# CALCULATING CONSTRAINTS ON RELATIONAL EXPRESSIONS

by

A. Klug

Computer Sciences Technical Report #350

April 1979

# Calculating Constraints on Relational Expressions[1]

A. Klug

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

## Abstract

A desirable feature of a database management system is the ability to support many views of the database via several user models. In order to provide this support while allowing the user to believe that his/her view and data model are the only ones, the database system must have a number of facilities. One of the most important of these is a mechanism to tell when view constraints will be satisfied given that the underlying database constraints are satisfied so that the user always sees what is expected.

This paper deals with a particular instance of this problem where the constraints are functional dependencies and the views are created through relational algebra expressions. The problem immediately reduces to the problem of calculating all valid functional dependencies (and other constraints) on a relational algebra expression over relations in the base schema. The problem is undecidable in general but we give a sound and complete algorithm when set difference is omitted from relational algebra.

[1] This research was supported in part by the National Research Council of Canada and the Mathematics Research Center of the University of Wisconsin-Madison.

## Table of Contents

## 1. Introduction

Database management systems exist to (among other things) give users facilities suited to their needs, privileges and levels of expertise. One tool for providing these facilities is the view (e.g., [ChGT]). A view can mask off parts of the database; it can rearrange the database's structure, and it can accept a customized set of operations. In general, a view can tailor the database to the needs and privileges of a user.

Ideally, users of a view should not have to be aware of the fact that they are interacting with a view of the database rather than with a "ground level" database. Now schemas generally contain various constraints in addition to object descriptions. That is, the schema not only tells the user what objects are available and in what relationships, but also what restrictions on the form and structures of the objects hold. This means that, if the user need not be aware of the level of indirection introduced by the view, there must be some built-in database mechanism which can ensure that these view constraints are always satisfied. An example will help clarify what we mean:

Suppose the underlying (or conceptual level[ANSI]) database is described by the relational schema:

Car-Part(CP#,Color,Wt)
Boat-Part(BP#,Color,Wt).

Here, CP# and BP# are keys of their respective relations.

Suppose also that some user wants to have a view according to the schema

Part(P#,Wt).

(We will specify below how this view is to be derived from the base relations.) The constraints in these schemas are that CP#, BP# and P# are keys in their respective relations.

Suppose the Part view was derived by the Sequel[ABCE] statement:

select CP#,Wt from Car-Part where color='red'.

Then the key constraint in the user view would always be satisfied since the view is just a subset of Car-Part.

Instead, suppose the Part view were equivalent to the retrieval:

(select CP#,Wt from Car-Part)
union
(select BP#,Wt from Boat-Part)

Here, the user may fail to be presented with a view in which P# is a key of the Part relation. This is because we might have a Car-Part with the same number as a Boat-Part, but the two weights may be different.

What is needed is a mechanism which can tell that the first view is acceptable while the second is not. More generally, a mechanism is needed which, given any underlying schema, any view schema, and any mapping between the two, can decide whether or not the constraints in the view schema will always be satisfied (assuming the underlying constraints are always satisfied). In terms of the framework proposed by the ANSI/X3/SPARC database study group[ANSI], we are recognizing the need for a processing function which can accept or reject mappings between a conceptual schema and an external schema. Note that correct rejection is just as important as correct acceptance. Correct acceptance means that the processor would not accept any mappings which would cause view constraint violations; correct acceptance means that the processing function would not reject any mapping which would never cause any constraint violation. Recognizing this type of correctness of mappings is an important problem and one which we study in this paper. More details on this subject may be found in [Klug].

The key constraints we used in the above example are closely related to functional dependencies. Functional dependency constraints, which state that certain attributes are in functional relationships with other attributes, are important and ubiquitous in data modelling. Any algorithms for the problem we are discussing must at least be applicable to schemas containing functional dependency constraints.

In the example, the mapping of the view to the base schema was made via the Sequel language. For purposes of the presentation, we will have all mappings expressed as relational algebra expressions[Codd]. Then calculating whether or not some functional dependency on a relation in a view is valid is the same as calculating whether or not this functional dependency is valid on the relational algebra expression corresponding to the view relation. Hence, the problem we consider in this paper is

the following:

> Given a schema s containing relations and functional dependencies and an arbitrary relational algebra expression e over relations in s, determine exactly the set of valid functional dependencies on e.

In the next section, we will give the basic definitions to be used throughout the paper. Section 3 will formulate the algorithm, and Section 4 will show it is sound. Then in Section 5 we accomplish the more difficult task of showing that the algorithm is complete. Finally, in Section 6 we draw some conclusions.

## 2. Definitions

In this section we will present the basic concepts of schema, relational algebra expression, structure, state and validity.

A relation declaration has the form 'name(degree)'; for example, R(10) defines a relation R of degree 10.

A relational algebra expression consists of relation names and the operators: projection '[.]', restriction '[.=.]', selection '[.≡.]', cross product '×', union '.⊔.' and difference '−'. The syntax is defined by the following BNFs:

[I] $e ::= \text{name} \mid e[X] \mid e[X=Y] \mid e[X\equiv V] \mid$
$e \times e \mid e \sqcup e \mid e - e$

[II] $e ::= \text{name} \mid e[X] \mid e[X=Y] \mid e[X\equiv V] \mid$
$e \times e \mid e \sqcup e$

[III] $e ::= e_1$

$e_1 ::= e_2 \mid e_1 \sqcup e_1 \mid e_1[X]$

$e_2 ::= e_3 \mid e_2[X=Y]$

$e_3 ::= e_4 \mid e_3 \times e_3$

$e_4 ::= name \mid e_4[X \equiv V]$

[IV] $e ::= e_1$

$e_1 ::= e_2 \mid e_1 \sqcup e_1 \mid e_1[X]$

$e_2 ::= e_3 \mid e_2 \times e_2$

$e_3 ::= name \mid e_3[X \equiv V]$

We will see later why we have four variants of relational algebra. Unless otherwise stated, the general form [I] is understood.

The join operator is a derived operator in this framework and is defined as follows:

R[X=Y]S is (R×S)[X=Y'], where Y'=Y+deg(R).

Intersection can also be defined from the given operators.

A functional dependency statement (FD) has the form 'expr:d*->d', where expr is a relational algebra expression, d* is a set of zero or more domain numbers, and d is a domain number. (We number relation domains rather than name them in order to facilitate joining relations with themselves.)

A schema consists of a set of relation declarations and functional dependency statements on the relations named in the schema.

A structure str for a schema s consists of a universe U of data elements, for each constant symbol i an interpretation $c_i \in$

U, and for each relation R(n) in the schema s , a set R(str) of n-tuples over U. (The constant symbol interpretations allow arbitrary data universes to be properly modelled.)

The value e(str) of a relational algebra expression e in a structure str is defined by interpreting the relational algebra operators in the usual manner:

[1] R(str) is "itself"

[2] $e[X](str) = \{t[X] : t \in e(str)\}$

[3] $e[X=Y](str) = \{t : t[X]=t[Y] \ \& \ t \in e(str)\}$

[4] $e[X \equiv V](str) = \{t : t[X]=V \ \& \ t \in e(str)\}$

[5] $(e_1 \times e_2)(str) = \{t_1 \times t_2 : t_1 \in e_1(str) \ \& \ t_2 \in e_2(str)\}$,
where $t_1 \times t_2$ denotes the concatenation of $t_1$ and $t_2$

[6] $(e_1 \sqcup e_2)(str) = \{t : t \in e_1(str) \text{ or } t \in e_2(str)\}$

[7] $(e_1 - e_2)(str) = \{t : t \in e_1(str) \ \& \ t \notin e_2(str)\}$

A functional dependency e:Z->A is true in a structure str if for every pair $t_1$, $t_2$ of tuples in e(str), if $t_1[Z]=t_2[Z]$, then $t_1[A]=t_2[A]$.

A structure str is a state with respect to a schema s if every functional dependency in s is true in str.

A functional dependency e:Z->A is valid with respect to a schema s if e:Z->A is true in every state of s.

We can now state our problem precisely:

Given a schema s and a relational algebra expression e over s, determine the valid FDs on e.

## 3. The Algorithm

This section will present rules for determining the valid constraints on a relational algebra expression. We limit our study to functional dependencies because FDs are important, familiar and well-understood. However, because the relations we look at are actually expressions involving restrictions and selections, it will be necessary to define two additional types of constraints which will reflect the structure introduced by these operators. We will call these constraints equality constraints (EQs).

Consider the following example:

s: R(2), S(2)
  R:1->2, S:1->2
e: R[2=1]S

In e the FDs 1->2 and 3->4 are valid. Because domains 2 and 3, which are the joined domains, are always equal in e, we can follow the FD 1->2 and then 3->4 to get a valid FD 1->4 in e. In order to be able to formally derive this FD, we need domain equalities (DEQs) as a constraint type:

A domain equality has the form e:X=Y. It is true in a structure str if t[X]=t[Y] for each t ∈ e(str). A DEQ e:X=Y is valid with respect to a schema s if it is true in every state of s.

Next consider the example:

s: R(3), R:1,2->3
e: R[2≡'8']

In this example, states for the expression e will always have a value '8' in domain 2. Thus, given a value for domain 1, the value of domain 2 is determined, and therefore domain 3 is determined by the FD. Hence, the FD 1->3 is valid in e. In order to be able to formally derive this FD, we need value equalities (VEQs) as a constraint type:

A value equality has the form e:X≡V. It is true in a structure str if t[X]=V for every t ∈ e(str). A VEQ e:X≡V is valid with respect to a schema s if it is true in every state of s.

We now proceed to derive inference rules for these constraints. The natural way to develop these rules is in two steps. First, we will need rules describing the derivation of new constraints from old ones on the same expression. For example, regardless of what the expression is, we can derive 1->3 from 1->2 and 2->3. The second group of rules will tell how to derive constraints on an expression from the derivable constraints on its subexpressions. For example, from R:2->3 we can derive 1->2 on the projection R[2,3].

We will first discuss constraints on single expressions.

When only FDs are considered, complete sets of rules already exist(e.g. [Arms]). The following three rules are one such set:

[A1] $\vdash$ X->X (reflexivity)
[A2] X->Z $\vdash$ X⊔Y->Z (augmentation)
[A3] X->Y, Y⊔Z->W $\vdash$ X⊔Z->W (pseudotransitivity)

(where '⊢' means "derives")

See [Bern] for examples.

Now suppose that EQs are also involved.

The set of rules must include ones reflecting the basic properties of equality: Everything equals itself; hence, X=X will always be derivable. The constraint X=Y implies the constraint Y=X. Also X=Y and Y=Z imply X=Z.

VEQs also interact with DEQs: If X=Y and Y≡V are constraints, then the X-domains are constant and also equal V, i.e., X≡V is a valid constraint. If X≡V and Y≡V are derivable, then so is X=Y.

EQs also interact with FDs. If X=Y holds and X appears on the left- or right-hand-side of an FD, then the FD with X replaced by Y should also hold. If X≡V holds and X appears on the left-hand-side of an FD, then since the X-domain is constant, we can drop X from the FD.

From this discussion we see that the following rules are valid:

```
[i]     Z->A, X=Y  ⊢  (Z-X)⊔Y->A
[ii]    Z->A, A=B  ⊢  Z->B
[iii]   Z->A, X≡V  ⊢  (Z-X)->A
```

Instead, we will use the following two rules which are equivalent to [i] - [iii]:

```
[a]   X=Y  ⊢  X->Y
[b]   X≡V  ⊢  ∅->X
```

With these two rules, we can drop the reflexive rule [A2] above, and we can also derive rules [i], [ii] and [iii] using transitivity.

There is another kind of interaction which is more difficult to detect. It is a problem which could not occur if we only considered constraints on a single relation, and it is to this problem that much of this paper is devoted. To illustrate, consider the following simple example:

```
s: R(2), R:1->2
e: R[1=1]R
```

For this expression, the derivable constraints will include the FDs 1->2 and 3->4, and also the EQ 1=3. But the two FDs are really the "same" since they arise from the same FD on the same base relation. Because their domains are equated by the EQ, their ranges must also be equal, i.e., the EQ 2=4 must also hold.

In more general terms, whenever one or more relations appear as components on both sides of a join, some of the FDs on the join might actually be the "same".

The following is a slightly more complicated example:

```
s: R(3), S(3), T(3)
   R:1->2
e: (R⊓S)[1,2][1=4](S[2=1]R)
```

In the expression e, the (same) FD on R appears as 1->2 and as 1->7. A similar effect occurs with unions which will be illustrated shortly.

