Assignments for XML-Documents

K. Benecke

***Abstract*— In the paper an important building stone for an end user XML-algebra is presented. We believe that a program, which is a term, build with user-friendly and expressive mass data operations, is user-friendly, too. In this paper assignments are considered. An assignment A:= B+C (or A:= B intersect C), for example, induces an inner extension, by which a new „column" A is introduced in an XML-document. Here, a term like B+C is evaluated for each pair of B- and C-values of the document, which belong together. The formalization of the corresponding inner-extensions is the main contribution of this paper. It is based on the initial algebraic approach and a corresponding specification of an XML-document as an abstract data type.**

***Index Terms*—Query Language, XML, Assignments**

I. Introduction

We believe that XQuery is a powerful tool for the manipulation of XML-documents, but it seems to us that XQuery is more a programming language than a query language for end users. In our opinion our XML-algebra is user-friendlier than XQuery, because we do not use recursive functions and nested loops. Further, we apply operations in an ordinary way. If we consider for example an XPath expression *document(XX.xml)//YY*, then a list of YY-values „results". If // would be an ordinary operation, then it would not be possible to go to the parents of YY, because these parents exist only in the original document but not in the result of //. Also therefore, we believe that XQuery with its sub language XPath is too complicated.

We think that our XML-algebra consists of powerful and user-friendly operations. One of the most interesting operations is *stroke*, which allows to restructure an XML-document into another document only by describing the scheme of the desired document. Sorting and aggregations of data can accompany this restructuring for example (for details compare [2], [3]).

Stroke is implemented for nested relations in C and for nested relations with optional values in CAML (compare [7] and [10]). The CAML implementation is based on an algebraic specification (compare [1]) and it includes the implementation of *stroke* as well as implementations of the XML-algebra operations *extension* (a supplementary operation to the *join*), *path* (for the solution of problems, like bill of material problems), *selection*, and inner extension (corresponds to an assignment). The implementation of these operations is based on a definition of non-first-normal-relation with optional values, too. In this paper the inner extension operation is generalized in several ways compared to its specification in [1]. Because a table has now not only names for elementary columns, we can use beside single data operations as addition, … also operations as intersection, ... Now each basic operation can be defined on (complex) tables and not only on simple "values". Second, if we extend a tuple, then it may occur - because of recursive structures - that this tuple has to be extended at several points. In order to find these points an additional argument of *inn-ext* is necessary. And third, it is possible - because of optional values - that if we extend a tuple at a certain position a corresponding value does not exist. Therefore, we have to extend the document sometimes by *A?* and not only by the new column *A*.

Section 1 contains a little introduction using explanatory examples. Then, problems in the design of the inner extension are sketched also on illustrating examples. The examples show that it is necessary to extend a document in some situations by a new column *A* and in others by a column *"A?"*. This results into the formal definition of two different operations *inn-ext1* and *inn-ext1?*.

In section 2 the essential parts of the specification of XML-documents of [4] are summarized, including the axioms that describe the sorts *Scheme* and *Table*.

The last section contains the axioms and further auxiliary operations of *inn-ext1?* in full length. Unfortunately, this specification is much longer and more complex than the specification in [1]. We try to "verify" these axioms soon using an OCAML implementation (compare [8]).

II. Introducing Examples

Query 1: Compute the area and circumference of a rectangle.

where A := 3.89
B := 7.98
AREA := A*B
CIRCUMFERENCE := 2*(A+B)

Result: <TUP0> <A>3.89</A>
<B>7.98</B>
<AREA>31.04<AREA>
<CIRCUMFERENCE>23.74
</CIRCUMFERENCE>
</TUP0>

(Here TUP0 is a short hand for (A, B, AREA, CIRCUMFERENCE) or shorter:

<< A B AREA CIRCUMFERENCE:
3.89 7.98 31.04 23.74>>

K. Benecke is with the Otto-von-Guericke-University Magdeburg, Germany, IWS/FIN, Postfach 4120,
39016 Magdeburg e-mail: benecke@ iws.cs.uni-magdeburg.de).

