Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Leonid Novak, Alexandre Zamulin

AN XML ALGEBRA FOR XQUERY
(PRELIMINARY COMMUNICATION)

Preprint
125

Novosibirsk 2005

An XML algebra supporting the query language XQuery is presented. The algebra is in fact a number of expression constructing operators algebraically defined. The introduction of expression constructing operators instead of high-order operations using functions as parameters has permitted us to remain in the limits of first-order structures whose instance a many-sorted algebra is. The set of operators of the presented algebra substantially differs from the set of operators of relation algebra. The difference is caused by the more complex structure of the XML document compared to the relation. In fact, only selection by predicate test is more or less the same in both algebra. At the same time, the XML algebra in addition permits selection by node test. The projection operator of relation algebra is replaced by the path expression and a number of navigating functions permitting selection of different parts of the document tree. The join operator is replaced by a number of unnesting join expressions permitting creation of a stream of flat tuples on the base of several possibly nested parts of the document tree. In addition, a number of node constructing expressions permitting update of the current algebra by introduction of new are defined.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Леонид Новак, Александр Замулин

XML-АЛГЕБРА ДЛЯ ЯЗЫКА ЗАПРОСОВ XQUERY

Препринт
125

Новосибирск 2005

Предложена XML-алгебра, поддерживающая язык запросов Xquery и представляющая собой ряд операторов конструирования выражений. Введение таких операторов вместо операций высокого уровня, использующих функции в качестве аргументов, позволило остаться в рамках структур первого порядка, примером которых являются многоосновные алгебры. Предложенный набор операторов существенно отличается по составу от набора операторов реляционной алгебры. Различие объясняется более сложной структурой XML-документа по сравнению с отношением. Фактически только выбор по предикату похож на соответствующую операцию реляционной алгебры, но в то же время имеется возможность выбора узлов дерева по их типу. Операция проекции заменена на путевое выражение и ряд навигационных функций, позволяющих выбирать различные части дерева документа. Операция соединения заменена на выражение раскрывающего соединения, позволяющего сформировать поток плоских кортежей на базе нескольких, возможно вложенных друг в друга частей дерева документа. В дополнение ко всему этому определен ряд конструирующих выражений, служащих для создания новых узлов дерева.

1. INTRODUCTION

A formal model of the database state corresponding to the XQuery 1.0 and XPath 2.0 data model [25] and consisting of document trees defined by XML Schema [22] has been presented in [12, 13]. This model regards the database state as a many-sorted algebra composed of the states of individual nodes representing information items of a document. However, no algebra resembling relation algebra for this date model is proposed in the paper, and elaboration of such an algebra supporting the XQuery language has been proclaimed as a subject of further research.

It should be noted that a number of papers proposing different XML algebras have been published [2, 6, 8, 7, 9, 14, 18, 21, 27]. Their typical flaws are:

- use of an artificial data model suitable to the authors,
- meaning by algebra something different from what is meant by algebra in mathematics,
- use of functions and predicates as operation arguments, while the algebra is a first-order structure;
- ignoring the fact that the result of a query may belong to an algebra different from the algebra of the query arguments;
- informal description of the algebra ignoring significant details of operations.

One of the aims of this paper is to propose an XML algebra that is free of the above flaws. Another aim is to elaborate such an XML algebra that could support the XQuery language [23], which is a de-facto standard of an XML query language.

Not all features of XQuery are taken into account in the algebra proposed in the paper. We consider that an XQuery interpreter should exist whose task is to interpret an XQuery query in terms of the algebra while performing the work corresponding to the following XQuery features:

- atomization,
- computation of Effective Boolean Value,
- evaluation of branching (conditional and type switch) expressions,
- evaluation of type-checking expressions (instance of, cast, treat),
- evaluation of content expressions of node constructors.

Specification of such an interpreter is not a subject of this paper.

*The work is supported in part by Russian Foundation for Basic Research under Grant 04-01-00272.

The remainder of the paper is organized as follows. A brief review of the XML data model presented in [12] is given in Section 2. Basic notions of signatures and expressions are introduced in Section 3. An example database schema used for illustration of XML algebra operations is given in Section 4. Navigating functions used for traversing a document tree are defined in Section 5. Several forms of querying expressions are formally described in Section 6. Different kinds of node constructors changing the database state are defined in Section 7. Related work is reviewed in Section 8 and some conclusions are drawn in Section 9.

2. MAIN COMPONENTS OF THE XML DATABASE MODEL

The data model presented in [12] is described by means of many-sorted algebras [4].

Definition. A many-sorted signature Σ is a pair (T, F), where T is a set of *sorts* and F a set of *operators* indexed by their *profiles*. An operator is either a symbol or a name, and a profile is either an element of T or $t_1, ..., t_n \rightarrow t_{n+1}$, where t_i is an element of T.

Definition. A many-sorted algebra $\mathtt{A}$ of signature $\Sigma = (T, F)$ is constructed by associating

- a set with each element $t \in T$, denoted by $\mathtt{A_t}$ in the sequel;
- an element $\mathtt{c^A} \in \mathtt{A_t}$ with each operator c indexed by the profile t;
- a function $\mathtt{f^A} : \mathtt{A_{t_1}} \times ... \times \mathtt{A_{t_n}} \rightarrow \mathtt{A_{t_{n+1}}}$ with each operator f indexed by the profile $t_1, ..., t_n \rightarrow t_{n+1}$.

The family of sets associated with the signature sorts in algebra $\mathtt{A}$ is called the *algebra carrier* and denoted by $\mathtt{|A|}$. An algebra of signature Σ is called a Σ-algebra.

Since the XML database model presented in [25] intensively uses the notion of *type*, we consider that the set T consists of type names and the set F of operators defined for each type. The function associated with an operator is often called an *operation*. The type system of the model includes a number of atomic types (*xs:Boolean*, *xs:Integer*, etc.), defining atomic values, and the type *xdt:untypedAtomic* denoting atomic data, such as text that has not been assigned a more specific type. It is assumed that each atomic type is equipped with conventional operations. The type *untypedAtomic* does not possess operations other than the partial functions converting (casting) values of this type into the values of the other atomic types. The type system also includes the set type constructor $Set(t)$, the sequence type constructor $Seq(t)$, the union type constructor $Union(t_1, ..., t_n)$, where $t, t_1, ..., t_n$

are types, and the enumeration type constructor $Enumeration(I_1, ..., I_n)$, where $I_1, ..., I_n$ are identifiers.

The following operations are applicable to all sets: "$\cup$" (union), "$\cap$" (intersection), "$\subset$" (inclusion), "$\in$" (membership), and *count* (the number of elements). If s is a sequence, then $asSet(s)$ is a set containing the same elements as s without duplicates.

A sequence, like a set, is often defined in this paper by *comprehension*, which generally has the following form:

$$(f(x_1, ..., x_n) \mid P(x_1, ..., x_n)),$$

where $x_1, ..., x_n$ are universally quantified variables, f a function name, and P a predicate. Typical sequence type constructors are the empty sequence constructor () and the singleton sequence constructor (e) where e is an atomic value or node. If s_1 and s_2 are two sequences of the same type, then $s_1 + s_2$ is a *concatenation* of the sequences, such that the first element of s_2 follows the last element of s_1 (the notation $(\mathtt{v_1}, ..., \mathtt{v_n})$ can be considered as a shorthand for $(v_1) + ... + (v_n)$). Also, $s_1 \cup s_2$ is a *union* of the sequences, such that the resulting sequence contains the elements of both sequences (retaining duplicate elements) in an indefinite order. The partial operation $itemize : Seq(t) \rightarrow t$ converts a singleton sequence into its element. The predicate $\in$ checks whether an element belongs to a sequence. The length of a sequence s is denoted by $|s|$ in this paper. A set can be converted into a sequence by the operation $asSeq$ (the order of the elements is not defined).

Several operations are applicable to sets and sequences of numerical values. These are avg, sum, max, and min.

The union type constructor plays an important role in the data model since a sequence may consist of items of different types. There are several predefined union types in the data model. The type *xdt:anyAtomicType* is the union of all the atomic types and the type *xdt:untypedAtomic*. The type *xs:anySimple* is the union of *xdt:anyAtomicType* and specific user-defined list and union types. The type *xs:anySimple* is the union of *Node* and *xdt:anyAtomicType*, and finally the type *xs:anyType* is the union of all types[1]. The following law exists for "flattening" the union of union types:

$$Union(t_1, ..., t_i, ..., t_n) = Union(t_1, ..., t_{i1}, ..., t_{im}, ..., t_n)$$

[1]In fact, the type *xs:anyType* does not include node types in the XQuery data model; it includes them in our data model for generality.

if $t_i = Union(t_{i1}, ..., t_{im})$. The following law permits us to get rid of duplicate component types in a union type:

$$Union(t_1, ..., t_i, ..., t_j, ..., t_n) = Union(t_1, ..., t_i, ..., t_n)$$

if $t_i = t_j$.

An XML database schema consists of a number of document definitions. Each individual document consists of *information items* with a definite *document order*. An information item of a document is mapped to a *node* in the database. A node, like an object of an object-oriented database [11], possesses an *identifier* and *state* represented by the values of node *accessors* resembling *observing methods* of an object-oriented database (we mean a node identifier by a node in the sequel). Each node is an instance of type $Node$, which is the union type for the types $Document$, $Element$, $Attribute$, $Text$, $Namespace$, $ProcInstruction$, and $Comment$ whose respective instances are document, element, attribute, text, namespace, processing instruction, and comment nodes[2]. Finally, each node and each atomic value are instances of the type $Item$, which is the union of types *xdt:anyAtomicType* and $Node$. Only atomic values and nodes may be components of sets and sequences (i.e., in the type $Set(t)$ or $Seq(t)$, the type t is either an atomic type or a node type or a union of atomic/node types). Atomic values and nodes are called *items* in the sequel.

All nodes in a database are arranged in linear order in such a way that if a node of one document tree precedes a node of another document tree, then all the nodes of the first tree precede all the nodes of the second tree. The operation $\mathtt{nd}_1 << \mathtt{nd}_2$ results in `true` if the node $\mathtt{nd}_1$ precedes the node $\mathtt{nd}_2$ (see [1] for an algorithm implementing this operation).

It is assumed that each data type is equipped with an equality predicate permitting to check for equality two values of the type; the equality of nodes is based on the identity of node identifiers.

In addition to the types used in [12], we use record (tuple) types in this paper. A record type $\mathtt{rec}\ p_1 : t_1, ..., p_n : t_n\ \mathtt{end}$ is equipped with a record construction operation *rec* producing a record on the base of record field values and projecting operations p_i producing field values of a given record. If $p_1, ..., p_n$ are identifiers and $v_1, ..., v_n$ are values of respective types $t_1, ..., t_n$, then $\mathtt{rec}(\mathtt{v}_1, ..., \mathtt{v}_n)$ is a record constructing expression of type $\mathtt{rec}\ p_1 : t_1, ..., p_n : t_n\ \mathtt{end}$.

[2]To save space, we do not consider the last three kinds of nodes in this paper. However, there is no technical problem in taking them into consideration if needed.

3. SIGNATURES AND EXPRESSIONS

A database schema defines a database signature $\Sigma = (T, F)$. F includes, in addition to the operators defined in data types, node accessors defined in [25], all the function names and operators defined in [26], the names of navigating functions defined in this paper and some special constants defined in the sequel. Node accessors used in the paper are *node-kind*, *node-name*, *parent*, *string-value*, *type*, *children*, *attributes*, and *nilled*. Two extra functions with signatures

$document\_order : Seq(Node) \rightarrow Seq(Node)$ and
$reverse\_order : Seq(Node) \rightarrow Seq(Node)$

are also part of our F. The first function orders the argument sequence in document order, and the second one orders it in reverse document order.

Any particular database state is an algebra of this signature as it is explained above. The signature Σ (as any other signature) may be *enriched* by new sorts and/or operators. In this case a Σ-algebra $\mathtt{A}$ is extended with new sets and/or functions associated with the new signature components.

Using operators from F, we can construct *expressions*. Each expression has a certain type. In a type hierarchy, a subtype expression is also a supertype expression.

Given a Σ-algebra $\mathtt{A}$, an expression can be *interpreted* or *evaluated*, yielding a certain algebra element. However, in contrast to conventional expressions of many-sorted signatures whose interpretation never changes neither the signature nor the algebra, XML expressions may be classified into three groups:

- conventional algebraic expressions written and evaluated in the same signature and algebra;
- expressions written in one signature and interpreted in an algebra of an enriched signature;
- expressions whose interpretation changes the algebra and produces an element of the new algebra.

There is nothing special with respect to the expressions of the first group. The situation with the expressions of the second group is more complex. An example of such an expression in relation algebra is projection of a relation onto a set of attributes or join of two relations. In either case the type of the resulting relation may be different from the relation types defined in the database schema. A query compiler, when parsing such an expression, constructs a new type and enriches the original signature with it. The current algebra is extended by the new type (sort and operations) as well,

and the query is interpreted in the new algebra. If a signature Σ is enriched to signature Σ' and a Σ-algebra $\mathtt{A}$ is extended to Σ'-algebra $\mathtt{A}'$, we use the index $\mathtt{A}$ to denote those components of $\mathtt{A}'$ that are the same as in $\mathtt{A}$.

