Converting Nested Algebra Expressions into Flat Algebra Expressions

JAN PAREDAENS
University of Antwerp
and
DIRK VAN GUCHT
Indiana University

Nested relations generalize ordinary flat relations by allowing tuple values to be either atomic or set valued. The nested algebra is a generalization of the flat relational algebra to manipulate nested relations. In this paper we study the expressive power of the nested algebra relative to its operation on flat relational databases. We show that the flat relational algebra is rich enough to extract the same "flat information" from a flat database as the nested algebra does. Theoretically, this result implies that recursive queries such as the transitive closure of a binary relation cannot be expressed in the nested algebra. Practically, this result is relevant to (flat) relational query optimization.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*query languages*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Algebraic query transformation, nested algebra, nested calculus, nested relations, relational databases

1. INTRODUCTION

In 1977 Makinouchi [29] proposed generalizing the relational database model by removing the first normal form assumption. Jaeschke and Schek [24] introduced a generalization of the ordinary relational model by allowing relations with set-valued attributes and by adding two restructuring operators, the nest and the unnest operators, to manipulate such homogeneously nested relations of powerset type. A similar generalization was proposed by

Authors' addresses: J. Paredaens, Department of Mathematics and Computer Science, University of Antwerp, B-2610 Antwerpen, Belgium; D. Van Gucht, Computer Science Department, Indiana University, Bloomington, IN 47405.

An abstract of this paper appeared as "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions," in *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex.), 1988, pp. 29–38.

Ozsoyoglu, Ozsoyoglu, and Matos [31]. (In addition, they defined a calculus to manipulate one-level nested relations and showed equivalence of their algebraic and calculus-like data manipulation languages.) Thomas and Fischer [43] generalized Jaeschke and Schek's model and algebra and allowed nested relations of arbitrary (but fixed) depth (see also [2, 4, 12, 18, 21, 32, 38]). Roth, Korth, and Silberschatz [37] defined a calculus-like query language for the nested relational model of Thomas and Fischer. Since then, numerous SQL-like query languages [28, 34, 35, 37], graphical-oriented query languages [22], and datalog-like languages [1, 5, 6, 9, 25, 30] have been introduced for this model or for slight generalizations of it. Also, various groups [7, 13–17, 27, 33, 39, 41] have started with the implementation of the nested relational database model, some on top of an existing database management system and others from scratch.

In this paper we focus on the nested algebra as proposed by Thomas and Fischer [43]. The key problem we address is the following: "What is the expressive power of the nested algebra when we are only interested in its operation on flat relations and when the result is also flat?" In other words, "Is it possible to write queries in the nested algebra with flat operands as input and with a flat output that cannot be expressed in the ordinary (flat) relational algebra?" We show that the answer to this question is negative. Hence, the flat algebra is rich enough to extract the same "flat information" from a flat database as the nested algebra does. This result has interesting theoretical as well as practical consequences:

—Recursive queries such as the transitive closure of a binary relation cannot be expressed in the nested algebra, because if they could, the transitive closure could be expressed in the flat algebra, contradicting a result of Aho and Ullman [3]. In contrast, when the powerset operator is added to the nested algebra, then it is possible, as shown by Abiteboul and Beeri [1] and by Gyssens and Van Gucht [19, 20], to express recursive queries. Hence, the nested algebra with the powerset operator is strictly more powerful than the classical nested algebra studied in this paper. In this light, it is also interesting to mention the results of Hull and Su [23] and of Kuper and Vardi [26] regarding the expressiveness of the powerset algebra. They show that there exists a noncollapsing hierarchy of classes of powerset algebra expressions. In particular, they show that the class of powerset algebra expressions in which one can use up to n ($n \geq 0$) (nested) applications of the powerset operator is strictly less expressive than the class of nested algebra expressions that can use up to $n + 1$ (nested) applications of the powerset operator. Our result shows that an analogous property does not hold for the nested algebra; that is, the number of allowed (nested) applications of the nest operator plays no role in the expressive power of the nested algebra expressions (for a related result, see [8]).

—Several researchers [33, 41] have proposed building database management systems that support the nested relational database model and provide a flat relational interface. Our result indicates that such systems have the

potential to optimize the user's query, expressed in a flat relational query language, by using properties of the nested algebra [10, 40]. Alternatively, it is an interesting research problem to study the degree to which current relational query optimizers can be extended to take advantage of this result.

2. FORMALISM

In this section we define *relation schemes*, *relation instances*, *nested algebra*, and *nested calculus*.

2.1 Scheme of a Relation

We define *schemes* to be lists of attributes. Two kinds of attributes are considered: *atomic attributes*, represented by their name, and *structured attributes*, represented by their name followed by their scheme.

$$\langle Attribute\rangle \rightarrow \langle Identifier\rangle$$

Such an attribute is called *atomic*, and $\langle Identifier\rangle$ is called the *name* of the attribute.

$$\langle Attribute\rangle \rightarrow \langle Identifier\rangle\langle Scheme\rangle$$

Such an attribute is called *structured*, and $\langle Identifier\rangle$ is called the *name* of the attribute.

$$\begin{aligned}\langle List\_of\_attributes\rangle &\rightarrow \langle Empty\_list\rangle \\ &\quad \mid \langle Non\_empty\_list\_of\_attributes\rangle \\ \langle Non\_empty\_list\_of\_attributes\rangle &\rightarrow \langle Attribute\rangle \mid \langle Attribute\rangle, \\ &\quad \langle Non\_empty\_list\_of\_attributes\rangle\end{aligned}$$

Two lists of attributes λ_1 and λ_2 are called *compatible* (i.e., the (ordered) attributes have the same nesting structure) if and only if (iff) they are both empty, or $\lambda_1 = \alpha_1, \lambda'_1$ and $\lambda_2 = \alpha_2, \lambda'_2$ with λ'_1, λ'_2 compatible lists of attributes and α_1 and α_2 both atomic attributes or both structured attributes with compatible schemes.

$$\langle Scheme\rangle \rightarrow (\_\langle Non\_empty\_list\_of\_attributes\rangle\_)$$

All of the identifiers in a scheme have to be different. Two schemes are called *compatible* iff their respective lists of attributes are compatible. A scheme is called *flat* iff all of its attributes are atomic.

These are some examples of schemes:

$$(A, B, C)$$

$$(A, E(D(B), C))$$

A *database scheme* is a set of schemes.

2.2 Instances of a Relation Scheme

Let S be the scheme $(\alpha_1, \ldots, \alpha_n)$, where α_i stands for an attribute, either atomic or structured. The *instances* of S, denoted $Inst(S)$, are the set

$$\{s \mid s \text{ is a finite subset of } values(\alpha_1) \times \cdots \times values(\alpha_n)\},$$

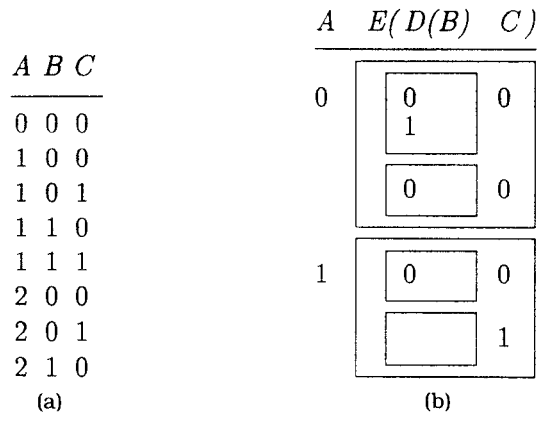

Fig. 1. (a) Instance s_1 of $Inst(A, B, C)$; (b) instance s_2 of $Inst(A, E(D)(B), C)$.

where $values(A)$ is the set of natural numbers for A, an atomic attribute, and $values(A(\lambda)) = Inst((\lambda))$, where λ is a nonempty list of attributes.

