <AUTHOR>Buneman</AUTHOR>
  <AUTHOR>Suciu</AUTHOR>
  <TITLE>Data on the Web</TITLE>
  <REVIEW>A <EM>fine</EM> book.</REVIEW>
</BOOK>
```

$\in$

```
 BOOK*
```

How do we exploit keys and relative keys?

Selection — Type may be narrowed

Return book with title "Data on the Web"

```
treat as element BOOK? (
  /BOOKS/BOOK[TITLE = "Data on the Web"]
)
```

$\in$

```
BOOK?
```

Can exploit static type to reduce dynamic checking
Here, only need to check length of book sequence, not type

Iteration — Type may be too broad

Return all Amazon and Fatbrain books by Buneman

```
define element AMAZON-BOOK { TITLE, AUTHOR+ }
define element FATBRAIN-BOOK { AUTHOR+, TITLE }
define element BOOKS { AMAZON-BOOK*, FATBRAIN-BOOK* }

for $book in (/BOOKS/AMAZON-BOOK, /BOOKS/FATBRAIN-BOOK)
where $book/AUTHOR = "Buneman" return
  $book
```

$\in$

```
( AMAZON-BOOK | FATBRAIN-BOOK )*
```

$\not\subseteq$

```
AMAZON-BOOK*, FATBRAIN-BOOK*
```

How best to trade off simplicity vs. accuracy?

Part I.6

Construction

Construction in XQuery

Return year and title of all books published before 2000

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

⇒

```
<BOOK YEAR="1999 2003">
  <TITLE>Data on the Web</TITLE>
</BOOK>
```

∈

```
element BOOK {
  attribute YEAR { integer+ },
  element TITLE { string }
} *
```

Construction — mapping into XQuery core

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

=

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

Part I.7

Grouping

Grouping

Return titles for each author

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

⇒

```
<AUTHOR NAME="Abiteboul">
  <TITLE>Data on the Web</TITLE>
</AUTHOR>,
<AUTHOR NAME="Buneman">
  <TITLE>Data on the Web</TITLE>
  <TITLE>XML in Scotland</TITLE>
</AUTHOR>,
<AUTHOR NAME="Suciu">
  <TITLE>Data on the Web</TITLE>
</AUTHOR>
```

Grouping — Type may be too broad

Return titles for each author

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

$\in$

```
element AUTHOR {
  attribute NAME { string },
  element TITLE { string } *
}
```

$\not\subseteq$

```
element AUTHOR {
  attribute NAME { string },
  element TITLE { string } +
}
```

Grouping — Type may be narrowed

Return titles for each author

```
define element TITLE { string }

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

∈

```
element AUTHOR {
  attribute NAME { string },
  element TITLE { string } +
}
```

Part I.8

Join

Join

Books that cost more at Amazon than at Fatbrain

```
define element BOOKS { BOOK* }
define element BOOK { TITLE, PRICE, ISBN }

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

Join — Unordered

Books that cost more at Amazon than at Fatbrain, in any order

```
unordered(
  for $am in $amazon/BOOKS/BOOK,
      $fat in $fatbrain/BOOKS/BOOK
  where $am/ISBN = $fat/ISBN
    and $am/PRICE > $fat/PRICE
  return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
)
```

Reordering required for cost-effective computation of joins

Join — Sorted

```
for $am in $amazon/BOOKS/BOOK,
    $fat in $fatbrain/BOOKS/BOOK
where $am/ISBN = $fat/ISBN
 and $am/PRICE > $fat/PRICE
return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
sortby TITLE
```

Join — Laws

```
for $am in $amazon/BOOKS/BOOK,
    $fat in $fatbrain/BOOKS/BOOK
where $am/ISBN = $fat/ISBN
  and $am/PRICE > $fat/PRICE
return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
sortby TITLE
```

=

```
unordered(
  for $am in $amazon/BOOKS/BOOK,
      $fat in $fatbrain/BOOKS/BOOK
  where $am/ISBN = $fat/ISBN
    and $am/PRICE > $fat/PRICE
  return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
) sortby TITLE
```

Join — Laws

```
unordered(
  for $am in $amazon/BOOKS/BOOK,
      $fat in $fatbrain/BOOKS/BOOK
  where $am/ISBN = $fat/ISBN
    and $am/PRICE > $fat/PRICE
  return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
) sortby TITLE
```

=

```
unordered(
  for $am in unordered($amazon/BOOKS/BOOK),
      $fat in unordered($fatbrain/BOOKS/BOOK)
  where $am/ISBN = $fat/ISBN
    and $am/PRICE > $fat/PRICE
  return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
) sortby TITLE
```

Left outer join

Books at Amazon and Fatbrain with both prices,
and all other books at Amazon with price

```
for $am in $amazon/BOOKS/BOOK, $fat in $fatbrain/BOOKS/BOOK
where $am/ISBN = $fat/ISBN
return <BOOK>{ $am/TITLE, $am/PRICE, $fat/PRICE }</BOOK>
,
for $am in $amazon/BOOKS/BOOK
where not($am/ISBN = $fatbrain/BOOKS/BOOK/ISBN)
return <BOOK>{ $am/TITLE, $am/PRICE }</BOOK>
```

$\in$

```
element BOOK { TITLE, PRICE, PRICE } *
,
element BOOK { TITLE, PRICE } *
```

Why type closure is important

Closure problems for Schema

- Deterministic content model
- Consistent element restriction

```
element BOOK { TITLE, PRICE, PRICE } *
,
element BOOK { TITLE, PRICE } *
```

$\subseteq$

```
element BOOK { TITLE, PRICE+ } *
```

The first type is *not* a legal Schema type
The second type *is* a legal Schema type
Both are legal XQuery types

Part I.9

Nulls and three-valued logic

Books with price and optional shipping price

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

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

Approaches to missing data

Books costing $50.00, where default shipping is $5.00

```
for $book in /BOOKS/BOOK
where $book/PRICE + if_absent($book/SHIPPING, 5.00) = 50.00
return $book/TITLE
```

⇒

```
<TITLE>Data on the Web</TITLE>,
<TITLE>XML in Scotland</TITLE>
```

Books costing $50.00, where missing shipping is unknown

```
for $book in /BOOKS/BOOK
where $book/PRICE + $book/SHIPPING = 50.00
return $book/TITLE
```

⇒

```
<TITLE>Data on the Web</TITLE>
```

Arithmetic, Truth tables

+	()	0	1
()	()	()	()
0	()	0	1
1	()	1	2

*	()	0	1
()	()	()	()
0	()	0	0
1	()	0	1

OR3	()	false	true
()	()	()	true
false	()	false	true
true	true	true	true

AND3	()	false	true
()	()	false	()
false	false	false	false
true	()	false	true

NOT3	
()	()
false	true
true	false

Part I.10

Type errors

Type error 1: Missing or misspelled element

Return TITLE and ISBN of each book

```
define element BOOKS { BOOK* }
define element BOOK { TITLE, PRICE }
define element TITLE { xsd:string }
define element PRICE { xsd:decimal }

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

$\in$

```
element ANSWER { TITLE } *
```

Finding an error by omission

Return title and ISBN of each book

```
define element BOOKS { BOOK* }
define element BOOK { TITLE, PRICE }
define element TITLE { xsd:string }
define element PRICE { xsd:decimal }

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

Report an error any sub-expression of type `()`, other than the expression `()` itself

Finding an error by assertion

Return title and ISBN of each book

```
define element BOOKS { BOOK* }
define element BOOK { TITLE, PRICE }
define element TITLE { xsd:string }
define element PRICE { xsd:decimal }
define element ANSWER { TITLE, ISBN }
define element ISBN { xsd:string }

for $book in /BOOKS/BOOK return
  assert as element ANSWER (
    <ANSWER>{ $book/TITLE, $book/ISBN }</ANSWER>
  )
```

Assertions might be added automatically, e.g. when there is a global element declaration and no conflicting local declarations

Type Error 2: Improper type

```
define element BOOKS { BOOK* }
define element BOOK { TITLE, PRICE, SHIPPING, SHIPCOST? }
define element TITLE { xsd:string }
define element PRICE { xsd:decimal }
define element SHIPPING { xsd:boolean }
define element SHIPCOST { xsd:decimal }

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

Type error: `decimal + boolean`

Type Error 3: Unhandled null

```
define element BOOKS { BOOK* }
define element BOOK { TITLE, PRICE, SHIPPING? }
define element TITLE { xsd:string }
define element PRICE { xsd:decimal }
define element SHIPPING { xsd:decimal }
define element ANSWER { TITLE, TOTAL }
define element TOTAL { xsd:decimal }

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

Type error: $\mathtt{xsd:decimal?} \not\subseteq \mathtt{xsd:decimal}$

Part I.11

Functions

Functions

Simplify book by dropping optional year

```
define element BOOK { @YEAR?, AUTHOR, TITLE }
define attribute YEAR { xsd:integer }
define element AUTHOR { xsd:string }
define element TITLE { xsd:string }
define function simple (element BOOK $b) returns element BOOK {
  <BOOK> $b/AUTHOR, $b/TITLE </BOOK>
}
```

Compute total cost of book

```
define element BOOK { TITLE, PRICE, SHIPPING? }
define element TITLE { xsd:string }
define element PRICE { xsd:decimal }
define element SHIPPING { xsd:decimal }
define function cost (element BOOK $b) returns xsd:integer? {
  $b/PRICE + $b/SHIPPING
}
```

Part I.12

Recursion

A part hierarchy

```
define type PART { COMPLEX | SIMPLE }
define type COST { @ASSEMBLE | @TOTAL }
define element COMPLEX { @NAME & #COST, #PART* }
define element SIMPLE { @NAME & @TOTAL }
define attribute NAME { xsd:string }
define attribute ASSEMBLE { xsd:decimal }
define attribute TOTAL { xsd:decimal }

<COMPLEX NAME="system" ASSEMBLE="500.00">
  <SIMPLE NAME="monitor" TOTAL="1000.00"/>
  <SIMPLE NAME="keyboard" TOTAL="500.00"/>
  <COMPLEX NAME="pc" ASSEMBLE="500.00">
    <SIMPLE NAME="processor" TOTAL="2000.00"/>
    <SIMPLE NAME="dvd" TOTAL="1000.00"/>
  </COMPLEX>
</COMPLEX>
```

A recursive function

```
define function total (#PART $part) returns #PART {
  if ($part instance of SIMPLE) then $part else
    let $parts := $part/(COMPLEX | SIMPLE)/total(.)
    return
      <COMPLEX NAME="$part/@NAME" TOTAL="
          $part/@ASSEMBLE + sum($parts/@TOTAL)">{
        $parts
      }</COMPLEX>
}
```

⇒

```
<COMPLEX NAME="system" TOTAL="5000.00">
  <SIMPLE NAME="monitor" TOTAL="1000.00"/>
  <SIMPLE NAME="keyboard" TOTAL="500.00"/>
  <COMPLEX NAME="pc" TOTAL="3500.00">
    <SIMPLE NAME="processor" TOTAL="2000.00"/>
    <SIMPLE NAME="dvd" TOTAL="1000.00"/>
  </COMPLEX>
</COMPLEX>
```

Part I.13

Wildcard types

Wildcards types and computed names

Turn all attributes into elements, and vice versa

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

swizzle(<TEST A="a" B="b">
         <C>c</C>
         <D>d</D>
        </TEST>)
```

⇒

```
<TEST C="c" D="D">
  <A>a</A>
  <B>b</B>
</TEST>
```

∈

```
element
```

Part I.14

Syntax

Templates

Convert book listings to HTML format

```
<HTML><H1>My favorite books</H1>
  <UL>{
    for $book in /BOOKS/BOOK return
      <LI>
        <EM>{ data($book/TITLE) }</EM>,
        { data($book/@YEAR)[position()=last()] }.
      </LI>
  }</UL>
</HTML>
```

⇒

```
<HTML><H1>My favorite books</H1>
  <UL>
    <LI><EM>Data on the Web</EM>, 2003.</LI>
    <LI><EM>XML in Scotland</EM>, 2002.</LI>
  </UL>
</HTML>
```

XQueryX

A query in XQuery:

```
for $b in document("bib.xml")//book
where $b/publisher = "Morgan Kaufmann" and $b/year = "1998"
return $b/title
```

The same query in XQueryX:

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
 <q:flwr>
   <q:forAssignment variable="$b">
     <q:step axis="SLASHSLASH">
       <q:function name="document">
         <q:constant datatype="CHARSTRING">bib.xml</q:constant>
       </q:function>
       <q:identifier>book</q:identifier>
     </q:step>
   </q:forAssignment>
```

XQueryX, continued

```
<q:where>
  <q:function name="AND">
    <q:function name="EQUALS">
      <q:step axis="CHILD">
        <q:variable>$b</q:variable>
        <q:identifier>publisher</q:identifier>
      </q:step>
      <q:constant datatype="CHARSTRING">Morgan Kaufmann</q:consta
    </q:function>
    <q:function name="EQUALS">
      <q:step axis="CHILD">
        <q:variable>$b</q:variable>
        <q:identifier>year</q:identifier>
      </q:step>
      <q:constant datatype="CHARSTRING">1998</q:constant>
    </q:function>
  </q:function>
</q:where>
```

XQueryX, continued^2

```
    <q:return>
      <q:step axis="CHILD">
        <q:variable>$b</q:variable>
        <q:identifier>title</q:identifier>
      </q:step>
    </q:return>
  </q:flwr>
</q:query>
```

Part II

XQuery laws and XQuery core

“I never come across one of Laplace’s ‘Thus it plainly appears’ without feeling sure that I have hours of hard work in front of me.”

— Bowditch

Part II.1

XPath and XQuery

XPath and XQuery

Converting XPath into XQuery core

e**/**a

$=$

sidoaed(for $dot in e **return $dot/**$a$**)**

sidoaed $=$ sort in document order and eliminate duplicates

Why sidoaed is needed

```
<WARNING>
  <P>
    Do <EM>not</EM> press button,
    computer will <EM>explode!</EM>
  </P>
</WARNING>
```

Select all nodes inside warning

```
/WARNING//*
```

⇒

```
<P>
  Do <EM>not</EM> press button,
  computer will <EM>explode!</EM>
</P>,
<EM>not</EM>,
<EM>explode!</EM>
```

Why sidoaed is needed, continued

Select text in all emphasis nodes (list order)

```
for $x in /WARNING//* return $x/text()
```

⇒

```
"Do ",
" press button, computer will ",
"not",
"explode!"
```

Select text in all emphasis nodes (document order)