An expression of the third group is the most difficult to process because the processing generally produces a side-effect (i.e., the expression, being interpreted in a certain algebra, may change the algebra and produce an element of this algebra). An example of this kind of expression is a *node constructing expression* whose interpretation produces a new node in a new algebra. We consider that such an expression is based on a function yielding a pair, an algebra and an algebra element. Note that a node construction expression is an expression and, according to the syntax of XQuery, can be used anywhere an expression is needed. This means that generally the interpretation of any expression may produce a side-effect. We will indicate this in the interpretations of the majority of the expressions, omitting it only in some simple cases.

We always write an expression e in italics. Its interpretation in algebra $\mathtt{A}$ is written as $[\![e]\!]^{\mathtt{A}}$. The result of the interpretation is generally written as $\langle \mathtt{A}', \mathtt{e} \rangle$, where $\mathtt{A}'$ is a new algebra and $\mathtt{e}$ is the evaluation of e in $\mathtt{A}'$. However, where there may be no confusion, we write just $\mathtt{e}$ for the interpretation of e. If e is an expression of type $Seq(t)$, we sometimes write: "e denotes a sequence of items of type t".

XQuery does not make difference between an item and singleton sequence. In the algebra, they are elements of different sorts by definition. For this reason, we should slightly modify the conventional definition and interpretation of the expression constructed by function symbol application. So, if $f : u_1, ..., u_n \to u$ is a function signature and u_i, $i = 1, ..., n$, is either an atomic/node type t_i or $seq(t_i)$ and e_i is an expression either of type t_i or $seq(t_i)$, then $f(e_1, ..., e_n)$ is an expression of type u.

Interpretation. Given an algebra $\mathtt{A}$,

$[\![f(e_1, ..., e_n)]\!]^{\mathtt{A}} = [\![f(e'_1, ..., e'_n)]\!]^{\mathtt{A}}$, where

$$e'_i = \begin{cases} e_i \text{ if } u_i \text{ is } t_i \text{ and } e_i \text{ is an expression of type } t_i \\ (e_i) \text{ if } u_i \text{ is } seq(t_i) \text{ and } e_i \text{ is an expression of type } t_i \\ itemize(e_i) \text{ if } u_i \text{ is } t_i \text{ and } e_i \text{ is an expression of type } seq(t_i) \end{cases}$$

Given a signature Σ and a set of variables X, we write "Σ-expression e" if e is composed exclusively of operators of Σ, and we write "(Σ, X)-expression e" if e contains, in addition, variables from X. If $\mathtt{A}$ is a Σ-algebra and $\xi : X \to |A|$ a variable assignment, then the notation $e\xi^{\mathtt{A}}$, or simply $\mathtt{e}\xi$,

means in the sequel the interpretation of e in algebra `A` with the variables bound to the indicated algebra elements.

The expression syntax is conventional with conventional operator priorities. Generally, an expression is parsed from left to right. The cases where we use special syntax or special parsing order will be mentioned explicitly.

The definitions of some functions and expressions use node accessors defined in [25] or the standard functions defined in [26]. We prefix the former by `dm` and the latter by `fn`.

4. RUNNING EXAMPLE

The examples given in the paper are mainly based on the queries presented in [17, 23] for a database containing documents of the following structure.

```
< bib >
                . . .
   < book year = ... >
      < title > ... < /title >
      < author > ... < /author >
      < author > ... < /author >
                . . .
      <publisher>...</publisher>
      < price > ... < /price >
   < /book >
                . . .

   < proc >
      < title > ... < /title >
                . . .
      <editor>...</editor>
      <editor>...</editor>
                . . .
      <article>
         < author > ... < /author >
         < author > ... < /author >
                . . .
      </article>
                . . .
   < /proc >
                . . .
< /bib >
```

5. NAVIGATION FUNCTIONS

In addition to node accessors, XQuery uses a number of functions producing different parts of a document tree relative to a specified node. Each of these functions is defined below for a node `nd` of a certain algebra `A`.

1. *child* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing all children nodes of the argument node if any. Definition:

a) $\texttt{child(nd)} = \texttt{dm:children(nd)}$
if $\texttt{dm:node-kind(nd)} \in \{\texttt{document, element}\}$,

b) $\texttt{child(nd)} = ()$, otherwise.

2. *descendant* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing all descendants of a node in the "parent-children" hierarchy if any. Definition:

a) if $\texttt{dm:node-kind(nd)} \in \{\texttt{document, element}\}$,
then $\texttt{descendant(nd)} = \texttt{document\_order(s)}$, where
$\texttt{s} = \texttt{dm:children(nd)} \cup (\texttt{dm:children}(\texttt{nd}_1) \mid \texttt{nd}_1 \in \texttt{s})$,

b) $\texttt{descendant(nd)} = ()$, otherwise.

3. *descendant_or_self* : $Node \rightarrow Seq(Node)$. This function yields a sequence consisting of the argument node and all its descendants in the "parent-children" hierarchy. Definition:

a) $\texttt{descendant\_or\_self(nd)} = (\texttt{nd}) \cup \texttt{descendant(nd)}$
if $\texttt{dm:node-kind(nd)} \in \{\texttt{document, element}\}$;

b) $\texttt{descendant\_or\_self(nd)} = ()$, otherwise.

4. *parent* : $Node \rightarrow Seq(Node)$. The function yields a sequence containing the parent of the argument node if any. Definition:

$\texttt{parent(nd)} = \texttt{dm:parent(nd)}$.

5. *ancestor* : $Node \rightarrow Seq(Node)$. The function yields a sequence containing the ancestors of the argument node.Definition:

$\texttt{ancestor(nd)} = \texttt{reverse\_order(s)}$, where
$\texttt{s} = \texttt{dm:parent(nd)} \cup (\texttt{dm:parent}(\texttt{nd}_1) \mid \texttt{nd}_1 \in \texttt{s})$.

6. *ancestor_or_self* : $Node \rightarrow Seq(Node)$. The function yields a sequence containing the argument node and all its ancestors. Definition:

$\texttt{ancestor\_or\_self(nd)} = (\texttt{nd}) \cup \texttt{ancestor(nd)}$.

7. *following_sibling* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing the sibling nodes following the argument node. Definition:

a) $\texttt{following\_sibling(nd)} = ()$
if $\texttt{dm:node-kind(nd)} \in \{\texttt{document, attribute}\}$,

b) $\texttt{following\_sibling(nd)} = \texttt{document\_order(s)}$, where
$\texttt{s=(nd}_1|\texttt{nd}_1 \in \texttt{dm:children(dm:parent(nd))}$ and $\texttt{nd} << \texttt{nd}_1)$,
otherwise.

8. *following* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing all nodes that are descendants of the root node of the tree containing the argument node, are not descendants of the argument node, and occur after it in document order. Definition: let $\texttt{rnd} = \texttt{fn:root(nd)}$, $\texttt{descendant(rnd)} = \texttt{s}_1$, and $\texttt{descendant(nd)} = \texttt{s}_2$, then
$\texttt{following(nd)} = \texttt{document\_order(s)}$, where
$\texttt{s} = (\texttt{nd}_1 \mid \texttt{nd}_1 \in \texttt{s}_1$ and $\texttt{nd}_1 \notin \texttt{s}_2$ and $\texttt{nd} << \texttt{nd}_1)$.

9. *preceding_sibling* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing the sibling nodes preceding the argument node. Definition:

a) $\texttt{preceding\_sibling(nd)} = ()$
if $\texttt{dm:node-kind(nd)} \in \{\texttt{document, attribute}\}$,

b) $\texttt{preceding\_sibling(nd)} = \texttt{reverse\_order(s)}$, where
$\texttt{s=(nd}_1|\texttt{nd}_1 \in \texttt{dm:children(dm:parent(nd))}$ and $\texttt{nd}_1 << \texttt{nd})$, otherwise.

10. *preceding* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing all nodes that are descendants of the root node of the tree containing the argument node, are not ancestors of the argument node, and occur before it in the document node. Definition. Let $\texttt{rnd} = \texttt{fn:root(nd)}$, $\texttt{descendant(rnd)} = \texttt{s}_2$, and $\texttt{descendant(nd)} = \texttt{s}_3$, then
$\texttt{preceding(nd)} = \texttt{reverse\_order(s)}$, where
$\texttt{s} = (\texttt{nd}_1 \mid \texttt{nd}_1 \in \texttt{s}_2$ and $\texttt{nd}_1 \notin \texttt{s}_3$ and $\texttt{nd}_1 << \texttt{nd})$.

11. *attribute* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing the attribute nodes of the argument element node. Definition:

a) $\texttt{attribute(nd)} = \texttt{dm:attributes(nd)}$
if $\texttt{dm:node-kind(nd) = element}$,

b) $\texttt{attribute(nd)} = ()$, otherwise.

12. *self* : $Node \rightarrow Seq(Node)$. This function yields a sequence containing

the argument node itself. Definition:

`self(nd)` = (`nd`).

Notation. The call of each of the above functions is written in this paper using the dot notation, i.e., as a call of a method in an object-oriented language; for instance, *nd.child*, *nd.parent*, etc.

6. QUERYING EXPRESSIONS

In addition to elementary expressions constructed with the use of navigating functions defined above, an XML data model must include facilities for constructing more complex expressions representing data retrieval or update. The set of all possible expressions in an XML data model constitutes an *XML algebra*[3].

Generally, an XQuery query has the following form:

for x_1 **in** s_1, x_2 **in** $s_2(x_1)$, ... , x_m **in** $s_m(x_1, ..., x_{m-1})$
let $y_1 := e_1(x_1, ..., x_m)$, $y_2 := e_2(x_1, ..., x_m, y_1)$, ... ,
$y_n := e_n(x_1, ..., x_m, y_1, ..., y_n)$
where $p(x_1, ..., x_m, y_1, ..., y_n)$
order by $e(x_1, ..., x_m, y_1, ..., y_n)$
return $f(x_1, ..., x_m, y_1, ..., y_n)$

where s_i has to be a sequence, and p, e and f are expressions involving the variables $x_1, ..., x_m, y_1, ..., y_n$. Normally, s_i's are nested sequences. Thus, to represent such a query in XML algebra, we need an expression that evaluates to a sequence of tuples of items belonging to possibly nested sequences (clauses **for** and **let**), an expression that evaluates to a subsequence of a sequence according to selection criteria (clause **where**), ordering expression, and an expression that constructs the resulting sequence (clause **return**). These expressions are defined in the sequel.

The signature of our algebras is supposed to include the enumeration type $orderingMode = Enumeration(ordered, unodered)$ and the constant $order\_mode : orderingMode$ set to one of indicated values in any algebra of the signature. The value of the constant governs the ordering of the sequence resulting from the evaluation of some expressions.

[3]We define special forms of expressions rather than functions to avoid the problem of higher-order functions (a conventional algebra is a first-order structure).

6.1. Unnesting join expression

This expression in fact replaces the *join* operation of the relation algebra because relationships between different sequences of nodes in the XML database are represented by node identifiers rather than by relation keys. Unnesting join of a sequence s, such that each its item s_i refers to a sequence s'_i, consists of pairing of s_i with each element of s'_i. Some sequences participating in the operation may be independent of each other, in this case we have a join of a forest of trees.

We first define three auxiliary expressions serving to support different kinds of FOR and LET clauses.

1. If X is a (possibly empty) set of variables, y an identifier, and s a (Σ, X)-expression of type $Seq(t)$, then $\lessdot y : s \gtrdot$ is an expression of type $t' = Seq(\texttt{rec}\ y : t\ \texttt{end})$ of signature Σ' obtained by enriching Σ by type t'.

Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\xi : X \to |\mathtt{A}|$ a variable assignment, $\mathtt{A}'$ a Σ'-algebra extending $\mathtt{A}$ by type $\mathtt{A}_{t'}$, and $[\![s\xi]\!]^{\mathtt{A}'} = \langle \mathtt{A}^{\mathtt{s}}, \mathtt{s} \rangle$, then:

$$[\![\lessdot y : s \gtrdot]\!]^{\mathtt{A}'} = \langle \mathtt{A}^{\mathtt{s}}, \mathtt{s}' \rangle,$$

where $\mathtt{s}' = (\texttt{rec}\ (\mathtt{v}) \mid \mathtt{v} \in \mathtt{s})\rangle$.
This expression supports the FOR clause with a single range variable. The following rule controls the ordering of the resulting sequence $\mathtt{s}'$: let $\mathtt{s}'[\mathtt{j1}] = \texttt{rec}(\mathtt{s}[\mathtt{i1}])$, $\mathtt{s}'[\mathtt{j2}] = \texttt{rec}(\mathtt{s}[\mathtt{i2}])$, then

$$\texttt{order\_mode} = \texttt{ordered} \Rightarrow (\mathtt{i1} < \mathtt{i2} \Leftrightarrow \mathtt{j1} < \mathtt{j2}),$$

i.e., if ordering mode is `ordered`, then the resulting sequence is ordered according to the order of the values in the argument sequence.