The first assignment results into a table containing only an „A-column“. This table is extended by each of the following assignments by an additional column.
For comparison purposes we formulate this query in XQuery (compare [6] and [5]), too:

```
let xs:float $A := 3.89, xs:float $B := 7.98
return <TUP0>   <A>{$A}</A>
                <B>{$B}</B>
                <AREA>{$A * $B} </AREA>
        <CIRCUMFERENCE>{2*($A+$B)}
         </CIRCUMFERENCE>
    </TUP0>
```

Query 2 Compute the area and circumferences of several circles.

```
from<<R*: 1.37  2.49  4.86>>
where  PI := 3.1415
       AREA := R*R*PI
       CIRCUMFERENCE := 2 * R *PI

Result:
      <TUP0><PI>3.1415</PI>
            <TUP1*>
              <TUP1> <R>1.37</R>
                     <AREA>5.89</AREA>
                     <CIRCUMFERENCE>8.60
                      </CIRCUMFERENCE>
              </TUP1>
              <TUP1> <R>2.49</R>
                     <AREA>27.15</AREA>
                     <CIRCUMFERENCE>15.64
                      </CIRCUMFERENCE>
              </TUP1>
              <TUP1> <R>4.86</R>
                     <AREA>74.20</AREA>
                     <CIRCUMFERENCE>30.53
                      </CIRCUMFERENCE>
              </TUP1>
            </TUP1*>
       </TUP0>
```

Or shorter:

```
<<  PI          (  R      AREA  CIRCUMFERENCE)*:
    3.1415         1.37   5.89    8.60
                   2.49   27.15   15.64
                   4.86   74.20   30.53>>
```

By the from-part an unnamed table of type R* (a list of R-values) is created. This table is extended stepwise to (PI, R*), (PI, (R, AREA)*), and finally to a table of type (PI, (R, AREA, CIRCUMFERENCE)*). If we replace the second assignment by „PI := 3.1415 AT R“, then a „first normal form relation“ of type (R, PI, AREA, CIRCUMFERENCE)* results.
In [10] also collection assignments are implemented such that the from-part could be replaced by *L(R) := (1.37, 2.49, 4.86)* (Here, *L* abbreviates *list*) . The corresponding table has the same type as the table of the from-part. Due to richer base structures, such assignments are not needed now. We could replace this by *RS := <<L(R): 1.37 2.49 4.86>>* . Here a table of type *RS* with dtd(RS) = L(R) results. By the above following inner extensions a table of type (RS, PI) with dtd(RS) = L(R, AREA, CIRCUMFERENCE) results.
Query 2 in XQuery:

```
let xs:float $PI := 3.1415
      return <TUP0><PI>$PI</PI>
                for xs:float $R IN (1.37, 2.49, 4.86)
       return <TUP1><R>{$R}</R
                <AREA>{ $PI * $R * $R }</AREA>
        <CIRCUMFERENCE>{ 2* $PI * $R }
                </CIRCUMFERENCE>
    </TUP1>
  </TUP0>
```

Query 3: Compute the Fibonacci numbers until 40.

```
from<<X*: 1 TO 40>>
where  FIB[1; 2; N+1] := [1; 1; FIB[N-1] + FIB[N]]

    Result:   <<  (  X        FIB)*:
                     1        1
                     2        1
                     3        2
                     4        3
                     5        5
                     6        8
                     ...
                     40       102334155>>
```

It is possible to add a selecting condition „X= 40“, if only the final number is desired.

Query 4: Euclidian algorithm for the computation of the greatest common divisors from 786524 and 564.

```
where
   AA := 786524
   BB := 564
   L(A, B, REMAINDER)[1; N+1 TO REMAINDER = 0]
    :=[AA, BB, AA mod BB;
     B[N], REMAINDER[N], B[N] mod REMAINDER[N]]
```

The last two queries demonstrate how a tabular view of computations is maintained. Query 3 will be processed in general with help of a recursive function and query 4 by a while-loop. We believe that both features should be avoided in end user languages. In both examples the computations start with 1 and go forward. This seems to be very natural compared with the backward computations of recursive functions. Further, the complexity of this recursive function is exponential. Further query 4 does not require the user understanding that variables are overwriten in loops.
We have a CAML-implementation of both kinds of assignments. These implementations work up to now only for non-first-normal-first-relations with optional values and also not for arbitrary XML-documents. We shall see that the specifications of assignments, which are based on one term only, are already very complicated. Therefore we will only consider these “simple“ assignments in the following.

Think of a document **X1** with dtd(X1) = (A?, B)* and an assignment C := A + B, then the resulting document should be of type ((A, C)?, B)*. If an A- and B-value exists, then also a C-value exists, which can be written at this position.

But, if we consider a document **X2** with dtd(X2) = (A?, B?)* and the same assignment, then neither ((A, C)?, B?)* nor (A?, (B, C)?)* are appropriate types of the result. We cannot compose a result for the first scheme, if an A-value exists and a B-value is missing. On the other hand, we cannot construct a result for the second type, if a B-value exists and an A-value does not. Therefore, the resulting scheme should be (A?, B?, C?)*. Here a C-value can be written, if and only if an A- and a B-value exists. In this case the given type should be extended by C? and not by C. From formal point of view we will realize the former extension by an operation *inn-ext1?* and the latter by *inn-ext1*. The operation *inn-ext1?* is also needed, if the given table contains no question mark, namely if the given term contains partial operations like division of integers or floats.

