An 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
Philip Wadler	Bell Labs, Lucent Technologies	wadler@research.bell-labs.com

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

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 [34]. The type system is close to that used in XDuce [21]. Our type system can detect common type errors and support optimization.

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, but does not address features such as attributes, element identity, namespaces, collation, and key constraints, among others. 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. After reading the tutorial, the reader will have seen the entire algebra and should be able to write example queries. A summary of the algebra's operators and type system is given in Section 3. We present some equivalence and optimization laws of the algebra in Section 4. Finally, we give the static typing rules for the algebra in Section 5. Although it contains the most challenging material, we have tried to make the content as accessible as possible. Nonetheless, this section is not necessary for understanding or using the algebra,

therefore the reader may skip this section. In Section 6, we discuss open issues and problems. In Appendix A, we give a formal mapping relating the algebra to the XML Query Data Model.

Cited literature includes: SQL [16], OQL [4, 5, 13], NRA [8, 15, 20, 22], Quilt [11], UnQL [3], XDuce [21], XML-QL[17], XPath [33], 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" maxOccurs="unbounded"/>
    </xsd:complexType>
  </xsd:element>
</xsd:group>
```

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

```
type Bib =
```

```
  bib [ Book{0,*} ]
type Book =
  book [
    title  [ String ],
    year   [ Integer ],
    author [ String ]{1,*}
  ]
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 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 `book`

elements contained in `bib0`:

```
    bib0/book/author
==> author [ "Abiteboul" ],
    author [ "Buneman" ],
    author [ "Suciu" ],
    author [ "Fernandez" ],
    author [ "Suciu" ]
:   author [ String ] {0,*}
```

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

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.

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{0,*}
bib0/book/author : author [ String ]{0,*}
```

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 author before the title, and remove the year.

```
    for b <- bib0/book in
      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 ]{1,*},
      title[ String ]
    ]{0,*}
```

The `<-` is pronounced "drawn from". 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{0,*}
b                : Book
b/author         : author [ String ]{1,*}
b/title          : title  [ String ]
```

The type system can determine that `b` is always `Book`, therefore the type of `b/author` is `author[ String ]{1,*}` and the type of `b/title` is `title[ String ]`.

2.4 Projection, revisited

The previous examples use the `/` operator liberally, but in fact we use `/` as a convenient abbreviation for expressions built from two lower-level operators, `project` and `children`, possibly combined with `for` expressions.

The `children` operator returns the children of an element. The `project` operator takes a tag name and a forest and returns a new forest containing only elements with the given tag. For example, the expression:

```
bib0/book
```

is equivalent to the expression:

```
project book (children(bib0))
```

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

```
bib0/book/author
```

is equivalent to the expression:

```
for b <- project book (children(bib0)) in
  project author (children(b))
```

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

$$\texttt{for } v \texttt{ <- } e \texttt{ in}\\ \quad \texttt{project } a \texttt{ (children(}v\texttt{))}$$

where e is an expression, a is a tag, and v is a fresh variable (one that does not appear in the expression being converted).

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

```
for v <- bib0 in
  project book (children(v))
```

In Section 4 we introduce laws of the algebra, which allow us to simplify this to the previous expression

```
project book (children(bib0))
```

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

```
for v2 <- (for v1 <- bib0 in
             project book (children(v1))) in
  project author (children(v2))
```

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

```
for b <- project book (children(bib0)) in
  project author (children(b))
```

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.

2.5 Selection

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

```
    for b <- bib0/book in
      where value(b/year) <= 2000 then
          b
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   Book{0,*}
```

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

In general, an expression of the form

$$\texttt{where}\ e_1\ \texttt{then}\ e_2$$

is converted to the form

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{()}$$

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.

2.6 Quantification

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

```
for b <- bib0/book in
  for a <- b/author in
    where value(a) = "Buneman" then
      b
```

```
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   Book{0,*}
```

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

```
    for b <- bib0/book in
      where empty(for a <- b/author in
                  where value(a) = "Buneman" then
                    a) then
        b
==> book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
:   Book{0,*}
```

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 <- bib0/book in
      where empty(for a <- b/author in
                  where value(a) <> "Buneman" then
                    a) then
        b
==> ()
:   Book{0,*}
```

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

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

2.7 Join

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

```
    type Reviews  =
      reviews [
        book [
          title  [ String ],
          review [ String ]
        ]{0,*}
      ]
    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` expressions to join the two sources `review0` and `bib0` on title values. The result combines the title, authors, and reviews for each book.

```
    for b <- bib0/book in
      for r <- review0/book in
        where value(b/title) = value(r/title) then
          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 ] {1,*},
      review [ String ]
    ] {0,*}
```

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 6, we discuss extending the algebra to support unordered forests, which would permit commutable joins.