To tell when two FDs have the same origins, they need to be given labels or identifiers, so that when two labels are equal (or "essentially" equal — as explained below), the two FDs are one and the same. Note that the "origins" of an FD in a relational expression will be tree-like. The tree structure is not only due to the axioms for FDs, especially pseudotransitivity, but also to the tree-structure of the relational expressions themselves. The FD identifiers are essentially derivation trees.

The next examples will informally describe what FD identifiers look like.

```
s: R(4), S(2)
   R:1->2, R:2,3->4
   S:1->2
e: R[4=1]S
```

The FD 1,3->6 will be derivable in e by rules we will present shortly. To see what the label of this FD is, we will make an informal derivation:

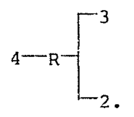
In the join R[4=1]S, the domain names of S have been shifted by four units (the length of R). So the FD S:1->2 in the join can be represented as:

6—2—S—1—5,

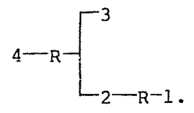
where the arcs 6— and —5 represent the relabelling. (In this and in the following examples, the roots of trees will be either at the left or at the top.) The EQ constraint 4=5 (which gives an FD 4->5) is valid in the join R[4=1]S because of the join-condition, and so we may add the arc 5—4 to the above FD label, obtaining:

6—2—S—1—5—4

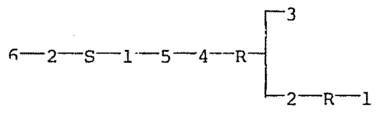
as a label for the FD 4->6 which is valid in the join. The FD R:2,3->4 is not renamed, so its label in the join is the tree:



The FD R:1->2 combines with R:2,3->4 by transitivity yielding 1,3->4, whose label is



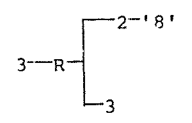
We again apply transitivity to derive in the join the FD 1,3->6. Thus the FD in the first example can be represented by the following tree:



When selections occur, there are special nodes which appear in FD labels. For example:

```
s: R(3),  R:1,2->3
e: R[2≡'8']
```

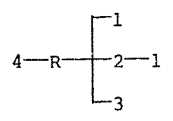
As we have mentioned before, the FD 1->3 is valid in this expression. Its label should be the following:



The next example illustrates the case when the label may have more than one leaf with the same domain name:

s: R(4), S(2)
  R:1,2,3->4
  S:1->2
e: (R[1=2])[1=2]S

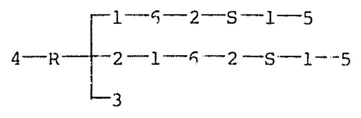
In e, we will have the FD 1,3->4 which results from the FD R:1,2,3->4 and the restriction on R in e. Its label is the following:



The EQ from the join-condition, 1=6, will combine with the FD S:1->2, which in the join is 5->6, to give the FD 5->1 with the label:

1—6—2—S—1—5.

Composing the FDs 1,3->4 and 5->1 in e to produce 3,5->4, will correspond to merging the previous two trees at the '1' nodes. Since there are two terminal '1' nodes in the tree for 1,3->4, two copies of the tree for 5->1 must be joined:



These examples suggest the general approach: An FD Z->A on a base relation R is represented by itself: a tree whose root is A, whose leaves are the domains in Z and with an interior node labelled R. Whenever we derive a new FD by pseudotransitivity, we "graft" the two corresponding labels together — copies of the root of one to the matching leaves of the other. Whenever a new FD is derived from an old one by using a DEQ, we can add a new label to the appropriate leaves and/or to the root. The shrinking of the left-hand-side of an FD by application of a VEQ can be reflected in the tree label by adding to the appropriate leaves a terminal node labelled with the VEQ's value.

The purpose of the FD identifiers just described is to provide a means to tell when two FDs are the same. However, the labels need not be identical for the FDs to be the same. Being able to tell when two identifiers or derivation trees represent the same FD is an important part of our algorithm. Before presenting the rules for the equivalence of FD identifiers, we will give some examples of FD identifiers being the same.

First, take the following example:

s: R(2), S(2)
  R:1->2
e: R[1=1](S[1=1]R).

The FDs 1->2 and 1->6 in e have by the construction the respective labels:

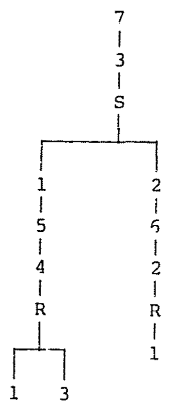
2—R—1 and 6—4—2—R—1—3—5—3—1.

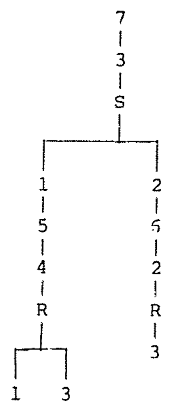
The two FDs are the same, although their identifiers are not. However, the identifiers differ only in the "renaming" of the nodes. This example generalizes to the rule that if any two FDs have labels which differ only in "renaming segments", then they are the same FD.

There are two other ways in which FD identifiers may differ while still describing the same functional dependency. We will give two examples to illustrate:

s: R(4), R:1,3->4, R:1->2, R:3->2
   S(3), S:1,2->3
e: R[2,4=2,1]S

In e there is the FD 1,3->7 which can be represented by either of the trees:



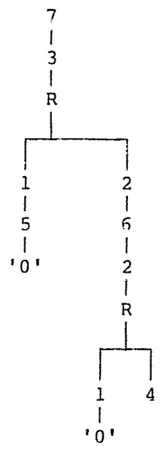


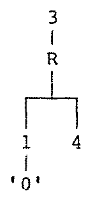
If e occurred twice within a larger relation, we might have derived the FD in the two occurrences with the two different identifiers. We would have to know that these identifiers represent the same FD. (These identifiers represent the same FD because the part R:1->2 will determine the same R-tuple as R:3->2 due to the presence of the FD R:1,3->4.)

The second example is:

s: R(4), R:1,2->3, R:1,4->2, R:1,4->3
e: (R[2=2]R)[1,5≡'0,0']

In e, the FDs 4->7 and 4->3 can be associated, respectively, with the trees:





We must be able to "collapse" the identifier on the left and recognize it as equalling the one on the right. (Composing FDs across joins of the same relation is the same as composing FDs within a single relation.)

Finally, we note that a single FD may need to have several identifiers associated with it. If an expression e is a union $e_1 \sqcup e_2$, and an FD Z->A is valid in e, then we will need to associate one or more identifiers for Z->A from $e_1$ and one or more from $e_2$. A more detailed example will be given in Section 4.

When the FD e:Z->A has the identifier id associated with it, we will write the association as e:id:Z->A or just id:Z->A and call this a <u>labelled</u> <u>FD</u>. When Z->A has several identifiers associated with it[2], we will write $\{id_i:i\in S\}$:Z->A or just $\{id_i\}$:Z->A.

The first part of the algorithm for calculating constraints on expressions consists of a procedure for generating a "free structure"[3]. With respect to two FD identifiers, the free structure will show the equivalence or non-equivalence of the identifiers.

First we define a function "gentab" which generates (as side-effects) tables of tuples with formal values. Then a function "infervals" equates values according to the FDs present. If the appropriate domains of the two tuples corresponding to the roots of the two identifiers $id_1$, $id_2$ are equated then the value of the predicate $Eqv(id_1,id_2)$ is true, otherwise it is false.

2 The need for several identifiers will be explained shortly.
3 This concept is similar to the notion of Henkin interpretation[Henk].

```
function Eqv($id_1$,$id_2$ : FD-ident) : boolean;
begin
 fstr := empty;  /* start with empty state */
 gentab($id_1$);
 gentab($id_2$);
 infervals;
 if $id_1$(fstr;V) = $id_2$(fstr;V)
   then Eqv := true
   else Eqv := false
end;

function gentab(id : FD-ident) : formal-value;
begin
 if id is a domain leaf 'n' then gentab := '$v_n$ /*n-th formal value*/
 else if id is a value leaf 'v' then gentab := v
 else if succ(id) is a domain node then gentab := gentab(succ(id))
 else /* succ(id) is a relation name R; root(id) is a domain A, and
         the children of node R are $id_1,\dots,id_n$ whose roots are
         labelled $Z_1,\dots,Z_n$ */
     begin
       add new tuple t to R(fstr);
       for i:=1 to n do t[$Z_i$] := gentab($id_i$);
       for Y not in {$Z_1,\dots,Z_n$} do t[Y] := new_formal_value;
       gentab := t[A]
     end
end;
```

```
procedure infervals;
begin
  changes := true;
  while(changes) do
    begin
      changes := false;
      for each R in schema do
        for each $t_1$, $t_2$ in R(fstr) do
          if FD R:Z->A and $t_1$[Z]=$t_2$[Z] and $t_1$[A]$\neq t_2$[A]
            then begin changes := true; equateval($t_1$[A],$t_2$[A]) end
    end
end;

procedure equateval(x,y : formal-value);
begin
  for each R in schema do
    for each t in R(fstr) do
      for each domain Z do if t[Z]=x then t[Z] := y
end.
```

The predicate Eqv constitutes one part of our algorithm for determining valid constraints on relational algebra expressions. The next definition specifies the inference rules which we have already informally discussed for deriving constraints on a single expression. This constitutes the second part of the algorithm:

Let s be a schema; let e be an expression over s. The <u>rules for deriving</u> new EQs and labelled FDs from old ones on e are as follows[4]:

[1] $X=Y \vdash Y=X$

[2] $X=Y,\ Y=Z \vdash X=Z$

[3] $X=Y,\ Y\equiv V \vdash X\equiv V$

[4] $X\equiv V,\ Y\equiv V \vdash X=Y$

[5] $\vdash X=X$

[6] $X=Y \vdash id{:}X\text{->}Y$,
where id is $X\text{->}Y$

[7] $X\equiv V \vdash \emptyset\text{->}X$
where id is $\text{'V'->}X$

[8] $\{id_{1i}\}{:}Z\text{->}A_1,\ \{id_{2j}\}{:}Z\text{->}A_2,\ \forall i,j\ Eqv(id_{1i})=Eqv(id_{2j}) \vdash A_1=A_2$

[9] $\{id_{1i}\}{:}Z\text{->}A,\ \{id_{2j}\}{:}A\sqcup X\text{->}B \vdash \{id_{3ij}\}{:}Z\sqcup X\text{->}B$,
where $id_{3ij}$ is obtained from $id_{2j}$ by merging a copy of $id_{1i}$ to every leaf of $id_{2j}$ labelled A.

Any expression will always have just a finite number of derivable constraints. (There are only a finite number of EQs and FDs on an expression altogether.) The process of generating all derivable constraints with these rules from a given set of constraints will be called taking the <u>closure</u> and will be denoted 'Cl'.

Two additional conditions are used in the closure construction:

[a] If an FD is already present, it will not be added with another identifier. This will prevent non-termination from

[4] For technical reasons which will become apparent shortly, we have not explicitly included the augmentation rule. However, our notion of completeness will be appropriately modified, and there will be no loss of generality.

repeated application of rules [9] with FDs from rule [8].

[b] Any FDs Z->A such that an FD $Z_1$->A is present with $Z_1 \subseteq Z$, $Z_1 \neq Z$, are removed.

We have specified inference rules for constraints on one expression from given constraints. Now we want to give rules for generating constraints on expressions from constraints derived on subexpressions. First we will informally discuss how this should be done, and then we will present the formal rules:

For a base relation R, we take the FDs in the schema belonging to R and label them by themselves. Then the closure operator is applied.

For projections, there are two special points. First, in order for an FD or an EQ to "survive" a projection, all of the referenced domains must be included in the projection. Second, we must rename the domains by the order of their appearance in the projection list. Take the following example:

s: R(10), R:2,4,6->8
$e_1$: R[2,4,6,8]
$e_2$: R[1,2,3,4,6,8]
$e_3$: R[8,7,6,2,4]
$e_4$: R[2,6,7,8]

In the projection $e_1$, the FD appears as 1,2,3->4. In $e_2$, the FD is 2,4,5->6. In $e_3$, it is 2,4,5->1, and in $e_4$, there is no FD. If C is the set of derivable constraints for some expression e, then to get the set of derivable constraints for a projection e[X], we need to find all the constraints in C which survive the

projection, but we do not need to apply the closure operator. The reason is that C is already closed, and if one or more constraints in C survive the projection and also combine according to one of the rules [1]-[9] defining closure, then the resulting constraint will also survive projection because every domain in the result appears in at least one of the original constraints. Notationally, the condition for Z->A to be derivable on e[X] is that X[Z]->X[A] be derivable on e. The notation X[Z] is like array subscripting: An attribute A is in X[Z] if and only if there is some $j \in Z$ such that A is in the j-th position of X. For example, if X is 2,8,4,6 and Z is {2,4}, then X[Z] is the set {6,8}.

