An Algebra for XML Query

Mary Fernandez[1], Jerome Simeon[2], and Philip Wadler[3]

[1] ATT Labs, mff@research.att.com
[2] Bell Labs, Lucent Technologies simeon@research.bell-labs.com
[3] Avaya Labs, wadler@avaya.com

Abstract. This document proposes an algebra for XML Query. The algebra has been submitted to the W3C XML Query Working Group. A novel feature of the algebra is the use of regular-expression types, similar in power to DTDs or XML Schemas, and closely related to Hasoya, Pierce, and Vouillon's work on Xduce. The iteration construct involves novel typing rules not encountered elsewhere (even in Xduce).

1 Introduction

This document proposes an algebra for XML Query.

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

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

In the database world, it is common for a query language to exploit schemas or types; this happens in SQL, OQL, and NRA, among others. The purpose of types is twofold. Types can be used to detect certain kinds of errors at compile time and to support query optimization. DTDs and XML Schema can be thought of as providing something like types for XML. Our algebra uses a simple type system that captures the essence of XML Schema [35]. The type system is close to that used in XDuce [19]. Our type system can detect common type errors and support optimization. A novel aspect of the type system (not found in Xduce) is the description of projection in terms of iteration, and the typing rules for iteration that make this viable.

The best way to learn any language is to use it. To better familiarize readers with the algebra, we have implemented a type checker and an interpreter for the algebra in OCaml[24]. A demonstration version of the system is available at `http://www.cs.bell-labs.com/~wadler/topics/xml.html#xalgebra`

The demo system allows you to type in your own queries to be type checked and evaluated. All the examples in this paper can be executed by the demo system.

This paper describes the key features of the algebra. For simplicity, we restrict our attention to only three scalar types (strings, integers, and booleans), but we believe the system will smoothly extend to cover the continuum of scalar types found in XML Schema. Other important features that we do not tackle include attributes, namespaces, element identity, collation, and key constraints, among others. Again, we believe they can be added within the framework given here.

The paper is organized as follows. A tutorial introduction is presented in Section 2. Section 3 explains key aspects of projection and iteration. A summary of the algebra's operators and type system is given in Section 4. We present some equivalence and optimization laws of the algebra in Section 5. Finally, we give the static typing rules for the algebra in Section 6. Section 7 discusses open issues and problems.

Cited literature includes: SQL [16], OQL [4,5,13], NRA [8,15,21,22], Quilt [11], UnQL [3], XDuce [19], XML Query [33,34], XML Schema [35,36], XML-QL [17], XPath [32], XQL [25], and YaTL [14].

2 The Algebra by Example

This section introduces the main features of the algebra, using familiar examples based on accessing a database of books.

2.1 Data and Types

Consider the following sample data:

```
<bib>
  <book>
    <title>Data on the Web</title>
    <year>1999</year>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
  </book>
  <book>
    <title>XML Query</title>
    <year>2001</year>
    <author>Fernandez</author>
    <author>Suciu</author>
  </book>
</bib>
```

Here is a fragment of a XML Schema for such data.

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

<xsd:group name="Book">
  <xsd:element name="book">
    <xsd:complexType>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="year" type="xsd:integer"/>
      <xsd:element name="author" type="xsd:integer"
        minOccurs="1" maxOccurs="unbounded"/>
    </xsd:complexType>
  </xsd:element>
</xsd:group>
```

This data and schema is represented in our algebra as follows:

```
type Bib =
  bib [ Book* ]
type Book =
  book [
    title  [ String ],
    year   [ Integer ],
    author [ String ]+
  ]
let bib0 : Bib =
  bib [
    book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ],
    book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
  ]
```

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

The Bib type consists of a bib element containing zero or more value of type Book. The Book type consists of a book element containing a title element (which contains a string), a year element (which contains an integer), and one or more author elements (which contain strings).

The Bib type corresponds to a single bib element, which contains a *forest* of zero or more Book elements. We use the term forest to refer to a sequence of (zero or more) elements. Every element can be viewed as a forest of length one.

The Book type corresponds to a single book element, which contains one title element, followed by one year element, followed by one or more author elements. A title or author element contains a string value and a year element contains an integer.

The variable bib0 is bound to a literal XML value, which is the data model representation of the earlier XML document. The bib element contains two book elements.

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

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

```
    let book0 : Book =
      book [
        title  [ "Data on the Web" ],
        year   [ 1999 ],
        author [ "Abiteboul" ],
        author [ "Buneman" ],
        author [ "Suciu" ]
      ]
```

2.2 Projection

The simplest operation is projection. The algebra uses a notation similar in appearance and meaning to path navigation in XPath.

The following expression returns all author elements contained in book0:

```
    book0/author
==> author [ "Abiteboul" ],
    author [ "Buneman" ],
    author [ "Suciu" ]
:   author [ String ]+
```

The above example and the ones that follow have three parts. First is an expression in the algebra. Second, following the ==>, is the value of this expression.

Third, following the `:`, is the type of the expression, which is (of course) also a legal type for the value.

The following expression returns all `author` elements contained in `book` elements contained in `bib0`:

```
    bib0/book/author
==> author [ "Abiteboul" ],
    author [ "Buneman" ],
    author [ "Suciu" ],
    author [ "Fernandez" ],
    author [ "Suciu" ]
:   author [ String ]*
```

Note that in the result, the document order of `author` elements is preserved and that duplicate elements are also preserved.

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

```
    bib0             : Bib
    bib0/book        : Book*
    bib0/book/author : author [ String ]*
```

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

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

2.3 Iteration

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

```
    for b in bib0/book do
      book [ b/author, b/title ]
==> book [
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ],
      title  [ "Data on the Web" ]
    ],
    book [
      author [ "Fernandez" ],
      author [ "Suciu" ],
      title  [ "XML Query" ]
    ]
:   book [
      author[ String ]+,
      title[ String ]
    ]*
```

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

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

```
    bib0/book         : Book*
    b                 : Book
    b/author          : author  [ String ]+
    b/title           : title   [ String ]
```

The type system can determine that `b` is always `Book`, therefore the type of `b/author` is `author[ String ]+` and the type of `b/title` is `title[ String ]`.

In general, the value of a `for` loop is a forest. If the body of the loop itself yields a forest, then all of the forests are concatenated together. For instance, the expression:

```
    for b in bib0/book do
      b/author
```

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

Here we have explained the typing of `for` loops by example. In fact, the typing rules are rather subtle, and one of the more interesting aspects of the algebra, and will be explained further below.

2.4 Selection

Projection and `for` loops can serve as the basis for many interesting queries. The next three sections show how they provide the power for selection, quantification, join, and regrouping.

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

```
    for b in bib0/book do
      where value(b/year) <= 2000 do
          b
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   Book*
```

The `value` operator returns the scalar (i.e., string, integer, or boolean) content of an element.

An expression of the form

where e_1 do e_2

is just syntactic sugar for

if e_1 then e_2 else ()

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

According to this rule, the expression above translates to

```
    for b <- bib0/book in
      if value(b/year) < 2000 then b else ()
```

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

2.5 Quantification

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

```
    for b in bib0/book do
      for a in b/author do
        where value(a) = "Buneman" do
          b
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   Book*
```

In contrast, we can use the empty operator to find all books that have *no* author whose name is Buneman:

```
    for b in bib0/book do
      where empty(for a in b/author do
                    where value(a) = "Buneman" do
                      a) do
        b
==> book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
:   Book*
```

The empty expression checks that its argument is the empty sequence ().

We can also use the empty operator to find all books where all the authors are Buneman, by checking that there are no authors that are not Buneman:

```
    for b in bib0/book do
      where empty(for a in b/author do
                    where value(a) <> "Buneman" do
                      a) do
        b
==> ()
:   Book*
```

There are no such books, so the result is the empty sequence. Appropriate use of empty (possibly combined with not) can express universally or existentially quantified expressions.

Here is a good place to introduce the let expression, which binds a local variable to a value. Introducing local variables may improve readability. For example, the following expression is exactly equivalent to the previous one.