2.8 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 <- unique(bib0/book/author) in
      biblio [
        a,
        for b <- bib0/book in
          for a2 <- b/author in
            where value(a) = value(a2) then
              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 ] {0,*}
    ] {0,*}
```

Readers may recognize this expression as a self-join of books on authors. The expression `unique(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 `unique` is an example of a built-in function. It takes a forest of elements and removes duplicates; the order of the resulting forest is not defined.

The type of the result expression may seem surprising: each `biblio` element may contain *zero* or more `title` elements, even though in `bib0`, every `author` co-occurs with a `title`. Recognizing such a constraint is outside the scope of the type system, so the resulting type is not as precise as we would like. We can, however, obtain the more precise type by re-expressing this query using the `group` operator, which is described in Section 2.11.

2.9 Querying Order

Often it is useful to query the order of elements in a forest. There are two kinds of order among elements. *Document* or *global* order refers to the total order among all elements in a document and corresponds to their sequential appearance. *Local order* refers to the order among sibling elements in a forest. Currently, the algebra supports querying of local order, but not global order. We discuss global order in Section 6.

The `index` function pairs an integer index with each element in a forest:

```
    index(book0/author)
==> pair [ fst [ 1 ], snd [ author [ "Abiteboul" ] ] ],
    pair [ fst [ 2 ], snd [ author [ "Buneman" ] ] ],
    pair [ fst [ 3 ], snd [ author [ "Suciu" ] ] ]
:   pair [ fst [ Integer ], snd [ author [ String ] ] ] {1,*}
```

Note that the result type guarantees at least one pair exists in the result, because `(book0/author)` always contains one or more authors.

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

```
    for p <- index(book0/author) in
      where (1 <= value(p/fst) and value(p/fst) <= 2) then
        p/snd/author
==> author [ "Abiteboul" ],
    author [ "Buneman" ]
:   author [ String ] {0,*}
```

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

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

2.10 Sorting

To sort a sequence of elements, we use a technique similar to that used above for `index`. Each element in a sequence can be paired with a *key* value on which the sequence is sorted. The following expression chooses a book's title as a key and pairs the key with the book's review:

```
    for b <- review0/book in
      pair [ fst [ b/title ], snd [ b ] ]
==> pair [
      fst [ "XML Query" ],
      snd [ book [ title ["XML Query" ], review [ "A darn fine book." ] ] ]
    ],
    pair [
      fst [ "Data on the Web" ],
      snd [ book [ title [ "Data on the Web" ], review [ "This is great!" ] ] ]
    ]
:   pair [
      fst [ String ],
      snd [ book [ title [String], review [String] ] ]
    ] {0,*}
```

Given a sequence of pairs, the `sort` operator sorts the pairs on the values of `fst` elements, which contains the key:

```
    let pairs = (for b <- review0/book in
                   pair [ fst [ b/title ], snd [ b ] ]) in
      sort(pairs)
==> pair [
      fst [ "Data on the Web" ],
      snd [ book [ title [ "Data on the Web" ], review [ "This is great!" ] ] ]
    ],
    pair [
      fst [ "XML Query" ],
      snd [ book [ title ["XML Query" ], review [ "A darn fine book." ] ] ]
    ]
:   pair [
      fst [ String ],
      snd [ book [ title [ String ], review [ String ] ] ]
    ] {0,*}
```

Finally, we can extract the sorted sequence by iterating over the sorted pairs and selecting out each pair's `snd` value:

```
    let pairs = (for b <- review0/book in
                   pair [ fst [ b/title ], snd [ b ] ]) in
      sort(pairs)/snd/book
==> book [ title  [ "Data on the Web" ],
           review [ "This is great!" ] ],
    book [ title  [ "XML Query" ],
           review [ "This is pretty good too!" ] ]
:   book [ title [ String ], review [ String ] ] ] {0,*}
```

This three-stage approach to sort a sequence may seem unusual and potentially inefficient. The advantage of this approach, however, is that the `sort` operator is *first order*, that is, it does not take a function as an argument. Often sort operators are *higher order*, that is, they may take as arguments a function to compare two items in a sequence and possibly, a function to extract the key value from each item. Adding higher-order functions to the algebra complicates typing so we have chosen a first-order approach. We discuss adding higher-order functions in Section 6.

2.11 Grouping

We can group elements on a key value in the same way we sort on a key value. This expression chooses a book's author as a key and pairs the key with the book's title; the `group` operator groups the pairs on the values of the `fst` elements:

```
    group(for b <- bib0/book in
            for a <- b/author in
              pair [ fst [ a ], snd [ b/title ] ])
==> pair [
      fst [ author [ "Abiteboul" ] ],
      snd [ title  [ "Data on the Web" ] ]
    ],
    pair [
      fst [ author [ "Buneman" ] ],
      snd [ title  [ "Data on the Web" ] ]
    ],
    pair [
      fst [ author [ "Suciu" ] ],
```

```
      snd [ title  [ "Data on the Web" ], title  [ "XML Query" ]  ]
    ],
    pair [
      fst [ author [ "Fernandez" ] ],
      snd [ title  [ "XML Query" ] ]
    ],
:   pair [
      fst [ author [ String ] ],
      snd [ title [ String ]{1,*} ]
    ] {0,*}
```

Note that the result type is precise: each pair is guaranteed to include one title, therefore each grouped pair contains at least one title.

This expression regroups each author with the titles of his/her publications.

