Technical Report

Nr.: FIN-009-2009

A Restructuring Operation for XML Documents

Klaus Benecke, Xuefeng Li

Arbeitsgruppe Theoretische Informatik

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-009-2009

A Restructuring Operation for XML Documents

Klaus Benecke, Xuefeng Li

Arbeitsgruppe Theoretische Informatik

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Klaus Benecke, Xuefeng Li
Postfach 4120
39016 Magdeburg
E-Mail: benecke@iws.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: 95

Redaktionsschluss: 19.05.2009

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

A Restructuring Operation for XML Documents

Klaus Benecke, Xuefeng Li

IWS/FIN Otto-von-Guericke University Magdeburg
Postfach 4120 39016 Magdeburg, Germany
benecke@iws.cs.uni-magdeburg.de

***Abstract*— This paper describes a new restructuring operation. If an arbitrary XML-file with DTD and a (target) DTD is given then by the operation *stroke* the document is transformed to a XML file, which corresponds to the target DTD. Simultaneously with this restructuring target data can be sorted and arbitrary aggregations can be realized. Therefore, we believe that our *stroke* operation is especially the first sorting algorithm for structured data. The definition of *stroke* profits by a new understanding of XML, where we distinguish not only at DTD- but also at document-level between tuples and collections of (sub) documents. Because in our understanding database tables are also XML-structures we think that *stroke* is not only useful for queries on single or composed XML-documents but also for queries on databases and also for queries on search engines. Here, it allows especially to extract tuples of nodes (fragments) of the given XML-structure. The paper presents use cases for restructuring of XML documents and a description of the procedural definition and implementation of *stroke*. A corresponding functional definition (implementation) of *stroke* in OCAML is the kernel of the paper. Finally a short experimental evaluation of the both implementations with corresponding galax programs is given.**

I. Introduction

The restructuring operation *stroke* is one of the best operations of our query language OttoQL (OsTfälisch Table Oriented)([1]). A NF^2-version of *stroke* is published in [2]. It can be compared with the operation *restruct* of [3], but it is generalized to XML and it allows additionally to sort, and aggregate data. XQuery (see [4] or [5]) has no restructuring operation. Therefore we can formulate some queries more userfriendly than XQuery. Consider at first Query XMP: 1.1.9.2 Q2 from [5]: Create a flat list of all the title-author pairs, with each pair enclosed in a ”result” element.

XQuery:

```
<results>
{
for $b in doc("XMP/bib.xml")/bib/book,
    $t in $b/title,
    $a in $b/author
return
    <result>
        { $t }
        { $a }
    </result>
}
</results>
```

OttoQL:

```
aus doc("bib.xml")
gib results results=L(result) &&
    result=title,author
```

Here, ”L” abbreviates list and ”&&” connects two lines to a logical unit. The gib-part is realized by the *stroke* operation. Because we represent our documents often by tables we could simplify the query in the following way:

```
aus doc("bib.xml")
gib L(title,author)
```

Now we consider query XMP: 1.1.9.4 Q4: For each author in the bibliography, list the author’s name and the titles of all books by that author, grouped inside a ”result” element.

XQuery:

```
<results>
{
  let $a := doc("XMP/bib.xml")//author
  for $last in distinct-values($a/last),
      $first in distinct-values
                 ($a[last=$last]/first)
  order by $last, $first
  return
   <result>
    <author>
       <last>{ $last }</last>
       <first>{ $first }</first>
    </author>
    {
     for $b in doc("XMP/bib.xml")//book
     where some $ba in $b/author
        satisfies ($ba/last = $last
        and $ba/first=$first)
     return $b/title
          }
   </result>
}
</results>
```

OttoQL:

```
aus doc("bib.xml")
gib M(author,L(title))
```

Here, M abbreviates set (Menge). A result set of the gib-part does not contain the atomic fields (here author) twice. That means a set contains no duplicates. For each author value the list of titles is collected. We see already on these small examples that OttoQL has a much simpler syntax than XQuery and even XPath. In the core of OttoQL the operations select, ext (extension of a document by a new tag (column), *stroke*,... are applied one after the other. In section II a definition of the type XML-document is presented. This definition is the fundament of *stroke* and the whole data model of OttoQL. Section III presents the most important examples, to illustrate *stroke*. It becomes clear that *stroke* replaces the following operations: projection, distinct, aggregate, sort, union, nest, and unnest. But for recursive structures a further operation (giball) is used. This operation is much simpler than *stroke*, but not considered in this paper. It corresponds to the doubleslash operation of XPath. In the next section a procedural algorithm of *stroke* is described. This algorithm seems to be very simple, but it is not so easy to implement it as we believe. The kernel of the restructuring algorithm is a restructuring table *umstruc4*, by which is described which source levels are inserted into which target levels. In the section V we consider the functional implementation of *stroke*. This implementation is a result of an algebraic specification with initial semantics (see [6]). We have choosen a description in OCAML, because we think that more people are familiar with OCAML than with algebraic specification languages. In section VI we compare our both implementations. As expected the procedural one is more efficient than the functional one. For us it was surprising that both implementations are much more efficient than the restructuring with *GALAX*. In some examples they differ by the the factor 100.

II. A New Understanding of XML

In this section we present our understanding of XML in the syntax of OCAML ([7]). An XML document is also called tabment (TABle+docuMENT).

```
type coll_sym = Set | Bag | List | Set_minus
  | Bag_minus | List_minus | Any | S1 ;;
                (* collection types: S, B, L, S-, B-, L-, A, ? *)
                      (* S-, B-, L- for downwards sorting *)

type name = string;;                       (* column names *)

type scheme =                     (* schemes of documents *)
  Empty_s                               (* empty scheme *)
  | Inj of name                  (* each name is a scheme *)
  | Coll_s of coll_sym * scheme (*schemes for collections*)
  | Tuple_s of scheme list           (* schemes for tuples *)
  | Alternate_s of scheme list;;     (* schemes for choice *)

type value =           (* disjoint union of elementary types *)
  Bar                  (* a dash; only for school of interest *)
  | Int_v of big_int          (* each big integer is a value *)
  | Float_v of float
  | Bool_v of bool
  | String_v of string;;

type tabment =          (* type for tables resp. documents *)
  Empty_t                    (* empty tabment: error value *)
  | El_tab of value    (* an elementary value is a tabment *)
  | Tuple_t of tabment list            (* tuple of tabments *)
  | Coll_t of (coll_sym * scheme) * (tabment list)
                                  (* collection of tabments *)
  | Tag0 of name * tabment
                        (* a tabment is enclosed by a name *)
  | Alternate_t of (scheme list) * tabment;;
   (* the type of the tabment is changed to a choice type *)
```

Examples: The XML document "Hallo" can be represented by the OCAML term

```
El_tab(String_v "Hallo")
```

and the XML document

```
<X><A>a</A><A>b</A></X>
```

can be represented for example by

```
Tag0("X",Tuple_t[
  Tag0("A",El_tab(String_v "a"));
  Tag0("A",El_tab(String_v "b"))])
```

or by

```
Tag0("X",Coll_t((List, Inj "A"),
   [Tag0("A",El_tab(String_v "a"));
    Tag0("A",El_tab(String_v "b"))])).
```

The XML document *students0.xml* of figure 1 can be represented as table (Table 1) and as OCAML term (figure 2). We summarize the differences between the common understanding XML documents and the specified tabments:

1) The specification does not distinguish between XML-attributes and XML-elements; an attribute is signaled by a preceding "@".
2) Unlike to XML, a tabment need not have a root tag.
3) In the tabment, and not only in the scheme specification, a tuple of several elements is distinguished from a collection of these elements. This is an advantage, for the specification and implementation of our powerful tabment operations (restructuring stroke, selection, extension *ext*, *vertical*, ...).
4) The specification handles beside lists (L) additional proper collection types: Set (M), Bag (B), and Any (A). The "collection" type S1 is not a proper collection. It is used for optional values (?).