```
/WARNING//*/text()
```

=

```
sidoaed(for $x in /WARNING//* return $x/text())
```

⇒

```
"Do ",
"not",
" press button, computer will ",
"explode!"
```

Part II.2

Laws

Some laws

$$\begin{array}{ll} & \texttt{for \$}v\texttt{ in () return }e \\ = & \texttt{(empty)} \\ & \texttt{()} \end{array}$$

$$\begin{array}{ll} & \texttt{for \$}v\texttt{ in (}e_1\texttt{ , }e_2\texttt{) return }e_3 \\ = & \texttt{(sequence)} \\ & \texttt{(for \$}v\texttt{ in }e_1\texttt{ return }e_3\texttt{) , (for \$}v\texttt{ in }e_2\texttt{ return }e_3\texttt{)} \end{array}$$

$$\begin{array}{ll} & \texttt{data(element }a\texttt{ \{ }d\texttt{ \})} \\ = & \texttt{(data)} \\ & d \end{array}$$

More laws

```
  for $v in e return $v
= (left unit)
  e

  for $v in e1 return e2
= (right unit), if e1 is a singleton
  let $v := e1 return e2

  for $v1 in e1 return (for $v2 in e2 return e3)
= (associative)
  for $v2 in (for $v1 in e1 return e2) return e3
```

Using the laws — evaluation

```
  for $x in (<A>1</A>,<A>2</A>) return <B>{data($x)}</B>
=   (sequence)
  for $x in <A>1</A> return <B>{data($x)}</B> ,
  for $x in <A>2</A> return <B>{data($x)}</B>
=   (right unit)
  let $x := <A>1</A> return <B>{data($x)}</B> ,
  let $x := <A>2</A> return <B>{data($x)}</B>
=   (let)
  <B>{data(<A>1</A>)}</B> ,
  <B>{data(<A>2</A>)}</B>
=   (data)
  <B>1</B>,<B>2</B>
```

Using the laws — loop fusion

```
  let $b := for $x in $a return <B>{ data($x) }</B>
  return for $y in $b return <C>{ data($y) }</C>
=   (let)
  for $y in (
    for $x in $a return <B>{ data($x) }</B>
  ) return <C>{ data($y) }</C>
=   (associative)
  for $x in $a return
    (for $y in <B>{ data($x) }</B> return <C>{ data($y) }</C>)
=   (right unit)
  for $x in $a return <C>{ data(<B>{ data($x) }</B>) }</C>
=   (data)
  for $x in $a return <C>{ data($x) }</C>
```

Part II.3

XQuery core

An example in XQuery

Join books and review by title