We need to make the following remarks:

—All atomic attributes have the same *values* set, namely, the set of the natural numbers.

—Two schemes are compatible iff their sets of instances are equal.

In Figure 1 we show an instance s_1 of $Inst(A, B, C)$ and an instance s_2 of $Inst(A, E(D(B), C))$, respectively.

2.3 Nested Algebra

We define a nested algebra for manipulating schemes and their instances, similar to the one introduced by Thomas and Fischer [43]. It should be noted that the empty operator and the renaming operator were not considered in [43]. The empty operator is introduced here to create empty structured component values; the nest operator cannot create such values, intuitively because nesting an empty instance again yields an empty instance and *not* an instance with a single tuple equal to the empty set. The algebra consists of nine operators, which are defined as follows:

(1) *Union operator* $\cup$: Let (λ_1) and (λ_2) be compatible schemes, and let $s_1 \in Inst((\lambda_1))$ and $s_2 \in Inst((\lambda_2))$. Then $s_1 \cup s_2$ is the (standard) union of s_1 and s_2 and is an instance of the scheme (λ_1).

(2) *Difference operator* $-$: Let (λ_1) and (λ_2) be compatible schemes, and let $s_1 \in Inst((\lambda_1))$ and $s_2 \in Inst((\lambda_2))$. Then $s_1 - s_2$ is the (standard) difference of s_1 and s_2 and is an instance of the scheme (λ_1).

(3) *Cross product operator* $\times$: Let (λ_1) and (λ_2) be schemes with no common identifiers, and let $s_1 \in Inst((\lambda_1))$ and $s_2 \in Inst((\lambda_2))$. Then $s_1 \times s_2$ is the (standard) cross product of s_1 and s_2 and is an instance of the scheme (λ_1, λ_2).

A	$D(B)$	C
0	0 1	0
0	0	0
1	0	0
1		1

(a)

A	B	C
0	0	0
0	1	0
1	0	0

(b)

Fig. 3. (a) Instance $\mu_{E(D(B),C)}(s_2)$; (b) instance $\mu_{D(B)}(\mu_{E(D(B),C)}(s_2))$.

(8) *Empty operator* $\varnothing$: Let (λ) be a scheme, and let A be no identifier of λ. Let $s \in Inst((\lambda))$. Then $\varnothing_A(s)$ is an instance of the scheme $(A(\lambda))$ that has only one element, being the empty instance of the scheme (λ).

(9) *Renaming operator* ρ: Let (λ) be a scheme, and let $s \in Inst((\lambda))$. Then $\rho(s, A, B)$ is the (standard) renaming of A by B in s. It is an instance of the scheme (λ), with the identifier A replaced by the identifier B.

Algebraic expressions of the nested algebra are defined in the usual way. An expression in the nested algebra is called a *flat-flat expression (ff-expression)* iff its operands and its result are flat. Here are some examples of expressions in the nested algebra. Reconsider instance s_1 of Figure 1.

$$\Pi_1(s_1) \times \Pi_2(s_1)$$
$$\mu_{E(C)}(\mu_{D(B)}(\sigma_{2=3}(\nu_{C;E}(\nu_{B;D}(s_1)))))$$
$$\Pi_1(\sigma_{2=3}(\nu_{C;E}(\nu_{B;D}(s_1))))$$
$$\Pi_1(s_1) \times \varnothing_D(s_1) \times \Pi_3(s_1)$$

The first, second, and third examples are ff-expressions; the fourth is not. The results of the second, third, and fourth expressions are shown in Figure 4.

2.4 Nested Calculus

We define a calculus for manipulating schemes and their instances, similar to the ones introduced by Roth, Korth, and Silberschatz [37] and by Abiteboul and Beeri [1].

A *query* in the nested calculus has the form

$$\{t[\lambda] \mid f(t)\}$$

where t is an identifier, called the *target variable*; λ is a nonempty list of attributes, called the *scheme* of t; and $f(t)$ is a formula. The scheme of this query is (λ). The free variables of this query are those of f, except for t.

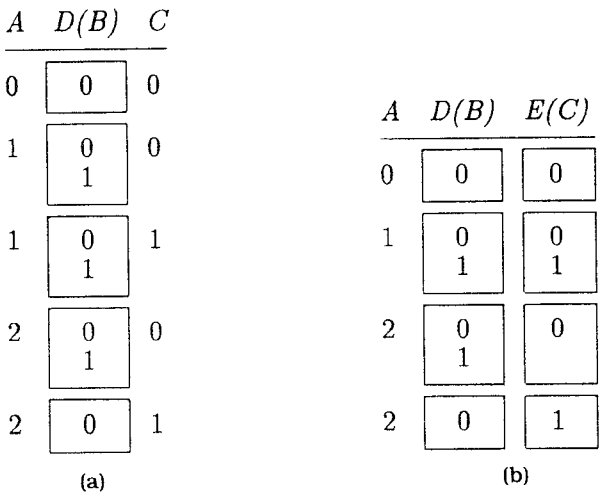

Fig. 2. (a) Instance $\nu_{B,D}(s_1)$; (b) instance $\nu_{C,E}(\nu_{B,D}(s_1))$.

(4) *Projection operator* Π: Let (λ) be the scheme $(\alpha_1, \ldots, \alpha_n)$, let $i_1, \ldots, i_k$ $(k \geq 1)$ be the indexes of different attributes $\alpha_{i_1}, \ldots, \alpha_{i_k}$ of λ, and let $s \in Inst((\lambda))$. Then $\Pi_{i_1, \ldots, i_k}(s)$ is the (standard) projection on the α_{i_1} through α_{i_k} attributes and is an instance of the scheme $(\alpha_{i_1}, \ldots, \alpha_{i_k})$.

(5) *Selection operator* σ: Let (λ) be a scheme; let i and j be the indexes of different attributes of (λ), and let $s \in Inst((\lambda))$. Then $\sigma_{i=j}(s)$ is the largest subset of s consisting of elements with equal i and j components.