In cross products, renaming also occurs. Given relations R(m) and S(n), the domains of S in the cross product R×S have been "shifted" by the length m of R. That is, they number m+1 through m+n. All constraints of S are valid in the cross product, but they have also been renamed accordingly.

For a restriction e[X=Y], we add the constraint X=Y to the ones already holding in e and take the closure. For a selection e[X≡V], we similarly add X≡V to the constraints of e and take the closure.

For a union $e_1 \sqcup e_2$, a constraint basically must hold in both components in order to hold in $e_1 \sqcup e_2$; any constraint valid on only one component can be violated by tuples in the other component, and these tuples will appear in the union. To get the EQs valid in $e_1 \sqcup e_2$, we simply take the intersection of the sets of valid EQs for $e_1$ and $e_2$. To calculate the valid FDs, the FD identifiers must be considered; i.e., "intersection" must make sure that the FD identifiers are Eqv. The next example will illustrate this:

s: R(2), S(2)
R:1->2, S:1->2

$e_1$: R$\sqcup$S

$e_2$: R[2≡'2'] $\sqcup$ R[2≡'5']

In $e_1$, the FD 1->2 does not hold. For example, in the state st, where R(st) = {(0,0)} and S(st) = {(0,1)}, the FDs on the base relations are true, but (R$\sqcup$S)(st) = {(0,0),(0,1)}, and this violates the FD 1->2.

Now consider the expression $e_2$. Clearly the FD 1->2 will be valid in this expression (since, e.g., R[2='2']$\sqcup$R[2='5'] $\subseteq$ R). This is because 1->2 is the "same" FD in each component. In general, for an FD id:Z->A to appear in $e_1 \sqcup e_2$, it must be valid in both $e_1$ and $e_2$ and have the "same" identifier in both components.

If the two FDs are Eqv, then the FD is in the union. We must, however, attach both identifiers to the FD. The following example will illustrate why.

```
s:   R(7)  R:1,2->4  R:2,3->5  R:1,2->6  R:1,3->6
           R:4,5,6->7
e1:  (((R[4=4]R)[5=5]R)[6=6]R)[7+1=21+1,7+2=14+2,14+3=21+3]
     [7+1,7+2,1+3,0+7]
e2:  ((R[4=4]R)[5=5]R)[6=6]R)[7+2=14+2,7+2=21+2,14+3=21+3]
     [7+1,7+2,14+3,0+7]
```

$e_3$: R[1,2,3,7]

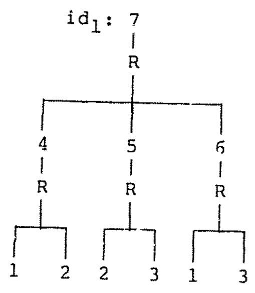
$e_4$: $e_1 \sqcup e_3$

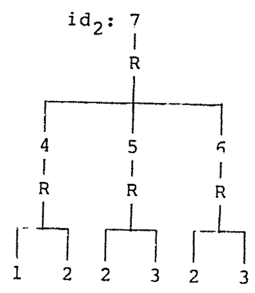
$e_5$: $e_2 \sqcup e_3$

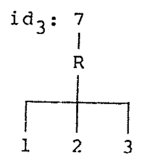
$e_6$: $e_1 \sqcup e_2$

$e_7$: $e_4 \sqcup e_2 = (e_1 \sqcup e_3) \sqcup e_2$

In $e_1$ and $e_2$ we have written each domain in the restrictions as a displacement plus a domain of R, e.g., 14+2 denotes the second domain of the third copy of R in the join. In each of $e_1$, $e_2$ and $e_3$ the FD 1,2,3->4 is derivable and will have the following respective identifiers (with relabelling nodes removed):







And we have Eqv($id_1$,$id_2$), Eqv($id_2$,$id_3$) but not Eqv($id_1$ $id_2$). However, on $e_6$ this FD is not valid. If we had only retained $id_3$ for the FD in $e_4$, then we would have incorrectly derived the FD on $e_6$. We could conceivably have a procedure to decide which identifier should be retained, but it is simpler just to keep them all.

The constraints derivable on a set difference will be the constraints derivable on the first component. There will be at least that many derivable constraints because the difference is a subset of the first component. More will be said about set difference later.

This discussion is formalized by the following definition:

Let s be a schema and e an expression over s. The set Drv(e) of <u>derivable constraints</u> on e is defined by the following rules which use induction on the number of operations in e and in which "Cl" denotes the closure operator:

[1] Drv(R) — Take the closure of R's FDs in the schema and have each FD R:Z->A labelled by the tree consisting of a root segment R—A (A is the root) and an arc d— into node R for each d ∈ Z.

[2] Drv(e[X]) — Take all DEQs Y=Z where X[Y]=X[Z] is in Drv(e), all VEQs Y≡V where X[Y]≡V is in Drv(e), and all FDs $id_2$:Z->A such that $id_1$:X[Z]->X[A] is in Drv(e), where $id_2$ is obtained from $id_1$ by attaching to each leaf X[d] of $id_1$ the arc d—, and to the root X[A] of $id_1$ the arc —A.

[3] Drv(e[X=Y]) — Add X=Y to Drv(e) and take the closure.

[4] Drv(e[X≡V]) — Add X≡V to Drv(e) and take the closure.

[5] Drv($e_1$×$e_2$) — Rename the constraints in Drv($e_2$) according to the degree of $e_1$, i.e., A DEQ X=Y becomes X+k=Y+k (k=degree($e_1$)); a VEQ becomes X+k≡V, and an FD $id_1$:Z->A becomes $id_2$:Z+k->A+k, where $id_2$ is obtained from $id_1$ by adding an arc d+k— to each leaf labelled d, and an arc —A+k to the root. Then add Drv($e_2$) renamed to Drv($e_1$) and take the closure.

[6] Drv($e_1 \sqcup e_2$) — An EQ X=Y or X≡V is in Drv($e_1 \sqcup e_2$) if it is in both Drv($e_1$) and Drv($e_2$). If {$id_{1i}$}:Z->A is in Drv($e_1$) and {$id_{2j}$}:Z->A is in Drv($e_2$) and ∀i,j Eqv($id_{1i}$,$id_{2j}$), then {$id_{1i}$,$id_{2j}$}:Z->A is Drv($e_1 \sqcup e_2$).

[7] Drv($e_1$−$e_2$) — Use Drv($e_1$).

We will say that e is <u>consistent</u> (with respect to s) if for no domain X and for no distinct values $V_1$, $V_2$ are both X≡$V_1$ and X≡$V_2$ members of Drv(e). It is meaningless to try to derive constraints on expressions which are not consistent. □

Our algorithm now consists of the functions Drv, Cl and Eqv. With these rules, a solution may be proposed to Problem 2 posed earlier:

<u>Proposition</u>. Given a schema s and an expression e, an EQ e:X=Y or e:X≡V is valid if and only if it is in Drv(e); an FD e:Z->A is valid if and only if for some subset $Z_1 \subseteq Z$, e:$Z_1$->A is in Drv(e)[5].

The verification of this proposition has two parts. First, are the rules sound? That is, are any invalid FDs or EQs generated? Second, are the rules complete? Do they find all of

[5] We have left out the augmentation rule, but this notion of soundness/completeness means that the rule can always be applied as the very last step in a derivation.

the valid FDs and EQs? The question of soundness is considered in the next section.

## 4. Soundness of the Algorithm

The formal development of the operator Drv was in three parts: First, Eqv was defined; then we defined the closure rules using Eqv, and finally the derivable constraints Drv in terms of closure and Eqv. To prove the soundness of the rules, we first prove some properties of Eqv and Cl and then properties of Drv.

Functional dependencies, as their name implies, are closely related to ordinary mathematical functions. If, say, in R(n) there is the FD Z->A, then given any state st, the projection R[Z,A](st) defines a partial function $U^k$->U, where k is the number of domains in Z (and U is the data universe of the state). Similarly, given any state, FD identifiers define partial functions. It is convenient to take as the domain of these latter functions the set of infinite sequences of elements of the universe. In this way, all functions will have the same domain. The function we associate with an identifier will then actually depend only on the positions in the sequence corresponding to the integer labels of the leaves of id. That is, if, for example, the leaves of id are 2,5,7, then in an argument $(x_1,x_2,\ldots,x_n)$, only values $x_2$, $x_5$ and $x_7$ will affect the value of the function. Given a state st, we will denote the function determined by an identifier id by id(st) or id(st; ). We will first define the functions determined by identifiers,

and then we will compare these functions with the FDs to which the FD identifiers are attached.

The _partial function_ id(st) whose domain is the set of sequences of universe elements is defined by the following Algol-like procedure, where x is a sequence of domain values:

```
id(st;x) =
    if id is a domain leaf i then $x_i$
    else if id is a value leaf 'v' then v
(*) else if succ(id) is an integer node then succ(id)(st;x)
    else  /* succ(id) is a selation name R; root(id) is a
              domain A, and the set of children of R is
              {$id_1,\ldots,id_n$} whose roots are labelled $Z_1,\ldots,Z_n$ */
         if $id_1$(st;x),...,$id_n$(st;x) are defined
            then begin
                   $v_1$<-$id_1$(st;x);
                   ...
                   $v_n$<-$id_n$(st;x);
                   if R[$Z_1,\ldots,Z_n \equiv v_1,\ldots,v_n$][A](st) = {a}
                   /* i.e., if there is a tuple in R(st) whose
                      Z-values equal the v's */
                      then a  /* i.e., return the unique A-value */
                      else undefined
                 end
            else undefined                ☐
```

Now identifiers were not associated with FDs in a haphazard fashion. They were chosen so that the function determined by the identifier essentially equals the function determined by the

FD. To decide, as in rule [8] of the definition of closure, if $A_1=A_2$ should be inferred from $id_1$:Z->$A_1$ and $id_2$:Z->$A_2$ is then a question of deciding if $id_1$(st) = $id_2$(st) for all states st, i.e., if the functions determined by the identifiers are equal. We are given as an hypothesis of rule [8] in the definition of closure the truth of Eqv($id_1$,$id_2$). Hence, we need to know that the truth of Eqv means equality of the identifier functions. This is proved in the following theorem:

*Theorem 1*. Let s be a schema and suppose $id_1$ and $id_2$ are associated with FDs derived on expressions over s. Then Eqv($id_1$,$id_2$) true implies that for all states st and value sequences x, if $id_1$(st;x) and $id_2$(st;x) are defined, then they are equal.

Proof. The proof may be found in the appendix. □

To prove completeness, it will be necessary to know that the converse of the above theorem will hold. This is proved in the next theorem.

*Theorem 2*. Let s be a schema and let $id_1$ and $id_2$ be associated with FDs derived on expressions over s. Then Eqv($id_1$,$id_2$) false implies that there exists a state st and a value sequence x such that $id_1$(st;x) and $id_2$(st;x) are defined and unequal.

Proof. This is immediate from the definition of Eqv: Take st to be the "free state" fstr and the value sequence x to be V. □

We have claimed that the function determined by an FD Z->A defined on an expression e is essentially the same as the function determined by the associated identifier. The precise

meaning of "essentially" is the following: Suppose a state st and a sequence x is given. The values of x which are relevant to the FD are x[Z]. The application of the function to the input x corresponds to the selection expression e[Z≡x[Z]]. To extract the A-value, the projection e[Z≡x[Z]][A] is used. If the function is defined at st and x, there is one element in e[Z≡x[Z]][A](st), otherwise it is empty. The "essential" equality is then a ∈ e[Z≡x[Z]][A](st) if and only if id(st;x) is defined and equal to a[6]. We do not prove this equality separately but together with the soundness statement itself below.