What is the result, if we consider C := A + B for a document **X3** of type "(D, A*, B*)*" ?

How can C-values be computed, if the result would be of type "(D, (A, C)*, B*)*"?

Because we have no corresponding B-value for each A-value (or better a list of corresponding B-values), we cannot compute C-values. The same holds for the result type (D, A*, (B, C)*)*. Can we compute C-values, if the result type is (D, A*, B*, C*)* ? This makes sense only, if a C-value is computed for each combination A- and B-value. Thus, in general a great output C*-collection (Cartesian product) results, and further it is not visible to which input data a C-value belongs to. Then we can say C := A+B is not applicable to X3 or in other words the application of C := A+B on X3 results in X3. If the user still wants to apply this assignment, he has to transform X3 at first to a document of type (D, (A, B)*)*, for example. But this problem is not considered in this paper.

Now, let **X4** be given with dtd(X4) = (A, B)*, dtd(A) = (A1, A2), and dtd(B) = (B1, B2). Consider the assignment C:= A1 + B1. Here, we could extend the DTD of A or the DTD of B, but we believe that our specification is simpler, if we extend the DTD of X4 to

(A, B, C)*. If the user wishes to extend the DTD of A, then he has to write

C := A1 + B1 AT A1. The A1 will be an additional (second) argument of our corresponding inn-ext1-operation.

The next example corresponds to the assignment A := (B = D) AT D. If „AT D" is missing, then the system has to generate this additional argument:

	F: S			
				S
B	C	D		E
2	1	2		3
				4
	5	6		

(t1)

) =

	F:	S		
				S
B	C	D	A	E
2	1	2	true	3
				4
	5	6	false	

(t2)

inn-ext1(A,{D}, B=D, t1) = t2

Now, we turn to recursive documents:

dtd(**X5**) = PERSON*

dtd(PERSON) = (NAME, LOCATION, SALARY, MGR?, CHILD*)

dtd(MGR) = dtd(CHILD) = PERSON

We are interested in the simple assignment

NET := SALARY*0.66

inn-ext1(NET, {SALARY}, SALARY*0.66, X5) has the DTD PERSON, with

dtd(PERSON) = (NAME, LOCATION, SALARY, NET, MGR?, CHILD*), and

dtd(MGR) = dtd(CHILD) = PERSON.

If we consider the specification of *inn-ext1* in [1], then it becomes visible that for each sub table, containing a component, which has a name as head and which is contained in the given term, then this name of the term is occupied by the component of the table. In our case this means, that for each outermost person (elements of X5) the SALARY-name is occupied. Now, the term SALARY*0.66 can be completely evaluated and the result appears in the new NET-column. Since in our case SALARY appears repeatedly also in each MGR- and CHILD-component, we have to extend these components, too.

They are extended by the same superordinated NET-value. This approach seems to be inadequate for this application. If we want that always the salary of the corresponding person is taken for the inner extension, then we have to replace *SALARY* by //SALARY.

If we consider

Query 5: Extend the salary of each person by its net salary.

```
from   X5.XML
where  NET := //SALARY*0.66,
```

then inn-ext1(NET, {SALARY}, //SALARY*0.66, X5) has the same DTD as the above extension, but each person is extended by its own SALARY-value.

Query 5 in XQuery:

```
define new_person (element $p)
   returns element
{  <person>
      { $p/NAME }
      { $p/LOCATION }
      { $p/SALARY }
      <net> { $p/SALARY*0.66 }</net>
      for $m IN $p/mgr/person
      return <mgr> {new_person($m)}</mgr>
      for $c in $p/child/person
      return <child> {new_person($c)}</child>
   </person>
}

      <result>
         for $p in document("x5.xml")/person
         return new_person($p)
      </result>
```

From this example we can learn also that the following specification of *inn-ext1* has to differ deeply from the specification of [1]. In the above example is visible that it is not enough to extend a tuple only at one position. If we apply *inn-ext1* to a PERSON-tuple, we have to extend the PERSON-level and further we have to apply *inn-ext1* to the MGR?- and the CHILD?-components, too. To specify this, we say that we extend each component of a tuple and because we have the additional *name*-argument (AT *name*), we can describe that components without this name remain unchanged.

Because of these changes in specification now assignments of type

A := INT*0.9, will be possible, too. By such an assignment each INT-value of a table will be extended.