```
    for b in bib0/book do
      let nonbunemans = (for a in b/author do
                           where value(a) <> "Buneman" do
                             a) do
        where empty(nonbunemans) do
          b
```

Local variables can also be used to avoid repetition when the same subexpression appears more than once in a query.

2.6 Join

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

```
    type Reviews  =
      reviews [
        book [
          title  [ String ],
          review [ String ]
        ]*
      ]
    let review0 : Reviews =
      reviews [
        book [
          title  [ "XML Query" ],
          review [ "A darn fine book." ]
        ],
        book [
          title  [ "Data on the Web" ],
          review [ "This is great!" ]
        ]
      ]
```

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

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

```
    for b in bib0/book do
      for r in review0/book do
        where value(b/title) = value(r/title) do
          book [ b/title, b/author, r/review ]
==>
    book [
      title  [ "Data on the Web" ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
      review [ "A darn fine book." ]
    ],
    book [
      title  [ "XML Query" ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
      review [ "This is great!" ]
    ]
:   book [
      title [ String ],
      author [ String ]+
      review [ String ]
    ]*
```

Note that the outer-most for expression determines the order of the result. Readers familiar with optimization of relational join queries know that relational joins commute, i.e., they can be evaluated in any order. This is not true for the XML algebra: changing the order of the first two for expressions would produce different output. In Section 7, we discuss extending the algebra to support unordered forests, which would permit commutable joins.

2.7 Restructuring

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

```
    for a in distinct(bib0/book/author) do
      biblio [
        a,
        for b in bib0/book do
          for a2 in b/author do
            where value(a) = value(a2) do
              b/title
      ]
==> biblio [
      author [ "Abiteboul" ],
      title  [ "Data on the Web" ]
    ],
    biblio [
      author [ "Buneman" ],
      title  [ "Data on the Web" ]
    ],
    biblio [
      author [ "Suciu" ],
      title  [ "Data on the Web" ],
      title  [ "XML Query" ]
    ],
    biblio [
      author [ "Fernandez" ],
      title  [ "XML Query" ]
    ]
:   biblio [
      author [ String ],
      title  [ String ]*
    ]*
```

Readers may recognize this expression as a self-join of books on authors. The expression distinct(bib0/book/author) produces a forest of author elements with no duplicates. The outer for expression binds a to each author element, and the inner `for` expression selects the title of each book that has some author equal to `a`.

Here `distinct` is an example of a built-in function. It takes a forest of elements and removes duplicates.

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

2.8 Aggregation

We have already seen several several built-in functions, such as `children`, `distinct`, and `value`. In addition to these, the algebra has five built-in aggregation functions: `avg`, `count`, `max`, `min` and `sum`.

This expression selects books that have more than two authors:

```
    for b in bib0/book do
      where count(b/author) > 2 do
        b
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   Book*
```

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

2.9 Functions

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

```
    for b in bib0/book do
      where empty(for a in b/author do
                    where value(a) = "Buneman" do
                      a) do
        b
==> book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
:   Book*
```

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

```
     fun notauthor (s : String; b : Book) : Boolean =
       empty(for a in b/author do
               where value(a) = s do
                 a)
```

The query can then be re-expressed as follows.

```
    for b in bib0/book do
      where notauthor("Buneman"; b) do
        b
==> book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
:   Book*
```

We use semicolon rather than comma to separate function arguments, since comma is used to concatenate forests.

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

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

2.10 Structural Recursion

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

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

```
    type Part =
      Basic | Composite
    type Basic =
      basic [
        cost [ Integer ]
      ]
    type Composite =
      composite [
        assembly_cost [ Integer ],
        subparts [ Part+ ]
      ]
```

And here is some sample data.

```
  let part0 : Part =
    composite [
      assembly_cost [ 12 ],
      subparts [
        composite [
          assembly_cost [ 22 ],
          subparts [
            basic [ cost [ 33 ] ]
          ]
        ],
        basic [ cost [ 7 ] ]
      ]
    ]
```

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

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

```
  type Part2 =
    part [
      total_cost [ Integer ],
      subparts [ Part2* ]
    ]
```

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

```
  fun convert(p : Part) : Part2 =
    case p of
      b : basic =>
        part[
          total_cost[ value(b/cost) ],
          subparts[]
        ]
    | c : composite =>
        let s = (for q in children(c/subparts) do convert(q)) in
          part[
            total_cost[
         value(c/assembly_cost) +
              sum(for t in s/total_cost do value(t))
            ],
            subparts[ s ]
          ]
     end
```

Each branch of the case is labeled with an element name, `basic` or `composite`, and with a corresponding variable, `b` or `c`. The `case` expression checks whether the value of `p` is a `basic` or `composite` element, and evaluates the corresponding branch. If the first branch is taken then `b` is bound to the value of `p`, and the branch retuns a new part with total cost the same as the cost of `b`, and with no subparts. If the second branch is taken then `c` is bound to the value of `p`. The function is recursively applied to each of the subparts of `c`, giving a list of new subparts `s`. The branch returns a new part with total cost computed by adding the assembly cost of `c` to the sum of the total cost of each subpart in `s`, and with subparts `s`.

One might wonder why `b` and `c` are required, since they have the same value as `p`. The reason why is that `p`, `b`, and `c` have different types.

```
    p : Part
    b : Basic
    c : Composite
```

The types of `b` and `c` are more precise than the type of `p`, because which branch is taken depends upon the type of value in `p`.

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

```
    convert(part0)
==> part [
      total_cost [ 74 ],
      subparts [
        part [
          total_cost [ 55 ],
          subparts [
            part [
              total_cost [ 33 ],
              subparts []
            ]
          ]
        ],
        part [
          total_cost [ 7 ],
          subparts []
        ]
      ]
    ]
:   Part2
```

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

2.11 Processing any well-formed document

Recursive types allow us to define a type that matches any well-formed XML document. This type is called `UrTree`:

```
    type UrTree  =
        UrScalar
      | ~ [ UrTree* ]
```

Here `UrScalar` is a built-in scalar type. It stands for the most general scalar type, and all other scalar types (like `Integer` or `String`) are subtypes of it. The tilde (`~`) is used to indicate a wild-card type. In general, `~[`*t*`]` indicates the type of elements that may have any tag, but must have children of type *t*. So an `UrTree` is either an `UrScalar` or a wildcard element with zero or more children, each of which is itself an `UrTree`. In other words, any single element or scalar has type `UrTree`.

The use of `UrScalar` is a small, but necessary, extension to XML Schema, since XML Schema provides no most general scalar type. In contrast, the use of tilde is a significant extension to XML Schema, because XML Schema has no type corresponding to `~[`*t*`]`, where *t* is some type other than `UrTree*`. It is not clear that this extension is necessary, since the more restrictive expressiveness of XML Schema wildcards may be adequate. Also, note that `UrTree*` is equivalent to the `UrType` in XML Schema.

In particular, our earlier data also has type `UrTree`.