With the above constructions, we can prove the soundness of the closure operator. We hypothesize two properties of a set S of constraints. One property says that all elements of S are valid constraints. The other property is a statement that FD identifiers "agree" with the FDs as discussed above. The theorem then states that the closure operator preserves these two properties. Of course, we are primarily interested in the preservation of validity. However, the second property is needed during the induction steps and for the completeness theorem.

*Theorem 3*. Let s be a schema and e an expression over s, and consider the following two statements:

(i) every constraint in the set is valid;

(ii) for every FD {$id_i$}:Z->A in the set, for every state and

[6] There is no obvious way to adjust FD identifiers for an augmentation rule. If the identifier for, say, $Z \sqcup X$->A is the same as for Z->A, then the above relationship will only hold in the "only if" direction.

every sequence x of universe elements, a ∈ e[Z≡x[Z]][A](st) if and only if a = $id_i$(st;x) for some $id_i$.

Let S be a set of EQs and labelled FDs defined on an expression e such that (i) and (ii) hold on S. Then (i) and (ii) hold on Cl(S).

Proof. The proof uses induction on the length of the derivation and can be found in the appendix. □

With this theorem, it is now easy to show the soundness of Drv: (We again must prove the extra clause (ii) as in Theorem 3.)

<u>Theorem</u> <u>4</u>. Let s be a schema, and let e be an expression over s. Then statements (i) and (ii) of Theorem 3 hold on the set Drv(e).

Proof. This proof uses induction on the number of relational algebra operators in e and can be found in the appendix. □

<u>Corollary</u>. Let s be a schema and e be an expression over s. Then every EQ in Drv(e) is valid, and every FD e:Z->A such that $Z_1$->A is in Drv(e) for some $Z_1 \subseteq Z$ is valid.

## <u>5</u>. <u>Completeness</u> <u>of</u> <u>the</u> <u>Algorithm</u>

We now know that Drv does not generate any invalid constraints. The next step in a solution to the problem of calculating constraints on relational algebra expressions is to prove completeness: Does Drv generate all of the valid constraints?

As we have defined Drv, the answer is no, Drv does not generate all valid constraints. The following example illustrates this:

s1: R(2), S(2), R:1->2

e: (R⊔S)-(S-R)

Drv(R⊔S) = ∅

Drv(e) = ∅

Clearly, the expression (R⊔S)-(S-R) is equivalent to R, and therefore the FD 1->2 is valid in (R⊔S)-(S-R). However, the rules calculate no FDs on e. It seems, then, that the constraint formula for set difference must have some knowledge of when formulas are equivalent so that it can recognize, at least as far as FDs and EQs are concerned, (R⊔S)-(S-R) as being equivalent to R. It turns out, however, that this is impossible: Functional dependencies are sufficient to capture the notion of equivalence of two expressions, and this problem is undecidable.

The following definition specifies what we mean by equivalence, and the next theorem shows the relationship between FDs and the equivalence property:

Let s be a schema, and let $e_1$ and $e_2$ be expressions over s of degree n. Then $e_1$ and $e_2$ are <u>equivalent</u> (<u>with</u> <u>respect</u> <u>to</u> s), written $e_1 \equiv e_2$, if for every state st, $e_1(st) \equiv e_2(st)$. ||

<u>Theorem</u> <u>5</u>. Let s be a schema, and let R be an n-ary relation over s such that for every domain X of R, the FD R:∅->X is in s. If $e_1$ and $e_2$ are expressions of degree n which do not contain R, then $e_1 \equiv e_2$ if and only if every possible FD ∅->X is

valid in the expression $E = R \sqcup (e_1-e_2) \sqcup (e_2-e_1)$.

Proof. If $e_1 \equiv e_2$, then it is immediate that for every state st, E(st) = R(st), and so every FD $\emptyset$->X is valid in E. Now suppose $e_1 \not\equiv e_2$. This means that there is some state st and some tuple t such that either $t \in e_1(st)$ and $t \notin e_2(st)$, or $t \in e_2(st)$ and $t \notin e_1(st)$. In either case, $t \in ((e_1-e_2) \sqcup (e_2-e_1))(st)$. Let $t' \neq t$ be any tuple not in $(e_1 \sqcup e_2)(st)$. (If $(e_1 \sqcup e_2)(st)=U^n$, we can extend the universe U by adding one more element u, and we can then construct tuple t' from u.) Because R does not appear in either $e_1$ or $e_2$, we may assume that R(st)={t'}. Then E(st) contains both t and t' and therefore some FD $\emptyset$->X is not true in E(st). []

It is also possible to prove this theorem without using the new relation R, that is, by only using relations appearing in $e_1$ and $e_2$.

This theorem will be enough to show the impossibility of calculating all valid FDs on arbitrary relational algebra expressions. This is because the problem of determining the equivalence of two relational algebra expressions is undecidable as the following theorem states:

Theorem 6. Let s be a schema. The following problem is undecidable:

Given arbitrary relational expressions $e_1$ and $e_2$ over s of degree n, determine if $e_1 \equiv e_2$.

Proof. The proof is based on a similar theorem by M. Solomon[Solo] and can be found in the appendix. []

With this undecidability result, we can prove the undecidability of calculating constraints on expressions:

Theorem 7. The problem of determining all valid constraints in a given relational expression of syntax (I) over a given schema is undecidable.

Proof. If there were a procedure for calculating all constraints valid in a relational algebra expression, then by Theorem 5 there would be a procedure for determining the equivalence of relational algebra expressions. But this latter problem is undecidable by Theorem 6. []

Given this undecidability result, we next ask what changes can be made to the mapping language which will allow a decidable complete set of derivation rules.

There are two easy-to-specify restrictions which can be placed on the relational algebra syntax which will eliminate the above incompleteness. One restriction is to allow relation names to appear at most once in an expression. Then expressions such as $(R \sqcup S)-S$ will not be allowed, and the constraints valid on a set difference will always be exactly the ones valid on the first component of the difference.

Another restriction is to disallow set difference as a relational algebra operator. Without set difference, we cannot perform the reduction of Theorem 4, and the complete calculation of constraints on relational algebra expressions not using set difference will be possible. This second restriction is the one we shall investigate.

Let us consider, then, the language of syntax (II) in which set difference has been omitted. Does completeness hold now? With this language, the answer is still no, not all FDs and EQs are detected.

Before we give a counterexample (pointed out by E.F. Codd), we will briefly discuss certain behavior of joins of projections. For this discussion only, we will use different notation which will allow examples of such joins to be written more succinctly. The example relation R will have domains named A, B and C. A join, denoted by '⋈' will be understood to be an equi-join on the like-named domains of its components, and it will eliminate one of the joined domains.

Ordinarily, a join of projections such as R[AC]⋈R[BC] will contain all the tuples in R, plus extra tuples, i.e., R C R[AC]⋈R[BC], but R ≠ R[AC]⋈R[BC]. Take the following example:

R

| A | B | C |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 2 |

R[AC]

| A | C |
|---|---|
| 0 | 2 |
| 3 | 2 |

R[BC]

| B | C |
|---|---|
| 1 | 2 |
| 4 | 2 |

R[AC]⋈R[BC]

| A | B | C |
|---|---|---|
| 0 | 1 | 2 |
| 0 | 4 | 2 |
| 3 | 1 | 2 |
| 3 | 4 | 2 |

If enough structure is added to R, namely in the form of FDs, then no extra tuples will appear in the join; that is, the join will be "non-loss". In our example, the FD C->B is sufficient to cause R to have a non-loss join. (The above example violates the FD C->B.) General necessary and sufficient conditions for a relation to have a non-loss join are given in [AhBU].

Now let us return to constructing a counterexample to the

completeness problem. We use the relation R as above (returning to the usual notation) and add a second FD 1,2->3 to the FD 3->2 already mentioned. We have:

s: R(3), R:1,2->3, R:3->2

Drv(R[1,3]) = ∅

Drv(R[2,3]) = {2->1, 1->1, 2->2, 1,2->1, 1,2->2}

Drv(R[1,3][2=2]R[2,3]) = {4->3, 2->3, 1->1, ...}

Drv((R[1,3][2=2]R[2,3])[1,3,4] = {3->2, 1->1, ...}

(The ellipsis represents the trivial FDs arising from the reflexive rule.) Yet, we know that for any state st, that

R(st) = (R[1,3][2=2]R[2,3])[1,3,4](st),

i.e., that R has a non-loss join. (The expression on the right corresponds to R[AC]⋈R[BC].) Therefore, the FD 1,2->3 is valid in the expression, but it is not calculated by Drv.

Intuitively, it is not hard to see why this happened. The FD 1,2->3 was lost through a projection, but the later join was non-loss and so the FD reappeared. This problem can be eliminated by restricting the mapping language. The problem arose by doing projections and then undoing them by joins. A restricted mapping language which does all joins (formally, cross products) before any projections will be just as powerful as the unrestricted language and will not cause Drv to lose any FDs as happened above. This is what the third version of the relational algebra syntax does. It first allows cross products on base relations or selections. Then restrictions are allowed, and lastly, union and projections are allowed. (The reason selections are moved inward to base relations will be given later.)

In order to show that syntax (III) is just as powerful as syntax (II), we show that every syntax-(II)-expression can be transformed to an equivalent syntax-(III)-expression. The transformations move cross products inward until their operands are either base relations, selections or other cross products. They also move selections in to base relations. The following theorem accomplishes this:

<u>Theorem</u> 8. Let s be a schema, and let $e_1$, $e_2$ and $e_3$ be expressions over s. Then the following equivalences are valid:

[1a] $e_1[X]\times e_2 \equiv (e_1\times e_2)[X,D2]$,
where D2 is $\deg(e_1)+1, \deg(e_1)+2, \ldots, \deg(e_1)+\deg(e_2)$

[1b] $e_1\times(e_2[X]) \equiv (e_1\times e_2)[D1,X']$, where $X'=X+\deg(e_1)$ and D1 is $1,2,\ldots,\deg(e_1)$

[2a] $e_1[X=Y]\times e_2 \equiv (e_1\times e_2)[X=Y]$

[2b] $e_1\times(e_2[X=Y]) \equiv (e_1\times e_2)[X'=Y']$,
where $X'=X+\deg(e_1)$ and $Y'=Y+\deg(e_1)$

[3a] $(e_1\sqcup e_2)\times e_3 \equiv (e_1\times e_3)\sqcup(e_2\times e_3)$

[3b] $e_1\times(e_2\sqcup e_3) \equiv (e_1\times e_2)\sqcup(e_1\times e_3)$

[4] $e[X][Y\equiv V] \equiv e[Y'\equiv V][X]$, where $X[Y]=Y'$

[5] $e[X=Y][Z\equiv V] \equiv e[Z\equiv V][X=Y]$

[6] $(e_1\sqcup e_2)[X\equiv V] \equiv e_1[X\equiv V]\sqcup e_2[X\equiv V]$

[7a] $(e_1\times e_2)[X\equiv V] \equiv e_1[X\equiv V]\times e_2$, where $X\leq\deg(e_1)$

[7b] $(e_1\times e_2)[X\equiv V] \equiv e_1\times(e_2[X'\equiv V])$, where $X>\deg(e_1)$ and $X'=X-\deg(e_1)$

[8] $e[X][Y=Z] \equiv e[Y'=Z'][X]$, where $Y'=X[Y]$ and $Z'=X[Z]$

[9] $(e_1\sqcup e_2)[X=Y] \equiv e_1[X=Y]\sqcup e_2[X=Y]$

Proof. The proof is straightforward and is omitted. ||

The equivalences of Theorem 8 can be viewed as transformation rules: Given an expression e according to syntax (II), match some part of e with the left-hand-side of an equivalence of Theorem 8 to produce a new expression e' from the right-hand-side of the equivalence. Repeat the process with e'. Continue as long as possible. The result will be an expression according to syntax (III) which is equivalent to e. We record this result in the following theorem.

<u>Theorem</u> 9. Every expression of syntax (II) is equivalent to an expression of syntax (III). ||

By this theorem, we are justified in assuming that all expressions conform to syntax III.

