Optimization Rules for the End User Programming Language OttoVonG

1 Introduction

At first, we have a look at a table, on which the failing of certain well-known commuting rules for selection can be demonstrated.

```
<< M( A,   L( B,      C))::
      1       2       3
              4       5 >>
     (tabment T0)
```

In the following σ is the selection operation.
Counter-examples for commuting rules of selection:

(a) $\sigma_{B=4}(\sigma_{B::C=3}(T0)) \neq \sigma_{B::C=3}(\sigma_{B=4}(T0))$

The left hand side is an empty tabment, contrary to the right hand side. The reason is that the condition *B::C=3* selects elements of the **fix level** of *(A::)B=4* (*B::C=3* **refers** to a fix level (B,C) of the quantified condition *B=4*.). In section 4 we shall see that we can commute both above conditions in another sense. *B=4* can absorb *B::C=3*.

(b) $\sigma_{B::pos(B)=1}(\sigma_{B::B=4}(T0)) \neq \sigma_{B::B=4}(\sigma_{B::pos(B)=1}(T0))$

The left hand side contains the subtuple (<<B:: 4>>,<<C:: 5>>),
whereas the L(B,C)-collection of the right- hand-side is empty. The reason is again that the condition *B::B=4* selects in the fix level of the position selecting condition *B::pos(B)=1*.

(c) $\sigma_{B::C=3}(\sigma_{L(C)[-1]=5}(T0)) \neq \sigma_{L(C)[-1]=5}(\sigma_{B::C=3}(T0))$

Here, we have again a position selecting and a content selecting condition. L(C)[-1] desciribes the last C-element of the list L(B,C). The result of the left hand side contains an inner singleton and the result of the right hand side is empty. Here, *L(C)[-1]=5* refers to *(A,L(B,C))* and has the fix level *(B,C)*.

(d) $\sigma_{B=2}(T0) \neq T0$ **except** $(\sigma_{ne(B=2)}(T0))$

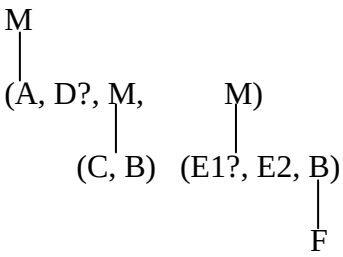
Here, *ne* (slavic) is the negation and *except* the set difference.The left hand side is T0 and the right hand side the empty set of type M(A,L(B,C))

In section 2 some basic definitions like superordination of attributes or hierarchical paths are introduced. Although we have presented the failing of simple commuting rules in the introduction, there are enough cases, which are presented in section 3, where conditions can be commuted. Also in the case, where 2 conditions do not commute, there may be potential for optimization. If a condition selects in the fix level of the following condition, then the following condition can absorb the preceeding one. In sections 5 and 6 we consider the optimization potential of a gib-part (stroke-operation). Stroke may be accompanied by loss of information. This loss of information can explicitely expressed in preceeding conditions or in a preceeding forget operation. Rules with extension operation and some cases, where they fail are considered in section 7.

2 Some Basic Definitions

A **name** is a string, which is used for tags (column names). A **slashed name** is recursively defined. Each name is a slashed name. If *sn* is a slashed name and *n* a name, then *sn/n* and *sn//n* are slashed names. TABMENT is a special name for a whole tabment, which is allowed only as a first name in a slashed name.
An **attribute** is defined recursively again. Each *slashed name* is an attribute. If *sn* is a slashed name, then *pos(sn)* is an attribute. If *sn* is a slashed name, then each of the following terms *L(sn), B(sn), M(sn), A(sn)* is an attribute. If *att* is an attribute, then also *att[i]* is an attribute, where *i* is an integer.
That means for example: Reasonable attributes of a tabment of type M(A,B,M(C,D,L(E,F))) are A; B; ... M(A); M(B); ...; M(F); B(F); L(F); C[1] (first component of C, if C is a tuple and first element of C, if C is a collection); L(C)[-1] (last element of L(C)).
A **collection attribute** is an attribute of type *C(sn)*, where $C \in \{ M, B, L, A\}$ or of type *sn*, if *sn* is a slashed name for a collection type or of type *att[i]*, if i-th component respectively element represents a collection.
Positional attributes are of type *pos(sn), **where** sn* is a slashed name or *att[i]*, if *att* is a collection attribute.
A name *A* is **superordinated** to a name *B* in a scheme *s*, if an occurrence of *A* is in the tree representation of *s* higher than an occurrence of *B* or in the same level as *B*.
The tree representation of