2. If X is a (possibly empty) set of variables, y and i identifiers, and s a (Σ, X)-expression of type $Seq(t)$, then $\lessdot y, i : s \gtrdot$ is an expression of type $t' = Seq(\texttt{rec}\ i : \mathit{integer},\ y : t\ \texttt{end})$ of signature Σ' obtained by enriching Σ by the type t'. Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\xi : X \to |\mathtt{A}|$ a variable assignment, $\mathtt{A}'$ a Σ'-algebra extending $\mathtt{A}$ by type $\mathtt{A}_{t'}$, and $[\![s\xi]\!]^{\mathtt{A}'} = \langle \mathtt{A}^{\mathtt{s}}, \mathtt{s} \rangle$, then:

$$[\![\lessdot y, i : e \gtrdot]\!]^{\mathtt{A}'} = \langle \mathtt{A}^{\mathtt{s}}, (\texttt{rec}\ (\mathtt{i}, \mathtt{s}[\mathtt{i}]) \mid \mathtt{i} = 1, ..., |\mathtt{s}|)\rangle.$$

This expression supports the FOR clause with a range variable and a positional variable. The ordering of the values in the resulting sequence obeys the same rule as the previous one.

3. If X is a (possibly empty) set of variables, y an identifier, and e a (Σ, X)-expression either of type t, then $\lessdot y = e \gtrdot$ is an expression of type $t' = Seq(\texttt{rec}\ y : t\ \texttt{end})$ of signature Σ' obtained by enriching Σ by the type t'.

Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\xi : X \to |\mathtt{A}|$ a variable assignment, $\mathtt{A}'$ a Σ'-algebra extending $\mathtt{A}$ by type $\mathtt{A}_{\mathtt{t}'}$, and $[\![e\xi]\!]^{\mathtt{A}'} = \langle \mathtt{A}^{\mathtt{e}}, \mathtt{e} \rangle$, then:

$$[\![\lessdot y = e \gtrdot]\!]^{\mathtt{A}'} = \langle \mathtt{A}^{\mathtt{e}}, (\texttt{rec}\ (\mathtt{e})) \rangle.$$

This expression supports the LET clause with a single variable name.

4. Finally we define an expression supporting any combination of FOR and LET clauses. If s_1 is a Σ-expression of type $Seq(\texttt{rec}\ x_{11} : t_{11},\ ...,\ x_{1m} : t_{1m}\ \texttt{end})\}$ and s_2 a $(\Sigma, \{x_{11}, ..., x_{1m}\})$-expression of type $Seq(\texttt{rec}\ x_{21} : t_{21},\ ...,\ x_{2n} : t_{2n}\ \texttt{end})\}$, then $s_1 * s_2$ is an expression of type $t' = Seq(\texttt{rec}\ x_{11} : t_{11},\ ...,\ x_{1m} : t_{1m}, x_{21} : t_{21},\ ...,\ x_{2n} : t_{2n}\ \texttt{end})$ of signature Σ' obtained by enriching Σ by the type t'.

Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\mathtt{A}'$ a Σ'-algebra extending $\mathtt{A}$ by type $\mathtt{A}_{\mathtt{t}'}$, $[\![s_1]\!]^{\mathtt{A}'} = \langle \mathtt{A}^1, \mathtt{s}_1 \rangle$, and $\mathtt{k} = |\mathtt{s}_1|$. Further, $\forall\ \mathtt{i} = 1, ..., \mathtt{k}$ let
$\mathtt{s}_1[\mathtt{i}] = \texttt{rec}(\mathtt{v}_{\mathtt{i}1}, ..., \mathtt{v}_{\mathtt{im}})$,
$\xi_i = \{x_1 \mapsto \mathtt{v}_{\mathtt{i}1}, ..., x_m \mapsto \mathtt{v}_{\mathtt{im}}\}$,
$[\![s_2\xi_1]\!]^{\mathtt{A}^1} = \langle \mathtt{AA}^1, \mathtt{s}_{21} \rangle$, ... $[\![s_2\xi_k]\!]^{\mathtt{A}^{k-1}} = \langle \mathtt{AA}^{\mathtt{k}}, \mathtt{s}_{2\mathtt{k}} \rangle$,
$\mathtt{ss}_{\mathtt{i}} = (\texttt{rec}\ (\mathtt{v}_{\mathtt{i}1}, ..., \mathtt{v}_{\mathtt{im}}, \mathtt{w}_1, ..., \mathtt{w}_{\mathtt{n}}) \mid\ \texttt{rec}(\mathtt{w}_1, ..., \mathtt{w}_{\mathtt{n}}) \in \mathtt{s}_{2\mathtt{i}})$,
then:

$$[\![s_1 * s_2]\!]^{\mathtt{A}'} = \langle \mathtt{AA}^{\mathtt{n}}, \mathtt{s}' \rangle,\ \text{where}\ \mathtt{s}' = \mathtt{ss}_1 \cup ... \cup \mathtt{ss}_{\mathtt{k}}.$$

The following rule controls the ordering of the resulting sequence $\mathtt{s}'$ if `order_mode = ordered`. Let
$\mathtt{i1}, \mathtt{i2} \in \{1, ..., |\mathtt{s}_1|\}$, $\mathtt{j1}, \mathtt{j2} \in \{1, ..., |\mathtt{s}_2|\}$, $\mathtt{k1}, \mathtt{k2} \in \{1, ..., |\mathtt{s}'|\}$,
$\mathtt{s}_1[\mathtt{i1}] = \texttt{rec}(\mathtt{v}_{\mathtt{i}11}, ..., \mathtt{v}_{\mathtt{i1m}})$,
$\mathtt{s}_1[\mathtt{i2}] = \texttt{rec}(\mathtt{v}_{\mathtt{i}21}, ..., \mathtt{v}_{\mathtt{i2m}})$,
$\mathtt{s}_{2\mathtt{i}}[\mathtt{j1}] = \texttt{rec}(\mathtt{w}_{\mathtt{j}11}, ..., \mathtt{w}_{\mathtt{j1n}})$,
$\mathtt{s}_{2\mathtt{i}}[\mathtt{j2}] = \texttt{rec}(\mathtt{w}_{\mathtt{j}21}, ..., \mathtt{w}_{\mathtt{j2n}})$,
$\mathtt{s}'[\mathtt{k1}] = \texttt{rec}(\mathtt{v}_{\mathtt{i}11}, ..., \mathtt{v}_{\mathtt{i1m}}, \mathtt{w}_{\mathtt{j}11}, ..., \mathtt{w}_{\mathtt{j1n}})$,
$\mathtt{s}'[\mathtt{k2}] = \texttt{rec}(\mathtt{v}_{\mathtt{i}21}, ..., \mathtt{v}_{\mathtt{i2m}}, \mathtt{w}_{\mathtt{j}21}, ..., \mathtt{w}_{\mathtt{j2n}})$,
then:

$$\mathtt{k1} < \mathtt{k2} \iff \mathtt{i1} < \mathtt{i2} \vee (\mathtt{i1} = \mathtt{i2}\ \wedge\ \mathtt{j1} < \mathtt{j2}),$$

i.e., if ordering mode is `ordered`, then the order of the resulting sequence is primarily determined by the order of the values in the left argument

sequence and secondarily by the order of the values in the right argument sequence.

Examples. If the variable *books* denotes a sequence of book nodes, then

$\ll$*x: books*$\gg$ *$\ll$*y:x.child::element(authors)*$\gg$

is an expression evaluating to a sequence of pairs of `book` and `author` nodes so that a book node is paired with its each child author node. The following expression:

$\ll$*x: (1, 2, 3)*$\gg$ *$\ll$*y: (4, 5, 6)*$\gg$

evaluates in fact to the Cartesian product of the indicated sequences while the expression

$\ll$*x: (1, 2, 3)*$\gg$ *$\ll$*y = (x+1, x+2)*$\gg$

evaluates to the following sequence of tuples:

($\langle$1, (2,3)$\rangle$, $\langle$2, (3,4)$\rangle$, $\langle$3, (4,5)$\rangle$).

If variable *pets* denotes a sequence (`"cat"`, `"dog"`, `"pig"`), then the expression:

$\ll$*t,i:pets*$\gg$

evaluates to the following sequence of pairs

($\langle$ 1, `"cat"`$\rangle$, $\langle$ 2, `"dog"`$\rangle$,$\langle$ 3, `"pig"`$\rangle$).

6.2. Quantified expressions

Universal quantification and existential quantification are widely used in XQuery. The corresponding algebra expressions can be defined as follows.

If $t_1, t_2..., t_n$ are types from the signature Σ, $X = \{x_1, x_2, ..., x_n\}$ a set of variables, s_1 a Σ-expression of type $Seq(t_1)$, s_2 a $(\Sigma, \{x_1\})$-expression of type $Seq(t_2)$, ... , s_n a $(\Sigma, \{x_1, ..., x_{n-1}\})$-expression of type $Seq(t_n)$, and b a $(\Sigma, \{x_1, ..., x_n\})$-expression either of type *Boolean* or $Seq(Boolean)$, then

forall$(x_1 : s_1, x_2 : s_2, ..., x_n : s_n)!b$ and

exists$(x_1 : s_1, x_2 : s_2, ..., x_n : s_n)!b$

are expressions of type *Boolean*.

Interpretation. Let b' be b if b has type *Boolean* and *itemize*(b) in the opposite case, and $[\![\ll x_1 : s_1 \gg * \ll x_2 : s_2 \gg * ... * \ll x_n : s_n \gg]\!]^{\mathtt{A}} = \langle \mathtt{A^n}, \mathtt{s} \rangle$, then:

$[\![$**forall**$(x_1 : e_1, x_2 : e_2, ..., x_n : e_n)!b]\!]^{\mathtt{A}} = \langle \mathtt{A}, \forall\ \mathtt{rec}(\mathtt{v_1}, ..., \mathtt{v_n}) \in \mathtt{s} :\ \mathtt{b'} \rangle$,

$[\![$**exists**$(x_1 : e_1, x_2 : e_2, ..., x_n : e_n)!b]\!]^{\mathtt{A}} = \langle \mathtt{A}, \exists\ \mathtt{rec}(\mathtt{v_1}, ..., \mathtt{v_n}) \in \mathtt{s} :\ \mathtt{b'} \rangle$,

assuming that $[\![b'\xi_i]\!]^{\mathtt{A^n}} = \langle \mathtt{A^b}, \mathtt{b'} \rangle$, where $\xi_i = \{x_1 \mapsto \mathtt{v_1}, ..., x_n \mapsto \mathtt{v_n}\}$ is a variable assignment. Note that although a number of intermediate algebras may be created in the process of evaluating the involved expressions, the original

algebra A is part of the final result since none of the nodes constructed in an intermediate algebra is needed as soon as the whole expression has been evaluated.

6.3. Selection expressions

A *selection expression* serves for selecting part of a sequence basing on a *selection criteria*. In comparison to relational model and object model, the set of selection criteria in XML algebra is much broader and includes node *kind tests* in addition to *predicate tests*.

Each selection expression has the following form: $s :: p$, where s is a sequence and p a selection condition. A general feature of all these expressions is that they are *order retaining*, i.e., if $\mathtt{nd}_1$ and $\mathtt{nd}_2$ are in the resulting sequence and $\mathtt{nd}_1 << \mathtt{nd}_2$ in s, then $\mathtt{nd}_1 << \mathtt{nd}_2$ in the resulting sequence. The interpretation of these expressions takes place in an algebra A, and it does not change the algebra.

6.3.1. Kind tests

Let s denote a sequence of nodes. Then:
1) $s :: element()$ denotes a sequence of nodes $\mathtt{s}'$ defined as follows:
$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & dm:node-kind(nd) = element).

2) $s :: attribute()$ denotes the following sequence of nodes $\mathtt{s}'$:
$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & dm:node-kind(nd) = attribute).
3) $s :: text()$ denotes the following sequence of nodes $\mathtt{s}'$:
$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & dm:node-kind(nd) = text).

Example. If the variable *books* denotes a sequence of nodes that are descendants of a *bib* node, then $books :: text()$ denotes a sequence consisting only of text nodes contained in *books*.

4) $s :: document()$ denotes a singleton sequence of nodes $\mathtt{s}'$ defined as follows:
$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & dm:node-kind(nd) = document).

5) if n is a QName, then $s :: element(n)$ denotes the following sequence of nodes $\mathtt{s}'$:
$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & dm:node-kind(nd) = element
& dm:node-name(nd) = n).
Example. If the variable *book_data* denotes a sequence of nodes that are

children of a *book* node, then *book_data* :: *element*(*author*) denotes a sequence consisting only of author element nodes of a particular book.