Before we continue with the completeness problem, let us review the example which was used to show that Drv was not complete for syntax-(II)-expressions. Recall that we had a relation R(3) with FDs 1,2->3 and 3->2. For the expression

(R[1,3][2=2]R[2,3])[1,3,4],

we calculated

$$\text{Drv}((R[1,3][2=2]R[2,3])[1,3,4]) = \{3\text{->}2,\ \ldots\}.$$

In actual fact, the FD 1,2->3 is also valid but is not calculated by Drv. Let us see what happens by applying the above equivalence transformations. One of the two possible transformations is:

```
  (R[1,3][2=2]R[2,3])[1,3,4]  -->  (by [1a])
 (R[3=2]R[2,3])[1,3,4,5][1,3,4]  -->  (by [1b])
(R[3=3]R)[1,2,3,5,6][1,3,4,5][1,3,4].
```

We calculate the constraints valid in this last expression as follows:

```
Drv(R) = {1,2->3  3->2  ...}
Drv(R[3=3]R) = {1,2->3  3->2  3=6  4,5->6  6->5  1,2->6
                  6->2  4,5->2 4,5->3  3->5  2=5  1,5->3
                  2,4->6  1,5->6  2,4->3 ...}
                 (3->2 and 3->5 are the same FD)
Drv((R[3=3]R)[1,2,3,5,6]) = { 1,2->3  3->2  3=5  5->4  1,2->5
                              5->2  3->4  2=4  1,4->3  1,4->5  ...}
Drv((R[3=3]R)[1,2,3,5,6][1,3,4,5]) = {2=4  4->3  2->3  1,3->2
                                       1,3->4  ...}
Drv((R[3=3]R)[1,2,3,5,6][1,3,4,5][1,3,4] = {3->2  1,2->3  ...}
```

These are exactly the constraints in R (to which the expression is equivalent). The counterexample is no longer a problem, and this is an intuitive argument that completeness holds for the restricted syntax.

In the above discussions, we have first removed set difference from the mapping language, and then we restricted the syntax so that cross products are performed before projections. With these restrictions we can finally prove completeness of the derivation rules. We begin with a theorem which proves completeness for expressions without the restriction operator (syntax IV), and then we prove it for any expression of syntax III.

Theorem 10. Let s be a schema; let e be a consistent expression over s expressed in syntax IV. Then any valid EQ on e is a member of Drv(e), and if Z->A is valid on e, then for some $Z_1 \subseteq Z$, $Z_1$->A $\in$ Drv(e).

Proof. Proofs of completeness theorems generally follow the contrapositive direction: If c is not a member of Drv(e), i.e., if c is not derivable by the given rules, then c is not valid, i.e., there is some state st such that c is false in e(st). The state st is called a counterexample state. In the proofs of completeness for FDs and for MVDs[BeFH], the counterexample state is a single set of tuples since only one relation is being dealt with. In the present situation there are (possibly) many relations, each of which must be assigned a set of tuples for the counterexample state. This cannot be done in as straightforward a fashion as in the case for one relation because the expression e may have several occurrences of the same base relation. For example, suppose e has the form $e_1 \times e_2$ and that base relation R appears in both $e_1$ and $e_2$. If we try to arrive at an assignment of tuples to R by a recursive procedure (which is natural since relation algebra expressions are recursively defined), the procedure call on $e_1$ may result in an assignment to R which conflicts with that of the procedure call on $e_2$. In the proof of this theorem, which can be found in the appendix, we are able to overcome this problem because of the special structure of operations embodied in syntax IV. We first use results by other authors for the case of base relations and selections on base relations. Then for cross products of selections, we show how counterexample tuple sets on subexpressions can be integreted into the whole expression. The case of union uses Theorem 2, and the case for projection is straightforward. □

Now we want to use this theorem to prove completeness for expressions of syntax III, which contain restriction operators. We do this by defining a transformation from expressions of syntax III to those of syntax IV; that is, it is a transformation which removes occurrences of the restriction operator. This transformation will not be an equivalence, but it will have two properties which will be sufficient for our purposes. Namely, the resultant expression will (have values which will) always be a subset of the original expression, and the resultant expression will have the same set of derivable constraints as the original. We need to use this indirect approach because if we tried a direct inductive argument for a restriction e[X=Y] we could get a counterexample state on e; but after applying the restriction operator [X=Y], we could never be sure that the tuples were still present.

Let s be a schema and let e be an expression over s according to syntax III. We may write e as $r_1[X_1] \sqcup \ldots \sqcup r_n[X_n]$, where each $r_i$ is a restriction of a cross product (see the definition of syntax III). The <u>expression</u> <u>G(e)</u> over s conforming to syntax IV is defined as follows:

Let r be one of the terms $r_1,\ldots,r_n$, and let X=Y be one of the restrictions in r. Let r* be r with the restriction X=Y removed. Next, there are three cases: X≡V ∈ Drv(r*) for some V; Y≡V ∈ Drv(r*) for some V; and no VEQ on X or Y is in Drv(r*). In the first case replace r by r*[Y≡V]; in the second case

replace r by r*[X≡V], and in the third case replace r by $r^*[X \equiv V_1][Y \equiv V_1] \sqcup r^*[X \equiv V_2][Y \equiv V_2]$, where $V_1 \neq V_2$, and neither $V_1$ nor $V_2$ appear in e or in s. Now rearrange the resulting expression so that it is again of the form $r_1[X_1] \sqcup \ldots \sqcup r_n[X_n]$, where each $r_i$ is a restriction of a cross product. The new expression will have one less restriction than the original. Repeat this process until there are no more restrictions. The result is G(e). □

<u>Theorem</u> <u>11</u>. Let s be a schema, and let e be an expression over s of syntax III. Then Drv(G(e))=Drv(e), and for every structure str, G(e)(str) $\subseteq$ e(str).

Proof. The proof may be found in the appendix. □

The goal of this paper now appears as a simple corollary of Theorem 11:

<u>Theorem</u> <u>12</u>. Let s be a schema, and let e be a consistent expression over s expressed in syntax III. Then any valid EQ on e is a member of Drv(e), and if Z->A is valid on e, then for some $Z_1 \subseteq Z$, $Z_1$->A ∈ Drv(e).

Proof. Suppose e is consistent and c ∉ Drv(e). Since e is consistent, G(e) is consistent. If c is an EQ not in Drv(e)=Drv(G(e)), or if it is an FD Z->A such that $Z_1$->A ∉ Drv(e)=Drv(G(e)) for every $Z_1 \subseteq Z$, then by Theorem 10, there is a state st such that c is false in G(e)(st). Since G(e)(st) $\subseteq$ e(st), c is also false in e(st). □

## 6. Summary and Conclusions

This paper has presented results about calculating functional dependencies on relational algebra expressions. The motivation for studying this problem arose from the desire to be able to determine correct view mappings. A correct view mapping is one in which the constraints in the underlying schema and the properties of the mapping ensure that the constraints in the view schema are always satisfied (in the view). Given the underlying schema, the view schema and the mapping, we need a decision procedure to tell us the correctness or incorrectness of the mapping. This is a basic prerequisite for implementing multilevel database systems[ANSI].

The basic facilities of view mappings can be modelled by relational algebra, and a basic type of constraint appearing in schemas is the functional dependency; hence, the problem of recognizing correct view mappings is in large part the problem of calculating the valid functional dependencies on relational algebra expressions over the underlying schema. We studied this problem in detail in the preceding sections.

The results were both negative and positive. We found that the problem as stated is simply unsolvable. If we drop the set difference operator from relational algebra, we found that there is an algorithm for calculating functional dependencies on relational algebra expressions. The algorithm makes use of the derivation rules already known for functional dependencies on one relation, but it is complicated by the need to compare

derivations of functional dependencies. With suitable modification to the relational algebra language which does not decrease its power, we have shown that the algorithm is both sound — no invalid functional dependencies are accepted — and complete — all valid functional dependencies are found.

In terms of the feasibility of constructing the mapping processors in the framework of the ANSI/X3/SPARC Study Group, we may conclude that the processors can in principle be constructed, but care must be taken in the design of the mapping language, or else the processors cannot exactly determine the correct mappings.

We are currently working on a number of logical extensions to this work:

Although functional dependencies are very important constraint types, they are not the only ones which have a wide applicability. For example, there are constraints in hierarchical models which require that each child segment occurrence have a parent. In Codasyl network models, a similar construct is that a record type has mandatory membership in a set type. In relational models we have what are called foreign key constraints. These are all cases of what we call subset constraints. Their interaction with functional dependencies and mappings is being investigated.

In this paper we only studied structure mappings. A complete view specification will also include operation mappings. An operation applied to the view must be mapped to operations on the underlying relations such that the effect will be as if the view were operated on directly. We are currently investigating sound and complete algorithms for this type of problem.

## 7. References

[ABCE] Astrahan M.M., Blasgen M.W., Chamberlin D.D., Eswaran K.P., Gray J.N., Griffiths P.P., King W.F., Lorie R.A., McJones P.R., Mehl J.W., Putzolu G.R., Traiger I.L., Wade B.W. and Watson V. "System R: Relational Approach to Database Management" ACM-TODS 1, 2, pp.97-137, 1976

[AhBU] Aho A.V., Beeri C. and Ullman J.D. "The Theory of Joins in Relational Databases" Proc. 18th IEEE Symp. on Foundations of Computer Science, 1977

[ANSI] "The ANSI/X3/SPARC Framework: Report of the Study Group on Data Base Management Systems" D.Tsichritzis and A.Klug (eds.), AFIPS Press, 1977

[Arms] Armstrong W.W. "Dependency Structures of Data Base Relationships" Information Processing 74, North Holland Pub. Co.,1974

[BeFH] Beeri C., Fagin R. and Howard J.H. "A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations" Proc. ACM-SIGMOD Conf. 1977

[Bern] Bernstein P. "Synthesizing Third Normal Form Relations from Functional Dependencies," ACM-TODS, 1, pp.277-298, 1976

[ChGT] Chamberlin D.D., Gray J.N. and Traiger I.L. "Views, Authorization and Locking in a Relational Data Base System" AFIPS Conference Proceedings, 44, 1975

[Codd] Codd E.F. "Relational completeness of Data Base Sub-languages" Data Base Systems, R. Rustin (ed.), Prentice Hall, 1972

[Henk] Henkin L. "The Completeness of the First-Order Functional Calculus" J. Symbolic Logic, 14, pp.159-166, 1949

[Klug] Klug A. "Theory of Database Mappings," Ph.D. Thesis, University of Toronto, 1978

[Solo] Solomon M. "Undecidability of the Equivalence Problem for Relational Expressions" Bell Laboratories, to appear

## 8. Appendix: Proofs of Selected Theorems

Theorem 1. Let s be a schema and suppose $id_1$ and $id_2$ are associated with FDs derived on expression over s. Then $Eqv(id_1,id_2)$ true implies that for all states st and value sequences x, if $id_1(st;x)$ and $id_2(st;x)$ are defined, then they are equal.

Proof. By removing unnecessary tuples, we may consider the given state st to be the result of a procedure gentabl in which the new_formal_value function is replaced by a function not necessarily generating unique values. Suppose we apply gentabl to $id_1$ and $id_2$, obtaining a structure strl. Clearly, any equality relationships holding in the free structure fstr (before applying the infervals procedure) will also hold on the analogous tuples in strl. Now apply infervals to strl and fstr. This will not change the above property: Any equality holding in fstr will still hold on analogous tuples in strl. But strl is already a state, hence $id_1(fstr,V) = id_2(fstr,V)$ implies $id_1(strl;x) = id_2(strl;x)$, i.e., $id_1(st;x) = id_2(st;x)$. □

Theorem 3. Let s be a schema and e be an expression over s, and consider the following two statements:

(i) Every constraint in the set is valid;

(ii) for every FD $\{id_i\}$:Z->A in the set, for every state st and every sequence x of universe elements, a ∈ e[Z≡x[Z]][A](st) if and only if for some i, $id_i(st;x)=a$.

Let S be a set of EQs and labelled FDs defined on an expression e such that (i) and (ii) hold on S. Then (i) and (ii) hold on Cl(S).

Proof. We assume (i) and (ii) hold on S.

If c ∈ Cl(S) is present by one of clauses [1] through [5], then property (i) holds because of the basic properties of equality.

If id:X->Y ∈ Cl(S) because X=Y ∈ Cl(S) (clause [6]), then X->Y is valid because equality is functional (and valid by the induction hypothesis). Also, for any state st and sequence x, a ∈ e[X≡x[X]][Y](st) if and only if for some t ∈ e(st), x[X] = t[X] = t[Y] = a; i.e., if and only if a = id(st;x).

If id:ø->X ∈ Cl(S) because X≡V ∈ Cl(S) (clause [7]), then ø->X is valid because the X domain is constant. Also, for any state st and sequence x, a ∈ e[ø≡x[ø]][X](st) = e[X](st), if and only if a = V = id(st;x).