III. *Stroke* by Examples

The content of this section can be considered as use cases for a complex restructuring operation. Most examples refer to an XML document *students.xml*.

students.xml: M(STID, SURNAME, FIRSTNAME, MIDDLENAME?, FACULTY, LOCATION, ZIP, STREET, L(EXAM), L(HOBBY), YEAR_OF_REG, CURR_VITAE)

TABLE I

XML DOCUMENT STUDENTS0.XML REPRESENTED AS TAB FILE

```
<<L(STID, NAME,    FACULTY,         SEX, L(COURSE,  MARK), L(HOBBY))::
    2     Mueller Computer Science F      DB        2.       reading
                                          EAD       1.3      schwimming
                                          MATHS     2.
    3     Schulz  Mathematics      M      ALGEBRA   1.3
                                          ANALYSIS  1.
                                          NUMERICS  1.7 >>
```

```
Tag0("STUDENTS",
  Coll_t((List,Inj "STUDENT"), [
    Tag0("STUDENT",Tuple_t[
      Tag0("STID",El_tab (Int_v (big_int_of_string "2")));
      Tag0("NAME",El_tab (String_v ("Mueller")));
      Tag0("FACULTY",El_tab (String_v "Computer Science"));
      Tag0("SEX",El_tab (String_v "F"));
      Coll_t((List,Inj "EXAM"),[
        Tag0 ("EXAM",Tuple_t[
          Tag0("COURSE",El_tab(String_v "DB"));
          Tag0("MARK",El_tab (Float_v (2.)))]);
        Tag0("EXAM",Tuple_t[
          Tag0("COURSE",El_tab(String_v "EAD"));
          Tag0("MARK",El_tab(Float_v 1.3))]);
          Tag0("EXAM",Tuple_t[
          Tag0("COURSE",El_tab(String_v "MATHS"));
          Tag0("MARK",El_tab(Float_v 2.))])]);
      Coll_t((List,Inj "HOBBY"),[
        Tag0("HOBBY",El_tab(String_v "reading"));
        Tag0("HOBBY",El_tab(String_v "schwimming"))])]);
    Tag0("STUDENT",Tuple_t[
      Tag0("STID",El_tab (Int_v (big_int_of_string  "3")));
      Tag0("NAME",El_tab (String_v "Schulz"));
      Tag0("FACULTY",El_tab(String_v "Mathematics"));
      Tag0 ("SEX",El_tab(String_v "M"));
      Coll_t((List,Inj "EXAM"),[
         Tag0("EXAM",Tuple_t[
           Tag0("COURSE",El_tab(String_v "ALGEBRA"));
           Tag0("MARK",El_tab (Float_v 1.3))]);
         Tag0("EXAM",Tuple_t[
           Tag0("COURSE",El_tab(String_v "ANALYSIS"));
           Tag0("MARK",El_tab(Float_v 1.))]);
         Tag0("EXAM",Tuple_t[
           Tag0("COURSE",El_tab(String_v "NUMERICS"));
           Tag0("MARK",El_tab(Float_v 1.7))]);]);
      Coll_t((List,Inj "HOBBY"),[])])]))
```

Fig. 2. students0.xml as OCAML term

```
<!DOCTYPE STUDENTS [
<!ELEMENT STUDENTS (STUDENT*)>
<!ELEMENT STUDENT (STID,NAME,FACULTY,
                   SEX,EXAM*,HOBBY*)>
<!ELEMENT STID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT FACULTY (#PCDATA)>
<!ELEMENT SEX (#PCDATA)>
<!ELEMENT EXAM (COURSE,MARK)>
<!ELEMENT COURSE (#PCDATA)>
<!ELEMENT MARK (#PCDATA)>
<!ELEMENT HOBBY (#PCDATA)> ]>
<STUDENTS>
  <STUDENT>
    <STID>2</STID>
    <NAME>Mueller</NAME>
    <FACULTY>Computer Science</FACULTY>
    <SEX>F</SEX>
    <EXAM>
      <COURSE>DB</COURSE>
      <MARK>2.</MARK>
    </EXAM>
    <EXAM>
      <COURSE>EAD</COURSE>
      <MARK>1.3</MARK>
    </EXAM>
    <EXAM>
      <COURSE>MATHS</COURSE>
      <MARK>2.</MARK>
    </EXAM>
    <HOBBY>reading</HOBBY>
    <HOBBY>schwimming</HOBBY>
  </STUDENT>
  <STUDENT>
    <STID>3</STID>
    <NAME>Schulz</NAME>
    <FACULTY>Mathematics</FACULTY>
    <SEX>M</SEX>
    <EXAM>
      <COURSE>ALGEBRA</COURSE>
      <MARK>1.3</MARK>
    </EXAM>
    <EXAM>
      <COURSE>ANALYSIS</COURSE>
      <MARK>1.</MARK>
    </EXAM>
    <EXAM>
      <COURSE>NUMERICS</COURSE>
      <MARK>1.7</MARK>
    </EXAM>
  </STUDENT>
</STUDENTS>
```

Fig. 1. XML file students0.xml

EXAM = (COURSE, MARK, DATE) Therefore, we will omit in these examples the from-part:

```
aus doc("students.xml")
```

Program 1: Projection

```
gib L(SURNAME, FIRSTNAME)
```

The result is not sorted, the order of elements in the given table remains, and duplicates are not eliminated.
Program 2a: Sorting

```
gib B(SURNAME, FIRSTNAME, FACULTY)
```

The result is sorted by (SURNAME, FIRSTNAME, FACULTY) and duplicates are not eliminated. The sorting is outside the theoretical specification.
Program 2b: Sorting with duplicate elimination

```
gib M(FACULTY, S-(MARK, &&
      B(SURNAME, FIRSTNAME)))
```

The outmost collection is sorted by FACULTY, the next inner collection by MARK downwards, and the most inner collection by SURNAME, FIRSTNAME. Each faculty appears only once and within each faculty each mark apperars only once.
Program 3: Distinct

```
gib M(SURNAME, FIRSTNAME)
```

The result is sorted by (SURNAME, FIRSTNAME) and duplicates are eliminated.
Program 4 a: Unnest

```
gib L(SURNAME, FIRSTNAME, COURSE, MARK)
```

The result is a four column flat table, where students without examinations disappear.
Program 4 b:

```
gib B(SURNAME, MIDDLENAME)
```

Students without middle name disappear. The name pairs are sorted. If we want all students in the result, we only have to add a question mark:
Program 4 c:

```
gib B(SURNAME, MIDDLENAME?)
```

Now the system is not forced to build complete pairs. Here, we see, the *stroke* operation follows the main principle of **minimal loss of information**.
Program 5: Nest, respectively Group By

```
gib M(FACULTY, L(SURNAME, FIRSTNAME))
```

Additionally to nest, data are sorted by FACULTY.
Program 6 a: Restruct [3]

```
gib M(COURSE, B(SURNAME, &&
         FIRSTNAME, MARK, DATE))
```

In the given document COURSE is subordinated to the names and in the target document names are subordinated to

COURSE. Each COURSE appears only once, but the inner bag may contain duplicates, contrary to restruct of [3].

