Notes on XML Data, XQuery, Type System and Type Inference
Revision 1

This revision introduces examples and provides some implementation directives.

1 Data

We will assume that the labeled trees corresponding to XML data consist of text nodes and element nodes.

- Each text node has a typed value that comes from the set **String** or the set **Integer**.
- Each element node x has a name $l(x)$ and content $c(x)$, which is a list (sequence) of nodes.

We ignore document nodes (at the cost of being unable to check the name of the root element of a document), attribute, comment, processing instruction, and namespace nodes.

2 Type System

We will assume the following type system, which captures the essentials of XML Schema and XQuery's type system.

$$\begin{array}{rcl} T & ::= & \mathsf{String} \\ & | & \mathsf{Integer} \\ & | & \mathsf{element}\ n\ \{T_1\} \\ & | & T_1, T_2 \\ & | & T_1 | T_2 \\ & | & T_1* \\ & | & \epsilon \end{array}$$

Given a type T we will say that T is atomic ($atom(T)$ is true) if T is String, $\mathsf{Integer}$ or $\mathsf{element}$.

The *extension* $[\![T]\!]$ of a type T is a set of sequences of nodes. It is defined as follows, where if x is a node and s_1 and s_2 are sequences, we denote by $[x]$ the singleton list consisting of the node x and we denote by "s_1, s_2" the concatenation of s_1 and s_2.

$$\begin{array}{lcl} [\![\mathsf{String}]\!] & = & \{[x] \mid x \in \mathbf{String}\} \\ [\![\mathsf{Integer}]\!] & = & \{[x] \mid x \in \mathbf{Integer}\} \\ [\![\mathsf{element}\ n\ \{T_1\}]\!] & = & \{[x] \mid l(x) = n \wedge c(x) \in [\![T_1]\!]\} \\ [\![T_1, T_2]\!] & = & \{s_1, s_2 \mid s_1 \in [\![T_1]\!], s_2 \in [\![T_2]\!]\} \\ [\![T_1 | T_2]\!] & = & \{s \mid s \in [\![T_1]\!] \vee s \in [\![T_2]\!]\} \\ [\![T*]\!] & = & \{s_1, s_2, \ldots, s_n \mid n \geq 0, s_i \in [\![T]\!], i = 1, \ldots, n\} \\ [\![\epsilon]\!] & = & \{[]\} \end{array}$$

2.1 Type File

A file with the type of the single input document of your XQuery will also be given at the processor. The first line of this type file will stand for the type of the root element. The other lines will provide type definitions. The exact syntax of the file is open-ended. The following example provides two possibilities.

EXAMPLE 2.1 Consider a bibliography file, where the root element is named `bibliography` and contains zero or more books, followed by zero or more reviews. Each book is an element named `book`, it has one title, multiple authors, one year of publication and one publisher. Similarly, reviews have a single title and multiple comments.

The following is a possible type file, where we name all "element" types.

```
element bibliography { Book*, Review* }
Book         : element book { Title, Author*, Year, Publisher }
Title        : element title { String }
Author       : element author { String }
Year         : element year { Integer }
Publisher    : element publisher { String }
Review       : element review { Title, Comment* }
Comment      : element comment { String }
```

Indeed, you may even choose a type file with no named types, other than the root type.

```
element bibliography {
                  element book { element title { String },
                                 element author { String }*,
                                 element year { Integer },
                                 element publisher { String }
                               }*,
                  element review { element title { String },
                                   element comment { String }*,
                                 }*
                     }
```

Any type file syntax that is convenient for your type processing code and reasonably readable is acceptable. ◇