Suppose that {$id_{1i}$}:Z->$A_i$ and {$id_{2j}$}:Z->$A_2$ are in Cl(S) and that statement (ii) above is true for them and also that Eqv($id_{1i}$,$id_{2j}$) = true for all i,j. Then for every state st and any t ∈ e(st), t[$A_1$] ∈ e[Z≡t[Z]][$A_1$](st), so for some i, t[$A_1$]=$id_{1i}$(st;t). Similarly, for some j, t[$A_2$]=$id_{2j}$(st;t). Hence t[$A_1$]=t[$A_2$] and so $A_1$=$A_2$ is valid for e.

Suppose $id_{1i}$:Z->A and $id_{2j}$:A$\sqcup$X->B are in Cl(S) and satisfy (i) and (ii), and that $id_{3ij}$:Z$\sqcup$X->B is obtained per clause [9]. By inspection of the algorithm, we can see that $id_{3ij}$(st)(x) = $id_{2j}$(st)(x[$id_{1i}$(st)(x)/A]) (where x[y/k] denotes the sequence obtained from x by putting y at position k). By definition, b ∈ e[Z$\sqcup$X≡x[Z$\sqcup$X]][A](st) if and only if there is a t ∈ e(st) such that t[B]=b and t[Z$\sqcup$X]=x[Z$\sqcup$X]. This is true if and only if there is a t ∈ e(st) such that t[B]=b and t[A$\sqcup$X] = x[t[A]/A][A$\sqcup$X] and t[Z]=x[Z]. By induction, we know that b ∈ e[A$\sqcup$X≡x[t[A]/A][A$\sqcup$X]][B](st) if and only if b =

$id_{2j}$(st;x[t[A]/A]), and that t[A] ∈ e[Z≡x[Z]][A](st) if and only if t[A] = $id_{1i}$(st;x). Hence b ∈ e[Z$\sqcup$X≡x[Z$\sqcup$X]][B](st) if and only if b = $id_{2j}$(st;x[t[A]/A]) = $id_{3ij}$(st;x). □

Theorem 4. Let s be a schema, and let e be an expression over s. Then statements (i) and (ii) of Theorem 3 hold on the set Drv(e).

Proof. This proof is quite straightforward, with most of the work having been done in the previous theorem. We will only show how clause (ii) proceeds for projection and how clause (i) proceeds for union:

Suppose $id_2$:Z->A ∈ Drv(e[X]), where $id_1$:X[Z]->X[A] is in Drv(e) and $id_2$ is as described in clause [2]. First note that $id_2$(st)(x) = $id_1$(st)(x'), where x'=x[x[Z]/X[Z]]. Then a ∈ e[X][Z=x[Z]][A](st) if and only if there is a t ∈ e(st) such that a= t[X][A] = t[X[A]] and t[X[Z]] = t[X][Z] = x[Z] = x'[X[Z]], i.e., if and only if a ∈ e[X[Z]=x'[X[Z]]][X[A]](st), which means, by induction, that a = $id_1$(st)(x') = $id_2$(st)(x).

For union, suppose e is $e_1 \sqcup e_2$ and that {$id_{1i}$,$id_{2j}$}:Z->A ∈ Drv(e), where {$id_{1i}$}:Z->A ∈ Drv($e_1$), and {$id_{2j}$}:Z->A ∈ Drv($e_2$) and Eqv($id_{1i}$,$id_{2j}$) ∀i,j. Suppose t, t' are in e(st) and that t[Z]=t'[Z]. There is some $id_{km}$ (k=1 or 2) such that t[A]=$id_{km}$(st;t) and there is some $id_{1n}$ (l=1 or 2) such that t'[A]=$id_{1n}$(st;t). Regardless of the values of k and l, we have Eqv($id_{km}$,$id_{1n}$), and so t[A]=t'[A]. □

Theorem 5. Let s be a schema. The following problem is undecidable:

Given arbitrary relational expressions $e_1$ and $e_2$ over s of degree n, determine if $e_1 \equiv e_2$.

Proof. Let us say that two expressions $e_1$ and $e_2$ are strongly equivalent (with respect to some schema s) if $e_1(str)=e_2(str)$ for all structures str for s. In [Solo] it is proved that the strong equivalence problem is undecidable when the expressions do not contain selections. Clearly, the problem is still undecidable when selections can also appear. From this we conclude that equivalence problem (i) is undecidable since an algorithm for it would still work when the schema contains no constraints.

As an aside, we will show that we can also reduce equivalence problem (i) to the strong equivalence problem:

First we show how to express constraints as relational algebra equalities: An FD R:Z->A is true in a structure str if and only if (R[Z=Z]R)[A=A'](str) = (R[Z=Z]R)(str), where A'=A+deg(R); a DEQ R:X=Y is true in str if and only if R[X=Y](str) = R(str), and a VEQ R:X=V is true in str if and only if R[X=V](str) = R(str).

Given a schema s, let L be the cross product of the left-hand-sides of every equality as above corresponding to each constraint in s, and let R be the cross product of the corresponding right-hand-sides. Note that L(str)=R(str) if and only if str is a state, and that L(str) $\subseteq$ R(str) for all structures str. Now suppose that the relations in s are $R_1,\ldots,R_n$. define the expression $W = R_1[1] \sqcup \ldots \sqcup R_n[1]$. Then W has degree 1, and $W(str)=\emptyset$ if and only if $R(str)=\emptyset$ for each R $\in$ s. Now define $E = W-(W \times (R-L))[1]$. This expression has the property that $E(str)=\emptyset$ if and only if str is not a state or str is empty.

We now have the reduction: $e_1 \equiv e_2$ if and only if $e_1 \times E$ is strongly equivalent to $e_2 \times E$. □

Theorem 10. Let s be a schema; let e be an expression over s according to syntax IV be consistent. Then any valid EQ on e is a member of Drv(e), and if Z->A is valid on e, then for some $Z_1 \subseteq Z$, $Z_1$->A $\in$ Drv(e).

Proof. First consider a selection $e=R[X_1 \equiv V_1]\ldots[X_m \equiv V_m]$, where $X_i$, $V_i$ are single domains or values, respectivily. We first define a set of equivalence classes of domains $\{E_i : 1 \leq i \leq n\}$ determined by the EQ constraints on e. This will allow us to remove EQs from consideration and to utilize existing theorems on the completeness of rules for FDs.

For each set $E_i$ we associate a value $V_i$: If X $\in$ $E_i$ and $X \equiv V \in$ Drv(R), then $V_i$=V. (This value will be unique if it exists since R is consistent.) Otherwise, $V_i$ is an arbitrary, unique integer. (Hence, $i \neq j$ implies $V_i \neq V_j$.) We first define a state $st_1$ such that $e(st_1)$ contains only one tuple t. Namely, t is such that $t[X]=V_j$ if and only if X $\in$ $E_j$. Note that since $R(st_1)$ has a cardinality of one, all FDs are true in $R(st_1)$. Now, if X=Y is not derivable, then for some i and j, X $\in$ $E_i$, Y $\in$ $E_j$, and $i \neq j$. Hence (in $e(st_1)$), $t[X]=V_i$, $t[Y]=V_j$, and so $t[X] \neq t[Y]$. The state $st_1$ is therefore a counterexample state showing that e:X=Y is not valid.

Similarly, if $X \equiv V$ is not derivable, then $V_i \neq V$, where X $\in$ $E_i$ and $t[X]=V_i$, $t[X] \neq V$, again showing that $st_1$ is a counterexample to the validity of $e:X \equiv V$.

Now consider an FD X->Y such that for no $X_1 \subseteq X$ is $X_1$->Y $\in$ Drv(e). Using the sets $\{E_i : 1 \leq i \leq n\}$ defined above we define a function g which maps FDs on e to FDs on an n-ary relation R'. First, we write g(X)={i} if X $\in$ $E_i$ and $V_i$ was an arbitrary value; otherwise g(X)=$\emptyset$ (where $V_i$ was assigned because X$\equiv$$V_i$ was derivable). Then for a set Z={$Z_1$...$Z_n$} of domains, g(Z) = g($Z_1$)$\sqcup$...$\sqcup$g($Z_n$), and finally, g(Z->A) = g(Z)->g(A). We let s' be the schema consisting of R' and also every FD g(Z->A) where R:Z->A $\in$ s. First we note that g(X)->g(Y) is not derivable in s', i.e., there is no X' $\subseteq$ g(X) such that X'->g(Y) $\in$ Drv(R'). (X->Y is the given non-derivable FD.) Since the reflexive and pseudotransitivity rules are complete (according to our interpretation which incorporates augmentation) for FDs on one relation (with no EQs)[Arms], there exists a state R'($st_2$) such that g(X)->g(Y) is false in R'($st_2$). We now use g to construct a counterexample state $st_3$ for R:X->Y. To do this we define a function h, an inverse of g, which, from tuples of R'($st_2$), will yield tuples of R($st_3$). Namely, if t' $\in$ R'($st_2$), then h(t') is the tuple t such that t[X]=V if X$\equiv$V is derivable, and t[X]=t'[i] when $V_i$ was arbitrary and X $\in$ $E_i$. Then we let R($st_3$) = h(R'($st_2$)). It is not hard to see that all constraints in s are true in R($st_3$) and that X->Y is false in R($st_3$).

We next assume that the expression e is a cross product of selections or other cross products. That is, e is of the form $R_1[X_1 \equiv V_1] \times \ldots \times R_n[X_n \equiv V_n]$, where each $X_i$ and $V_i$ may be a list. Suppose that Z->A is such that $Z_1$->A $\notin$ Drv(e) for every $Z_1 \subseteq$ Z. Let $Z_1 \subseteq$ Z be the domains of e appearing in the same selection term that A appears in. ($Z_1$ may be empty.) Let us also

assume that this term is first in e, i.e., that we can write e as $R[X_1 \equiv V_1] \times e_2$. Then for no $Z_2 \subseteq Z_1$ is $Z_2$->A $\in$ Drv(R[$Z_1 \equiv V_1$]) for otherwise we would have $Z_2$->A $\in$ Drv(e). By the induction hypothesis, there is a state $st_1$ such that $Z_1$->A is false in R[$X_1 \equiv V_1$]($st_1$). If $e_2$($st_1$) (the remainder of e) is nonempty, then Z->A will also be false in e($st_1$), for if $t_1$ and $t_2$ are tuples of R[$X_1 \equiv V_1$]($st_1$) contradicting $Z_1$->A and if t' is any tuple of $e_2$($st_2$), then $t_1 \times t'$ and $t_2 \times t'$ are tuples in e($st_1$) contradicting Z->A. It remains to show that if $e_2(st_1)=\emptyset$, that we can modify the state to get $e_2$($st_1$) nonempty while retaining tuples in R[$X_1 \equiv V_1$] contradicting $Z_1$->A.

First consider the subexpression r of e consisting of all the selections on the base relation R: r = $R[X_1 \equiv V_1] \times \ldots \times R[X_m \equiv V_m]$. (The Xs and Vs have been renumbered and any selection term not in r is on a base relation other than R.) We will write this as $s_1 \times \ldots \times s_m$. Before we proceed to modify $st_1$ so that r is nonempty, we prove a lemma. Note that every domain Y in r can be written $d_i$+X where $1 \leq X \leq$deg(R) and $d_i$ (a "displacement") equals (i-1)deg(R), where Y is a domain in the term $s_i$. Then we have:

(i) if $d_i$+X=$d_j$+Y $\in$ Drv(r), i$\neq$j, X is different from Y, and no VEQ $d_i$+X$\equiv$V or $d_j$+Y$\equiv$V is derivable in r, then $d_i$+X=$d_j$+X and $d_i$+Y=$d_j$+Y are in Drv(r), and X=Y is in Drv(R);

(ii) if $d_i$+X$\equiv$V $\in$ Drv(r), then $\emptyset$->X $\in$ Drv($s_i$), and

(iii) if $d_i$+X=$d_j$+X $\in$ Drv(r), then $\emptyset$->X $\in$ Drv($s_i$)$\sqcap$Drv($s_j$).

This lemma describes the kind of EQs which can hold on a cross product of selection terms, each term on the same base relation. A formal proof of the properties is given, but first we will give an intuitive discussion.

Consider a cross product $r = s_1 \times \ldots \times s_n$, where each $s_i$ has the form $R[X_i \equiv V_i]$, with $X_i, V_i$ lists.