Program 6 b:

```
gib M(COURSE, HOBBY)
```

The result of query 6 b is an empty set, because COURSE and HOBBY are not on a hierarchical path in the DTD of students.xml (see figure 3). A restructuring with *restruct* results in general in a non-empty set, because an inner join of M(COURSE, MARK, DATE)- and L(HOBBY)-tables is realized before restructuring. During restructuring with *stroke* (COURSE, MARK, DATE)-tuples are lengthened only by superordinated student data as STID, SURNAME, ... and not by HOBBY. Later HOBBY is lengthened by STID, SURNAME,... . If we want to combine all COURSE and HOBBY tuples of each student we have to realize a corresponding join at first or to *unnest* the data in an additional step. To combine COURSE- and HOBBY-subtuple contradicts to the principle of *stroke* that each (lengthened) subtuple is inserted only once into zero, one or more levels of the target structure. *stroke* follows also the **minimal cost** priciple. But, it is also possible to realize the Cartesian product only with *stroke*.

Program 6 c:

```
gib M(COURSE,L(HOBBY)) ATOM::L(HOBBY)
```

If we would omit the ATOM clause then for each COURSE an empty list of HOBBYs would result, because COURSE and HOBBY are not on a hierarchical path. Since L(HOBBY) is one level higher then HOBBY, COURSE and L(HOBBY) are on one hierarchical path. This is visible in figure 3. Therefore, in the result appears each list of hobbies for each course. By a following additional gib-part the flat Cartesian product can be generated:

Program 6 d:

```
gib M(COURSE,L(HOBBY)) ATOM::L(HOBBY)
gib M(COURSE,HOBBY)
```

Program 7 a: Aggregations simultaneously in different levels of target table. (Compute the total count of marks, the count for each faculty, and for each student.)

```
gib AG,M(FACULTY,AG,B(SURNAME,AG)) &&
    AG := count(MARK)
```

The first AG of the target scheme is the total number, the second the count for each faculty, and the third AG is the count for each student. In the next section we will see that it is not necessary to use the student identifier STID in the target structure in this case. If we use *M(SURNAME, AG)* then the total numbers of marks of all students with the same name are computed. The aggregations are realized during restructuring with *stroke*.

Program 7 b: A simultaneous horizontal and vertical aggregation. Let **students2.xml** the same document as *students.xml*, but with another EXAM-type:

EXAM2: (COURSE, EXERCISE1, EXERCISE2, EXERCISE3)

(Compute the total sum of points, the sum for each faculty, and for each student.)

```
aus doc("students2.xml")
gib AG,M(FACULTY,AG,B(SURNAME,AG)) &&
    AG := sum(EXAM2)
```

Here, it is assumed that COURSE is of type TEXT such that COURSE-data do not contribute to the sums. The exercise tags are assumed to be integers or floats.

Program 7 c: Conditional aggregations (Find for each faculty and each student the number of very good and very bad marks.)

```
gib M(FACULTY,C1,C5,B(NAME,C1,C5)) &&
C1:=count(case MARK &&
          when(MARK=1)endcase) &&
C5:=count(case MARK when(MARK=5)endcase)
```

Program 8 a: Union (Collect for each student all hobbies and courses in one column, where duplicates are omitted.)

```
rename COURSE by HOBBY
gib B(NAME, M(HOBBY))
```

Here, student by student is inserted into target structure and for each student each COURSE now renamed by HOBBY is inserted. After the insertion of COURSEs the original HOBBYs are inserted one after the other. If we would rename the LOCATION for example by HOBBY then all inner collections contain only the LOCATION with new name.

Program 8 b: Counterexample for union

```
rename LOCATION by HOBBY
gib B(NAME, M(HOBBY))
```

Here, the student level is inserted into NAME- and HOBBY-level. Therefore, there is no need to insert the proper HOBBY data into the target. Because the subordinated HOBBY data would be lengthened by STUDENT records the lengthened element would contain two HOBBYs and it would be unclear, which of both we have to take. We recognize again that one of the second main principles of *stroke* is restructuring with **minimal cost**.

Program 9: Joins: We consider three flat relations:

STUDFLAT.xml: M(STID, NAME, SEX)

EXAMFLAT.xml: M(STID, COURSE, MARK)

HOBBYFLAT.xml: M(STID, HOBBY)

Program 9 a: Natural join of two relations

```
aus doc("STUDFLAT.xml"),&&
     doc("EXAMFLAT.xml")
gib M(STID,(NAME,SEX)?, &&
         L(COURSE,MARK))
gib M(STID,NAME,SEX,COURSE, MARK)
```

The comma in the from-part stands for pairs and not for the Cartesian product. Therefore, the resulting scheme of the first line is:

M(STID, NAME, SEX), M(STID, COURSE, MARK)

Because of the last row a flat 1NF-relation results, which is

```
             M
             |
(STID, SURNAME, FIRSTNAME, MIDDLENAME?,..., M,          L, YEAR_OF_REG, CURR_VITAE)
                                            |           |
                                  (COURSE, MARK, DATE) (HOBBY)
```

Fig. 3. The "final" scheme of students.xml as hierarchical graph

more unnatural than the result of second row. We can not omit the sign "?" in the stroke-part of the second row, because the insertion of a EXAMFLAT-tuple would then require a (STID, NAME, SEX) comparison and not only a STID-comparison.

Program 9 b: "Join" of the above 3 tables

```
aus doc("STUDFLAT.xml"), &&
    doc("EXAMFLAT.xml"), &&
    doc("HOBBYFLAT.xml")
gib M(STID, (NAME, SEX)?, &&
    L(COURSE, MARK), L(HOBBY))
```

Program 9 c: The flat natural join of all three tables

```
aus doc("STUDFLAT.tab"), &&
    doc("EXAMFLAT.tab"), &&
    doc("HOBBYFLAT.tab")
gib M(STID,(NAME,SEX)?, &&
      L(COURSE,MARK),L(HOBBY))
gib M(STID, NAME, SEX, L(COURSE, &&
       MARK, L(HOBBY))) ATOM::L(HOBBY)
gib M(STID,NAME,SEX,COURSE, &&
      MARK,HOBBY)
```

Program 10: Restructuring with choice

```
<<L(A,L(C,D)),L(X,Y,M(C,D))::
    1   2 3     1 5   6 7
        7 8           6 9
                1 6   8 8 >>
gib M((A|X),M(C,L(D)))
```

Result as Tab-file:

```
<<M((A| X), M(C, L(D)))::
     1        2    3
              7    8
        1     6    7 9
              8    8>>
```

If the user wishes on toplevel only one value, he can rename A by X.

Program 11: Restructuring with any collection

```
<<L(C,M(D,E)),L(X,Y,M(C,Z))::
    1   2 3     1 5   6 7
        9 9           6 9
                2 6   8 8 >>
gib M(TUP) TUP=C,A()
```

Result as XML file:

```
<root>
  <TUP>
    <C>1</C>
    <C>1</C>
    <D>2</D>
    <E>3</E>
    <D>9</D>
    <E>9</E>
  </TUP>
  <TUP>
    <C>6</C>
    <X>1</X>
    <Y>5</Y>
    <C>6</C>
    <Z>7</Z>
    <X>1</X>
    <Y>5</Y>
    <C>6</C>
    <Z>9</Z>
  </TUP>
  <TUP>
    <C>8</C>
    <X>2</X>
    <Y>6</Y>
    <C>8</C>
    <Z>8</Z>
  </TUP>
</root>
```

The system goes in depth until the C-level is reached. The corresponding lengthened elements are inserted in the C-level and in the ANY collection. Because the two given collections contain elements of different types the elements of the Any-collections have two different types. The structure of the result is better visible, if we copy the query in our online version [1] and compute the result as OCAML term. The whole data set is now sorted and structured by C. This collection type can be used to sort element of different types from different documents or from collections of documents of different type.

