The nested relational algebra : a tool to handle structured information

The Nested Relational Algebra: A Tool to handle Structured Information

by

G.J. Houben, J. Paredaens,
D. Tahon.

88/04

March 1988

COMPUTING SCIENCE NOTES

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands
All rights reserved
editor: F.A.J. van Neerven

The Nested Relational Algebra : A Tool to handle Structured Information

G.J. Houben, Univ. of Technology, Eindhoven, The Netherlands
J. Paredaens, University of Antwerp, Belgium
D. Tahon, University of Antwerp, Belgium

Introduction

In database theory, an algebra is a set of operators that express new relations in terms of one or two operand relations. In this way queries can be defined. These queries handle ***structural information.*** This information is the counterpart of the *computable information*, such as the calculation of the sum of numbers or the detection of the larger of two values. The latter information cannot be expressed in the algebra we use here.

A nested relation is a data structure that is used to represent structured information in a database. It can be considered as a table whose entries can be atomic or nested relations themselves.

Thomas and Fisher [TF] introduced a model that allows nested relations. They also defined an algebra of operators for it. Roth, Korth and Silberschatz [RKS] defined a calculus-like query language for the nested relational model. Since then different languages have been introduced that are based on this model.

This paper illustrates the expressive power of the nested relational algebra. It demonstrates that this algebra is a suitable model for the implementation of nested relational languages. It also gives some examples of queries that cannot be expressed in the nested algebra.

In section 1 we give the formal definition of the nested algebra. Next we prove that the operators of the algebra are independent. Section 3 till section 6 illustrate different classes of operators, expressions and queries that are expressible in the algebra.

Finally, section 7 gives some examples of operators that handle structured information but that cannot be expressed in the algebra.

1 Preliminaries

We define the relation schemes, the relation instances and the operators of the nested algebra. We use almost the same definitions as in [PVG].

1.1 Relation scheme

An *attribute* can be an identifier :

$$<attribute> \rightarrow <identifier>$$

Such an attribute is called *atomic* and $<identifier>$ is called the *name* of the attribute.

An *attribute* can also have the form

$$<attribute> \rightarrow <identifier><scheme>$$

Such an attribute is called *structured* and $<identifier>$ is called the *name* of the attribute.

$$\begin{aligned} <list\ of\ attributes> &\rightarrow <empty\ list> \mid \\ & \quad <non\!-\!empty\ list\ of\ attributes> \\ <non\!-\!empty\ list\ of\ attributes> &\rightarrow <attribute> \mid \\ & \quad <attribute><non\!-\!empty\ list\ of\ attributes> \end{aligned}$$

A *relation scheme* or a *scheme* has the form

$$<scheme> \rightarrow (<non\!-\!empty\ list\ of\ attributes>)$$

All identifiers in a scheme must be different.

Two attributes are called *compatible* if and only if they are both atomic or their schemes are compatible. Two schemes are compatible if and only if their corresponding attributes are compatible.

If α is an attribute of the scheme of a structured attribute β, we say that β is the *parent attribute* of α.

We say that a structured attribute β is an *ancestor* of an attribute α, if β is the parent attribute of α or if β is an ancestor of the parent attribute of α.

We say that an attribute *occurs* in a scheme if it is an attribute of that scheme or if it occurs in the scheme of a structured attribute. We say that a list of attributes occurs in a scheme if all its attributes are attributes of that scheme or if they occur in the scheme with the same parent attribute.

The *level* of an identifier in a scheme is the number of bracket-pairs that surround the identifier in the scheme. The level of an attribute is equal to the level of its name. The level of a list of attributes is equal to the level of its attributes.

(A, B, C) is a scheme, all identifiers of which are at level 1. $(A_1, B_1, C_1(D_2, E_2(F_3)))$ is a scheme where the indices indicate the level of each identifier.

1.2 Instances of a Relation Scheme

Let (λ) be the scheme $(\alpha_1, ..., \alpha_n)$, where α_1 stands for an attribute, either atomic or structured. The *set of instances* of (λ), denoted by $Inst((\lambda))$, is the set

$$Inst((\lambda)) = \{s | s \text{ is a finite subset of } values(\alpha_1) \bowtie \cdots \bowtie values(\alpha_n)\}$$

where $values(A)$ is the set of the natural numbers if A is an atomic attribute and $values(A(\lambda))$ $= Inst((\lambda))$ otherwise. The elements of an instance of (λ) are called *tuples* over the scheme (λ).

Remark that for simplicity we assume that all atomic attributes have the same values-set.

$<4, 5, 4>$ is a tuple of the instance

A	B	C
1	2	3
4	5	4
3	2	4

and $<1,3,\{<5,\{<2>\}>,<3,\{<2>,<4>,<5>\}>\}>$ is a tuple of the instance

$$\begin{array}{ccc}
A_1 & B_1 & C_1(D_2,E_2(F_3)) \\
\hline
1 & 2 & \left\{\begin{array}{cc} 3 & \left\{\begin{array}{c}4\\5\end{array}\right\} \\ 6 & \left\{\begin{array}{c}7\\8\\9\end{array}\right\} \end{array}\right\} \\
1 & 3 & \left\{\begin{array}{cc} 5 & \left\{\begin{array}{c}2\end{array}\right\} \\ 3 & \left\{\begin{array}{c}2\\4\\5\end{array}\right\} \end{array}\right\}
\end{array}$$

1.3 Operators in the Nested Algebra

We define the operators in the nested algebra similar to the one introduced and used in [HP], [PA], [PVG], [RKS], [S], [SPS], [VG], [HS], [OOM] and [TF]. The algebra consists of 8 operators which are called NA-operators, and which are defined as follows :

- The *union operator* $\cup$: let $s_1, s_2 \in Inst((\lambda))$. Then $s_1 \cup s_2$ is the set-theoretic union of s_1 and s_2 and is an instance of the scheme (λ).

- The *difference operator* : let $s_1, s_2 \in Inst((\lambda))$. Then $s_1 - s_2$ is the set-theoretic difference of s_1 and s_2 and is an instance of the scheme (λ).

- The *join operator* $\bowtie$: let $s_1 \in Inst((\lambda_1))$ and $s_2 \in Inst((\lambda_2))$, with (λ_1) and (λ_2) schemes without common identifiers. Then $s_1 \bowtie s_2$ is the (standard) cartesian product of s_1 and s_2 and is an instance of the scheme (λ_1, λ_2).

- The *renaming operator* ρ : let $s \in Inst((\lambda))$, with (λ) a scheme, having an attribute with name A. Let B be an identifier not occurring in (λ). Then $\rho_{A \to B}(s)$ is the (standard) renaming of A by B in s and it is an instance of the scheme obtained from (λ) by replacing A with B.

- The *selection operator* σ : let $s \in Inst((\lambda))$, with (λ) a scheme, having two compatible attributes α and β. Then $\sigma_{\alpha=\beta}(s)$ is the subset of s consisting of all the tuples with equal α and β components and it is an instance of the scheme (λ).

- The *projection operator* π : let $s \in Inst((\lambda))$, with (λ) a scheme, having attributes $\alpha_1, ..., \alpha_k$. Then $\pi_{\alpha_1,...,\alpha_k}(s)$ is the (standard) projection on the attributes α_1 through α_k and it is an instance of the scheme $(\alpha_1, ..., \alpha_k)$. If t is a tuple of s, we denote its projection on $\alpha_1, \ldots, \alpha_k$ by $t|_{\alpha_1,...,\alpha_k}$.

- The *nest operator* ν : let $s \in Inst((\lambda))$, with (λ) the scheme (λ_1, λ_2), where λ_2 is not empty and let A be an identifier not occurring in (λ). Then $\nu_{\lambda_2;A}(s)$ is an instance of the scheme $(\lambda_1, A(\lambda_2))$. If λ_1 is not empty, $\nu_{\lambda_2;A}(s)$ is the set of the elements $<t_1, a>$, where t_1 is a tuple over (λ_1) and where a is the set of the elements t_2, such that $<t_1, t_2>$ is in s. If λ_1 is empty, $\nu_{\lambda_2;A}(s)$ is the set that contains only one element, namely s. Notice that in this definition we have made the notational simplification that λ_2 is an end sequence of λ.

- The *unnest operator* μ : let $s \in Inst((\lambda))$, with (λ) the scheme $(\lambda_1, A(\lambda_2))$. Then $\mu_A(s)$ is an instance of the scheme (λ_1, λ_2) and $\mu_A(s)$ is the set of elements $< t_1, t_2 >$ such that there is an element $< t_1, a >$ in s with $t_2 \in a$. Notice the notational simplification that $A(\lambda_2)$ is an end sequence of λ.

We will sometimes use the notation $\rho_{A_1,\ldots,A_k \to B_1,\ldots,B_k}$ as the abbreviation of $\rho_{A_1 \to B_1} \cdots \rho_{A_k \to B_k}$ and $\sigma_{\alpha_1,\ldots,\alpha_k = \beta_1,\ldots,\beta_k}$ as the abbreviation of $\sigma_{\alpha_1 = \beta_1} \cdots \sigma_{\alpha_k = \beta_k}$. We also write attributes instead of attribute names or attribute names instead of attributes if possible.

Algebraic expressions of the nested algebra, called *NA-expressions*, are defined in the usual way.

Let r be an instance of the first scheme above and s an instance of the second one : $\sigma_{D=E}(\nu_{B;D}(\nu_{C;E}(r)))$ and $\mu_{E_2}(\mu_{C_1}(\pi_{C_1}(s)))$ are two NA-expressions.

We say that an operator on nested relations is *NA-expressible* if and only if there is an NA-expression that is defined for the same operands as the operator and that, for every operand, yields the same result as the operator. Furthermore, this expression may depend on the schemes but not on the instances of the operands.

2 NA-operators are independent

In this section, we will prove that the eight NA-operators, as defined in section 1, are independent. To prove this, it suffices that each operator is not expressible by the others. Therefore, we describe for each operator a property that is violated by the operator, but that is preserved by each of the seven other operators.

For most operators our assertion is rather obvious. In these cases we will restrict ourselves to an informal proof.

Definition 2.1

We define the *set of leaves* of a scheme, as the set of all the atomic attributes that occur in the scheme. So, if (λ) is a scheme, we have :

$$\mathcal{SL}((\lambda)) = \{A \mid A \text{ is an atomic attribute that occurs in } (\lambda)\}$$

□

projection : Let us consider the number of leaves that occur in a scheme. Since the projection is the only NA-operator which can *lower* this number, it is clear that the projection cannot be expressed by the seven other operators.

join : Analogous to the projection, the join operator is the only NA-operator which can *augment* the number of leaves in a scheme. Consequently, the join operator cannot be removed from the algebra without loss of expressibility.

In order to handle the nest and the unnest operators, we need to introduce some new definitions.

Definition 2.2

We define the *depth* of a scheme and of its corresponding instance to be the maximum level of all its identifiers.
Notation : $\mathcal{DEP}((\lambda)) = \mathcal{DEP}(s)$ if $s \in Inst((\lambda))$.

□

nest : The nest and the join are the only operators capable to increase the depth of its argument(s). However, if only one instance s, having depth l, is available, it is obvious that any expression, which does not make use of the nest operator, cannot create an instance out of s, having a depth greater than l. So, the nest operator positively adds expressibility to our nested algebra.

unnest : Let us consider the instance s such that :

$$\begin{array}{l} s \in Inst((A(\lambda))) \\ \mathcal{DEP}(s) = \mathcal{DEP}((A(\lambda))) = l > 1 \end{array}$$

Suppose this is the only instance available. Using the unnest operator we can decrease the depth of s to $l - 1$. Any NA-expression which does not use the unnest operator, would result in an instance with a depth of at least l. Consequently, the unnest operator cannot be expressed by the seven other operators.

renaming : Since "renaming" is the only operator that can rename atomic attributes, it cannot be expressed by the seven other operators.

The next operator we will handle is "difference", for which an additional definition is required.