(6) *Nest operator* ν: Let the scheme (λ) have the form (λ_1, λ_2), where λ_2 is not the empty list, and let A be no identifier of λ. Let $s \in Inst((\lambda))$. Then $\nu_{\lambda_2, A}(s)$ is the (standard) nest of s on the attributes of λ_2 and is an instance of the scheme $(\lambda_1, A(\lambda_2))$. $\nu_{\lambda_2; A}(s)$ is the set of all elements t_ν for which there is an element t in s such that t_ν restricted to λ_1 is equal to t restricted to λ_1, and $t_\nu(A(\lambda_2)) = \{v$ restricted to $\lambda_2 \mid v \in s$ and v restricted to λ_1 is equal to t_ν restricted to $\lambda_1\}$. Notice that in this definition we have made the notational simplifying assumption that λ_2 is an end sequence of λ and not a subsequence of λ. Reconsider instance s_1 of Figure 1. The instances $\nu_{B,D}(s_1)$ and $\nu_{C,E}(\nu_{B;D}(s_1))$ are shown in Figure 2.

(7) *Unnest operator* μ: Let the scheme (λ) have the form $(\lambda_1, A(\lambda_2))$, and let $s \in Inst((\lambda))$. Then $\mu_{A(\lambda_2)}(s)$ is the (standard) unnest of s on the $A(\lambda_2)$ attribute of λ and is an instance of the scheme (λ_1, λ_2). $\mu_{A(\lambda_2)}(s)$ is the set of all elements t_μ for which there is a element t in s such that t_μ restricted to λ_1 is equal to t restricted to λ_1 and t_μ restricted to λ_2 is an element of $t(A(\lambda_2))$. Observe that the application of the unnest operator can lose information when applied to instances containing empty-valued tuple components. Reconsider instance s_2 of Figure 1. The instances $\mu_{E(D(B),C)}(s_2)$ and $\mu_{D(B)}(\mu_{E(D(B),C)}(s_2))$ are shown in Figure 3.

A	B	C
0	0	0
1	0	0
1	0	1
1	1	0
1	1	1

A
0
1

A	D(B)	C
0		0
0		1
1		0
1		1
2		0
2		1

Fig. 4. Results of some nested algebraic expressions.

A *formula* can have one of the forms

$$(f_1 \vee \cdots \vee f_n)$$
$$(f_1 \wedge \cdots \wedge f_n)$$
$$\neg f_1$$
$$\exists t[\lambda]\, f_1$$

where all f_is are formulas; t is an identifier, called a *tuple variable* (all these ts are different and differ from the target variable); and λ is a nonempty list of attributes, called the scheme of t. Furthermore, a formula can have one of the forms

$$\langle term_1 \rangle = \langle term_2 \rangle$$
$$\langle term_1 \rangle \in \langle term_2 \rangle.$$

A *term* is a query and has one of the forms

TRUE
FALSE
t
$t(\alpha)$ ($t(\alpha)$ is called a *component* of t)
R

where t is a tuple variable, R is a relation identifier, and α is an attribute (atomic or structured). The scheme of $t(A)$, where A is an atomic attribute, is A. The scheme of $t(A(\lambda))$, where λ is a nonempty list of attributes and $A(\lambda)$ is a structured attribute, is (λ).

Note that $\langle term_1 \rangle = \langle term_2 \rangle$ is only a formula if the schemes of $\langle term_1 \rangle$ and $\langle term_2 \rangle$ are compatible, and that $\langle term_1 \rangle \in \langle term_2 \rangle$ is only a formula if the scheme of $\langle term_1 \rangle$ is λ and the scheme of $\langle term_2 \rangle$ is compatible with (λ).

Here are some examples of queries; R has the scheme $(A, B(C, D))$.

$$\{t[A] \mid \exists t_1[A, B(C, D)](t_1 \in R \wedge t(A) = t_1(A))\}$$

$$\{t[A, B(C, D)] \mid \neg\, t \in R\}$$

$$\{t_1[C] \mid \exists t_2[B(C)](t_2(B(C)) = \{t_3[C] \mid t_3 \in t_2(B(C))\} \wedge t_1 \in t_2(B(C)))\}$$

The first query is a projection, and the second, an (unsafe) complementation. In the third query, note how $t_2(B(C))$ is defined in terms of itself.

2.5 Flat Algebra

An expression of the nested algebra is also an expression of the *flat algebra* iff

(1) every attribute of every relation that occurs in the expression is atomic (notice that this eliminates the unnest operator from being present in such an expression), and
(2) neither the nest operator nor the empty operator occurs in the expression.

The first example in Section 2.3 is an expression of the flat algebra.

2.6 Flat Calculus

A query of the nested calculus is also a query of the *flat calculus* iff

(1) every attribute of every relation that occurs in the query is atomic,
(2) every attribute of the scheme of every variable that occurs in the query is atomic, and
(3) no term is a query.

3. TRANSLATION OF THE NESTED ALGEBRA TO THE NESTED CALCULUS

In this section we show how expressions of the nested algebra can be translated into the nested calculus. Let nae, nae_1, and nae_2 be expressions of the nested algebra.

—R, a relation with scheme (λ) (R denotes an element of $Inst((\lambda))$) is translated to $\{t[\lambda] \mid t \in R\}$.

We suppose that the nested algebraic expressions nae, nae_1, and nae_2 are, respectively, translated to $\{t[\lambda] \mid f(t)\}$, $\{t[\lambda_1] \mid f_1(t)\}$, and $\{t[\lambda_2] \mid f_2(t)\}$.

—*Union:* $nae_1 \cup nae_2$ (where λ_1 and λ_2 are compatible) is translated to $\{t[\lambda_1] \mid (f_1(t) \vee f_2(t))\}$.
—*Difference:* $nae_1 - nae_2$ (where λ_1 and λ_2 are compatible) is translated to $\{t[\lambda_1] \mid (f_1(t) \wedge \neg\, f_2(t))\}$.

—*Cross product:* $nae_1 \times nae_2$ (where λ_1 and λ_2 have no common identifiers) is translated to

$$\left\{t[\lambda_1, \lambda_2] \mid \left(\exists t_1[\lambda_1]\left(f_1(t_1) \bigwedge_{\alpha \in \lambda_1} t(\alpha) = t_1(\alpha)\right)\right.\right.$$
$$\left.\left. \wedge \exists t_2[\lambda_2]\left(f_2(t_2) \bigwedge_{\alpha \in \lambda_2} t(\alpha) = t_2(\alpha)\right)\right)\right\}.$$

—*Projection:* $\Pi_{i_1 \cdots i_k}$ (*nae*) (where $k \geq 1$ and $i_1, \ldots, i_k$ are the indexes of different attributes $\alpha_{i_1}, \ldots, \alpha_{i_k}$ of λ) is translated to $\{t[\alpha_{i_1}, \ldots, \alpha_{i_k}] \mid \exists t_1(\lambda)(f(t_1) \wedge \bigwedge_{j=1}^{k} t(\alpha_{i_j}) = t_1(\alpha_{i_j}))\}$.

—*Selection:* $\sigma_{i=j}(nae)$ (where i and j are the indexes of different attributes α_i and α_j of λ) is translated to $\{t[\lambda] \mid (f(t) \wedge t(\alpha_i) = t(\alpha_j))\}$.

—*Nest:* Suppose that λ has the form λ^1, λ^2 (where λ^2 is a nonempty list of attributes), and let A be no identifier of λ. $\nu_{\lambda^2; A}(nae)$ is translated to

$$\left\{t[\lambda^1, A(\lambda^2)] \mid \exists t_1[\lambda]\left(f(t_1) \bigwedge_{\alpha \in \lambda^1} t(\alpha) = t_1(\alpha)\right.\right.$$
$$\wedge t(A(\lambda^2)) = \left\{t_2[\lambda^2] \mid \exists t_3[\lambda]\left(f(t_3)\right.\right.$$
$$\left.\left.\left.\left. \times \bigwedge_{\alpha \in \lambda^1} t(\alpha) = t_3(\alpha) \bigwedge_{\alpha \in \lambda^2} t_2(\alpha) = t_3(\alpha)\right)\right\}\right)\right\}.$$

—*Unnest:* Suppose that λ has the form $\lambda^1, A(\lambda^2)$. $\mu_{A(\lambda^2)}(nae)$ is translated to

$$\left\{t[\lambda^1, \lambda^2] \mid \exists t_1[\lambda] \exists t_2[\lambda^2]\left(f(t_1) \bigwedge_{\alpha \in \lambda^1} t(\alpha)\right.\right.$$
$$\left.\left. = t_1(\alpha) \wedge t_2 \in t_1(A(\lambda^2)) \bigwedge_{\alpha \in \lambda^2} t(\alpha) = t_2(\alpha)\right)\right\}.$$

—*Empty:* Let (λ) be the scheme of *nae*, and let A be no identifier of λ. $\varnothing_A(nae)$ is translated to

$$\{t[A(\lambda)] \mid t(A(\lambda)) = \{t_1[\lambda] \mid f(t_1) \wedge \neg f(t_1)\}\}.$$

—*Renaming:* $\rho(nae, A, B)$ is translated to the translation of *nae* where every occurrence of A is replaced by B.

4. TRANSLATION OF ff-EXPRESSIONS INTO EXPRESSIONS OF THE FLAT ALGEBRA

In this section we show that ff-expressions of the nested algebra can be translated into equivalent expressions of the flat algebra. To establish this result, we first translate the ff-expression into a calculus query that satisfies

certain syntactic restrictions. We call such a query *constructive*. We then translate, with the help of several transformation rules, this constructive query into a safe query (in the sense of Ullman [44]) of the flat calculus. We finally use Codd's theorem [11, 44] about the equivalence of the safe flat calculus and the flat algebra. We would like to state that it is an open problem to find a completely algebraic strategy to achieve this result.

4.1 Existential Normal Form

The syntactic restrictions on the intermediate calculus queries are captured by requiring that the relevant calculus formulas are in a certain normal form. Intuitively, this normal form is broad enough to allow the formulation of nested algebra expressions, but restricted enough to disallow the formulation of calculus queries that would otherwise require a powerset operation in an algebraic setting (see also [1]).

A (nested calculus) formula g is in *existential normal form (enf)* iff

$$g \equiv (h_1 \vee \cdots \vee h_k), \qquad k \geq 1,$$

such that for all i, $1 \leq i \leq k$,

$$h_i \equiv \exists v_{i1}[\lambda_{i1}] \cdots \exists v_{in_i}[\lambda_{in_i}](c_{i1} \wedge \cdots \wedge c_{il_i} \wedge \neg d_i), \qquad n_i \geq 0, l_i \geq 0,$$

such that d_i is in enf or d_i is *FALSE* (in which case "$\wedge \neg d_i$" is not written) and

$$c_{ij} \equiv t_1(\alpha_1) = t_2(\alpha_2),$$

or

$$c_{ij} \equiv t_1 \in t_2(\alpha),$$

or

$$c_{ij} \equiv t_1(A(\lambda)) = \{t_2[\lambda] \mid f\}, \qquad \text{where } f \text{ is in enf,}$$

or

$$c_{ij} \equiv \{t_2[\lambda] \mid f\} = t_1(A(\lambda)), \qquad \text{where } f \text{ is in enf,}$$

or

$$c_{ij} \equiv t \in R,$$

or

$$c_{ij} \equiv t_1 \in \{t_2[\lambda] \mid f\}, \qquad \text{where } f \text{ is in enf,}$$

or

$$c_{ij} \equiv \{t_1[\lambda_1] \mid f_1\} = \{t_2[\lambda_2] | f_2\}, \qquad \text{where } f_1 \text{ and } f_2 \text{ are in enf.}$$

The formulas of the three examples in Section 2.4 are in enf.

We now define the disjunction, the conjunction, the negation, and the existential quantification for formulas in enf so that the resulting formulas again are in enf. Let

$$g \equiv \left(\bigvee_{i=1}^{k} h_i \right)$$

with

$$h_\iota \equiv \exists v_{i1}[\lambda_{\iota 1}] \cdots d\exists v_{\iota n_\iota}[\lambda_{\iota n_\iota}]\left(\bigwedge_{j=1}^{l_\iota} c_{\iota j} \wedge \neg d_\iota\right),$$

and let

$$g' \equiv \left(\bigvee_{\iota'=1}^{k'} h'_{\iota'}\right)$$

with

$$h'_{\iota'} \equiv \exists v'_{\iota' 1}[\lambda'_{\iota' 1}] \cdots \exists v'_{\iota' n'_{\iota'}}[\lambda'_{\iota' n'_{\iota'}}]\left(\bigwedge_{j'=1}^{l'_{\iota'}} c'_{\iota' j'} \wedge \neg d'_{\iota'}\right).$$

The following definitions result in enf formulas:

$$(g \stackrel{\vee}{-} g') \equiv (g \vee g'),$$

$$(g \stackrel{\wedge}{-} g') \equiv \left(\bigvee_{\iota=1}^{k} \bigvee_{\iota'=1}^{k'} h_{\iota\iota'}\right)$$

with

$$h_{\iota\iota'} \equiv \exists v_{\iota 1}[\lambda_{\iota 1}] \cdots \exists v_{\iota n_\iota}[\lambda_{\iota n_\iota}] \exists v'_{\iota' 1}[\lambda'_{\iota' 1}] \cdots \exists v'_{\iota' n'_{\iota'}}[\lambda'_{\iota' n'_{\iota'}}]\left(\bigwedge_{j=1}^{l_\iota} c_{\iota j} \bigwedge_{j'=1}^{l'_{\iota'}} c'_{\iota' j'} \wedge \neg (d_\iota \vee d'_{\iota'})\right),$$

$$\stackrel{\neg}{-} g \equiv \neg g, \underline{\exists} t[\lambda] g \equiv \left(\bigvee_{i=1}^{k} h''_i\right),$$

where $h''_i \equiv \exists t[\lambda] h_\iota$.

In the following lemma, we establish some simple logical equivalences:

LEMMA 1. *For each formula f_1 and f_2, we have*

(*A*) *If t_2 is not a free variable of f_1, then $(f_1 \wedge \exists t_2[\lambda] f_2(t_2))$ is logically equivalent to $\exists t_2[\lambda](f_1 \wedge f_2(t_2))$.*

(*B*) *If t_2 is not a free variable of $f_1(t_1)$ and t_1 is not a free variable of $f_2(t_2)$, then $(\exists t_1[\lambda_1] f_1(t_1) \wedge \exists t_2[\lambda_2] f_2(t_2))$ is logically equivalent to $\exists t_1[\lambda_1] \exists t_2[\lambda_2](f_1(t_1) \wedge f_2(t_2))$.*

(*C*) *$\exists t[\lambda](f_1(t) \vee f_2(t))$ is logically equivalent to $(\exists t[\lambda] f_1(t) \vee \exists t[\lambda] f_2(t))$.*

LEMMA 2. *$(g \stackrel{\vee}{-} g')$ (resp. $(g \stackrel{\wedge}{-} g')$, $\stackrel{\neg}{-} g$, $\underline{\exists} t[\lambda] g$) is logically equivalent to $(g \vee g')$ (resp. $(g \wedge g')$, $\neg g$, $\exists t[\lambda] g$).*

PROOF. This results from Lemma 1 and elementary logic. □

4.2 Constructive Queries

We start this section by defining *constructive queries*. Intuitively, this class of queries admits the formulation of all nested algebra expressions

(Lemma 6); however, queries that would require powerset operations in an algebraic setting cannot be expressed as constructive queries.

In constructive queries, all of the components of existentially quantified tuple variables have to be reachable in a constructibility graph in a *non*recursive (acyclic) way from relation or subquery nodes. Also, the components of tuple variables corresponding to the result of a query have to satisfy this property; these components are collected in a set denoted by F.

Later, in Section 4.3, we transform every constructive query into a flat constructive query. The *safety* of this query (in the sense of [44]) will be established in Lemma 8. Codd's theorem [11] regarding the translation of safe queries into relational algebra will then facilitate the final step in the translation.

Let g be in enf, and let $F = \{u_1(\alpha_{p_1}), \ldots, u_l(\alpha_{pl})\}$ be a set[1] of components of the variables $u_1, \ldots, u_l$ (these are called the variables of F) with

$$g \equiv (h_1 \vee \cdots \vee h_k),$$
$$h_i \equiv \exists v_{i1}[\lambda_{i1}] \cdots \exists v_{in_i}[\lambda_{in_i}](c_{i1} \wedge \cdots \wedge c_{il_i} \wedge \neg d_i).$$

We now define the constructibility graph of (h_i, F). An edge (x, y) denotes informally that the set of possible values of y is "bounded" by the set of possible values of x. The *constructibility graph* of (h_i, F), notated as $C(h_i, F)$, is the directed graph with the following nodes:

(1) all components of the tuple variables $v_{i1}, \ldots, v_{in_i}$ and all the elements of F;
(2) all of the tuple variables $v_{i1}, \ldots, v_{in_i}$ and the variables $u_1, \ldots, u_l$;
(3) all queries Q that appear in h_i on the highest level in some c_{ij}; and
(4) all relations R that appear in h_i on the highest level in some c_{ij};

and with the following edges:

(1) If $t_1(\alpha_1) = t_2(\alpha_2)$ is one of the c_{ij}s, then $(t_1(\alpha_1), t_2(\alpha_2))$ is an edge if $t_1(\alpha_1) \notin F$ and $(t_2(\alpha_2), t_1(\alpha_1))$ is an edge if $t_2(\alpha_2) \notin F$;
(2) if $t_1 \in t_2(\alpha)$ is one of the c_{ij}s, then $(t_2(\alpha), t_1)$ is an edge if $t_2(\alpha) \notin F$;
(3) if $t(A(\lambda)) = Q$ or $Q = t(A(\lambda))$ is one of the c_{ij}s, Q being a query, then $(Q, t(A(\lambda)))$ is an edge;
(4) if $t \in R$ is one of the c_{ij}s, then (R, t) is an edge;
(5) if $t \in Q$ is one of the c_{ij}s, Q being a query, then (Q, t) is an edge; and
(6) if t is a variable and $t(\alpha)$ is a component of t, then $(t, t(\alpha))$ is an edge.

A component $t(\alpha)$ is called *reachable* in a subgraph $C'(h_i, F)$ of $C(h_i, F)$ iff there is a path in $C'(h_i, F)$ from a query Q or a relation R to $t(\alpha)$. A component $t(\alpha)$ is called *reachable* in (h_i, F) iff $t(\alpha)$ is reachable in $C(h_i, F)$.

[1] The notions of constructibility graph, reachability, acyclicity, and constructiveness are defined relative to a set of variable components F. These variable components correspond to free variable components of the involved enfs (see, in particular, the definition of a constructive query $\{u[\lambda] \mid g\}$ in Section 4.2).

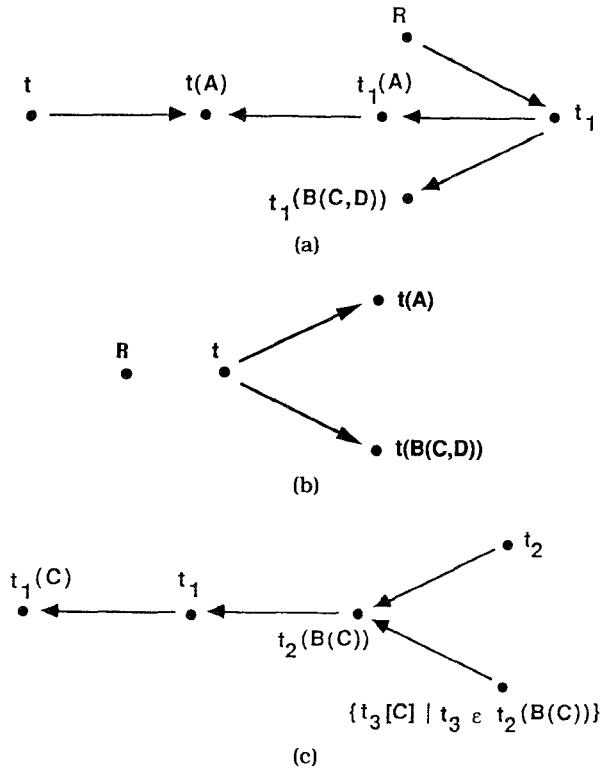

Fig. 5. Constructibility graphs of the three examples in Section 2.4. (a) $C(\exists t_1[A, B(C, D)](t_1 \in R \wedge t(A) = t_1(A)), \{t(A)\})$; (b) $C(\neg\, t \in R, \{t(A), t(B(C, D))\})$; (c) $C(\exists t_2[B(C)](t_2(B(C)) = \{t_3[C] \mid t_3 \in t_2(B(C))\} \wedge t_1 \in t_2(B(C)), \{t_1(C)\})$.

The component $t(\alpha)$ is called *reachable* in (g, F) iff it is reachable in (h_i, F), for all i, $1 \leq i \leq k$. In Figure 5a $t(A)$, $t_1(A)$, and $t_1(B(C, D))$ are reachable. In Figure 5b no component is reachable. In Figure 5c $t_2(B(C))$ and $t_1(C)$ are reachable.

Informally, (h_i, F) is called acyclic if all of its v variables and all of the elements of F are reachable in an acyclic subgraph of $C(h_i, F)$. Formally, (h_i, F) is called *acyclic* iff $C(h_i, F)$ has a subgraph $A(h_i, F)$ such that

(1) all of the components of the tuple variables $v_{i1}, \ldots, v_{in_i}$ (called the v *variables* in the sequel) and all of the elements of F are reachable in $A(h_i, F)$; and

(2) $A(h_i, F)$ augmented with all the edges $(t(\alpha), Q)$, with $t(\alpha)$ appearing in the query Q, is acyclic. We call the latter graph $AU(h_i, F)$.

Informally, (h_i, F) is called constructive if it is acyclic and if the negated part, d_i, and all of the subqueries that occur in h_i are constructive. Formally, (h_i, F) is called *constructive* iff

(1) (h_i, F) is acyclic;

(2) $(d_i, \varnothing)$ is constructive;

(3) if some c_{ij} has the form

$$v(\alpha) = \{w[\gamma] \mid f\},$$

or

$$\{w[\gamma] \mid f\} = v(\alpha),$$

or

$$v \in \{w[\gamma] \mid f\},$$

then $\{w[\gamma] \mid f\}$ is constructive;

(4) if some c_{ij} has the form

$$\{w_1[\gamma_1] \mid f_1\} = \{w_2[\gamma_2] \mid f_2\},$$

then $\{w_1[\gamma_1] \mid f_1\}$ and $\{w_2[\gamma_2] \mid f_2\}$ are constructive.

(g, F) is called *constructive* iff for all i, $1 \leq i \leq k$, (h_i, F) is constructive. The query $\{u[\lambda] \mid g\}$ is called *constructive* iff (g, F) is constructive, where $F = \{u(\alpha) \mid \alpha \in \lambda\}$. Note that, whenever $G \subset F$, (g, G) is constructive if (g, F) is also constructive.

The definition of the constructiveness of (h_i, F) is clearly *recursive* (see points (2), (3), and (4)). However, it is clear that in spite of the recursion this (purely syntax-oriented) definition is well defined.

The first example of Section 2.4 is constructive; the second is not (since neither component of t is reachable in $(\neg t \in R, \{t(A), t(B(C, D))\})$ (Figure 5b), and neither is the third (since the graph AU would be cyclic and therefore does not exist, as can be seen from Figure 5c).

In Lemmas 3–5 we show how constructiveness is preserved in various situations.

LEMMA 3. *Let g and g' be formulas, and let F and F' be sets of components.*

(*A*) *If (g, F) and (g', F) are both constructive, then $(g \stackrel{\vee}{-} g', F)$ is constructive.*

(*B*) *If (g, F) and (g', F') are both constructive and if g and g' have no common components or common variables, then $(g \stackrel{\wedge}{-} g', F \cup F')$ is constructive.*

(*C*) *If (g, F) and $(g', \varnothing)$ are both constructive, then $(g \stackrel{\wedge}{-} \neg g', F)$ is constructive.*

(*D*) *If (g, F) is constructive and F contains the set of all of the components of the variable t, then $(\exists t[\lambda]g, F_1)$ is constructive, F_1 being the set of components of F that are not components of t.*

PROOF

(A) Trivial, since $(g \vee g') \equiv (g \stackrel{\vee}{-} g')$.

(B) Clearly, $C(h_{ii'}, F \cup F') = C(h_i, F) \cup C(h'_{i'}, F')$. Now take $A(h_{ii'}, F \cup F') = A(h_i, F) \cup A(h'_{i'}, F')$; then $AU(h_{ii'}, F \cup F') = AU(h_i, F) \cup AU(h'_{i'}, F')$. Since g and g' have no components or common variables (and, hence, clearly no common queries), and since $AU(h_i, F)$ and

$AU(h'_{i'}, F')$ are both acyclic, there could only be a cycle in $AU(h_{ii'}, F \cup F')$ that contains a relation R. This is impossible since no arc ends in R.

(C) Clearly, $(\neg g', \emptyset)$ is constructive (see (2) of the definition of constructiveness), so by (B) we see that $(g \stackrel{\wedge}{=} \neg g', F)$ is constructive.

(D) We have to prove that (h'', F_1) is constructive. We can choose $A(h''_i, F_1)$ equal to $A(h_i, F)$. It should be clear that $AU(h''_i, F_1) = AU(h_i, F)$, which is an acyclic graph, and hence, (h''_i, F_1) is acyclic. We know that $(d_i, \emptyset)$ is constructive. Also, if a query occurs as a term, it remains constructive. This proves that (h''_i, F_1) is constructive, and therefore, $(\underline{\exists} t[\lambda]g, F_1)$ is constructive. □

LEMMA 4. *Let g be a formula, and let F be a set of components. If (g, F) is constructive and $t_1(\alpha_1)$ and $t_2(\alpha_2)$ are two elements of F, then $((g \stackrel{\wedge}{=} t_1(\alpha_1) = t_2(\alpha_2)), F)$ is constructive.*

PROOF. Clearly, $(g \stackrel{\wedge}{=} t_1(\alpha_1) = t_2(\alpha_2))$ is in enf and $((h_i \wedge t_1(\alpha_1) = t_2(\alpha_2)), F)$ is acyclic, since we can choose $A((h_i \wedge t_1(\alpha_1) = t_2(\alpha_2)), F) = A(h_i, F)$. Hence, $((h_i \wedge t_1(\alpha_1) = t_2(\alpha_2)), F)$ is constructive. □

LEMMA 5. *Let g be a formula, and let F be a set of components. Let $(\underline{\exists} t[\lambda]g, F)$ be constructive.*

(*A*) *If $t_1(\alpha_1)$ does not appear in $\underline{\exists} t[\lambda]g$, then $(\underline{\exists} t[\lambda](g \stackrel{\wedge}{=} t(\alpha) = t_1(\alpha_1)), F \cup \{t_1(\alpha_1)\})$ is constructive.*

(*B*) *If $t(A(\lambda_1))$ is a component of t and no component of t_1 or t_1 itself appears in $\underline{\exists} t[\lambda]g$, then $(\underline{\exists} t[\lambda](g \stackrel{\wedge}{=} t_1 \in t(A(\lambda_1))), F \cup \{t_1(\alpha) \mid \alpha \in \lambda_1\})$ is constructive.*

(*C*) *If Q is a constructive query and $t_1(\alpha)$ does not appear in $\underline{\exists} t[\lambda]g$, then $(\underline{\exists} t[\lambda](g \stackrel{\wedge}{=} t_1(\alpha) = Q), F \cup \{t_1(\alpha)\})$ is constructive.*

PROOF

(A) Let

$$\bar{h}_i \equiv \exists t[\lambda](h_i \wedge t(\alpha) = t_1(\alpha_1))$$

be the ith component of $\underline{\exists} t[\lambda](g \stackrel{\wedge}{=} t(\alpha) = t_1(\alpha_1))$, and let $\bar{F}$ be $F \cup \{t_1(\alpha_1)\}$. We have to prove that $(\bar{h}_i, \bar{F})$ is constructive. We first show that it is acyclic. We can choose $A(\bar{h}_i, \bar{F})$ as $A(\exists t[\lambda]h_i, F)$ plus the edge $(t(\alpha), t_1(\alpha_1))$. Hence, $AU(\bar{h}_i, \bar{F})$ is $AU(\exists t[\lambda]h_i, F)$ plus the edge $(t(\alpha, t_1(\alpha_1))$. It is easy to see that all of the components of the v variables, the variable t and the elements of F and $t_1(\alpha_1)$ are reachable in $A(\bar{h}_i, \bar{F})$. Furthermore, $AU(\bar{h}_i, \bar{F})$ is acyclic because $t_1(\alpha_1)$ does not appear in $\exists t[\lambda]g$. Clearly $(d_i, \emptyset)$ is constructive, and queries appearing in $\bar{h}_i$ remain constructive.

(B) Let

$$\bar{h}_i = \exists t[\lambda](h_i \wedge t_1 \in t(A(\lambda_1)))$$

be the ith component of $\underline{\exists} t[\lambda](g \stackrel{\wedge}{=} t_i \in t(A(\lambda_1)))$, and let $\bar{F} = F \cup \{t_1(\alpha) \mid \exists \alpha \in \lambda_1\}$. We have to show that $(\bar{h}_i, \bar{F})$ is constructive. We

first show that it is acyclic. We can choose $A(\bar{h}_i, \bar{F})$ as $A(\exists t[\lambda]h_i, F)$ plus the edge $(t(A(\lambda_1)), t_1)$ and the edges $(t_1, t_1(\alpha))$ for all $\alpha \in \lambda_1$. $AU(\bar{h}_i, \bar{F})$ is $AU(\exists t[\lambda]h_i, F)$ plus the edges just mentioned. It is easy to see that all of the components of the v variables, the variable t and the elements of $\{t_1(\alpha) \mid \alpha \in \lambda_1\}$ and of F are reachable in $A(\bar{h}_i, \bar{F})$. Furthermore, $AU(\bar{h}_i, \bar{F})$ is acyclic because no components of t or t itself appears in g. Clearly, $(d_i, \varnothing)$ is constructive, and queries appearing in $\bar{h}$ remain constructive.

(C) Let

$$\bar{h}_i \equiv \exists t[\lambda](h_i \wedge t_1(\alpha) = Q)$$

be the ith component of $\exists t[\lambda](g \stackrel{\wedge}{-} t_1(\alpha) = Q)$, and let $\bar{F} = F \cup \{t_1(\alpha)\}$. We have to show that $(\bar{h}_i, \bar{F})$ is constructive. We first show that it is acyclic. We can choose $A(\bar{h}_i, \bar{F})$ as $A(\exists t[\lambda]h_i, F)$ plus the edge $(Q, t_1(\alpha))$. $AU(\bar{h}_i, \bar{F})$ is $AU(\exists t[\lambda]h_i, F)$ plus the edge $(Q, t_1(\alpha))$ and the edges (x, Q), where x appears in Q. It is easy to see that the components of the v variables, the variable t_1 and the elements of F and $t_1(\alpha)$ are reachable in $A(\bar{h}_i, \bar{F})$. Furthermore, $AU(\bar{h}_i, \bar{F})$ is acyclic because $t_1(\alpha)$ does not appear in g. Clearly, $(d_i, \varnothing)$ is constructive, and queries appearing in $\bar{h}_i$ remain constructive. □

Lemmas 3–5 together with Section 3, where we discussed translation of a nested algebra expression into a nested calculus query, allow us to establish the following result:

LEMMA 6. *Every expression of the nested algebra can be translated into a constructive query.*[2]

PROOF. Let *nae* be an expression of the nested algebra. We prove the lemma by induction on the number of operators in *nae*.

Induction hypothesis. Every expression of the nested algebra with fewer than n ($n \geq 0$) operators can be translated into a constructive query.

Basis step. ($n = 0$). Then $nae \equiv R$ (with scheme (λ)). This expression can be translated into the query $\{t[\lambda] \mid t \in R\}$, which is clearly a constructive query.

Induction step. Suppose *nae* has n operators ($n > 0$).

Binary operators. Then

$$nae \equiv nae_1 \cup nae_2,$$

or

$$nae \equiv nae_1 - nae_2,$$

or

$$nae \equiv nae_1 \times nae_2.$$

[2] Although it is not necessary for our results, we conjecture that the converse of this lemma is also true. A possible way to prove this would be to compare constructive queries with the calculus queries introduced in [37] and with the strictly safe calculus queries introduced in [1].

Since nae_1 and nae_2 have fewer than n operators, they can be translated into the constructive queries $\{t_1[\lambda_1] | f_1(t_1)\}$ and $\{t_2[\lambda_2] | f_2(t_2)\}$, respectively.

(i) $nae \equiv nae_1 \cup nae_2$. nae can be translated into $\{t_1[\lambda_1] | (f_1(t_1) \stackrel{\vee}{-} f_2(t_1))\}$. This query is constructive because of Lemma 3(A).

(ii) $nae \equiv nae_1 - nae_2$. nae can be translated into $\{t_1[\lambda_1] | (f_1(t_1) \stackrel{\wedge}{-} f_2(t_1))\}$. This query is constructive because of Lemma 3(C).

(iii) $nae \equiv nae_1 \times nae_2$. Notice that in this case λ_1 and λ_2 have no common identifiers. nae can be translated into

$$\{t[\lambda_1, \lambda_2] | (\exists t_1[\lambda_1](f_1(t_1) \stackrel{\wedge}{-}_{\alpha \in \lambda_1} t(\alpha) = t_1(\alpha)) \stackrel{\wedge}{-} \exists t_2[\lambda_2](f_2(t_2) \stackrel{\wedge}{-}_{\alpha \in \lambda_2} t(\alpha) = t_2(\alpha)))\}.$$

This query is constructive because of Lemmas 3(D), 5(A), and 3(B).

Unary operators. Then

$$nae \equiv \Pi_{i_1, \ldots, i_k}(nae_1),$$

or

$$nae \equiv \sigma_{i=j}(nae_1),$$

or

$$nae \equiv \nu_{\lambda^2; A}(nae_1),$$

or

$$nae \equiv \mu_{A(\lambda^1)}(nae_1),$$

or

$$nae \equiv \varnothing_A(nae_1),$$

or

$$nae \equiv \rho(nae_1).$$

Since nae_1 has fewer than n operators, it can be translated into the constructive query $\{t_1[\lambda_1] | f_1(t_1)\}$.

(i) $nae \equiv \Pi_{i_1, \ldots, i_k}(nae_1)$. nae can be translated into

$$\left\{t[\alpha_{i_1}, \ldots, \alpha_{i_k}] | \exists t_1[\lambda_1]\left(f_1(t_1) \stackrel{\wedge}{-}_{1 \le j \le k} t(\alpha_{i_j}) = t_1(\alpha_{i_j})\right)\right\}.$$

This query is constructive because of Lemmas 3(D) and 5(A).

(ii) $nae \equiv \sigma_{i=j}(nae_1)$. nae can be translated into

$$\{t_1[\lambda_1] | (f_1(t_1) \stackrel{\wedge}{-} t_1(\alpha_i) = t_1(\alpha_j))\}.$$

This query is constructive because of Lemma 3(B).