```
for $b in /BOOKS/BOOK, $r in /REVIEWS/BOOK
where $b/TITLE = $r/TITLE
return
   <BOOK>{
     $b/TITLE,
     $b/AUTHOR,
     $r/REVIEW
   }</BOOK>
```

The same example in XQuery core

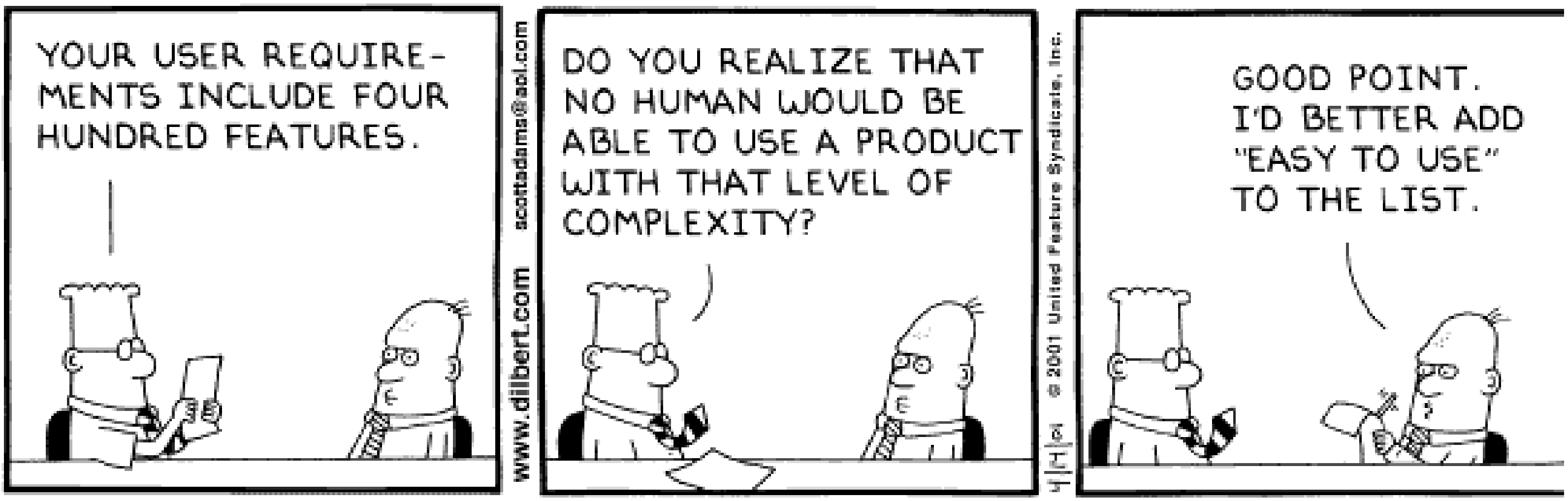
```
for $b in (
  for $dot in $root return
    for $dot in $dot/child::BOOKS return $dot/child::BOOK
) return
  for $r in (
    for $dot in $root return
      for $dot in $dot/child::REVIEWS return $dot/child::BOOK
  ) return
    if (
      not(empty(
        for $v1 in (
          for $dot in $b return $dot/child::TITLE
        ) return
          for $v2 in (
            for $dot in $r return $dot/child::TITLE
          ) return
            if (eq($v1,$v2)) then $v1 else ()
      ))
    ) then (
      element BOOK {
        for $dot in $b return $dot/child::TITLE ,
        for $dot in $b return $dot/child::AUTHOR ,
        for $dot in $r return $dot/child::REVIEW
      }
    )
    else ()
```

XQuery core: a syntactic subset of XQuery

- only one variable per iteration by for
- no where clause
- only simple path expressions iteratorVariable/Axis::NodeTest
- only simple element and attribute constructors
- sort by
- function calls

The 4 C's of XQuery core

- Closure:
  input: XML node sequence
  output: XML node sequence
- Compositionality:
  expressions composed of expressions
  no side-effects
- Correctness:
  dynamic semantics (query evaluation time)
  static semantics (query compilation time)
- Completeness:
  XQuery surface syntax can be expressed completely
  relationally complete (at least)


YOUR USER REQUIRE-MENTS INCLUDE FOUR HUNDRED FEATURES.
scottadams@aol.com
www.dilbert.com
DO YOU REALIZE THAT NO HUMAN WOULD BE ABLE TO USE A PRODUCT WITH THAT LEVEL OF COMPLEXITY?
© 2001 United Feature Syndicate, Inc.
4/14/01
GOOD POINT. I'D BETTER ADD "EASY TO USE" TO THE LIST.

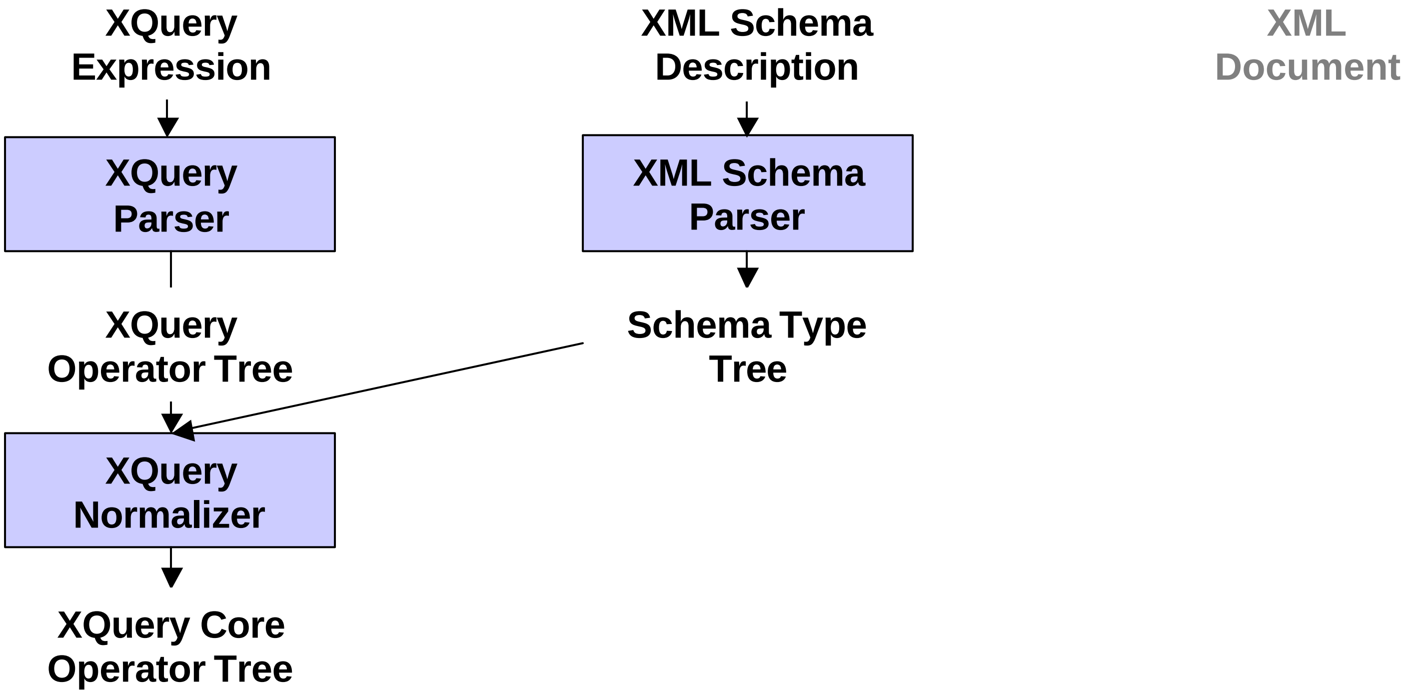
“Besides it is an error to believe that rigor in the proof is the enemy of simplicity. On the contrary we find it confirmed by numerous examples that the rigorous method is at the same time the simpler and the more easily comprehended. The very effort for rigor forces us to find out simpler methods of proof.”

— Hilbert

Part III

XQuery Processing Model

Analysis Step 1: Map to XQuery Core



XQuery Expression

XML Schema Description

XML Document

XQuery Parser

XML Schema Parser

XQuery Operator Tree

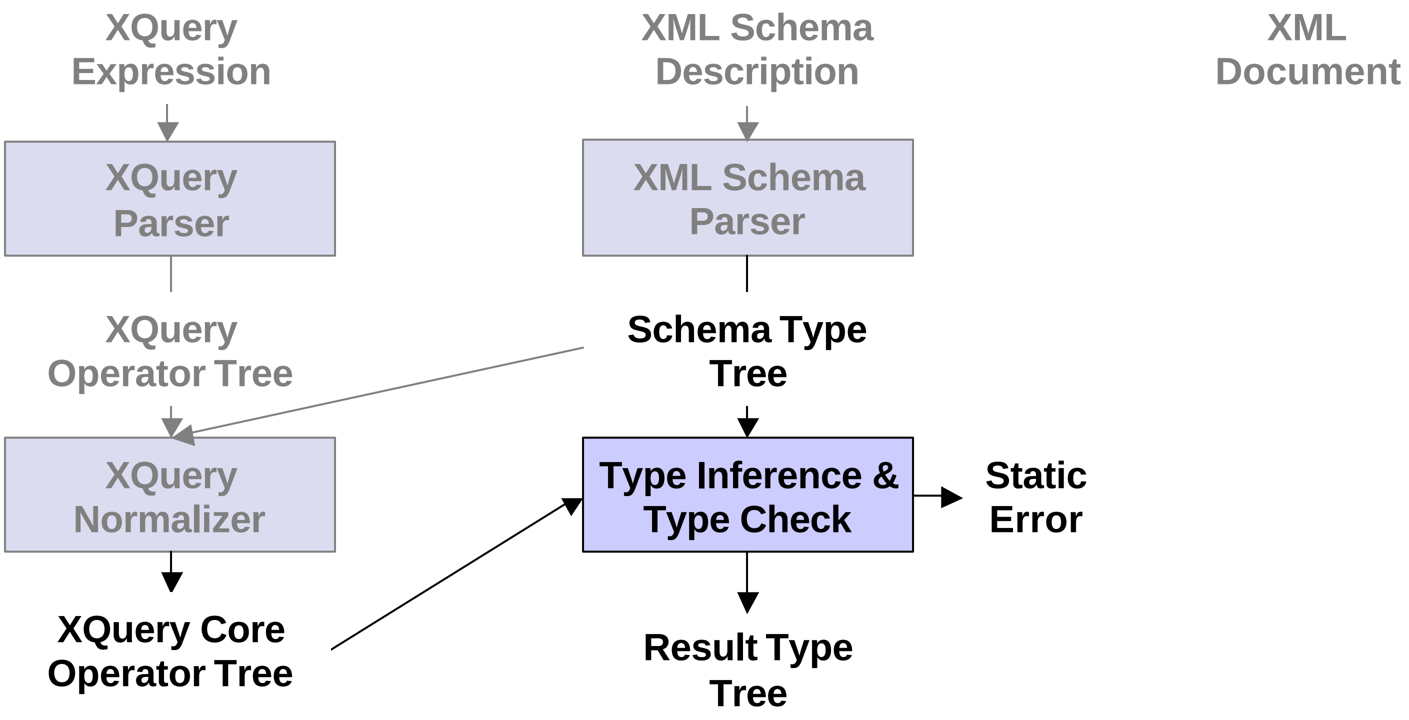
Schema Type Tree

XQuery Normalizer

XQuery Core Operator Tree

🟪 **Query Analysis Step 1: Mapping to XQuery Core**

Analysis Step 2: Infer and Check Type



XQuery Expression

XQuery Parser

XQuery Operator Tree

XQuery Normalizer

XQuery Core Operator Tree

XML Schema Description

XML Schema Parser

Schema Type Tree

Type Inference & Type Check

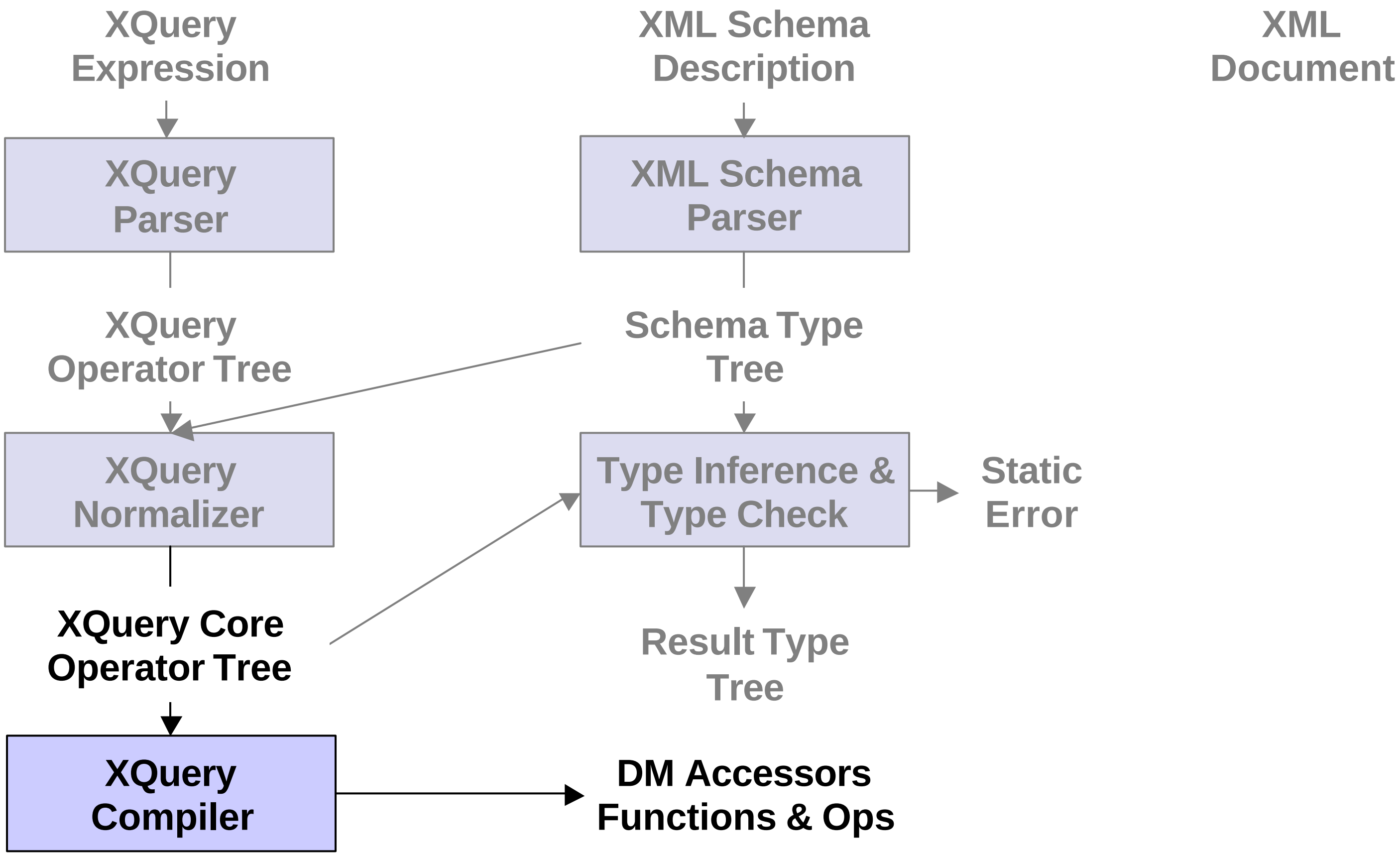
Static Error

Result Type Tree

XML Document

🟪 **Query Analysis Step 2: Type Inference & Check**

Analysis Step 3: Generate DM Accessors



XQuery Expression

XQuery Parser

XQuery Operator Tree

XQuery Normalizer

XQuery Core Operator Tree

XQuery Compiler

XML Schema Description

XML Schema Parser

Schema Type Tree

Type Inference & Type Check

Static Error

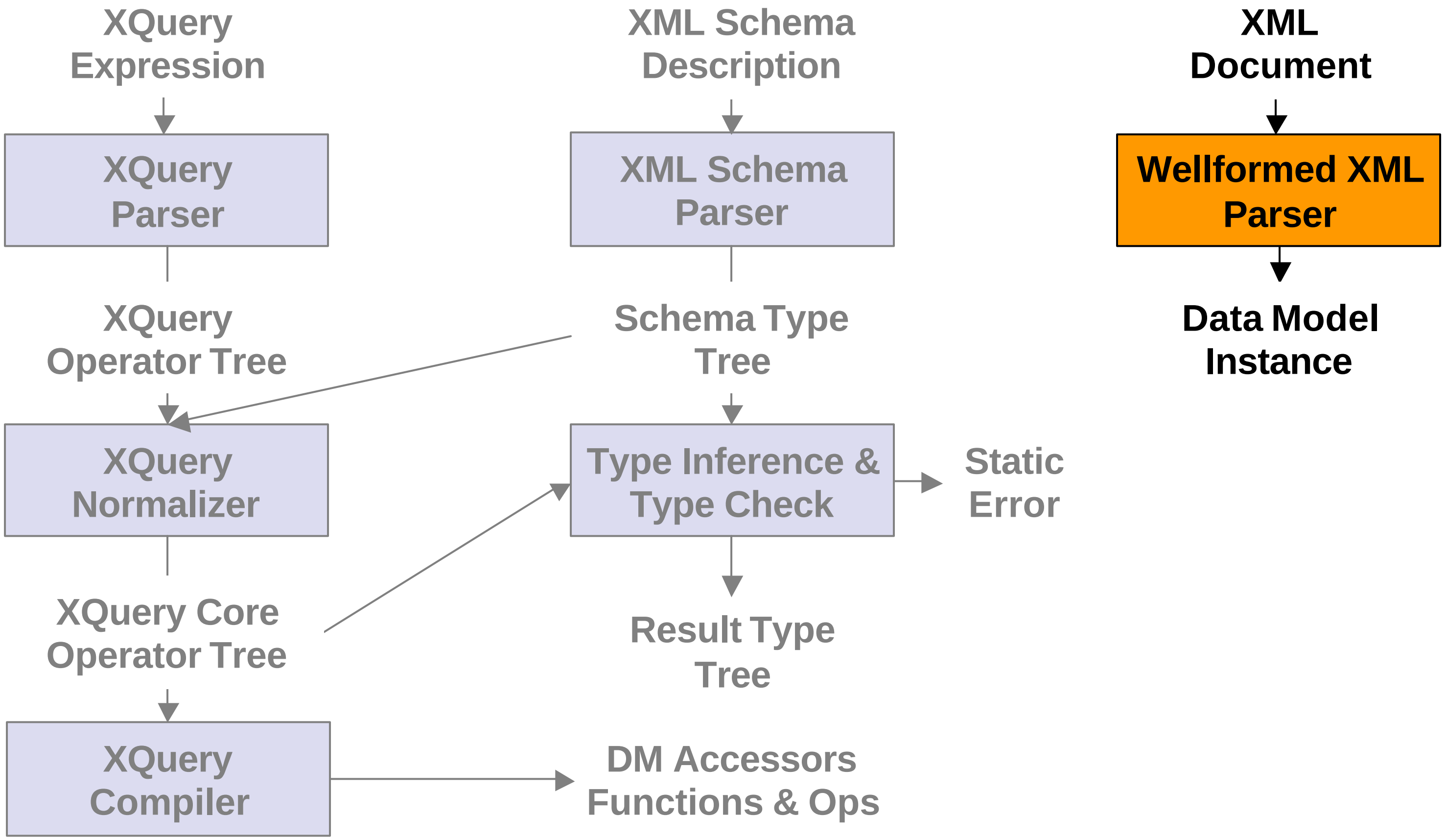
Result Type Tree

DM Accessors Functions & Ops

XML Document

▨ **Query Analysis Step 3: XQuery Compilation**

Eval Step 1: Generate DM Instance



XQuery Expression

XQuery Parser

XQuery Operator Tree

XQuery Normalizer

XQuery Core Operator Tree

XQuery Compiler

XML Schema Description

XML Schema Parser

Schema Type Tree

Type Inference & Type Check

Static Error

Result Type Tree

DM Accessors Functions & Ops

XML Document

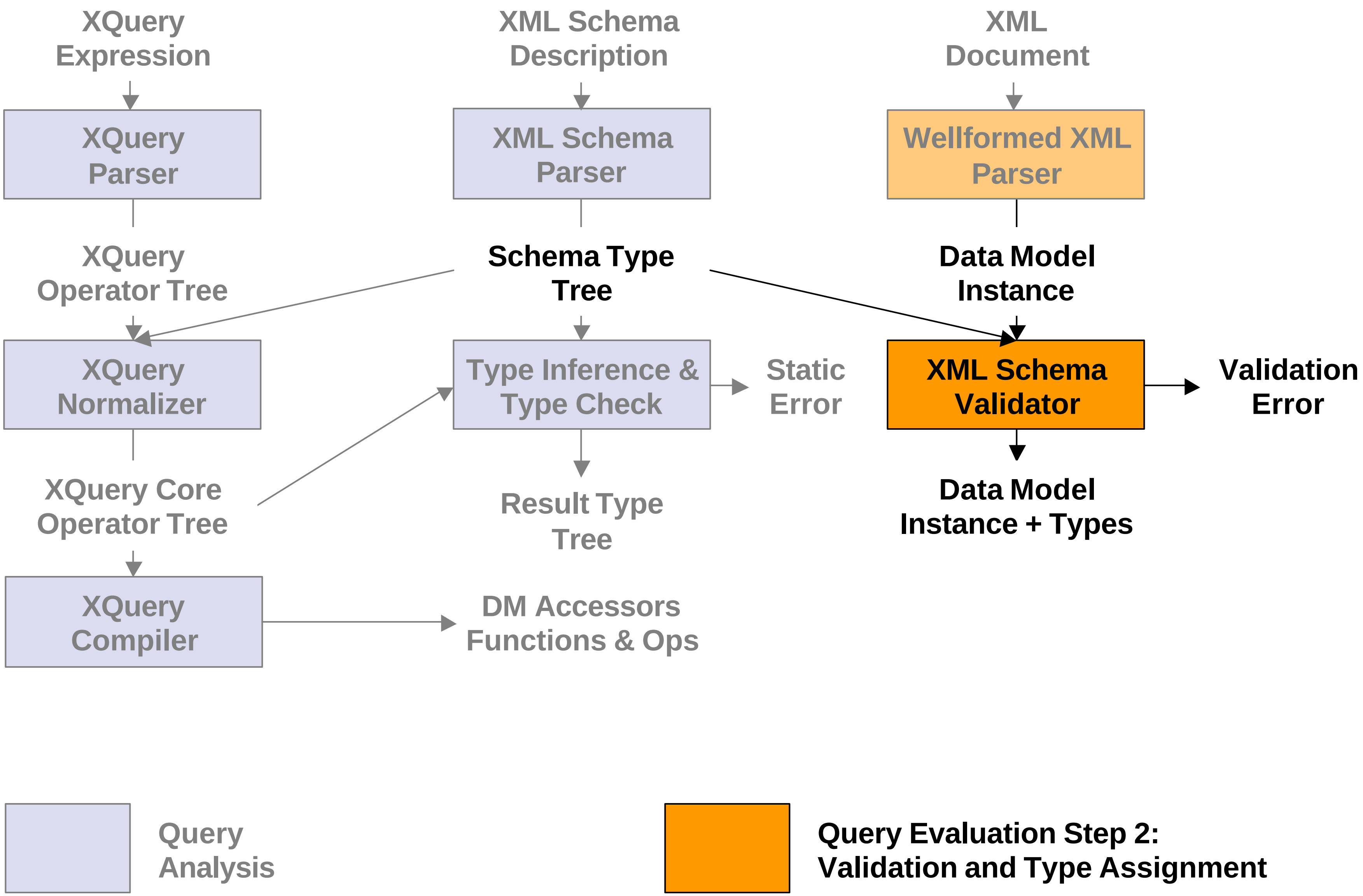
Wellformed XML Parser

Data Model Instance

▨ Query Analysis

🟧 **Query Evaluation Step 1: Instantiating the Data Model**