M(A, D?, M(C, B), M(E1?, E2, B(F)) is for example:



(tree1)

In this scheme *A* is superordinated to each other name, *C* is only superordinated to *B*, *E1* is superordinated to *E2* and *F* and *E2* to *E1* and *F*. The notion of superordination can easily be extended to attributes, which contain only simple slashed names. Namely, if we define that *M(A)* is always one level higher than *A* and *A[i]* and is at the same level as *A*. In the same way *pos(A)* is at the same level as *A*. Therefore *F* is not superordinated to *E2*, but *M(F)* is superordinated to *E2* (at the same level). *L(F)* is superordinated to *F*.

Two attributes *A* and *B* are on a **hierarchical path in a scheme *s***, if there exist occurrences of *A* and *B* in *s* such that the occurrence of *A* is superordinated to the occurrence *B* or an occurrence of *B* is superordinated to an occurrence of *A*.

Examples:
A and *B* are on a hierarchical path in *M(A, B, C)*; *M(A?, M(B))*; *M(C,M(B1,B,L(A1,A)))*; *M(A,M(B),M(B))*, but not in *M(C, M(B), M(A))*, and not in *M(C, M(D, M(A)), M(B)))*.
M(A) and *A* are on a hierarchical path in *M(A, B)*.

In the following we want to define the notion of a hierarchical path with respect to a given DTD (document type definition).

Example:

NAME	TYPE
TABMENT	L(A?, B, M(C, D))
A	TEXT
B	TEXT
C	E, F
D	M(H)
E	TEXT
F	M(G)
G	ZAHL
H	TEXT
J	TEXT

(dtd1)

But at first the following

Definition

A **complete slashed name** *sn* with respect to a DTD is a slashed name $sn=n_1/n_2/n_3.../n_i$, where $n_2 \in names(type(n_1))$, $n_3 \in names(type(n_2))$,..., $n_i \in names(type(n_{i-1}))$.

Examples:

TABMENT/C, TABMENT/C/E/TEXT and TABMENT/D/H are complete slashed names with respect to dtd1, but not TABMENT/E and C/TEXT

The definition of a hierarchial path requires the definition of set of schemes of a column name (attribute) *n*.

$\mathbf{H_1}(n,dtd) = \{type(n)\}$

$\mathbf{H_{i+1}}(n,dtd) = H_i(n,dtd) \cup \{ s: \exists s'' \in H_i(n,dtd) \ \& \ \exists(n',s') \in dtd \ \& \ s=replace(s'', n', s') \}$

replace(s", n', s) results from scheme *s"* by replacing an occurrence of name *n'* of *s"* by scheme *s*

$\mathbf{H(n,dtd)} = \cup^{\infty}_{i=1} H_i(n,dtd)$

Example:

H(C,dtd1) = {(E,F); (TEXT,F); (E,M(G)); (E,M(ZAHL));(TEXT,M(G));(TEXT,M(ZAHL))}

Definition:

H(dtd) = H("TABMENT", dtd) (hull of schemes of a DTD)

left(dtd) = {n: (n, s) ∈ dtd}

right(n, dtd)={n: n ∈ names(H(n, dtd))}

Definition:

A name *n* is **recursive** with respect to a DTD *dtd*, if $n \in right(n, dtd)$. A DTD *dtd* is recursive, if it contains a recursive name *n*, which is contained in right("TABMENT",dtd).

Lemma:

a) If H(dtd) is infinite, then the DTD dtd is recursive.
b) If a DTD *dtd* is recursive and monotone (each right side s of (n,s) contains more than one name, then *H(dtd)* is infinite.

Definition:

Two attributes *A* and *B* without slash and doubleslash are on a **hierarchical path with respect to the DTD** *dtd*, if there exists a scheme $s \in H(dtd)$, such that *A* and *B* are on a hierarchical path on *s*.