```
    for p <- group(for b <- bib0/book in
                   for a <- b/author in
                     pair [ fst [ a ], snd [ b/title ] ]) in
      biblio [ p/fst/author, p/snd/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] {1,*}
    ] {0,*}
```

The reader should compare this expression with the expression in Section 2.8, which yields the same value but with a less precise type. Furthermore, the version using `group` can be implemented more efficiently.

2.12 Aggregation

We have already seen several several built-in functions: `index`, `unique`, `sort` and `group`. In addition to these functions, 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 <- bib0/book in
      where count(b/author) > 2 then
        b
```

```
==> book [
      title  [ "Data on the Web" ],
      year   [ 1999 ],
      author [ "Abiteboul" ],
      author [ "Buneman" ],
      author [ "Suciu" ]
    ]
:   Book{0,*}
```

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

2.13 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 <- bib0/book in
      where empty(for a <- b/author in
                  where value(a) = "Buneman" then
                    a) then
        b
==> book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
:   Book{0,*}
```

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 <- b/author in
               where value(a) = s then
                 a)
```

The query can then be re-expressed as follows.

```
    for b <- bib0/book in
      where notauthor("Buneman"; b) then
        b
==> book [
      title  [ "XML Query" ],
      year   [ 2001 ],
      author [ "Fernandez" ],
      author [ "Suciu" ]
    ]
:   Book{0,*}
```

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.

Functions make the algebra extensible. We have seen examples of built-in functions (`sort`, `group`, and `unique`) and examples of user-defined functions (`notauthor`). In addition to built-in and user-defined functions, the algebra could support externally defined functions, i.e., functions that are not defined in the algebra itself, but in some external language. This would make special-purpose implementations of, for example, full-text search functions available in the algebra. We discuss support for externally defined functions in Section 6.

2.14 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 [
    cost [ Integer ]
  ]
| composite [
    assembly_cost [ Integer ],
    subparts [ Part{1,*} ]
  ]
```

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{0,*} ]
  ]
```

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

```
fun convert(x : Part) : Part2 =
  case x of
    basic [ c ] =>
      let p = basic [ c ] in
        part [
          total_cost [ value(p/cost) ],
          subparts []
        ]
  | q =>
    let s = (for y <- children(q/subparts) in convert(y)) in
      part [
        total_cost [
          value(q/assembly_cost) + sum(for v <- s/total_cost in value(v))
        ],
        subparts[ s ]
      ]
```

The `case` expression checks whether the value of `x` has the form `basic[c]`, that is whether `x` is a `basic` element, and if so then `c` is bound to the children of `x`. The `let` expression binds `p` to an element `basic[c]`, so if this branch is taken then `p` is the same as `x`, but with a more precise type (it must be a basic part, not a composite). This branch returns a part with total cost the same as the cost of `x`, and with no subparts.

Otherwise, the `case` expression binds `q` to the value of `x`, so if this branch is taken, then `q` is the same as `x`, but with a more precise type (it must be a composite, not a basic part). This branch recursively applies the function to each subpart of `q`, giving a list of new subparts `s`. It returns a part containing the total cost computed by adding the assembly cost of `q` to the sum of the total cost of each new subpart in `s`, and containing the subparts `s`.

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 []
        ]
      ]
    ]
```

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

2.15 Functions for all well-formed documents

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

```
type UrTree  =
    UrScalar
  | ~ [ UrTree{0,*} ]
```

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, $\tilde{}[t]$ 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 $\tilde{}[t]$, where t is some type other than `UrTree {0,*}`. 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{0,*}` 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_body =
  ( UrScalar
  | b [ HTML_body ]
  | ul [ (li [ HTML_body ]){0,*} ]
  ) {0,*}
```

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

Now, here is the function that performs the conversion.

```
fun html_of_xml( x : UrTree ) : Html_Body =
  case x of
    s : UrScalar =>
      s
  | e =>
      b [ name(e) ],
      ul [ for y <- children(e) in li [ html_of_xml(y) ] ]
```

The `case` expression checks whether the value of `x` is a subtype of `UrScalar`, and if so then `s` is bound to that value, so if this branch is taken then `s` is the same as `x`, but with a more precise type (it must be a scalar, not an element). This branch returns the scalar.

Otherwise, the `case` expression binds `e` to the value of `x`, so if this branch is taken, then `e` is the same as `x`, but with a more precise type (it must be an element, not a scalar). This branch returns a `b` element containing the name of the element, and a `ul` element containing one `li` element for each child or the element. The function is recursively applied to get the content of the `li` element.