Eval Step 2: Validate and Assign Types



XQuery Expression

XQuery Parser

XQuery Operator Tree

XQuery Normalizer

XQuery Core Operator Tree

XQuery Compiler

DM Accessors Functions & Ops

XML Schema Description

XML Schema Parser

Schema Type Tree

Type Inference & Type Check

Static Error

Result Type Tree

XML Document

Wellformed XML Parser

Data Model Instance

XML Schema Validator

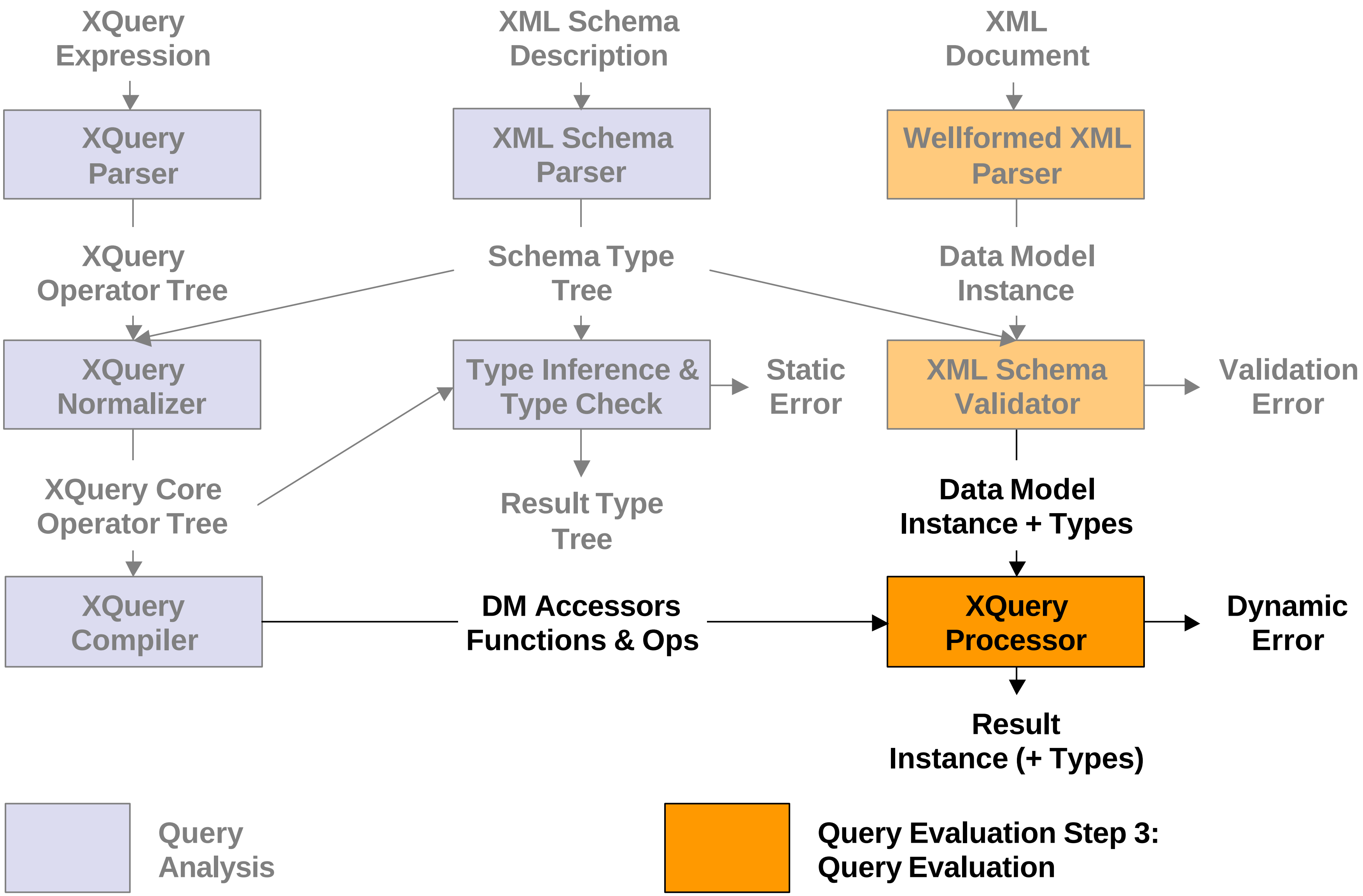
Validation Error

Data Model Instance + Types

🟪 Query Analysis

🟧 **Query Evaluation Step 2: Validation and Type Assignment**

Eval Step 3: Query Evaluation



XQuery Expression

XQuery Parser

XQuery Operator Tree

XQuery Normalizer

XQuery Core Operator Tree

XQuery Compiler

XML Schema Description

XML Schema Parser

Schema Type Tree

Type Inference & Type Check

Static Error

Result Type Tree

DM Accessors Functions & Ops

XML Document

Wellformed XML Parser

Data Model Instance

XML Schema Validator

Validation Error

Data Model Instance + Types

XQuery Processor

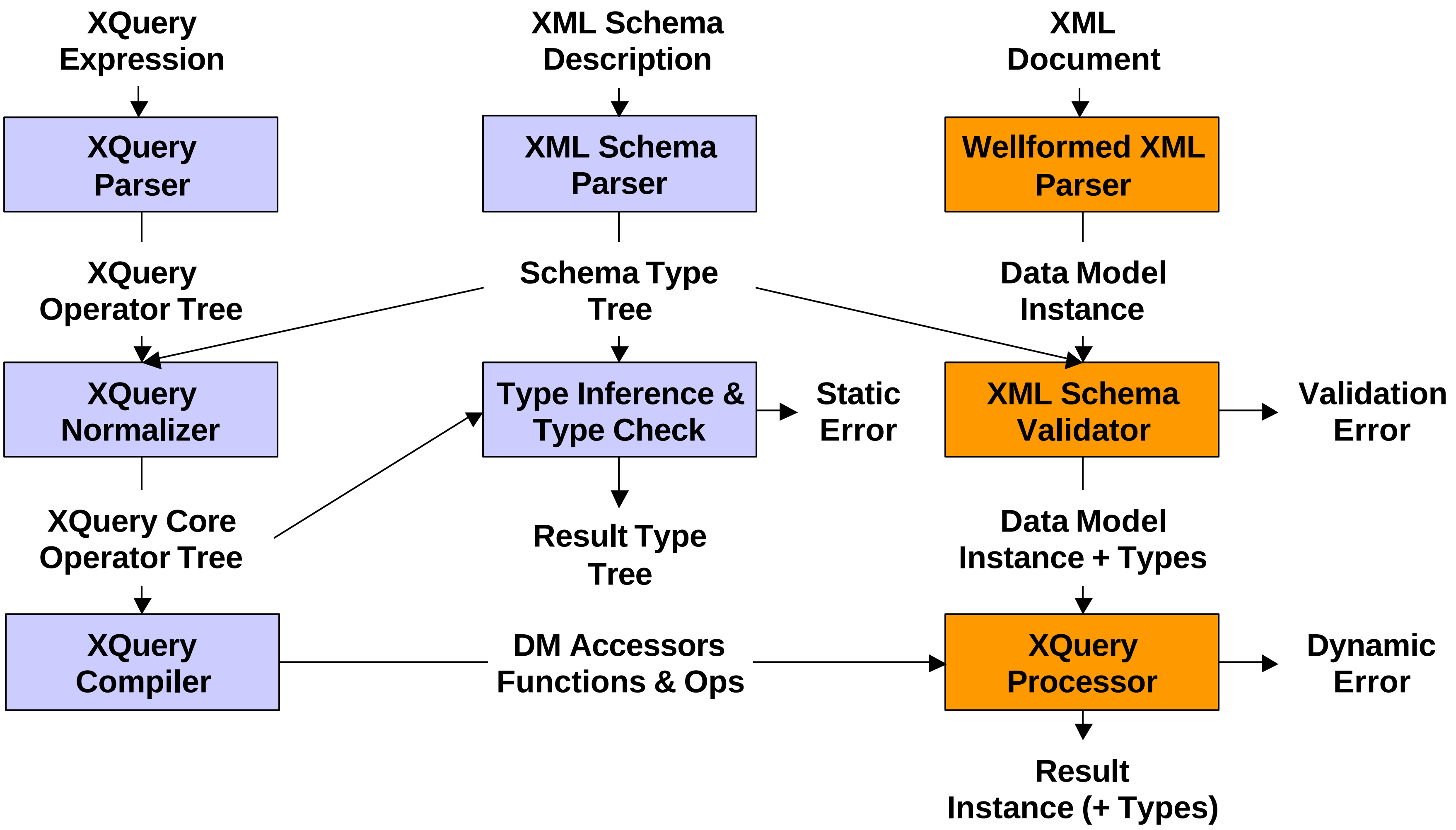
Dynamic Error

Result Instance (+ Types)

Query Analysis

Query Evaluation Step 3: Query Evaluation

XQuery Processing Model



XQuery Expression

XQuery Parser

XQuery Operator Tree

XQuery Normalizer

XQuery Core Operator Tree

XQuery Compiler

XML Schema Description

XML Schema Parser

Schema Type Tree

Type Inference & Type Check

Static Error

Result Type Tree

DM Accessors Functions & Ops

XML Document

Wellformed XML Parser

Data Model Instance

XML Schema Validator

Validation Error

Data Model Instance + Types

XQuery Processor

Dynamic Error

Result Instance (+ Types)

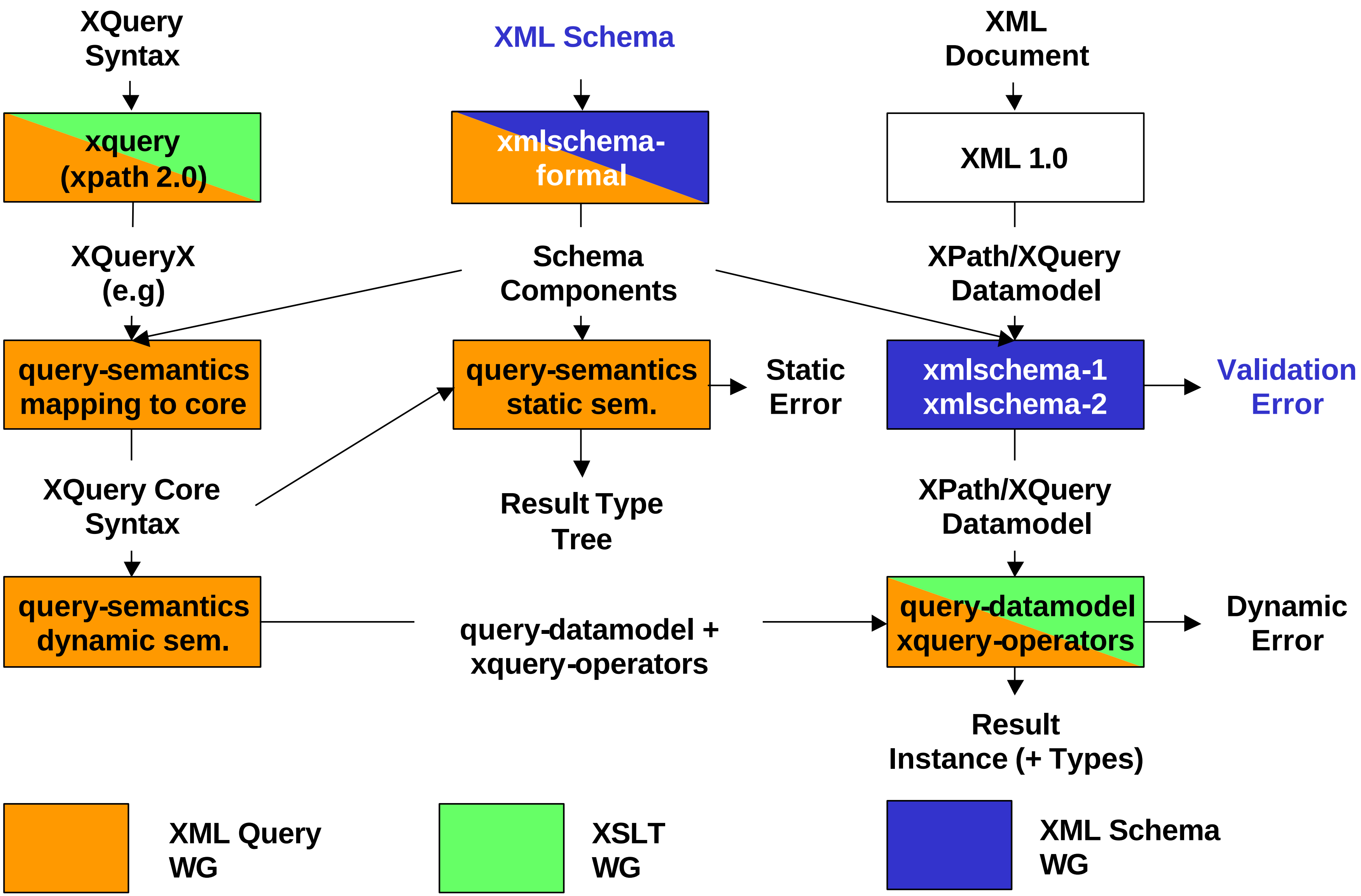
🟪 Query Analysis

🟧 Query Evaluation

XQuery Processing Model: Idealizations

- Query normalization and compilation:
  static type information is useful for logical optimization.
  a real implementation translates to and optimizes further on the basis of a physical algebra.
- Loading and validating XML documents:
  a real implementation can operate on typed datamodel instances directly.
- Representing data model instances:
  a real implementation is free to choose native, relational, or object-oriented representation.

XQuery et al. Specifications



XQuery Syntax

XML Schema

XML Document

xquery (xpath 2.0)

xmlschema-formal

XML 1.0

XQueryX (e.g)

Schema Components

XPath/XQuery Datamodel

query-semantics mapping to core

query-semantics static sem.

Static Error

xmlschema-1 xmlschema-2

Validation Error

XQuery Core Syntax

Result Type Tree

XPath/XQuery Datamodel

query-semantics dynamic sem.

query-datamodel + xquery-operators

query-datamodel xquery-operators

Dynamic Error

Result Instance (+ Types)

🟧 XML Query WG

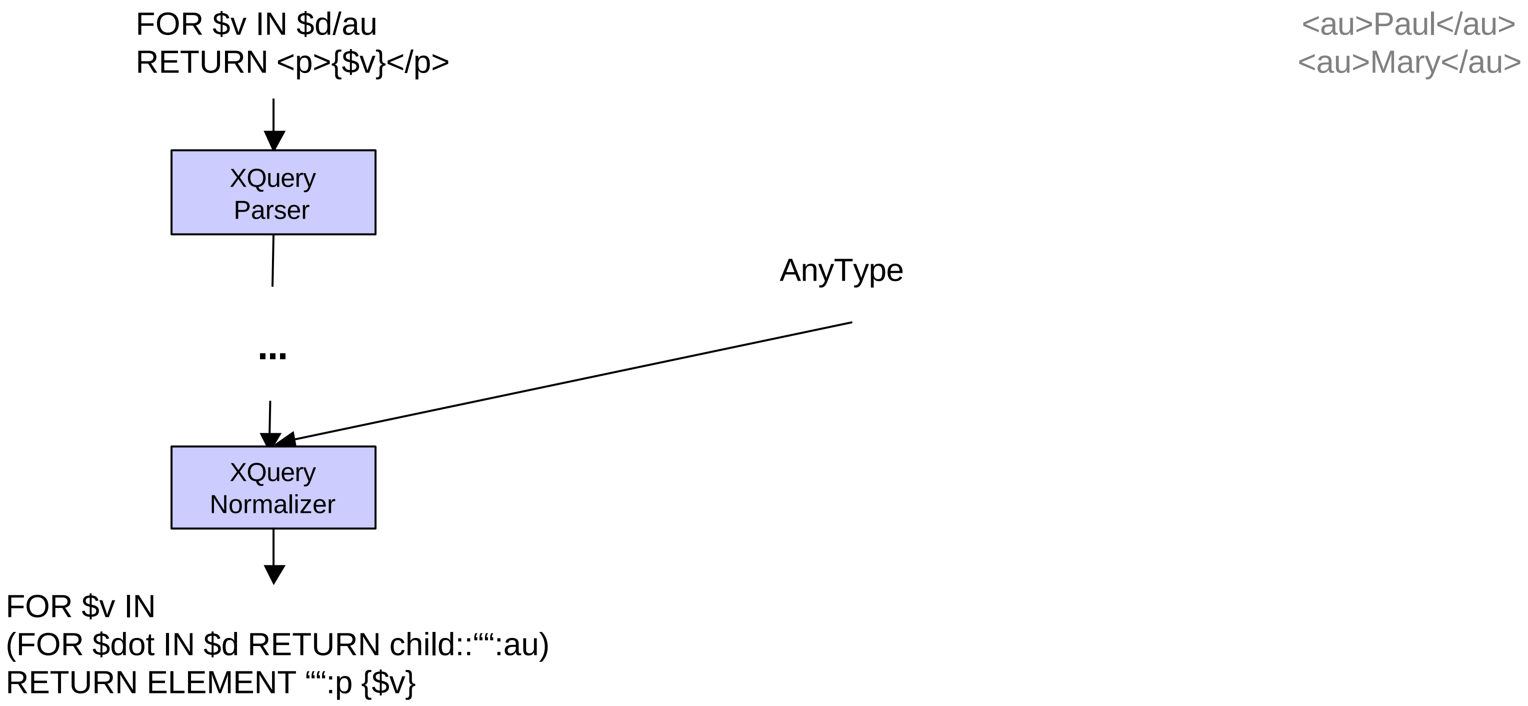
🟩 XSLT WG

🟦 XML Schema WG

XQuery et al. Specifications: Legend

- XQuery 1.0: An XML Query Language (WD)
  `http://www.w3.org/TR/xquery/`
- XML Syntax for XQuery 1.0 (WD)
  `http://www.w3.org/TR/xqueryx/`
- XQuery 1.0 Formal Semantics (WD)
  `http://www.w3.org/TR/query-semantics/`
  xquery core syntax, mapping to core,
  static semantics, dynamic semantics
- XQuery 1.0 and XPath 2.0 Data Model (WD)
  `http://www.w3.org/TR/query-datamodel/`
  node-constructors, value-constructors, accessors
- XQuery 1.0 and XPath 2.0 Functions and Operators (WD)
  `http://www.w3.org/TR/xquery-operators/`
- XML Schema: Formal Description (WD)
  `http://www.w3.org/TR/xmlschema-formal/`
- XML Schema Parts (1,2) (Recs)
  `http://www.w3.org/TR/xmlschema-1/`
  `http://www.w3.org/TR/xmlschema-2/`

Without Schema (1) Map to XQuery Core

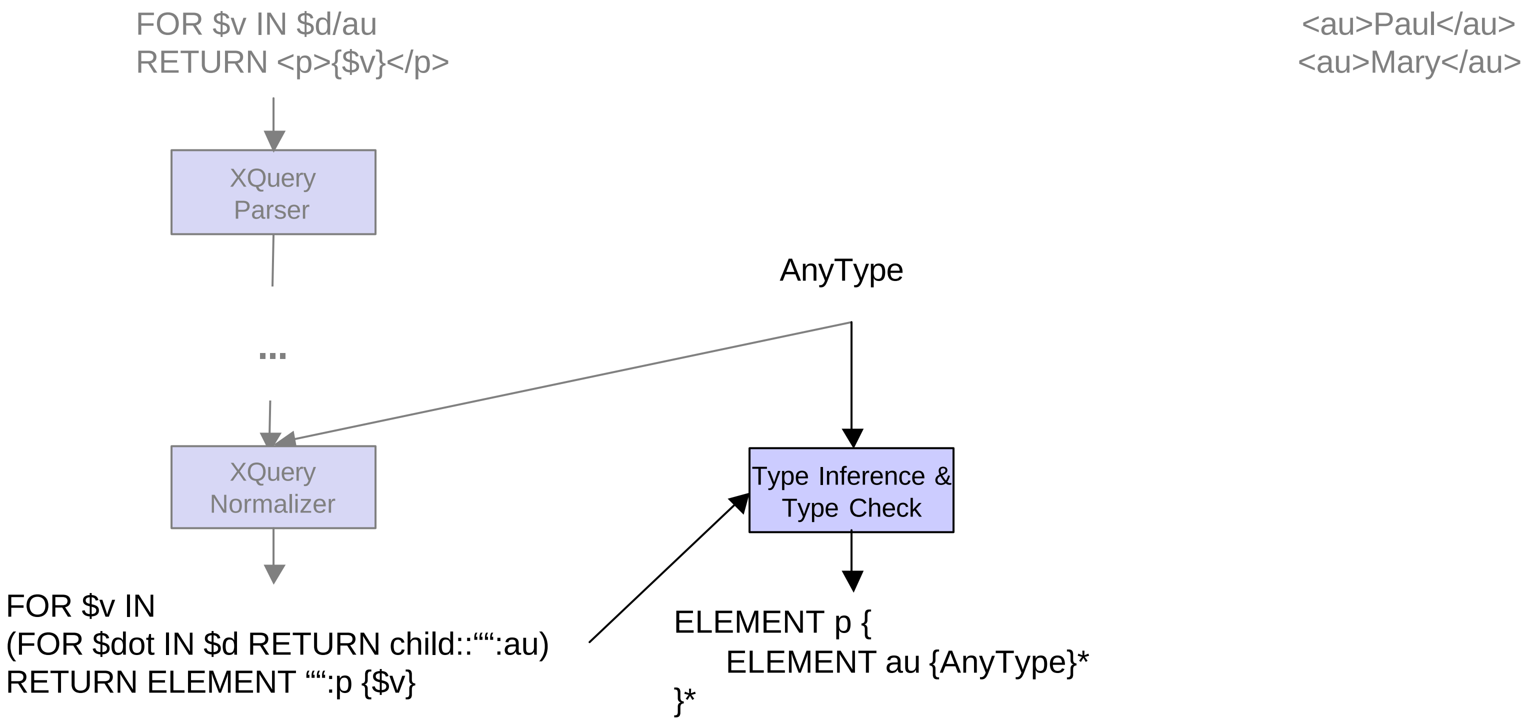


FOR $v IN $d/au
RETURN <p>{$v}</p>

<au>Paul</au>
<au>Mary</au>

FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {$v}

Without Schema (2) Infer Type