Definition:

Two slashed names *A=A1/A2* and *B*, not containing doubleslash ("//"), with simple name *A1*, are on a **hierarchical path** with respect to the DTD *dtd*, if *A'* and *B'* are on a hierarchical path with respect to dtd', where *A'* and *B'* result from *A* and *B* bei ommiting all strings "*A1/*" and *dtd'* results from *dtd* by omitting *(A1,type(A1))* from *dtd* and replacing each *A1* of a right hand side of dtd by *type(A1)*.

Two slashed names A and B are on a **hierarchical path,** if each occurence of a double slash X//Y in A or B can be replaced by a sequence X/X1/X2/.../Xi/Y and the resulting slashed names **A', B'** are on a hierarchical path.

Definition

The **final scheme** *s* of an unrecursive DTD *dtd* is a scheme $s \in H(dtd)$, which results from *type(TABMENT)* by all possible replacements except replacements of names *n* with $type(n) \in \{TEXT, ZAHL, PZAHL, BOOL, BAR\}$.

Example:

final-scheme(dtd1)=L(A?,B,M(E,M(G),M(H)))

The final scheme is used in the tab-representation of unrecursive XML-documents.

If the final scheme contains no elementary tags (TEXT,...) and no name occurs twice, then the final scheme describes a non-first-normal-form relation. (A final scheme can contain elementary types (ZAHL,ZAHL,ZAHL) if RGB is for example of this type.)

A slashed name *sname* can be extended to a **complete slashed name** $sname'=n_1/n_2/.../n_i$ *(*containing no doubleslash) such that n_1 is the first and n_i is the last name of *sname* and each name of *sname* occurs in *sname'* in the same order and s_k (k=1..i-1) from $(n_k,s_k) \in$ dtd contains n_{k+1}. It is possible that a slashed name can be extended to several complete slashed names; if the DTD *dtd* is recursive, then infinite many complete slashed names can exist.

Each condition *sname1::cond1* contains a level determiner *sname1*. By this condition elements of type *level(sname1)* are selected.

A **level** *level(sname, dtd)* **of a slashed name** *sname* with respect to a DTD *dtd* is each scheme *lev*, which can be determined in the following way:

1. Extend the slashed name *TABMENT//sname* to a complete slashed name $sname'=n_1/n_2/.../n_i$
2. Beginning with $k=i$ test whether n_k is in a proper collection $c(s)$ $(c \in \{ M, B, L, A\})$ of $type(n_{k-1})$. If this is the case, then take the most inner collection *c(lev)*, containing n_k. lev is a desired level. If this condition fails (n_k is not in a collection of $type(n_{k-1})$, then replace *k* by *k-1* and repeat the procedure.

Examples:

level(H, dtd1) = H (*sname'=TABMENT/D/H*)
level(A, dtd1) = A?, B, M(C, D) *(sname'=TABMENT/A)*
level(C, dtd1) = level(F, dtd1) = (C, D) (*sname'= TABMENT/C/F*)
level(C/F, dtd1) = (C, D) *(sname'= TABMENT/C/F)*
level(C//G, dtd1) = G *(sname'= TABMENT/C/F/G)*
level(TEXT,dtd1) = a) A?,B,M(C,D) b) C,D c) H

An **expression** can be defined recursively again:

1. Each attribute is an expression; each tabment is an expression
2. If A and B are attributes, then A+B; A - B; …; (A, B); A=B; A<B; A in B; A in2 B;… are expressions. (the last four are Boolean valued)
3. If A is an expression, then sin(A); cos(A); … are expressions
4. If A, B, C are expressions, then A subtext (B,C); … is an expression.