By the next simple example is demonstrated that recursive names and unrecursive names may occur in one term.

dtd(**X6**) = (A, B*)

dtd(B) = (C, B*)

If we consider the assignment D := A + //C, then the following DTD results:

dtd(inn-ext1(D, {C}, A+//C)) = (A, B*) with

dtd(B) = (C, D, B*)

For each C-value a superordinated A-value exists such that the term can be evaluated for each C-value. In the specification we have to use two terms (expressions) to model this. In a term *e* only the unrecursive names are occupied such that this term contains all superordinated values of the table *t*. In a term *e'* all names are occupied. If *e'* contains no free variables, then the corresponding table is extended by the new e'-value and for further extensions *e'* is replaced by *e*. This replacement is always done in the above example, if we go into a table of type *tag0(B, t')*.

The specifications of *inn-ext1* and *inn-ext1?* are very similar. Therefore, we consider only the latter. For the former we have to replace *coll(s1, inj(n))* by *inj(n)*, and *add(empty(coll(s1, inj(n))), x)* by *x*, and *empty(coll(s1, inj(n)))* by *empty-t*.

It remains the question, how the system can decide, in which cases it has to apply *inn-ext1* and in which cases *inn-ext1?* ? This is not a very difficult question, because in any case, where we apply *inn-ext1*, we also could apply *inn-ext1?*. The only difference is that the resulting tables have an unnatural structure in the second case. For example, in query 1 the following extensions result: empty-s to A, A to (A, B), (A, B) to (A, B, AREA) and this scheme to (A, B, AREA, CIRCUMFERENCE). If we apply *inn-ext1?* the following equivalent extensions result:

empty-s to A?, A? to (A, B?)?, (A, B?)? to (A, (B, C?)?)?, and this to (A, (B, (C, D?)?)?)?.

The resulting structure would look a little more pleasingly, if we do not extend *AT A* or *AT B*,..., but if we would allow to extend *AT A?*. Then a table of type (A?, B?, C?, D?) would be yielded. We will not follow this discussion in the paper, because the user can always generate a natural structure by using the *stroke*-operation. With *stroke* both schemes can be transformed into a table of type (A, B, C, D). Further, the specification would become unnecessary complicated. And third, we have *inn-ext1*.

It remains to decide in which cases the system can use *inn-ext1* instead of *inn-ext1?*. The latter is applicable, if the given term does not contain a partial operation and if the given term does not contain two names A and B such that A? and B? occurs in the DTD of the given table.

III. Algebraic Specification of XML-Documents

We do not present a detailed specification of XML-documents (sort Table) here, but we introduce all needed sorts and generating operations. For details compare [4]. We use the algebraic specification language of [9]. The semantic is described by the initial algebra. This means that specified objects can be represented by terms in generating operations and two terms are equal, if and only if it can be deduced from the given axioms (implications, where the right and left hand side are equation systems). Operations may have a defining equation system.

sorts **Bool, Nat** // Boolean values and natural numbers
opers **true, false** ⟶ Bool
zero, one ⟶ Nat
succ (Nat) ⟶ Nat // successor of a natural number
(Nat **+, *** Nat) ⟶ Nat // addition and multiplication
(Nat **<, >**,...Nat) ⟶ Bool // smaller-relation, ...
and, or (Bool, Bool) ⟶ Bool
sorts **Coll-sym** // collection symbols
opers **set, bag, list, s1** ⟶ Coll-sym
sorts **Letter, Digit, Separator, Connector** //
Value // elementary values =Letters + Digits + Separators + Connectors + Booleans
opers **let** (Letter) ⟶ Value // each letter is a value, …
dig (Digit) ⟶ Value
sep (Separator) ⟶ Value
bo (Bool) ⟶ Value
co (Connector) ⟶ Value
sorts **Name** // simple names for tags
sorts **Scheme**
opers **empty-s** ⟶ Scheme // empty scheme
inj (Name) ⟶ Scheme // each name is a scheme
pair-s (Scheme, Scheme) ⟶ Scheme
//2-tuple of schemes
coll (Coll-sym, Scheme) ⟶ Scheme
alternate-s (Scheme, Scheme) ⟶ Scheme
axioms s, s', s'': Scheme
pair-s(s, empty-s) = pair-s(empty-s, s) = s
pair-s(pair-s(s, s'), s'') = pair-s(s, pair-s(s', s''))
alternate-s(alternate-s(s,s'),s'')
=alternate-s(s,alternate-s(s',s''))
end
def
opers **comp-no** (Scheme) ⟶ Nat
// the number of components of a scheme
equal-s (Scheme, Scheme) ⟶ Bool
// unspecified; simple equality relation
comp? (s: Scheme, s': Scheme) ⟶ Bool