Definition 2.3

Let $(\lambda) = (\alpha_1, \ldots, \alpha_k)$ be a scheme, a and b atomic values.
Then we say that an instance $s \in Inst((\lambda))$ is *clean with respect to the ordered pair* (a, b) *over scheme* (λ), if and only if

1. $\forall t \in s, \forall \alpha_i$ *atomic attribute* : $t\,|_{\alpha_i} = a \Rightarrow \exists t' \in s : t'\,|_{\alpha_i} = b$ and
   $\forall j \neq i : t'\,|_{\alpha_j} = t\,|_{\alpha_j}$

2. $\forall t \in s, \forall \alpha_l$ *structured attribute with scheme* (λ') : $t\,|_{\alpha_l}$ is clean with respect to (a, b) over scheme (λ')

□

difference : It is easy to check that the operators π, $\cup$, ρ, $\bowtie$, ν, μ, and σ produce clean instances with respect to (a, b), when applied on clean instances with respect to the same pair. Now, let us consider the instances $s = \{a, b\}$ and $s' = \{b\} \in Inst((A))$ which are both clean with respect to (a, b) over (A). Clearly, $s - s' = \{a\}$ is not clean with respect to (a, b), which proves the independency of the difference operator.

Definition 2.4

Let (λ) be a scheme. We define the function $\mathcal{ADOM}$ such that $\mathcal{ADOM}(A,s)$ is the set of all atomic values of the atomic attribute A occurring in s. We call $\mathcal{ADOM}(A,s)$ the *active domain* of attribute A in instance s.

□

union : Let $(\lambda) = (A_1, A_2)$ be a scheme with A_1 and A_2 atomic attributes and $s \in \mathit{Inst}((\lambda))$ defined by : $s = \{<1,2>\}$. So, $\mathcal{ADOM}(A_1,s) = \{1\}$ and $\mathcal{ADOM}(A_2,s) = \{2\}$. With the aid of the union operator we can create an instance s' with both 1 and 2 as value of one attribute, say A_3.

$$s' = \rho_{A_1 \rightarrow A_3}(\pi_{A_1}(s)) \cup \rho_{A_2 \rightarrow A_3}(\pi_{A_2}(s)).$$

So s' is the instance of scheme (A_3) equal to $\{<1>,<2>\}$.
Hence, the result is an instance with an active domain that is larger than the original active domains. This is not possible without using the union operator. Consequently, the union operator cannot be expressed by the seven other operators.

The last operator we will have to deal with is the selection operator.

Definition 2.5

Let (λ) be a scheme, $s \in \mathit{Inst}((\lambda))$ and θ a set of atomic values.
We define :

s is *strict complete with respect to θ*

$\Longleftrightarrow$

$s = s_\theta$ with $s_\theta = \{t \mid$ t tuple over the scheme of s and
$\forall A$ atomic attributes of the scheme of s : $t|_A \in \theta$ and
$\forall \alpha$ structured attributes of the scheme of s : $t|_\alpha$ is strict complete with respect to $\theta\}$

s is *complete with respect to θ*

$\Longleftrightarrow$

s is strict complete with respect to θ or $s = \emptyset$

□

The following are examples of complete instances with respect to $\{a,b\}$:

A
a
b

A

A	B
a	a
a	b
b	a
b	b

A	$B(C)$
a	$\{\begin{smallmatrix}a\\b\end{smallmatrix}\}$
b	$\{\begin{smallmatrix}a\\b\end{smallmatrix}\}$

$A(B)$
$\{\begin{smallmatrix}a\\b\end{smallmatrix}\}$

selection : The reader is invited to check that the operators π, $\cup$, $-$, ρ, $\bowtie$, ν, and μ produce complete instances with respect to θ, when applied on complete instances with respect to θ. That this does not hold for the selection is illustrated by the following example :

Let s be the instance

A	B
a	a
a	b
b	a
b	b

It holds that s is complete with respect to $\{a, b\}$, but $\sigma_{A=B}(s)$ is not complete with respect to $\{a, b\}$!

As a consequence, using the selection operator allows us to express manipulations that are not expressible without it.

3 Expressibility of NA-operators at all levels

In this section we will first define the application of an NA-operator at a higher level and then we will prove that such an application is NA-expressible.

3.1 Definitions

The attributes occurring in the definitions of the NA-operators are called the *parameter attributes* of the NA-operators.
If the unary NA-operators $(\sigma, \pi, \rho, \nu, \mu)$ are applied according to their definitions, then we say that they are applied at level 1, because all parameter attributes are attributes at level 1. Since there appear no attributes in the definitions of the binary NA-operators $(\cup, -, \bowtie)$, we will say that they are applied at level 0.

Suppose $k \geq 1$, $s \in Inst((\lambda))$. Let $A(\lambda_2)$ and $B(\lambda_3)$ be compatible structured attributes at level k of a list λ_1 occurring in λ, and let $C(\lambda_4)$ be a structured attribute compatible with $A(\lambda_2)$ (and hence with $B(\lambda_3)$) and which has no identifiers in common with λ.
Then we say that we apply the union $A \cup B$ (resp. difference $A - B$) at level k on instance s , denoted by $[C(\lambda_4) := A \cup B](s)$ (resp. $[C(\lambda_4) := A - B](s)$), if :

- $[C(\lambda_4) := A \cup B](s)$ (resp. $[C(\lambda_4) := A - B](s)$) is an instance of the scheme obtained out of (λ) by adding to λ_1 the new attribute $C(\lambda_4)$
- $[C(\lambda_4) := A \cup B](s)$ (resp. $[C(\lambda_4) := A - B](s)$) equals the instance obtained by adding to each tuple, of every λ_1-value in s , the $C(\lambda_4)$-value that is the union (resp. difference) of the $A(\lambda_2)$-value and the $B(\lambda_3)$-value.

In the same way, we can define the application of the join at level $k \geq 1$:
Suppose $s \in Inst((\lambda))$. Let $A(\lambda_2)$ and $B(\lambda_3)$ be structured attributes at level k of a list λ_1 occurring in λ, and let $C(\lambda_4)$ be an attribute such that (λ_4) is compatible with (λ_2, λ_3) and such that $C(\lambda_4)$ has no identifiers in common with λ.
Then we say that we apply the join $A \bowtie B$ at level k on instance s , denoted by $[C(\lambda_4) := A \bowtie B](s)$, if :

- $[C(\lambda_4) := A \bowtie B](s)$ is an instance of the scheme obtained by adding to λ_1 the new attribute $C(\lambda_4)$

- $[C(\lambda_4) := A \bowtie B](s)$ equals the instance obtained by adding to each tuple of every λ_1-value in s, the $C(\lambda_4)$-value that is the cartesian product of the $A(\lambda_2)$-value and the $B(\lambda_3)$-value.

Suppose $k \geq 2$, $s \in Inst((\lambda))$ and let α and β be compatible attributes at level k in (λ) having the same parent attribute $Z(\lambda')$,
Then we say that we apply the selection $\alpha = \beta$ at level k on s, denoted by $\sigma_{\alpha=\beta}(s)$, if :

- $\sigma_{\alpha=\beta}(s)$ is an instance of the scheme (λ)
- $\sigma_{\alpha=\beta}(s)$ equals the instance obtained by replacing each $Z(\lambda')$-value in s by the value that is the selection $\alpha = \beta$ of the $Z(\lambda')$-value.

In the same way, the other unary NA-operators are applied at level $k \geq 2$.
We will confine ourselves to the notations :

- $\pi_{\alpha_{i_1},\ldots,\alpha_{i_l}}(s)$ with $\alpha_{i_1},\ldots,\alpha_{i_l}$ attributes at level k, having the same parent attribute
- $\rho_{A\rightarrow B}(s)$ with A the name of an attribute at level k and B an identifier not occurring in the scheme of s
- $\nu_{\lambda;C}(s)$ with λ a list of attributes at level k and C an identifier not occuring in the scheme of s
- $\mu_C(s)$ with C the name of a structured attribute at level k

Hence, the notations for the unary operators applied at level k, do not differ from those at level 1. Note that in the application of the binary operators, we add a new structured attribute on level k, but in the application of the unary operators, we substitute a structured attribute on level k.

The following examples demonstrate for an instance s, what the resulting scheme looks like, when an NA-operator is applied at a higher level on the instance s.
Suppose $s \in Inst((A_1, B_1(C_2(D_3), E_2, F_2(G_3)), H_1(I_2, J_2)))$ then

- the union $C_2 \cup F_2$ (an application at level 2), $[K_2(L_3) := C_2 \cup F_2](s)$, is an instance of the scheme
$$(A_1, B_1(C_2(D_3), E_2, F_2(G_3), K_2(L_3)), H_1(I_2, J_2))$$
- the join $B_1 \bowtie H_1$ (an application at level 1), $[M_1(N_2(O_3), P_2, Q_2(R_3), S_2, T_2) := B_1 \bowtie H_1](s)$, is an instance of the scheme
$$(A_1, B_1(C_2(D_3), E_2, F_2(G_3)), H_1(I_2, J_2), M_1(N_2(O_3), P_2, Q_2(R_3), S_2, T_2))$$
- the projection π_{F_2} (an application at level 2), $\pi_{F_2}(s)$, is an instance of the scheme
$$(A_1, B_1(F_2(G_3)), H_1(I_2, J_2))$$

3.2 The copy operator and the empty operator

In section 3.3 we will show that every application of an NA-operator at level $k \geq 1$, is NA-expressible. To this end we will frequently need an operator (*copy*) which duplicates an attribute.

Definition 3.1

Suppose $s \in Inst((\lambda))$. Let α be an attribute of λ and let β be an attribute compatible with α such that all identifiers occurring in β do not occur in λ.
Then we define $copy_{\alpha \to \beta}(s)$ such that :

- $copy_{\alpha \to \beta}(s)$ is an instance of the scheme (λ, β)
- $copy_{\alpha \to \beta}(s)$ is the instance obtained by adding to each tuple t of s the value $t|_\beta$, with $t|_\beta = t|_\alpha$.

We say that *copy* is applied at level 1 since α is an attribute of λ at level 1.

□

Theorem 3.1 *The application of copy at level* 1 *is NA-expressible.*

Proof
Suppose $s \in Inst((\lambda))$. Let α be an attribute of λ and let β be an attribute compatible with α such that all identifiers occurring in β do not occur in λ.
Then $copy_{\alpha \to \beta}(s)$ equals $\sigma_{\alpha=\beta}(s \bowtie \rho_{\alpha \to \beta}(\pi_\alpha(s)))$.

□

Definition 3.2

Suppose $k \geq 2$ and $s \in Inst((\lambda))$. Let $Z(\lambda_1)$ be a structured attribute occurring in λ at level $k-1$, let α be an attribute of λ_1 and let β be an attribute compatible with α such that all identifiers occurring in β do not occur in λ.
Then we say that we apply copy $\alpha \to \beta$ at level k on instance s, denoted $copy_{\alpha \to \beta}(s)$, if :

- $copy_{\alpha \to \beta}(s)$ is an instance of the scheme obtained by replacing in (λ) the attribute $Z(\lambda_1)$ by $Z(\lambda_1, \beta)$
- $copy_{\alpha \to \beta}(s)$ is the instance obtained by replacing each tuple t over $Z(\lambda_1)$ in s, by the tuple t' over $Z(\lambda_1, \beta)$, with $t'|_{\lambda_1} = t$ and $t'|_\beta = t|_\alpha$.

□

Another kind of operator we will frequently need, is the *empty* operator, which creates an instance with the empty set as its only tuple.

Definition 3.3

Let (λ) be a scheme, $s \in Inst((\lambda))$ and A an identifier that does not occur in (λ).
Then the scheme of $emp_A(s)$ is $(A(\lambda))$, and $emp_A(s) = \{<\emptyset>\}$.

□

Theorem 3.2 *The emp operator is NA-expressible.*

Proof
Suppose $s \in Inst((\lambda))$ and A an identifier not occurring in λ.
Then $emp_A(s)$ equals $\nu_{\lambda;A}(s - s)$.

□

3.3 How to express an NA-operator at level k ?

In this section we will show how the application of each NA-operator at level $k \geq 1$ can be expressed. We will introduce an induction technique which can be used for all eight NA-operators and which assumes that the application at level 1 is NA-expressible for each NA-operator.

3.3.1 The NA-operators at level 1

We only have to prove NA-expressibility at level 1 for the binary operators, since the application of the unary operators at level 1 corresponds to the definitions of section 1.

Lemma 3.1 *The application of the union operator at level* 1 *is NA-expressible.*

Proof

We show for a specific example how the application of the union at level 1 can be expressed in the nested algebra. This example can be generalized in a natural way.