A condition *sname1::cond1* is called **simple**, if *cond1* is a well defined Boolean valued expression and if it contains no positional attribute and each slashed name is of elementary

type (TEXT, …) and the deepest name from *cond1* is at the same level as *sname1*. In this case the condition "contains" no (implicit) existential quantifier.
A condition is called **relational**, if it is of type *sname1:::cond1* and *sname1::cond1* is simple.
A condition *sname:::cond* is an abbreviation of the sequence of conditions:

$sname_1$:: cond
$sname_2$:: cond
$sname_3$:: cond
…
$sname_k$:: cond,

where the sequence contains excactly one row for each proper collection, which contains *$sname_1$ = sname*.
In the above given tabment *T0 B:::C=3* and *A:::A=1* are relational conditions, contrary to *A:::B=4*. In this case *B:::C=3* is an abbreviation of *B::C=3* followed by *A::C=3* or vice versa.
A condition can have zero, one, or several fix levels. The simple condition *B::C=3* (consider tabment T0) refers to level(B) (=level(C)) = *(B,C)* has no fix level. Generally, simple conditions have no fix levels. The (quantified) condition *B=4 (*abbreviates *A::B=4)* refers to *(A,L(B,C))* that means it selects *(A,L(B,C))*-elements and has the fix level *(B,C)*. That means the truth value of the condition depends on the *L(B,C)*-collection. If *(B, C)*-elements are eliminated by another condition, then the truth value for evaluating a *level(A)*-element may change.
A scheme *lev* is a **fix level** of a condition *sname1::cond1*, if one of the following cases is satisfied:
(1) *cond1* contains an attribute *C(sname)*, or *pos(sname) with lev=level(sname), or att[i]* and *att* is of collection type and *lev=level(att)*. (The level of *att[i]* is the same as the level of *att*.)
(2) *cond1* contains an attribute *att*, where a slashed name *sname2* exists such that *lev=level(sname2)*, where *sname2* is superordinated to *att* and *sname1* is superordinated to *sname2* and *sname1* and *sname2* are not from one level.
(3) *cond1* contains an attribute *att* and in *att* is contained an inner collection *C(lev)*.
Examples with respect to dtd1:
(1)
C:: M(G)=M(1 2)
has fix level (G), but not (C,D)
C:: pos(G) <40
has fix level (G), but not (C,D)
C:: M(G)[3]=2
has fix level (G), but not (C,D)
(2)
A:: G=1
has fix the levels: (G); and (C,D); but not (A?,B,M(C,D))
A:: 1 in M(G)
has fix level (C,D) (*sname2=C*), but not (A?,B,M(C,D)) (and (G) because of (1))
(3)
D=M(1 2)
has fix level *H*.

3 Commuting Operations

To prove commuting rules for condtions we have to have at first a closer look to commuting rules for extensions.

ext-ext

$\varepsilon_{assi1}(\varepsilon_{assi2}(tab)) = \varepsilon_{assi2}(\varepsilon_{assi1}(tab))$,
this rule holds, for assignments:

assi1: name_tup1 := expr_tup1 at sname1
assi2: name_tup2 := expr_tup2 at sname2

if the following three conditions hold:

(1) *sname1* ≠ *sname2*
(2) if *sname1* is not contained in *sname2* or in a name of *expr_tup2*
and *sname2* is not contained in *sname1* or in a name of *expr_tup1*
(3) *name1_tup* and *name2_tup* introduce only new names (not contained in the given tabment and not contained in the right sides of assi1 and assi2

Counter examples for ext-ext:

counter example1: (*sname1=sname2,* this example is of formal importance only)

```
A:=1
ext C:=3 at A
ext B:=2 at A
```

results in

```
<<  A,    B,    C::
    1     2     3>>
```

```
A:=1
ext B:=2 at A
ext C:=3 at A
```

results in the same tabment, but with switched column names:

```
<<  A,    C,    B::
    1     3     2>>
```

This switching could be made undone by a following gib-part.

counter example2: (*sname1 = sname2* and *name_tup2* contains a name from given tabment)

```
A:=1
ext B:=2 at A
ext A:=3 at A
```

results in

```
<<  A,    A,    B::
    1     3     2>>,
```

but

```
A := 1
ext A:=3 at A
ext B:=2 at A
```

results in a tabment with 4 columns

```
<<  A,    B,    A,    B::
    1     2     3     2>>.
```

counter example3: (*sname1* is contained in *sname2*)

```
B := 1
tag0 A
ext    C:=B+1 at B
ext    D:=sum(A) at A
```

The absence of certain commuting rules for the selection, which we observed in the introduction, is not only a disadvantage. It is on the other hand a reason for the expressive

power of our data model. For further considerations we will use the following example tabments.