3 XQuery

We consider the following subset (and slight modification) of XQuery

$$\begin{array}{lcl}
XQuery & ::= & Var \\
 & | & XQuery_1, XQuery_2 \\
 & | & \langle n\rangle\{XQuery_1\}\langle/n\rangle \\
 & | & XQuery_1/n_1/\ldots/n_m \\
 & | & XQuery_1/n_1/\ldots/\mathsf{node}() \\
 & | & \mathsf{for}\ V_1\ \mathsf{in}\ XQuery_1 \\
 & & \quad\vdots \\
 & & \quad V_n\ \mathsf{in}\ XQuery_n \\
 & & \mathsf{let}\ V_1' := XQuery_1' \\
 & & \quad\vdots \\
 & & \quad V_m' := XQuery_m' \\
 & & \mathsf{where}\ Cond\ \mathsf{return}\ XQuery_r \\
 & | & \mathsf{doc}(FileName) \\
Cond & ::= & XQuery_1\ \mathsf{eq}\ (XQuery_2|Constant) \\
 & | & XQuery_1 = (XQuery_2|Constant) \\
 & | & \mathsf{empty}(XQuery_1) \\
 & | & \mathsf{some}\ V_1\ \mathsf{in}\ XQuery_1 \\
 & & \quad\vdots \\
 & & \quad V_n\ \mathsf{in}\ XQuery_n \\
 & & \mathsf{satisfies}\ Cond \\
 & | & Cond_1\ \mathsf{AND}\ Cond_2 \\
 & | & Cond_1\ \mathsf{OR}\ Cond_2 \\
 & | & \mathsf{NOT}\ Cond_1 \\
Constant & | & StringLiteral \\
 & | & IntegerLiteral
\end{array}$$

We will assume that our queries involve only one input document so that the only type file provided refers to the single input.

Semantics of `eq` We will depart slightly from XQuery semantics and assume that eq receives two list arguments and returns true if the two lists are deep equal, where deep equality of lists is defined as follows:

1. the two lists have equal length, say n,
2. for each $i = 1, \ldots, n$, the ith tree of the left argument is deep equal to the ith tree of the right argument

Two trees are deep equal if their root nodes have the same label and, recursively, the children list of the root node of the first is deep equal to the children list of the root node of the second.

Some examples:

```
(<a> 5 </a>) eq 5 -> FALSE
(<a> 5 </a>) eq (<a> 5 </a> <b> 4 </b>) -> FALSE
(<a> 5 </a>) eq (<a> 5 </a>) -> TRUE
```

Associativity in XQuery Path expressions have stronger associativity than concatenation - unless parentheses are used as indicated in the following example.

EXAMPLE 3.1 The expression

```
$V , $W / a
```

is equivalent to

```
$V , ($W / a)
```

If one wants the concatenation to happen first and then run the path "`/a`" on the result of the concatenation, he should write

```
($V , $W) / a
```

◇

4 XQuery Core

We consider the following XQuery core, to which all expressions of the considered XQuery can be reduced:

$$\begin{array}{lcl}
\mathit{XQuery} & ::= & \mathit{Var} \\
 & | & \mathit{XQuery}_1, \mathit{XQuery}_2 \\
 & | & \langle n\rangle\{\mathit{XQuery}_1\}\langle/n\rangle \\
 & | & \mathit{XQuery}_1/n \\
 & | & \mathit{XQuery}_1/\mathsf{node}() \\
 & | & \mathsf{for}\ \mathit{Var}\ \mathsf{in}\ \mathit{XQuery}_1\ \mathsf{where}\ \mathit{Cond}\ \mathsf{return}\ \mathit{XQuery}_2 \\
 & | & \mathsf{doc}(\mathit{FileName}) \\
\mathit{Cond} & ::= & \mathit{XQuery}_1\ \mathsf{eq}\ (\mathit{XQuery}_2|\mathit{Constant}) \\
 & | & \mathsf{empty}(\mathit{XQuery}_1) \\
 & | & \mathit{Cond}_1\ \mathsf{AND}\ \mathit{Cond}_2 \\
 & | & \mathit{Cond}_1\ \mathsf{OR}\ \mathit{Cond}_2 \\
 & | & \mathsf{NOT}\ \mathit{Cond}_1 \\
\mathit{Constant} & | & \mathit{StringLiteral} \\
 & | & \mathit{IntegerLiteral}
\end{array}$$

5 Type Inference

We statically assign a type to each subexpression of an XQuery by employing a list of equations that connect the type of an expression to the types of its constituent subexpressions.