An EQ within a single term, e.g., $d_i+X=d_i+Y$ could be the result of the selections on that term: $d_i+X \equiv V$ and $d_i+Y \equiv V$ are in $Drv(s_i)$. If this is not the case, then the E must be the result of the EQ X=Y which must be in Drv(R). This is because the only linking from one term to another in r is through VEQs with the same value. The transitivity rule may have been used in the derivation of $d_i+X=d_i+Y$, but eventually we will arrive at the above result.

Consider an EQ $d_i+X=d_j+X$, an equality of corresponding domains in different terms. As above, this equality could be the result of VEQs $d_i+X \equiv V$ in $Drv(s_i)$ and $d_j+X \equiv V$ in $Drv(s_j)$. If this is not the case, it must (eventually) be the result of an FD Z->X and VEQs $d_i+Z \equiv V$ in $Drv(s_i)$ and $d_j+Z \equiv V$ in $Drv(s_j)$. By composing, we get $\emptyset \to d_i+X$ in $Drv(s_i) \sqcap Drv(s_j)$. That is, if an EQ is derived by equivalent FDs, the left-hand-sides of the FDs must be equal because of VEQs and this means that the domains

Consider an EQ $d_i+X=d_j+Y$ with $i \neq j$ and $X \neq Y$. Again, this EQ may be the immediate result of VEQs. If not, it will be the result of the previous two cases using transitivity. That is, $d_i+X=d_j+Y$ will be derivable in r, and X=Y will be derivable in R. We can further derive $d_i+Y=d_j+Y$ in r.

Finally, consider a VEQ $d_i+X \equiv V$ in Drv(r). If this VEQ is also in $Drv(s_i)$, then $\emptyset \to X$ is in $Drv(s_i)$. But $d_i+X$ will be constant in $s_i$ even if the VEQ is not derivable in $s_i$, since cross products do not filter out any tuples.

To prove (i), note that rule [2] of C1 (transitivity) is the only one which can generate the indicated EQ. So we use induction on the number of applications of rule [2].

There are only two possible pairs of EQs to which transitivity can be applied and which themselves are not the result of transitivity. One pair consists of $d_i+X=d_k+X$, which would be the result of rule [8], and $d_k+X=d_j+Y$, which would be the result of an EQ X=Y derivable in $s_i$ and $s_j$. From X=Y we can get $d_i+Y=d_i+X$, which yields $d_i+Y=d_j+Y$ derivable in r, and also $d_j+Y=d_j+X$, which yields $d_i+X=d_j+X$ derivable in r. Now the EQ X=Y must, in fact, be derivable in R, for the only other way to get an EQ in a selection on a base relation is to derive it from VEQs which we assumed impossible.

The other pair of EQs is like the one above with the roles of X and Y interchanged.

Now assume the hypotheses hold and also the there are EQs $d_i+X=d_k+Z$ and $d_k+Z=d_j+Y$ derivable in r. First assume $X \neq Z \neq Y$. We cannot have any VEQs $d_k+Z \equiv V$ since this would yield $d_i+X \equiv V$ and $d_j+Y \equiv V$. By induction, we have $d_k+Z=d_j+Z$ and $d_i+Z=d_k+Z$ derivable in r, and X=Z and Y=Z derivable in R. We therefore have X=Y derivable in R and $d_i+Z=d_j+Z$ derivable in r, and from this we get $d_i+X=d_j+X$ and $d_i+Y=d_j+Y$ derivable in r. If Z is X, then we have $d_i+X=d_k+X$ and $d_k+X=d_j+Y$ derivable in r. By induction on the second EQ, we have $d_k+X=d_j+X$ and $d_k+Y=d_j+Y$ derivable in r and $X=Y$ derivable in R. From this we derive $d_i+X=d_j+X$ and $d_i+Y=d_j+Y$ in r. If Z is Y we proceed analogously.

To prove part (ii), first assume $X \equiv V \in Drv(s_i)$. Then, clearly, $\emptyset \to X \in Drv(s_i)$. If $d_i+X=d_j+Y$ and $d_j+Y \equiv V$ are in $Drv(r)$ and $i \neq j$ and $X \neq Y$, then by (i) $d_i+X=d_j+X \in Drv(r)$. By (iii), $\emptyset \to X \in Drv(s_i)$. If $i=j$ or X is Y, we also get $\emptyset \to X \in Drv(s_i)$.

To prove (iii), we have that if $d_i+X \equiv V$ and $d_j+X \equiv V$ are in $Drv(r)$, then $\emptyset \to X \in Drv(s_i) \sqcap Drv(s_j)$ by (ii). If $d_i+X=d_k+Y$ and $d_k+Y=d_j+X$ are in $Drv(r)$, and $X \neq Y$, then by (i), $d_i+X=d_k+X$ and $d_k+X=d_j+X$ are in $Drv(r)$, so by induction, $\emptyset \to X \in Drv(s_i)$ and $\emptyset \to X \in Drv(s_j)$. If the EQ is the result of rule [8], then there must be an FD $Z \to X \in Drv(R)$ such that $d_i+Z=d_j+Z$ is in $Drv(r)$. By induction, $\emptyset \to Z \in Drv(s_i) \sqcap Drv(s_j)$, and therefore $\emptyset \to X \in Drv(s_i) \sqcap Drv(s_j)$.

This proves the lemma.

We now will indicate how to modify $st_1$. We may suppose that $R(st_1) = R[X_1 \equiv V_1](st_1) = \{t_1, t_2\}$. (If not, delete the extra tuples; they do not add anything.) First let u be a 1:1 function defined on values appearing in $t_1$ and $t_2$ such that its image is distinct from the values in $t_1$ and $t_2$ and from the values appearing in the selection terms of r. Let $E_1, \ldots, E_n$ be the equivalence classes of the domains of r under '='. Associate a value $V_i$ with each $E_i$ as follows: (i) $V_i=V$ if $X \equiv V \in Drv(r)$ for some $X \in E_i$; (ii) if there is some $X \in E_i$ such that $1 \leq X \leq deg(R)$ (if X is a domain of $s_1$), then $V_i=u(t_1[X])$; (iii) otherwise $V_i$ is an arbitrary unique value. Define $R(st_2) = \{t_1', t_2'\}$ as follows: $t_1'[X] = V_k$ if $X \in E_k$ and $V_k$ is assigned by a VEQ;

$t_1'[X] = u(t_1[X])$, otherwise. Also, $t_2'[X] = V_k$ if $X \in E_k$ and $V_k$ is assigned by a VEQ; $t_2'[X] = u(t_2[X])$, otherwise.

We first show that $st_2$ is a state, that $t_1'$ and $t_2'$ appear in $s_1$, and that $st_2$ is still a counterexample state for $Z_1 \to A$:

If $X \equiv V$ is a component of $X_1 \equiv V_1$, then $V_k$ is assigned by a VEQ, where $X \in E_k$ and $V_k=V$. Hence $t_1'[X]=t_2'[X]=V$. Thus $R(st_2)$ satisfies the VEQs of $R[X_1 \equiv V_1]$, and $R[X_1 \equiv V_1](st_2) = R(st_2)$.

For the other properties of $R(st_2)$, we will show that $t_1'[X]=t_2'[X]$ if and only if $t_1[X]=t_2[X]$. First, it is clear from the definition that $t_1[X]=t_2[X]$ implies $t_1'[X]=t_2'[X]$.

Now suppose $t_1'[X]=t_2'[X]$. If $V_k$ is not assigned by a VEQ, where $X \in E_k$, then $t_1'[X] = u(t_1[X])$ and $t_2'[X] = u(t_2[X])$, and since u is 1:1, we get $t_1[X]=t_2[X]$.

Now suppose $t_1'[X] = V_k = t_2'[X]$, where $X \in E_k$ and $V_k$ is assigned by a VEQ. This means that $X \equiv V_k$ is derivable in r. By part (ii) of the lemma, $\emptyset \to X$ is in $Drv(s_1)$. Since $t_1, t_2 \in R[X_1 \equiv V_1](st_1)$, we have $t_1[X]=t_2[X]$. This proves $t_1[X]=t_2[X]$ if and only if $t_1'[X]=t_2'[X]$. From this we can conclude that FDs and DEQs which are true (false) in $R(st_1)$ are true (false) in $R(st_2)$. Thus $st_2$ is a state of s, and $Z_1 \to A$ is false in $R[X_1 \equiv V_1](st_2)$.

The next step is to show that we can add tuples to $R(st_2)$ to get a nonempty state for every other selection in r.

Consider a selection term $s_i=R[X_i \equiv V_i]$ $(i=2,\ldots,m)$ in r of displacement $d_i$; that is, the domains of the selection are $d_i+1, \ldots, d_i+deg(R)$. Define a tuple $t_i$ by $t_i[X]=V_k$ where $d_i+X \in$

$E_k$. By construction, if $t_i$ is placed in $R(st_2)$, then $t_i$ will appear in $R[X_i \equiv V_i]$. It is also easy to see that $t_i$ will satisfy all EQs in schema s. We must show that when $t_2,\ldots,t_m$ so constructed, are added to $R(st_2)$, that no FDs are violated. First we show that there will be no FD violations among $t_2,\ldots,t_m$. So suppose $R:W\text{->}B$ is in s and $t_i[W]=t_j[W]$. Then for each component $W_k$ of W, $t_i[W_k]=t_j[W_k]$. If one of the values, say $V_w$, was assigned by clause (i) or (iii) of the definition of $\{E_i\}$, then both $d_i+W_k$ and $d_k+W_k$ are in $E_w$ since this $V_w$ is distinct from all other values associated with E-sets. If one of the values was assigned by the second clause, then both were, and we have, for some $U_1$ and $U_2$, $1 \le U_1, U_2 \le deg(R)$, $d_i+W_k=U_1$ and $d_j+W_k=U_2$ derivable. If $U_1$ is $W_k$, we have $d_i+W_k=W_k$ derivable in r; if not, then part (i) of the lemma will give $d_i+W_k=W_k$ derivable in r. Similarly, $d_j+W_k=W_k$ is derivable in r. Hence $d_i+W_k=d_j+W_k$ is derivable in r. The component k was arbitrary, so we may write: $d_i+W=d_j+W$ is derivable in r. Now $d_i+W\text{->}d_i+B$ and $d_j+W\text{->}d_j+B$ are also derivable in r and have the same identifier. We therefore have $d_i+B=d_j+B \in Drv(r)$, and therefore $t_i[B]=t_j[B]$ since $d_i+B$ and $d_j+B$ are in the same E-set.

Now we will show that there will be no FD violations between $t_1'$ or $t_2'$ and any of the $t_i$, $i=2,\ldots,m$. So suppose $R:W\text{->}B$ is in s and $t'[W]=t_i[W]$ (where t' is $t_1'$ or $t_2'$). For each component $W_j$ of W, $t'[W_j]=t_i[W_j]$. First suppose $V_k$ is assigned by a VEQ, or by the third clause, where $d_i+W_j \in E_k$. The values assigned to these E-sets are unique, so we also have $0+W_j \in E_k$, i.e., $W_j=d_i+W_j$ is derivable in r. Next suppose $V_k$ was assigned by the second clause: There is an $X \in E_k$ with $1 \le X \le deg(R)$. We

have $d_i+W_j=X$ derivable, but from the lemma, we also get $d_i+W_j=W_j$ derivable. Collectively, we have $W=d_i+W$ derivable in r. Since $W\text{->}B$ and $d_i+W\text{->}d_i+B$ have the same identifier, we get $B=d_i+B$ in Drv(r). If these domains (B and $d_i+B$) are in an E-set $E_k$ whose value is assigned by a VEQ, then $t'[B]=t_i[B]=V_k$. Otherwise, $V_k$ is equal to $u(t_1[X])$, where $1 \le X \le deg(R)$ and $X \in E_k$. Since X=B will be in Drv(r), it is also in $Drv(R[X_1 \equiv V_1])$ and if t' is $t_1'$, then we know $t_i[B]=u(t_1[X])=u(t_1[B])=t_1'[B]$. If t' is $t_2'$, then from $B=d_i+B$ we can conclude that $\emptyset\text{->}B$ is in $Drv(R[X_1 \equiv V_1])$, and this also yields $t_i[B]=t_1'[B]=t_2'[B]$. Thus no FDs are violated, and we may define a state $st_3$ by $R(st_3) = R(st_2) \sqcup \{t_2,\ldots,t_n\}$. This state will have the property that $r(st_3) \ne \emptyset$ and $Z_1\text{->}A$ is false in $R[X_1 \equiv V_1](st_3)$.