Program 12: Optional values are no proper collections given: **students3.xml:** L(NAME, COURSE?, HOBBY?)

```
aus doc("students3.xml")
gib M(NAME, COURSE, HOBBY)
```

Each student with COURSE **and** HOBBY appears in the result, contrary to this query on "students.xml".

Program 13 a: Tags in the target DTD

```
gib M(NAME, MARKS) &&
```

```
NAME=(SURNAME, FIRSTNAME, MIDDLENAME) &&
MARKS=L(MARK)
```

The restructuring, which corresponds to the above query, is similar to a restructuring of the following target scheme: M(SURNAME, FIRSTNAME, MIDDLENAME, L(MARK)). The additional tags NAME for each corresponding triple and MARKS for each corresponding collection are the only difference.

Program 13 b: *stroke* cannot go into tags, if they are atomic

```
gib M(COURSE, L(SURNAME, EXAM))
```

In this target scheme EXAM is considered as atomic. Therefore, *stroke* does not see any COURSE-value such that we get in any case an empty result. We do not consider this as a serious problem because we can use (MARK, DATE) instead of EXAM, which would have less redundancy. In the following query EXAM is not atomic, because it is redefined.

Program 13 c:

```
gib M(COURSE, L(SURNAME, EXAM)) &&
    EXAM = (MARK, DATE)
```

Program 14: Long strings

```
gib M(FACULTY, B(NAME, L(CURR_VITAE)))
```

The target structure would look better, if we omit the inner list symbol, but then the bag is sorted by (NAME, CURR_VITAE). That means comparisons of the CURR_VITAE-values would be necessary. Each list contains only one element, but it does not require these comparisons. The list can also be replaced by ”?” or the ”B” by an ”M”.

Program 15: Recursive target DTDs

```
gib L(E) E = (STID, L(E))
```

Recursive target DTDs are not considered in our implementation, because otherwise an infinite result would be generated, as in the above example.

IV. Procedural Implementation of *Stroke*

Let’s start with a special but not untypical flat source file
zufall1000.xml: L(X, Y, Z)
of random triples of integers between 0 and 9, which contains each triple exactly ones.

Program 16 a:

```
aus doc("zufall1000.xml")
gib M(X,Y,Z)
```

If we assume that the given file contains 1000 elements then the insertion of one triple requires in average 500 complete comparisons. If we choose a higher structured target scheme

Program 16 b:

```
aus doc("zufall1000.xml")
gib M(X, M(Y, Z))
```

then an insertion of one triple requires in average roughly 5 X-comparisons and 50 (Y, Z)-comparisons, because the X chain consists of 10 elements and each (Y, Z) chain of 100 elements. We see that the additional X-level has a similar effect as skip pointers. In the same way the number of comparisons in the child-twin-pointer structure is reduced if we introduce one further level:

Program 16 c:

```
aus doc("zufall1000.xml")
gib M(X, M(Y, M(Z)))
```

Here the insertion of an (X, Y, Z)-element requires in average 5 X-, 5 Y-, and 5 Z-comparisons. This are 5 complete comparisons compared with 500 comparisons in Program 16 a. Again this query is similar to an implementation with two level skip pointers. We see, without measuring the response time of any query that a high structured target table can be generated in general quicker than a flat one. We believe that a user will in general prefer a high structured table, because it is looks more lucid than a flat one. Further, he can add corresponding aggregations in higher levels. Now, we shall see that a structured source table will result in a yet quicker restructuring. Let
zufall10_10_10.xml: L(X, L(Y, L(Z)))
be a random file, which contains 10 distinct X-values, for each X-value 10 distinct Y-values, and for each (X, Y)-value 10 distinct Z-values, where these data are randomly sorted.

Program 17 a:

```
aus doc("zufall10_10_10.xml")
gib M(X, M(Y, Z))
```

Here, an X-value is inserted into the X-chain (5 comparisons), the Y-values cannot be inserted, and each Z-value is lengthened by its superordinated Y and X values, and can therefore be inserted into the (Y, Z) level. The number of comparisons is similar to the number of Program 16 b, but in the X-chain only 10 elements are inserted, contrary to 1000.

Program 17 b:

```
aus doc("zufall10_10_10.xml")
gib M(X, M(Y, M(Z)))
```

Here, the total number of complete comparisons for restructuring is (10*5+100*5+1000*5)/3=1819 compared to 1000*(5+5+5)/3=5000 comparisons of Program 16 c.
The procedural implementation is based on restructuring tables. By **umstruc4** is described, which source (hierarchical) level (qhl) is inserted into which target level (zhl). Table II shows this for the Programs 16 and 17. Because, our current procedural implementation of *stroke* does not allow tags on non elementary tabments and does not allow the choice operator and optional values in the target scheme we restrict ourself to restructuring tables of some of the remaining queries on students.xml. This file has 3 levels the STID-level, the EXAM-level, and the HOBBY-level. The superordinated level of the last two levels is the STID-level. In Table III Program 7 a is described. Here, each STID-level element is inserted in both target levels. By these insertions the AGG-components are occupied by neutral values. For the count aggregation this

TABLE II

UMSTRUC4 FOR PROGRAM 16 AND 17

Query	source level	target level
Program 16 a	1	1
Program 16 b	1	1 2
Program 16 c	1	1 2 3
Program 17 a	1 2 3	1 2 3
Program 17 b	1 2 3	1 2 3

TABLE III

UMSTRUC4 FOR PROGRAMS ON STUDENTS.XML

Query	source level	target level
Program 2 a	1	1
Program 2 b	1 2	1 2 3
Program 5	1	1 2
Program 6 a	2	1 2
Program 6 b		
Program 7 a	1	1 2
Program 8 a	1 2 3	1 2 2
Program 8 b	1	1 2

is zero. After the insertion of a first level element the pointers for FACULTY and SURNAME are fix. For each EXAM-element now a one is added at each of the three levels. That means no further comparisons are necessary for realizing the aggregations.

V. FUNCTIONAL IMPLEMENTATION OF *stroke* IN OCAML

Originally, *stroke* was defined for NF^2-relations in an algebraic specification language of [6], which was based on initial semantics. Because OCAML is more easy to understand, we shall present the essential part of the definition for XML documents in OCAML. The definition of *stroke* is based on the definition of an *insert* operation. To restructure a source tabment *st* to a target DTD *tdtd* means to insert *st* into the empty tabment or a tuple of empty tabments with corresponding neutral values for aggregations of level zero. We handle aggregations in a simplified way. Before restructuring corresponding extensions of the source table are realized such that the source table contains already the names of all aggregations. In Program 7 c for example the following two extensions are realized before *stroke*.

```
ext C1:=case 1 when (MARK=1) endcase
ext C5:=case 1 when (MARK=5) endcase
```

The DTD of the resulting extended source tabment differs only in EXAM with the given one:
EXAM:(COURSE, MARK, C5?, C1?, DATE)
C5 and C1 contain a 1, if the mark is 5 and 1, respectively, and are empty otherwise. Therefore, we need only the name and the type of an aggregation in an parameter of *insert*. These names and the remaining names are contained in a variable *p* of type

```
type t_agg = {
  mutable summe: name list;
  ...
  mutable avg: name list;
  mutable atom: scheme list;
  mutable unimportant: name list}
```