// each component of s occurs in s'
coll? (s: Scheme) ⟶ Bool
// s is a scheme for a collection
red (s:Scheme iff coll?(s) = true) ⟶ Scheme
// a collection scheme is reduced by the topmost collection
// symbol
coll-type (s:Scheme iff coll?(s)) ⟶ Coll-sym
// the collection type of a collection

sorts **Table**

opers **empty-t** ⟶ Table
el-tab (Value) ⟶ Table
// an elementary table (contains one value)
empty (s: Scheme iff coll?(s)) ⟶ Table
add (t1: Table, t2: Table iff red(head(t1)) = head(t2))
⟶ Table
pair (Table, Table) ⟶ Table
alternate1 (t: Table, s: Scheme) ⟶ Table
alternate2 (s: Scheme, t: Table) ⟶ Table
tag0 (n: Name, t: Table iff dtd(n) = head(t) or
dtd(n) = inj(any)) ⟶ Table
head (Table) ⟶ Scheme

axioms n: Name; s, s', s": Scheme; t, t', t1, t2, t3: Table; l: Letter, d: Digit, se: Separator, b: Bool
head(empty-t) = empty-s
head(el-tab(let(l)) = inj(letter), head(el-tab(dig(d))) = inj(digit)
head(el-tab(sep(se))) = inj(separator), head(el-tab(bo(b))) = inj(bool)
if coll?(s) then head(empty(s)) = s
if t = add(t1, t2) then head(t) = head(t1)
head(pair(t1, t2)) = pair-s(head(t1), head(t2))
head(alternate1(t, s)) = alternate-s(head(t), s)
head(alternate2(s, t) = alternate-s(s, head(t))
if t = tag0(n, t') then head(t) = inj(n)
pair(empty-t, t) = pair(t, empty-t) = t
pair(t1, pair(t2, t3)) = pair(pair(t1, t2), t3)
alternate1(alternate2(s', t), s") = alternate2(s', alternate1(t, s"))
alternate1(alternate1(t, s'), s") = alternate1(t, alternate-s(s',s"))
alternate2(s', alternate2(s", t)) = alternate2(alternate-s(s',s"), t)
if coll-type(head(t1)) = set & red(head(t1))=head(t2)=head(t3)
then add(add(t1, t2), t3) = add(add(t1, t3), t2)
if coll-type(head(t1)) = bag & red(head(t1)) = head(t2)
= head(t3) then add(add(t1, t2), t3) = add(add(t1, t3), t2)
if head(t1) = coll(set, head(t2))
then add(add(t1, t2), t2) = add(t1, t2)
if coll-type(head(t1)) = s1 & head(t2) = head(t3)
= red(head(t1)) & t1 = empty(s)
then add(add(t1, t2), t3) = add(t1, t2)

end

sorts **Names** // set of name objects

opers **empty-n** ⟶ Names // the empty set of names
{ Name **}** ⟶ Names // a singleton of names
union-n (Names, Names) ⟶ Names
// set theoretic union
in-n (Name, Names) ⟶ Bool // element relation
inclu-n (Names, Names) ⟶ Bool // inclusion relation
intersect-n (Names, Names) ⟶ Names // intersection
minus-n (Names, Names) ⟶ Names // set difference

…

IV. SPECIFICATION OF *INNEXT1?*

opers **comp2?** (n: Name, t: Table) ⟶ Bool
(*n* is a comp2?-component of *t*, that means there exists a component of *t* or an *n*-table within *s1*-collections or alternate expressions)
deep-names-s (s: Scheme) ⟶ Names
(all names of *s* and *dtd(n)* for *n* from *dtd(n)*, starting with *s*; a name *n* is recursive, if *n* occurs in deep-names-s(dtd(n)))

axioms n, n': Name; ns: Names; v: Value; s: Scheme;
t, t': Table
comp2?(n, el-tab(v)) = (equal-s(inj(n), head(el-tab(v)))
comp2?(n, tag0(n', t)) = (equal-n(n, n') or comp2?(n, t'))
comp2?(n, pair(t, t')) = (comp2?(n, t) or comp2?(n, t'))
comp2?(n, empty-t) = false
comp2?(n, alternate1(t, s)) = comp2?(n, t)
comp2?(n, alternate2(s, t)) = comp2?(n, t)
if truecoll?(head(t)) then comp2?(n, t) = false
comp2?(n, empty(coll(s1, s))) = false
if t = add(empty(coll(s1, s)), t')
then comp2?(n, t) = comp2?(n, t')