We still may have $e(st_3)=\emptyset$ because selections on other relations may be empty. However, by a process similar to the one above, we may add tuples to get a nonempty cross product.

The case for EQs which are not derivable in the cross product can be handled in an analogous fashion.

This completes the case for cross product.

Now suppose that e is a union $e_1 \sqcup e_2$ and that $Z_1\text{->}A \notin Drv(e_1 \sqcup e_2)$ for every $Z_1 \subseteq Z$. In particular, we have $Z\text{->}A \notin Drv(e_1 \sqcup e_2)$. Suppose $Z\text{->}A$ is not derivable in $e_1$. By induction there is a state $st_1$ such that $Z\text{->}A$ is false in $e_1(st_1)$. Then $Z\text{->}A$ will also be false in $(e_1 \sqcup e_2)(st_1)$. If $Z\text{->}A$ is not derivable in $e_2$, we proceed in a similar manner. In the remaining case $\{id_{1i}:Z\text{->}A \in Drv(e_1)$, $\{id_{2j}\}:Z\text{->}A \in Drv(e_2)$ but $\neg Eqv(id_{1i},id_{2j})$ for some i,j. By Theorem 2, there is a state st and a valuation x such that $id_{1i}(st;x) \neq id_{2j}(st;x)$, and by Theorem 3 there are tuples $t_1 \in e_1(st)$ and $t_2 \in e_2(st)$ such that $t_1[Z]=t_2[Z]$ but $t_1[A] \neq t_2[A]$. Thus we will have $t_1$, $t_2 \in (e_1 \sqcup e_2)(st)$, and these tuples will contradict the FD Z->A.

If an EQ is not derivable in $e_1 \sqcup e_2$, it is not derivable in either $e_1$ or $e_2$. We construct a counterexample state by induction on the appropriate component ($e_1$ or $e_2$), and this will also also be a counterexample state for the union.

In the last case, e is a projection $e_1[X]$. If c is any constraint not derivable in e, then $c_1$ will not be derivable in $e_1$, where $c_1$ is c with each domain Y replaced by X[Y]. By induction, we construct a counterexample state for $c_1$ in $e_1$ and this will also be a counterexample state for c in e. □

The theorem demonstrating the properties of G needs the following lemma which says that moving operators around does not change the set of derived constraints.

Lemma. Let s be a schema, and let e, $e_1$ and $e_2$ be expressions over s. Then

(i) $Drv(e[X=Y][Z\equiv V]) = Drv(e[Z\equiv V][X=Y])$

(ii) $Drv((e_1 \times e_2)[X\equiv V]) = Drv(e_1[X\equiv V] \times e_2)$, if $X \leq deg(R)$

(iii) $Drv((e_1 \times e_2)[X\equiv V]) = Drv(e_1 \times (e_2[X\equiv V]))$, if $X > deg(R)$

(iv) $Drv((e_1 \sqcup e_2)[X]) = Drv(e_1[X] \sqcup e_2[X])$

Proof. (i) From the definitions we have

$Drv(e[X=Y][Z\equiv V]) = Cl(Cl(Drv(e) \sqcup X=Y) \sqcup Z\equiv V)$ and

$Drv(e[Z\equiv V][X=Y]) = Cl(Cl(Drv(e) \sqcup Z\equiv V) \sqcup X=Y)$.

It is not hard to show that the inclusion $Cl(Cl(Drv(e) \sqcup X=Y) \sqcup Z\equiv V) \subseteq Cl(Cl((Drv(e) \sqcup Z\equiv V) \sqcup X=Y))$ is true and also that the reverse inclusion holds.

(ii) The definitions give us

$Drv((e_1 \times e_2)[X\equiv V]) = Cl(Cl(Drv(e_1) \sqcup Drv'(e_2)) \sqcup X\equiv V)$ and
$Drv(e_1[X\equiv V] \times e_2) = Cl(Cl(Drv(e_1) \sqcup X\equiv V) \sqcup Drv'(e_2))$.

Again we can show equality by showing inclusion in both directions.

Part (iii) is analogous.

(iv) The formulas give us

$Drv((e_1 \sqcup e_2)[X]) = \{id_2{:}Z\text{->}A : id_1{:}X[Z]\text{->}X[A] \in Drv(e_1 \sqcup e_2)\} \sqcup$
$\{Y=Z : X[Y]=X[Z] \in Drv(e_1 \sqcup e_2)\} \sqcup$
$\{Y\equiv V : X[Y]\equiv V \in Drv(e_1 \sqcup e_2)\}$, where $id_1$ and $id_2$ are as in the definition, and

$Drv(e_1[X] \sqcup e_2[X]) = \{x \in Drv(e_1[X]) \sqcap Drv(e_2[X]) : x \text{ is an EQ}\} \sqcup$
$\{id_1{:}Z\text{->}A \in Drv(e_1[X]) :$ there is $id_2{:}Z\text{->}A$ in $Drv(e_2[X])$ with $Rdc(id_1) = Rdc(id_2)\}$.

First we see that

$\{Y=Z : X[Y]=X[Z] \in Drv(e_1 \sqcup e_2)\} \sqcup \{Y\equiv V : X[Y]\equiv V \in Drv(e_1 \sqcup e_2)\}$
=
$\{x \in Drv(e_1[X]) \sqcap Drv(e_2[X]) : x \text{ is an EQ}\}$.

Now an FD $id_1{:}X[Z]\text{->}X[A]$ is in $Drv(e_1 \sqcup e_2)$ if and only if $id_1{:}X[Z]\text{->}X[A] \in Drv(e_1)$ and there is $id_2{:}X[Z]\text{->}X[A]$ in $Drv(e_2)$ with $Rdc(id_1)=Rdc(id_2)$. This is true if and only if $id_1'{:}Z\text{->}A$

$\in$ Drv($e_1$[X]) and $id_2'$:Z->A $\in$ Drv($e_2$[X]), where $id_1'$ ($id_2'$) is obtained from $id_1$ ($id_2$) by adding an arc to each leaf node and to the root node. Now Rdc($id_1'$)=Rdc($id_1$)=Rdc($id_2$)=Rdc($id_2'$), and so the condition is equivalent to $id_1'$:Z->A $\in$ Drv($e_1$[X]$\sqcup e_2$[X]). This shows that the two sets of FDs are the same. □

Theorem 11. Let s be a schema, and let e be an expression over s of syntax III. Then Drv(G(e))=Drv(e), and for every structure str, G(e)(str) $\subseteq$ e(str).

Proof. We first note that for any expressions $e_1,e_2,e_3$, if $e_1$(str) $\subseteq$ $e_2$(str), then $e_1$[X≡V](str) $\subseteq$ $e_2$[X≡V](str), ($e_1 \times e_3$)(str) $\subseteq$ ($e_2 \times e_3$)(str), ($e_3 \times e_1$)(str) $\subseteq$ ($e_3 \times e_2$)(str), $e_1$[X=Y](str) $\subseteq$ $e_2$[X=Y](str), ($e_1 \sqcup e_3$)(str) $\subseteq$ ($e_2 \sqcup e_3$)(str) and $e_1$[X](str) $\subseteq$ $e_2$[X](str). In other words, the set-inclusion condition is preserved by all the relational algebra operators (except, of course, set difference).

Now suppose e has the form $r_1[X_1] \sqcup \ldots \sqcup r_n[X_n]$, where each $r_i$ is a restriciton of a cross product. Let $r_i \in \{r_1,\ldots,r_n\}$ which contains a restriction X=Y. As in the definition, r* is $r_i$ with X=Y removed. We treat each of the three cases in the definition.

In the first case, since X≡V $\in$ Drv(r*), we have X=Y $\in$ Drv(r*[Y≡V]). Therefore Drv($r_i$) $\subseteq$ Drv(r*[Y≡V]). On the other hand, since X≡V and X=Y are in Drv($r_i$), we have Drv(r*[Y≡V]) $\subseteq$ Drv($r_i$). Therefore Drv(r*[Y≡V]) = Drv($r_i$). This implies that Drv($r_1[X_1] \sqcup \ldots \sqcup r_i[X_i] \sqcup \ldots \sqcup r_n[X_n]$) = Drv($r_1[X_1] \sqcup \ldots \sqcup$ r*[Y≡V]$[X_i] \sqcup \ldots \sqcup r_n[X_n]$). Moving the selection

term Y≡V inside to its base relation gives us, by the previous lemma Drv($r_1[X_1] \sqcup \ldots \sqcup r_i'[X_i] \sqcup \ldots \sqcup r_n[X_n]$), where $r_i'$ has one less restriction than $r_i$.

It is easy to see that r*[Y≡V](str) $\subseteq$ $r_i$(str) for all structures str. (In fact, r*[Y≡V] ≡ $r_i$.)

The second case is analogous to the first case.

In the third case, $r_i$ is to be replaced by r' = r*[X≡$V_1$][Y≡$V_1$]$\sqcup$r*[X≡$V_2$][Y≡$V_2$], where $V_1 \neq V_2$ are new values. The EQ X=Y is derivable in both terms of this union, so Drv($r_i$) $\subseteq$ Drv(r'). Any VEQ Z≡V in Drv(r') must be derivable in both r*[X≡$V_1$][Y≡$V_1$] and r*[X≡$V_2$][Y≡$V_2$]. This means that $V \neq V_1$ and $V \neq V_2$, and therefore Z≡V is derivable in r*. From this we conclude that Z≡V $\in$ Drv($r_i$). Suppose an EQ Z=W is derivable in both r*[X≡$V_1$][Y≡$V_1$] and r*[X≡$V_2$][Y≡$V_2$]. Then it is either derivable in r*, or it follows from X≡$V_1$, Y≡$V_1$ or X≡$V_2$, Y≡$V_2$ via the EQ X=Y. In both cases it is derivable in $r_i$.

Now suppose an FD Z->A is derivable in r'. Since it must have equivalent identifiers in both r*[X≡$V_1$][Y≡$V_1$] and in r*[X≡$V_2$][Y≡$V_2$], the identifiers must not have any nodes labelled with either $V_1$ or $V_2$. This means no FDs ∅->X or ∅->Y whose identifiers would be '$V_1$'->X, '$V_2$'->X, However, the FDs X->Y and Y->X, whose identifiers are themselves can be used, but if so, it is because the FD was derived from the EQ X=Y (unless X->Y or Y->X $\in$ Drv(r*) which is still fine.) Thus Z->A will also be derivable in $r_i$. We have therefore shown that Drv(r') $\subseteq$ Drv($r_i$). Let $r_i'$ = r*[X≡$V_1$][Y≡$V_2$] and $r_i''$ = r*[X≡$V_2$][Y≡$V_2$]. Then by the previous lemma, we have Drv($r_1[X_1] \sqcup \ldots \sqcup r_i[X_i] \sqcup \ldots \sqcup r_n[X_n]$) =

$Drv(r_1[X_1] \sqcup \ldots \sqcup r_i'[X_i] \sqcup r_i''[X_i] \sqcup \ldots \sqcup r_n[X_n])$. We also note that if $t \in r^*[X \equiv V_1][Y \equiv V_1](str)$, then $t[X]=t[Y]$, and so $t \in r^*[X=Y](str) = r_i(str)$. Similarly, we get $r^*[X \equiv V_2][Y \equiv V_2](str) \subseteq r_i(str)$. Thus $r_i'(str) \subseteq r_i(str)$.

We have shown for every step in the construction of G(e), that the Drv-set does not change, and that structures of the new expression are subsets of structures of the original expression e. Therefore Drv(G(e))=Drv(e), and for all structures str, $G(e)(str) \subseteq e(str)$. □

Theorem 12. Let s be a schema, and let e be an expression over s which is consistent and of syntax III. Then any valid EQ on e is a member of Drv(e), and if Z->A is valid on e, then for some $Z_1 \subseteq Z$, $Z_1$->A $\in$ Drv(e).

Proof. Suppose e is consistent and c $\notin$ Drv(e). Since e is consistent, G(e) is consistent. If c is an EQ not in Drv(e)=Drv(G(e)), or if it is an FD Z->A such that $Z_1$->A $\notin$ Drv(e)=Drv(G(e)) for every $Z_1 \subset Z$, then by Theorem 10, there is a state st such that c is false in G(e)(st). Since $G(e)(st) \subseteq e(st)$, c is also false in e(st). □