Let $(\lambda) = (A, B(E), C(F))$ be a scheme with A, E and F atomic attributes and let $s \in Inst((\lambda))$.

We want to produce the instance $s' = [D(G) := B \cup C](s)$.

From s we construct :

$$\begin{aligned} s_1 &= copy_{B(E) \to D(G)}(s) \\ s_2 &= copy_{C(F) \to D(G)}(s) \\ s_3 &= s_1 \cup s_2 \\ s_4 &= \sigma_{B=C}(s_3) \\ s_5 &= s_3 - s_4 \end{aligned}$$

Now s_5 only holds tuples, which have at least one of the values of $B(E)$ or $C(F)$ not empty. Therefore we can unnest the $D(G)$-attribute without loosing the A-value.

$$s_6 = \mu_D(s_5)$$

If we nest the G-attribute again, we obtain as $D(G)$-values, the sets that are the union of the corresponding $B(E)$- and $C(F)$-value.

$$s_7 = \nu_{G;D}(s_6)$$

Now we only have to unite s_7 and s_4.

$$s_8 = s_7 \cup s_4$$

It is obvious that s_8 now equals s'.

□

Note : As one can see in the proof above, tuples that have the empty set as value of a structured attribute, require a special treatment. If not, these tuples would disappear after unnesting the structured attribute. A special treatment is needed for all operators at higher levels.

Lemma 3.2 *The application of the difference at level* 1 *is NA-expressible.*

Proof

As for the union operator, we will here too prove the expressibility by means of a specific example that can easily be generalized.

Let $(\lambda) = (A, B(E), C(F))$ be a scheme with A, E and F atomic attributes. Let $s \in Inst((\lambda))$. We want to produce the instance $s' = [D(G) := B - C](s)$.

Consider the following sequence of expressions :

$$\begin{aligned} s_1 &= \mu_D(copy_{B(E)\to D(G)}(s)) \\ s_2 &= \mu_D(copy_{C(F)\to D(G)}(s)) \\ s_3 &= s_1 - s_2 \\ s_4 &= \nu_{G;D}(s_3) \end{aligned}$$

s_4 only contains tuples that do not have the empty set as value of $B(E)$, since those tuples have disappeared in the first unnest and should therefore be treated differently.

$$\begin{aligned} s_5 &= \pi_{A,B,C}(s_3) \\ s_6 &= s - s_5 \end{aligned}$$

s_6 is the instance which contains the tuples that have the empty set as $B(E)$-value. We have to add those tuples with an empty $D(G)$-value.

$$\begin{aligned} s_7 &= emp_D(\pi_G(s_3)) \\ s_8 &= s_6 \bowtie s_7 \\ s_9 &= s_4 \cup s_8 \end{aligned}$$

Now s_9 equals s'.

□

Lemma 3.3 *The application of the join at level* 1 *is NA-expressible.*

Proof

Let $(\lambda) = (A, B(E), C(F))$ be a scheme with A, E and F atomic attributes. Let $s \in Inst((\lambda))$. We want to produce the instance $s' = [D(G, H) := B \bowtie C](s)$.

Consider the following sequence of expressions :

$$\begin{aligned} s_1 &= \mu_{B'}(copy_{B(E)\to B'(G)}(s)) \\ s_2 &= \mu_{C'}(copy_{C(F)\to C'(H)}(s)) \\ s_3 &= \rho_{A,B(E),C(F)\to A',B'(E'),C'(F')}(s_1) \\ s_4 &= s_3 \bowtie s_2 \\ s_5 &= \sigma_{A,B,C=A',B',C'}(s_4) \\ s_6 &= \pi_{A,B,C,G,H}(s_5) \\ s_7 &= \nu_{G,H;D}(s_6) \end{aligned}$$

s_7 only contains tuples that do not have the empty set as value of $B(E)$ or $C(F)$, since such tuples have disappeared in the unnests. Therefore, we have to treat those tuples in another way.

$$s_8 = s - \pi_{A,B,C}(s_7)$$

Now s_8 is the instance containing the tuples that have the empty set as $B(E)$- or $C(F)$-value. We must add these tuples with an empty $D(G,H)$-value.

$$s_9 = s_8 \bowtie emp_D(\mu_D(\pi_D(s_7)))$$
$$s_{10} = s_7 \cup s_9$$

Now s_{10} equals s'.

□

3.3.2 The NA-operators at level $k > 1$

For the union operator, we will now give the proof of the induction step from $k-1$ to k. The reader is invited to check the proof for the other seven operators.

Theorem 3.3 *The application of the union at level $k > 1$ is expressible.*

Proof

The application of the union at level 1 is expressible as shown in lemma 3.1.
induction hypothesis : The application of the union at level $k-1$ is expressible.
Suppose $s \in Inst((\lambda))$. Let $A(\lambda_1)$ and $B(\lambda_2)$ be compatible structured attributes occurring in λ at level k , having the same parent attribute $Z(\lambda_3)$. Let $Y(\lambda_4)$ be the ancestor of $A(\lambda_1)$ (and hence of $B(\lambda_2)$) at level 1 (if $k=2$ then $Y(\lambda_4)$ equals $Z(\lambda_3)$).
We want to produce the instance $[C(\lambda_5) := A \cup B](s)$.
Suppose λ_6 is the list obtained out of λ_4 by adding to λ_3 the new attribute $C(\lambda_5)$ and suppose λ_7 is the list obtained by replacing in λ, the list λ_4 by λ_6.
We can obtain the instance $[C(\lambda_5) := A \cup B](s)$ by :

$$s_1 = copy_{Y(\lambda_4)\to Y'(\lambda_4')}(s)$$
$$s_2 = \mu_Y(s_1)$$
$$s_3 = [C(\lambda_5) := A \cup B](s_2) \quad \text{(this is an application at level } k-1\text{ !)}$$
$$s_4 = \nu_{\lambda_6;Y}(s_3)$$
$$s_5 = \pi_{\lambda_7}(s_4)$$

s_5 only contains tuples that do not have the empty set as $Y(\lambda_6)$-value, since those tuples have disappeared unnesting $Y(\lambda_4)$.

$$s_6 = \rho_{\lambda_4\to\lambda_4'}(emp_{Y'}(\pi_{\lambda_4}(s_2)))$$
$$s_7 = s \bowtie s_6$$
$$s_8 = \pi_\lambda(\sigma_{Y=Y'}(s_7))$$

Now, s_8 contains the tuples of s with the empty set as $Y(\lambda_4)$-value. We have to add these tuples with the empty set as $Y(\lambda_6)$-value.

$$s_9 = \pi_{\lambda_7}(\rho_{Y(\lambda_4)\to Y'(\lambda_4')}(s_8) \bowtie emp_Y(\pi_{\lambda_6}(s_3)))$$
$$s_{10} = s_5 \cup s_9$$

Now s_{10} equals $[C(\lambda_5) := A \cup B](s)$.

□

We will demonstrate how the scheme of an instance is transformed in each step of the technique on a particular example.

Suppose $s \in Inst((A_1, B_1(C_2(D_3), E_2, F_2(G_3))))$ and we want to produce the instance $s_{10} = [H_2(I_3) := C_2 \cup F_2](s)$.

Then the schemes of the succeeding instances $s, s_1, \dots, s_{10}$ are as follows :

instance	*scheme*
s	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3)))$
s_1	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3)), B'_1(C'_2(D'_3), E'_2, F'_2(G'_3)))$
s_2	$(A_1, C_2(D_3), E_2, F_2(G_3), B'_1(C'_2(D'_3), E'_2, F'_2(G'_3)))$
s_3	$(A_1, C_2(D_3), E_2, F_2(G_3), B'_1(C'_2(D'_3), E'_2, F'_2(G'_3)), H_2(I_3))$
s_4	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3), H_2(I_3)), B'_1(C'_2(D'_3), E'_2, F'_2(G'_3)))$
s_5	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3), H_2(I_3)))$
s_6	$(B'_1(C'_2(D'_3), E'_2, F'_2(G'_3)))$
s_7	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3)), B'_1(C'_2(D'_3), E'_2, F'_2(G'_3)))$
s_8	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3)))$
s_9	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3), H_2(I_3)))$
s_{10}	$(A_1, B_1(C_2(D_3), E_2, F_2(G_3), H_2(I_3)))$

4 General selections and dependencies

In this section we will first prove that, besides the *equality* selection $\alpha = \beta$, many other selections are NA-expressible. Next, we will show that functional and join dependencies are NA-expressible.

4.1 Other selections

Definition 4.1

Let $(\lambda) = (\alpha_1, \dots, \alpha_n)$ be a scheme, $i, j \in \{1, \dots, n\}$, $i \neq j$, and $s \in Inst((\lambda))$.

- If α_i and α_j are compatible attributes then we define the *not-equal selection*

  $$\sigma_{\alpha_i \neq \alpha_j}(s) = \{x \in s \mid x|_{\alpha_i} \neq x|_{\alpha_j}\}$$

- If α_i is a structured attribute, then we define the *(not-)empty selection*

  $$\sigma_{\alpha_i = \emptyset}(s) = \{\, x \in s \mid x|_{\alpha_i} = \emptyset\}$$
  $$\sigma_{\alpha_i \neq \emptyset}(s) = \{\, x \in s \mid x|_{\alpha_i} \neq \emptyset\}$$

- If α_j is a structured attribute $A_j(B_j)$ such that B_j and $\alpha_i = A_i$ are compatible, then we define the *(not-)element selection*

  $$\sigma_{A_i \in \alpha_j}(s) = \{\, x \in s \mid x|_{A_i} \in x|_{\alpha_j}\}$$
  $$\sigma_{A_i \notin \alpha_j}(s) = \{\, x \in s \mid x|_{A_i} \notin x|_{\alpha_j}\}$$

- If α_i and α_j are compatible structured attributes, then we define the *(not-)subset selection*

$$\sigma_{\alpha_i \subset \alpha_j}(s) = \{ x \in s \mid x|_{\alpha_i} \subset x|_{\alpha_j} \}$$
$$\sigma_{\alpha_i \not\subset \alpha_j}(s) = \{ x \in s \mid x|_{\alpha_i} \not\subset x|_{\alpha_j} \}$$

Since the attributes occurring in the definitions are attributes at level 1, we say that these selections are applied at level 1. We can extend the definition to attributes at level $k > 1$ (having the same parent attribute), such that we can apply these selections at level k.

□

Theorem 4.1 *The applications of the selections not-equal, (not-)empty, (not-)element and (not-)subset at level* 1 *are NA-expressible.*

Proof

Let $(\lambda) = (\alpha_1, \ldots, \alpha_n)$ be a scheme, $i, j \in \{1, \ldots, n\}$, $i \neq j$, $s \in Inst((\lambda))$, and A_i the name of α_i. In the following, we assume that the conditions on the attributes α_i and α_j as stated in definition 4.1, are fulfilled.
The following equalities hold :

- $\sigma_{A_i \neq A_j}(s) = s - \sigma_{A_i = A_j}(s)$
- $\sigma_{A_i = \emptyset}(s) = \pi_\lambda(\sigma_{A_i = A_i''}([\alpha_i'' := \alpha_i - \alpha_i'](copy_{\alpha_i \to \alpha_i'}(s))))$
- $\sigma_{A_i \neq \emptyset}(s) = s - \sigma_{A_i = \emptyset}(s)$
- $\sigma_{A_i \in A_j}(s) = \pi_\lambda(\sigma_{A_i = B_j'}(\mu_{A_j'}(copy_{A_j(B_j) \to A_j'(B_j')}(s))))$
- $\sigma_{A_i \notin A_j}(s) = s - \sigma_{A_i \in A_j}(s)$
- $\sigma_{A_i \subset A_j}(s) = \pi_\lambda(\sigma_{A_i' = \emptyset}([\alpha_i' := \alpha_i - \alpha_j](s)))$
- $\sigma_{A_i \not\subset A_j}(s) = s - \sigma_{A_i \subset A_j}(s)$

□

Theorem 4.2 *The applications of the selections introduced in this section at level $k > 1$ are NA-expressible.*

Proof

A similar induction technique is used as in the proof of theorem 3.3.
Let $s \in Inst((\lambda))$, and let $Y(\lambda_1)$ be an attribute of λ such that $Y(\lambda_1)$ is the ancestor at level 1 of the attributes in the selection. Suppose the attributes in the selection are attributes occurring in (λ) at level k.
Consider the following sequence of expressions :

$s_1 = copy_{Y(\lambda_1) \to Y'(\lambda_1')}(s)$
$s_2 = \mu_Y(s_1)$
s_3 is obtained from s_2 by applying the selection at level $k - 1$
$s_4 = \nu_{\lambda_1; Y}(s_3)$
$s_5 = \pi_\lambda(s_4)$
$s_6 = \rho_{\lambda_1 \to \lambda_1'}(empy_{Y'}(\pi_{\lambda_1}(s_3)))$
$s_7 = s \bowtie s_6$
$s_8 = \pi_\lambda(\sigma_{Y = Y'}(s_7))$
$s_9 = s_5 \cup s_8$

Now s_9 equals the instance obtained by applying the selection.