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
```

2.16 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 Summary of the algebra

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

3.1 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 4, 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 5.

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.

Figure 2 summarizes the built-in functions of the algebra.

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

Figure 1: Algebra

3.2 Types

Figure 3 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 [34]. The type

`avg`, `count`, `min`, `max`, `sum`	Aggregation functions
`index`	Pairs each element of a forest with integer index.
`group`	Groups `pair` elements on key in `fst` element.
`sort`	Sorts `pair` elements on key in `fst` element.
`unique`	Removes duplicates from a forest.

Figure 2: Built-In Functions

system is close to that used in XDuce [21].

In the type system of Figure 3, 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 of a type with minimum and maximum bounds; the empty sequence type; or the empty choice type.

The bounds on a repetition type will be either a natural number (that is, either a positive integer or zero) or the special value $*$, meaning unbounded. We extend arithmetic to include $*$ in the obvious way:

$$
\begin{gathered}
m + * = * + m = * \\
0 \cdot * = * \cdot 0 = 0 \\
m \cdot * = * \cdot m = *, \text{if } m \neq 0 \\
m \min * = * \min m = m \\
m \max * = * \max m = * \\
m \leq * = \text{true} \\
* < m = \text{false}
\end{gathered}
$$

For technical reasons, we allow the lower bound of a repetition to be $*$. A repetition $t\{m, n\}$ is equivalent to the empty choice $\emptyset$ if $m > n$ or if m is $*$.

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 `UrScalar {0,*}`, or corresponding to $\tilde{}[t]$ where t is some type other than `UrTree {0,*}`. In general, mapping XML Schema types into internal types will not lose information, however, mapping internal types into XML Schema may lose information.

3.3 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.

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

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

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

tag	a			
type variable	x			
scalar type	s	::=	`Integer`	
		|	`String`	
		|	`Boolean`	
		|	`UrScalar`	
type	t	::=	x	type variable
		|	s	scalar type
		|	$a[t]$	element
		|	$\tilde{}[t]$	wildcard
		|	t, t	sequence
		|	$t \mid t$	choice
		|	$t\{m, n\}$	repetition
		|	$()$	empty sequence
		|	$\emptyset$	empty choice
bound	m, n	::=	natural number or `*`	
prime type	p	::=	s	
		|	$a[t]$	
		|	$\tilde{}[t]$	
		|	$p \mid p$	
		|	$\emptyset$	

Figure 3: Type System

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

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

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

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

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

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

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

$$\frac{\vdash d_1 \ : \ t \qquad \vdash d_2 \ : \ t\{m,n\}}{\vdash (d_1, d_2) \ : \ t\{m+1, n+1\}}$$

$$\frac{\vdash d_1 \ : \ t \qquad \vdash d_2 \ : \ t\{0,n\}}{\vdash (d_1, d_2) \ : \ t\{0, n+1\}}$$

$$\frac{}{\vdash () \ : \ t\{0,n\}}$$

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 $a[t] <: a[t']$. If $t <: t'$ and $m \geq m'$ and $n \leq n'$ then $t\{m,n\} <: t\{m',n'\}$. 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.

$$\begin{array}{lcl}
\texttt{UrScalar} & = & \texttt{Integer} \mid \texttt{String} \mid \texttt{Boolean} \\
(t_1, t_2), t_3 & = & t_1, (t_2, t_3) \\
t, () & = & t \\
(), t & = & t \\
t_1 \mid t_2 & = & t_2 \mid t_1 \\
(t_1 \mid t_2) \mid t_3 & = & t_1 \mid (t_2 \mid t_3) \\
t \mid \emptyset & = & t \\
\emptyset \mid t & = & t \\
t_1, (t_2 \mid t_3) & = & (t_1, t_2) \mid (t_1, t_3) \\
(t_1 \mid t_2), t_3 & = & (t_1, t_3) \mid (t_2, t_3) \\
t, \emptyset & = & \emptyset \\
\emptyset, t & = & \emptyset \\
a[t_1 \mid t_2] & = & a[t_1] \mid a[t_2] \\
\tilde{}[t_1 \mid t_2] & = & \tilde{}[t_1] \mid \tilde{}[t_2] \\
a[t] \mid \tilde{}[t] & = & \tilde{}[t] \\
t\{m+1, n+1\} & = & t, (t\{m,n\}) \\
t\{0, n+1\} & = & () \mid t, (t\{0,n\}) \\
t\{0,0\} & = & () \\
t\{m,n\} & = & \emptyset, \quad \text{if } m > n \text{ or } m = *
\end{array}$$

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

4 Equivalences and Optimization

4.1 Equivalences

Figure 4 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 into a `for` expression, which binds v to each element in the forest e, and returns the children elements of v that have tag a. Rule 2 rewrites `value`(e) as a `case` expression, which returns e's children if e contains a singleton scalar value. If e is a scalar

$$
\begin{array}{rcll}
e/a & \Rightarrow & \texttt{for}\ v\ \texttt{<-}\ e\ \texttt{in project}\ a\ (\texttt{children}(v)) & (1)\\
\texttt{value}(e) & \Rightarrow & \texttt{case}\ e\ \texttt{of} & (2)\\
 & & \quad v_1[v_2]\ \texttt{=>}\ \texttt{case}\ v_2\ \texttt{of} & \\
 & & \qquad\qquad v_3 : \texttt{UrScalar}\ \texttt{=>}\ v_3 & \\