Here, the first entry is the list of sum fields. The atomic components are collected in the last but one row. Here, *Inj n* is included, if *n* is contained in a right side of a target DTD entry and *n* is not a left side of a target DTD entry.. Tags of the left side of the target DTD are collected in the unimportant list. This list is needed only for auxiliary operations. Now, we present the most important auxiliary operations, which are needed for the specification of *insert*.

agg_n: t_agg * name → bool
a name appears in the aggregational components of the first argument.

at_comp_tt: t_agg * tabment → tabment
The atomic components of the given (target) tabment. Here, non-atomic collections and unimportant tags of the target are omitted.

at_comp_st: t_agg * tabment → tabment
The atomic component of a (source) tabment. Contrary to *at_comp_tt*, here, atomic values in optional names are included. If for example X, Z, and $M(Y)$ are atomic, and W is a sum aggregation then for a tabment *t* of type $X, Z?, W, M(Y), L(Y2), B(Y3, M(Y4))$ the operations yield in tabments of the following types :
at_comp_tt: X, M(Y)
at_comp_st: X, Z, M(Y)
The following operations simplify the specification of *insert* a little.

non_at_coll_st: t_agg * tabment → tabment
at_agg_comp_st: t_agg * tabment → tabment
at_agg_comp_tt: t_agg * tabment → tabment
The non-atomic collections in the above example are $L(Y2), B(Y3, M(Y4))$, the atomic and aggregational components with respect to the source are $X, Z, W, M(Y)$, and with respect to the target $X, W, M(Y)$

agg: t_agg * tabment * tabment → tabment
By $agg\ p\ tt\ st$ to all aggregational components of tt all corresponding values from st are added (sum) and the other components of tt remain unchanged. agg goes in st into depth, but not in tt. This operation is needed, if we go with *insert* in depth of the target tabment tt. The operation

lengthen: t_agg * tabment → tabment
replaces the Cartesian product. It is applied if we cannot insert into a target tabment, because of missing atomic values in the source tabment. By $lengthen$ each element of a non-atomic collection of the (source) tabment is lengthened by its superordinated atomic and aggregational components. By

```
let insert= fun p ttdtd ->
  let rec ins source target = match source,target with
     (Coll_t((c,s),sts)), tt  when List.mem (Coll_s(c,s)) p.atom=false
      -> List.fold_left (function x -> (function y -> ins y x)) tt sts         (*A*)
    | (Tuple_t sts),tt when eq_tabment(at_agg_comp_st p (Tuple_t sts)) Empty_t
      -> List.fold_left (function x -> (function y -> ins y x)) tt sts         (*B*)
    | st, tt when eq_tabment(at_agg_comp_tt p tt) tt  -> agg p tt st           (*C*)
    | st, (El_tab v)  -> El_tab v                                              (*D*)
    | st, (Tag0(n,t)) -> Tag0(n, ins st t)                                     (*E*)
    | st, (Alternate_t(ss,t))  -> Alternate_t(ss,(ins st t))                   (*F*)
    | st, Empty_t              -> Empty_t                                      (*G*)
    | st, (Tuple_t tts)        -> Tuple_t(map (fun x->ins st x) tts)           (*H*)
    | st, (Coll_t((S1,s),[tt])) when type_t(occupy p ttdtd st s)=s
      -> if   in2 p s st [tt] ttdtd then Coll_t((S1,s),[agg p tt st])          (*I1*)
                                else Coll_t((S1,s),[tt])                       (*I2*)
    | st, (Coll_t((c,s),ts)) when type_t(occupy p ttdtd st s)=s &
      (List.mem c [Bag;List;List_minus;Bag_minus] or
      (List.mem c [Set;S1;Set_minus]) & not(in2 p s st ts ttdtd))
      -> Coll_t((c,s),(ins st (occupy p ttdtd st s))::ts)                      (*I3*)
    | st, (Coll_t((Any,s),tts)) -> Coll_t((Any,s),(st::tts))                   (*I4*)
    | st, (Coll_t((c,s),tts)) when type_t(occupy p ttdtd st s)=s
      -> let at=at_comp_tt p (occupy p ttdtd st s) in
         let tt2=try first_that (fun x->(eq_tabment (at_comp_tt p x) at)) tts
         with E_first_that -> raise(Fehler "Insert Fehler 2")
         and tts2=omit_first_that(fun x->(eq_tabment(at_comp_tt p x) at)) tts
         in Coll_t((c,s),((ins st tt2)::tts2))                                 (*I5*)
    | st, (Coll_t((c,s),tts))
      when equal_s(type_t(occupy p ttdtd st s)) s  = false
      ->  let l=(lengthen p st) in
          if empty_s1(l) then Coll_t((c,s),tts)                                (*J1*)
                         else ins l (Coll_t((c,s),tts))                        (*J2*)
    | _, _  -> raise(Fehler"insert Fehler 3")
  in fun st tt ->sort_t p (ins st tt);;
```

Fig. 4. Functional Definition of Stroke on the base of Insert in OCAML

the operation
occupy: t_agg * dtd * tabment * scheme → tabment
the given scheme is occupied by the atomic components of the tabment, where *Empty_t* results if no component exists in the tabment. The aggregational components are occupied by corresponding neutral values. Here, *dtd* abbreviates *(name * scheme) list*; the first component of the first element is always TABMENT.
eq_tabment: tabment * tabment → bool
is the equality relations for tabments and
type_t: tabment → scheme
is the scheme of an tabment.
in2: t_agg * scheme * tabment * tabment list * dtd → bool
describes whether the atomic components of the tabment occur as atomic components in an element of the list of tabments.
empty_s1: tabment → bool
A tabment is equal to *Empty_t* or a tuple of *Coll_t((S1,s),[])* components. The axioms of figure 4 describe the following situations. The insertion, element by element, of a non-atomic collection into a tabment is described by A. If the source is a tuple of non-atomic collections then the collections are inserted collection by collection (B). If the target is a tuple of atomic and aggregational components then at most aggregations have to be realized (C). If a single component is neither aggregational nor atomic then the rules D, E, F, respectively can be applied. G is trivial. The insertion into a tuple is realized componentwise (H). The insertion into an optional value changes the value only in the case that the atomic components are equal to the corresponding components of *st* (I1) otherwise the target remains unchanged (I2). The insertion of a new element into a collection is described by I3. The new element is generated by *occupy* and *insert*. In a target set it is required that the atomic components of *st* are yet not contained in the set. An element *st* is simply added to an ANY-collection (I4). If *tts* contains for *M* or *M-* already an element *tt2* with the atomic components of *st* then we have to omit *tt2* from the list and to add "*insert st tt2*" to the list (I5). If there are not enough atomic components of *st* for the insertion

into a collection it is tried to lengthen *st*. In general the new source tabment contains new collections with elements with more atomic components. After the insertion of a complete source structure the target tabment is sorted for all non-atomic sets and multisets at all levels.