□

4.2 Dependencies

We want to introduce an operator which checks whether or not a specific dependency holds in a given instance $s \neq \emptyset$. Since we only consider operators which have (a) relation(s) as argument(s) and a relation as result, we will define an operator which takes s as its argument, and with result s if the dependency holds in s, and result the empty instance over the scheme of s, if the dependency does not hold in s.

4.2.1 Functional dependency

Definition 4.2

Let $s \in \mathit{Inst}((\lambda))$, λ_1, λ_2, λ_3, λ_4, λ_5 and λ_6 attribute lists such that $\lambda = \lambda_3, \lambda_1, \lambda_4 = \lambda_5, \lambda_2, \lambda_6$.
Then we say that the *functional dependency* $\lambda_1 \rightarrow \lambda_2$ *holds in* s, if and only if

$$\forall t_1, t_2 \in s : t_1 |_{\lambda_1} = t_2 |_{\lambda_1} \Rightarrow t_1 |_{\lambda_2} = t_2 |_{\lambda_2}.$$

Since λ_1 and λ_2 are lists of attributes at level 1, we call it a functional dependency at level 1.
□

Note : It is obvious that if λ_2 is the empty list, the functional dependency $\lambda_1 \rightarrow \lambda_2$ holds in every instance s.

We will now prove that it is possible in the nested algebra, to check whether or not a given functional dependency (at level 1) holds in a given instance s. As mentioned above, we will do this by defining an operator FD, such that $FD_{\lambda_1 \rightarrow \lambda_2}(s) = s$ if the functional dependency $\lambda_1 \rightarrow \lambda_2$ holds in s, and $FD_{\lambda_1 \rightarrow \lambda_2}(s) = \emptyset_\lambda$ [1] if the functional dependency does not hold in s.

Theorem 4.3 *The functional dependencies at level* 1 *are NA-expressible.*

Proof
Let $s \in \mathit{Inst}((\lambda))$, λ_1, λ_2, λ_3, λ_4, λ_5 and λ_6 attribute lists such that $\lambda = \lambda_3, \lambda_1, \lambda_4 = \lambda_5, \lambda_2, \lambda_6$.
If λ_2 is the empty list then $FD_{\lambda_1 \rightarrow \lambda_2}(s) = s$.
Let us assume λ_2 is a non-empty list.
Consider the following expressions :

$$\begin{aligned}
s_1 &= \rho_{\lambda \rightarrow \lambda'}(s) \\
s_2 &= s \bowtie s_1 \\
s_3 &= \sigma_{\lambda_1 = \lambda_1'}(s_2) \\
s_4 &= \sigma_{\lambda_2 \neq \lambda_2'}(s_3) \\
s_5 &= \pi_{\lambda'}(s_4) \\
s_6 &= s \bowtie (s_1 - s_5) \\
s_7 &= [\text{if } \pi_\lambda = \rho_{\lambda' \rightarrow \lambda}(\pi_{\lambda'}) \text{ then } \pi_\lambda \text{ else } \pi_\lambda - \pi_\lambda](s_6) \quad [2]
\end{aligned}$$

Then s_7 equals $FD_{\lambda_1 \rightarrow \lambda_2}(s)$.

□

[1] $\emptyset_\lambda$ is a notation for the empty instance over scheme (λ)

[2] In section 5 we will prove that such a selective expression is NA-expressible.

Definition 4.3

Let $k > 1$ and $s \in \mathit{Inst}((\lambda))$, $A(\lambda')$ a structured attribute occurring in λ at level $k-1$, and let λ'_1, λ'_2, λ'_3, λ'_4, λ'_5 and λ'_6 be attribute lists such that $\lambda' = \lambda'_3, \lambda'_1, \lambda'_4 = \lambda'_5, \lambda'_2, \lambda'_6$.
Then we say that the *functional dependency* $\lambda'_1 \rightarrow \lambda'_2$ *at level k holds in s*, if and only if the functional dependency holds for each $A(\lambda')$-value s' of s.

□

Note : Again, it is obvious that if λ'_2 is the empty list, the functional dependency $\lambda'_1 \rightarrow \lambda'_2$ holds in every instance s.

Clearly, the following theorem holds :

Theorem 4.4 *The functional dependencies at level $k > 1$ are expressible.*

□

4.2.2 Join dependency

Definition 4.4

Let $s_1 \in \mathit{Inst}((\lambda_1))$ and $s_2 \in \mathit{Inst}((\lambda_2))$ with $\lambda_1 = \alpha_1, \ldots, \alpha_k, \alpha_{k+1}, \ldots, \alpha_l$ and $\lambda_2 = \alpha_{k+1}, \ldots, \alpha_l, \alpha_{l+1}, \ldots, \alpha_m$. Then $s_1 \bowtie_n s_2$ is the (standard) natural join of s_1 and s_2 and it is an instance of the scheme $(\alpha_1, \ldots, \alpha_k, \ldots, \alpha_l, \ldots, \alpha_m)$. Notice that, in this definition, we have made a notational simplification with respect to the ordering of the attribute lists.

□

Theorem 4.5 *The natural join is NA-expressible.*

Proof
Suppose s_1 and s_2 are as in definition 4.4. Then we have :

$$s_1 \bowtie_n s_2 = \pi_{\alpha_1,\ldots,\alpha_k,\ldots,\alpha_l,\ldots,\alpha_m}(\sigma_{\alpha_{k+1},\ldots,\alpha_l=\alpha'_{k+1},\ldots,\alpha'_l}(s_1 \bowtie \rho_{\alpha_{k+1},\ldots,\alpha_l\rightarrow\alpha'_{k+1},\ldots,\alpha'_l}(s_2)))$$

□

Note : The definition and the proof can easily be extended to the application at level k, with k at least 1.

Definition 4.5

Let $s \in \mathit{Inst}((\lambda))$, $\lambda_1, \ldots, \lambda_l$ non-empty lists of attributes of λ such that each attribute of λ is also an attribute of at least one λ_i $(i \in \{1, \ldots, l\})$. Then we say that the *join dependency* $\lambda_1 \bowtie \cdots \bowtie \lambda_l$ *holds in s*, if and only if

$$s = (\pi_{\lambda_1}(s)) \bowtie_n \cdots \bowtie_n (\pi_{\lambda_l}(s)).$$

Since $\lambda_1, \ldots, \lambda_l$ are lists of attributes at level 1, we call it a join dependency at level 1.

□

Note : The case that $l = 1$ is trivial because this would cause λ_1 to be equal to λ (resp. λ'). In the following we will assume $l \geq 2$.

We will now prove that it is possible in the nested algebra to check whether or not a given join dependency at level 1 holds in a given instance s. In analogy with the functional dependency, we will do this by defining an operator JD, such that $JD_{\lambda_1 \bowtie \cdots \bowtie \lambda_n}(s) = s$ if the join dependency $\lambda_1 \bowtie \cdots \bowtie \lambda_n$ holds in s, and $JD_{\lambda_1 \bowtie \cdots \bowtie \lambda_n}(s) = \emptyset_\lambda$ if the join dependency does not hold in s.

Theorem 4.6 *The join dependencies at level* 1 *are NA-expressible.*

Proof

Let $s \in Inst((\lambda))$, $\lambda_1, \ldots, \lambda_l$ non-empty lists of attributes of λ such that each attribute of λ is also an attribute of at least one λ_i $(i \in \{1, \ldots, l\})$.
Consider the following expressions :

$$\begin{aligned} s_1 &= (\pi_{\lambda_1}(s)) \bowtie_n \cdots \bowtie_n (\pi_{\lambda_l}(s)) \\ s_2 &= s \bowtie \rho_{\lambda \to \lambda'}(s_1) \\ s_3 &= [\text{if } \pi_\lambda = \rho_{\lambda' \to \lambda}(\pi_{\lambda'}) \text{ then } \pi_\lambda \text{ else } \pi_\lambda - \pi_\lambda](s_2) \end{aligned}$$

Then s_3 equals $JD_{\lambda_1 \bowtie \cdots \bowtie \lambda_l}(s)$.

□

Definition 4.6

Let $k > 1$ and $s \in Inst((\lambda))$, $A(\lambda')$ a structured attribute occurring in λ at level $k-1$, $\lambda_1, \ldots, \lambda_l$ non-empty lists of attributes of λ' such that each attribute of λ' is also an attribute of at least one λ_i $(i \in \{1, \ldots, l\})$.
Then we say that the *join dependency* $\lambda_1 \bowtie \cdots \bowtie \lambda_l$ *at level* k *holds in* s if and only if the join dependency holds for each $A(\lambda')$-value s' of s.

□

Note : Again, the case that $l = 1$ is trivial and hereafter we will assume $l \geq 2$.

Clearly, the following theorem holds :

Theorem 4.7 *The join dependencies at level* $k > 1$ *are NA-expressible.*

□

5 Selective Expressions

In this section we consider the selective expressions, i.e. we will give the formal definition and we consider the NA-expressibility.

The application of a selective expression on some instance s results either in the application of some NA-expression E_1 on s or in the application of some NA-expression E_2 on s, depending on the evaluation of some condition C on s. Such a selective expression is written as

$$[\text{if } C \text{ then } E_1 \text{ else } E_2].$$

The motivation for studying the selective expression originates from the definition of the least fix point (lfp) operator [CH]. If we claim that an NA-operator corresponds to a single statement (in the context of programming), then the notion of the lfp operator corresponds to that of the repetitive statement. Intuitively the selective expression corresponds to the selective statement. Since in [GVG] it is proved that the lfp operator is not NA-expressible, it is interesting to verify whether the selective expression is NA-expressible.

Definition 5.1

If E and E' are NA-expressions, then $E = \emptyset, E = E', E \subseteq E'$ and $E \in E'$ are ***basic if-conditions***. The application of a basic *if*-condition C on an instance s results in a boolean value, that is recursively defined by :

- the application of $E = \emptyset$ on s, with E an NA-expression, results in $E(s) = \emptyset$;
- the application of $E\phi E'$ on s with $\phi \in (=, \subseteq)$ and E and E' NA-expressions such that the schemes of $E(s)$ and $E'(s)$ are the same, results in $E(s)\phi E'(s)$;
- the application of $E \in E'$ on s with E and E' NA-expressions such that if the scheme of $E(s)$ equals (λ) then the scheme of $E'(s)$ equals $(A(\lambda))$, results in $E(s) \in E'(s)$.

If-conditions are composed of basic if-conditions using $\wedge, \vee$ and $\neg$. The application of an if-condition C on an instance s is a boolean value, that is recursively defined by :

- if C is a basic if-condition, then the application of C is already defined;
- the application of $C' \wedge C''$ results in $C'(s) \wedge C''(s)$;
- the application of $C' \vee C''$ results in $C'(s) \vee C''(s)$;
- the application of $\neg\, C'$ results in $\neg\, C'(s)$.