UNIVERSITY:
FACULTIES: M(FACULTY, DEAN, FACBUDGET)
INSTITUTES: M(INSTITUTE, MANAGER, BUDGET, FACULTY)
EMPS: M(ENO, NAME, FIRSTNAME, LOCATION, SALARY, SEX, PATENTCNT,
INSTITUTE, M(HOBBY), M(PROJECT, TIME))

Query 1a: Find from all employees from Magdeburg the 50 best earning.

aus	EMPS	
gib	B-(SALARY, NAME, LOCATION, SEX)	# sort by SALARY descending
mit	LOCATION=”Magdeburg”	# simple condition
mit	pos(SALARY) < 50	# position selecting condition

aus: from; gib: select; mit: where; B abbreviates bag;

1b: Give from the 50 best earning employees the employees from Magdeburg.

aus	EMPS
gib	B-(SALARY, NAME, LOCATION, SEX)
mit	pos(SALARY) < 50
mit	LOCATION=”Magdeburg”

It is evident, that the first condition of query 1a can also be realized before sorting.

Query 2: Give for each employee from Magdeburg all “important” projects.

aus	EMPS
mit	PROJECT:: TIME>10
mit	LOCATION = ”Magdeburg”

By query 2 on the source table the condition *PROJECT::TIME>10* is applied to all *M(PROJECT,TIME)*-collections and on the result the condition *LOCATION=”Magdeburg”* is applied. If *EMPS* is an XML-document, then the exchange of both conditions would probably improve efficiency. If EMPS is a file with a LOCATION-index, then the condition *LOCATION=”Magdeburg”,* should be applied at first, too. An eventually existing index for the column *TIME* is relatively useless, because no employee is eliminated by *PROJECT::TIME>10.*

sel-sel1

$\sigma_{sn2::c2}(\sigma_{sn1::c1}(tab)) = \sigma_{sn1::c1}(\sigma_{sn2::c2}(tab))$

If *sn1::c1* does not select in a fix level of *sn2::c2* and *sn::c2* does not select in a fix level of *sn1::c1*.

Remark1: If *sn1::c1* and *sn2::c2* are simple, then holds:

$\sigma_{sn2::c2}(\sigma_{sn1::c1}(tab)) = \sigma_{sn1::c1}(\sigma_{sn2::c2}(tab))$

This follows immedeately from sel-sel1, because simple conditions have no fix levels.

sel-sel2

$\sigma_{sn2:::c2}(\sigma_{sn1:::c1}(tab)) = \sigma_{sn1:::c1}(\sigma_{sn2:::c2}(tab))$

If *sn1:::c1* and *sn2:::c2* are relational. This rule is surprising, because both conditions are allowed to select in fix collection of the other condition.

From Relational Algebra is known that the selection of a conjunction of two conditions is equivalent to the successive application of the two conditions. It is clear that this rule cannot be applied, if both conditions refer to different levels. Query 1 teaches us that this rule fails also, if both conditions refer to the same level and one of the conditions is position selecting:

$\sigma_{pos(SALARY)<50\ i\ LOCATION="Magdeburg"}(EMPS) = \sigma_{LOCATION="Magdeburg"}(\sigma_{pos(SALARY)<50}(EMPS))$

$\neq \sigma_{pos(SALARY)<50}(\sigma_{LOCATION="Magdeburg"}(EMPS))$

Here *i* (Russian) is the conjunction. By the following example is demonstrated that also in the case that both conditions refer to the same collection scheme and both are not position selecting the above rule may fail.

If T1 is of type M(A,M(B),M(C)) and we consider the two conditions *B=2* and *C=3*

```
<<  M( A,   M(B), M(C))::
        1       2       3>>
        (tabment T1)
```

It holds $\sigma_{B=2\ i\ C=3}(T1) = \langle\langle M(A,M(B),M(C))::\rangle\rangle \neq$

$\neq \sigma_{B=2}(\sigma_{C=3}(T1)) = \sigma_{C=3}(\sigma_{B=2}(T1)) = T1$

The reason is that *T1* does not contain subtuples with a corresponding (*B,C)*-value. By the following example we demonstrate that also in the case that B and C are on a hierarchical path, the rule may fail:

```
<<  M( A,   M( B,   M( C))):
        1       2       3
                4         >>
        (tabment T2)
```