```
FOR $v IN $d/au
RETURN <p>{$v}</p>
```

```
<au>Paul</au>
<au>Mary</au>
```

XQuery Parser

...

XQuery Normalizer

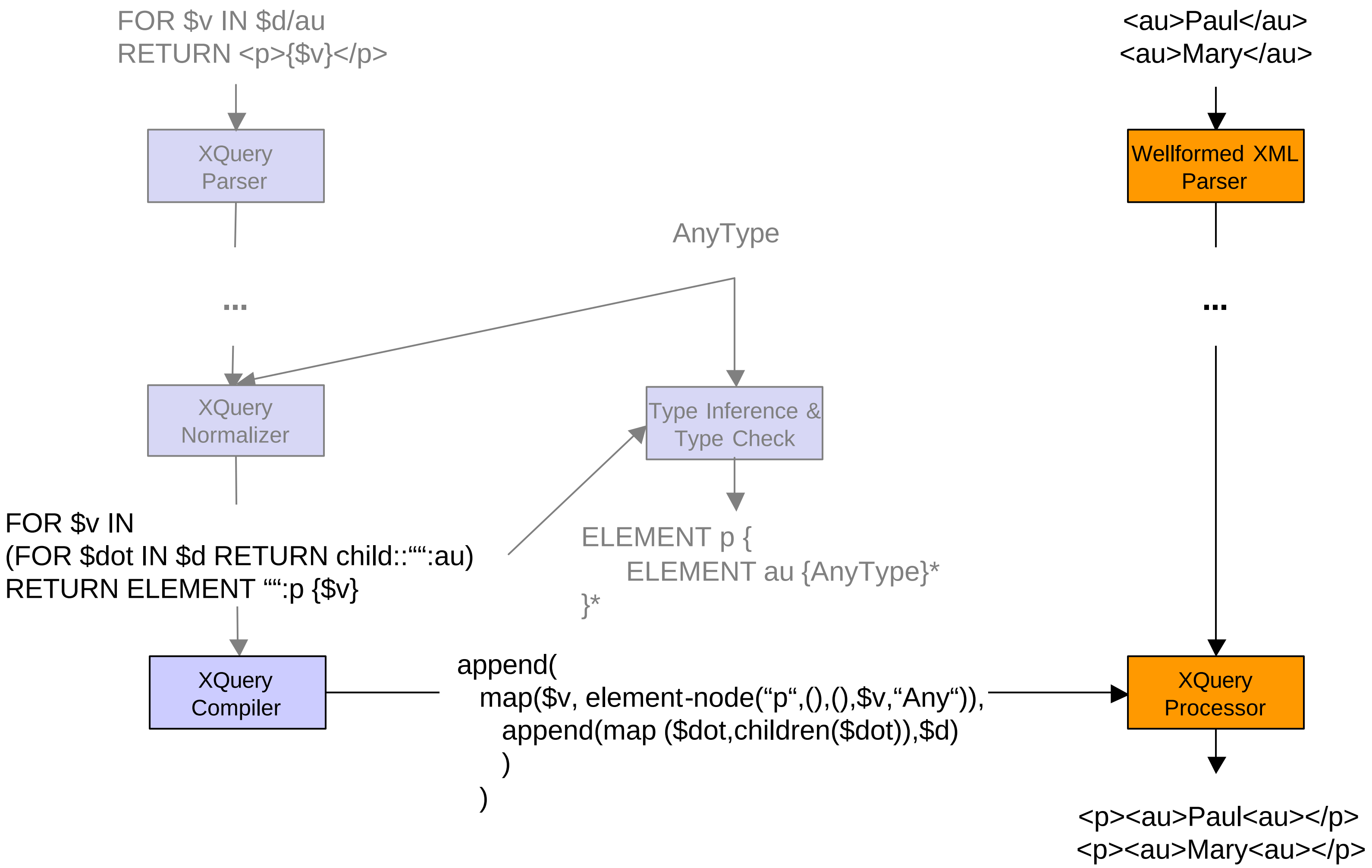
```
FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {$v}
```

AnyType

Type Inference & Type Check

```
ELEMENT p {
     ELEMENT au {AnyType}*
}*
```

Without Schema (3) Evaluate Query



FOR $v IN $d/au
RETURN <p>{$v}</p>

XQuery Parser

...

XQuery Normalizer

FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {$v}

XQuery Compiler

AnyType

Type Inference & Type Check

ELEMENT p {
    ELEMENT au {AnyType}*
}*

append(
  map($v, element-node(“p“,(),(),$v,“Any“)),
    append(map ($dot,children($dot)),$d)
)
)

<au>Paul</au>
<au>Mary</au>

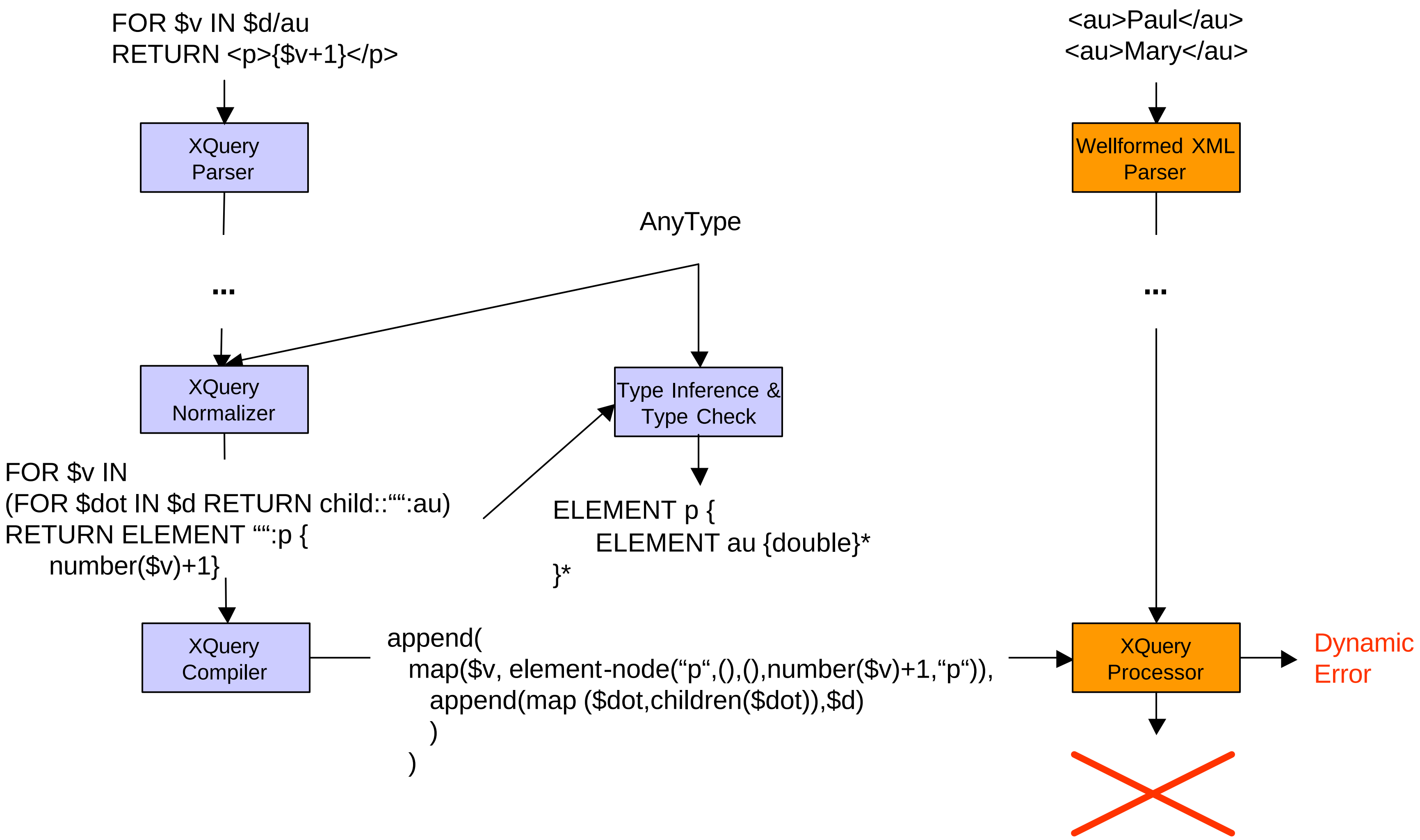
Wellformed XML Parser

...

XQuery Processor

<p><au>Paul<au></p>
<p><au>Mary<au></p>

Without Schema (4) Dynamic Error



FOR $v IN $d/au
RETURN <p>{$v+1}</p>

XQuery Parser

...

XQuery Normalizer

FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {
number($v)+1}

XQuery Compiler

AnyType

Type Inference & Type Check

ELEMENT p {
ELEMENT au {double}*
}*

append(
map($v, element-node(“p“,(),(),number($v)+1,“p“)),
append(map ($dot,children($dot)),$d)
)
)

<au>Paul</au>
<au>Mary</au>

Wellformed XML Parser

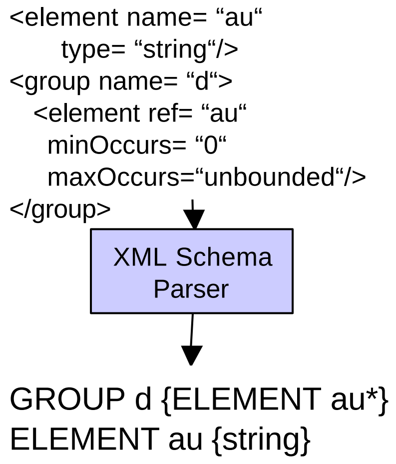
...

XQuery Processor

Dynamic Error

With Schema (1) Generate Types

FOR $v IN $d/au
RETURN <p>{$v}</p>

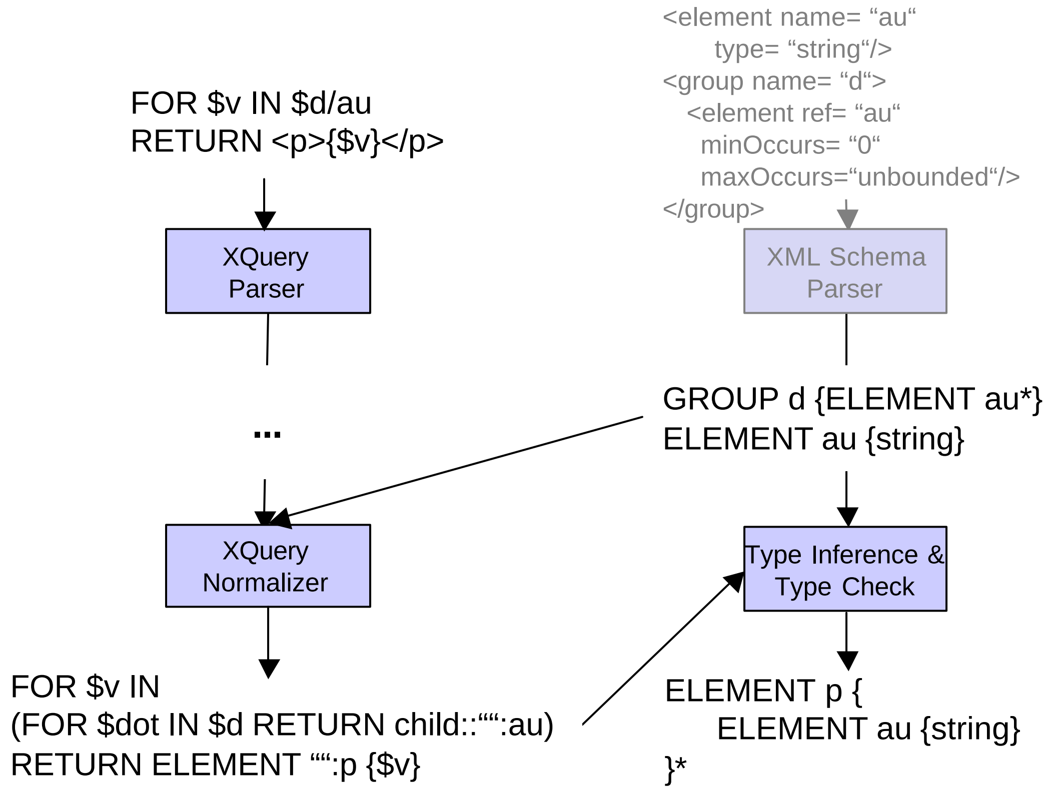