Path Expressions We want to find the type of e/a , where $e \in T$. The type inference needs an extra operator $\sigma_a e$ to be expressed. When applied on a list s the operator returns the following set:

$$\sigma_a s = \{x \mid x \in s \wedge l(x) = a\}$$

We provide some inference rules that describe an induction on the structure of the type T and they provide a tight inference (hence the use of $=$). The base rules of the induction are the following. Rules for $\mathsf{Integer}$ are similar to the ones for String.

$$\begin{array}{ll}
\mathit{type}(e/a \mid e \in \epsilon) & = \epsilon \\
\mathit{type}(e/a \mid e \in \mathsf{String}) & = \epsilon \\
\mathit{type}(e/a \mid e \in T_1, T_2) & = \mathit{type}(e/a \mid e \in T_1), \mathit{type}(e/a \mid e \in T_2) \\
\mathit{type}(e/a \mid e \in T_1|T_2) & = \mathit{type}(e/a \mid e \in T_1)|\mathit{type}(e/a \mid e \in T_2) \\
\mathit{type}(e/a \mid e \in T_1*) & = \mathit{type}(e/a \mid e \in T_1)* \\
\mathit{type}(e/a \mid e \in \mathsf{element}\ n\ \{T_1\}) & = \mathit{type}(\sigma_a e \mid e \in T_1)
\end{array}$$

$$\begin{array}{ll}
\mathit{type}(\sigma_a e \mid e \in \epsilon) & = \epsilon \\
\mathit{type}(\sigma_a e \mid e \in \mathsf{string}) & = \epsilon \\
\mathit{type}(\sigma_a e \mid e \in \mathsf{element}\ n\ \{T_1\}) & = \epsilon, if\ a \neq n \\
\mathit{type}(\sigma_a e \mid e \in \mathsf{element}\ a\ \{T_1\}) & = \mathsf{element}\ a\ \{T_1\} \\
\mathit{type}(\sigma_a e \mid e \in T_1, T_2) & = \mathit{type}(\sigma_a e \mid e \in T_1), \mathit{type}(\sigma_a e \mid e \in T_2) \\
\mathit{type}(\sigma_a e \mid e \in T_1|T_2) & = \mathit{type}(\sigma_a e \mid e \in T_1)|\mathit{type}(\sigma_a e \mid e \in T_2) \\
\mathit{type}(\sigma_a e \mid e \in T_1*) & = \mathit{type}(\sigma_a e \mid e \in T_1)*
\end{array}$$

Variable

$$\mathit{type}(V|V \in T) = T$$

Concatenation

$$\mathit{type}(e_1, e_2|e_1 \in T_1, e_2 \in T_2) = \mathit{type}(e_1|e_1 \in T_1), \mathit{type}(e_2|e_2 \in T_2)$$

Element Creation

$$\mathit{type}(\langle n\rangle e_1\langle/n\rangle|e_1 \in T_1) = elementn\{T_1\}$$

For expressions We provide type inference rules for

$$\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e$$

The reductions for for including where are similar.

$$\begin{array}{ll}
\mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in \epsilon) & = \epsilon \\
\mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T) & = \mathit{type}(e|V \in T),\ if\ atom(T) \\
\mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_1, T_2) & = \mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_1), \\
 & \quad \mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_2) \\
\mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_1|T_2) & = \mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_1)| \\
 & \quad \mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_2) \\
\mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_1*) & = \mathit{type}(\mathsf{for}\ V\ \mathsf{in}\ g\ \mathsf{return}\ e \mid g \in T_1)*
\end{array}$$

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), year, publisher, price )>
<!ELEMENT year (#PCDATA)>
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

Figure 1: The bibliography DTD of the XMP use case

```
<!ELEMENT reviews (entry*)>
<!ELEMENT entry   (title, price, review)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT price   (#PCDATA)>
<!ELEMENT review  (#PCDATA)>
```

Figure 2: The reviews DTD of the XMP use case

6 Test Cases

The following is a modification of the W3C XMP XQuery Use Case that fits the subset of XQuery we work on. The XMP use case (`http://www.w3.org/TR/xmlquery-use-cases/#xmp`) is based on a bibliography file with the DTD of Figure 1. A corresponding type file is given in Figure 3. The type file also contains reviews, which in the original XMP use case are part of a separate file but we include in the bibliography file, so that we have the full input for our queries in a single file. Figure 4 provides sample data that we will use in the examples. In the original use case `year` is an attribute but we converted it into an element since we do not consider attributes. Corresponding changes to the queries were made. The queries have also been modified to work around the limitations of our XQuery subset.