```
    book0 : UrTree
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   UrTree
```

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

As an example, we define a recursive function that converts any XML data into HTML. We first give a simplified definition of HTML.

```
    type HTML =
      ( UrScalar
      | b [ HTML ]
      | ul [ (li [ HTML ])* ]
      )*
```

An HTML body consists of a sequence of zero or more items, each of which is either: a scalar; or a b element (boldface) with HTML content; or a ul element (unordered list), where the children are li elements (list item), each of which has HTML content.

Now, here is the function that performs the conversion.

```
  fun html_of_xml( t : UrTree ) : HTML =
    case t of
      s : UrScalar =>
        s
    | e =>
        b [ name(e) ],
        ul [ for c in children(e) do li [ html_of_xml(c) ] ]
    end
```

The case expression checks whether the value of x is a subtype of UrScalar or otherwise, and evaluates the corresponding branch. If the first branch is taken, then s is bound to the value of t, which must be a scalar, and the branch returns the scalar. If the second branch is taken, then e is bound to the value of t, which must not be a scalar, and hence must be an element. The branch returns the name of the element in boldface, followed by a list containing one item for each child of the element. The function is recursively applied to get the content of each list item.

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

```
    html_of_xml(book0)
==> b [ "book" ],
    ul [
      li [ b [ "title" ],  ul [ li [ "Data on the Web" ] ] ],
      li [ b [ "year" ],   ul [ li [ 1999 ] ] ],
      li [ b [ "author" ], ul [ li [ "Abiteboul" ] ] ],
      li [ b [ "author" ], ul [ li [ "Buneman" ] ] ],
      li [ b [ "author" ], ul [ li [ "Suciu" ] ] ]
    ]
:   Html_Body
```

2.12 Top-level Queries

A query consists of a sequence of top-level expressions, or *query items*, where each query item is either a type declaration, a function declaration, a global variable declaration, or a query expression. The order of query items is immaterial; all type, function, and global variable declarations may be mutually recursive.

A query can be evaluated by the query interpreter. Each query expression is evaluated in the environment specified by all of the declarations. (Typically, all of the declarations will precede all of the query expressions, but this is not required.) We have already seen examples of type, function, and global variable declarations. An example of a query expression is:

```
query html_of_xml(book0)
```

To transform any expression into a top-level query, we simply precede the expression by the `query` keyword.

3 Projection and iteration

This section describes key aspects of projection and iteration.

3.1 Relating projection to iteration

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

For example, the expression:

```
book0/author
```

is equivalent to the expression:

```
for c in children(book0) do
  case c of
    a : author => a
  | b => ()
  end
```

Here the `children` function returns a forest consisting of the children of the element `book0`, namely, a title element, a year element, and three author elements (the order is preserved). The `for` expression binds the variable `v` successively to each of these elements. Then the `case` expression selects a branch based on the value of `v`. If it is an `author` element then the first branch is evaluated, otherwise the second branch. If the first branch is evaluated, the variable `a` is bound to the same value as `x`, then the branch returns the value of `a`. If the second branch is evaluated, the variable `b` is bound to the same value as `x`, then then branch returns `()`, the empty sequence.

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

```
bib0/book/author
```

is equivalent to the expression:

```
for c in children(bib0) do
  case c of
    b : book =>
      for d in children(b) do
        case d of
          a : author => d
        | e => ()
        end
  | f => ()
  end
```

The `for` expression iterates over all `book` elements in `bib0` and binds the variable `b` to each such element. For each element bound to `b`, the inner expression returns all the `author` elements in `b`, and the resulting forests are concatenated together in order.

In general, an expression of the form e `/` a is converted to the form

```
for $v_1$ in $e$ do
  for $v_2$ in children($v_1$) do
    case $v_2$ of
      $v_3$ : $a$ => $v_3$
    | $v_4$ => ()
    end
```

where e is an expression, a is a tag, and v_1, v_2, v_3, v_4 are fresh variables (ones that do not appear in the expression being converted).

According to this rule, the expression `bib0/book` translates to

```
for v1 in bib0 do
  for v2 in children(v1) do
    case v2 of
      v3 : book => v3
    | v4 => ()
    end
```

In Section 5 we introduce laws of the algebra, which allow us to simplify this to the previous expression

```
for v2 in children(bib0) do
  case v2 of
    v3 : book => v3
  | v4 => ()
  end
```

Similarly, the expression `bib0/book/author` translates to

```
for v5 in (for v2 in children(bib0) do
             case v2 of
               v3 : book => v3
             | v4 => ()
             end) do
  for v6 in children(v5) do
    case v6 of
      v7 : author => v7
    | v8 => ()
    end
```

Again, the laws will allow us to simplify this to the previous expression

```
for v2 in children(bib0) do
  case v2 of
    v3 : book =>
      for v6 in children(v3) do
        case c of
          v7 : author => d
        | v8 => ()
        end
  | v4 => ()
  end
```

These examples illustrate an important feature of the algebra: high-level operators may be defined in terms of low-level operators, and the low-level operators may be subject to algebraic laws that can be used to further simplify the expression.

3.2 Typing iteration

The typing of `for` loops is rather subtle. We give an intuitive explanation here, and cover the detailed typing rules in Section 6.

A *unit* type is either an element type a`[`t`]`, a wildcard type `~[`t`]`, or a scalar type s. A `for` loop

`for` v `in` e_1 `do` e_2

is typed as follows. First, one finds the type of expression e_1. Next, for each unit type in this type one assumes the variable v has the unit type and one types the body e_2. Note that this means we may type the body of e_2 several times, once for each unit type in the type of e_1. Finally, the types of the body e_2 are combined, according to how the types were combined in e_1. That is, if the type of e_1 is formed with sequencing, then sequencing is used to combine the types of e_2, and similarly for choice or repetition.

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

```
for c in children(book0) do
  case c of
    a : author => a
  | b => ()
  end
```

The type of `children(book0)` is

```
title[String], year[Integer], author[String]+
```

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

assuming `c` has type	`title[String]`	the body has type	`()`
”	`year[Integer]`	”	`()`
”	`author[String]`	”	`author[String]`

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

```
(), (), author[String]+
```

as the type of the iteration, and simplifying yields

```
author[String]+
```

as the final type.

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

```
for c in children(book0) do
  case c of
    t : title  => titl [ value(t) ]
  | y : year   => ()
  | a : author => auth [ value(a) ]
  end
```

Again, the type of `children(book0)` is

```
title[String], year[Integer], author[String]+
```

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

assuming `c` has type	`title[String]`	the body has type	`titl[String]`
”	`year[Integer]`	”	`()`
”	`author[String]`	”	`auth[String]`

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

```
titl[String], (), auth[String]+
```

as the type of the iteration, and simplifying yields

```
titl[String], auth[String]+
```

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

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

```
for p in children(part0/subparts) do
  case p of
    b : basic     =>  b
  | c : composite =>  ()
  end
```

The type of `children(part0/subparts)` is

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

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

assuming `p` has type	`Basic`	the body has type	`Basic`
”	`Composite`	”	`()`

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

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

as the type of the iteration, and simplifying yields