$\sigma_{B=4\ i\ C=3}(T2) = \langle\langle M(A,M(B,M(C)))::\rangle\rangle \neq$

$\neq \sigma_{B=4}(\sigma_{C=3}(T2)) = \sigma_{C=3}(\sigma_{B=4}(T2)) = T2$

Nevertheless we can formulate our rule:

sel-conj1

$\sigma_{sn::c1}(\sigma_{sn::c2}(tab)) = \sigma_{sn::c1\ i\ c2}(tab)$

This rule holds, if neither *c1* nor *c2* contains positional attributes and all attributes from *c1* and *c2* are on one hierarchical path..

sel-conj2

$\sigma_{sn:::c1}(\sigma_{sn:::c2}(tab)) = \sigma_{sn:::c1\ i\ c2}(tab)$

This second rule holds, if both conditions are relational and all attributes from c1 and c2 are on a hierarchical path.

sel-conj3

$\sigma_{sn::c1\ i\ c2}(tab) = \sigma_{sn::c1\ i\ c2}(\sigma_{sn::c2}(\sigma_{sn::c1}(tab)))$,

if c2 is not position selecting.

sel-intersect

$\sigma_{c1}(\sigma_{c2}(tab)) = \sigma_{c1}(tab)$ intersect $\sigma_{c2}(tab)$

Here is assumed that tab is a set or bag and c1 and c2 are not position selecting, which select in the topmost collection (given set or bag).

4 Absorption of a Condition

The following query is easy to write in OttoVonG, but more complicated to write in German or English.

Query 3: Select all employees, which work on at least one of a list of four special projects, with a time contingent greater than 10. From the set of projects for each such employee only projects from the four-element list are desired.

```
aus   EMPS
mit   PROJECT:: PROJECT in L(“otto”  ”SQL” ”XQuery” ”XML”)#simple condition
mit   TIME > 10                                          # existential condition
```

Because the quantified condition *TIME>10* selects *EMPS*-records, it should be earlier applied than the conditions *PROJECT:: PROJECT in L(“otto” ”SQL” ”XQuery” ”XML”)*, which selects projects. We have seen in the introduction that we cannot commute in general the two conditions. But, if the condition, which selects in the fix collection of the second condition is absorbed by the second, we can commute these both conditions. Then the following mit-parts results:

mit TIME>10 i PROJECT in L(“otto” ”SQL” ”XQuery” ”XML”)
mit PROJECT:: PROJECT in L(“otto” ”SQL” ”XQuery” ”XML”)

absorb-sel

$\sigma_{sn2::c2}(\sigma_{sn1::c1}(tab)) = \sigma_{sn1::c1}(\sigma_{sn1::c1\ i\ c2}(tab))$

Here holds: *sn1::c1* is a simple condition and *sn1::cond1* refers to a fix level of *sn2::c2*.

5 Smuggling a Condition

Query 4: Find all projects, on which an employee from Magdeburg works with a time contingent greater than 10, and collect for each such project all these employees with corresponding time. The output data have to be sorted by PROJECT and the inner collections by NAME.

```
aus    EMPS
mit    LOCATION=”Magdeburg”
mit    PROJECT:: TIME>10
gib    M(PROJECT, M(NAME, ENO, TIME))
```

This restructuring is accompanied by loss of information. This loss can be expressed by a condition in the following way:

```
aus    EMPS
mit    LOCATION=”Magdeburg”
mit    PROJECT:: TIME>10
mit    PROJECT = PROJECT
gib    M(PROJECT, M(NAME, ENO, TIME))
```

The last condition can absorb the last but one, and then both conditions can be summarized to a ":::"-condition.

```
aus    EMPS
mit    LOCATION=”Magdeburg”
mit    PROJECT::: TIME>10
```

gib M(PROJECT, M(NAME, ENO, TIME))

sel-before-stroke

$stroke_{dtd}(st) = stroke_{dtd}(\sigma_{att1::att2=att2}(st))$

We can put *att1::att2=att2* ahead to the restructuring operation *stroke*, if *att1* is in *st* higher than *att2* and the following condition holds:

att1-segments cannot be inserted into target structure, but segments, which are deeper (or same level) than *att2*. The rule is also applicable, if the latter segments are needed only for aggregations.