Next we provide a list of queries from the XML use case, appropriately modified to avoid the limitations of our XQuery subset.

Q1 List books published by Addison-Wesley in 1992, including their year and title.
Solution in XQuery:

```
<bib>
 {
  for $b in doc("input")/book
  where $b/publisher/node() = "Addison-Wesley" and $b/year/node() = 1992
  return
    <book>
```

```
element bib { element book { element title { String },
                             ( element author { element last { String },
                                                element first {String}
                                              }*
                              | element editor { element last { String },
                                                 element first {String},
                                                 element affiliation {String}
                                               }*
                             )
                                element year { Integer },
                                element publisher { String },
                                element price { Integer}
            }*,
                  element entry { element title { String },
                                 element price { String },
                                 element review { String }*,
                               }*
```

Figure 3: A type file for the modified XMP use case

```
     { $b/title, $b/year }
    </book>
 }
</bib>
```

Expected Result:

```
<bib>
    <book>
        <title>Advanced Programming in the Unix environment</title>
        <year>1992</year>
    </book>
</bib>
```

Q2 Create a flat list of all the title-author pairs, with each pair enclosed in a “result” element. Solution in XQuery:

```
<results>
  {
    for $b in doc("input")/book,
        $t in $b/title,
        $a in $b/author
    return
        <result>
            { $t, $a }
```

```
<bib>
    <book>
        <year>1994</year>
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price> 65.95</price>
    </book>
    <book>
        <year>1992</year>
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>
    <book>
        <year>2000</year>
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>
    <book>
        <year>1999</year>
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
            <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
    <entry>
        <title>Data on the Web</title>
        <price>34.95</price>
        <review>
               A very good discussion of semi-structured database
               systems and XML.
        </review>
    </entry>
    <entry>
        <title>Advanced Programming in the Unix environment</title>
        <price>65.95</price>
        <review>
               A clear and detailed discussion of UNIX programming.
        </review>
    </entry>
    <entry>
        <title>TCP/IP Illustrated</title>
        <price>65.95</price>
        <review>
               One of the best books on TCP/IP.
        </review>
</reviews>
</bib>
```

Figure 4: Sample Data

```
        </result>
  }
</results>
```

Expected Result:

```
<results>
    <result>
        <title>TCP/IP Illustrated</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </result>
    <result>
        <title>Advanced Programming in the Unix environment</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </result>
    <result>
        <title>Data on the Web</title>
        <author>
            <last>Abiteboul</last>
            <first>Serge</first>
        </author>
    </result>
    <result>
        <title>Data on the Web</title>
        <author>
            <last>Buneman</last>
            <first>Peter</first>
        </author>
    </result>
    <result>
        <title>Data on the Web</title>
        <author>
            <last>Suciu</last>
            <first>Dan</first>
        </author>
    </result>
</results>
```

Q3 For each book in the bibliography, list the title and authors, grouped inside a "result" element.
Solution in XQuery:

```
<results>
```

```
{
    for $b in doc("input")/bib/book
    return
        <result>
            { $b/title,
              $b/author  }
        </result>
}
</results>
```

Expected Result:

```
<results>
    <result>
        <title>TCP/IP Illustrated</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </result>
    <result>
        <title>Advanced Programming in the Unix environment</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </result>
    <result>
        <title>Data on the Web</title>
        <author>
            <last>Abiteboul</last>
            <first>Serge</first>
        </author>
        <author>
            <last>Buneman</last>
            <first>Peter</first>
        </author>
        <author>
            <last>Suciu</last>
            <first>Dan</first>
        </author>
    </result>
    <result>
        <title>The Economics of Technology and Content for Digital TV</title>
    </result>
</results>
```

Q6 For each `book` with a matching `entry`, list the title of the book and its price from the entry.

Solution in XQuery:

```
<books-with-prices>
  {
    for $b in doc("input")/book,
        $a in doc("input")/entry
    where $b/title = $a/title
    return
        <book-with-prices>
            { $b/title,
              <price-review>{ $a/price/node() }</price-review>,
              <price>{ $b/price/node() }</price>
            }
        </book-with-prices>
  }
</books-with-prices>
```

Expected Result:

```
<books-with-prices>
    <book-with-prices>
        <title>TCP/IP Illustrated</title>
        <price-review>65.95</price-review>
        <price> 65.95</price>
    </book-with-prices>
    <book-with-prices>
        <title>Advanced Programming in the Unix environment</title>
        <price-review>65.95</price-review>
        <price>65.95</price>
    </book-with-prices>
    <book-with-prices>
        <title>Data on the Web</title>
        <price-review>34.95</price-review>
        <price>39.95</price>
    </book-with-prices>
</books-with-prices>
```

Q11 For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the book title and the editor's affiliation.