□

Examples of basic if-conditions, that can be applied on instances of the scheme $(A, B, C, D, E(F))$, are

$$\pi_{A,B,D} = \emptyset;$$
$$\pi_A \cup \rho_{B\rightarrow A}(\pi_B) = \pi_A;$$
$$\pi_{A,B,C}(\sigma_{A=B}) \subseteq \rho_{D\rightarrow C}(\pi_{A,B,D}(\sigma_{A=D}));$$
$$\pi_A \in \rho_{F\rightarrow A}(\pi_E).$$

Definition 5.2

Let C be an if-condition and E_1 and E_2 NA-expressions such that for an instance s, $E_1(s)$ and $E_2(s)$ have the same scheme. Then [if C then E_1 else E_2] is called a *selective expression* and its application on an instance s is recursively defined by :

$$[\text{if } C \text{ then } E_1 \text{ else } E_2](s) = \begin{cases} E_1(s) \text{ if } C(s) \text{ holds,} \\ E_2(s) \text{ if } \neg C(s) \text{ holds.} \end{cases}$$

□

Two examples of selective expressions, applicable to instances of the scheme (A, B, C, D), are :

[if $\pi_A = \rho_{B\rightarrow A}(\pi_B)$
then $\sigma_{A=B}(\pi_{A,B,C})$ else $\rho_{D\rightarrow C}(\pi_{A,B,D})$];

[if $\pi_A = \rho_{B\rightarrow A}(\pi_B) \wedge \neg(\sigma_{C=D} = \emptyset)$
then $\sigma_{A=B}(\nu_{C;E}(\pi_{A,B,C}))$ else $\rho_{D\rightarrow C}(\nu_{D;E}(\pi_{A,B,D}))$].

Another example, applicable to instances of the scheme $(A, B(C), D)$, is :

$$\begin{array}{l}[\text{if } \pi_A \subseteq \rho_{C\to A}(\pi_C(\mu_B)) \wedge \pi_D = \pi_D(\sigma_{A=D}) \\ \text{then } \pi_{A,B} \text{ else } \rho_{D\to A}(\pi_{D,B})].\end{array}$$

So, a selective expression specifies two expressions, one of which is evaluated dependent on the evaluation of the if-condition. The next theorem considers the NA-expressibility of these expressions.

Theorem 5.1 *Every selective expression is NA-expressible.*

Proof

In order to prove that for every selective expression an equivalent NA-expression can be constructed, we first show that for selective expressions of the kind

$$[\text{if } E = \emptyset \text{ then } E_1 \text{ else } E_2]$$

an equivalent NA-expression can be constructed. Subsequently we show that other selective expressions can be reduced to selective expressions of the above kind.

Let E, E', E_1 and E_2 be NA-expressions.

Consider the selective expression

$$[\text{if } E = \emptyset \text{ then } E_1 \text{ else } E_2].$$

If λ is the scheme obtained by applying E_1 or E_2, and E' is a renaming of E such that the scheme obtained by applying E' has no attributes in common with λ, then the expression is equivalent with

(i) $$(E_1 - \pi_\lambda(E_1 \bowtie E')) \cup \pi_\lambda(E_2 \bowtie E').$$

Note that λ and the renaming depend on the scheme of the instance on which the selective expression is applied.

The equivalence is due to the fact that, if $E = \emptyset$ holds, then $E' = \emptyset$, hence $\pi_\lambda(E_i \bowtie E') = \emptyset$ (for $i = 1, 2$) and so the expression (i) equals E_1. On the other hand, if $E \neq \emptyset$, then $E' \neq \emptyset$, hence $\pi_\lambda(E_i \bowtie E') = E_i$ and so the expression (i) equals E_2.

The following equivalences prove that basic if-conditions of another kind can be reduced to the kind $E = \emptyset$:

$$\begin{array}{l} E = E' \equiv (E - E') \cup (E' - E) = \emptyset; \\ E \subseteq E' \equiv (E - E') = \emptyset; \\ E \in E' \equiv \nu_{\lambda;A}(E) \subseteq E', \end{array}$$

where $A(\lambda)$ is the scheme obtained by applying E'.

The following equivalences show that selective expressions with non-basic if-conditions are

NA-expressible : (let C and C' be if-conditions and E_1 and E_2 be NA-expressions)

[if $C \wedge C'$ then E_1 else E_2]
$\equiv$
[if C then E else E_2]
with $E \equiv$ [if C' then E_1 else E_2];

[if $C \vee C'$ then E_1 else E_2
$\equiv$
[if C then E_1 else E]
with $E \equiv$ [if C' then E_1 else E_2];

[if $\neg C$ then E_1 else E_2]
$\equiv$
[if C then E_2 else E_1].

This clearly proves that every selective expression is NA-expressible.

□

In the above definition of selective expressions the NA-expressions in selective expressions are applied at level 1. Therefore we say that these selective expressions are applied at level 1. Now we extend the definition in such a way that application at any level is possible.

Definition 5.3

Let E be a selective expression. Let s be an instance of scheme (λ), let $A(\lambda')$ be a structured attribute at level $k-1$ in (λ) with k at least 2. If E is such that for all $A(\lambda)$-values v in s $E(v)$ is well-defined, $E(s)$ is defined to be the instance obtained from s by replacing each $A(\lambda')$-value v in s by $E(v)$.
We call this the *application* of E *at level* k.

□

Note that in the definition we describe how the instance is changed, but, of course, the scheme must be changed correspondingly.
An example of an application of a selective expression at level 2 of scheme $(A, B(C(D), E(F), G, H), I)$ is :

[if $\pi_C \subseteq \rho_{E\rightarrow C}(\rho_{F\rightarrow D}(\pi_E))$
then π_G else $\rho_{H\rightarrow G}(\pi_H)$].

Again we can prove by induction that the application of selective expressions at level k, k at least 2, is NA-expressible.

Theorem 5.2 *Every application of a selective expression at level k, for k at least 2, is NA-expressible.*

Proof

We do not give the full proof here, but only an outline of it. First an equivalent expression is constructed for the application of a selective expression at level 2.

Then the same method, as used earlier, is used to construct, from an equivalent expression E' for the application of a selective expression E at level k, an NA-expression E'' equivalent with the application of E at level $k-1$, for k at least 2.

□

After extending the definition to allow for application at higher levels, we can generalize the notion even further such that application at multiple levels is possible.

Intuitively, application at multiple levels means that the expressions used in the condition of some selective expression do not need to be at the same level as the then- and else-expression. So, if the then- and else-expression manipulate some attribute value v at level k, then the expressions in the condition do not need to manipulate the same value v, but they are allowed to manipulate an attribute value v', as long as this value v' is uniquely corresponding to v. This implies that v and v' must be values of attributes α and α' such that the parent attribute of α' is an ancestor attribute of α.

First we define the argument attribute of an NA-expression.

Definition 5.4

If ψ is an NA-operator with parameter attributes that are attributes of the list of an attribute $X(\lambda)$, then the *argument attribute* of ψ, denoted by $AA(\psi)$, equals X. (If the parameter attributes are attributes at level 1, then $AA(\psi)$ equals $\Re$; think of $\Re$ as the imaginary parent attribute of the attributes at level 1.) If E is an NA-expression, then $AA(E)$ is defined to be the attribute α at the highest level such that for each NA-operator ψ in E, it holds that either α is an ancestor of $AA(\psi)$ or α equals $AA(\psi)$ (so all the parameter attributes in E must be descendants of α).

□

Consider the scheme $(A(B(C(D,E(F(G,H),I),J(K),L),M(N),O),P),Q)$ and the expression Exp, which is equal to

$$\pi_G(\nu_{D,E;X}(\sigma_{G=H}([Y := C \bowtie M](\mu_B)))).$$

Then $AA(Exp) = A$.

So, the argument attribute is the attribute at the highest level, such that the expression manipulates only the value of this attribute. In the above example the argument attribute can not be at a higher level than A, since the unnest has parameter attribute B. On the other hand, the value of $\Re$ is manipulated, but, since A is the attribute at the highest level for which the value is manipulated and we want to be as specific as possible, we say that A is the argument attribute of Exp.

Definition 5.5

A *selective expression* [if C then E_1 else E_2] can be *applied at multiple levels*, if for every NA-expression E in C it holds that the parent attribute of $AA(E)$ is an ancestor attribute of $AA(E_1)$ or $AA(E)$ equals $\Re$.

The application on an instance s is defined to be the application on every $AA(E_1)$-value v of either E_1 or E_2, dependent of the evaluation of C. Since for every expression E in C there is a unique $AA(E)$-value corresponding with v, we can define $C(v)$ in the same way as in the original definition, the only exception being that here every expression E is applied to the unique $AA(E)$-value corresponding to v.

The $AA(E)$-value corresponding to v is the value v' such that v' is the $AA(E)$-component of the tuple of which the α-value contains v, where we suppose that α is the ancestor of $AA(E_1)$ with the same parent as $AA(E)$. If $AA(E)$ equals $\Re$, then v' is the whole instance.

□

Consider for example the scheme $(A(B(C(D,E),F(G,H)),I,J(K)))$. For a selective expression where the then-expression has parameter attributes D and E, as in $\sigma_{D=E}$ (so C-values are manipulated), the expressions of the condition must manipulate values of C, F, B, I, J or $\Re$.

A well-defined selective expression for the above scheme is

$$[\text{if } I \in J \ \wedge \ \sigma_{D=E} = F \text{ then } \sigma_{D=E} \text{ else } \sigma_{D=E}(\rho_{D,E \to E,D})].$$

Here the argument attribute of the then-expression is C, and with every C-value there are uniquely corresponding I-, J-, C- and F-values.

Note that it is reasonable to require that for every basic if-condition $E = E'$, $E \subseteq E'$ or $E \in E'$ it must hold that $AA(E)$ and $AA(E')$ are in the same list.

Since we can reduce the application of selective expressions at multiple levels to the application at one (higher) level, we have the following theorem.

Theorem 5.3 *The application of selective expressions at multiple levels is NA-expressible.*

Proof
We do not give the full proof here. However, it is obvious that such a selective expression is equivalent to a selective expression, with a basic if-condition (cf. the proof of the NA-expressablility of selective expressions with non-basic if-conditions). This implies that this selective expression is such that the then- and else-expressions manipulate a value v at level k, say, and the condition manipulates a value v' at level l, with l at most k. It is clear that the expression at level k is equivalent to some expression at level l, such that the manipulation of v is in fact the manipulation of some value v'' at level l, and therefore the selective expression is a selective expression applied at level l.

□

6 Assignment Expressions

The notation used for the application of binary operators at level 1 does only allow for the application of one operation at a time. In this section we will generalize this, allowing expressions like

$$[G(H) := A \cup \pi_F],$$

which can be applied to instances of the scheme $(A(B),C(D,E,F))$. In such an expression we specify a new attribute based on an NA-expression, which relates attributes at level 1.

Definition 6.1

A *level-1-expression* is either a binary NA-operator applied on two level-1-expressions (using infix-notation) or an NA-expression with an argument attribute at level 1.

The application of a level-1-expression E on a tuple t of an instance s of scheme (λ) is defined by :

- if E is an NA-expression f, with argument attribute α at level 1, then $E(t) = f(t(\alpha))$; (N.B. if $f = \alpha$, then $E(t) = t(\alpha)$)
- if E equals $E'\phi E''$, with E' and E'' level-1-expressions and ϕ a binary NA-operator, then $E(t) = E'(t)\phi E''(t)$.

Let E be a level-1-expression and s an instance of scheme (λ). Suppose for a tuple t of s, $E(t)$ is a tuple over the attribute λ', and λ'' is compatible with λ', but does not contain identifiers from λ.
Then $[\lambda'' := E]$ is called an *assignment expression* (at level 1) and its application on s is an instance of scheme (λ, λ'') defined by :

$$[\lambda'' := E](s) = \{\, t' \mid \exists t \in s : t'(\lambda) = t \wedge t'(\lambda'') = E(t)\}.$$

□

Consider for example the scheme $(A(B), C(D,E), F(G,H))$ and the level-1-expression E

$$(A \cup \pi_D) \bowtie \sigma_{G=H}.$$

If t is a tuple over this scheme, then

$$E(t) = (t(A(B)) \cup t(C(D))) \bowtie \sigma_{G=H}(t(F(G,H))).$$

If s is an instance of this scheme, then

$$[A'(B',G',H') := E](s)$$

is such that each tuple t in s is augmented with an attribute $A'(B',G',H')$ whose value equals $E(t)$.

Theorem 6.1 *Every assignment expression is NA-expressible.*

Proof
Let s be an instance of scheme (λ) and $[\lambda' := E]$ an assignment expression (at level 1 of (λ)). We will construct an equivalent expression for $[\lambda' := E](s)$.

If the expression E equals $E'\phi E''$, with ϕ a binary operator, then

$$[\lambda' := E](s) \;=\; \pi_{\lambda,\lambda'}([\lambda' := \lambda_1 \phi \lambda_2]([\lambda_1 := E']([\lambda_2 := E''])))(s),$$

for proper λ_1 and λ_2.