Further examples:

st: M(A, M(B, M(C))) → M(C, M(A))

A- and *B*-segments will not be inserted by *stroke*, but *C*-values with superordinated *A*- and *B*-segments. Therefore the conditions *A::B=B,* and *B::C=C* can be smuggled. The condition *A::C=C* is then satisfied automatically, such that there is no need to introduce this condition. The selective power of both conditions is stronger, if we apply at first *B::C=C,* and then *A::B=B*.

st: M(A, M(B, M(C))) → M(B, M(C)) the condition *A::B=B* can be smuggled, because A-segments cannot be inserted. But, if we have a target scheme *M(A, C), M(A, M(C))*, then also *A*-segments without corresponding *C*-values appear in the target structure. Therefore *A::B=B* cannot be smuggled.

We remark that the *sel-before-stroke*-rule can be applied to optional values (?-collections) of a scheme in the same way as a proper collection like a set:

st: M(A, M(B)) → M(A, B?)

A::B=B cannot be smuggled.

st: M(A, B, C?, D?, M(E)) → M(B, C, D, E)

Here, *A::C=C i D=D i E=E* can be smuggled in.

If *B* is an optional value or *B* is a level deeper than *A*, then a condition *A::B=B* can also be replaced by *A::B?!=empty* or by *A::M(B)!=empty*.

The next rule is simple:

::-condition-to :::-condition

$\sigma_{n2:::cond2}(\sigma_{n1::cond1}(tab)) = \sigma_{n2:::cond2}(\sigma_{n1:::cond1}(tab))$

Here we have to presuppose only that n2 is deeper than n. If we omit at the end all empty collections, then we can omit corresponding empty collections also in a former step.

Example:

```
ext INSTIS:=INSTITUTES  #ext:extension of a tabment (by a new (complex) column)
ext E:=EMPS at FACULTY
mit ENO:: LOCATION=”Hadmersleben” i INSTIS/INSTITUTE=E/INSTITUTE
mit HOBBY::: HOBBY=”chess”
```

can be transformed to:

```
ext INSTIS:=INSTITUTES
ext E:=EMPS at FACULTY
mit ENO::: LOCATION=”Hadmersleben” i INSTIS/INSTITUTE=E/INSTITUTE
mit HOBBY::: HOBBY=”chess”
```

6 Smuggling a Forget-Operation φ

The forget-operation φ is a simple operation, which is similar to the relational projection, but which differs from projection in 3 points.

1. The (indexed) argument of forget is not a list of attributes, which are intended to remain in the resulting structure, but the list of attributes, which are to omit.
2. forget does not omit duplicates in sets.
3. forget can be used also in recursive structures

Because of point 1 and 3 forget is in some situations more expressive than a projection and also than *stroke*. Assume we have a recursive document BOOK.xml of type L(SECTION), where the type of SECTION is (TITLE, CONTENT, L(SECTION)), then
$\varphi_{CONTENT}$(BOOK) is neither equal to "π_{TITLE}(BOOK)" nor to "$\pi_{SECTION,TITLE}$(BOOK)" and not to

gib L(SECTION) &&
SECTION=(TITLE, L(SECTION)).

In the latter case a stackoverflow results. Therefore, forget is in some situations more expressive than π and also *stroke*.
Formally we will handle the second point (not eliminating duplicates) by replacing each corresponding set symbol M by a bag symbol B.

Query 5: Group the employees from Magdeburg by institute and sort by institute and name.

aus EMPS
mit LOCATION="Magdeburg"
gib M(INSTITUTE, B(NAME, SALARY))

Here, columns like HOBBY, FIRSTNAME, ... can be omitted before the restructuring is realized. This "projection" can be realized also before selection, although it is not clear in any case whether this is worth to do.