6) if n is a QName and t a type name, then s :: *element*(n, t) denotes the following sequence of nodes $\mathtt{s}'$:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `element`
& `dm:node-name(nd) = n` & `dm:type-name(nd) = t`
& `dm` : `nilled(nd)` = `false`).

7) if t is a type name, then s :: *element*$(*, t)$ denotes the following sequence of nodes $\mathtt{s}'$:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `element`
& `dm:type-name(nd) = t` & `dm` : `nilled(nd)` = `false`).

8) if n is a QName and t a type name, then s :: *element*$(n, t?)$ denotes the following sequence of nodes $\mathtt{s}'$:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `element`
& `dm:node-name(nd) = n` & `dm:type-name(nd) = t`).

9) if t is a type name, then s :: *element*$(*, t?)$ denotes the following sequence of nodes $\mathtt{s}'$:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `element`
& `dm:type-name(nd) = t`).

10) if n is a QName, then s :: *attribute*(n) denotes a singleton sequence of nodes $\mathtt{s}'$ defined as follows:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `attribute`
& `dm:node-name(nd) = n`).

11) if n is a QName and t a type name, then s :: *attribute*(n, t) denotes the following sequence of nodes $\mathtt{s}'$:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `attribute` &
`dm:node-name(nd) = n` & `dm:type-name(nd) = t`).

12) if t is a type name, then s :: *attribute*$(*, t)$ denotes the following sequence of nodes $\mathtt{s}'$:

$\mathtt{s}' = (\mathtt{nd} \mid \mathtt{nd} \in \mathtt{s}$ & `dm:node-kind(nd)` = `attribute`
& `dm:type-name(nd) = t`).

6.3.2. Predicate tests

Let x be a variable, t a type from the signature Σ, s a $(\Sigma, \{x\})$-expression of

type $Seq(t)$, y a variable of type t, and p a $(\Sigma, \{y\})$-expression either of type *Boolean* or $Seq(Boolean)$, then **select**$(y : s) :: p$ is a $(\Sigma, \{x\})$-expression of type $Seq(t)$.

Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\nu = \{x \mapsto |\mathtt{A}|\}$,
$[\![s\nu]\!]^{\mathtt{A}} = \langle \mathtt{A}', (\mathtt{v}_1, \ldots, \mathtt{v}_n)\rangle$, i = 1, ..., n,
p' be p if p has type *Boolean* and $itemize(p)$ in the opposite case,
$\xi_i = \{y \mapsto \mathtt{v}_i\}$, $[\![p'\xi_1]\!]^{\mathtt{A}'} = \langle \mathtt{A}^1, \mathtt{p}_1\rangle$, ..., $[\![p'\xi_n]\!]^{\mathtt{A}'} = \langle \mathtt{A}^n, \mathtt{p}_n\rangle$,
then

$$[\![\textbf{select}(y : s) :: p]\!]^{\mathtt{A}} = \langle \mathtt{A}', (\mathtt{v}_i \mid \mathtt{p}_i)\rangle.$$

Note that the algebra $\mathtt{A}'$ is part of the final result since any node that may be created in an intermediate algebra $\mathtt{A}^i$, `i = 1, ..., n`, is not needed as soon as the predicate p' has been evaluated.

Example. If the variable *books* denotes a sequence of book nodes, then

select*(x: books) :: typed-value(x.attribute :: attribute(year))= 2000*

denotes a sequence of book nodes for the books published in 2000 and

select*(x: books) :: typed-value(x.child :: element(price)) > 100*

denotes a sequence of book nodes for the books whose price is greater than 100 dollars.

Note that both selection expressions do not depend on a global variable, which may be the case when the selection expression is part of the right step in the path expression (see next section).

The expression can be written in a simpler form if t is a record type. Formally: if x is a variable, s a $(\Sigma, \{x\})$-expression of type $Seq(\mathtt{rec}\ x_1 : t_1, ..., x_n : t_n\ \mathtt{end})$ and p a $(\Sigma, \{x_1, ..., x_n\})$-expression either of type *Boolean* or $Seq(Boolean)$, then **select**$(s) :: p$ is a $(\Sigma, \{x\})$-expression of type $Seq(t)$.

Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\nu = \{x \mapsto |\mathtt{A}|\}$,
$[\![s\nu]\!]^{\mathtt{A}} = \langle \mathtt{A}', (\mathtt{r}_1, ..., \mathtt{r}_m)\rangle$, i = 1, ..., m, $\mathtt{r}_i = \mathtt{rec}(\mathtt{v}_{i1}, ..., \mathtt{v}_{in})$,
$\xi_i = \{x_1 \mapsto \mathtt{v}_{i1}, ..., x_n \mapsto \mathtt{v}_{in}\}$,
p' be p if p has type *Boolean* and $itemize(p)$ in the opposite case,
$[\![p'\xi_1]\!]^{\mathtt{A}'} = \langle \mathtt{A}^1, \mathtt{p}_1\rangle$, ... $[\![p'\xi_m]\!]^{\mathtt{A}'} = \langle \mathtt{A}^m, \mathtt{p}_m\rangle$,
then

$$[\![\textbf{select}(s) :: p]\!]^{\mathtt{A}} = \langle \mathtt{A}', (\mathtt{r}_i \mid \mathtt{p}_i)\rangle.$$

Example. If *books* denotes a sequence of book nodes, then

select$(\lessdot x : books \gtrdot * \lessdot y = x.child{::}element(author) \gtrdot) :: count(y) > 2$

is a selection expression. Note that the local variable x ranges over books, y denotes the authors of a particular book, the expression

$$(\lessdot x : books \gtrdot * \lessdot y = x.child :: element(author) \gtrdot)$$

is a sequence of pairs $\langle book, sequence\ of\ authors\rangle$, and the predicate $count(y) > 2$ leaves in the sequence only those pairs where there are more than two authors.

6.4. Path expression

This kind of expression permits one to navigate over a tree by using navigating functions.

If x, y are variables, s_1 a $(\Sigma, \{x\})$-expression of type $Seq(Node)$, t_2 an atomic/node type, and s_2 a $(\Sigma, \{y\})$-expression of type $Seq(t_2)$, then **path**$(y : s_1)/s_2$ is a $(\Sigma, \{x\})$-expression of type $Seq(t_2)$. The expressions s_1 and s_2 are called *left step* and *right step*, respectively.

Interpretation. Let $\mathtt{A}$ be a Σ-algebra, $\nu = \{x \mapsto |\mathtt{A}|\}$,
$[\![s_1\nu]\!]^{\mathtt{A}} = \langle \mathtt{A}', (\mathtt{nd}_1, \ldots, \mathtt{nd}_n)\rangle$, i = 1, ..., n, $\xi_i = \{y \mapsto \mathtt{nd}_i\}$,
$[\![s_2\xi_1]\!]^{\mathtt{A}'} = \langle \mathtt{A}^1, \mathtt{v}_1\rangle$, ... $[\![s_2\xi_n]\!]^{\mathtt{A}^{n-1}} = \langle \mathtt{A}^n, \mathtt{v}_n\rangle$, then

$$[\![\mathbf{path}(y : s_1)/s_2]\!]^{\mathtt{A}} = \begin{cases} \langle \mathtt{A}^n, \mathtt{f}(\mathtt{asSeq}(\mathtt{asSet}(\mathtt{v}_1 \cup \ldots \cup \mathtt{v}_n)))\rangle \\ \qquad\qquad\qquad\qquad \text{if } t_2 \text{ is a node type} \\ \langle \mathtt{A}^n, \mathtt{v}_1 + \ldots + \mathtt{v}_n\rangle \text{ if } t_2 \text{ is an atomic type,} \end{cases}$$

where `f` is `document_order` if `order_mode = ordered`, and identity function in the opposite case. Note that an ordered set is the result of the interpretation of this expression in the first case and a sequence in the second case.

Examples:

1) The following expression consists of two path subexpressions[4].

path*(x: fn:doc("books.xml") /*
path *(y: x.child::element(bib)) / y.child::element(book)*;

In the main path expression (not depending on a global variable), the left step is represented by the function *fn:doc()*, which produces a singleton sequence containing a document node. The right step is represented by another path expression (depending on the global variable x), which is evaluated for each element of the sequence produced (singleton sequence in this case). In this expression, the left step *x.child::element(bib)* gives us an element node at the top of the node hierarchy, which is used by the right step *y.child::element(book)* selecting the book elements within the bib elements.

[4]In this example and henceforth, it is considered that the operator '.' has a higher priority that the operator '::' which, in its turn, has a higher priority than the operator '/'. There is also no attempt to use any syntactic sugar in expressions.

2) If *books* denotes a sequence of book nodes, then

path(*y*:**select**(*x: books) :: typed-value(x.attribute :: attribute(year)) = 2000)/ y.child :: element(title)*;

is an expression evaluating to the titles of the books published in 2000 (note that x ranges over all books and y ranges only over those books that satisfy the selection condition).

3) Let *doc* denote the following document:

$< a >$
$< b >< c > 1 < /c >< c > 2 < /c >< /b >$
$< b >< c > 3 < /c >< c > 4 < /c >< /b >$
$< /a >$,

then the expression

path$(x : doc.child :: element(a))/$
path$(y : x.child :: element(b))/seq(y.child :: element(c)[2])$

evaluates to

$(< c > 2 < /c >, < c > 4 < /c >)$ or $(< c > 4 < /c >, < c > 2 < /c >)$.

6.5. Ordering expressions

In XQuery, the clause **order by** in the FLWOR expression orders a sequence of tuples (records) produced by evaluation of the preceding clauses, basing on the values of a number of expressions evaluated for each tuple of the sequence. Therefore, an ordering expression in our algebra serves to order a sequence of tuples (records) basing on the values of one or more of *ordering keys*, which are empty or singleton sequences. .

Generally, two values of the same ordering key are compared using a predefined operation ">" (greater). However, in case the ordering key has the string type, the name of a specific collation used for ordering may be indicated (as a string value). We will take both options into account.

Let t be a record type `rec` $x_1 : t_1$, ..., $x_n : t_n$ `end`, s a sequence of type $Seq(t)$, $e_1, ..., e_l$ be $(\Sigma, \{x_1, ..., x_n\})$-expressions each denoting either an empty or a singleton sequence of type $Seq(t'_k)$ where t'_k is an atomic type, a_k and b_k are one of the symbols '↑' or '↓' (a indicates whether the order is ascending ('↑') or descending ('↓') and b indicates whether the empty sequence has preference ('↑') or not ('↓')), and c_k is a possibly nonempty string if t'_k is the type `string` and the empty string in all other cases, then

stable_order$(e_1[a_1, b_1, c_1], ..., e_l[a_l, b_l, c_l] : s)$ and
order$(e_1[a_1, b_1, c_1], ..., e_l[a_l, b_l, c_l] : s)$

are expressions of type $Seq(t)$.

Interpretation. Let $\mathtt{A}$ be an algebra and $[\![s]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{s} \rangle$. Assume for simplicity that the interpretation of an ordering key does not update the algebra (it is difficult to imagine that an ordering key contains a node constructor). Then

$[\![\mathbf{stable\_order}(e_1[a_1, b_1, c_1], ..., e_l[a_l, b_l, c_l] : s)]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{s}' \rangle$ and
$[\![\mathbf{order}(e_1[a_1, b_1, c_1], ..., e_l[a_l, b_l, c_l] : s)]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{s}' \rangle$.

The interpretation of the first expression should produce a sequence $\mathtt{s}'$ containing the same items as $\mathtt{s}$ (i.e., $\mathtt{el} \in \mathtt{s} \iff \mathtt{el} \in \mathtt{s}'$) in the order dictated by a, b, and c. The second expression differs from the first one in retaining the relative positions of two items having equal values of the ordering key.