```
<element name= “au“
      type= “string“/>
<group name= “d“>
   <element ref= “au“
     minOccurs= “0“
     maxOccurs=“unbounded“/>
</group>
```

GROUP d {ELEMENT au*}
ELEMENT au {string}

<au>Paul</au>
<au>Mary</au>

With Schema (2) Infer Type



```
FOR $v IN $d/au
RETURN <p>{$v}</p>
```

XQuery Parser

...

XQuery Normalizer

```
FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {$v}
```

```
<element name= “au“
      type= “string“/>
<group name= “d“>
   <element ref= “au“
     minOccurs= “0“
     maxOccurs=“unbounded“/>
</group>
```

XML Schema Parser

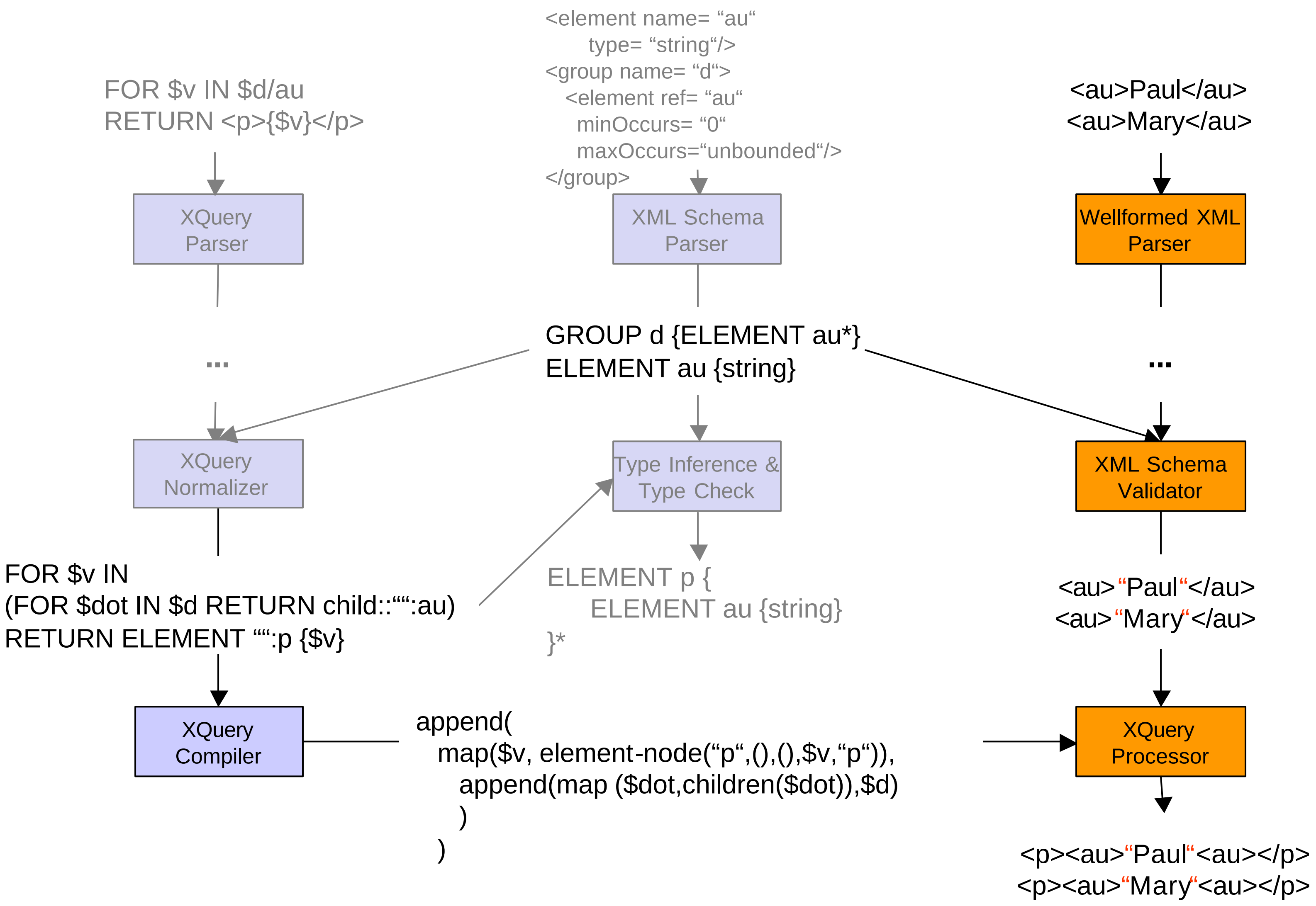
```
GROUP d {ELEMENT au*}
ELEMENT au {string}
```

Type Inference & Type Check

```
ELEMENT p {
      ELEMENT au {string}
}*
```

```
<au>Paul</au>
<au>Mary</au>
```

With Schema (3) Validate and Evaluate



FOR $v IN $d/au
RETURN <p>{$v}</p>

XQuery Parser

...

XQuery Normalizer

FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {$v}

XQuery Compiler

<element name= “au“
type= “string“/>
<group name= “d“>
<element ref= “au“
minOccurs= “0“
maxOccurs=“unbounded“/>
</group>

XML Schema Parser

GROUP d {ELEMENT au*}
ELEMENT au {string}

Type Inference & Type Check

ELEMENT p {
ELEMENT au {string}
}*

append(
map($v, element-node(“p“,(),(),$v,“p“)),
append(map ($dot,children($dot)),$d)
)
)

<au>Paul</au>
<au>Mary</au>

Wellformed XML Parser

...

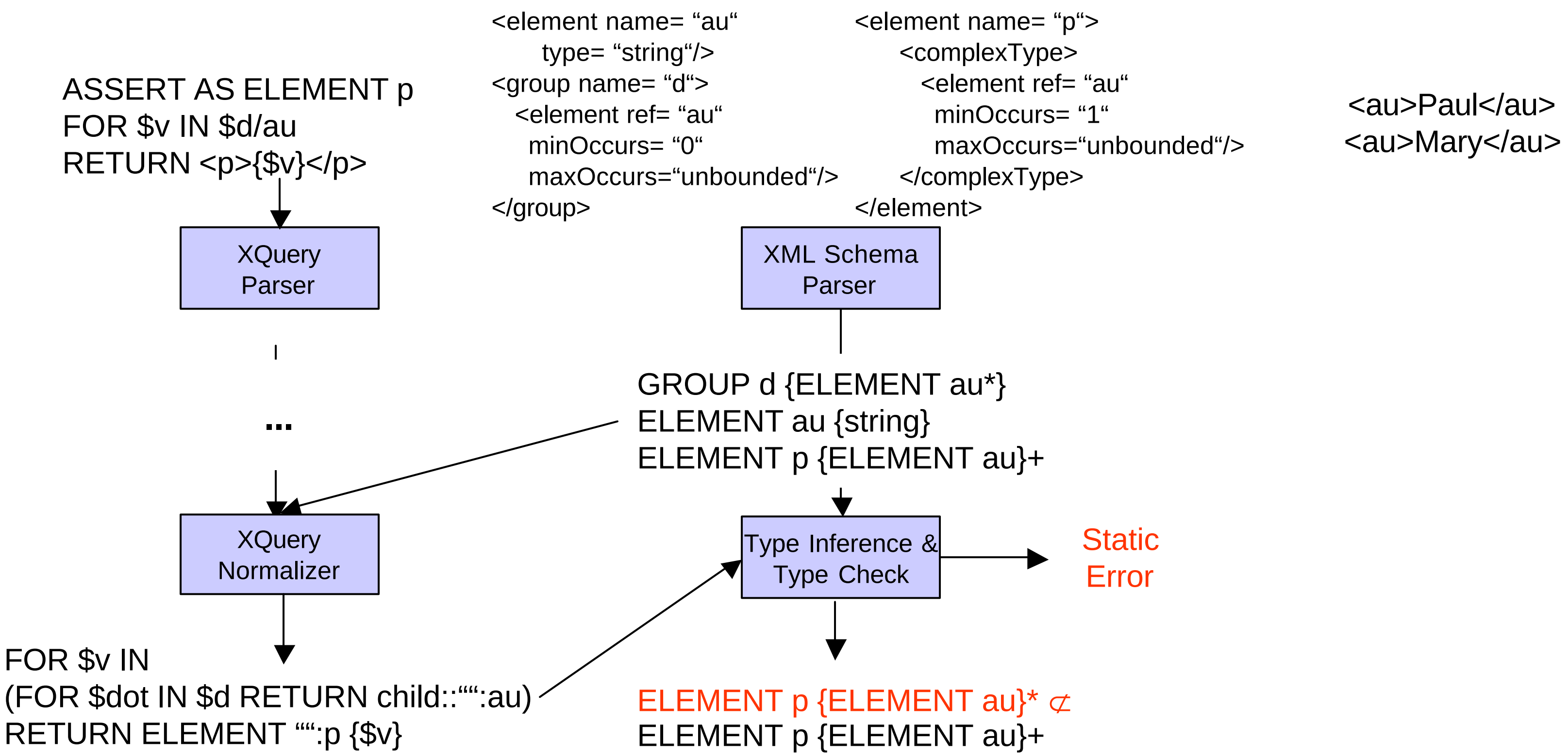
XML Schema Validator

<au>“Paul“</au>
<au>“Mary“</au>

XQuery Processor

<p><au>“Paul“<au></p>
<p><au>“Mary“<au></p>

With Schema (4) Static Error



```
ASSERT AS ELEMENT p
FOR $v IN $d/au
RETURN <p>{$v}</p>
```

```
<element name= “au“
      type= “string“/>
<group name= “d“>
   <element ref= “au“
     minOccurs= “0“
     maxOccurs=“unbounded“/>
</group>
```

```
<element name= “p“>
      <complexType>
         <element ref= “au“
           minOccurs= “1“
           maxOccurs=“unbounded“/>
      </complexType>
</element>
```

<au>Paul</au>
<au>Mary</au>

GROUP d {ELEMENT au*}
ELEMENT au {string}
ELEMENT p {ELEMENT au}+

Static Error

FOR $v IN
(FOR $dot IN $d RETURN child::““:au)
RETURN ELEMENT ““:p {$v}

ELEMENT p {ELEMENT au}* ⊄
ELEMENT p {ELEMENT au}+

Part IV

From XML Schema to XQuery Types

XML Schema vs. XQuery Types

- XML Schema:
  structural constraints on types
  name constraints on types
  range and identity constraints on values
  type assignment and *determinism constraint*

- XQuery Types as a subset:
  structural constraints on types
  local and global elements
  derivation hierarchies, substitution groups by *union*
  name constraints are an open issue
  no costly range and identity constraints

- XQuery Types as a superset:
  XQuery needs closure for inferred types, thus no *determinism constraint* and no *consistent element restriction*.

XQuery Types

unit type	u	$::=$	`string`	string
		$\mid$	`integer`	integer
		$\mid$	`attribute` a `{` t `}`	attribute
		$\mid$	`attribute * {` t `}`	wildcard attribute
		$\mid$	`element` a `{` t `}`	element
		$\mid$	`element * {` t `}`	wildcard element
type	t	$::=$	u	unit type
		$\mid$	`()`	empty sequence
		$\mid$	t `,` t	sequence
		$\mid$	$t \mid t$	choice
		$\mid$	t`?`	optional
		$\mid$	t`+`	one or more
		$\mid$	t`*`	zero or more
		$\mid$	x	type reference

Expressive power of XQuery types

Tree grammars and tree automata

	deterministic	non-deterministic
top-down	Class 1	Class 2
bottom-up	Class 2	Class 2

Tree grammar Class 0: DTD (global elements only)

Tree automata Class 1: Schema (determinism constraint)

Tree automata Class 2: XQuery, XDuce, Relax

$$\text{Class } 0 < \text{Class } 1 < \text{Class } 2$$

Class 0 and Class 2 have good closure properties.

Class 1 does not.

Importing schemas and using types

- SCHEMA *targetNamespace*
  SCHEMA *targetNamespace* AT *schemaLocation*
  import schemas
- VALIDATE *expr*
  validate and assign types to the results of *expr*
  (a loaded document or a query)
- ASSERT AS *type* (*expr*)
  check statically whether the type of (*expr*) matches *type*.
- TREAT AS *type* (*expr*)
  check dynamically whether the type of (*expr*) matches *type*
- CAST AS *type* (*expr*)
  convert simple types according to conversion table
  open issue: converting complex types.

Primitive and simple types

Schema

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

XQuery type

```
DEFINE TYPE myInteger { xsd:integer }
DEFINE TYPE listOfMyIntType { myInteger* }
```

Local simple types

Schema

```
<xsd:element name="quantity">
  <xsd:simpleType>
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxExclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

XQuery type

```
DEFINE ELEMENT quantity { xsd:positiveInteger }
```

Ignore: id, final, annotation, minExclusive, minInclusive, maxExclusive, maxInclusive, totalDigits, fractionDigits, length, minLength, maxLength, enumeration, whiteSpace, pattern attributes.