aus EMPS
mit LOCATION="Magdeburg"
forget HOBBY, FIRSTNAME, PROJECT, TIME
gib M(INSTITUTE, B(NAME, SALARY))

forget-before-stroke
$stroke_{dtd}(st) = stroke_{dtd}(\varphi_{attlist}(st))$
All elementary names from *st* (names of type *TEXT, BOOL, ...)*, which do not occur in a right side of *dtd* can be taken to *attlist*. Further, if a segment *x* of the source with all its subordinated levels cannot be inserted into target structure, then all attributes from *x* and all subordinated attributes can be added to *attlist.*
Examples:
st: M(A, M(B, M(C))) → M(A, B)
attlist = C (M(C))
st: M(A, M(B), M(C1, C2)) → M(B, M(C1))
attlist = C1, C2, A
st: M(A1, A2, M(B), M(C, M(D))) → M(A2, M(B, A1, M(D)))
attlist=(C, D) (M(C, M(D)))
st: M(A1, A2, M(B), M(C)) → M(A1, B, C)
attlist = (A1, A2, B, C) (M(A1, A2, M(B), M(C))), (Because no level can be inserted into the target structure. The resulting table is in any case empty.)

Further we can commute forget with selection:
forget-sel:
$\varphi_{attlist}(\sigma_{cond}(tab))=\sigma_{cond}(\varphi_{attlist}(tab))$
Here, it is presupposed that the condition *cond* contains no name of *attlist*.

7 Further Rules with the Extension Operation ext (ε)

Motivating example:
Query 6: Give all budgets and the totals and subtotals for all "great" faculties and institutes.

```
ext   FAC:=FACULTIES
ext   INSTI := INSTITUTES at FACBUDGET
mit   INSTITUTE:: FAC/FACULTY = INSTI/FACULTY
mit   FACBUDGET > 100000
mit   INSTITUTE:: BUDGET > 10000
gib   FACBUD, INSTIBUD, M(FACULTY, FACBUDGET, INSTIBUD, &&
      M(INSTITUTE, BUDGET)) &&
      FACBUD := sum(FACBUDGET) &&
      INSTIBUD := sum(BUDGET)
```

By the above extension of the flat table FACULTIES by the flat table INSTITUTE, a structured tabment of the following type results:
type of TABMENT = FAC
type of FAC = M(FACULTY, DEAN, FACBUDGET, INSTI)
type of INSTI = M(INSTITUTE, MANAGER, BUDGET, FACULTY)
In the query the condition FACBUDGET>100000 can be realized before extension and the remaining condition INSTITUTE::BUDGET>10000 be realized before the "join"-condition.

```
aus   INSTITUTES
mit   BUDGET > 10000
=:$temp
ext   FAC:=FACULTIES
mit   FACBUDGET > 100000
ext   INSTI := $temp at FACBUDGET
mit   INSTITUTE:: FAC/FACULTY = INSTI/FACULTY
...
```

sel-ext1
$\sigma_{cond}(\varepsilon_{assi}(tab)) = \varepsilon_{assi}(\sigma_{cond}(tab))$,
this rule holds, if all operations are applicable, and the extension does not introduce a name, which is used in the condition.

sel-ext2
$\sigma_{cond}(\varepsilon_{X:=tab2\ at\ Y}(tab1)) = \varepsilon_{X:=(\sigma_{cond}(tab2))\ at\ Y}(tab1)$,
here is presupposed that all operations are applicable, that *cond* is a ::-condition and does not contain a name from tab1.

Counter example for sel-ext1:

```
<<  L(  A,     B)::
        1      2>>
ext B:=3 at B
mit B=3
```

results in

```
        <<  L(  A,      B,      B)::
                1       2       3>>,
```
but
```
        << L(   A,      B)::
                1       2>>
        mit B=3
        ext B:=3 at B
```
results in
```
   <<L(A, B, B)::
    >>.
```

8 Related Work

We think our approach is unique in the following points:

- The select operation does not change the type of the given document; therefore we can change the order of two conditions also if they refer to different levels. If *../A[cond1]/B[cond2]* is a reasonable XPath- expression, then *../B[cond2]/A[cond1]* is in general meaningless. In the same way nowhere is considered to commute a condition from an inner FLWOR-construct with a condition of the outer one, because this is nonsens, too.
- Because of the compactness of our restructurring operation *stroke* we can generate selections or forget operations, which can fasten the query implementation.
- Nowhere, we found a rule like absorb-sel.
- The data model is powerful enough to express queries on relational data, XML, and to define (hierarchical) views.