Solution in XQuery:

```
<bib>
{
        for $b in doc("input")/book
        where not empty($b/author)
        return
            <book>
                { $b/title,
                  $b/author }
            </book>
}
{
        for $b in doc("input")/book
        where not empty($b/editor)
        return
          <reference>
            { $b/title,
              $b/editor/affiliation}
          </reference>
}
</bib>
```

Expected Result:

```
<bib>
    <book>
        <title>TCP/IP Illustrated</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </book>
    <book>
        <title>Advanced Programming in the Unix environment</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </book>
    <book>
        <title>Data on the Web</title>
        <author>
            <last>Abiteboul</last>
            <first>Serge</first>
        </author>
        <author>
            <last>Buneman</last>
            <first>Peter</first>
        </author>
        <author>
            <last>Suciu</last>
            <first>Dan</first>
        </author>
    </book>
    <reference>
        <title>The Economics of Technology and Content for Digital TV</title>
        <affiliation>CITI</affiliation>
    </reference>
</bib>
```

7 Applying Database Optimization Techniques on XQuery

The XQuery engines built in the first part of the project miss the query optimization opportunities we know from the database literature. This may lead to unacceptable performance, as the following example shows.

EXAMPLE 7.1 Consider the following query, which is a modification of Test Case **Q6**.

```
    for $b in doc("input")/book
        $a in doc("input")/entry
        $tb in $b/title
        $ta in $a/title
    where $tb eq $ta
    return
        <book-with-prices>
            { $tb,
              <price-review>{ $a/price/node() }</price-review>,
              <price>{ $b/price/node() }</price>
            }
        </book-with-prices>
```

Assume the query is evaluated on an input file with 10^5 books and 10^5 reviews. Then in the brute force evaluation of the plan the `where` clause will be evaluated 10^{10} times, which is unacceptable from a performance point of view, since there is a much more efficient way to evaluate the query. In particular, one may:

1. collect a tuple set B, consisting of all tuples (b, t_b) of books and their titles
2. collect a tuple set E, consisting of all tuples (a, t_a) of entries and their titles
3. join the tuple sets B and E on the titles and derive a new tuple set R consisting of tuples (b, t_b, a, t_a)
4. produce a `book_with_prices` element for every tuple of R.

The above plan can employ efficient join methods for Step 3. For example, one may index the tuples of B by title and then for each tuple of E find matching tuples of B using the index. Other efficient join methods can also be used (e.g., sort merge join). ◇

Introducing A Join Operator Let us introduce a new XQuery operator, called `join` that

1. inputs two lists of tuples. Each tuple is of the form

```
tuple
   <$a_1$>$v_1$</$a_1$>
   ⋮
   <$a_n$>$v_n$</$a_n$>
tuple
```

where the strings $a_1, \ldots, a_n$ are the attribute names. Each attribute value $v_1, \ldots, v_n$ is a list of XML elements in the general case. The tuples of each list are homogeneous, in the sense that all tuples have the same attributes.

2. inputs two lists of attribute names, originating in the two tuple lists. We augment the XQuery syntax with a "list of constants" notation. In particular, we denote the list of attributes $a_1, \ldots, a_n$ as $[a_1, \ldots, a_n]$.

3. and outputs a list of tuples, where the output order is non-specific.

Using the join operator we can write more efficient versions of our queries, as the following example shows.

EXAMPLE 7.2 We rewrite the query of Example 7.1, using the join operator.

```
for $tuple in join(for $b in doc("input")/book
                       $tb in $b/title
                   return <tuple> <b> {$b} </b> <tb> {$tb} </tb> </tuple>,

                   for $a in doc("input")/entry
                       $ta in $a/title
                   return <tuple> <a> {$a} </a> <ta> {$ta} </ta> </tuple>,

                   [tb], [ta]
                  )
return
        <book-with-prices>
            { $tuple,
              <price-review>{ $a/price/node() }</price-review>,
              <price>{ $b/price/node() }</price>
            }
        </book-with-prices>
```

The join operator above has four arguments. The first two arguments deliver the book tuples and entry tuples, while the third and fourth arguments specify that the join is on attributes `tb` and `ta`. ◇

Yet Another Subset of XQuery Consider the following subset of non-core XQuery, where there are no nested FLWR expressions and hence it is easier to introduce join operations. One may construct a corresponding core syntax.

$$\begin{array}{lcl}
\mathit{XQuery} & | & \textsf{for } V_1 \textsf{ in } \mathit{Path}_1 \\
 & & \quad\vdots \\
 & & \quad V_n \textsf{ in } \mathit{Path}_n \\
 & & \textsf{where } \mathit{Cond} \textsf{ return } \mathit{Return} \\
\mathit{Path} & ::= & (\textsf{doc}(\mathit{FileName})|\mathit{Var})/n_1/\ldots/n_m \\
 & | & (\textsf{doc}(\mathit{FileName})|\mathit{Var})/n_1/\ldots/\textsf{node}() \\
\mathit{Return} & ::= & \mathit{Var} \\
 & | & \mathit{Return}_1, \mathit{Return}_2 \\
 & | & \langle n\rangle\{\mathit{Return}_1\}\langle/n\rangle \\
 & | & \mathit{Path}_1 \\
\mathit{Cond} & ::= & \mathit{Var}_1 \textsf{ eq } (\mathit{Var}_2|\mathit{Constant}) \\
 & | & \mathit{Cond}_1 \textsf{ AND } \mathit{Cond}_2 \\
\mathit{Constant} & | & \mathit{StringLiteral} \\
 & | & \mathit{IntegerLiteral}
\end{array}$$

Queries of the above subset can be rewritten to make efficient use of the join operator, as the following example shows.

EXAMPLE 7.3 The following query returns triplets of books, where the first book has a first author named `John`, the second book has a common author with the first book and a common author with the third book. Notice that the query below may generate multiple triplets with the same books - even triplets where the same three books appear in the same order.

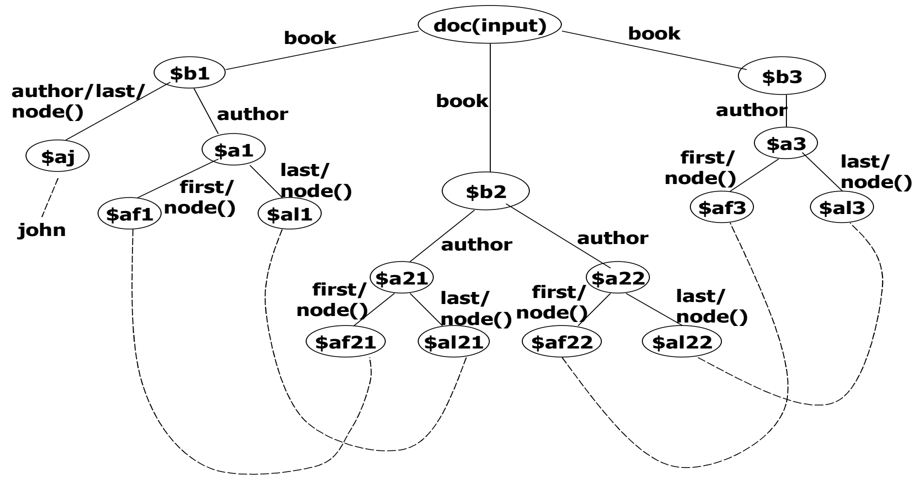
```
for $b1 in doc("input")/book
    $aj in $b1/author/first/node()
    $a1 in $b1/author
    $af1 in $a1/first/node()
    $al1 in $a1/last/node()
    $b2 in doc("input")/book
    $a21 in $b2/author
    $af21 in $a21/first/node()
    $al21 in $a21/last/node()
    $a22 in $b2/author
    $af22 in $a22/first/node()
    $al22 in $a22/last/node()
    $b3 in doc("input")/book
    $a3 in $b3/author
    $af3 in $a3/first/node()
    $al3 in $a3/last/node()
where $aj eq "John" AND
      $af1 eq $af21 AND $al1 eq $al21 AND
      $af22 eq $af3 AND $al22 eq $al3
return <triplet> {$b1, $b2, $b3} </triplet>
```

Notice how the rewriting uses two joins. Notice also that the joins are on pairs of attributes. The first join is on the pairs of attributes `[af1, al1]` and `[af21, al21]`. The second join is on the pairs of attributes `[af22, al22]` and `[af3, al3]`.

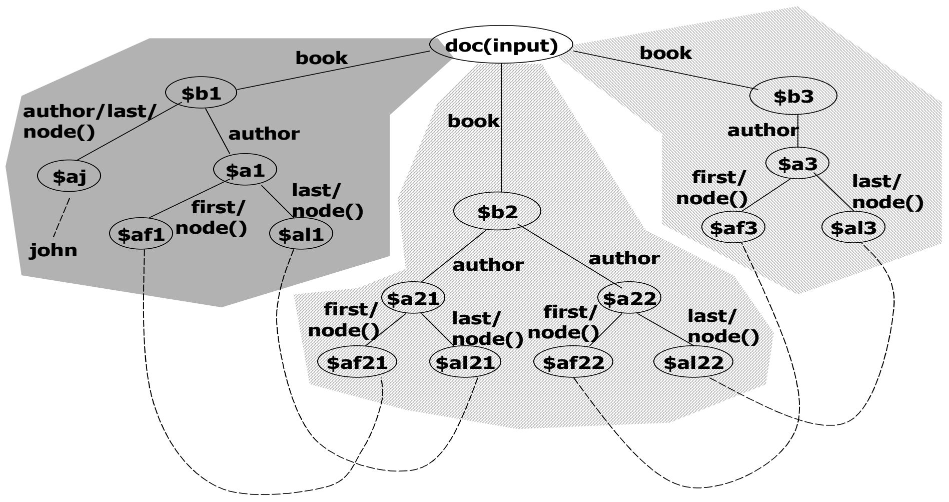
```
for $tuple in join(
               join(for $b1 in doc("input")/book
                        $aj in $b1/author/first/node()
                        $a1 in $b1/author
                        $af1 in $a1/first/node()
                        $al1 in $a1/last/node()
                    where $aj eq "John"
                    return <tuple> <b1>{$b1}</b1>
                                   <af1>{$af1}</af1>
                                   <al1>{$al1}</al1>
                           </tuple>,

                    for $b2 in doc("input")/book
                        $a21 in $b2/author
                        $af21 in $a21/first/node()
                        $al21 in $a21/last/node()
                        $a22 in $b2/author
                        $af22 in $a22/first/node()
                        $al22 in $a22/last/node()
                    return <tuple> <b2>{$b2}</b2>
                                   <af21>{$af21}</af21>
                                   <al21>{$al21}</al21>
                                   <af22>{$af22}</af22>
                                   <al22>{$al22}</al22>
                           </tuple>,

                    [af1, al1], [af21, al21]
                   ),

               for $b3 in doc("input")/book
                   $a3 in $b3/author
                   $af3 in $a3/first/node()
                   $al3 in $a3/last/node()
               return <tuple> <b3>{$b3}</b3>
                              <af3>{$af3}</af3>
                              <al3>{$al3}</al3>
                      </tuple>,

               [af22, al22], [af3, al3]
              )
return <triplet> {$tuple/b1, $tuple/b2, $tuple/b3} </triplet>
```

◇

Implementation Ideas There are multiple algorithms that may lead to rewritings such as the ones described above. The key component of all algorithms is the analysis of navigations performed in the `in` clauses and of the conditions in the `where` clause. A useful tool would be to construct a labeled graph that represents a simplified version of Navig queries and captures the navigation and the conditions.[1] For example the following graph illustrates the navigation and condition structure of the query of Example 7.3. The nodes are labeled with variables or the input document, while the solid edges are labeled with the navigations that connect the variables of the nodes. The dotted edges correspond to the conditions of the `where` clause.



The task of locating the joins can be viewed as a task of locating partitions, as shown below that connect via the dotted edges, which, in effect, become the join conditions.



[1]Navig queries capture also nested FLWR expressions and `some` conditions, which we ignore.