 & & \qquad\quad \texttt{|}\quad v_4\ \texttt{=>}\ v_4 : \emptyset & \\
 & & \quad \texttt{|}\quad v_5\ \texttt{=>}\ v_5 : \emptyset & \\
\texttt{children}(e) & \Rightarrow & \texttt{case}\ e\ \texttt{of}\ \tilde{}v_1[v_2]\ \texttt{=>}\ v_2\ \texttt{|}\ v_3\ \texttt{=>}\ v_3 : \emptyset & (3)\\
\texttt{name}(e) & \Rightarrow & \texttt{case}\ e\ \texttt{of}\ \tilde{}v_1[v_2]\ \texttt{=>}\ v_1\ \texttt{|}\ v_3\ \texttt{=>}\ v_3 : \emptyset & (4)\\
\texttt{where}\ e_1\ \texttt{then}\ e_2 & \Rightarrow & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ () & (5)\\
\texttt{let}\ v : t\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \Rightarrow & \texttt{let}\ v\ \texttt{=}\ (e_1 : t)\ \texttt{in}\ e_2 & (6)
\end{array}
$$

Figure 4: Definitions

value, its value is returned with the empty choice type, which may indicate an error. Similarly, if e's child is not a singleton scalar value, its children are returned with the empty choice type. Rule 3 rewrites the `children` expression as a `case` expression that matches any singleton element and return the element's children bound to v_2. Rule 4 is similar, but it returns the singleton's tag name. In Rules 3 and 4, if e is not a singleton element, a value with the empty choice type is returned. Rule 5 simply rewrites a `where` expression by an `if` − `then` − `else` expression. Rule 6 rewrite the `let` expression with a type as a `let` expression without a type by moving the type constraint into the expression.

4.2 Optimizations

Figure 5 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.

Rules 7, 8, and 9 simplify iterations. Rule 7 rewrites an iteration over the empty sequence as the empty sequence. Rule 8 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 9 eliminates an iteration over a single element or scalar. If e_1 is a prime type, then e_1 can be substituted for occurrences of v in e_2.

Rules 10 and 11 eliminate trivial `case` expressions.

Rule 12 eliminates an iteration when the result expression is simply the iteration variable v.

Rules 13–18 are used to commute expressions. Each rule actually abbreviates a number of other rules, since the *context variable* E stands for a number of different expressions. The notation $E[e]$ stands for one of the six expressions given with expression e replacing the hole [] that appears in each of the alternatives. For instance, one of the expansions of Rule 15 is the following, when E is taken to be `for` v `<-` [] `in` e.

$$\texttt{for}\ v_2\ \texttt{<-}\ (\texttt{for}\ v_1\ \texttt{<-}\ e_1\ \texttt{in}\ e_2)\ \texttt{in}\ e_3 \quad\Rightarrow\quad \texttt{for}\ v_1\ \texttt{<-}\ e_1\ \texttt{in}\ (\texttt{for}\ v_2\ \texttt{<-}\ e_2\ \texttt{in}\ e_3)$$

In Section 2.4 we claimed that the expression `bib0/book` translates to

$$
\begin{array}{rcl}
E & ::= & \texttt{if [] then } e_1 \texttt{ else } e_2 \\
& | & \texttt{let } v \texttt{ = [] in } e \\
& | & \texttt{for } v \texttt{ <- [] in } e \\
& | & \texttt{case [] of } a[v_1] \texttt{ => } e_1 \texttt{ | } v_2 \texttt{ => } e_2 \\
& | & \texttt{case [] of } \tilde{}v_1[v_2] \texttt{ => } e_1 \texttt{ | } v_3 \texttt{ => } e_2 \\
& | & \texttt{case [] of } v_1 : s \texttt{ => } e_1 \texttt{ | } v_2 \texttt{ => } e_2
\end{array}
$$

$$
\begin{array}{rcll}
\texttt{for } v \texttt{ <- } () \texttt{ in } e & \Rightarrow & () & (7) \\
\texttt{for } v \texttt{ <- } (e_1, e_2) \texttt{ in } e_3 & \Rightarrow & (\texttt{for } v \texttt{ <- } e_1 \texttt{ in } e_3), (\texttt{for } v \texttt{ <- } e_2 \texttt{ in } e_3) & (8) \\
\texttt{for } v \texttt{ <- } e_1 \texttt{ in } e_2 & \Rightarrow & e_2\{e_1/v\}, \quad \text{if } e : p & (9) \\
\texttt{case } a[e_1] \texttt{ of } a[v_1] \texttt{ => } e_2 \texttt{ | } v_2 \texttt{ => } e_3 & \Rightarrow & e_2\{e_1/v_1\} & (10) \\
\texttt{case } a'[e_1] \texttt{ of } a[v_1] \texttt{ => } e_2 \texttt{ | } v_2 \texttt{ => } e_3 & \Rightarrow & e_3\{a'[e_1]/v_2\}, \quad \text{if } a \neq a' & (11) \\
\texttt{for } v \texttt{ <- } e \texttt{ in } v & \Rightarrow & e & (12) \\
E[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3] & \Rightarrow & \texttt{if } e_1 \texttt{ then } E[e_2] \texttt{ else } E[e_3] & (13) \\
E[\texttt{let } v \texttt{ = } e_1 \texttt{ in } e_2] & \Rightarrow & \texttt{let } v \texttt{ = } e_1 \texttt{ in } E[e_2] & (14) \\
E[\texttt{for } v \texttt{ <- } e_1 \texttt{ in } e_2] & \Rightarrow & \texttt{for } v \texttt{ <- } e_1 \texttt{ in } E[e_2] & (15) \\
E[\texttt{case } e_1 \texttt{ of } a[v_1] \texttt{ => } e_2 \texttt{ | } v_2 \texttt{ => } e_3] & \Rightarrow & \texttt{case } e_1 \texttt{ of } a[v_1] \texttt{ => } E[e_2] \texttt{ | } v_2 \texttt{ => } E[e_3] & (16) \\
E[\texttt{case } e_1 \texttt{ of } \tilde{}v_1[v_2] \texttt{ => } e_2 \texttt{ | } v_3 \texttt{ => } e_3] & \Rightarrow & \texttt{case } e_1 \texttt{ of } \tilde{}v_1[v_2] \texttt{ => } E[e_2] \texttt{ | } v_3 \texttt{ => } E[e_3] & (17) \\
E[\texttt{case } e_1 \texttt{ of } v_1 : s \texttt{ => } e_2 \texttt{ | } v_2 \texttt{ => } e_3] & \Rightarrow & \texttt{case } e_1 \texttt{ of } v_1 : s \texttt{ => } E[e_2] \texttt{ | } v_2 \texttt{ => } E[e_3] & (18)
\end{array}
$$

Figure 5: Optimization Laws