Two auxiliary Boolean expressions are used for comparing the values of an ordering key, which are empty or singleton sequences. They are defined as follows. If t is a type with an operation $>\colon t, t \to \mathit{Boolean}$, s_1 and s_2 are empty or singleton expressions of type $\mathit{Seq}(t)$, and *col* a possibly nonempty string if t is *String* and the empty string in all other cases, then

$esf_{s_1,s_2,col}$ and $esl_{s_1,s_2,col}$

are expressions of type *Boolean*. The first expression evaluates to `true` if the first sequence is empty and the second one is not; if both of them are non-empty sequences and *col* is not empty, then it evaluates to `true` iff the result of the standard function $\mathtt{fn} : \mathtt{compare}(\mathtt{s}_2, \mathtt{s}_1, \mathtt{col})$ is less than zero; if both of them are non-empty sequences and *col* is empty, then it evaluates to `true` iff the element of the first sequence is greater then the element of the second sequence (the empty sequence precedes a nonempty one). The second expression evaluates to `true` if the second sequence is empty and the first one is not or both of them are non-empty sequences and the conditions of the previous case hold (the empty sequence follows a nonempty one)[5]. The expressions are interpreted as follows.

$$\mathtt{esf}_{\mathtt{s}_1,\mathtt{s}_2,\mathtt{col}} = \begin{cases} \mathtt{true} \text{ if } |\mathtt{s}_1| = 0 \ \wedge \ |\mathtt{s}_2| > 0 \\ \mathtt{false} \text{ if } |\mathtt{s}_1| = 0 \wedge |\mathtt{s}_2| = 0 \vee |\mathtt{s}_1| > 0 \wedge |\mathtt{s}_2| = 0 \\ \mathtt{fn} : \mathtt{compare}(\mathtt{s}_2, \mathtt{s}_1, \mathtt{col}) < 0 \text{ if } |\mathtt{col}| > 0 \wedge |\mathtt{s}_1| > 0 \wedge |\mathtt{s}_2| > 0 \\ \mathtt{v}_1 > \mathtt{v}_2 \text{ if } |\mathtt{col}| = 0 \wedge \mathtt{s}_1 = (\mathtt{v}_1) \wedge \mathtt{s}_2 = (\mathtt{v}_2). \end{cases}$$

$$\mathtt{esl}_{\mathtt{s}_1,\mathtt{s}_2,\mathtt{col}} = \begin{cases} \mathtt{true} \text{ if } |\mathtt{s}_1| > 0 \ \wedge \ |\mathtt{s}_2| = 0 \\ \mathtt{false} \text{ if } |\mathtt{s}_1| = 0 \wedge |\mathtt{s}_2| = 0 \vee |\mathtt{s}_1| = 0 \wedge |\mathtt{s}_2| > 0 \\ \mathtt{fn} : \mathtt{compare}(\mathtt{s}_2, \mathtt{s}_1, \mathtt{col}) < 0 \text{ if } |\mathtt{col}| > 0 \wedge |\mathtt{s}_1| > 0 \wedge |\mathtt{s}_2| > 0 \\ \mathtt{v}_1 > \mathtt{v}_2 \text{ if } |\mathtt{col}| = 0 \wedge \mathtt{s}_1 = (\mathtt{v}_1) \wedge \mathtt{s}_2 = (\mathtt{v}_2). \end{cases}$$

Now, our task is to construct a bijection $\sigma : \{1, ..., |\mathtt{s}|\} \to \{1, ..., |\mathtt{s}|\}$ such that $\mathtt{s}[\mathtt{i}] = \mathtt{s}'[\sigma(\mathtt{i})]$, where $1 \leq i \leq |s|$. We present an algorithm

[5]It is assumed, of course, that both the operation ">" and the function $\mathtt{fn} : \mathtt{compare}$ are implemented as nonreflexive, antisymmetric, and transitive relations.

constructing the bijection. Let i and j be natural numbers such that $1 \leq i, j \leq |s|$ and $i \neq j$, $\mathtt{s[i]} = \mathtt{rec}(\mathtt{v_{i1}}, ..., \mathtt{v_{in}})$, $\mathtt{s[j]} = \mathtt{rec}(\mathtt{v_{j1}}, ..., \mathtt{v_{jn}})$, $\xi_1 = \{x_1 \mapsto \mathtt{v_{i1}}, ..., x_n \mapsto \mathtt{v_{in}}\}$, $\xi_2 = \{x_1 \mapsto \mathtt{v_{j1}}, ..., x_n \mapsto \mathtt{v_{jn}}\}$, and $k = 1$, then

1) let $\mathtt{s_{1k}} = \mathtt{e_k}\xi_1$ and $\mathtt{s_{2k}} = \mathtt{e_k}\xi_2$ in:
2) if $\mathtt{a_k} = \ '\uparrow'$ (ascending order)
3) then if $\mathtt{b_k} = \ '\uparrow'$ (empty sequence first)
4) then if $\mathtt{esf}_{\mathtt{s_{1k}},\mathtt{s_{2k}},\mathtt{c_k}}$ then $\sigma(\mathtt{j}) > \sigma(\mathtt{i})$
5) else if $\mathtt{esf}_{\mathtt{s_{2k}},\mathtt{s_{1k}},\mathtt{c_k}}$ then $\sigma(\mathtt{i}) > \sigma(\mathtt{j})$
6) else (equal sequences)
7) if $k = l$ (all ordering expressions are taken into account)
8) then if $\mathtt{i} < \mathtt{j}$ then $\sigma(\mathtt{i}) < \sigma(\mathtt{j})$ else $\sigma(\mathtt{j}) < \sigma(\mathtt{i})$
9) else start item 1 with $k = k + 1$
10) else (empty sequence last)
11) if $\mathtt{esl}_{\mathtt{s_{1k}},\mathtt{s_{2k}},\mathtt{c_k}}$ then $\sigma(\mathtt{i}) > \sigma(\mathtt{j})$
12) else if $\mathtt{esl}_{\mathtt{s_{2k}},\mathtt{s_{1k}},\mathtt{c_k}}$ then $\sigma(\mathtt{j}) > \sigma(\mathtt{i})$
13) else (equal sequences)
14) if $k = l$ (all ordering expressions are taken into account)
15) then if $\mathtt{i} < \mathtt{j}$ then $\sigma(\mathtt{i}) < \sigma(\mathtt{j})$ else $\sigma(\mathtt{j}) < \sigma(\mathtt{i})$
16) else start item 1 with $k = k + 1$
17) else (descending order)
18) if $\mathtt{b_k} = \ '\uparrow'$ (empty sequence first)
19) then if $\mathtt{esf}_{\mathtt{s_{1k}},\mathtt{s_{2k}},\mathtt{c_k}}$ then $\sigma(\mathtt{j}) < \sigma(\mathtt{i})$
20) else if $\mathtt{esf}_{\mathtt{s_{2k}},\mathtt{s_{1k}},\mathtt{c_k}}$ then $\sigma(\mathtt{i}) < \sigma(\mathtt{j})$
21) else (equal sequences)
22) if $k = l$ (all ordering expressions are taken into account)
23) then if $\mathtt{i} < \mathtt{j}$ then $\sigma(\mathtt{i}) < \sigma(\mathtt{j})$ else $\sigma(\mathtt{j}) < \sigma(\mathtt{i})$
24) else start item 1 with $k = k + 1$
25) else (empty sequence last)
26) if $\mathtt{esl}_{\mathtt{s_{1k}},\mathtt{s_{2k}},\mathtt{c_k}}$ then $\sigma(\mathtt{j}) < \sigma(\mathtt{i})$
27) else if $\mathtt{esl}_{\mathtt{s_{2k}},\mathtt{s_{1k}},\mathtt{c_k}}$ then $\sigma(\mathtt{i}) < \sigma(\mathtt{j})$
28) else (equal sequences)
29) if $k = l$ (all ordering expressions are taken into account)
30) then if $\mathtt{i} < \mathtt{j}$ then $\sigma(\mathtt{i}) < \sigma(\mathtt{j})$ else $\sigma(\mathtt{j}) < \sigma(\mathtt{i})$
31) else start item 1 with $k = k + 1$

For the second expression, the ordering is defined in the same way with the exception that there should be

$\sigma(\mathtt{i}) < \sigma(\mathtt{j})$ *or* $\sigma(\mathtt{i}) > \sigma(\mathtt{j})$
instead of *if-then-else* clauses in lines 8, 15, 23, and 30.

Example. If *books* denotes a sequence of type $Seq($**rec** *book* : $Element, price$: $Seq(Integer)$ **end**, then the following expression indicates ordering the records in the descending order of book prices (records without indicated prices last):
order$(price[\downarrow, \downarrow, ()] : books)$.

6.6. Mapping expression

This expression denotes the result of a FLWOR query. The constructor of this expression takes a sequence of tuples (records) and an expression and produces a final sequence by evaluating the expression on each tuple of the first sequence. Formally: if s is a Σ-expression of type $Seq($**rec** $x_1 : t_1$, ..., $x_n : t_n$ **end**) and e a $(\Sigma, \{x_1, ..., x_n\})$-expression either of type t or $Seq(t)$, then $s_1 \triangleright e$ is a Σ-expression of type $Seq(t)$, called a *mapping expression.*

Interpretation. Let $[\![s]\!]^{\mathtt{A}} = \langle \mathtt{A}', (\mathtt{r}_1, ..., \mathtt{r}_\mathtt{m})\rangle$, i = 1, ..., m,
$\mathtt{r}_\mathtt{i} = \mathtt{rec}(\mathtt{v}_{\mathtt{i}1}, ..., \mathtt{v}_{\mathtt{i}n})$, $\xi_\mathtt{i} = \{x_1 \mapsto \mathtt{v}_{\mathtt{i}1}, ..., x_n \mapsto \mathtt{v}_{\mathtt{i}n}\}$,
$e' = (e)$ if e has type t and $e' = e$ in the opposite case,
$[\![e'\xi_1]\!]^{\mathtt{A}'} = \langle \mathtt{A}^1, \mathtt{v}_1\rangle$, ... $[\![e'\xi_m]\!]^{\mathtt{A}^{\mathtt{m}-1}} = \langle \mathtt{A}^\mathtt{m}, \mathtt{v}_\mathtt{m}\rangle$, then $[\![s \triangleright e]\!]^{\mathtt{A}} = \langle \mathtt{A}^\mathtt{m}, \mathtt{v}_1 + ... + \mathtt{v}_\mathtt{m}\rangle$.
Note that the resulting sequence retains the order of the input sequence.

Example. Assume the variable *proc* denotes a sequence of proceedings nodes, and we want to pose the following query: "*find the titles of all proceedings whose editors have not have a publication in the proceedings they have edited.*". It can represented by the following expression:
select$(\lessdot x : proc \gtrdot * \lessdot y : x.child :: element(editor) \gtrdot *$
$\lessdot z : x.descendant :: element(author) \gtrdot) :: y \neq z \triangleright x.child :: element(title)$.
The first operator "*" creates a stream of pairs of $(proc, author)$ nodes, the second operator "*" converts it into a stream of triples of $(proc, author, title)$ nodes, the predicate $y \neq z$ selects in the stream those tuples where *editor* and *author* are different nodes, and finally the operator "$\triangleright$" produces the sequence of the titles of the remaining proceedings.

6.7. Sequence expressions

XQuery possesses a number of sequence constructing and manipulating expressions. They are supported in our algebra by several expressions defined as follows.

1. If $e_1, ..., e_n$ are Σ-expressions of respective types $t_1, ..., t_n$, where t_i is either $Seq(t'_i)$ or t'_i where t'_i is an atomic or node type, then $seq(e_1, ..., e_n)$ is a Σ-expression of type $Seq(t)$ where $t = Union(t'_1, ..., t'_n)$.

Interpretation. Let e'_i be e_i if e_i is a sequence, and (e_i) if it is an item, $\mathtt{A}$ an algebra, and $[\![e'_1]\!]^{\mathtt{A}} = \langle \mathtt{A^1}, \mathtt{e_1} \rangle$, ..., $[\![e'_n]\!]^{\mathtt{A^{n-1}}} = \langle \mathtt{A^n}, \mathtt{e_n} \rangle$, then

$$[\![seq(e_1, ..., e_n)]\!]^{\mathtt{A}} = \langle \mathtt{A^n}, \mathtt{e_1} + ... + \mathtt{e_n} \rangle.$$

2. If e_1 and e_2 are Σ-expressions of type $Integer$, then $range(e_1, e_2)$ is a Σ-expression of type $Seq(Integer)$.

Interpretation. Let $\mathtt{A}$ be an algebra and $[\![e_1]\!]^{\mathtt{A}} = \langle \mathtt{A^1}, \mathtt{e_1} \rangle$, $[\![e_2]\!]^{\mathtt{A^1}} = \langle \mathtt{A^2}, \mathtt{e_2} \rangle$, then

a) $[\![range(e_1, e_2)]\!]^{\mathtt{A}} = \langle \mathtt{A^2}, (\mathtt{v_1}, \mathtt{v_2}, ..., \mathtt{v_n}) \rangle$,
where $\mathtt{v_1} = \mathtt{e_1}$, $\mathtt{v_n} = \mathtt{e_2}$, and $\mathtt{v_{i+1}} = \mathtt{v_i} + 1$, for $i = 1, ..., n-1$, if $\mathtt{e_1} \leq \mathtt{e_2}$;

b) $[\![range(e_1, e_2)]\!]^{\mathtt{A}} = ()$, otherwise.

3. If s_1 and s_2 are expressions of type $Seq(Node)$ then

$$union(s_1, s_2), intersect(s_1, s_2), \text{ and } except(s_1, s_2)$$

are expressions of type $Seq(Node)$ interpreted as follows. Let $\mathtt{A}$ be an algebra, $[\![s_1]\!]^{\mathtt{A}} = \langle \mathtt{A^1}, \mathtt{s_1} \rangle$, $[\![s_2]\!]^{\mathtt{A}} = \langle \mathtt{A^2}, \mathtt{s_2} \rangle$, then

$[\![union(s_1, s_2)]\!]^{\mathtt{A}} = \langle \mathtt{A^2}, \mathtt{f(asSeq(asSet(s_1) \cup asSet(s_2)))} \rangle$;
$[\![intersect(s_1, s_2)]\!]^{\mathtt{A}} = \langle \mathtt{A^2}, \mathtt{f(asSeq(asSet(s_1) \cap asSet(s_2)))} \rangle$;
$[\![except(s_1, s_2)]\!]^{\mathtt{A}} = \langle \mathtt{A^2}, \mathtt{f(asSeq(asSet(s_1) \backslash asSet(s_2)))} \rangle$;

where `f` is `document_order` if `order_mode = ordered`, and identity function in the opposite case.