If the expression E equals α, with α an attribute at level 1, then

$$[\lambda' := E](s) = copy_{\alpha\rightarrow\lambda}(s).$$

If the expression E equals f, with f an NA-operator with an argument attribute α at level 1, then

$$[\lambda' := E](s) = g(copy_{\alpha\rightarrow\lambda'})(s),$$

where g is an NA-expression equivalent with f, such that each identifier from α is replaced by the corresponding one from λ'.

If the expression E equals $f(f')$, with f an NA-operator and f' an NA-expression with argument attribute α, then

$$[\lambda' := E](s) = g([\lambda' := f'])(s),$$

where g is an NA-expression equivalent with f, such that each identifier from α is replaced by the corresponding one from λ'.

□

It will be obvious how we can generalize the definition of assignment expression to allow for application at any level, analogous to definition 5.3.

Definition 6.2

Let E be an assignment expression. Let s be an instance of scheme (λ), let $A(\lambda')$ be a structured attribute at level $k-1$ in (λ) with k at least 2. If E is such that for all $A(\lambda')$-values v in s $E(v)$ is well-defined, $E(s)$ is defined to be the instance obtained from s by replacing each $A(\lambda')$-value v in s by $E(v)$.
We call this the *application* of E *at level* k.

□

Note that in the definition we describe how the instance is changed, but, of course, the scheme must be changed correspondingly.

Analogous to Theorem 5.2 it is possible to prove by induction that the application at any level is NA-expressible.

Theorem 6.2 *Every application of an assignment expression at level k, with k at least 1, is NA-expressible.*

□

So, for example, the application at level 2 on an instance of the scheme $(A(B, C(D), E(F(G), H)), I)$ of

$$[C'(D') := C \cup \pi_G(\nu_F)]$$

results in an instance of the scheme $(A(B, C(D), E(F(G), H), C'(D')), I)$.

From the expressibility of assignment expressions at any level we can also deduce that we can write assignment expressions within assignment expressions, since for every assignment expression an equivalent NA-expression exists.

As we have extended the definition of selective expressions to allow for application at multiple levels, we can also do this for assignment expressions. This means that for all the NA-expressions in a level-1-expression the argument attributes do not need to be at the same level. For example, we want to be able to apply

$$[C'(D') := C \bowtie \pi_F]$$

on instances of the scheme $(A(B, C(D)), E(F, G))$, resulting in well-defined instances of the scheme $(A(B, C(D), C'(D')), E(F, G))$.

For the definition of the application at multiple levels it is required that the attributes, that are manipulated by the NA-expressions within the level-1-expression, are uniquely corresponding. This means that one of the NA-expressions manipulates values of an attribute, α say, and the other NA-expressions manipulate α'-values such that the parent attribute of α' is an ancestor attribute of α.

Definition 6.3

An *assignment expression* $[\lambda := E]$ can be *applied at multiple levels*, if there exists an NA-expression E_0 in E such that for every NA-expression E_1 in E it holds that the parent attribute of $AA(E_1)$ is an ancestor attribute of $AA(E_0)$ or $AA(E_1)$ equals $\Re$ (N.B. $AA(E_0) \neq \Re$).

Let P be the parent attribute of $AA(E_0)$ and s an instance. Then the application on s is defined by adding to every tuple t in every P-value v of s the λ-value v', where v' is obtained by applying first all the NA-expressions to the values of their argument attribute, that is uniquely corresponding to the value of $t(AA(E_0))$, and then applying all the binary operators to these values.

□

Of course, the way in which attribute values correspond uniquely is the same as in definition 5.5, where the application of selective expressions at multiple levels was defined.

Theorem 6.3 *The application of assignment expressions at multiple levels is NA-expressible.*

Proof
As in the proof of theorem 5.3 it is possible to construct for every NA-expression within the level-1-expression an equivalent expression such that all the NA-expressions are expressions at the same level, k say, thus having an assignment expression at level k.

□

7 Operations that are not NA-expressible

In this final section we define a number of operations on nested relations for which we prove that they are not NA-expressible. Hence, additional operators or functions are needed in the nested algebra for implementing these operations.

Basically we prove that

- the operation that selects the sets with the greatest cardinality out of a number of given sets is not NA-expressible
- the operation that gives all the pairs of connected nodes in an undirected graph is not NA-expressible

From these two results we deduce a number of interesting operations that are not NA-expressible. The proof technique that we propose can be used to proof the "non-NA-expressibility" of many other operations.

Definition 7.1

A scheme is called *flat* if and only if all its attributes are atomic, i.e. if and only if all its identifiers have level 1.
An NA-expression is called flat if and only if every attribute of every relation that occurs in the expression is atomic and if the nest operator does not occur in the expression.

□

Definition 7.2

Consider the scheme $(A(B))$ and an instance s of this scheme.
The operator $Maxc$ that selects those tuples from s, for which their only component has the greatest cardinality, is called the *maximal-cardinality operator.*
The scheme of $Maxc(s)$ is $(A(B))$.

□

Example :

$$\begin{array}{c} A(B) \\ \hline \left\{\begin{array}{c}1\\2\\3\end{array}\right\} \\ \left\{\begin{array}{c}3\\4\\5\\6\end{array}\right\} \\ \left\{\begin{array}{c}1\\4\\5\\6\end{array}\right\} \\ \left\{\begin{array}{c}1\end{array}\right\} \\ \\ s \end{array} \qquad\qquad \begin{array}{c} A(B) \\ \hline \left\{\begin{array}{c}3\\4\\5\\6\end{array}\right\} \\ \left\{\begin{array}{c}1\\4\\5\\6\end{array}\right\} \\ \\ Maxc(s) \end{array}$$

Definition 7.3

We will use the following notation $[p, q]$ for the instance over the scheme (A, B)

$$[p,q] = \{(a,-1) \mid -p \leq a \leq -2\} \cup \{(a,1) \mid 2 \leq a \leq q\}$$

□

Definition 7.4

In the sequel, p and q are two given different positive integers.
The calculus expression $\{(a_1, \dots, a_n) \mid f(a_1, \dots, a_n)\}$ is called a *derivable calculus expression* (*dce*) iff

1. f is a disjunction of disjuncts

2. every disjunct is false or is a conjunction of factors

3. every factor is a term or its negation
4. every term has the form
   (a) $a_i = a_j$
   (b) $|a_i| = 1$
   (c) $a_i \star a_j > 0$
5. every non false disjunct has for every $i, 1 \leq i \leq n$, the factor $|a_i| = 1$ or $\neg(|a_i| = 1)$

The variables a_i range over $-p, \ldots, -1, 1, \ldots, q$. The semantics of a *dce* is clear.

$\square$

Lemma 7.1 *If a disjunct satisfies one of the following conditions, it is equivalent with $false$.*

1. $\exists i, 1 \leq i \leq n : \neg(a_i = a_i)$ *is a factor*
2. $\exists i, 1 \leq i \leq n : \neg(a_i \star a_i > 0)$ *is a factor*
3. $\exists i, 1 \leq i \leq n :$ *both* $|a_i| = 1$ *and* $\neg(|a_i| = 1)$ *are factors*

$\square$

If a disjunct satisfies one of the conditions of lemma 7.1, we will substitute it by $false$ in the sequel.

Lemma 7.2 *If a disjunct contains $a_i = a_i$, for some $i, 1 \leq i \leq n$, it is equivalent to the disjunct without this factor. If a disjunct contains $a_i \star a_i > 0$, for some $i, 1 \leq i \leq n$, it is equivalent to the disjunct without this factor.*

$\square$

If a disjunct satisfies one of the conditions of lemma 7.2, we will delete the corresponding factor in the sequel.

Lemma 7.3 $[p, q] = \{(a_1, a_2) \mid a_1 \star a_2 > 0 \wedge |a_2| = 1 \wedge \neg(|a_1| = 1)\}$.

$\square$

Lemma 7.4 *If the flat expressions E_1 and E_2 have only one operand, $[p, q]$, and if they are both expressible by dce's, then their union, join and difference are expressible by dce's. Furthermore, their selections and renamings are expressible by dce's.*

Proof
Let $E_1([p,q]) = \{(a_1, \ldots, a_n) \mid f_1(a_1, \ldots, a_n)\}$ and $E_2([p,q]) = \{(a_1, \ldots, a_n) \mid f_2(a_1, \ldots, a_n)\}$.

$E_1 \cup E_2([p,q]) = \{(a_1, \ldots, a_n) \mid f_1(a_1, \ldots, a_n) \vee f_2(a_1, \ldots, a_n)\}$
$E_1 - E_2([p,q]) = \{(a_1, \ldots, a_n) \mid f(a_1, \ldots, a_n)\}$ where $f(a_1, \ldots, a_n)$ is a disjunction equivalent with $f_1(a_1, \ldots, a_n) \wedge \neg(f_2(a_1, \ldots, a_n))$.

Let $E_3([p,q]) = \{(b_1, \ldots, b_m) \mid f_3(b_1, \ldots, b_m)\}$.

$E_1 \bowtie E_3([p,q]) = \{(a_1,\ldots,a_n,b_1,\ldots,b_m) \mid f(a_1,\ldots,a_n,b_1,\ldots,b_m)\}$, where $f(a_1,\ldots,a_n,$ $b_1,\ldots,b_m)$ is a disjunction equivalent with $f_1(a_1,\ldots,a_n) \wedge f_3(b_1,\ldots,b_m)$, and $a_1,\ldots,a_n,$ $b_1,\ldots,b_m$ all different.

$\sigma_{i=j}(E_1)([p,q]) = \{(a_1,\ldots,a_n) \mid f(a_1,\ldots,a_n)\}$, where $f(a_1,\ldots,a_n)$ is a disjunction equivalent with $f_1(a_1,\ldots,a_n) \wedge a_i = a_j$.

□

Lemma 7.5 *If the flat expression E has only one operand, $[p,q]$, and if it is expressible by dce's, then its projections are expressible by dce's, for p and q great enough.*

Proof

We start with $E([p,q]) = \{(a_1,\ldots,a_n) \mid f(a_1,\ldots,a_n)\}$. We will construct a *dce* that represents one of its projections. Consider the projection on $(a_2,\ldots,a_n)$. a_1 is the only attribute that is projected-out. Clearly, every projection can be written as a sequence of projections where only one attribute is projected-out. Therefore, our result will be general. Suppose also that f has only one disjunct. Otherwise we apply the construction to every disjunct of f.

The projection is expressed by $\{(a_2,\ldots,a_n) \mid \exists a_1 : f(a_1,\ldots,a_n)\}$. We will eliminate the existential quantor.