Complex-type declarations (1)

Schema

```
<xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>
```

Complex-type declarations (2)

XQuery type

```
DEFINE ELEMENT purchaseOrder { PurchaseOrderType }
DEFINE ELEMENT comment { xsd:string }
DEFINE TYPE PurchaseOrderType {
   ATTRIBUTE orderDate { xsd:date }?,
   ELEMENT shipTo { USAddress },
   ELEMENT billTo { USAddress },
   ELEMENT comment?,
   ELEMENT items { Items },
}
```

`<sequence>` ⇒ `','`

`<choice>` ⇒ `'|'`

`<all>` ⇒ `'&'`

Open issue: name of group PurchaseOrderType is insignificant.

Local elements and anonymous types (1)

Schema

```
<xsd:complexType name="Items"
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"  type="xsd:decimal"/>
      <xsd:element ref="comment"   minOccurs="0"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
```

Local elements and anonymous types (2)

XQuery type

```
DEFINE TYPE Items {
   ELEMENT item {
      ELEMENT productName { xsd:string },
      ELEMENT quantity { xsd:positiveInteger },
      ELEMENT USPrice { xsd:decimal },
      ELEMENT comment?,
      ELEMENT shipDate { xsd:date }?,
      ATTRIBUTE partNum { SKU }
   }*
}
```

Local elements are supported by nested declarations

Occurrence constraints

Schema

```
<xsd:simpleType name="SomeUSStates">
 <xsd:restriction base="USStateList">
  <xsd:length value="3"/>
 </xsd:restriction>
</xsd:simpleType>
```

XQuery type

```
DEFINE TYPE SomeUSStates { USState+ }
```

Only ? for $\{0,1\}$, $*$ for $\{0,\text{unbounded}\}$, $+$ for $\{1, \text{unbounded}\}$
More specific occurrence constraints only by explicit enumeration.

Derivation by restriction (1)

Schema

```
<complexType name="ConfirmedItems">
 <complexContent>
   <restriction base="Items">
    <xsd:sequence>
    <element name="item" minOccurs="1" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"  type="xsd:decimal"/>
      <xsd:element ref="comment"   minOccurs="0"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
    ...
```

Derivation by restriction (2)

XQuery type

An instance of type ConfirmedItems is also of type Items.

```
DEFINE TYPE ConfirmedItems {
   ELEMENT item {
      ELEMENT productName { xsd:string },
      ELEMENT quantity { xsd:positiveInteger },
      ELEMENT USPrice { decimal },
      ELEMENT ipo:comment?,
      ELEMENT shipDate { xsd:date }?,
      ATTRIBUTE partNum { SKU }
   }+
}
```

Only structural part is preserved, complex type name ConfirmedItem is not preserved (open issue).

Derivation by extension (1)

Schema

```
<complexType name="Address">
  <element name="street" type="string"/>
  <element name="city"   type="string"/>
</complexType>
<complexType name="USAddress">
 <complexContent>
  <extension base="Address">
    <element name="state" type="USState"/>
    <element name="zip"   type="positiveInteger"/>
  </extension>
 </complexContent>
</complexType>
<complexType name="UKAddress">
 <complexContent>
  <extension base="Address">
    <element name="postcode" type="UKPostcode"/>
    <attribute name="exportCode" type="positiveInteger" fixed="1"/>
  </extension>
 </complexContent>
</complexType>
```

Derivation by extension (2)

XQuery type

```
DEFINE TYPE Address {
   ELEMENT street { xsd:string },
   ELEMENT city { xsd:string }
   ( ()  {!-- possibly empty, except if Address is abstract --}
         {!-- extensions from USAddress --}
   | (ELEMENT state { USState },
      ELEMENT zip { xsd:positiveInteger })

         !-- extensions from UKAddress --
   | (ELEMENT postcode { UKPostcode },
      ATTRIBUTE exportCode { xsd:positiveInteger })
   )
}
```

Group contains base type and all types derived from it. Thereby USAddress and UKAddress are substitutable for Address.

Substitution groups (1)

Schema

```
<element name="shipTo" address="ipo:Address">
<element name="shipToUS" type="ipo:USAddress"
         substitutionGroup="ipo:shipTo"/>
<element name="order">
  <complexType>
    <sequence>
     <element name="item" type="integer"/>
     <element ref="shipTo"/>
    </sequence>
  </complexType>
<element>
```

Substitution groups (2)

XQuery types

```
DEFINE ELEMENT shipTo { Address }
DEFINE ELEMENT shipToUS { USAddress }

DEFINE TYPE shipTo_group {
  shipTo | shipToUS
}

DEFINE ELEMENT order {
  ELEMENT item { integer },
  shipTo_group
}
```

Union semantics: group contains 'representative' element & all elements in its substitution group

XML Schema vs. XQuery Types - summary

XQuery types are aware of

- Global and local elements
- Sequence, choice, and simple repetition
- Derivation hierarchies and substitution groups
- Mixed content
- Built-in simple types

XQuery types are not aware of

- complex type names
  open issue
- value constraints
  check with VALIDATE

Part V

Type Inference and Subsumption

What is a type system?

- Validation: Value has type

$$v \in t$$

- Static semantics: Expression has type

$$e : t$$

- Dynamic semantics: Expression has value

$$e \Rightarrow v$$

- Soundness theorem: Values, expressions, and types match

$$\text{if} \quad e : t \quad \text{and} \quad e \Rightarrow v \quad \text{then} \quad v \in t$$

What is a type system? (with variables)

- Validation: Value has type

$$v \in t$$

- Static semantics: Expression has type

$$\bar{x} : \bar{t} \vdash e : t$$

- Dynamic semantics: Expression has value

$$\bar{x} \Rightarrow \bar{v} \vdash e \Rightarrow v$$

- Soundness theorem: Values, expressions, and types match

  if $\bar{v} \in \bar{t}$ and $\bar{x} : \bar{t} \vdash e : t$ and $\bar{x} \Rightarrow \bar{v} \vdash e \Rightarrow v$ then $v \in t$

Documents

string	s	::=	`""` , `"a"`, `"b"`, `...`, `"aa"`, `...`	
integer	i	::=	`...`, `-1`, `0`, `1`, `...`	
document	d	::=	s	string
		$\mid$	i	integer
		$\mid$	`attribute` a `{` d `}`	attribute
		$\mid$	`element` a `{` d `}`	element
		$\mid$	`()`	empty sequence
		$\mid$	d , d	sequence

Type of a document

- Overall Approach:

  Walk down the document tree

  Prove the type of d by proving the types of its constituent nodes.

- Example:

$$\frac{d \in t}{\texttt{element}\ a\ \texttt{\{}\ d\ \texttt{\}} \in \texttt{element}\ a\ \texttt{\{}\ t\ \texttt{\}}} \qquad \text{(element)}$$

Read: the type of `element` a `{` d `}` is `element` a `{` t `}` if the type of d is t.

Type of a document — $d \in t$

$$\frac{}{s \in \texttt{string}} \qquad \text{(string)}$$

$$\frac{}{i \in \texttt{integer}} \qquad \text{(integer)}$$

$$\frac{d \in t}{\texttt{element } a \texttt{ \{ } d \texttt{ \} } \in \texttt{element } a \texttt{ \{ } t \texttt{ \}}} \qquad \text{(element)}$$

$$\frac{d \in t}{\texttt{element } a \texttt{ \{ } d \texttt{ \} } \in \texttt{element * \{ } t \texttt{ \}}} \qquad \text{(any element)}$$

$$\frac{d \in t}{\texttt{attribute } a \texttt{ \{ } d \texttt{ \} } \in \texttt{element } a \texttt{ \{ } t \texttt{ \}}} \qquad \text{(attribute)}$$

$$\frac{d \in t}{\texttt{attribute } a \texttt{ \{ } d \texttt{ \} } \in \texttt{element * \{ } t \texttt{ \}}} \qquad \text{(any attribute)}$$

$$\frac{d \in t \qquad \texttt{define group } x \texttt{ \{ } t \texttt{ \}}}{d \in x} \qquad \text{(group)}$$

Type of a document, continued

$$\frac{}{() \in ()} \qquad \text{(empty)}$$

$$\frac{d_1 \in t_1 \qquad d_2 \in t_2}{d_1 \mathbin{,} d_2 \in t_1 \mathbin{,} t_2} \qquad \text{(sequence)}$$

$$\frac{d_1 \in t_1}{d_1 \in t_1 \mid t_2} \qquad \text{(choice 1)}$$

$$\frac{d_2 \in t_2}{d_2 \in t_1 \mid t_2} \qquad \text{(choice 2)}$$

$$\frac{d \in t\texttt{+?}}{d \in t\texttt{*}} \qquad \text{(star)}$$

$$\frac{d \in t \mathbin{,} t\texttt{*}}{d \in t\texttt{+}} \qquad \text{(plus)}$$

$$\frac{d \in () \mid t}{d \in t\texttt{?}} \qquad \text{(option)}$$

Type of an expression

- Overall Approach:

  Walk down the operator tree

  Compute the type of $expr$ from the types of its constituent expressions.

- Example:

$$\frac{e_1 \in t_1 \qquad e_2 \in t_2}{e_1 \; , \, e_2 \in t_1 \; , \, t_2} \qquad \text{(sequence)}$$

  Read: the type of e_1 , e_2 is a sequence of the type of e_1 and the type of e_2

Type of an expression — $E \vdash e \in t$

$$\text{environment} \quad E \quad ::= \quad \$v_1 \in t_1, \ldots, \$v_n \in t_n$$

$$\frac{E \quad \text{contains} \quad \$v \in t}{E \vdash \$v \in t} \qquad \text{(variable)}$$

$$\frac{E \vdash e_1 \in t_1 \qquad E, \$v \in t_1 \vdash e_2 \in t_2}{E \vdash \texttt{let } \$v \texttt{ := } e_1 \texttt{ return } e_2 \in t_2} \qquad \text{(let)}$$

$$\frac{}{E \vdash \texttt{()} \in \texttt{()}} \qquad \text{(empty)}$$

$$\frac{E \vdash e_1 \in t_1 \qquad E \vdash e_2 \in t_2}{E \vdash e_1 \texttt{ , } e_2 \in t_1 \texttt{ , } t_2} \qquad \text{(sequence)}$$

$$\frac{E \vdash e \in t_1 \qquad t_1 \cap t_2 \neq \emptyset}{E \vdash \texttt{treat as } t_2 \texttt{ (}e\texttt{)} \in t_2} \qquad \text{(treat as)}$$

$$\frac{E \vdash e \in t_1 \qquad t_1 \subseteq t_2}{E \vdash \texttt{assert as } t_2 \texttt{ (}e\texttt{)} \in t_2} \qquad \text{(assert as)}$$

Typing FOR loops

Return all Amazon and Fatbrain books by Buneman

```
define element AMAZON-BOOK { TITLE, AUTHOR+ }
define element FATBRAIN-BOOK { AUTHOR+, TITLE }
define element BOOKS { AMAZON-BOOK*, FATBRAIN-BOOK* }
for $book in (/BOOKS/AMAZON-BOOK, /BOOKS/FATBRAIN-BOOK)
where $book/AUTHOR = "Buneman" return
  $book
```

$\in$

```
( AMAZON-BOOK | FATBRAIN-BOOK )*
```

$$\frac{\begin{array}{c} E \vdash e_1 \in t_1 \\ E,\ \$x \in \mathsf{P}(t_1) \vdash e_2 \in t_2 \end{array}}{E \vdash \texttt{for}\ \$x\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \in t_2 \cdot \mathsf{Q}(t_1)} \quad \text{(for)}$$

$$\begin{array}{lcl} \mathsf{P}(\texttt{AMAZON-BOOK*,FATBRAIN-BOOK*}) & = & \texttt{AMAZON-BOOK | FATBRAIN-BOOK} \\ \mathsf{Q}(\texttt{AMAZON-BOOK*,FATBRAIN-BOOK*}) & = & * \end{array}$$

Prime types

unit type	u	$::=$	`string`	string
		$\mid$	`integer`	integer
		$\mid$	`attribute` a `{` t `}`	attribute
		$\mid$	`attribute * {` t `}`	any attribute
		$\mid$	`element` a `{` t `}`	element
		$\mid$	`element * {` t `}`	any element
prime type	p	$::=$	u	unit type
		$\mid$	$p \mid p$	choice

Quantifiers

quantifier q ::= () exactly zero
| - exactly one
| ? zero or one
| + one or more
| * zero or more

$$
\begin{aligned}
t \cdot \texttt{()} &= \texttt{()}\\
t \cdot \texttt{-} &= t\\
t \cdot \texttt{?} &= t\texttt{?}\\
t \cdot \texttt{+} &= t\texttt{+}\\
t \cdot \texttt{*} &= t\texttt{*}
\end{aligned}
$$