deep-names-s(empty-s) = empty-n
deep-names-s(inj(letter)) = {letter},...
(for all names without DTD)
if dtd(n) = s then deep-names-s(inj(n))
= union-n({n}, deep-names-s(dtd(n))
deep-names-s(pair-s(s, s'))
= union-n(deep-names-s(s), deep-names-s(s'))
deep-names-s(coll(c, s)) = deep-names-s(s)
deep-names-s(alternate-s(s, s'))
= union-n(deep-names-s(s), deep-names-s(s'))

def

sorts **Expr** (formal expression (term), built from single data and mass data ground operations and relations; these expressions will be used as right hand sides of assignments)

opers **n-ex** (Name) ⟶ Expr
// each name is an expression
rec-n-ex (Name) ⟶ Expr
// each recursive name (//n) is an expression
v-ex (Table) ⟶ Expr
// each table is an expression (value) (constant)
+-ex ; <-ex (e1: Expr, e2: Expr) ⟶ Expr
=-ex (e1: Expr, e2: Expr ⟶ Expr
and-ex ; or-ex (e1: Expr, e2: Expr) ⟶ Expr
non-ex (e: Expr) ⟶ Expr
if-then-else-ex (e1: Expr, e2: Expr, e3: Expr) ⟶ Expr
intersect-ex (e1: Expr, e2: Expr) ⟶ Expr
(Intersection; because of this operation, it is not enough to introduce only simple values (v-ex) as constants)
inclu-ex (e1: Expr, e2: Expr) ⟶ Expr

end

def
opers **+-t** (t: Table, t': Table) ⟶ Table
intersect (t: Table, t': Table iff coll?(head(t))
& head(t) = head(t')) ⟶ Table
// Ordinary intersection of tables, which are collections of
// the same type
intersect-t (t: Table, t': Table) ⟶ Table
//Here we consider only two special operations, which
//build the base for the inn-ext1-
//operations; by *inn-ext1* and *inn-ext1?* these
//operations are applied for all corresponding inner sub
//tables; by these operations the outmost tags are
//omitted and the operations are applied two the
second //argument of *tag*.
axioms n, n': Name, v, v': Table
+-t(t, t') = sum(pair(t, t'))
//the specification of the function *sum* is similar to
//the specification of function *all* from [3]; computing
//*sum(t)* it results a table <*int*>*i*</*int*>, if within *t* no
//float-number exists, it results <*float*>*f*</*float*>, if *t*
//contains a float. All numbers, occurring in *t* are
//added.

if t = empty(s) & s = head(t')
then intersect(t, t') = empty(coll(set, red(s)))
if t = add(t1, t2) & in(t2, t')
then intersect(t, t') = add(intersect(t1, t'), t2)
if t = add(t1, t2) & in(t2, t') = false
then intersect(t, t') = intersect(t1, t')
(The specification of the element relation *in* is simple
and contained in [3].)

if t = tag0(n, t1) & t' = tag0(n', t1')
& t2 = stroke(head(t1), t1') & coll?(head(t1))
then intersect-t(t, t') = intersect(t1, t2)
end
opers **occupy-ex** (e: Expr, t: Table) ⟶ Expr
//All names from *e* , which occur as components in
//the sense of *comp2?* in *t* are occupied by
//corresponding sub tables; if a name *n* occurs twice
//in topmost level of *t*, then the leftmost *n*-sub table
//is taken.
unrec-occupy-ex (e: Expr, t: Table) ⟶ Expr
//Like occupy-ex, but only the unrecursive names are
//occupied.
names-ex (Expr) ⟶ Names
//The set of all names, which occur in the given expression.
rec-names-ex (Expr) ⟶ Names
unrec-names-ex (Expr) ⟶ Names
val-ex (e: Expr iff names-ex(e) = empty-n) ⟶ Table
(The value of an expression without free names)
axioms n : Name ; t, t1, t2, t3, v: Table; e, e1, e2: Expr
if comp2?(n, t2)=true & extract-comp2(t, {n})=triple(t1, t2, t3)
& comp2?(n, t1) = false & comp -no(t2) = 1
then occupy-ex(n-ex(n), t) = v-ex(t2)
if comp2?(n, t) = false
then occupy-ex(n-ex(n), t) = n-ex(n)
if comp2?(n, t) = true
then occupy-ex(rec-n-ex(n), t) = v-ex(extract-comp2(t, {n})
if comp2?(n, t) = false
then occupy-ex(rec-n-ex(n), t) = n-ex(n)
occupy-ex(v-ex(v), t) = v-ex(v)
occupy-ex(+-ex(e1, e2), t)
= +-ex(occupy-ex(e1, t), occupy-ex(e2, t))
(analogous equations for <-ex, and the remaining
operations)