4. If s_1 and s_2 are expressions of type $Seq(anyAtomicType)$ and $\odot$ is one of the relation symbols "=", "!=", "<", "<=", ">", or ">=", then $s_1 \odot s_2$ is an expression of type $Boolean$.

Interpretation. This expression implements the operation of *general comparison*. It consists in existential comparison of the sequences' components so that the expression evaluates to true iff the relation holds for at least one pair of the components. Since the components may have different types, the comparison may need casting of one operand to the type of the other operand. We use for this purpose the function *fs:convert-operand* defined in Section 7.1.3 of [24] and define the semantics of the expression as the semantics of the expression constructed of the components defined above:

$[\![s_1 \odot s_2]\!]^{\mathtt{A}} = [\![\mathbf{exists}(x_1 : s_1, x_2 : s_2)!$
$(\lessdot y_1,\ \mathit{fs{:}convert\text{-}operand}(x_1, x_2) \gtrdot * \lessdot y_1,\ \mathit{fs{:}convert\text{-}operand}(x_2, x_1))\triangleright$
$y_1 \odot y_2]\!]^{\mathtt{A}}$.

7. NODE CONSTRUCTORS

This is a set of expressions copying existing nodes or constructing new nodes. The interpretation of these expressions updates the current algebra and produces an element of the new algebra. Therefore, we use a notion of pair $\langle \mathtt{A}, \mathtt{v} \rangle$, where $\mathtt{A}$ is an algebra and $\mathtt{v} \in |\mathtt{A}|$ is an algebra element, as the result of interpretation. The set of all pairs $\langle \mathtt{A}, \mathtt{v} \rangle$ where $\mathtt{A}$ is a Σ-algebra and $\mathtt{v}$ a value of type t is denoted by $\mathcal{A}_t(\Sigma)$. The functions `fst` and `snd` applied to such a pair produce its first and second component, respectively.

The signature of our algebras is supposed to include the data type $\mathit{constructionMode} = \mathit{Enumaration}(\mathit{strip}, \mathit{preserve})$ and the constant $\mathit{con\_mode} : \mathit{constructionMode}$ that governs the values of some node accessors (they may have different values in different algebras depending on whether $\mathit{con\_mode}$ is set to `strip` or `preserve` in the algebra).

7.1. Node copying

This facility is used in XQuery where parts of existing document trees are used in the construction of new elements or documents.

If s is an expression of type $\mathit{Seq}(\mathit{Node})$, then $\mathit{copy\_node}(s)$ and $\mathit{copy\_nodes}(s)$ are expressions of type $\mathit{Seq}(\mathit{Node})$. In a Σ-state $\mathtt{A}$ they are respectively interpreted by the functions

$$\mathtt{copy\_node}^{\mathtt{A}} : \mathtt{A}_{\mathtt{Node}} \times \mathtt{A}_{\mathtt{Element}} \to \mathcal{A}_{\mathit{Node}}(\Sigma)$$

and

$$\mathtt{copy\_nodes}^{\mathtt{A}} : \mathtt{A}_{\mathtt{Seq(Node)}} \times \mathtt{A}_{\mathtt{Element}} \to \mathcal{A}_{\mathit{Seq(Node)}}(\Sigma)$$

as follows[6]:

$[\![\mathit{copy\_node}(s)]\!]^{\mathtt{A}} = \mathtt{copy\_node}^{\mathtt{A}}(\mathtt{s}, \mathtt{NULL})$,
$[\![\mathit{copy\_nodes}(s)]\!]^{\mathtt{A}} = \mathtt{copy\_nodes}^{\mathtt{A}}(\mathtt{s}, \mathtt{NULL})$,

where `NULL` indicates a nonexisting node. The function $\mathtt{copy\_nodes}^{\mathtt{A}}$ is defined with the use of the function $\mathtt{copy\_node}^{\mathtt{A}}$ as follows. Let

[6]The first argument of both functions is/are the node/nodes to be copied, and the second argument, if not `NULL`, is the parent node of each new node.

$\mathtt{end} \in \mathtt{A}_{\mathtt{Element}}$, $\mathtt{s} = (\mathtt{nd}_1, ..., \mathtt{nd}_\mathtt{m})$,
$\mathtt{copy\_node}^{\mathtt{A}}(\mathtt{nd}_1, \mathtt{end}) = \langle \mathtt{A}^1, \mathtt{nd}'_1 \rangle$, ..., $\mathtt{copy\_node}^{\mathtt{A}^{\mathtt{m}-1}}(\mathtt{nd}_\mathtt{m}, \mathtt{end}) = \langle \mathtt{A}^\mathtt{m}, \mathtt{nd}'_\mathtt{m} \rangle$,
then

$$\mathtt{copy\_nodes}^{\mathtt{A}}((\mathtt{nd}_1, ..., \mathtt{nd}_\mathtt{m}), \mathtt{end}) = \langle \mathtt{A}^\mathtt{m}, (\mathtt{nd}'_1, ..., \mathtt{nd}'_\mathtt{m}) \rangle.$$

Several auxiliary functions are used in the definition of `copy_node` as follows.

If $\mathtt{nd} \in \mathtt{A}_{\mathtt{Node}}$ and $\mathtt{end} \in \mathtt{A}_{\mathtt{Element}}$, then

$$\mathtt{copy\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) = \begin{cases} \mathtt{copy\_text\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) \text{ if } \mathtt{nd} \in \mathtt{A}_{\mathtt{Text}} \\ \mathtt{copy\_attribute\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) \text{ if } \mathtt{nd} \in \mathtt{A}_{\mathtt{Attribute}} \\ \mathtt{copy\_element\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) \text{ if } \mathtt{nd} \in \mathtt{A}_{\mathtt{Element}} \\ \mathtt{copy\_document\_node}^{\mathtt{A}}(\mathtt{nd}) \text{ if } \mathtt{nd} \in \mathtt{A}_{\mathtt{Document}} \end{cases}$$

The function

$$\mathtt{copy\_text\_node}^{\mathtt{A}} : \mathtt{A}_{\mathtt{text}} \times \mathtt{A}_{\mathtt{Element}} \to \mathcal{A}_{Text}(\Sigma)$$

produces a clone of a text node. Definition:

$$\mathtt{copy\_text\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) = \langle \mathtt{A}', \mathtt{nd}' \rangle,$$

where $\mathtt{nd}'$ is a node such that $\mathtt{nd}' \notin \mathtt{A}_{\mathtt{Text}}$, and $\mathtt{A}'$ is the following extension of $\mathtt{A}$:

1) $\mathtt{A}'_{\mathtt{Text}} = \mathtt{A}_{\mathtt{Text}} \cup \{\mathtt{nd}'\}$;
2a) $\mathtt{parent}(\mathtt{nd}') = \mathtt{end}$ and $\mathtt{children}^{\mathtt{A}'}(\mathtt{end}) = \mathtt{children}^{\mathtt{A}}(\mathtt{end}) + (\mathtt{nd}')$ if $\mathtt{end} \neq \mathtt{NULL}$;
2b) $\mathtt{parent}(\mathtt{nd}') = ()$, otherwise
3) $\mathtt{acc}(\mathtt{nd}') = \mathtt{acc}(\mathtt{nd})$ for any other node accessor $\mathtt{acc}$.

The function

$$\mathtt{copy\_attribute\_node}^{\mathtt{A}} : \mathtt{A}_{\mathtt{Attribute}} \times \mathtt{A}_{\mathtt{Element}} \to \mathcal{A}_{Attribute}(\Sigma)$$

produces a clone of an attribute node.

Definition:

$$\mathtt{copy\_attribute\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) = \langle \mathtt{A}', \mathtt{nd}' \rangle,$$

where $\mathtt{nd}'$ is a node such that $\mathtt{nd}' \notin \mathtt{A}_{\mathtt{Attribute}}$, and $\mathtt{A}'$ is the following extension of $\mathtt{A}$:

1) $\mathtt{A}'_{\mathtt{Attribute}} = \mathtt{A}_{\mathtt{Attribute}} \cup \{\mathtt{nd}'\}$;
2a) $\mathtt{parent}(\mathtt{nd}') = \mathtt{end}$ and $\mathtt{attributes}^{\mathtt{A}'}(\mathtt{end}) = \mathtt{attributes}^{\mathtt{A}}(\mathtt{end}) + (\mathtt{nd}')$ if $\mathtt{end} \neq \mathtt{NULL}$;
2b) $\mathtt{parent}(\mathtt{nd}') = ()$, otherwise;
3a) `type-name(nd') = xdt:untypedAtomic`,

$\mathtt{string\text{-}value(nd')} = \mathtt{string\text{-}value(nd)}$,
$\mathtt{typed\text{-}value(nd')} = \mathtt{string\text{-}value(nd')}$, and $\mathtt{acc(nd')} = \mathtt{acc(nd)}$ for any other node accessor $\mathtt{acc}$ if $\mathtt{con\_mode} = \mathtt{strip}$;

3b) $\mathtt{acc(nd')} = \mathtt{acc(nd)}$ for any other node accessor $\mathtt{acc}$ if $\mathtt{con\_mode} = \mathtt{preserved}$.

The function

$$\mathtt{copy\_element\_node}^{\mathtt{A}} : \mathtt{A}_{\mathtt{Element}} \times \mathtt{A}_{\mathtt{Element}} \to \mathcal{A}_{Element}(\Sigma)$$

produces a clone of an element node. Definition. Assume for simplicity the following order of node copying: the element node itself, its attribute nodes, its child nodes. Then

$$\mathtt{copy\_element\_node}^{\mathtt{A}}(\mathtt{nd}, \mathtt{end}) = \langle \mathtt{A}', \mathtt{nd}' \rangle,$$

where $\mathtt{nd}'$ is a node such that $\mathtt{nd}' \notin |\mathtt{A}|$, and $\mathtt{A}'$ is an extension of $\mathtt{A}$ produced as follows. Let

1. $\mathtt{A}^1$ be an extension of $\mathtt{A}$ such that
   - $\mathtt{nd}' \in \mathtt{A}^1_{\mathtt{Element}}$,
     - $\mathtt{parent(nd')} = \mathtt{end}$ and $\mathtt{children}^{\mathtt{A}^1}(\mathtt{end}) = \mathtt{children}^{\mathtt{A}}(\mathtt{end}) + (\mathtt{nd}')$ if $\mathtt{end} \neq \mathtt{NULL}$;
     - $\mathtt{parent(nd')} = ()$, otherwise;
   - $\mathtt{children(nd')} = (), \mathtt{attributes(nd')} = ()$,
   - one of the following alternatives holds:
     - $\mathtt{type\text{-}name(nd')} = \mathtt{xdt{:}untyped}$, $\mathtt{string\text{-}value(nd')} = \mathtt{string\text{-}value(nd)}$, $\mathtt{typed\text{-}value(nd')} = \mathtt{string\text{-}value(nd')}$, $\mathtt{nilled(nd')} = \mathtt{false}$, and $\mathtt{acc(nd')} = \mathtt{acc(nd)}$ for any other node accessor $\mathtt{acc}$ if $\mathtt{con\_mode} = \mathtt{strip}$;
     - $\mathtt{acc(nd')} = \mathtt{acc(nd)}$ for any other node accessor $\mathtt{acc}$ if $\mathtt{con\_mode} = \mathtt{preserved}$.
2. $\mathtt{A}^{\mathtt{at_m}} = \mathtt{fst}(\mathtt{copy\_nodes}^{\mathtt{A}^1}(\mathtt{attributes(nd)}, \mathtt{nd}'))$;

then $\mathtt{A}' = \mathtt{fst}(\mathtt{copy\_nodes}^{\mathtt{A}^{\mathtt{at_m}}}(\mathtt{children(nd)}, \mathtt{nd}'))$.

The function

$$\mathtt{copy\_document\_node}^{\mathtt{A}} : \mathtt{A}_{\mathtt{Document}} \to \mathcal{A}_{Document}(\Sigma)$$

produces a clone of a document node. Definition. Assume for simplicity the following order of node copying: the document node itself, its child nodes. Then

$$\mathtt{copy\_document\_node}^{\mathtt{A}}(\mathtt{nd}) = \langle \mathtt{A}', \mathtt{nd}' \rangle,$$

where $\mathtt{nd}'$ is a node such that $\mathtt{nd}' \notin |\mathtt{A}|$, and $\mathtt{A}'$ is an extension of $\mathtt{A}$ produced as follows.