```
Basic*
```

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

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

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

$$a[t_1 \mid t_2] = a[t_1] \mid a[t_2]$$

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

4 Summary of the algebra

In this section, we summarize the algebra and present the grammars for expressions and types.

4.1 Expressions

Figure 1 contains the grammar for the algebra, i.e., the convenient concrete syntax in which a user may write a query. A few of these expressions can be rewritten as other expressions in a smaller *core* algebra; such reducible expressions are labeled with "*". We define the algebra's typing rules on the smaller core algebra. In Section 5, we give the laws that relate a user expression with its equivalent expression in the core algebra. Typing rules for the core algebra are defined in Section 6.

We have seen examples of most of the expressions, so we will only point out details here. We define a subset of expressions that correspond to *data values.* An expression is a data value if it consists only of scalar constant, element, sequence, and empty sequence expressions.

We have not defined the semantics of the binary operators in the algebra. It might be useful to define more than one type of equality over scalar and element values. We leave that to future work.

4.2 Types

Figure 2 contains the grammar for the algebra's type system. We have already seen many examples of types. Here, we point out some details.

Our algebra uses a simple type system that captures the essence of XML Schema [35]. The type system is close to that used in XDuce [19].

In the type system of Figure 2, a scalar type may be a `UrScalar`, `Boolean`, `Integer`, or `String`. In XML Schema, a scalar type is defined by one of fourteen primitive datatypes and a list of facets. A type hierarchy is induced between scalar types by containment of facets. The algebra's type system can be generalized to support these types without much increase in its complexity. We added `UrScalar`, because XML Schema does not support a most general scalar type.

A type is either: a type variable; a scalar type; an element type with literal tag a and content type t; a *wildcard* type with an unknown tag and content type t; a sequence of two types, a choice of two types; a repetition type; the empty sequence type; or the empty choice type.

The algebra's external type system, that is, the type definitions associated with input and output documents, is XML Schema. The internal types are in some ways more expressive than XML Schema, for example, XML Schema has no type corresponding to `Integer*` (which is required as the type of the argument to an aggregation operator like `sum` or `min` or `max`), or corresponding to `~[`t`]` where t is some type other than `UrTree*`. In general, mapping XML Schema types into internal types will not lose information, however, mapping internal types into XML Schema may lose information.

tag	a				
function	f				
variable	v				
integer	c_{int}	$::=$	$\cdots \mid -1 \mid 0 \mid 1 \mid \cdots$		
string	c_{str}	$::=$	$\texttt{""} \mid \texttt{"a"} \mid \texttt{"b"} \mid \cdots \mid \texttt{"aa"} \mid \cdots$		
boolean	c_{bool}	$::=$	$\texttt{false} \mid \texttt{true}$		
constant	c	$::=$	$c_{\text{int}} \mid c_{\text{str}} \mid c_{\text{bool}}$		
operator	op	$::=$	$\texttt{+} \mid \texttt{-} \mid \texttt{and} \mid \texttt{or}$		
		$\mid$	$\texttt{=} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>}$		
expression	e	$::=$	c	scalar constant	
		$\mid$	v	variable	
		$\mid$	$a\texttt{[}e\texttt{]}$	element	
		$\mid$	$\texttt{~}e\texttt{[}e\texttt{]}$	computed element	
		$\mid$	$e\ \texttt{,}\ e$	sequence	
		$\mid$	$\texttt{()}$	empty sequence	
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional	
		$\mid$	$\texttt{let}\ v\ \texttt{=}\ e\ \texttt{do}\ e$	local binding	
		$\mid$	$\texttt{for}\ v\ \texttt{in}\ e\ \texttt{do}\ e$	iteration	
		$\mid$	$\texttt{case}\ e\ \texttt{of}\ v\texttt{:}p\ \texttt{=>}\ e\ \texttt{\textbar}\ v\ \texttt{=>}\ e\ \texttt{end}$	case	
		$\mid$	$f\texttt{(}e\texttt{;}\ldots\texttt{;}e\texttt{)}$	function application	
		$\mid$	$e\ \texttt{:}\ t$	explicit type	
		$\mid$	$\texttt{empty(}e\texttt{)}$	emptiness predicate	
		$\mid$	$\texttt{error}$	error	
		$\mid$	$e\,\texttt{+}\,e$	plus	
		$\mid$	$e\,\texttt{=}\,e$	equal	
		$\mid$	$\texttt{children(}e\texttt{)}$	children	
		$\mid$	$\texttt{name(}e\texttt{)}$	element name	
		$\mid$	$e\ \texttt{/}\ a$	projection	$*$
		$\mid$	$\texttt{where}\ e\ \texttt{then}\ e$	conditional	$*$
		$\mid$	$\texttt{value(}e\texttt{)}$	scalar content	$*$
		$\mid$	$\texttt{let}\ v : t\ \texttt{=}\ e\ \texttt{do}\ e$	local binding	$*$
pattern	p	$::=$	a	element	
		$\mid$	$\texttt{~}$	wildcard	
		$\mid$	s	scalar	
query item	q	$::=$	$\texttt{type}\ x\ \texttt{=}\ t$	type declaration	
		$\mid$	$\texttt{fun}\ f\texttt{(}v\texttt{:}t\texttt{;}\ldots\texttt{;}v\texttt{:}t\texttt{):}t\ \texttt{=}\ e$	function declaration	
		$\mid$	$\texttt{let}\ v\ \texttt{:}\ t\ \texttt{=}\ e$	global declaration	
		$\mid$	$\texttt{query}\ e$	query expression	
data	d	$::=$	c	scalar constant	
		$\mid$	$a\texttt{[}d\texttt{]}$	element	
		$\mid$	$d\ \texttt{,}\ d$	sequence	
		$\mid$	$\texttt{()}$	empty sequence	

Fig. 1. Algebra

tag	a			
type name	x			
scalar type	s	::=	`Integer`	
		|	`String`	
		|	`Boolean`	
		|	`UrScalar`	
type	t	::=	x	type name
		|	s	scalar type
		|	$a\,\mathtt{[}t\mathtt{]}$	element
		|	$\mathtt{\tilde{}[}t\mathtt{]}$	wildcard
		|	$t\ \mathtt{,}\ t$	sequence
		|	$t\ \mathtt{\vert}\ t$	choice
		|	$t\mathtt{*}$	repetition
		|	`()`	empty sequence
		|	$\emptyset$	empty choice
unit type	u	::=	$a\,\mathtt{[}t\mathtt{]}$	element
		|	$\mathtt{\tilde{}[}t\mathtt{]}$	wildcard
		|	s	scalar type

Fig. 2. Type System

4.3 Relating values to types

Recall that *data* is the subset of expressions that consists only of scalar constant, element, sequence, and empty sequence expressions. We write $\vdash d\ :\ t$ if data d has type t. The following type rules define this relation.

$$\frac{}{\vdash c_{\text{int}}\ :\ \mathtt{Integer}}$$

$$\frac{}{\vdash c_{\text{str}}\ :\ \mathtt{String}}$$

$$\frac{}{\vdash c_{\text{bool}}\ :\ \mathtt{Boolean}}$$

$$\frac{}{\vdash c\ :\ \mathtt{UrScalar}}$$

$$\frac{\vdash d\ :\ t}{\vdash a\,\mathtt{[}d\mathtt{]}\ :\ a\,\mathtt{[}t\mathtt{]}}$$

$$\frac{\vdash d\ :\ t}{\vdash a\,\mathtt{[}d\mathtt{]}\ :\ \mathtt{\tilde{}[}t\mathtt{]}}$$

$$\frac{\vdash d_1 \ : \ t_1 \qquad \vdash d_2 \ : \ t_2}{\vdash d_1 \ \texttt{,} \ d_2 \ : \ t_1 \ \texttt{,} \ t_2}$$

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

$$\frac{\vdash d \ : \ t_1}{\vdash d \ : \ t_1 \ \texttt{|} \ t_2}$$

$$\frac{\vdash d \ : \ t_2}{\vdash d \ : \ (t_1 \ \texttt{|} \ t_2)}$$

$$\frac{\vdash d_1 \ : \ t \qquad \vdash d_2 \ : \ t\texttt{*}}{\vdash (d_1, d_2) \ : \ t\texttt{*}}$$

$$\frac{}{\vdash () \ : \ t\texttt{*}}$$

We write $t_1 <: t_2$ if for every data d such that $\vdash d \ : \ t_1$ it is also the case that $\vdash d \ : \ t_2$, that is t_1 is a subtype of t_2. It is easy to see that $<:$ is a partial order, that is it is reflexive, $t <: t$, and it is transitive, if $t_1 <: t_2$ and $t_2 <: t_3$ then $t_1 <: t_3$. We also have that $\emptyset <: t$ for any type t, and $a[t] <: \tilde{}[t]$. We have $s <: \texttt{UrScalar}$ for every scalar type s. We have $t_1 <: (t_1 \mid t_2)$ and $t_2 <: (t_1 \mid t_2)$ for any t_1 and t_2. If $t <: t'$, then $a[t] <: a[t']$ and $t* <: t'*$. And if $t_1 <: t_1'$ and $t_2 <: t_2'$ then $t_1, t_2 <: t_1', t_2'$.

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

$$\begin{array}{ll}
\texttt{UrScalar} & = \texttt{Integer} \mid \texttt{String} \mid \texttt{Boolean} \\
(t_1, t_2), t_3 & = t_1, (t_2, t_3) \\
t, () & = t \\
(), t & = t \\
t_1 \mid t_2 & = t_2 \mid t_1 \\
(t_1 \mid t_2) \mid t_3 & = t_1 \mid (t_2 \mid t_3) \\
t \mid \emptyset & = t \\
\emptyset \mid t & = t \\
t_1, (t_2 \mid t_3) & = (t_1, t_2) \mid (t_1, t_3) \\
(t_1 \mid t_2), t_3 & = (t_1, t_3) \mid (t_2, t_3) \\
t, \emptyset & = \emptyset \\
\emptyset, t & = \emptyset \\
a[t] \mid \tilde{}[t] & = \tilde{}[t] \\
t* & = () \mid t, t*
\end{array}$$

We also have that $t_1 <: t_2$ if and only iff $t_1 \mid t_2 = t_2$.

We define $t?$ and $t+$ as abbreviations, by the following equivalences.

$$t? = () \mid t$$
$$t+ = t, t*$$

```
$e/a$
  $\Rightarrow$ for $v_1$ in $e$ do                      (1)
       for $v_2$ in children($v_1$) do
         case $v_2$ of
           $v_3$ : $a$ => $v_3$
         | $v_4$ => ()

where $e_1$ then $e_2$
  $\Rightarrow$ if $e_1$ then $e_2$ else ()              (2)

value($e$)
  $\Rightarrow$ case children($e$) of                    (3)
       $v_1$ : UrScalar => $v_1$
     | $v_2$ => $v_2 : \emptyset$

let $v : t$ = $e_1$ do $e_2$
  let $v$ = ($e_1$ : $t$) do $e_2$                       (4)
```

Fig. 3. Definitions

5 Equivalences and Optimization

5.1 Equivalences

Figure 3 contains the laws that relate the reducible expressions (i.e., those labeled with "*" in Figure 1) to equivalent expressions. In these definitions, $e1\{e2/v\}$ denotes the expression $e1$ in which all occurrences of v are replaced by $e2$.

In Rule 1, the projection expression e/a is rewritten as described previously. Rule 2 rewrites a `where` expression as a conditional, as described previously. Rule 3 rewrites `value(`e`)` as a `case` expression which checks whether the content of e is a scalar value, and if so, returns it. If e is not scalar value, its value is returned with the empty choice type, which may indicate an error. Rule 4 rewrites the `let` expression with a type as a `let` expression without a type by moving the type constraint into the expression.

5.2 Optimizations

Figure 4 contains a dozen algebraic simplification laws. In a relational query engine, algebraic simplifications are often applied by a query optimizer before a physical execution plan is generated; algebraic simplification can often reduce the size of the intermediate results computed by a query interpreter. The purpose of our laws is similar – they eliminate unnecessary `for` or `case` expressions, or they enable other optimizations by reordering or distributing computations. The set of laws given is suggestive, rather than complete.

$$
\begin{aligned}
E ::= {} & \texttt{if } [\,] \texttt{ then } e_1 \texttt{ else } e_2 \\
\mid {} & \texttt{let } v \texttt{ = } [\,] \texttt{ do } e \\
\mid {} & \texttt{for } v \texttt{ in } [\,] \texttt{ do } e \\
\mid {} & \texttt{case } [\,] \texttt{ of } v_1\texttt{:}p \texttt{ => } e_1 \texttt{ | } v_2 \texttt{ => } e_2 \texttt{ end}
\end{aligned}
$$

$$\texttt{for } v \texttt{ in () do } e \Rightarrow \texttt{()} \tag{5}$$

$$
\begin{aligned}
& \texttt{for } v \texttt{ in (} e_1 \texttt{ , } e_2 \texttt{) do } e_3 \\
& \quad \Rightarrow \texttt{(for } v \texttt{ in } e_1 \texttt{ do } e_3\texttt{) , (for } v \texttt{ in } e_2 \texttt{ do } e_3\texttt{)}
\end{aligned} \tag{6}
$$

$$
\begin{aligned}
& \texttt{for } v \texttt{ in } e_1 \texttt{ do } e_2 \\
& \quad \Rightarrow e_2\big\{e_1/v\big\}, \quad \text{if } e : u
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
& \texttt{case } a\texttt{[}e_0\texttt{] of } v_1\texttt{:}a \texttt{ => } e_1 \texttt{ | } v_2 \texttt{ => } e_2 \texttt{ end} \\
& \quad \Rightarrow e_1\big\{a\texttt{[}e_0\texttt{]}/v_1\big\}
\end{aligned} \tag{8}
$$

$$
\begin{aligned}
& \texttt{case } a'\texttt{[}e_0\texttt{] of } v_1\texttt{:}a \texttt{ => } e_1 \texttt{ | } v_2 \texttt{ => } e_2 \texttt{ end} \\
& \quad \Rightarrow e_2\big\{a'\texttt{[}e_0\texttt{]}/v_2\big\}, \quad \text{if } a \neq a'
\end{aligned} \tag{9}
$$

$$\texttt{for } v \texttt{ in } e \texttt{ do } v \Rightarrow e \tag{10}$$

$$
\begin{aligned}
& E[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3] \\
& \quad \Rightarrow \texttt{if } e_1 \texttt{ then } E[e_2] \texttt{ else } E[e_3]
\end{aligned} \tag{11}
$$

$$
\begin{aligned}
& E[\texttt{let } v \texttt{ = } e_1 \texttt{ do } e_2] \\
& \quad \Rightarrow \texttt{let } v \texttt{ = } e_1 \texttt{ do } E[e_2]
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
& E[\texttt{for } v \texttt{ in } e_1 \texttt{ do } e_2] \\
& \quad \Rightarrow \texttt{for } v \texttt{ in } e_1 \texttt{ do } E[e_2]
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
& E[\texttt{case } e_0 \texttt{ of } v_1\texttt{:}p \texttt{ => } e_1 \texttt{ | } v_2 \texttt{ => } e_2 \texttt{ end}] \\
& \quad \Rightarrow \texttt{case } e_0 \texttt{ of } v_1\texttt{:}p \texttt{ => } E[e_1] \texttt{ | } v_2 \texttt{ => } E[e_2] \texttt{ end}
\end{aligned} \tag{14}
$$

Fig. 4. Optimization Laws

Rules 5, 6, and 7 simplify iterations. Rule 5 rewrites an iteration over the empty sequence as the empty sequence. Rule 6 distributes iteration through sequence: iterating over the sequence e_1 `,` e_2 is equivalent to the sequence of two iterations, one over e_1 and one over e_2. Rule 7 eliminates an iteration over a single element or scalar. If e_1 is a unit type, then e_1 can be substituted for occurrences of v in e_2.

Rules 8 and 9 eliminate trivial `case` expressions.

Rule 10 eliminates an iteration when the result expression is simply the iteration variable v.

Rules 11–16 commute expressions. Each rule actually abbreviates a number of other rules, since the *context variable* E stands for a number of different expressions. The notation $E[e]$ stands for one of the six expressions given with expression e replacing the hole $[\,]$ that appears in each of the alternatives. For instance, one of the expansions of Rule 13 is the following, when E is taken to be `for` v `in` $[\,]$ `do` e.

$$\begin{array}{l}\texttt{for}\ v_2\ \texttt{in}\ (\texttt{for}\ v_1\ \texttt{in}\ e_1\ \texttt{do}\ e_2)\ \texttt{do}\ e_3 \\ \quad \Rightarrow \texttt{for}\ v_1\ \texttt{in}\ e_1\ \texttt{do}\ (\texttt{for}\ v_2\ \texttt{in}\ e_2\ \texttt{do}\ e_3)\end{array}$$

Rules 7 and 10 together with the above expansion of Rule 13 are exactly analogous to the three monad laws used with list, bag, and set comprehensions in nested relational algebra [6, 8, 22, 21] algebra, and derived from a similar use in functional programming [28]. In effect, these three laws show that the `for` loop introduced here is the analogue of a monad for semi-structured data.

Note that the sophisticated type rule for `for` loops ensures that the left side of Rule 10 is well typed whenever the right side is. (Originally, a less sophisticated type rule was used, for which this is not the case.)

In Section 3.1 we claimed that the expression `bib0/book` translates to

```
for v1 in bib0 do
  for v2 in children(v1) do
    case v2 of
      v3 : book => v3
    | v4 => ()
    end
```

and that this simplifies to

```
for v2 in children(bib0) do
  case v2 of
    v3 : book => v3
  | v4 => ()
  end
```

We can now see that the translation happens via Rule 1, and the simplification happens via Rule 7.

In that Section, we also claimed that the expression `bib0/book/author` translates to

```
for v5 in (for v2 in children(bib0) do
             case v2 of
               v3 : book => v3
             | v4 => ()
             end) do
  for v6 in children(v5) do
    case v6 of
      v7 : author => v7
    | v8 => ()
    end
```

and that this simplifies to

```
for v2 in children(bib0) do
  case v2 of
    v3 : book =>
      for v6 in children(v3) do
        case c of
          v7 : author => d
        | v8 => ()
        end
  | v4 => ()
  end
```

We can now see that the translation happens via two applications of Rule 1, and the simplification happens via Rule 7 and the above instance of Rule 13.

To reiterate, these examples illustrate an important feature of the algebra: high-level operators may be defined in terms of low-level operators, and the low-level operators may be subject to algebraic laws that can be used to further simplify the expression.

6 Type Rules

We explain our type system in the form commonly used in the programming languages community. For a textbook introduction to type systems, see, for example, Mitchell [23].

6.1 Environments

The type rules make use of an environment that specifies the types of variables and functions. The type environment is denoted by Γ, and is composed of a comma-separated list of variable types, $v : t$ or function types, $f : (t_1; \ldots; t_n) \rightarrow t$. We retrieve type information from the environment by writing $(v : t) \in \Gamma$ to look up a variable, or by writing $(f : (t_1; \ldots; t_n) \rightarrow t) \in \Gamma$ to look up a function.

The type checking starts with an environment that contains all the types declared for functions and global variables. For instance, before typing the first query of Section 2.2, the environment contains: $\Gamma = \texttt{bib0} : \texttt{Bib}, \texttt{book0} : \texttt{Book}$. While doing the type-checking, new variables will be added in the environment. For instance, when typing the query of section 2.3, variable `b` will be typed with `Book`, and added in the environment. This will result in a new environment $\Gamma' = \Gamma, \texttt{b} : \texttt{Book}$.

6.2 Type rules

We write $\Gamma \vdash e \ : \ t$ if in environment Γ the expression e has type t.

The definition of `for` uses an auxiliary type judgement, given below, and the definition of `case` uses an auxiliary function, given below.

$$\frac{}{\Gamma \vdash c_{\text{int}} \ : \ \texttt{Integer}}$$

$$\frac{}{\Gamma \vdash c_{\text{str}} \ : \ \texttt{String}}$$

$$\frac{}{\Gamma \vdash c_{\text{bool}} \ : \ \texttt{Boolean}}$$

$$\frac{(v : t) \in \Gamma}{\Gamma \vdash v \ : \ t}$$

$$\frac{\Gamma \vdash e \ : \ t}{\Gamma \vdash a\texttt{[}e\texttt{]} \ : \ a\texttt{[}t\texttt{]}}$$

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{String} \qquad \Gamma \vdash e_2 \ : \ t}{\Gamma \vdash \texttt{\~{}}e_1\texttt{[}e_2\texttt{]} \ : \ \texttt{\~{}[}t\texttt{]}}$$

$$\frac{\Gamma \vdash e_1 \ : \ t_1 \qquad \Gamma \vdash e_2 \ : \ t_2}{\Gamma \vdash e_1 \ \texttt{,} \ e_2 \ : \ t_1 \ \texttt{,} \ t_2}$$

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

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{Boolean} \qquad \Gamma \vdash e_2 \ : \ t_2 \qquad \Gamma \vdash e_3 \ : \ t_3}{\Gamma \vdash \texttt{if} \ e_1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3 \ : \ (t_2 \mid t_3)}$$

$$\frac{\Gamma \vdash e_1 \ : \ t_1 \qquad \Gamma, v : t_1 \vdash e_2 \ : \ t_2}{\Gamma \vdash \texttt{let} \ v \ \texttt{=} \ e_1 \ \texttt{do} \ e_2 \ : \ t_2}$$

$$\frac{\Gamma \vdash e_1 \ : \ t_1 \qquad \Gamma ; \texttt{for} \ v : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{for} \ v \ \texttt{in} \ e_1 \ \texttt{do} \ e_2 \ : \ t_2}$$

$$\frac{\mathit{\Gamma} \vdash e_0 \ : \ u \qquad u' \mid t' = \mathit{split}^p(u) \qquad \mathit{\Gamma},\, v_1 : u' \vdash e_1 \ : \ t_1 \qquad \mathit{\Gamma},\, v_2 : t' \vdash e_2 \ : \ t_2}{\mathit{\Gamma} \vdash \mathtt{case}\ e_0\ \mathtt{of}\ v_1\mathtt{:}p\ \mathtt{=>}\ e_2\ \mathtt{|}\ v_2\ \mathtt{=>}\ e_3\ \mathtt{end}\ :\ (t_1 \text{ if } u' \neq \emptyset) \mid (t_2 \text{ if } t' \neq \emptyset)}$$

$$\frac{\begin{array}{c}(f : (t_1; \ldots ; t_n) \to t) \in \mathit{\Gamma} \\ \mathit{\Gamma} \vdash e_1 \ : \ t'_1 \qquad t'_1 <: t_1 \\ \ldots \\ \mathit{\Gamma} \vdash e_n \ : \ t'_n \qquad t'_n <: t_n\end{array}}{\mathit{\Gamma} \vdash f(e_1; \ldots ; e_n) \ : \ t}$$

$$\frac{\mathit{\Gamma} \vdash e \ : \ t}{\mathit{\Gamma} \vdash \mathtt{empty(}e\mathtt{)} \ : \ \mathtt{Boolean}}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathtt{error} \ : \ \emptyset}$$

$$\frac{\mathit{\Gamma} \vdash e \ : \ t' \qquad t' <: t}{\mathit{\Gamma} \vdash \mathtt{(}e \ \mathtt{:} \ t\mathtt{)} \ : \ t}$$

$$\frac{\mathit{\Gamma} \vdash e_1 \ : \ \mathtt{Integer} \qquad \mathit{\Gamma} \vdash e_2 \ : \ \mathtt{Integer}}{\mathit{\Gamma} \vdash e_1 \,\mathtt{+}\, e_2 \ : \ \mathtt{Integer}}$$

$$\frac{\mathit{\Gamma} \vdash e_1 \ : \ t_1 \qquad \mathit{\Gamma} \vdash e_2 \ : \ t_2}{\mathit{\Gamma} \vdash e_1 \,\mathtt{=}\, e_2 \ : \ \mathtt{Boolean}}$$

$$\frac{\mathit{\Gamma} \vdash e \ : \ \mathtt{Integer*}}{\mathit{\Gamma} \vdash \mathtt{sum}\ e \ : \ \mathtt{Integer}}$$

$$\frac{\mathit{\Gamma} \vdash e \ : \ t}{\mathit{\Gamma} \vdash \mathtt{count}\ e \ : \ \mathtt{Integer}}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathtt{error} \ : \ \emptyset}$$

The definition of `for` uses the following auxiliary judgement. We write $\mathit{\Gamma} \vdash v \ : \ tet'$ if in environment $\mathit{\Gamma}$ where the bound variable of an iteration v has type t_1 that the body e of the iteration hast type t_2.

$$\frac{\mathit{\Gamma},\, v : u \vdash e \ : \ t'}{\mathit{\Gamma};\, \mathtt{for}\ v : u \vdash e : t'}$$

$$\frac{}{\mathit{\Gamma};\, \mathtt{for}\ v : \mathtt{()} \vdash e : \mathtt{()}}$$

$$\frac{\mathit{\Gamma};\, \mathtt{for}\ v : t_1 \vdash e : t'_1 \qquad \mathit{\Gamma};\, \mathtt{for}\ v : t_2 \vdash e : t'_2}{\mathit{\Gamma};\, \mathtt{for}\ v : t_1 \ \mathtt{,}\ t_2 \vdash e : t'_1 \ \mathtt{,}\ t'_2}$$

$$\frac{}{\mathit{\Gamma};\ \texttt{for}\ v:\emptyset \vdash e:\emptyset}$$

$$\frac{\mathit{\Gamma};\ \texttt{for}\ v:t_1 \vdash e:t'_1 \qquad \mathit{\Gamma};\ \texttt{for}\ v:t_2 \vdash e:t'_2}{\mathit{\Gamma};\ \texttt{for}\ v:t_1\ \texttt{|}\ t_2 \vdash e:t'_1\ \texttt{|}\ t'_2}$$

$$\frac{\mathit{\Gamma};\ \texttt{for}\ v:t \vdash e:t'}{\mathit{\Gamma};\ \texttt{for}\ v:t\texttt{*} \vdash e:t'\texttt{*}}$$

To determine the types in a `case` expression, we use the function $\mathit{split}^p(t)$, where p is a pattern (either an element a, or a wildcard $\texttt{\~{}}$, or a scalar s) and t is a type. For mnemonic convenience we write $a[t'] \mid t'' = \mathit{split}^a(t)$ or $\texttt{\~{}}[t'] \mid t'' = \mathit{split}^{\texttt{\~{}}}(t)$ or $s' {<:} s \mid t' = \mathit{split}^s(t)$ but one should think of the function as returning a pair consisting of two types t and t', or in the last instance a scalar type s' and a type t'. The function $\mathit{split}^p(t)$ is undefined if type t involves sequencing, since a `case` expression acts on elements or scalars, not sequences.

$$\begin{array}{lll}
\mathit{split}^a(s) & = a[\emptyset] \mid s & \\
\mathit{split}^a(a[t]) & = a[t] \mid \emptyset & \\
\mathit{split}^a(a'[t]) & = a[\emptyset] \mid a'[t] & \text{if } a \neq a' \\
\mathit{split}^a(\texttt{\~{}}[t]) & = a[t] \mid a[t] & \\
\mathit{split}^a(t_1 \mid t_2) & = a[t'_1 \mid t'_2] \mid (t''_1 \mid t''_2) & \text{where } a[t'_i] \mid t''_i = \mathit{split}^a(t_i) \\
\mathit{split}^a(\emptyset) & = a[\emptyset] \mid \emptyset &
\end{array}$$

$$\begin{array}{lll}
\mathit{split}^{\texttt{\~{}}}(s) & = \texttt{\~{}}[\emptyset] \mid s & \\
\mathit{split}^{\texttt{\~{}}}(a[t]) & = \texttt{\~{}}[t] \mid \emptyset & \\
\mathit{split}^{\texttt{\~{}}}(\texttt{\~{}}[t]) & = \texttt{\~{}}[t] \mid \emptyset & \\
\mathit{split}^{\texttt{\~{}}}(t_1 \mid t_2) & = \texttt{\~{}}[t'_1 \mid t'_2] \mid (t''_1 \mid t''_2) & \text{where } \texttt{\~{}}[t'_i] \mid t''_i = \mathit{split}^{\texttt{\~{}}}(t_i) \\
\mathit{split}^{\texttt{\~{}}}(\emptyset) & = \texttt{\~{}}[\emptyset] \mid \emptyset &
\end{array}$$

$$\begin{array}{lll}
\mathit{split}^s(s') & = s' <: s \mid \emptyset & \text{if } s' <: s \\
 & = \emptyset <: s \mid s' & \text{otherwise} \\
\mathit{split}^s(a[t]) & = \emptyset <: s \mid a[t] & \\
\mathit{split}^s(\texttt{\~{}}[t]) & = \emptyset <: s \mid \texttt{\~{}}[t] & \\
\mathit{split}^s(t_1 \mid t_2) & = (s_1 \mid s_2) <: s \mid (t'_1 \mid t'_2) & \text{where } s_i <: s \mid t'_i = \mathit{split}^s(t_i) \\
\mathit{split}^s(\emptyset) & = \emptyset <: s \mid \emptyset &
\end{array}$$

6.3 Top-level expressions

We write $\mathit{\Gamma} \vdash q$ if in environment $\mathit{\Gamma}$ the query item q is well-typed.

$$\frac{}{\mathit{\Gamma} \vdash \texttt{type}\ x = t}$$

$$\frac{\mathit{\Gamma},\ v_1:t_1,\ \ldots,\ v_n:t_n \vdash e\ :\ t' \qquad t' <: t}{\mathit{\Gamma} \vdash f\texttt{(}v_1\texttt{:}t_1\texttt{;}\ldots\texttt{;}v_n\texttt{:}t_n\texttt{)}\texttt{:}t\ \texttt{=}\ e}$$

$$\frac{\Gamma \vdash e \ : \ t' \qquad t' <: t}{\Gamma \vdash \texttt{let}\ v\ \texttt{:}\ t\ \texttt{=}\ e}$$

$$\frac{\Gamma \vdash e \ : \ t}{\Gamma \vdash \texttt{query}\ e}$$

We extract the relevant component of a type environment from a query item q with the function *environment*(q).

$$\begin{array}{ll}
\mathit{environment}(\texttt{type}\ x\ \texttt{=}\ t) & = () \\
\mathit{environment}(\texttt{fun}\ f\,\texttt{(}v_1\texttt{:}t_1\texttt{;}\ \ldots\texttt{;}\ v_n\texttt{:}t_n\texttt{)}\texttt{:}t) & = f : (t_1;\ \ldots;\ t_n) \rightarrow t \\
\mathit{environment}(\texttt{let}\ v\ \texttt{:}\ t\ \texttt{=}\ e) & = v : t
\end{array}$$

We write $\vdash q_1 \ldots q_n$ if the sequence of query items $q_1 \ldots q_n$ is well typed.

$$\frac{\begin{array}{c}\Gamma = \mathit{environment}(q_1),\ \ldots,\ \mathit{environment}(q_n) \\ \Gamma \vdash q_1 \qquad \cdots \qquad \Gamma \vdash q_n\end{array}}{\vdash q_1 \ldots q_n}$$

7 Discussion

The algebra has several important characteristics: its operators are orthogonal, strongly typed, and they obey laws of equivalence and optimization.

There are many issues to resolve in the completion of the algebra. We enumerate some of these here.

Data Model. Currently, all forests in the data model are ordered. It may be useful to have unordered forests. The `distinct` operator, for example, produces an inherently unordered forest. Unordered forests can benefit from many optimizations for the relational algebra, such as commutable joins.

The data model and algebra do not define a global order on documents. Querying global order is often required in document-oriented queries.

Currently, the algebra does not support reference values, which are defined in the XML Query Data Model. The algebra's type system should be extended to support reference types and the data model operators `ref` and `deref` should be supported.

Type System. As discussed, the algebra's internal type system is closely related to the type system of XDuce. A potentially significant problem is that the algebra's types may lose information when converted into XML Schema types, for example, when a result is serialized into an XML document and XML Schema.