if comp2?(n, t) = true
then unrec-occupy-ex(n-ex(n), t) = v-ex(extract-comp2(t, {n})
if comp2?(n, t) = false
then unrec-occupy-ex(n-ex(n), t) = n-ex(n)
unrec-occupy-ex(rec-n-ex(n), t) = n-ex(n)
unrec-occupy-ex(v-ex(v), t) = v-ex(v)
unrec-occupy-ex(+-ex(e1, e2), t)
= +-ex(occupy-ex(e1, t), occupy-ex(e2, t))
(analogous equations for <-ex, and the remaining
operations)

names-ex(n-ex(n)) = {n}
names-ex(rec-n-ex(n)) = {n}
names-ex(v-ex(t)) = empty-n
names-ex(+-ex(e1, e2))
= union-n(names-ex(e1), names-ex(e2))
(analogous equations for the remaining operations)

rec-names-ex(n-ex(n)) = empty-n
names-ex(rec-n-ex(n)) = {n}
names-ex(v-ex(t)) = empty-n
names-ex(+-ex(e1, e2))
= union-n(names-ex(e1), names-ex(e2))
(analogous equations for the remaining operations)

val-ex(v-ex(v)) = v
if names-ex(e1) = names-ex(e2) = empty-n
then val-ex(+-ex(e1, e2)) = +-t(val-ex(e1), val-ex(e2)) &
& val-ex(intersect-ex(e1, e2))
= intersect-t(val-ex(e1), val-ex(e2))
& val-ex(=-ex(e1, e2)) = equal-t(val-ex(e1), val-ex(e2))
…
end
def
opers
inn-ext1?-s (n: Name, n': Names, s: Scheme iff card(n') <= 1)
⟶ Scheme
//The given scheme *s* is extended by a new name *n*
//direct behind the name *n'*; the scheme is extended
//at all positions, where *n'* occurs.
a-inn-ext1?(n: Name, n': Names, e: Expr, e': Expr, t: Table
iff card(n') <= 1) ⟶ Table
//Auxiliary operation; in *e* each unrecursive name is
//occupied exactly once; in *e'* each name is
//occupied; after each inner extension e' is replaced
//by the topical *e*.
inn-ext1? (n: Name, n': Names, e: Expr, t: Table

```
        iff card(n') <=1 ) ⟶ Table
          //Right beside each name n' the given table t is
          //extended by one new column named n?, the
          //corresponding values are computed by e from
          //superordinated values, formally we consider n' as a
          //one elementary set of names, to allow also an empty
          //set of names; if an assignment N := e AT POS is
          //given, then n' ={POS}. If the AT-part is missing,
          //and the expression contains only one name p, then
          //we choose n' = {p}. If the expression contains no
          //name, then n' is the empty set, otherwise for n' the
          //deepest and from all deepest names the rightmost
          //name is taken.

axioms n, n', n": Name; ns: Names; e, e': Expr;
   s, s', s": Scheme; t, t1, t2, t': Table
inn-ext1?-s(n, {n'}, inj(n')) = pair-s(inj(n'), coll(s1, inj(n)))
if n' != n" then inn-ext1?-s(n, {n'}, inj(n")) = inj(n")
inn-ext1?-s(n, {n'}, pair-s(s, s'))
   = pair-s(inn-ext1?-s(n, {n'}, s), inn-ext1?-s(n, {n'}, s'))
inn-ext1?-s(n, {n'}, coll(c, s')))
   = coll(c, inn-ext1?-s(n, {n'}, s'))
inn-ext1?-s(n, {n'}, alternate-s(s, s'))
   = alternate-s(inn-ext1?-s(n, {n'}, s), inn-ext1?-s(n, {n'}, s'))
inn-ext1?-s(n, {n'}, empty-s) = empty-s
inn-ext1?-s(n, empty-n, s) = pair-s(s, coll(s1, inj(n)))

if head(t) = inj(n') & names-ex(occupy-ex(e', t)) = empty-n
   then a-inn-ext1?(n, {n'}, e, e', t) =
   = pair(t, add(empty(coll(s1, inj(n)))
            , tag0(n, val-ex(occupy-ex(e, t)))))
if head(t) = inj(n') & names-ex(occupy-ex(e', t)) != empty-n &
   then a-inn-ext1?(n, {n'}, e, e', t)
   = pair(t, empty(coll(s1, inj(n))))
if head(t) = inj(n") & n" != n'
   & in-n(n', deep-names-s(inj(n"))) = false
   then a-inn-ext1?(n, {n'}, e, e', t) = t
if t = tag0(n", t') & n" != n' & in-n(n', deep-names-s(inj(n")))
   & rec-names-ex(e') = empty-n
   then a-inn-ext1?(n, {n'}, e, e', t)
           = tag0(n", a-inn-ext1?(n, {n'}, e, e, t'))
if t = tag0(n", t') & n" != n' & in-n(n', deep-names-s(inj(n")))
   & rec-names-ex(e') != empty-n
   then a-inn-ext1?(n, {n'}, e, e', t)
           = tag0(n", a-inn-ext1?(n, {n'}, e, e', t'))
if t = empty(s) then a-inn-ext1?(n, {n'}, e, e', t)
                        = empty(inn-ext1?-s(n, {n'}, s))
if t = add(t1, t2)
   then a-inn-ext1?(n, {n'}, e, e', t) =
   = add(a-inn-ext1?(n, {n'}, e, e', t1)
           , a-inn-ext1?(n, {n'}, e, e', t2))
if t = alternate1(t', s)
   then a-inn-ext1?(n, {n'}, e, e', t) =
   = alternate1(a-inn-ext1?(n, {n'}, e, e', t')
       , inn-ext1?-s(n, {n'}, s))
if t = alternate2(s, t')
   then a-inn-ext1?(n, {n'}, e, e', t) =
            = alternate2(inn-ext1?-s(n, {n'}, s)
                        , a-inn-ext1?(n, {n'}, e, e', t'))
if t = pair(t1, t2)
   then a-inn-ext1?(n, {n'}, e, e', t) =
            = pair(a-inn-ext1?(n, {n'}, unrec-occupy-ex(e, t),
            occupy-ex(e', t), t1),
            a-inn-ext1?(n, {n'},
            unrec-occupy-ex(e, t), occupy-ex(e', t), t2))
if names-ex(e') = empty-n
   then a-inn-ext1?(n, {n'}, e, e', empty-t) = empty-t
if names-ex(e') != empty-n
   then a-inn-ext1?(n, {n'}, e, e', empty-t) =
           = add(empty(coll(s1, inj(n))), tag0(n, val-ex(e)))
if names-ex(occupy(e', t)) != empty-n
   then a-inn-ext1?(n, empty-n, e, e', t) =
   = pair(t, add(empty(coll(s1, inj(n))), tag0(n, val-ex(e))))
if names-ex(occupy(e', t)) = empty-n
   then a-inn-ext1?(n, empty-n, e, e', t) =
           = pair(t, empty(coll(s1, inj(n))))

if card(ns) <= 1
   then inn-ext1?(n, ns, e, t) = a-inn-ext1?(n, ns, e, e, t)
end
```