```
for v <- bib0 in
  project book (children(v))
```

and that this simplifies to

```
project book (children(bib0))
```

We can now see that the translation happens via Rule 1, and the simplification happens via Rule 9.

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

```
for v2 <- (for v1 <- bib0 in
             project book (children(v1))) in
  project author (children(v2))
```

and that this simplifies to

```
for b <- project book (children(bib0)) in
  project author (children(b))
```

We can now see that the translation happens via two applications of Rule 1, and the simplification happens via Rule 9 and the above instance of Rule 15.

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.

5 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].

5.1 Prime types and factoring

We begin by explaining *prime types* and *factoring*, which are important to the type system.

Many of the operations in the algebra have an operator that should be of a repetition type. This is true for the built-in operators `index`, `sort`, `unique`, and `group`, and it is also true for the expression iterated over in a `for` expression.

For example, consider the following.

```
    index (children (book0))
==> pair [ fst [ 1 ], snd [ title [ "Data on the Web" ] ],
    pair [ fst [ 2 ], snd [ author [ "Abiteboul" ] ] ],
    pair [ fst [ 3 ], snd [ author [ "Buneman" ] ] ],
    pair [ fst [ 4 ], snd [ author [ "Suciu" ] ] ],
    pair [ fst [ 5 ], snd [ year [ 1999 ] ]
```

How should we describe the type of the result? What we need to do is find p, m, and n such that

$$\texttt{children(book0)} : \; p\{m,n\}$$

and then the type is given by

$$\texttt{index(children(book0))} : \; \texttt{pair [fst [String], snd [} p \texttt{]] } \{m,n\}$$

In the case of books, the values of p are:

```
title [ String ] | author [ String ] | year [ Integer ]
```

the value of m is 3 (because there will be one title, at least one author, and one year element) and the value of n is * (because there may be any number of author elements).

We call a type like p a *prime* type. It general, it may contain scalar, element, wildcard, choice, and empty choice types, but it will not contain repetition, sequence, or empty sequence types (except, perhaps, within an element or wildcard type). The definition of prime types appears in Figure 3.

We want to be able to convert any type t to a type of the form $p\{m,n\}$, where $t <: p\{m,n\}$, so that any value that has type t also has type $p\{m,n\}$. This is done by the function *factor*, shown in Figure 6. For example,

$$\begin{aligned} &\textit{factor}(\texttt{ title[String], author[String]\{1,*\}, year[Integer] }) \\ &\quad = \texttt{(title[String] | author[String] | year[Integer])\{3,*\}} \end{aligned}$$

For mnemonic convenience we write $p\{m,n\} = \textit{factor}(t)$, but one should actually think of the function as returning a triple consisting of a prime type p and two bounds m and n.

Just as factoring a number yields a product of prime numbers, factoring a type yields a repetition of prime types. Further, the result yielded by factoring is in some sense optimal. If $p\{m,n\} = \textit{factor}(t)$

$$
\begin{array}{lcll}
\mathit{factor}(s) & = & s\{1,1\} & \\
\mathit{factor}(a[t]) & = & a[t]\{1,1\} & \\
\mathit{factor}(\tilde{}[t]) & = & \tilde{}[t]\{1,1\} & \\
\mathit{factor}(t_1, t_2) & = & (p_1 \mid p_2)\{m_1 + m_2, n_1 + n_2\} & \text{where } p_i\{m_i, n_i\} = \mathit{factor}(t_i) \\
\mathit{factor}(t_1 \mid t_2) & = & (p_1 \mid p_2)\{m_1 \min m_2, n_1 \max n_2\} & \text{where } p_i\{m_i, n_i\} = \mathit{factor}(t_i) \\
\mathit{factor}(t\{m,n\}) & = & p\{m' \cdot m, n' \cdot n\} & \text{where } p\{m', n'\} = \mathit{factor}(t) \\
\mathit{factor}(()) & = & \emptyset\{0,0\} & \\
\mathit{factor}(\emptyset) & = & \emptyset\{*,0\} & \\
 & & & \\
\mathit{project}_a(s) & = & \emptyset & \\
\mathit{project}_a(a[t]) & = & a[t]\{1,1\} & \\
\mathit{project}_a(a'[t]) & = & \emptyset\{0,0\} & \text{if } a \neq a' \\
\mathit{project}_a(\tilde{}[t]) & = & a[t]\{1,1\} & \\
\mathit{project}_a(t_1, t_2) & = & (p_1 \mid p_2)\{m_1 + m_2, n_1 + n_2\} & \text{where } p_i\{m_i, n_i\} = \mathit{project}_a(t_i) \\
\mathit{project}_a(t_1 \mid t_2) & = & (p_1 \mid p_2)\{m_1 \min m_2, n_1 \max n_2\} & \text{where } p_i\{m_i, n_i\} = \mathit{project}_a(t_i) \\
\mathit{project}_a(t\{m,n\}) & = & p\{m' \cdot m, n' \cdot n\} & \text{where } p\{m', n'\} = \mathit{project}_a(t) \\
\mathit{project}_a(()) & = & \emptyset\{0,0\} & \\
\mathit{project}_a(\emptyset) & = & \emptyset\{*,0\} & \\
 & & & \\
\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(\tilde{}[t]) & = & a[t] \mid a[t] & \\
\mathit{split}_a(p_1 \mid p_2) & = & a[t_1 \mid t_2] \mid (p_1' \mid p_2') & \text{where } a[t_i] \mid p_i' = \mathit{split}_a(p_i) \\
\mathit{split}_a(\emptyset) & = & a[\emptyset] \mid \emptyset & \\
 & & & \\
\mathit{split}\text{-}(s) & = & \tilde{}[\emptyset] \mid s & \\
\mathit{split}\text{-}(a[t]) & = & \tilde{}[t] \mid \emptyset & \\
\mathit{split}\text{-}(\tilde{}[t]) & = & \tilde{}[t] \mid \emptyset & \\
\mathit{split}\text{-}(p_1 \mid p_2) & = & \tilde{}[t_1 \mid t_2] \mid (p_1' \mid p_2') & \text{where } \tilde{}[t_i] \mid p_i' = \mathit{split}\text{-}(p_i) \\
\mathit{split}\text{-}(\emptyset) & = & \tilde{}[\emptyset] \mid \emptyset & \\
 & & & \\
\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(\tilde{}[t]) & = & \emptyset <: s \mid \tilde{}[t] & \\
\mathit{split}_s(p_1 \mid p_2) & = & (s_1 \mid s_2) <: s \mid (p_1' \mid p_2') & \text{where } s_i <: s \mid p_i' = \mathit{split}_s(p_i) \\
\mathit{split}_s(\emptyset) & = & \emptyset <: s \mid \emptyset &
\end{array}
$$

Figure 6: Definitions of auxiliary functions

then $t <: p\{m,n\}$ and furthermore for any p', m', and n' such that $t <: p'\{m',n'\}$ we have that $p <: p'$ and $m \geq m'$ and $n \leq n'$. Also, if $t = t'$, then $\mathit{factor}(t) = \mathit{factor}(t')$. In particular, the choice of the lower bound $*$ for $\mathit{factor}(\emptyset)$ guarantees that $\mathit{factor}(t) = \mathit{factor}(t \mid \emptyset)$, since $m \min * = m$.

Several other useful auxiliary functions are also shown in Figure 6.

To determine the type of a `project` expression, we use the function $\mathit{project}_a(t)$, which is similar to *factor*, except it removes all elements with a label different from a. For mnemonic convenience we write $p\{m,n\} = \mathit{project}_a(t)$, but one should think of the function as returning a triple consisting of a prime

type p and two bounds m and n.

To determine the types in a `case` expression, we use the function $split_a(p)$ or $split\text{-}(p)$ or $split_s(p)$, depending on whether the form matched is $a[v]$ or $\tilde{}v_1[v_2]$ or $v : s$. For mnemonic convenience we write $a[t] \mid p' = split_a(p)$ or $\tilde{}[t] \mid p' = split\text{-}(p)$ or $s' <: s \mid p' = split_s(p)$ but one should think of the function as returning a pair consisting of a type t and a prime type p', or in the last instance a scalar type s' and a prime type p'.

5.2 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, $f : (t_1; \ldots; t_n) \to t$. We retrieve type information from the environment by writing $(v : t) \in \Gamma$ to look up a variable, or by writing $(f : (t_1; \ldots; t_n) \to t) \in \Gamma$ 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: Γ = `bib0 : Bib, book0 : 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 $\Gamma' = \Gamma, \texttt{b} : \texttt{Book}$.

5.3 Type rules

***** The following would benefit from a more detailed explanation! *****

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e \ : \ t \qquad p\{m,n\} = project_a(t)}{\Gamma \vdash \texttt{project}\ a\ e \ : \ p\{m,n\}}$$