The type system is currently first order: it does not support function types nor higher-order functions. Higher-order functions are useful for specifying, for example, sorting and grouping operators, which take other functions as arguments.

The type system is currently monomorphic: it does not permit the definition of a function over generalized types. Polymorphic functions are useful for factoring equivalent functions, each of which operate on a fixed type. The lack of polymorphism is one of the principal weaknesses of the type system.

Operators. We intentionally did not define equality or relational operators on element and scalar types undefined. These operators should be defined by consensus.

It may be useful to add a fixed-point operator, which can be used in lieu of recursive functions to compute, for example, the transitive closure of a collection.

Functions. There is no explicit support for externally defined functions.

The set of builtin functions may be extended to support other important operators.

Recursion. Currently, the algebra does not guarantee termination of recursive expressions. In order to ensure termination, we might require that a recursive function take one argument that is a singleton element, and any recursive invocation should be on a descendant of that element; since any element 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.)

References

1. S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.
2. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.
3. P. Buneman, M. Fernandez, D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, to appear.
4. Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. *Theoretical Computer Science* 116(1&2):59–94, August 1993.
5. Francois Bancilhon, Paris Kanellakis, Claude Delobel. *Building an Object-Oriented Database System.* Morgan Kaufmann, 1990.
6. Peter Buneman, Leonid Libkin, Dan Suciu, Van Tannen, and Limsoon Wong. Comprehension Syntax. *SIGMOD Record*, 23:87–96, 1994.
7. David Beech, Ashok Malhotra, Michael Rys. A Formal Data Model and Algebra for XML. W3C XML Query working group note, September 1999.
8. Peter Buneman, Shamim Naqvi, Val Tannen, Limsoon Wong. Principles of programming with complex object and collection types. *Theoretical Computer Science* 149(1):3–48, 1995.
9. Catriel Beeri and Yariv Tzaban, SAL: An Algebra for Semistructured Data and XML, *International Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Pennsylvania, June 1999.
10. R. G. Cattell. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

11. Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.
12. Vassilis Christophides and Sophie Cluet and Jérôme Siméon. On Wrapping Query Languages and Efficient XML Integration. *Proceedings of ACM SIGMOD Conference on Management of Data*, Dallas, Texas, May 2000.
13. S. Cluet and G. Moerkotte. Nested queries in object bases. *Workshop on Database Programming Languages*, pages 226–242, New York, August 1993.
14. S. Cluet, S. Jacqmin and J. Siméon The New YAT$_{\text{L}}$: Design and Specifications. *Technical Report*, INRIA, 1999.
15. L. S. Colby. A recursive algebra for nested relations. *Information Systems* 15(5):567–582, 1990.
16. Hugh Darwen (Contributor) and Chris Date. *Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL* Addison-Wesley, 1997.
17. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *International World Wide Web Conference*, 1999. `http://www.research.att.com/~mff/files/final.html`
18. J. A. Goguen, J. W. Thatcher, E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology*, pages 80–149, Prentice Hall, 1978.
19. Haruio Hosoya, Benjamin Pierce, XDuce : A Typed XML Processing Language (Preliminary Report) *WebDB Workshop* 2000.
20. M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 393–402, San Diego, California, June 1992.
21. Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *Journal of Computer and Systems Sciences*, 55(2):241–272, October 1997.
22. Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. *SIGMOD* 1996.
23. John C. Mitchell *Foundations for Programming Languages.* MIT Press, 1998.
24. The Caml Language. `http://pauillac.inria.fr/caml/`.
25. J. Robie, editor. XQL '99 Proposal, 1999. `http://metalab.unc.edu/xql/xql-proposal.html`.
26. H.-J. Schek and M. H. Scholl. The relational model with relational-valued attributes. *Information Systems* 11(2):137–147, 1986.
27. S. J. Thomas and P. C. Fischer. Nested Relational Structures. In *Advances in Computing Research: The Theory of Databases*, JAI Press, London, 1986.
28. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461-493, 1992.
29. Philip Wadler. A formal semantics of patterns in XSLT. Markup Technologies, Philadelphia, December 1999.
30. Limsoon Wong. An introduction to the Kleisli query system and a commentary on the influence of functional programming on its implementation. *Journal of Functional Programming*, to appear.
31. World-Wide Web Consortium XML Query Data Model, Working Draft, May 2000. `http://www.w3.org/TR/query-datamodel`.
32. World-Wide Web Consortium, XML Path Language (XPath): Version 1.0. November, 1999. `/www.w3.org/TR/xpath.html`

33. World-Wide Web Consortium, XML Query: Requirements, Working Draft. August 2000. http://www.w3.org/TR/xmlquery-req
34. World-Wide Web Consortium, XML Query: Data Model, Working Draft. May 2000. http://www.w3.org/TR/query-datamodel/
35. World-Wide Web Consortium, XML Schema Part 1: Structures, Working Draft. April 2000. http://www.w3.org/TR/xmlschema-1
36. World-Wide Web Consortium, XML Schema Part 2: Datatypes, Working Draft, April 2000. http://www.w3.org/TR/xmlschema-2.
37. World-Wide Web Consortium, XSL Transformations (XSLT), Version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xslt.