VI. Experimental Evaluation

In the following we will consider only four examples for comparisons. *stroke* denotes the functional and *stroke+* the procedural implementation. We used the following test environment:
CPU: Intel(R) CORE(TM) DUO T2250 @ 1.73 GHz
Primary storage: 1024 MB
Operating system: Ubuntu 8.04 with Kernel Linux 2.6.24-23 generic
Hard disk: SATA 80 GB (5400 RPM)
Input: **zufall1000t.xml:** L(TUP) TUP=(X,Y,Z)
QueryA:

```
$T:=TIME
aus  "zufall1000t.xml"
gib+ M(X,M(Y,B(Z)))
ext T34:=TIME - $T
```

The above aus-part can be used only with ”gib+”

```
$T1:=TIME
aus doc("zufall1000t.xml")
$T2:=TIME
gib M(X,M(Y,B(Z)))
ext T3:=$T2 - $T1
ext T4:=TIME - $T2
```

```
<results>
{
 let $t := doc("zufall1000t.xml")//TUP
 for $x in distinct-values($t/X)
 order by $x
 return <TUP1>
        <X>{$x}</X>
        {for $y in distinct-values
        ($t[X=$x]/Y)
         order by $y
         return <TUP2>
               <Y>{$y}</Y>
               {for $z in ($t
               [X=$x and Y=$y]/Z)
               order by $z
               return
               $z}
               </TUP2>}
        </TUP1>
}
</results>
```

The time for XQuery was measured with the GALAX-system [8] by the following command:
time galax-run program.xq -language xqueryp
QueryB:

```
$T:=TIME
aus "zufall1000t.xml"
gib+ M(Z,M(Y,B(X)))
ext T34:=TIME - $T
```

```
$T1:=TIME
aus doc ("zufall1000t.xml")
$T2:=TIME
gib M(Z,M(Y,B(X)))
ext T3:=$T2 -$T1
ext T4:=TIME - $T2 ,
```

```
<results>
{
 let $t := doc("zufall1000t.xml")//TUP
 for $z in distinct-values($t/Z)
 order by $z
 return <TUP1>
        <Z>{$z}</Z>
        {for $y in distinct-values
        ($t[Z=$z]/Y)
         order by $y
         return <TUP2>
               <Y>{$y}</Y>
               {for $x in
               ($t[Z=$z and Y=$y]/X)
               order by $x
               return
               $x}
               </TUP2>}
        </TUP1>
}
</results>
```

Now, we consider queries which are similar to the queries of the introduction. Here, zufall90000t.xml is a file of 300 tuples with 90000 A-values. Its type is TUP* and TUP is of type A,L(C). We omit the stroke+ programs:
QueryC:

```
$T1:=TIME
from doc ("zufall90000t.xml")
$T2:=TIME
stroke L(A,C)
ext T3:=$T2 -$T1
ext T4:=TIME - $T2
```

```
<results>
{ for $b in doc("zufall90000t.xml")//TUP
  for $a in $b/A
  for $c in $b/C
  return
      <result>
          { $a }
          { $c }
      </result>
}
</results>
```

TABLE IV
CPU TIMES FOR QUERIES A - D IN SECONDS

Query	procedural	functional	XQuery (Galax)
QueryA	0.096	T4: 0.048 T3: 0.772	7.608
QueryB	0.104	T4: 0.076 T3: 0.748	13.385
QueryC	3.248	T4: 1.312 T3: 433.647	24.49
QueryD	5.536	T4: 51.34 T3: 435.999	572.992

QueryD:

```
$T1:=TIME
from doc("zufall90000t.xml")
$T2:=TIME
gib M(C,M(A))
ext T3:=$T2 - $T1
ext T4:=TIME - $T2
```

```
<results>
{ let $r := doc("zufall90000t.xml")
  /root
  for $c in distinct-values($r/TUP/C)
  let $ts := $r/TUP[C=$c]
  order by $c
  return
    <TUP>
      <C>{$c}</C>
      { for $a in distinct-values($ts/A)
        order by $a
        return <A>{$a}</A> }
    </TUP>
}
</results>
```

The parser of the functional implementation is a DOM-parser. We plan to substitute it by an SAX-parser. The CPU times for the procedural implementation and XQuery implementation in Table IV are the total times.

VII. RELATED WORK

The *restruct* operation of [3] is most similar to our *stroke* operation. But, there are deep differences. *Stroke* is defined for arbitrary XML input documents and *restruct* only for V-relations. V-relations are non-first-normal-relations, which contain at each level a key, which consists of elementary fields only. *Stroke* allows not only sets in in- and output, but also bags, lists and any collections. Behind *stroke* is an algebraic specification and behind the kernel of *stroke* additional a procedural algorithm. Behind *restruct* is a set of facts, which is build out of all joinable flat segments of the given V-relation. Therefore this operation uses contrary to *stroke* the Cartesian product in its definition. *Stroke* only uses a *lengthen* operation, by which tuples are lengthened by its uniquely existing superordinated segments. Because it is allowed that a superordinated component is a collection, the Cartesian product can be simulated. Using not the Cartesian product concept fits to the principles of **minimal cost** and **minimal result information**. The restructuring of XML-documents in XQuery differs deeply from the restructuring with *stroke*. The main difference is that the XQuery user has to find alone the way to restructure data, but the OttoQL user has to write only the DTD. We finally only want to remark that the considerations of query XMP: 1.1.9.4 Q4 are not completely correct. Namely, if we assume that a book contains an author *au* twice then the corresponding title would appear for *au* twice in the OttoQL query, but not in the XQuery program. OttoQL does not use node identity. The result of an (intermediate) operation is always a new document, with new nodes.

Schema-Free XQuery [9] allows to formulate precise queries without perfect knowledge of the document structure. The given document structure can change in a certain extent and the schema-free queries have not to be rewritten. In this paper the following examples are presented, which are correct for two given different schemas. In schema *A* the *year* tag is outside the *book* tag and in schema *B* within. Without problems we can formulate the example queries of this paper in OttoQL. For comparision purposes the first query is repeated from [9].

Query1: Find title and year of the publications, of which Mary is an author.

```
for $a in doc("bib.xml")//author,
    $b in doc("bib.xml")//title,
    $c in doc("bib.xml")//year
where $a/text()="Mary" and
    exists mclas($a,$b,$c)
return <result>{$b,$c}</result>
```

```
aus doc("bib.xml")
mit title:: author="Mary"
gib B(title, year)
```

Query2: Find additional authors of the publications, of which Mary is an author.

```
aus doc("bib.xml")
mit title:: author="Mary"
ohne author:: author="Mary"
gib B(author)
```

There are two big differences between both approaches. OttoQL requires the DTD of the given XML-documents and Schema-free XQuery needs additional nodes as aricle and book to compute the mclas (meaningful lowest common ancestors) of the nodes of the given query.

XSearch [10] is a semantic search engine for XML, with a simple syntax, suitable for naive users. As in Schema-free XQuery the queries can be applied for documents with varying schemas. We will consider examples of this paper:

Q1: Find pairs of titles and authors, belonging to the same article.

```
Q1(+title:, author:)
```

The + signals that a title has to exist in the result. **Q2** looks for volumes, authors with the name Kempster, and authors who have published with Kempster.

```
Q2(+volume:, +author:Kempster, author:)
```

If author has an additional name tag, then the query

```
Q3(+volume:, +name:Kempster, name:)
```

does not express the desired meaning. The corresponding OttoQL queries: Q1:

```
gib B(title,author?)
or better
gib M(title,L(author))
```

Q2:

```
mit title:"Kempster"
gib M(volume,M(author))
```

Q3: (here the query is correct)

```
mit title:"Kempster"
gib M(volume,M(name))
```

We see that XSearch differs again deeply from OttoQL.

- Selection and restructuring are interwoven.
- $Q(+A1, +A2, ..., +An)$ corresponds to the target structure $L(A1, ..., An)$ (all-pair semantic). Missing "+" signs do not correspond exactly to "?" signs.
- Star semantics corresponds to $L(A1, A2?, A3?, ..., An?)$.
- Restructuring in XSearch uses Cartesian product (for example Q2). In a target structure $L(volume, author, author?)$ both author components will be occupied by the same author.
- To find semantically related nodes XSearch requires suitable tuple and collection tags.
- XSeach does not need a DTD.
- Queries of type Q(A:, :b) cannot be expressed in the present version of OttoQL.

VIII. Future Work

Firstly, we have to replace our dom parser by an SAX parser. Then we have to generalize our procedural implementation. Further, we or somebody else, have to develop and prove optimization rules for *stroke* and the other operations of OttoQL. We should try to use *stroke* in a search engine for XML-documents. If we realize a pilot system on top of a relational system, then we should not only allow flat source structures. By the program fragment

```
aus STUDFLAT
rename STID by STUDID
ext EXAMFLAT at SEX    # extension
mit STUDID=STID       # join condition
```

for example results a structured table of type M(STID, NAME, SEX, M(STID, COURSE, MARK)), which has in many situations advantages in comparison to a flat table. These are: more efficient restructuring, aggregations are possible at student level and exam level simultaneously, different join operations can be used by different conditions, additional selection possibilities,... .

IX. Conclusions

The axioms of our functional definition of *stroke* (Figure 4) make clear that XML-documents should by defined by simple generating operations like Tag0, Coll_t, Tuple_t,... and not by the both operations Pcdata and Element. We believe that it is possible to develop OttoQL with *stroke*, as one of its most powerful operations, to an easy to use query language for XML and databases. If we extend the selection mechanisms by keyword queries combined with *stroke* then OttoQL can in our opinion also be extended to a language for search engines for Intranet or Internet of XML-data and tables. There are a lot of advantages of such an approach. For example:

- The end-user has to learn only one computer language.
- Because of keyword queries the user can get familiar with the language step by step.
- The results of an Intranet- or Internet-query can be handled with the same tools as for querying the Intranet resp. Internet.

Acknowledgment

I would like to thank W. Reichstein for his first proceduaral one step implementation of *stroke* for HSQ-files (hierarchcal sequential files), D. Schamschurko for his first functional implementation in Caml-Light for non-first-normal-form relations, and Xuefeng Li for his procedural implementation for XML files.

References

[1] K. Benecke and M. Schnabel. Internet server for ottoql. [Online]. Available: http://otto.cs.uni-magdeburg.de/otto/web/index.html

[2] K. Benecke, "A powerful tool for object-oriented manipulation," in *On Object Oriented Database: Analysis, Design & Construction*. IFIP TC2/WG 2.6 Working Conference, July 1991, pp. 95–121.

[3] S. Abiteboul and N. Bidot, "Non-first-normal-form relations: An algebra allowing data restructuring," *J. Comput. System Sci*, no. 5, pp. 361–393, 1986.

[4] S. Boag, D. Chamberlain, and D. F. et. al, "Xquery 1.0: An xml query language," *J. Comput. System Sci*. [Online]. Available: http://www.w3.org/TR/xquery/

[5] D. Chamberlain et. al. (ed.), "Xml query use cases." [Online]. Available: http://www.w3.org/TR/xmlquery-use-cases

[6] H. Reichel, *Initial Computability, Algebraic Specifications, and Partial Algebras*. Oxford, UK: Clarendon Press, 1987.

[7] E. Chailloux, P. Manoury, and B. Pagano, *Developing Applications With Objective Caml*, Paris, France, 2000. [Online]. Available: http://caml.inria.fr/oreilly-book/

[8] Mary Fernndez, Jrme Simon,et. al., "Xquery implementation of galax." [Online]. Available: https://launchpad.net/ubuntu/+source/galax/1.1-4

[9] Y.Li, C. Yu, and H.V.Jagadish, "Schema-free xquery," in *VLDB Conference*, 2004, pp. 72–83.

[10] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "Xseearch: A semantic search engine or xml," in *VLDB Conference*, 2003.

Appendix

Here, we present the OCAML-code of the auxiliary functions mentioned in section V.

```
(* agg_n p n: n is an aggregational name with respect to p *)
let agg_n =fun p n ->
  (mem n p.all)    or
  (mem n p.exists) or
  (mem n p.summe)  or
  (mem n p.max)    or
  (mem n p.min)    or
  (mem n p.prod);;

(* at_comp p t: the atomic components of top level of t
   with respect to the target tabment *)
let at_comp_tt =
  let rec f p = function
      Tuple_t t -> elim_Tuple_t(Tuple_t (elim_Empty_t2(map (f p) t)))
    | El_tab v  -> if   mem (Inj (type_v v)) p.atom
      then El_tab v
      else Empty_t
    | Tag0(n,t) -> if   mem (Inj n) p.atom
      then Tag0(n,t)
      else (if mem n p.unimportant
            then f p t
            else Empty_t)
    | Alternate_t (ss,t)  -> if   mem (Alternate_s ss) p.atom
      then Alternate_t (ss,t)
      else f p t
    | Coll_t ((c,s),ts)   -> if   mem (Coll_s (c,s)) p.atom
      then Coll_t((c,s),ts)
      else Empty_t
    | Empty_t             -> Empty_t
  in  f;;
```

```
(* at_comp_st p t: the atomic components of top level of t for a source tabment t *)
let at_comp_st =
  let rec f p = function
      Tuple_t ts -> elim_Tuple_t(Tuple_t (elim_Empty_t2 (map (f p) ts)))
    | El_tab v   -> if   mem (Inj (type_v v)) p.atom
      then El_tab v
      else Empty_t
    | Tag0(n,t)  -> if   mem (Inj n) p.atom
      then Tag0(n,t)
      else (if agg_n p n
            then Empty_t
            else f p t)
    | Alternate_t(ss,t)  -> if   mem (Alternate_s ss) p.atom
      then Alternate_t(ss,t)
      else f p t
    | Coll_t((S1,s),[t]) -> if   mem (Coll_s(S1,s)) p.atom
      then Coll_t((S1,s),[t])
      else f p t
    | Coll_t((S1,s),[])  -> Coll_t((S1,s),[])
    | Coll_t ((c,s),ts)  -> if   mem (Coll_s (c,s)) p.atom
      then Coll_t((c,s),ts)
      else Empty_t
    | Empty_t            -> Empty_t
  in  f;;

(* non_at_coll_st p t: non-atomic collections (of top level) of t *)
let non_at_coll_st =
  let rec f p = function
      Tuple_t ts -> elim_Tuple_t(Tuple_t (elim_Empty_t2(map (f p) ts)))
    | Tag0(n,t)  -> if   (agg_n p n) or (mem (Inj n) p.atom)
      then Empty_t
      else f p t
    | El_tab v             -> Empty_t
    | Alternate_t(ss,t)    -> if   mem (Alternate_s ss) p.atom
      then Empty_t
      else f p t
    | Coll_t ((S1,s),[t]) -> if   mem (Coll_s(S1,s)) p.atom
      then Empty_t
      else f p t
    | Coll_t ((S1,s),[])  -> Empty_t
    | Coll_t ((c,s),ts)   -> if   mem (Coll_s (c,s)) p.atom
      then Empty_t
      else Coll_t ((c,s),ts)
    | Empty_t             -> Empty_t
  in f;;
```

```
(* at_agg_comp_tt p t: atomic and aggregational top level components
   of target tabment t *)
let at_agg_comp_tt =
  let rec f p = function
      Tuple_t ts  -> elim_Tuple_t(Tuple_t (elim_Empty_t2(map (f p) ts)))
    | Tag0(n,t)   -> if   (agg_n p n) or (mem (Inj n) p.atom)
                     then Tag0(n,t)
                     else Empty_t
    | El_tab v    -> let  n=type_v v in
                     if   agg_n p n  or (mem (Inj n) p.atom)
                     then El_tab v
                     else Empty_t
    | Alternate_t(ss,t)   -> if   mem (Alternate_s ss) p.atom
      then Alternate_t (ss,t)
      else Alternate_t(ss,f p t)
    | Coll_t ((c,s),ts)   -> if   mem (Coll_s (c,s)) p.atom
                             then Coll_t((c,s),ts)
                             else Empty_t
    | Empty_t             -> Empty_t
  in f;;

(* at_agg_comp_st p t: atomic and aggregational top level components
   of source tabment t *)
let at_agg_comp_st =
  let rec f p = function
      Tuple_t ts  -> elim_Tuple_t(Tuple_t (elim_Empty_t2(map (f p) ts)))
    | Tag0(n,t)   -> if (agg_n p n) or (mem (Inj n) p.atom)
                     then Tag0(n,t)
                     else f p t
    | El_tab v    -> let n=type_v v in
        if   agg_n p n  or (mem (Inj n) p.atom)
        then El_tab v
        else Empty_t
    | Alternate_t(ss,t)   -> if   mem (Alternate_s ss) p.atom
      then Alternate_t (ss,t)
      else f p t
    | Coll_t ((S1,s),[t]) -> if   mem (Coll_s(S1,s)) p.atom
      then Coll_t((S1,s),[t])
      else f p t
    | Coll_t((S1,s),[])   -> Coll_t((S1,s),[])
    | Coll_t ((c,s),ts)   -> if   mem (Coll_s (c,s)) p.atom
      then Coll_t((c,s),ts)
      else Empty_t
    | Empty_t             -> Empty_t
  in f;;
```

```
(* agg p tt st: the toplevel aggregational components of tt are extended
   by all corresponding aggregational values of st *)
let rec agg p target source = match target,source with
     (Tag0(n,t2)), (Tag0(n1,t1)) ->
       if n=n1 then
         if mem n p.summe  then Tag0(n,plus2_t t2 (sum_t t1))  else
           if mem n p.all    then Tag0(n,and_t  t2  (all_t t1))  else
             if mem n p.exists then Tag0(n,or_t   t2  (ex_t t1))   else
               if mem n p.max    then Tag0(n,max_t (Tuple_t[t2; t1]))  else
                 if mem n p.min    then Tag0(n,min_t (Tuple_t[t2; t1]))  else
                   if mem n p.prod   then Tag0(n,mul2_t t2  (prod_t t1)) else
                     Tag0(n, t2)
       else agg p (Tag0(n,t2)) t1
   | (Tag0(n,t)), st when not(is_inn_comp [n] st)-> Tag0(n,t)
   | (Tag0(n,t)), st ->
       if agg_n p n
       then
         match st with
             (Tuple_t sts) -> it_list (fun x y -> agg p x y) (Tag0(n,t)) sts
           | (Coll_t (_,sts)) -> it_list (fun x y -> agg p x y) (Tag0(n,t)) sts
           | (Alternate_t(ss, st)) -> agg p (Tag0(n,t)) st
           | _ -> raise (Never "agg")
       else Tag0(n, t)
   |  (Tuple_t ts), st -> Tuple_t (map (fun x -> agg p x st) ts)
   |  t, _ -> t;;

(* lengthen p t: each element of a non-atomic component of t is lengthened by
   the atomic and aggregational components of t *)
let rec lengthen =
  let f x y = elim_Tuple_t(Tuple_t (elim_Empty_t2 [x;y]))
  in  fun p st ->
    match non_at_coll_st p st with
        (Tuple_t ts) -> let sta = at_agg_comp_st p st in
          Tuple_t (map (fun x -> lengthen p (f sta x)) ts)
      | (Coll_t((c,s),ts)) ->
          let statag = at_agg_comp_st p st in
          let s1n= names_s(type_t(statag)) in
          let ts2 = map (fun x -> forget2_t x s1n) ts in
            (Coll_t ((c,elim_Tuple_s(Tuple_s (elim_Empty_s2[type_t statag;
                                                                 forget_s s s1n]))),
                     map (f statag) ts2))
      | _ -> Empty_t;;
```

```
(* occupy p dtd st s: occupy all possible atomic components of s by corresponding
   atomic components of st and all aggregational top level components by neutral
   values and all nonaggregational collections by empty collections *)
let occupy= fun p ttdtd ->
  let rec occu source target_s = match source,target_s with
      st, (Inj n) ->
        if mem n p.summe  then Tag0(n,El_tab(Int_v zero_big_int))  else
          if mem n p.exists then Tag0(n,El_tab(Bool_v false))        else
            if mem n p.all    then Tag0(n,El_tab(Bool_v true))         else
              if mem n p.max    then Tag0(n,Empty_t)                     else
                if mem n p.min    then Tag0(n,Empty_t)                     else
                  if mem n p.prod   then Tag0(n,El_tab(Int_v unit_big_int))  else
                    if mem (Inj n) p.atom
                    then
                      (if (type_t st = Inj n)
                       then st
                       else  (match st with
                                  (Tuple_t sts)->(try first_that
                                  (fun x -> (type_t (occu x (Inj n)) = Inj n)) sts
                                             with E_first_that ->  Empty_t)
                                | _ -> Empty_t))
                    else
                      (if mem n p.unimportant
                       then  (let oc=occu st (type_n n ttdtd) in
                                if type_t(oc)=type_n n ttdtd & type_t(oc) != Empty_s
                                then Tag0(n,oc)
                                else Empty_t)
                       else Empty_t)
    | st, (Tuple_s ss)       -> elim_Tuple_t(Tuple_t(elim_Empty_t2 (map (occu st) ss)))
    | (Coll_t((c,s),sts)),(Coll_s(c',s')) ->
                                if   List.mem (Coll_s(c',s')) p.atom & c=c' & s=s'
                                then t7
                                else if   List.mem (Coll_s(c',s')) p.atom
                                                    or List.mem (Coll_s(c,s)) p.atom
                                    then Empty_t
                                    else Coll_t((c',s'),[])
    | st, (Coll_s(c,s))  -> if   mem (Coll_s(c,s)) p.atom
                            then (match st with
               (Tuple_t sts)->(try first_that
                 (fun x -> (type_t (occu x (Coll_s(c,s))) = Coll_s(c,s))) sts
                  with E_first_that ->  Empty_t)
             | _ -> Empty_t)
      else Coll_t((c,s),[])
    | (Alternate_t(ss1,st)), (Alternate_s ss2)  ->
                          if   List.mem (Alternate_s ss2) p.atom & ss1=ss2
                          then t7
                          else if   List.mem (Alternate_s ss1) p.atom
                               then Empty_t
                               else occu st (Alternate_s ss2)
    | st, (Alternate_s ss) -> if List.mem (Alternate_s ss) p.atom
                         then
                         (match st with
                         (Tuple_t sts)->(try first_that
                         (fun x -> (type_t (occu x (Alternate_s ss)))
                                                    = Alternate_s ss) sts
                                with E_first_that ->  Empty_t)
    | _ -> Empty_t)
      else let s1=(try first_that (fun x-> x=type_t(occu st x )) ss
                                      with E_first_that -> Empty_s) in
        if s1=Empty_s
        then Empty_t
        else Alternate_t (ss,(occu st s1))
    | st, Empty_s -> Empty_t
  in fun st1 s -> occu (at_comp_st p st1) s;;
```

```
(* in2 p s t tts dtd: the atomic target components of t are contained as components in
   the list of tabments tts *)
let rec in2 p ss t tts dtd = match ss,t,tts,dtd with
    s, st, [], ttdtd -> false
  | s, st, (t::ts), ttdtd ->
      (eq_tabment (at_comp_tt p (occupy p ttdtd st s)) (at_comp_tt p t))
      or in2 p s st ts ttdtd;;
```