,	()	-	?	+	*
()	()	-	?	+	*
-	-	+	+	+	+
?	?	+	*	+	*
+	+	+	+	+	+
*	*	+	*	+	*

|	()	-	?	+	*
()	()	?	?	*	*
-	?	-	?	+	*
?	?	?	?	*	*
+	*	+	*	+	*
*	*	*	*	*	*

·	()	-	?	+	*
()	()	()	()	()	()
-	()	-	?	+	*
?	()	?	?	*	*
+	()	+	*	+	*
*	()	*	*	*	*

≤	()	-	?	+	*
()	≤		≤		≤
-		≤	≤	≤	≤
?			≤		≤
+				≤	≤
*					≤

Factoring

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\mathsf{P}'(u) & = & \{u\} & \mathsf{Q}(u) & = & \text{-} \\
\mathsf{P}'(()) & = & \{\} & \mathsf{Q}(()) & = & () \\
\mathsf{P}'(t_1 \,,\, t_2) & = & \mathsf{P}'(t_1) \cup \mathsf{P}'(t_2) & \mathsf{Q}(t_1 \,,\, t_2) & = & \mathsf{Q}(t_1) \,,\, \mathsf{Q}(t_2) \\
\mathsf{P}'(t_1 \mid t_2) & = & \mathsf{P}'(t_1) \cup \mathsf{P}'(t_2) & \mathsf{Q}(t_1 \mid t_2) & = & \mathsf{Q}(t_1) \mid \mathsf{Q}(t_2) \\
\mathsf{P}'(t?) & = & \mathsf{P}'(t) & \mathsf{Q}(t?) & = & \mathsf{Q}(t) \cdot ? \\
\mathsf{P}'(t+) & = & \mathsf{P}'(t) & \mathsf{Q}(t+) & = & \mathsf{Q}(t) \cdot + \\
\mathsf{P}'(t*) & = & \mathsf{P}'(t) & \mathsf{Q}(t*) & = & \mathsf{Q}(t) \cdot *
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{P}(t) & = & () & \text{if } \mathsf{P}'(t) = \{\} \\
 & = & u_1 \mid \cdots \mid u_n & \text{if } \mathsf{P}'(t) = \{u_1, \ldots, u_n\}
\end{array}
$$

Factoring theorem. For every type t, prime type p, and quantifier q, we have $t \subseteq p \cdot q$ iff $\mathsf{P}(t) \subseteq p?$ and $\mathsf{Q}(t) \leq q$.

Corollary. For every type t, we have $t \subseteq \mathsf{P}(t) \cdot \mathsf{Q}(t)$.

Uses of factoring

$$\frac{\begin{array}{c} E \vdash e_1 \in t_1 \\ E,\ \$x \in \mathsf{P}(t_1) \vdash e_2 \in t_2 \end{array}}{E \vdash \texttt{for}\ \$x\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \in t_2 \cdot \mathsf{Q}(t_1)} \qquad \text{(for)}$$

$$\frac{E \vdash e \in t}{E \vdash \texttt{unordered(}e\texttt{)} \in \mathsf{P}(t) \cdot \mathsf{Q}(t)} \qquad \text{(unordered)}$$

$$\frac{E \vdash e \in t}{E \vdash \texttt{distinct(}e\texttt{)} \in \mathsf{P}(t) \cdot \mathsf{Q}(t)} \qquad \text{(distinct)}$$

$$\frac{\begin{array}{cc} E \vdash e_1 \in \texttt{integer} \cdot q_1 & q_1 \leq \texttt{?} \\ E \vdash e_2 \in \texttt{integer} \cdot q_2 & q_2 \leq \texttt{?} \end{array}}{E \vdash e_1 \texttt{ + } e_2 \in \texttt{integer} \cdot q_1 \cdot q_2} \qquad \text{(arithmetic)}$$

Subtyping and type equivalence

Definition. Write $t_1 \subseteq t_2$ iff for all d, if $d \in t_1$ then $d \in t_2$.

Definition. Write $t_1 = t_2$ iff $t_1 \subseteq t_2$ and $t_2 \subseteq t_1$.

Examples

$$
\begin{gathered}
t \subseteq t\texttt{?} \subseteq t\texttt{*} \\
t \subseteq t\texttt{+} \subseteq t\texttt{*} \\
t_1 \subseteq t_1 \mid t_2 \\
t \texttt{,} () = t = () \texttt{,} t \\
t_1 \texttt{,} (t_2 \mid t_3) = (t_1 \texttt{,} t_2) \mid (t_1 \texttt{,} t_3) \\
\texttt{element}\ a\ \texttt{\{}\ t_1 \mid t_2\ \texttt{\}} = \texttt{element}\ a\ \texttt{\{}\ t_1\ \texttt{\}} \mid \texttt{element}\ a\ \texttt{\{}\ t_2\ \texttt{\}}
\end{gathered}
$$

Can decide whether $t_1 \subseteq t_2$ using tree automata:

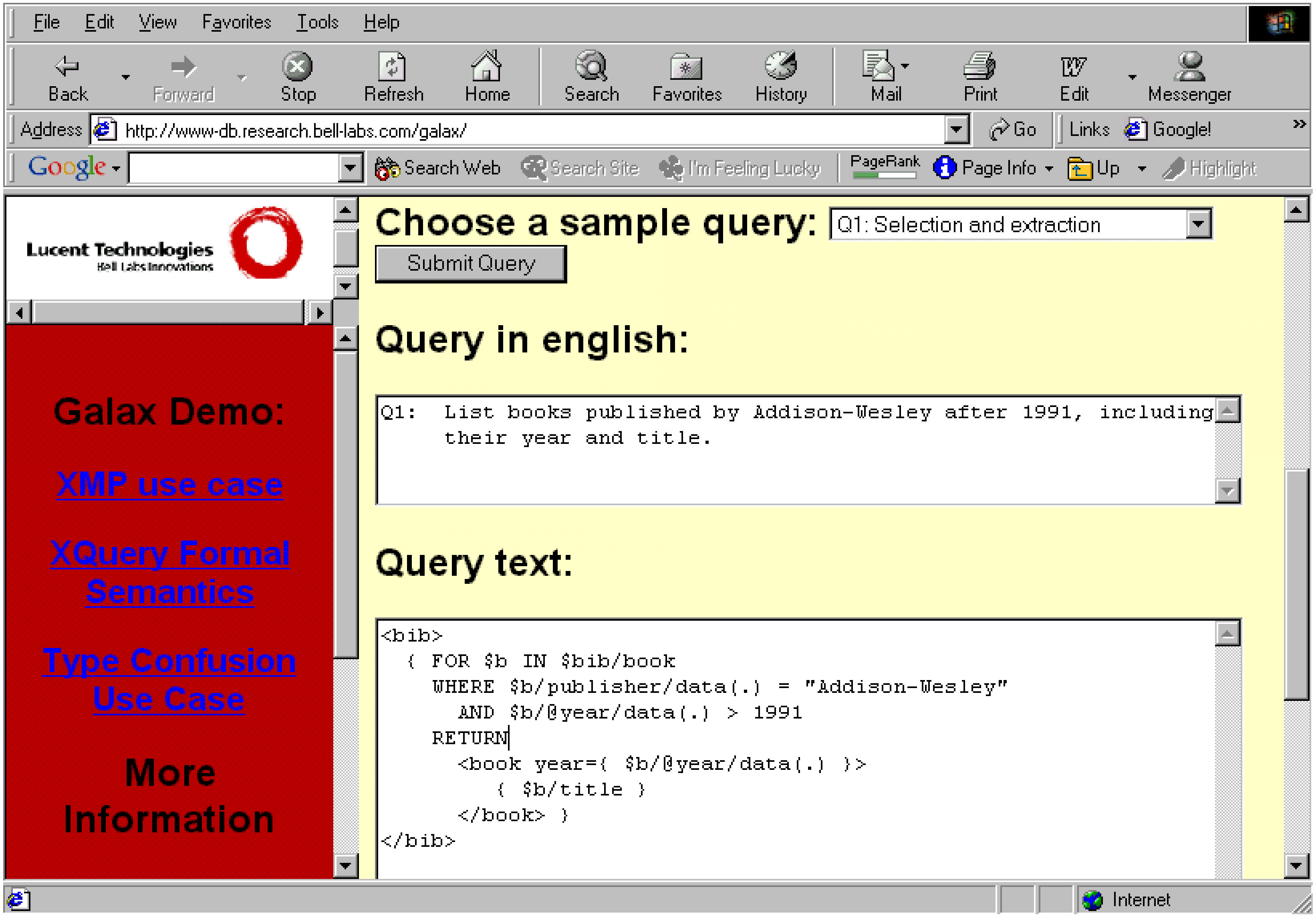
$Language(t_1) \subseteq Language(t_2)$ iff

$Language(t_1) \cap Language(Complement(t_2)) = \emptyset$.

Part VI

Further reading and experimenting

Galax



Choose a sample query: Q1: Selection and extraction

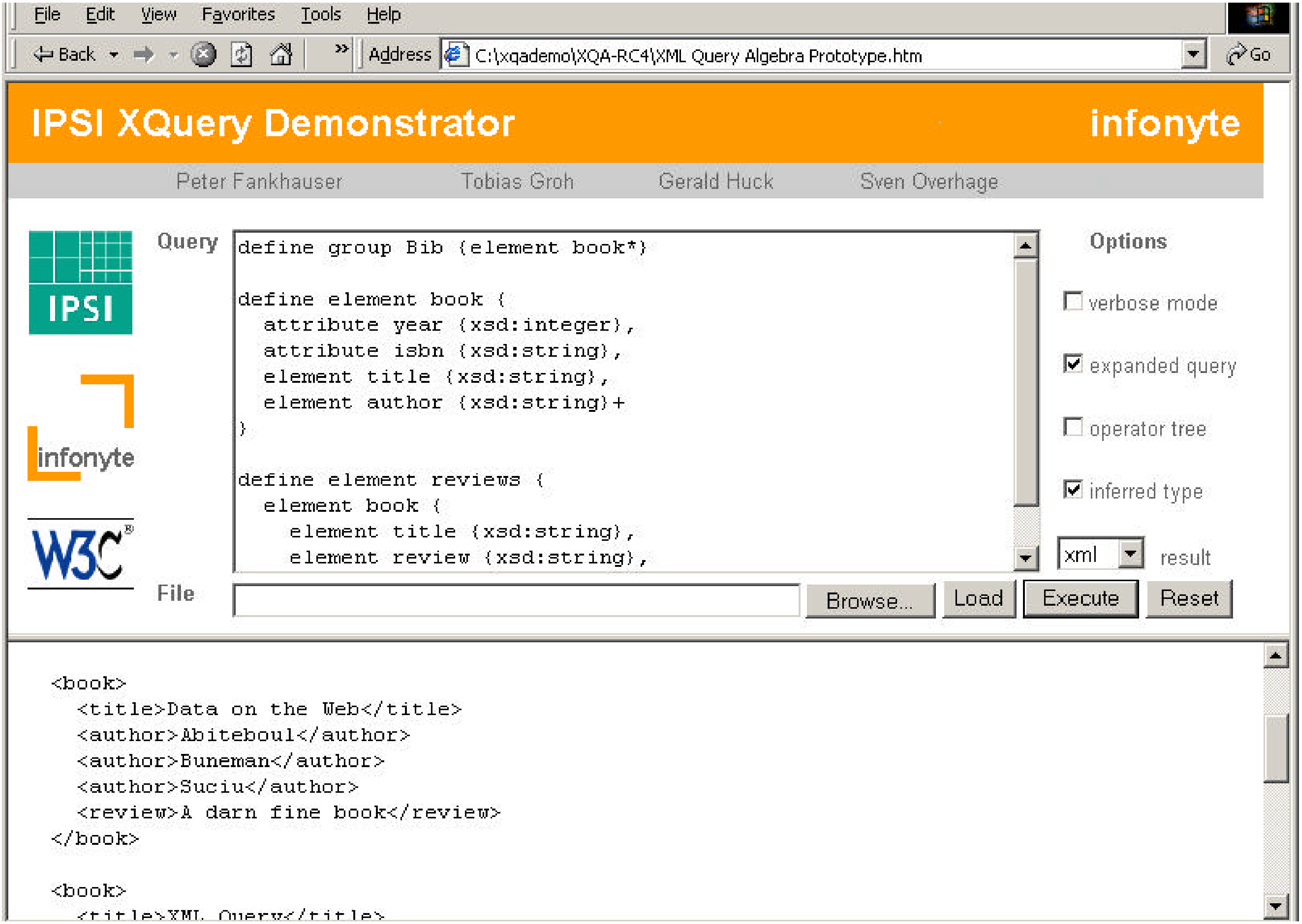
Query in english:

Q1: List books published by Addison-Wesley after 1991, including their year and title.

Query text:

```
<bib>
  { FOR $b IN $bib/book
    WHERE $b/publisher/data(.) = "Addison-Wesley"
      AND $b/@year/data(.) > 1991
    RETURN
      <book year={ $b/@year/data(.) }>
         { $b/title }
      </book> }
</bib>
```

IPSI XQuery Demonstrator



Options:

☐ verbose mode

☑ expanded query

☐ operator tree

☑ inferred type

xml result

Query:

```
define group Bib {element book*}

define element book {
  attribute year {xsd:integer},
  attribute isbn {xsd:string},
  element title {xsd:string},
  element author {xsd:string}+
}

define element reviews {
  element book {
    element title {xsd:string},
    element review {xsd:string},
```

```
<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>
```

Links

Phil's XML page

`http://www.research.avayalabs.com/~wadler/xml/`

W3C XML Query page

`http://www.w3.org/XML/Query.html`

XML Query demonstrations

- `Galax - AT&T, Lucent, and Avaya`
  - `http://www-db.research.bell-labs.com/galax/`
- `Quip - Software AG`
  - `http://www.softwareag.com/developer/quip/`
- `XQuery demo - Microsoft`
  - `http://131.107.228.20/xquerydemo/`
- `Fraunhofer IPSI XQuery Prototype`
  - `http://xml.ipsi.fhg.de/xquerydemo/`
- `XQengine - Fatdog`
  - `http://www.fatdog.com/`
- `X-Hive`
  - `http://217.77.130.189/xquery/index.html`
- `OpenLink`
  - `http://demo.openlinksw.com:8391/xquery/demo.vsp`