1. If f is $false$, the projection is expressed by $\{(a_2,\ldots,a_n) \mid false\}$.
2. Suppose that f contains some factors $a_1 = a_i$. For each a_i, take all the factors of f that contain a_1, substitute a_1 by a_i, add these new factors to f. Delete then all factors $a_1 = a_i$. Let f' be the resulting function. Clearly, $\{(a_2,\ldots,a_n) \mid \exists a_1 : f(a_1,a_2,\ldots,a_n)\}$ can be written as $\{(a_2,\ldots,a_n) \mid f'(a_2,\ldots,a_n)\}$, which is a *dce*.
3. Suppose that f does not contain any factor of the form $a_1 = a_i$. We calculate the closure of f, notated $\hat{f}$, by applying the following rules recursively :
   - if $a_i = a_j$ in $\hat{f}$, then add $a_j = a_i$
   - if $\neg(a_i = a_j)$ in $\hat{f}$, then add $\neg(a_j = a_i)$
   - if $a_i \star a_j > 0$ in $\hat{f}$, then add $a_j \star a_i > 0$
   - if $\neg(a_i \star a_j > 0)$ in $\hat{f}$, then add $\neg(a_j \star a_i > 0)$
   - if $a_i = a_j$ in $\hat{f}$, then add $a_i \star a_j > 0$
   - if $a_i = a_j$ and $a_i = a_k$ in $\hat{f}$, then add $a_j = a_k$
   - if $a_i = a_j$ and $\neg(a_i = a_k)$ in $\hat{f}$, then add $\neg(a_j = a_k)$
   - if $a_i = a_j$ and $|a_i| = 1$ in $\hat{f}$, then add $|a_j| = 1$
   - if $a_i = a_j$ and $\neg(|a_i| = 1)$ in $\hat{f}$, then add $\neg(|a_j| = 1)$
   - if $a_i = a_j$ and $a_i \star a_k > 0$ in $\hat{f}$, then add $a_j \star a_k > 0$
   - if $a_i = a_j$ and $\neg(a_i \star a_k > 0)$ in $\hat{f}$, then add $\neg(a_j \star a_k > 0)$
   - if $|a_i| = 1$ and $\neg(|a_j| = 1)$ in $\hat{f}$, then add $\neg(a_i = a_j)$
   - if $a_i \star a_j > 0$ and $a_i \star a_k > 0$ in $\hat{f}$, then add $a_j \star a_k > 0$
   - if $a_i \star a_j > 0$ and $\neg(a_i \star a_k > 0)$ in $\hat{f}$, then add $\neg(a_j \star a_k > 0)$
   - if $\neg(a_i \star a_j > 0)$ in $\hat{f}$, then add $\neg(a_i = a_j)$

- if $\neg(a_i \star a_j > 0)$ and $\neg(a_i \star a_k > 0)$ in $\hat{f}$, then add $a_j \star a_k > 0$
- if $|a_i| = 1$ and $|a_j| = 1$ and $\neg(a_i = a_j)$ in $\hat{f}$, then add $\neg(a_i \star a_j > 0)$
- if $|a_i| = 1$ and $|a_j| = 1$ and $a_i \star a_j > 0$ in $\hat{f}$, then add $a_i = a_j$
- if $|a_i| = 1$ and $|a_j| = 1$ and $\neg(a_i \star a_j > 0)$ and $\neg(a_i = a_k)$ and $\neg(a_j = a_k)$ in $\hat{f}$, then add $\neg(|a_k| = 1)$

Next, we delete all the factors that contain a_1. Let f' be the resulting function. It is obvious that if $f(a_1, a_2, \ldots, a_n)$ holds, also $f'(a_2, \ldots, a_n)$ will hold. We have to prove :

> If $f'(a_2, \ldots, a_n)$ holds, then there exists a value a_1 such that $f(a_1, a_2, \ldots, a_n)$ holds.

(a) Suppose $|a_1| = 1$ in $\hat{f}$ and $a_1 \star a_i > 0$ in $\hat{f}$ for some i.
Take $t(a_1) = 1$ if $t(a_i) > 0$ and $t(a_1) = -1$ if $t(a_i) < 0$.
Then there cannot be a contradiction, since if this were the case, the choice for the value of $t(a_1)$ would contradict one of the factors of the disjunct :

- $\neg(a_1 = a_i)$ in $\hat{f}$ and $t(a_1) = t(a_i)$.
  But then $\neg(|a_i| = 1)$ and $|t(a_i)| = 1$ hold.
- There is an a_j, such that $a_j = a_1$ in $\hat{f}$ and $\neg(t(a_j) = t(a_1))$ holds.
  But $a_j = a_1$ is not possible.
- There is an a_j, such that $\neg(a_j = a_1)$ in $\hat{f}$ and $t(a_j) = t(a_1)$ holds.
  But then we must have $|a_j| = 1$ in $\hat{f}$, hence $\neg(a_1 \star a_j > 0)$ and $\neg(a_i \star a_j > 0)$ and $t(a_i) \star t(a_j) > 0$.
- $|a_1| = 1$ in $\hat{f}$ and $\neg(|t(a_1)| = 1)$.
  This is impossible.
- $\neg(|a_1| = 1)$ in $\hat{f}$ and $|t(a_1)| = 1$.
  This is impossible.
- There is an a_j, such that $a_j \star a_1 > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_1) > 0)$ holds.
  But then $a_i \star a_j > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_i) > 0)$.
- There is an a_j, such that $\neg(a_j \star a_1 > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_1) > 0$ holds.
  But then $\neg(a_i \star a_j > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_i) > 0$.

(b) Suppose $|a_1| = 1$ in $\hat{f}$ and $\neg(a_1 \star a_i > 0)$ in $\hat{f}$ for some i.
Take $t(a_1) = 1$ if $t(a_i) < 0$ and $t(a_1) = -1$ if $t(a_i) > 0$.
Then there cannot be a contradiction, since in that case the choice for the value of $t(a_1)$ would contradict one of the factors of the disjunct :

- $\neg(a_1 = a_i)$ in $\hat{f}$ and $t(a_1) = t(a_i)$.
  This is impossible.
- There is an a_j, such that $a_j = a_1$ in $\hat{f}$ and $\neg(t(a_j) = t(a_1))$ holds.
  But $a_j = a_1$ is not possible.
- There is an a_j, such that $\neg(a_j = a_1)$ in $\hat{f}$ and $t(a_j) = t(a_1)$ holds.
  But then we must have $|a_j| = 1$ in $\hat{f}$, hence $\neg(a_1 \star a_j > 0)$ and $a_i \star a_j > 0$ and $\neg(t(a_i) \star t(a_j) > 0)$.
- $|a_1| = 1$ in $\hat{f}$ and $\neg(|t(a_1)| = 1)$.
  This is impossible.
- $\neg(|a_1| = 1)$ in $\hat{f}$ and $|t(a_1)| = 1$.
  This is impossible.

- There is an a_j, such that $a_j \star a_1 > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_1) > 0)$ holds.
  But then we have $\neg(a_i \star a_j > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_i) > 0$.
- There is an a_j, such that $\neg(a_j \star a_1 > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_1) > 0$ holds.
  But then we have $a_i \star a_j > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_i) > 0)$.

(c) Suppose $|a_1| = 1$ in $\hat{f}$ and there is no $a_1 \star a_i > 0$ nor $\neg(a_1 \star a_i > 0)$ in $\hat{f}$.
Take $t(a_1) = 1$.
Then there cannot be a contradiction, since in that case the choice for the value of $t(a_1)$ would contradict one of the factors of the disjunct :

- $\neg(a_1 = a_i)$ in $\hat{f}$ and $t(a_i) = 1$ holds.
  But then we would have $|a_i| = 1$, hence $\neg(a_1 \star a_i > 0)$ and $t(a_1) \star t(a_i) > 0$.

(d) Suppose $\neg(|a_1| = 1)$ in $\hat{f}$ and $a_1 \star a_i > 0$ in $\hat{f}$ for some i.
Take for $t(a_1)$ a new value with the same sign as $t(a_i)$.
Then there cannot be a contradiction, since in that case the choice for the value of $t(a_1)$ would contradict one of the factors of the disjunct :

- $\neg(a_1 = a_i)$ in $\hat{f}$ and $t(a_1) = t(a_i)$.
  This is impossible.
- There is an a_j, such that $a_j = a_1$ in $\hat{f}$.
  This is impossible.
- There is an a_j, such that $\neg(a_j = a_1)$ in $\hat{f}$ and $t(a_j) = t(a_1)$ holds.
  This is impossible.
- $|a_1| = 1$ in $\hat{f}$ and $\neg(|t(a_1)| = 1)$.
  This is impossible.
- $\neg(|a_1| = 1)$ in $\hat{f}$ and $|t(a_1)| = 1$.
  This is impossible.
- There is an a_j, such that $a_j \star a_1 > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_1) > 0)$ holds.
  But then we have $a_i \star a_j > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_i) > 0)$.
- There is an a_j, such that $\neg(a_j \star a_1 > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_1) > 0$ holds.
  But then we have $\neg(a_i \star a_j > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_i) > 0$.

(e) Suppose $\neg(|a_1| = 1)$ in $\hat{f}$ and $\neg(a_1 \star a_i > 0)$ in $\hat{f}$ for some i.
Take for $t(a_1)$ a new value with the opposite sign as $t(a_i)$.
Then there cannot be a contradiction, since in that case the choice for the value of $t(a_1)$ would contradict one of the factors of the disjunct :

- $\neg(a_1 = a_i)$ in $\hat{f}$ and $t(a_1) = t(a_i)$.
  This is impossible.
- There is an a_j, such that $a_j = a_1$ in $\hat{f}$ and $\neg(t(a_j) = t(a_1))$ holds.
  This is impossible.
- There is an a_j, such that $\neg(a_j = a_1)$ in $\hat{f}$ and $t(a_j) = t(a_1)$ holds.
  This is impossible.
- $|a_1| = 1$ in $\hat{f}$ and $\neg(|t(a_1)| = 1)$.
  This is impossible.
- $\neg(|a_1| = 1)$ in $\hat{f}$ and $|t(a_1)| = 1$.
  This is impossible.
- There is an a_j, such that $a_j \star a_1 > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_1) > 0)$ holds.
  But then we have $\neg(a_i \star a_j > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_i) > 0$.

- There is an a_j, such that $\neg(a_j \star a_1 > 0)$ in $\hat{f}$ and $t(a_j) \star t(a_1) > 0$ holds.
  But then we have $a_i \star a_j > 0$ in $\hat{f}$ and $\neg(t(a_j) \star t(a_i) > 0)$.

(f) Suppose $\neg(|a_1| = 1)$ in $\hat{f}$ and there is no $a_1 \star a_i > 0$ nor $\neg(a_1 \star a_i > 0)$ in $\hat{f}$.
Take for $t(a_1)$ a new value.
Then there cannot be a contradiction, since in that case the choice for the value of $t(a_1)$ would contradict one of the factors of the disjunct :

- $\neg(a_1 = a_i)$ in $\hat{f}$ and $t(a_i) = t(a_1)$ holds.
  This is impossible.

□

Theorem 7.1 *For every expression E in the flat algebra with one operand of scheme $(A(B))$, it holds that there is a dce $\{(a_1, \ldots, a_n) \mid f(a_1, \ldots, a_n)\}$ such that for p and q great enough*

$$E([p,q]) = \{(a_1, \ldots, a_n) \mid f(a_1, \ldots, a_n)\}.$$

f is independent from p and q.

Proof

The proof is a direct consequence of lemma 7.4 and lemma 7.5.

□

Theorem 7.2 *Let M be an operator such that $M([p,q]) = \{a \mid -p \leq a \leq -2\}$ if $p > q$ and $M([p,q]) = \{a \mid 2 \leq a \leq q\}$ if $p < q$. M is not expressible in the flat algebra.*

Proof

If M would be expressible by the flat algebra we know from theorem 7.1 that $M([p,q]) = \{a \mid f(a)\}$.
Since this is a *dce*, there are only four possible values for $M([p,q])$: $\emptyset$, $\{-1, 1\}$, $\{a \mid -p \leq a \leq -2 \text{ or } 2 \leq a \leq q\}$, $\{a \mid -p \leq a \leq q,\ a \neq 0\}$.

□

Theorem 7.3 *Maxc is not NA-expressible.*

Proof

If *Maxc* would be NA-expressible, then M would be expressible by the flat algebra. Indeed, we start with $[p,q]$. We nest the first attribute and project on the first attribute. We apply *Maxc* to the result and we unnest. This gives an expression in the nested algebra that expresses M. By [PVG], there is also an equivalent expression in the flat algebra.

□

The second operator that is proved not to be NA-expressible, is the operation that gives all the pairs of connected nodes in an undirected graph.

Definition 7.5

Consider an instance s of the scheme (A, B) such that $<a, b> \in s \Longleftrightarrow <b, a> \in s$.
The operator P_C is defined by

$$P_C(s) = \{(c,d) \mid \exists\, c_0, c_1, \ldots, c_k \text{ with } c_0 = c,\ c_k = d, \\ (<c_{i-1}, c_i> \in s \text{ or } c_{i-1} = c_i) \text{ for } i = 1, \ldots, k\}.$$

The scheme of $P_C(s)$ is (A, B).
P_C is called the *path connectivity*.

□

Example :

A	B
1	2
2	1
2	3
3	2
4	5
5	4
5	5
6	6

s

A	B
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3
4	4
4	5
5	4
5	5
6	6

$P_C(s)$

Theorem 7.4 *P_C is not NA-expressible.*

Proof
This proof is rather analogous to the proof of the previous result.
We present a brief outline.