Finally, we remark that also the basic operations of our data model may appear in the right side of an assignment. Thus *A:=stroke(S(B, C), CC)* can be considered for example as an operation, which maps each inner table *CC* of a given document to another table (of type *S(B,C)*). In places like this, we can consider the target scheme as a constant.

REFERENCES

[1] K. Benecke, „Structured Tables – A new Paradigm for Databases and Programming Languages“, (German), Deutscher Universitätsverlag, Wiesbaden 1998,

[2] …, „Stroke-A Powerful Operation for XML-like Data”, Proc. SCI2002, Orlando July 2002, pp. 273-278

[3] … „Formal Specification of stroke for XML-Documents”, http://fuzzy.cs.uni-magdeburg.de/benecke.html

[4] …, “Understanding the Structure of XML-Documents”, submitted for publication

[5] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, J. Robie, “XML Query Use Cases”,W3C Working Draft 16 Aug 2002, http://www.w3.org/TR/xmlquery-use-cases

[6] D. Chamberlain, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, „XQuery: A Query Language for XML“, W3C Working Draft 16 August 2002, http://www.w3.org/TR/xquery

[7] X. Leroy, “The Caml Light system release 0.7-, Documentation and user’s manual”, http//: pauillac.inria.fr/caml, INRIA 1995

[8] X. Leroy et. al., The Objective Caml system release 3.06-Documentation and user’s manual”, http://caml.inria.fr/distrib/ocaml-3.06/ocaml-3.06-refman.pdf, August 2002

[9] H. Reichel, „Initial Computability, Algebraic Specifications, and Partial Algebras“, Akademie Verlag Berlin (Oxford-Press) 1987

[10] D. Schamschurko, „Implementation of the Trunk of the FROM-WHERE-STROKE-construct in CAML-Light“, (in German), Praktikumsbeleg, DeTeCSM Magdeburg, 1996

[11] K. Williams, et. al., “Professional XML Databases”, Wrox Press Ltd., Birmingham, 2000