Let $\mathtt{A}^1$ be an extension of $\mathtt{A}$ such that

- $\mathtt{nd}' \in \mathtt{A}^1_{\mathtt{Document}}$,
- $\mathtt{parent}(\mathtt{nd}') = ()$, $\mathtt{children}(\mathtt{nd}') = ()$ and $\mathtt{acc}(\mathtt{nd}') = \mathtt{acc}(\mathtt{nd})$ for any other node accessor $\mathtt{acc}$,

then $\mathtt{A}' = \mathtt{fst}(\mathtt{copy\_nodes}^{\mathtt{A}^1}(\mathtt{children}(\mathtt{nd}), \mathtt{nd}'))$.

7.2. Attribute node constructor

This is a node constructing expression whose interpretation produces a new attribute node on the base of a name and string value supplied in a query. Definition: If n is a $QName$ and e a $String$, then $attribute\_node(n, e)$ is an expression of type $Attribute$ interpreted as follows.

$[\![attribute\_node(n, e)]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{nd} \rangle$,

where $\mathtt{nd}$ is a node such that $\mathtt{nd} \notin \mathtt{A}_{\mathtt{Attribute}}$, and $\mathtt{A}'$ is the following extension of $\mathtt{A}$:

1) $\mathtt{A}'_{\mathtt{Attribute}} = \mathtt{A}_{\mathtt{Attribute}} \cup \{\mathtt{nd}\}$;
2) `node-name(nd) = n`, `node-kind(nd)` = "`attribute`", `parent(nd) = ()`, `type-name(nd)` = "`xdt:untypedAtomic`", `string-value(nd) = e`, `typed-value(nd)` = `string-value(nd)`.

See an example in Section 7.4.

7.3. Text node constructor

This is a node constructing expression whose interpretation produces a new text node on the base of a string value supplied in a query. Definition: If e is a $String$, then $text\_node(e)$ is an expression of type $Text$ interpreted as follows.

$[\![text\_node(e)]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{nd} \rangle$, where $\mathtt{nd}$ is a node such that $\mathtt{nd} \notin \mathtt{A}_{\mathtt{Text}}$, and $\mathtt{A}'$ is the following extension of $\mathtt{A}$:

1) $\mathtt{A}'_{\mathtt{Text}} = \mathtt{A}_{\mathtt{Text}} \cup \{\mathtt{nd}\}$;
2) `node-name(nd) = ()`, `node-kind(nd)` = "`text`", `parent(nd) = ()`, `type-name(nd)` = "`xdt:untypedAtomic`", `string-value(nd) = e`, `typed-value(nd)` = `string-value(nd)`.

See examples in Section 7.4.

7.4. Element node constructors

There are two forms of element node constructing expressions.
1. The first one constructs an element with simple content on the base of a string value supplied in a query. Definition: If n is a $QName$, $atseq$ an expression of type $Seq(Attribute)$[7], and e an expression of type $Text$ such that $parent(e) = ()$, then $element\_node(n, atseq, e)$ is an expression of type $Element$.

Interpretation. If $m > 0$ then let $[\![atseq]\!]^{\mathtt{A}} = \langle \mathtt{A}^{\mathtt{m}}, (\mathtt{and}_1, ..., \mathtt{and}_{\mathtt{m}})\rangle$, and $[\![e]\!]^{\mathtt{A}^{\mathtt{m}}} = \langle \mathtt{A}^{\mathtt{txt}}, \mathtt{tnd}\rangle$. Then:

$[\![element\_node(n, atseq, e)]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{nd}\rangle$,

where `nd` is an element such that $\mathtt{nd} \notin |\mathtt{A}^{\mathtt{txt}}|$, and $\mathtt{A}'$ is an extension of $\mathtt{A}^{\mathtt{txt}}$ produced as follows.

1. $\mathtt{A}'_{\mathtt{Element}} = \mathtt{A}^{\mathtt{txt}}_{\mathtt{Element}} \cup \{\mathtt{nd}\}$;
2. `node-name(nd) = n`, `node-kind(nd)` = "`element`", `parent(nd) = ()`, `string-value(nd)` = `string-value(tnd)`, `typed-value(nd)` = `string-value(nd)`, `children(nd) = (tnd)`, `nilled(nd) = false`, `parent(tnd) = nd`;
3. `type-name(nd)` = "`xdt:untyped`" if `con_mode = strip`, and `type-name(nd)` = "`xs:anyType`" if `con_mode = preserved`,
4. `attributes(nd)` $= \{\mathtt{and}_1, ..., \mathtt{and}_{\mathtt{m}}\}$ and for each $\mathtt{and} \in \{\mathtt{and}_1, ..., \mathtt{and}_{\mathtt{m}}\}$: `parent(and) = nd`.

If $m = 0$ (no attribute is associated with the element), then the above mentioned $\mathtt{A}^{\mathtt{m}} = \mathtt{A}$ and the item 4 is as follows:

4. `attributes(nd) = ()`.

Example. The following fragment of the XML text:

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

can be represented by the following element constructor:

```
element_node(title, (), text_node("Data on the Web"))
```

2. The second one constructs an element with complex content. Definition: If n is a $QName$, $atseq$ an expression of type $Seq(Attribute)$[8], and $elseq$ an expression of type $Seq(Union(Element, Text))$ such that if $type(elseq[i]) = Text$ then $type(elseq[i]) = Element$ (no adjacent text

[7] Constraints: 1) if $n_i = node\text{-}name(atseq[i])$, $n_j = node\text{-}name(atseq[j])$ and $i \neq j$, then $n_i \neq n_j$; 2) $parent(atseq[i]) = ()$. The constrains let one make sure that attributes have different names and none of them is part of an existing tree.

[8] See the above constraint.

nodes are allowed) and $parent(elseq[i]) = ()$ for any $i = 1, ..., |elseq|$, then $element\_node(n, atseq, elseq)$ is an expression of type *Element*.

Interpretation. If $|atseq| > 0$, then let $[\![atseq]\!]^{\mathtt{A}} = \langle \mathtt{AA^m}, (\mathtt{and_1}, ..., \mathtt{and_m})\rangle$, and if $|elseq| > 0$, then let $[\![elseq]\!]^{\mathtt{AA^m}} = \langle \mathtt{EA^k}, (\mathtt{end_1}, ..., \mathtt{end_k})\rangle$. Then:

$[\![element\_node(n, atseq, elseq)]\!]^{\mathtt{A}} = \langle \mathtt{A'}, \mathtt{nd}\rangle$,

where `nd` is an element such that $\mathtt{nd} \notin |\mathtt{EA^k}|$, and $\mathtt{A'}$ is an extension of $\mathtt{EA^k}$ produced as follows.

1. $\mathtt{A'_{Element}} = \mathtt{EA^k_{Element}} \cup \{\mathtt{nd}\}$;
2. `node-name(nd) = n`, `node-kind(nd)` = "`element`",
   `parent(nd) = ()`;
3. `type-name(nd)` = "`xdt:untyped`" if `con_mode = strip`, and
   `type-name(nd)` = "`xs: anyType`" if `con_mode = preserved`,
4. `attributes(nd)` $= \{\mathtt{and_1}, ..., \mathtt{and_m}\}$ and for each
   $\mathtt{and} \in \{\mathtt{and_1}, ..., \mathtt{and_m}\}$: $\mathtt{parent}^{\mathtt{A'}}(\mathtt{and}) = \mathtt{nd}$;
5. `string-value(nd) = string-value(end`$_1$`)+...`
   `+string-value(end`$_k$`)`,
   `typed-value(nd)` = `string-value(nd)`,
   `children(nd)` $= (\mathtt{end_1}, ..., \mathtt{end_k})$, `nilled(nd) = false`, and for each
   $\mathtt{end} \in \{\mathtt{end_1}, ..., \mathtt{end_k}\}$: $\mathtt{parent}^{\mathtt{A'}}(\mathtt{end}) = \mathtt{nd}$.

If $|atseq| = 0$ (no attribute is associated with the element), then the above mentioned $\mathtt{AA^m} = \mathtt{A}$ and the item 4 is as follows:

4. `attributes(nd) = ()`.

If $|elseq| = 0$ (neither an element or a text is associated with the element), then the above mentioned $\mathtt{EA^k} = \mathtt{AA^m}$ and the item 5 is as follows:

5. `string-value(nd) = ()`, `typed-value(nd)` = (),
   `children(nd) = ()`, `nilled(nd) = false`.

Example. The following fragment of the XML text:

```
<book> year="1992">
   <title>Data on the Web</title>
   <author>Abiteboul</author>
   <author>Buneman</author>
   <author>Suciu</author>
   <publisher>Morgan Kaufman Publishers</publisher>
   <price>65.95</price>
</book>
```

can be represented by the following element constructor:

```
element_node(book, (attribute_node(year, "1992")),
   (element_node(title, (), text_node("Data on the Web"))
    element_node(author, (), text_node("Abiteboul")),
    element_node(author, (), text_node("Buneman")),
    element_node(author, (), text_node("Suciu")),
    element_node(publisher, (),
                    text_node("Morgan Kaufman Publishers")),
    element_node(price, (), text_node("65.95")))).
```

A more complex example. The following XQuery query transforms a `bib` document (bound to the variable `$bib`) into a list in which each author's name appears only once, followed by a list of titles of books written by that author. The `fn:distinct-values` function is used to eliminate duplicates (by value) from a list of author nodes. The author list, and the lists of books published by each author, are returned in alphabetic order using the default collation.

```
<authlist>
 {
    for $a in fn:distinct-values($bib/book/author)
    order by $a
    return
       <author>
          <name> $a </name>
          <books>
           {
              for $b in $bib/book[author = $a]
              order by $b/title
              return $b/title
           }
          </books>
       </author>
 }
</authlist>
```

The query can be represented in the algebra as follows:

```
element_node(authlist, (),
   order(typed_value(a)[↑,↑,""]:
           ⩽a:  fn:distinct-values(path(x:  bib)/
                    path(y:  x.child::element(book))/
                       y.child::element(author))⩾)▷
      element_node(author, (),
         (element_node(name, (), text_node(string_value(a))),
          element_node(books, (), copy_nodes(
            order(typed_value(b/title)[↑,↑,""]:
             ⩽b:  path(x:  bib)/
                    select(y:  x.child::element(book))::
                       a ∈ y.child::element(author) ⩾▷
               b/title))
          )
         )
      )
)
```

7.5. Document node constructors

The result of the document node constructor is a new document node whose children are element and/or text nodes. Definition: If *elseq* is an expression of type $Seq(Union(Element, Text))$ such that if $type(elseq[i]) = Text$ then $type(elseq[i]) = Element$ (no adjacent text nodes are allowed), then $document\_node(el_1, ..., el_k)$ is an expression of type *Document*.

Interpretation. If $|elseq| > 0$, then let $[\![elseq]\!]^{\mathtt{A}} = \langle \mathtt{A}^{\mathtt{k}}, \mathtt{cnd}_1, ..., \mathtt{cnd}_{\mathtt{k}} \rangle$. Then

$[\![document\_node(elseq)]\!]^{\mathtt{A}} = \langle \mathtt{A}', \mathtt{nd} \rangle$,

where `nd` is an element such that $\mathtt{nd} \notin |\mathtt{A}^{\mathtt{k}}|$, and $\mathtt{A}'$ is an extension of $\mathtt{A}^{\mathtt{k}}$ produced as follows.

1. $\mathtt{A}'_{\mathtt{Document}} = \mathtt{A}^{\mathtt{k}}_{\mathtt{Document}} \cup \{\mathtt{nd}\}$;
2. `node-name(nd) = ()`, `node-kind(nd)` = "`document`",
   `parent(nd)` = (), `type-name(nd) = ()`, `attributes(nd) = ()`;
3. `string-value(nd)` = `string-value(cnd`$_1$`)+...`
   `+string-value(cnd`$_{\mathtt{k}}$`)`,
   `children(nd)` = ($\mathtt{cnd}_1, ..., \mathtt{cnd}_{\mathtt{k}}$) and for each $\mathtt{cnd} \in \{\mathtt{cnd}_1, ..., \mathtt{cnd}_{\mathtt{k}}\}$,
   `parent(cnd) = nd`.

If $|elseq| = 0$ (neither an element nor a text is associated with the document), then the above mentioned $\mathtt{A}^{\mathtt{k}} = \mathtt{A}$ and the item 3 is as follows:

3. `string-value(nd) = ()`, `children(nd) = ()`.

Example. The XQuery query

```
document
   {
      <author-list>
         fn:doc("bib.xml"/bib/book/author)
      </author-list>
   }
```

returning an XML document containing a root element named `author-list` is represented by the following algebra expression:

```
document_node((element(author_list, (),
            copy_nodes(fn:doc("bib.xml"/bib/book/author)))))
```

8. RELATED WORK

One of the first works presenting an XML algebra is [7] (an updated version of this work was proposed as a working draft of W3C [19]). The following operations are defined there: projection, selection, and iteration. The projection operation uses a sequence and an element name to produce a sequence of children nodes, i.e., it is a special case of our path expression where the second argument is the name of a child element rather than a sequence. The selection expression permits inspection of a sequence with selection of some of its items satisfying a predicate. There is no facility to select items basing on kind tests. The iteration operation permits processing a sequence of items, one item at a time. It resembles the conventional for-loop of an imperative programming language. In fact, this is the main operation of the algebra. The authors show how nested for-loops can be used to provide restructuring and joining of existing documents and, moreover, how projection can be formally expressed by iteration. There is no algebraic definition of any operation. One can say that just a simple query language is defined that has no relation to XQuery and cannot be used for defining its semantics.