Consider the instance

$$[p] = \{<a, b> \ | \ -p \leq a \leq p, \ -p \leq b \leq p, \ a \neq 0, \ b \neq 0, \ ((a = b - 1) \text{ or } (a = b + 1))\}$$

We first will prove that the result of every flat expression with operand $[p]$ can be expressed as $\{(a_1, \ldots, a_n) \mid f(a_1, \ldots, a_n)\}$ where every a_i ranges over $-p, -p+1, \ldots, -2, -1, 1, 2, \ldots, p$ and where

1. f is a disjunction of disjuncts
2. every disjunct is $false$ or is a conjunction of factors
3. every factor is a term or its negation
4. a term has the form
   - $a_i - a_j = k$
   - $a_i = k$
   - $a_i = p - k$
   - $a_i = -p - k$

   where k is a constant (pos. or neg.)

(We substitute a disjunct by *false* if necessary.)

Clearly $[p] = \{(a_1, a_2) \mid (a_1 - a_2 = 1) \vee (a_2 - a_1 = 1)\}$. Furthermore, the application of the union, the difference, the join, the selection and the renaming do not create problems.
As to the projection, suppose that

$$E([p]) = \{(a_1, \ldots, a_n) \mid f(a_1, \ldots, a_n)\}$$

where f has only one disjunct, and consider $\pi_{A_2,\ldots,A_n}(E([p]))$.
There are 4 cases.

1. If $a_1 - a_i = k$ is a factor, then add to f
   - $\bigwedge_{j=0}^{k-1} \neg(a_i = p - j)$ if $k > 0$
   - $\bigwedge_{j=k+1}^{0} \neg(a_i = -p - j)$ if $k < 0$
   - $\neg(a_i = -k)$

   and replace every a_1 by $a_i + k$.
2. If 1 does not hold and $a_i - a_1 = k$ is a factor, then add to f
   - $\bigwedge_{j=0}^{k-1} \neg(a_i = -p - (-j))$ if $k > 0$
   - $\bigwedge_{j=k+1}^{0} \neg(a_i = p - (-j))$ if $k < 0$
   - $\neg(a_i = k)$

   and replace every a_1 by $a_i - k$.
3. If 1 and 2 do not hold and $a_1 = k$ or $a_1 = p - k$ or $a_1 = -p - k$ is a factor, then substitute a_1 by its value.
4. If all terms that contain a_1 are negative, delete them.

So, every expression E in the flat algebra with one operand $[p]$, can be expressed as $\{(a_1, \ldots, a_n) \mid f(a_1, \ldots, a_n)\}$ with f independent from p.

Finally, we have to prove that

$$P_C([p]) = \{(a, b) \mid (0 < a \leq p,\ 0 < b \leq p) \vee (-p \leq a < 0,\ -p \leq b < 0)\}$$

is not expressible in the way above. If it were, then there exists a disjunct without positive terms. Take p greater than the greatest constant in this disjunct. The tuple $<-p, p>$ fulfils but does not belong to $P_C([p])$, which contradicts.
Hence P_C is not expressible in the flat algebra and hence, by [PVG], it is not NA-expressible.

□

Definition 7.6

Consider an instance s of the scheme $(A(B))$. The operator $Minc$ that selects the tuples of s that have the smallest cardinality, is called the *minimal cardinality operator.* The scheme of $Minc(s)$ is $(A(B))$.
Consider an instance s of the scheme $(A(B), C(D))$. The parameterized operator $CS^p_{A,C}$ selects the tuples $< a, b >$ of s with $card(a) - card(b) \leq p$. The scheme of $CS^p_{A,C}$ is

$(A(B), C(D))$.
Consider an instance s of the scheme $(A(B), C(D))$. The parameterized operator $CG^p_{A,C}$ selects the tuples $< a, b >$ of s with $card(a) - card(b) \geq p$. The scheme of $CG^p_{A,C}$ is $(A(B), C(D))$.
Consider an instance s of the scheme $(A(B), C(D), E(F))$. The parameterized operator $CS^p_{A,C,E}$ selects the tuples $<a, b, c>$ of s with $card(a) - card(b) \leq card(c) + p$. The scheme of $CS^p_{A,C,E}$ is $(A(B), C(D), E(F))$.
Consider an instance s of the scheme $(A(B), C(D), E(F))$. The parameterized operator $CG^p_{A,C,E}$ selects the tuples $<a, b, c>$ of s with $card(a) - card(b) \geq card(c) + p$. The scheme of $CG^p_{A,C,E}$ is $(A(B), C(D), E(F))$.

□

Theorem 7.5 *Minc, and the parameterized operations $CS^p_{A,C}$, $CG^p_{A,C}$, $CS^p_{A,C,E}$ and $CG^p_{A,C,E}$ are not NA-expressible.*

Proof

1. This is obvious for $Minc$.
2. Let $CS^0_{A,C}$ be NA-expressible and let s be an instance over the scheme $(A(B))$.
$$\pi_A(\sigma_{C'=C''}(\nu_{C;C'}(CS^0_{A,C}(s \bowtie \rho_{A(B)\to C(D)}(s))) \bowtie \nu_{A'';C''}(\rho_{A(B)\to A''(B'')}(s))))$$
would be an expression for $Minc$.
3. If $CS^p_{A,C}$ were NA-expressible, there would be an expression for $CS^0_{A,C}$.
4. If $CS^p_{A,C,E}$ were NA-expressible and s were an instance of $(A(B), C(D))$ then
$$\pi_{A,C}(CS^p_{A,C,E}(s \bowtie \pi_E([E(F) := C - C](s))))$$
would be a parameterized expression for $CS^p_{A,C}$.
5. Clearly we have an analogous proof for $CG^p_{A,C}$ and $CG^p_{A,C,E}$.

□

Definition 7.7

Consider an instance s of the scheme (A, B). The operator T_C is defined by

$$T_C(s) = \{(c, d) \mid \exists\, c_0, \ldots, c_k \text{ with } c_0 = c,\ c_k = d, \\ ((c_{i-1}, c_i) \in s \text{ or } c_{i-1} = c_i) \text{ for } i = 1, \ldots, k\}$$

T_C is called the *transitive closure*. The scheme of $T_C(s)$ is (A, B) .
Consider an instance s of the scheme $(B(C))$. The operator S_C is defined by

$$S_C(s) = \{(c, d) \mid \exists\, c_0, \ldots, c_k \text{ with } c_0 = c,\ c_k = d, \\ (c_{i-1} \cap c_i) \neq \emptyset,\ c_{i-1} \in s,\ c_i \in s \text{ for } i = 1, \ldots, k\}$$

S_C is called the *set connectivity.* The scheme of $S_C(s)$ is $(B(C), B'(C'))$.
Consider instances r and s of the schemes $(B(C))$ and (E, F) respectively. The operator St_C is defined by

$$\begin{aligned} St_C(r,s) = \{(c,d) \mid\ & \exists\, c_0,\ldots,c_k(c = c_0,\ d = c_k,\ c_0,\ldots,c_k \in r,\\ & \forall\, i = 1,\ldots,k\ (\exists(e_i,f_i) \in s : e_i \in c_{i-1} \text{ and } f_i \in c_i))\} \end{aligned}$$

St_C is called the *set transitive connectivity.* The scheme of $St_C(r,s)$ is $(B(C), B'(C'))$.
Consider an instance s of the scheme $(B(C))$. The operator Sa_C is defined by

$$\begin{aligned} Sa_C(s) = \{(c,d) \mid\ & \exists\, c_0,\ldots,c_k(c = c_0,\ d = c_k,\ c_0,\ldots,c_k \in s,\\ & (c_{i-1} \subset c_i \text{ or } c_{i-1} \supset c_i) \text{ for } i = 1,\ldots,k\} \end{aligned}$$

Sa_C is called the *set alternating connectivity.* The scheme of $Sa_C(s)$ is $(B(C), B'(C'))$.

□

Theorem 7.6 *The operators T_C, S_C, St_C and Sa_C are not NA-expressible.*

Proof

1. If T_C were NA-expressible, P_C would also be.
2. If S_C were NA-expressible, P_C would be by

$$\begin{aligned} s_1 &= S_C(\rho_{E(F)\to B(C)}(\pi_E([E(F) := C \cup D](\nu_{A;C}(\nu_{B;D}(copy_{A,B\to A',B'}(s))))))) \\ P_C(s) &= \rho_{C,C'\to A,B}(\mu_{B'}(\mu_B(s_1))) \end{aligned}$$

3. If St_C were NA-expressible, S_C would be by

$$S_C(s) = St_C(s, \rho_{C,C'\to E,F}(\sigma_{C=C'}(\mu_B(s) \bowtie \rho_{C\to C'}\mu_B(s))))$$

4. If Sa_C were NA-expressible, P_C would be by

$$\begin{aligned} s_1 &= \nu_{A;C}(\nu_{B;D}(copy_{A,B\to A',B'}(s))) \\ s_2 &= \rho_{E(F)\to B(C)}(\pi_E([E(F) := C \cup D](s_1))) \\ s_3 &= (\rho_{D(B)\to B(C)}(\pi_D(s_1))) \cup (\rho_{C(A)\to B(C)}(\pi_C(s_1))) \\ P_C(s) &= \rho_{C,C'\to A,B}(\mu_D(\mu_C(Sa_C(s_2 \cup s_3)))) \end{aligned}$$

□

8 Conclusion

This paper illustrates the expressive power of the nested relational algebra. It demonstrates that this algebra is a suitable model for the implementation of nested relational languages. It can also be used for implementing other models, such as complex objects, and even object oriented databases.

The nested relational model seems to be more appropriate to model information since the structure of a scheme reflects the structure of the information. On the other hand the nested relational model seems to be more complicated than the traditional flat relational model, in the sense that expressing queries is far more straightforward in the latter than in the former.

In the near future we have to look for simpler primitives. We also have to define a simple nested calculus that is equivalent with the algebra, and in which the queries are expressed in a more natural way than in the algebra.

Both the calculus and the algebra have to be generalized with aggregate functions, recursion, methods, etc.

We thus hope to gain a better insight in the development of an orthogonal and elegant query and database language that represents the structure of the information in a more direct and natural way than actual relational languages do.

References

[CH] A.K. Chandra, D. Harel, Structure and Complexity of Relational Queries, *Journal of Systems and Computers Sciences 25*, pp. 99-128, 1985.

[GVG] M. Gyssens, D. Van Gucht, The Powerset Operator as an Algebraic Tool for Understanding Least Fixpoint Semantics in the Context of Nested Relations, *Technical Report no. 233*, Indiana University, Bloomington, October 1987.

[HP] G.J. Houben, J. Paredaens, The R^2-Algebra : Extension of an Algebra for Nested Relations, *Tech.Rep. CSN 87/20*, Tech. University Eindhoven, 1987.

[HS] R. Hull, J. Su, On the Expressive Power of Database Queries with Intermediate Types, *ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database* Systems, pp. 39-51, 1988.

[OOM] G. Ozsoyoglu, Z.M. Ozoyoglu, V. Matos, Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions, *ACM TODS*, 12, 4, pp. 566-592, 1987.

[PA] P. Pistor, F. Andersen, Designing a Generalized NF^2 Model with an SQL-Type Language Interface, *Proc 12th VLDB*, Kyoto, pp. 278-288, 1986.

[PVG] J. Paredaens, D. Van Gucht, Possibilities and Limitations of using flat operators in Nested Algebra Expressions, *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 29-38, 1988.

[RKS] M.A. Roth, H.F. Korth, A. Silberschatz, Theory of Non-First-Normal-Form Relational Databases, *Tech. Rep. TR-84-36 (Revised January 1986)*, University of Texas, Austin, 1984.

[S] M.H. Scholl, Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations, *Proc. 1st ICDT*, Rome, Italy, Sept. 1986, in *Lecture Notes in Computer Science, 243*, G. Ausiello and P. Atzeni, eds. Springer-Verlag, pp. 380-396, 1987.

[SPS] M.H. Scholl, H.-B. Paul, H.-J. Schek, Supporting Flat Relations by a Nested Relational Kernel, *13th VLDB, Brighton*, 1987.

[SS] H.-J. Schek, M.H. Scholl, The Relational Model with relation-valued attributes, *information Systems, Vol. 11, 2*, pp. 137-147, 1986.

[TF] S.J. Thomas, P.C. Fisher, Nested Relational Structures, *Advanced in Computing Research III, The Theory of Databases*, P.C. Kanellakis, ed., *JAI Press*, pp. 269-307, 1986.

[VG] D. Van Gucht, On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model, *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 302-312, 1987.

86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distri-buted information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeter-ministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns, searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language .

87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples
87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to handle Structured Information
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling

88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits