A Data Model and Algebra for XML Query

Mary Fernández	AT&T Labs – Research	mff@research.att.com
Jérôme Siméon	Bell Labs, Lucent Technologies	simeon@research.bell-labs.com
Dan Suciu	AT&T Labs – Research	suciu@research.att.com
Philip Wadler	Bell Labs, Lucent Technologies	wadler@research.bell-labs.com

This document:
http://www.cs.bell-labs.com/ wadler/topics/xml.html#algebra

1 Introduction

This note presents a possible data model and algebra for an XML query language. It should be compared with the alternative proposal [3].

The algebra is derived from the *nested relational algebra*, which is a widely-used algebra for semi-structured and object-oriented databases. For instance, similar techniques are used in the implementation of OQL [2]. We differ from other presentations of nested relational algebra in that we make heavy use of *list comprehensions*, a standard notation in the functional programming community [1]. We find list comprehensions slighly easier to manipulate than the more traditional algebraic operators, but it is not hard to translate comprehensions into these operators (or vice versa).

One important aspect of XML is not covered by traditional nested relational algebras, namely, the structure imposed by a DTD or Schema. (So far as we can see, the proposal [3] also does not cover this aspect.) We extend the nested relational algebra with operators on *regular expressions* to capture this additional structure. The operators for regular expressions are also expressed with a comprehension notation, similar to that used for lists. Again, a similar technique is common in the functional programming community. (Chapter 11 of [1] uses a **do** notation that is quite similar to our comprehensions.)

We use the functional programming language Haskell as a notation for presenting the algebra. This allows us to use a notation that is formal and concrete, without having to invent one from scratch. It also means you can download and play with the algebra, for instance, to try implementing your favorite query. We use a slightly modified version of Haskell that supports regular expression comprehensions. Code that implements the algebra and a modified Hugs interpreter for Haskell can be downloaded from the URL at the head of this document.

The algebra is at the *logical* level, not the *physical* level. Hence, we do not have operators that exploit an index to compute joins efficiently. This concern is deferred to another level.

The remainder of this paper is organized as follows. Section 2 presents the data model. Section 3 presents the algebra. Section 4 presents some of the laws that apply to list comprehensions and regular expressions.

2 Data Model

Our data model is based on the data model for XSLT given in [4], which in turn is based on the data model in the XSLT recommendation [5]. One difference is the addition of reference nodes. The data model here is simpler in a few ways; notably, it merges attribute and element nodes, and eliminates

comment and PI nodes. But these differences are minor; if desired, it is easy to separate attribute and elements nodes, or to restore comment and PI nodes.

The basic type is `Node`. Each `Node` is one of three kinds: text, element, or reference. Correspondingly, we have three constructor functions

```
text          :: String -> Node
elem          :: Tag -> [Node] -> Node
ref           :: Node -> Node
```

For instance, the second line says that `elem` is a function with two arguments, a tag and a list of children nodes, and returns a node. In Haskell, we write `[Node]` for the type of lists where each element is of type `Node`.

Attributes are modeled as elements where the name begins with '`@`'. In our model, children are always in a list. Sometimes the order of this list is relevant, sometimes it is not; in particular, for attributes it is not.

As an example, the XML serialization

```
<bib>
  <book year="1999">
    <title>Data on the Web</title>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
    <year>1999</year>
  </book>
  <book year="1987">
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
  </book>
</bib>
```

corresponds to the model term

```
elem "bib" [
  elem "book" [
    elem "@year"  [ text "1999" ],
    elem "title"  [ text "Data on the Web" ],
    elem "author" [ text "Abiteboul" ],
    elem "author" [ text "Buneman" ],
    elem "author" [ text "Suciu" ]],
  elem "book" [
    elem "@year"  [ text "1987" ],
    elem "title"  [ text "Foundations of Databases" ],
    elem "author" [ text "Abiteboul" ],
    elem "author" [ text "Hull" ],
    elem "author" [ text "Vianu" ]]]
```

In later examples, we will refer to the above node as `bib0`, its first child as `book0`, and that node's first child as `year0`.

To find the type of a node, we have three boolean functions. For each node, exactly one of these returns true.

```
isText        :: Node -> Bool
isElem        :: Node -> Bool
isRef         :: Node -> Bool
```

For a text node, we can access the text.

```
string        :: Node -> String
```

For an element node, we can access its tag and children.

```
tag           :: Node -> Tag
children      :: Node -> [Node]
```

For a reference node, we can access the node referenced.

```
dereference   :: Node -> Node
```

The `string` function is undefined if applied to an element or a reference node, and similiarly for the other functions.

We can check a tag to see whether it is an attribute (begins with `@`).

```
tagAttr       :: Tag -> Bool
```

We can check whether a node is an element with a given tag, and whether it is an attribute.

```
is            :: Tag -> Node -> Bool
is t x        =  isElem x && tag x == t

isAttr        :: Node -> Bool
isAttr x      =  isElem x && tagAttr (tag x)
```

For example, `is "@year" year0` and `isAttr year0` each return true.

It is helpful to have a function that given a node, returns the content of the node. (For a text node, this is its string; for an attribute, this is the value of the attribute; for an element node, it is the concatenation of the value of the non-attribute children.)

```
value         :: Node -> String
```

For example, `value year0` is the string `"1999"`. The DOM provides a function similar to `value`. Later, we will see how to define `value` in terms of the rest of the data model, using structural recursion.

Modelling attributes as elements gives us some flexibility in modelling different types of attribute, as shown in the table below.

Attribute	multiplicity	child type
CDATA	one	text
NMTOKEN	one	text
NMTOKENS	many	text
ID	one	text
IDREF	one	reference
IDREFS	many	reference

The element representing an attribute of type CDATA, has one child of type text, while the element representing an attribute of type IDREFS has any number of children of type reference.

In our model, nodes do not contain a pointer to their parents. We believe this should be optional, as including such pointers can be expensive. Where desired, parent pointers can be represented by an element node with tag "`..`" and one child which is a reference to the parent.

3 Nested relational algebra

In the *relational* approach to databases, everything is a table. In the *nested relational* approach, data is composed of tuples and lists, arbitrarily nested (hence the name). In place of a table, one might have a list of tuples. But also, for instance, a bibliography might be represented by a list of tuples (one for each book), where one field of the tuples is itself a list of authors.

We concentrate here on lists (as opposed to bags and sets). The operations on bags and sets are similar to those on lists, and going into the details here would add little to the exposition other than length.

3.1 Tuples

Tuple values and tuple types are written in round brackets.

```
(1999,"Data on the Web",["Abiteboul","Buneman","Suciu"])
  :: (Int,String,[String])
```

Lists and tuples use related forms for values and types, we hope this is helpful rather than confusing.

To decompose values, we allow tuples to appear on the left-hand side of a definition. For example,

```
year                  :: (Int,String,[String])
year (x,y,l)          =  x
```

Given a tuple representing a book, this extracts the first component.

3.2 Comprehensions

The main tool we will use is the *list comprehension*, which is based on a familiar notation for sets. Here is an example.

```
[ value x | x <- children book0, is "author" x ]
      ==>  [ "Abiteboul", "Buneman", "Suciu" ]
```

This is read as follows: return the list the values of each node `x`, such that `x` is a child of `book0`, and `x` is an author node. (Recall that `value`, `children`, and `is` were all defined in the section on the data model).

Here is a second example.

```
[ value y | x <- children bib0, is "book" x, y <- children x, is "title" y ]
      ==>  [ "Data on the Web", "Foundations of Databases" ]
```

This is read as follows: return the list the values of each node `y`, such that `x` is a child of `bib0`, and `x` is a book node, and `y` is a child of node `x`, and `y` is a title node. Note that here the values of `y` depend on the values of `x`.

For completists, here is a more formal description of comprehensions. In general, a comprehension has the form

```
[ exp | qual1, ..., qualn ]
```

where `exp` is a *return* expression, and each `quali` is a *qualifier*. Each qualifier is either a *filter*, which has the form

```
bool-exp
```

where `bool-exp` is a boolean-valued expression, or a *generator* which has the form

```
pat <- list-exp
```

where `pat` is a *pattern*, and `list-exp` is a list-valued expression. A pattern is either a variable or a tuple of patterns. The arrow in a generator may be pronounced "drawn from". A variable appearing in a pattern is *bound*, and its scope is all qualifiers to its right and the return expression.

3.3 Using comprehensions to write queries

We can use comprehensions to express fundamental query operations such as navigation, cartesian product, nesting, and joins.

We can navigate from a node to all of its children elements with a given tag.

```
follow        :: Tag -> Node -> [Node]
follow t x    =  [ y | y <- children x, is t y ]
```

The term `follow t x` is similar in effect to `$x/t` in XQL or $\phi(E,t)(x)$ in [3]. The examples from the previous section can now be expressed more concisely.

```
[ value x | x <- follow "author" book0 ]
      ==>  [ "Abiteboul", "Buneman", "Suciu" ]
[ value y | x <- follow "book" bib0, y <- follow "title" x ]
      ==>  [ "Data on the Web", "Foundations of Databases" ]
```

Later we will see the reasoning rules that can be used to prove the old forms and the new forms equivalent.

Comprehensions make it easy to compute cartesian products.

```
[ (value y, value z)
| x <- follow "book" bib0,
  y <- follow "title" x,
  z <- follow "author" x ]
      ==>  [ ("Data on the Web", "Abiteboul"),
             ("Data on the Web", "Buneman"),
             ("Data on the Web", "Suciu"),
             ("Foundations of Databases", "Abiteboul"),
             ("Foundations of Databases", "Hull"),
             ("Foundations of Databases", "Vianu") ]
```

Here the comprehension returns pairs of each title with each author.

Comprehensions may be nested.

```
[ (int (value y),
   value z,
   [ value u | u <- follow "author" x ])
| x <- follow "book" bib0,
  y <- follow "@year" x,
  z <- follow "title" x ]
      ==>  [(1999,
             "Data on the Web",
             ["Abiteboul","Buneman","Suciu"])],
            (1991,
             "Foundations of Databases",
             ["Abiteboul,"Hull","Vianu"])]
```

Here a nested comprehension forms a list of all the authors of the book.

Comprehensions make it easy to compute joins. Let us assume a second data source of book reviews.

```
elem "reviews" [
  elem "book" [
    elem "title"  [ text "Data on the Web" ],
    elem "review" [ text "This is great!" ]]
  elem "book" [
    elem "title"  [ text "Foundations of Databases" ],
    elem "review" [ text "This is pretty good too!" ]]]
```

Call the above `reviews0`. Here is a comprehension that joins the two data sources.

```
[ (value y, int (value z), value w)
| x <- follow "book" bib0,
  y <- follow "title" x,
  z <- follow "@year" x,
  u <- follow "book" reviews0,
  v <- follow "title" u,
  w <- follow "review" u,
  y == v ]
      ==>  [("Data on the Web",
             1999,
             "This is great!"),
            ("Foundations of Databases",
             1991,
             "This is pretty good too!")]
```

This finds a book node in the bibliography and a book node in the reviews that have the same title, and returns the title, year, and review for all such nodes.

3.4 Additional operations on lists

We need a few additional operations on lists. These allow us to union queries, to query the position of an element in a list, and to form universal and existential queries.

We may append one list to another list. Append for lists serves much the same function as union for sets.

```
(++)          :: [a] -> [a] -> [a]
```

For example,

```
follow "title" book0 ++ follow "author" book0
      ==> [elem "title" [text "Data on the Web"],
           elem "author" [ text "Abiteboul" ],
           elem "author" [ text "Buneman" ],
           elem "author" [ text "Suciu" ]]
```

(We use infix notation for `++`, treating `x ++ y` and `(++) x y` as equivalent.)

We may pair each element in a list with its index, counting from zero.

```
index         :: [a] -> [(Int,a)]
```

For example,

```
index (follow "author" book0)
      ==>  [(0, elem "author" [ text "Abiteboul" ]),
            (1, elem "author" [ text "Buneman" ]),
            (2, elem "author" [ text "Suciu" ])]
```

One can use indexing to select elements based on their position in a list. For example

```
[ x | (i,x) <- index (follow "author" book0), i < 2 ]
      ==>  [(0, elem "author" [ text "Abiteboul" ]),
            (1, elem "author" [ text "Buneman" ])]
```

selects the first two authors of `book0`.

We may check whether a list is empty.

```
null          :: [a] -> Bool
```

One can use null to test whether there is an element of a list with a given property. For example,

```
null [ x | (i,x) <- index (follow "author" book0), i >= 1 ]
      ==>  False
```

is true if `book0` has at most one author. Both universal and existential queries can be formed in this way, since the two kinds of query are related by negation.

There is a function that takes a list of length one and returns its sole element, and is undefined otherwise.

```
the           :: [a] -> a
```

This function is useful when we expect there to be only one node of a given kind. For example,

```
the (follow "title" book0)
      ==>  elem "title" [text "Data on the Web"]
```

returns the title node of `book0`.

3.5 Structural recursion

We allow recursive and mutually recursive functions, in order to process trees with a recursive structure. In order to ensure termination, we limit when recursion may be used: a recursive function must take one argument that is a node, and any recursive invocation should be on a descendant of that node; since any node has a finite number of descendants, 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.)

For example, here is the definition of `value` promised earlier.

```
value         :: Node -> String
value x       =  if isText x then
                   string x
                 else if isElem x then
                   concat [ value y | y <- children x, not (isAttr y) ]
                 else if isRef x then
                   ""
```

For a text node, this extracts the string; for an element node it finds the value of each of the non-attribute children and concatenates them; for a reference it returns the empty string. Here

```
concat        :: [String] -> String
```

concatenates together a list of strings into a single string.

The following example is taken from Peter Frankhauser. Consider the following document.

```
<bookstore>
  <fiction>
    <sci-fi>
      <book>
        <isbn>0006482805</isbn>
        <title>Do androids dream of electric sheep</title>
        <author>Philip K. Dick</author>
      </book>
    </sci-fi>
    <fantasy>
      <mystery>
```

```
        <book>
          <isbn>0261102362</isbn>
          <title>The two towers</title>
          <author>JRR Tolkien</author>
        </book>
      </mystery>
    </fantasy>
  </fiction>
</bookstore>
```

Here we have books organized in an irregular hierarchy, the P.K.Dick book is on level two, the Tolkien book on level three. More generally, books can be organized at arbitrary level of an irregular hierarchy. A reasonable query might want to hide all but isbns but maintain their context in the orginal irregular hierarchy. That is, we want the following result.

```
<bookstore>
  <fiction>
    <sci-fi>
      <book>
        <isbn>0006482805</isbn>
      </book>
    </sci-fi>
    <fantasy>
      <mystery>
        <book>
          <isbn>0261102362</isbn>
        </book>
      </mystery>
    </fantasy>
  </fiction>
</bookstore>
```

The query should be applicable to any sort of document, regardless of how deeply the actual books are nested.

Here is a recursive function that computes the desired query.

```
isbns         :: Node -> Node
isbns x       =  if is "book" x then
                   elem "book" [ the (follow "isbn" x) ]
                 else
                   elem (tag x) [ isbns y | y <- children x ]
```

If the given node is a book node, then a new book node containing only the isbn is returned; otherwise the element is copied with the function recursively applied to each of its children.

3.6 Regular expression matching

Sequence is significant for some XML trees. Typically, sequence is described by a regular expression, as in a DTD or schema. Regular expressions are composed by sequencing, alternation, and repetition. We introduce a new type `Reg a`, which stands for a regular expression that returns a value of type `a` for each successful match.

Sequencing of regular expressions is written with a comprehension notation similar to that for lists. For example,

```
([ (x,y,u)
 | x <- item "@year",
   y <- item "title",
   u <- rep (item "author") ])
      :: Reg (Node,Node,[Node])
```

is a regular expression comprehension that matches a year, followed by a title, followed by zero or more authors. To distinguish them from list comprehensions, regular expression comprehensions will always be written in parentheses. For each successful match, the regular expression returns a triple consisting of two nodes (the year and title nodes) and a list of nodes (the author nodes). A regular expression matches against the children of a given node. A match returns a list, which is empty if the match fails, or contains multiple entries if the match is ambiguous. For example, if `reg0` is the regular expression above, then

```
match reg0 book0
      ==>  [(elem "@year" [text "1999"],
             elem "title" [text "Data on the Web"],
             [elem "author" [text "Abiteboul"],
              elem "author" [text "Buneman"],
              elem "author" [text "Suciu"]])]
```

In this case, the result is a list with one triple, since the match succeeds and is unambiguous.

Order matters with regular expressions. Here is another book element, identical to `book0` except the year is at the end.

```
elem "book" [
  elem "title"  [ text "Data on the Web" ],
  elem "author" [ text "Abiteboul" ],
  elem "author" [ text "Buneman" ],
  elem "author" [ text "Suciu" ],
  elem "@year"  [ text "1999" ]]
```

Call this `book1`. Then we have

```
match reg0 book1 ==> []
```

The match fails, and so returns the empty list.

There is an operator to indicate the cases where order is unimportant. For example, if `reg1` is the regular expression

```
([ (x,y,u)
 | x <- anywhere (item "@year"),
   y <- item "title",
   u <- rep (item "author") ])
      :: Reg (Node,Node,[Node])
```

then `match reg1 book0` and `match reg1 book1` return the same result. Normally, a regular expression looks for a match at the beginning of the given sequence of nodes. Applying `anywhere` allows it to find a match anywhere in the given sequence of nodes. In either case, subsequent regular expressions apply only to the remaining unmatched nodes.

In general, the match operation takes a regular expression and a node, and returns a list.

```
match         :: Reg a -> Node -> [a]
```

A match returns a list with one entry for each successful match of the regular expression against the children of the node. The list is empty if the match fails, and contains multiple entries if the match is ambiguous.

The most basic regular expression matches against a single node.

```
nodeItem      :: Reg Node
```

Like a list comprehension, a regular expression comprehension may contain a filter. For example, we can define regular expressions that match against a single text node, or a single element node with a given tag.

```
textItem      :: Reg Node
textItem      =  ([ x | x <- nodeItem, isText x ])

item          :: Tag -> Reg Node
item t        =  ([ x | x <- nodeItem, is t x ])
```

We saw an example with `item` above.

Regular expressions can be combined by alternation.

```
(+++)         :: Reg a -> Reg a -> Reg a
```

For example,

```
item "author" +++ item "editor"
```

will match either an author or an editor node.

A regular expression comprehension that contains only the filter `True` returns a value without matching against any items. This is useful for providing defaults. For example,

```
([ int (value x) | x <- item "@year" ]) +++ ([ 1999 | True ])
```

will either match a year node (and return the integer value of the year) or match nothing (and return the integer `1999`).

Regular expressions can be combined by repetition.

```
rep           :: Reg a -> Reg [a]
rep p         =  ([ [x]++l | x <- p, l <- rep p ]) +++ ([ [] | True ])
```

Repetition of regular expressions is defined using sequencing, alternation, and recursion. We saw an example with `rep` above.

Like a list comprehension, a regular expression comprehension may apply a function to the value returned. For example,

```
stringItem    :: Reg String
stringItem    =  ([ string x | x <- textItem ])
```

This regular expression matches a text item, but returns the string that is the value of the text node, rather than the node itself.

The function `step` is similar to `item`, except once it matches the item it also applies a regular expression to the item's children.

```
step          :: Tag -> Reg a -> Reg a
```

For example,

```
step "@year" stringItem :: Reg String
```

first matches a year node, and then matches `string` against the text node that is the child of the year node. As another example,

```
step "name"
  ([ (x,y)
   | x <- step "first" stringItem,
     y <- step "last" stringItem ])
      :: Reg (String, String)
```

matches a name node, and then matches against children representing the first and last names. The name node must contain only first and last children in the specified order or else the match fails. As a final example, here is a variant of something we saw earlier.

```
([ (int x,y,u)
 | x <- anywhere (step "@year" stringItem),
   y <- step "title" stringItem,
   u <- rep (step "author" stringItem) ])
      :: Reg (Int,String,[String])
```

This returns a tuple of integers and strings, which may be more convenient than a tuple of nodes.

3.7 Sorting and grouping

We can sort a list with respect to a given order.

```
sortBy        :: (a -> a -> Bool) -> [a] -> [a]
```

For instance,

```
sortBy (<=) [3,1,2,1]
      ==> [1,1,2,3]
```

Here `(<=) :: Int -> Int -> Bool` is true if its first argument is less than or equal to its second. Sorting is only defined if its first argument is transitive (that is, we will only write $sortByp$ when pxy and pyz implies pxz).

Often, we have a list of tuples and sort on only one field. We have a second version of sort that lets us specify a function to apply to the elements before comparison; usually this will extract a field.

```
sortOn        :: (a -> b) -> (b -> b -> Bool) -> [a] -> [a]
```

For instance

```
sortOn fst (<=) [(3,'d'),(1,'c'),(2,'b'),(1,'a')]
      ==> [(1,'c'),(1,'a'),(2,'b'),(3,'d')]
```

Here the order function is not total, so the order of the resulting elements may not be specified (unless we require a stable sort). It is easy to define `sortOn` in terms of `sortBy`.

```
sortOn f p l  =  [ x | (u,x) <- sort q [ (f x, x) | x <- l ] ]
      where
      q (u,x) (v,y)  =  p u v
```

We can also group a list with respect to a given equality operator.

```
groupBy       :: (a -> a -> Bool) -> [a] -> [[a]]
```

For instance,

```
groupBy (==) [3,1,2,1]
      == [[2],[1,1],[3]]
```

where `(==) :: Int -> Int -> Bool` is true if its arguments are equal. The order of the groups is unspecified, but one may choose to sort the resulting groups.

As with sorting, it is common to have a list of tuples and group on only one field. We have a second version of group, similar to the second version of sort.

```
groupOn       :: (a -> b) -> (b -> b -> Bool) -> [a] -> [(b,[a])]
```

For instance

```
groupOn fst (==) [(3,'d'),(1,'c'),(2,'b'),(1,'a')]
      ==>  [(3,[(3,'d')]),(1,[(1,'c'),(1,'a')]),(2,[(2,'b')])]
```

It is easy to define `groupOn` in terms of `groupBy`.

```
groupOn f p l =  [ (head [ w | (w,x) <- m ], [ x | (w,x) <- m ])
                 | m <- groupBy q [ (fst x, x) | x <- l ] ]
      where
      q (u,x) (v,y)  =  p u v
```

(Well, maybe not easy, but possible.)

3.8 An example

Here we give the translation of a query in XML-QL and a similar query in Yatl, to show how the algebra can express subtle semantic distinctions, such as how order is handled.

Here is the first example from the Exemplars paper, expressed in XML-QL.

```
CONSTRUCT <bib> {
  WHERE
    <bib>
      <book year=$y>
        <title>$t</title>
        <publisher><name>Addison-Wesley</name></publisher>
      </book>
    </bib> IN "www.bn.com/bib.xml",
    $y > 1991
  CONSTRUCT <book year=$y><title>$t</title></book>
} </bib>
```

XML-QL queries ignore the order in which elements appear. Here is its translation into our algebra.

```
query         :: Node -> Node
query x       =  elem "bib"
                   [ elem "book"
                       [ elem "@year" [text (value y)],
                         elem "title" [text (value t)]]
                   | a <- follow "bib" x,
                     b <- follow "book" a,
                     y <- follow "@year" b,
                     t <- follow "title" b,
                     p <- follow "publisher" b,
                     n <- follow "name" p,
                     int (value y) > 1991,
                     value n == "Addison Wesley"  ]
```

When order is irrelevant, we use list comprehensions and `follow`.

Here is the same example expressed in Yatl.

```
make
  bib [ *book [ @year [ $y ],
                title [ $t ] ] ]
match "www.bn.com/bib.xml" with
```

```
  bib [ *book [ @year [ $y ],
                title [ $t ] ],
                publisher [ name [ $n ] ] ]
where
  $n = "Addison-Wesley" and $y > 1991
```

This Yatl query differs from XML-QL in that it specifies that the order is significant. (In Yatl one uses square braces to indicate that order is significant, and curly braces to indicate that order is irrelevant.) Here is its translation into our algebra.

```
query         :: Node -> Node
query x       =  elem "bib"
                   [ elem "book"
                       [ elem "@year" [ text y ],
                         elem "title" [ text t ] ]
                     | (y,t,n) <- the (match bibReg x),
                       int y > 1991,
                       n == "Addison Wesley" ]

bibReg        :: Reg [(String,String,String)]
bibReg        =  step "bib"
                   (rep
                     (step "book"
                        ([ (y,t,n)
                         | y <- step "@year" stringItem,
                           t <- step "title" stringItem,
                           n <- step "publisher"
                                  (step "name" stringItem) ])))
```

When order is significant, we use regular expression comprehensions and `step`.

4 Equivalences

4.1 List comprehensions

List comprehensions can be characterized by the following laws. We write `q` to range over a list of qualifiers.

```
(0)  [ e | q ]               =  [ e | q, True ]
(1)  [ e | x <- [], q]       =  [ ]
(2)  [ e | x <- [v], q]      =  [ e | q ][x:=v]
(3)  [ e | x <- l ++ m, q ]  =  [ e | x <- l, q ] ++ [ e | x <- m, q ]
(4)  [ e | False, q ]        =  [ ]
(5)  [ e | True, q ]         =  [ e | q ]
(6)  [ e | True ]            =  [ e ]
```

Line (0) says we can always append the filter `True` to the end of a list of qualifiers without changing its meaning; doing so provides a simple way to terminate our recursive definition, as shown in line (6). Every list can be simplified to one of three forms: the empty list `[]`, the unit list `[v]`, or the append of two lists `l ++ m`, giving lines (1–3). In line (2), the notation `[ e | q ][x:=v]` stands for the comprehension `[ e | q ]` with all occurrences of variable `x` replaced by value `v` (or with corresponding variables repaced by corresponding expressions if `x` and `v` are tuples). Every filter can be simplified to either `False` or `True`, giving lines (4–5).

Here is how the above laws can be used to evaluate a simple comprehension.

```
[ x*x | x <- [1,2,3], odd x ]
  ==> (3)
[ x*x | x <- [1], odd x ] ++
[ x*x | x <- [2], odd x ] ++
[ x*x | x <- [3], odd x ]
  ==> (2)
[ 1*1 | odd 1 ] ++
[ 2*2 | odd 2 ] ++
[ 3*3 | odd 3 ]
  ==> (4,5)
[ 1*1 ] ++
[ ]     ++
[ 3*3 ]
  ==>
[ 1, 9 ]
```

The following laws are conseqences of the definition of list comprehensions. Let `d` and `e` be expressions, and let `p`, `q`, and `r` be lists of qualifiers.

```
(trivial)     [ x | x <- e ]                =  e
(flattening)  [ d | p, x <- [ e | q ], r ]  =  [ d[x:=e] | p, q, r[x:=e] ]
```

Here `d[x:=e]` and `r[x:=e]` stand for expression `d` and qualifier list `r` with all occurrences of variable `x` replaced by expression `e`. We call this the *flattening* law.

The flattening law can be used to move expressions together, enabling further optimizations. Here is a simple example.

```
[ x+1 | x <- [ y-1 | y <- u ] ]
  ==> (flattening)
[ (y-1)+1 | y <- u ]
  ==> (arithmetic)
[ y | y <- u ]
  ==> (trivial)
u
```

Recall that earlier we made the following definition.

```
follow        :: Tag -> Node -> [Node]
follow t x    =  [ y | y <- children x, is t y ]
```

We then claimed that the comprehension

```
[ value y | x <- children bib0, is "book" x, y <- children x, is "title" y ]
```

and the comprehension

```
[ value y | x <- follow "book" bib0, y <- follow "title" x ]
```

were equivalent. This can be seen immediately by two applications of the above law.

4.2 List comprehensions and algebra

Traditionally, nested relational algebra is presented using a few operators such as `map`, `concatenate`, and `filter`, instead of list comprehensions. We explain the equivalence here.

The operations `map`, `concatenate`, and `filter` can be defined in terms of list comprehensions as follows.

```
map  :: (a -> b) -> [a] -> [b]
map f l =  [ f x | x <- l ]

concat :: [[a]] -> [a]
concat u =  [ x | l <- u, x <- l ]

filter :: (a -> Bool) -> [a] -> [a]
filter p l =  [ x | x <- l, p x ]
```

Conversely, list comprehensions can be defined in terms of `map` and `concat` by the following equations.

```
[ e | x <- l, q ]    =  concat (map g l)
                        where   g x = [ e | q ]
[ e | b, q ]         =  if b then [ e | q ] else []
[ e | True ]         =  [ e ]
```

4.3 Sets

For sets, there are additional laws for comprehensions. We write `set` in front of the comprehension to indicate which sort of collection it acts upon. We can commute two qualifier lists, if neither binds any variable used by the other.

```
set [ e | p, q, r, s ]  =  set [ e | p, r, q, s ]
```

Here the bound variables of `q` must be disjoint from the free variables of `r`, and vice versa. Further, we can delete a qualifier if it mentions variables used nowhere else.

```
set [ e | p, q, r ]  =  set [ e | p, r ]
```

Here no bound variable of `q` appears in `e` or `r`.

For instance, we can use the above to push selections about, as happens in relational algebra.

```
  set [ u | (u,v) <- set [ (y,z) | y <- follow "title" x, z <- follow "author" x ] ]
=
  set [ y | y <- follow "title" x, z <- follow "author" x ]
=
  set [ y | y <- follow "title" x ]
=
  set (follow "title" x)
```

4.4 Regular expressions

To give a formal specification of regular expressions, we define a regular expression as a function.

```
type  Reg a   =  [Node] -> [(a,[Node])]
```

A regular expression is a function that takes as its argument the list of nodes to be matched, and returns a list of pairs. There is one pair for each successful match, containing the value of the match and the remaining nodes to be matched.

The `match` function applies the regular expression to the children of the given node, and returns the value of each successful match that has no remaining nodes to be matched.

```
match         :: Reg a -> Node -> [a]
match p x     =  [ y | (y,l) <- p (children x), null l ]
```

The `nodeItem` regular expression checks the list of nodes to be matched. If it is empty, the match fails (returns an empty list); otherwise, there is one successful match, the value is the node at the head of the list and the remaining list of nodes is the tail of the list.

```
nodeItem      :: Reg Node
nodeItem l    =  if null l then [] else [(head l, tail l)]
```

Alternation of regular expressions is defined in terms of append.

```
(p +++ q) l   =  p l ++ q l
```

Regular expression comprehensions are defined as follows. If `e` is an expression, `x` is a varialbe, `p` is a regular expression, and `q` is a list of qualifiers, then we have

```
([ e | x <- p, q ]) l  =  [ (y,n) | (x,m) <- p l, (y,n) <- ([ e | q ]) m ]
```

That is, first we apply regular expression `p` to the list of nodes `l`, yielding the value `x` and the remainder `m`, then we apply the remaining comprehension to the list of nodes `m`, yielding the value `y` and the remainder `n`, then we return the value `y` and the remainder `n`. If `e` is an expression, `b` is a boolean valued expression, and `q` is a list of qualifiers, then we have

```
([ e | b, q ]) l  =  if b then ([ e | q ]) l else []
```

A filter does not match any input, so if the filter is true then the list of nodes to be matched `l` is passed unchanged to the remaining comprehension, otherwise the match fails. Finally, if `e` is an expression, then we have

```
([ e | True ]) l  =  [ (e,l) ]
```

The expression is returned paired with the input.

Just as with lists, the trivial and flattening laws follow from the definition of regular expression comprehensions.

```
(trivial)     ([ x | x <- e ])                  =  e
(flattening)  ([ d | p, x <- ([ e | q ]), r ])  =  ([ d[x:=e] | p, q, r[x:=e] ])
```

Here `e` and `d` are expressions, and `p`, `q`, and `r` are lists of qualifiers. These laws can be used for optimization in much the same ways as the laws of list comprehensions.

References

[1] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[2] Francois Bancilhon, Paris Kanellakis, Claude Delobel. *Building an Object-Oriented Database System*. Morgan Kaufmann, 1990.

[3] David Beech, Ashok Malhotra, Michael Rys. A Formal Data Model and Algebra for XML. W3C XML Query working group note, September 1999.

[4] Philip Wadler. A formal semantics of patterns in XSLT. Markup Technologies, Philadelphia, December 1999.

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