A number of algebras were proposed in the process of design and development of the database system TIMBER [10]. A tree algebra, called TAX, is described in [9]. According to TAX, the database is a collection (set) of ordered labeled trees. For this reason, all operations of this algebra take collections of trees as input and produce a collection of trees as output. The algebra thus uses more complex data structures (trees) compared to

our algebra and therefore it is much more heavier. It is noted in [3] that the direct set-oriented evaluation of XQuery is possible, but can get quite complicated even for simple queries.

The complexity of the algebra has forced the authors of TAX to design, in addition, a lower-level algebra, called *physical algebra* (reported in the unpublished paper [15]), manipulating sequences of trees and serving for implementation of the TAX algebra. It is assumed that each of the operations of the physical algebra is likely to be available as an access method in any XML database. However, in the further development of the project the authors practically forgot of TAX and directly used an updated version of the physical algebra for implementation of a newly designed data structure, Generalized Tree Pattern [3], which represents an XQuery as a pattern consisting of one or more trees. All the operations of this algebra are described informally.

The next step in the project development was the introduction of the notion of a *tree logical class* as a labeled set of tree nodes matching a designated node and development of a new algebra, designed for manipulating tree logical classes [16]. The algebra uses the notion of *logical class reduction* for converting a heterogeneous set of trees into a homogenous set, thus eliminating the problem of performing set-oriented bulk operations on heterogeneous sets. However, there is no formal definition of the operations.

An XML algebra for data mining, called XAL, is reported in [28]. An XML document is regarded in XAL as a rooted directed graph with a partial relation on its edges. A XAL operation takes a set of nodes as input and produces a set of nodes as output. The main operations are selection, projection, product, and join. No detailed description of the operations is given.

A logical algebra and a physical algebra supporting XQuery are reported in [8]. It is claimed that an XQuery query is first translated into the logical algebra and then, after a possible optimization, is evaluated using the physical algebra. The logical algebra operates with ordered collections of primitive values or nodes or tuples of primitive values or nodes. Both algebras are described informally. Moreover, since many their operations use functions and predicates as operands, they are not algebras in fact.

An algebra whose operations work in a certain context is reported in the unpublished paper [21]. The context represents a point in an XML document during query evaluation. It is supplied to an operator as a path taken to reach its input. The authors argue that the use of the context

contributes to the construction of powerful query optimizers and evaluators. The input and output to all operations are *collections of collections* (called *outer* and *inner* collections, respectively). There is no hint in the paper on the motivation of the choice of the operations, which does not looks XQuery-oriented. For this reason the translation of an XQuery query to this algebra is not straightforward.

Another XML algebra, called XAT, is reported in the unpublished paper [27]. It is intended to support XQuery like the algebra described in this paper. The XAT data model represents data as hierarchical tables (collections of tuples). The set of XAT operators is divided in three groups: XML operators, SQL operators, and Special operators. The operators of the first group serve for performing the actions typical of the XML data model (`Navigate`, `Agregate`, `Composer`, etc.), the operators of the second group serve for performing SQL-like operations (`Progect`, `Select`, `Join`, `Groupbe`, etc.), and the operators of the third group serve to perform special functions like iterating through a collection of making a choice. No updating operator is reported. All operators are described informally, using examples. It is also informally described how an XQuery expression can be translated into the algebra. However, any subtle detail is ignored.

Relation-like flat tables are the main data structures used in the Xtasy algebra [18]. Each tuple in a table consists of variable-value pairs also referred to as bindings. The table is constructed by the *path* operator, which takes a database instance and *input filter* (a tree describing the paths to follow in the database, the variables ro bind, and the way to combine the results coming from different paths) as arguments. The opposite action is performed by the `return` operator, which takes a table and *output filter* (an element or attribute constructor) as arguments and produces an XML document. The other operators resemble the familiar relation algebra operators (Join, Selection, Projection, etc.). The semantics of all the operators are described informally. Only some simple XQuery queries can be represented by the algebra.

The tuple-oriented algebra described in the unpublished paper [14] resembles the previous one with the exception that the tuple can have a hierarchical structure, i.e., a tuple element can be a set of tuples. The algebra is also informally described.

An XML algebra designed for effective stream processing is briefly described in [2, 6]. The inputs and output of each operator of the algebra are streams represented as tuple sequences. *Projection* is similar to its re-

lational counterpart while *selection*, *join*, and *unnest* are similar to their object-oriented algebra counterparts proposed earlier by one of the authors of these papers [5]. Entirely new operators are *extraction*, which gets an XML data source and returns a singleton stream whose unique element contains the entire XML tree, and *reduce*, which produces the final result of a query in the form of an XML document. The semantics of the operators are defined by equations using list comprehensions and monoid calculus. Typing details are neglected. Only predicates are used as selection criteria. No navigating function is defined. Differences in paths expressions are not taken into account.

A set of primitive operations for modifying the structure and content of an XML document is proposed in [20]. It includes such operations as *delete* a node from a document tree, *rename* an existing node, *insert* a new node in the document tree, and *update* the content of a node. It is assumed that XQuery will be extended by these operations. The semantics of the operations are described informally.

9. CONCLUSION

We have presented an XML algebra supporting XQuery. The algebra is in fact a number of kinds of expressions (expression constructing operators) algebraically defined. The introduction of kinds of expressions instead of high-order operations using functions as parameters has permitted us to remain in the limits of first-order structures whose instance a many-sorted algebra is.

The set of kinds of expression of the presented algebra substantially differs from the set of operators of relation algebra. The difference is caused by the more complex structure of the XML document compared to the relation. In fact, only selection by predicate test is more or less the same in both algebras. At the same time, the XML algebra in addition permits selection by node test. The projection operator of relation algebra is replaced by the path expression and a number of navigating functions permitting selection of different parts of the document tree. The join operator is replaced by a number of unnesting join expressions permitting creation of a stream of flat tuples on the base of several possibly nested parts of the document tree.

In addition, we have defined a number of node constructing expressions permitting update of the current algebra by introduction of new nodes and corresponding node accessors. The evaluation of such an expression produces a new algebra as a side effect. Since XQuery allows expressions to be

nested with full generality, the evaluation of each expression theoretically may produce a side-effect. For this reason, the semantics of any expression in our approach is a pair, an algebra and a value, which corresponds one-to-one to the semantics of XQuery expressions. This feature is not present in any existing XML algebra. Another distinguishing feature of our algebra is that the first operand of many expressions (path, mapping, etc.) provides a context for the evaluation of the second operand, which may help in optimizing query performance.

Our algebra does not possess facilities corresponding to the branching and type-checking expressions of XQuery. As we have noted in Introduction, we consider these facilities more appropriate in the XQuery interpreter than in the XML algebra. Specification of such an interpreter is one of the directions of our future work.

In conclusion, the authors thank P. Emel'ianov for his valuable comments on the draft of the paper.

REFERENCES

1. M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002), Rome, Italy, September 17-21 2002*, LNCS, vol. 2461, pp. 152-164.
2. S. Bose, L. Fegaras, D. Levine, V. Chaluvadi. A Query Algebra for Fragmented XML Stream Data. *Proc. 9th Intnl. Conference on Databases and Programming Languages*, Posdam, Germant, Sept. 6-8, 2003.
3. Zhimin Chen, H.V. Jagadish, Laks V.S. Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. *Proc. VLDB Conf.*, Berlin, Germany, Sep. 2003.
4. H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol. 6, Springer, Berlin, 1985.
5. L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Transactions on Database Systems*, December 2000.
6. Leonidas Fegaras, David Levine, Sujoe Bose, Vamsi Chaluvadi. *Proc. 11th intern. conf. Information and Knowledge Management*, 2002, pp. 126-133.
7. M. Fernandez, J. Simeon, and P. Wadler. An Algebra for XML Query. *FST TCS, December 2000*, LNCS, vol. 1974, pp. 11-45.
8. D. Fisher, F. Lam, and R. K. Wong. Algebraic Transformation and Optimization for XQuery. *Advanced Web Technologies and Applications (Proc. 6th Asia-Pacific Web Conference, April 2004)*, LNCS, vol. 3007, pp. 201-210.
9. H. V. Jagodish, L. V. S. Lakshmanan, D. Srivastatva, and K. Thompson. Tax: A Tree Algebra for XML. *Proc. Intl. Workshop on Databases and Programming Languages*, Marino, Italy, Sept., 2001, pp. 149-164.
10. H.V.Jagadish, Shurug Al-Khalifa, Adriane Chapman, ea. TIMBER: A Native XML

Database. *The VLDB Journal*, Vol. 11 Issue 4 (2002) pp. 274-291
11. K. Lellahi, A.V. Zamulin. An object-oriented database as a dynamic system with implicit state. *A. Caplinskas and J. Eder (eds.). Advances in Databases and Information Systems (Proceedings of the 5th East European Conference, ADBIS 2001, Vilnus, Lithuania, September 2001)*, LNCS, vol. 2151, pp. 239-252.
12. L. Novak and A. Zamulin. Algebraic Semantics of XML Schema. *Preprint No. 117, Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences*, 2004; http://www.iis.nsk.su/persons/zamulin/zam-preprint117.ps.
13. L. Novak and A. Zamulin. Algebraic Semantics of XML Schema. *Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam (Eds.). Advances in Databases and Information Systems (Proceedings of the 9th East European Conference, ADBIS 2005, Tallin, Estonia, September 2005)*, LNCS, vol. ???, pp. ???.
14. Yannis Papakonstantinou, Vinayak Borkar, Maxim Orgiyan, Kostas Stathatos, Lucian Suta, Vasilis Vassalos, and Pavel Velikhov. *XML Queries and Algebra in the Enosys Integration Platform.* http://www.it.iitb.ac.in/ prasan/Courses/IT620/MISC/eip.pdf.
15. Stelios Paparizos, Shurug Al-Khalifa, H.V.Jagadish, Andrew Nierman and Yuqing Wu. *A Physical Algebra for XML.* http://www-personal.umich.edu/ spapariz/publications.html
16. Stelios Paparizos, Yuqing Wu, Laks V.S. Lakshmanan, and H.V. Jagadish. Tree Logical Classes for Efficient Evaluation of XQuery. *Proc. SIGMOD Conf.*, Jun. 2004, Paris, France.
17. . Jonathan Robie. XQUery: A Guided Tour. *Howard Katz (ed.). XQuery from the Experts*, Addison-Wesley, 2004, pp. 3-78.
18. Carlo Sartiani, Antonio Albano. Yet Another Query Algebra For XML Data. *IDEAS 2002*, pp. 106-115.
19. *The XML Query Algebra*, W3C Working Draft, 15 February 2001, http://www.w3.org/TR/2001/WD-query-algebra-20010215.
20. .I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. ACM SIGMOD*, 2001.
21. Stratis D. Viglas, Leonidas Galanis, David J. DeWitt, David Maier, Jeffrey F. Naughton. *Putting XML Query Algebras into Context.* http://www.cs.wisc.edu/niagara.
22. *XML Schema Part 1: Structures Second Edition*, W3C Working Draft, 28 October 2004, http://www.w3.org/TR/xmlschema-1/.
23. *XQuery 1.0: An XML Query Language.* W3C Working Draft, 4 April 2005, http://www.w3.org/TR/xquery.
24. *XQuery 1.0 and XPath 2.0 Formal Semantics*, W3C Working Draft 3 June 2005, http://www.w3.org/TR/xquery-semantics/.
25. *XQuery 1.0 and XPath 2.0 Data Model*, W3C Working Draft, 4 April 2005. http://www.w3.org/TR/xpath-datamodel/.
26. *XQuery 1.0 and XPath 2.0 Functions and Operators.* W3C Working Draft, 4 April 2005, http://www.w3.org/TR/xpath-functions.
27. X. Zhang and E. Rundensteiner. XAT: XML Algebra for Rainbow System. Worcester Polytechnic Institute, Technical Report WPI-CS-TR-02-24, July 2002.
28. M. Zhang, J.T. Yao. XML Algebra for Data Mining. *Proceedings of SPIE, Data Mining and Knowledge Discovery: Theory, Tools, and Technology VI*, 12-13 April 2004, Orlando, USA, vol. 5433, pp. 209-217.

Леонид Новак, Александр Замулин

XML-АЛГЕБРА ДЛЯ ЯЗЫКА ЗАПРОСОВ XQUERY

Препринт
125

Рукопись поступила в редакцию 21.05.2005
Рецензент П. Г. Емельянов
Редактор Т. М. Бульонкова

Подписано в печать 08.08.2005
Формат бумаги 60×84 1/16 Объем 2,3 уч.-изд.л., 2,5 п.л.
Тираж 60 экз.

ЗАО РИЦ "Прайс-курьер" 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 330-72-02