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

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

$$\frac{\Gamma \vdash e_1 \ : \ t_1 \qquad p_1\{m_1, n_1\} = \mathit{factor}(t_1) \qquad \Gamma, v : p \vdash e_2 \ : \ t_2 \qquad p_2\{m_2, n_2\} = \mathit{factor}(t_2)}{\Gamma \vdash \texttt{for}\ v\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 \ : \ p_2\{m_1 \cdot m_2, n_1 \cdot n_2\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \ : \ t_1 \qquad p\{1,1\} = \mathit{factor}(t_1) \qquad a[t] \mid p' = \mathit{split}_a(p) \\ \Gamma, v_1 : t \vdash e_2 \ : \ t_2 \qquad \Gamma, v_2 : p' \vdash e_3 \ : \ t_3\end{array}}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ a[v_1]\ \texttt{=>}\ e_2\ \texttt{|}\ v_2\ \texttt{=>}\ e_3 \ : \ t_2 \mid t_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \ : \ t_1 \qquad p\{1,1\} = \mathit{factor}(t_1) \qquad \tilde{}[t] \mid p' = \mathit{split}\text{-}(p) \\ \Gamma, v_1 : \texttt{String}, v_2 : t \vdash e_2 \ : \ t_2 \qquad \Gamma, v_3 : p' \vdash e_3 \ : \ t_3\end{array}}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ \tilde{}v_1[v_2]\ \texttt{=>}\ e_2\ \texttt{|}\ v_3\ \texttt{=>}\ e_3 \ : \ t_2 \mid t_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \ : \ t_1 \qquad p\{1,1\} = \mathit{factor}(t_1) \qquad s' <: s \mid p' = \mathit{split}_s(p) \\ \Gamma, v_1 : s' \vdash e_2 \ : \ t_2 \qquad \Gamma, v_2 : p' \vdash e_3 \ : \ t_3\end{array}}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ v_1 : s\ \texttt{=>}\ e_2\ \texttt{|}\ v_2\ \texttt{=>}\ e_3 \ : \ t_2 \mid t_3}$$

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

$$\frac{\Gamma \vdash e \ : \ t}{\Gamma \vdash \texttt{empty}\ e \ : \ \texttt{Boolean}}$$

$$\frac{}{\Gamma \vdash \texttt{error} \ : \ \emptyset}$$

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

5.4 Operators and built-in Functions

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{Integer} \qquad \Gamma \vdash e_2 \ : \ \texttt{Integer}}{\Gamma \vdash e_1\ id{+}\ e_2 \ : \ \texttt{Integer}}$$

$$\frac{\Gamma \vdash e_1 \ : \ t_1 \qquad \Gamma \vdash e_2 \ : \ t_2}{\Gamma \vdash e_1\ id{=}\ e_2 \ : \ \texttt{Boolean}}$$

$$\frac{\Gamma \vdash e \ : \ t \qquad p\{m,n\} = \mathit{factor}(t)}{\Gamma \vdash \texttt{index}\ e \ : \ \texttt{pair}\ [\texttt{fst}\ [\texttt{Integer}], \texttt{snd}\ [p]]\{m,n\}}$$

$$\frac{\Gamma \vdash e \ : \ \texttt{pair}\ [\texttt{fst}\ [t_1], \texttt{snd}\ [t_2]]\{m,n\}}{\Gamma \vdash \texttt{sort}\ e \ : \ \texttt{pair}\ [\texttt{fst}\ [t_1], \texttt{snd}\ [t_2]]\{m,n\}}$$

$$\frac{\Gamma \vdash e \ : \ \texttt{pair}\ [\texttt{fst}\ [t_1], \texttt{snd}\ [t_2]]\{m,n\}}{\Gamma \vdash \texttt{group}\ e \ : \ \texttt{pair}\ [\texttt{fst}\ [t_1], \texttt{snd}\ [t_2\{1,n\}]]\{m \min 1, n\}}$$

$$\frac{\Gamma \vdash e \ : \ t \qquad \texttt{Integer}\{m,n\} = \mathit{factor}(t)}{\Gamma \vdash \texttt{agg}\ e \ : \ \texttt{Integer}}$$

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

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

$$\frac{\Gamma \vdash e \ : \ t \qquad p\{m,n\} = \mathit{factor}(t)}{\Gamma \vdash \texttt{unique}\ e \ : \ p\{m \min 1, n\}}$$

$$\frac{}{\Gamma \vdash \texttt{error} \ : \ \emptyset}$$

5.5 Top-level expressions

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

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

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

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

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

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

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

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

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

6 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 `unique` 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 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.

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$_{\mathrm{L}}$: 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 J. 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] 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.

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

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

[22] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multi-dimensional 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 XSL Transformations (XSLT), Version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xslt.

[32] World-Wide Web Consortium XML Query Data Model, Working Draft, May 2000. http://www.w3.org/TR/query-datamodel.

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

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

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

A XML Query Data Model

The XML algebra uses the XML Query data model [32]. Because the XML examples in this document do not use many of the features of the data model, such as attributes, declared namespaces, comments, the algebra uses an abbreviated notation. Here, we give the mapping from the abbreviated algebraic expressions to complete expressions in the XML Query Data Model:

```
s         = ref (stringValue (s, ref Def_string, ref []));
i         = ref (integerValue(i, ref Def_integer));
a[ kids ] = ref (elemNode (qnameValue (null, a, ref Def_string), {}, {}, kids, (ref Def_t)));
```

For example, the string `s` constructs a string value and the integer `i` constructs an integer value. The term `a[ kids ]` constructs an `ElemNode` with the tag `a`, an empty set of namespaces, an empty set of attributes, the forest of children nodes `kids`, and the XML Schema type `t`, where `t` is the type of the expression `a[ kids ]`.

Again, to simplify presentation, types do not include attributes, but they can be added easily. Subtyping or containment relationships exist between types. We discuss subtyping in Section 5.