(iii) $nae \equiv \nu_{\lambda^2; A}(nae_1)$, with $\lambda_1 = \lambda^1, \lambda^2$. Then nae can be translated into

$$\{t[\lambda^1, A(\lambda^2)] | \exists t_1[\lambda](f(t_1) \stackrel{\wedge}{-}_{\alpha \in \lambda^1} t(\alpha) = t_1(\alpha) \stackrel{\wedge}{-} t(A(\lambda^2)) = \{t_2[\lambda^2] | \exists t_3[\lambda](f(t_3) \stackrel{\wedge}{-}_{\alpha \in \lambda^1} t(\alpha) = t_3(\alpha) \stackrel{\wedge}{-}_{\alpha \in \lambda^2} t_2(\alpha) = t_3(\alpha))\})\}.$$

This query is constructive because of Lemmas 3(D), 5(A), 5(C), and the fact that

$$\{t_2[\lambda^2] \mid \exists t_3[\lambda](f(t_3) \stackrel{\wedge}{=}_{\alpha\in\lambda^1} t(\alpha) = t_3(\alpha) \stackrel{\wedge}{=}_{\alpha\in\lambda^2} t_2(\alpha) = t_3(\alpha))\}$$

is a constructive query (which follows from Lemmas 3(D) and 5(A)).

(iv) $nae \equiv \mu_{A(\lambda^1)}(nae_1)$, with $\lambda_1 = A(\lambda^1), \lambda^2$. *nae* can be translated into

$$\{[\lambda^1, \lambda^2] \mid \exists t_1[\lambda^1] \exists t_2[\lambda^2](f_1(t_1) \stackrel{\wedge}{=}_{\alpha\in\lambda^1} t(\alpha) = t_1(\alpha) \stackrel{\wedge}{=} t_2 \in t_1(A(\lambda^1)) \stackrel{\wedge}{=}_{\alpha\in\lambda^2} t(\alpha) = t_2(\alpha))\}.$$

This query is constructive because of Lemmas 3(D), 5(A), and 5(B).

(v) $nae = \varnothing_A(nae_1)$. *nae* can be translated into

$$\{t[A(\lambda_1)] \mid t(A(\lambda_1)) = \{t_1[\lambda_1] \mid (f_1(t_1) \stackrel{\wedge}{=} \neg f_1(t_1))\}\}.$$

This query is constructive because of Lemmas 3(C) and 5(C).

(vi) $nae = \rho(nae_1, A, B)$. The renaming of a constructive query is a constructive query. □

4.3 Transformations Between Logically Equivalent Formulas

We now define several query transformation rules that preserve constructiveness and that will allow us to "flatten" constructive calculus queries. Four rules are considered: (1) The *query elimination* (T1) replaces subformulas of the form $t \in \{u[\lambda] \mid f(u)\}$ by the equivalent $f(t)$ subformulas. (2) The *query propagation* (T2) substitutes, if $t(\alpha) = Q$, every occurrence of $t(\alpha)$ by Q. Applying the query propagation repeatedly results in a query in which each structured component occurs at most once. (3) The *structured component elimination* (T3) then allows the removal of these components. After applying the structured component elimination, the query no longer contains structured components. The only reason for a query to be nonflat is occurrences of subformulas of the form $Q_1 = Q_2$ (where Q_1 and Q_2 are subqueries). (4) The *query equality elimination* can then be used to transform these subformulas.

Let g be a formula, let F be a set of components, and let (g, F) be constructive, with

$$g \equiv (h_1 \vee \cdots \vee h_k),$$
$$h_i \equiv \exists v_1[\lambda_1] \cdots \exists v_n[\lambda_n](c_1 \wedge \cdots \wedge c_l \wedge \neg d).$$

(T1) *Query membership elimination.* Let $c_j \equiv t_1 \in \{t_2[\lambda_2] \mid f(t_2)\}$. The query membership elimination rule substitutes g by g', with

$$g' \equiv (h_1 \vee \cdots \vee h'_{i-1} \vee h'_i \vee h_{i+1} \vee \cdots \vee h_k),$$
$$h'_i \equiv \exists v_1[\lambda_1] \cdots \exists v_n[\lambda_n]\left((c_1 \wedge \cdots \wedge c_{j-1} \wedge c_{j+1} \wedge \cdots \wedge c_l \wedge \neg d) \stackrel{\wedge}{=} f(t_1)\right).$$

(T2) *Query propagation.* Let $c_j \equiv t_1(\alpha) = Q$ or $c_j \equiv Q = t_1(\alpha)$ where $Q \equiv \{t_2[\lambda_2] | f(t_2)\}$, and let $(Q, t_1(\alpha))$ be an edge of $A(h_i, F)$. The query propagation rule substitutes g by g', with

$$g' \equiv (h_1 \vee \cdots \vee h_{i-1} \vee h'_i \vee h_{i+1} \vee \cdots \vee h_k),$$
$$h'_i \equiv \exists v_1[\lambda_1] \cdots \exists v_n[\lambda_n](c'_1 \wedge \cdots \wedge c'_{j-1} \wedge c_j \wedge c'_{j+1} \wedge \cdots \wedge c'_l \wedge \neg d'),$$

where c'_p $(p \neq j)$ (d', respectively) is c_p (d, respectively) with every occurrence of $t_1(\alpha)$ replaced by Q.

(T3) *Structured component elimination.* Consider the formula h_i, and suppose that c_j is the only term in which $v_m(A(\lambda))$ occurs. The structured component elimination rule substitutes g by g' in three steps:

(1) Delete the term c_j from h_i.
(2) Delete the attribute $A(\lambda)$ from the scheme of v_m.
(3) If the scheme of v_m becomes empty, remove $\exists v_m[\ ]$.

Call the resulting formula h'_i, and let

$$g' \equiv (h_1 \vee \cdots \vee h_{i-1} \vee h'_i \vee h_{i+1} \vee \cdots \vee h_k).$$

(T4) *Query equality elimination.* Let $c_j \equiv \{t_1[\gamma_1] | f_1(t_1)\} = \{t_2[\gamma_2] | f_2(t_2)\}$. The query equality elimination substitutes g by g', with

$$g' \equiv (h_1 \vee \cdots \vee h_{i-1} \vee h'_i \vee h_{i+1} \vee \cdots \vee h_k),$$
$$h'_i \equiv \exists v_1[\lambda_1] \cdots \exists v_n[\lambda_n](c_1 \wedge \cdots \wedge c_{j-1} \wedge c_{j+1} \wedge \cdots \wedge c_l \wedge$$
$$\neg (d \stackrel{\vee}{-} \exists t_1[\gamma_1](f_1(t_1) \stackrel{\wedge}{-} \neg f_2(t_1)) \stackrel{\vee}{-} \exists t_2[\gamma_2](f_2(t_2) \stackrel{\wedge}{-} \neg f_1(t_2)))).$$

LEMMA 7. *Query propagation, structured component elimination, query membership elimination, and query equality elimination preserve constructiveness.*

PROOF. In this proof we use the notation introduced in the transformation rules (T1), (T2), (T3), and (T4).

(T1) *Query membership elimination.* To show that query membership elimination preserves constructiveness, we first need to establish a property of constructibility graphs. Therefore, let

$$h_i \equiv \exists v_1[\lambda_1] \cdots \exists v_n[\lambda_n](c_1 \wedge \cdots \wedge c_l \wedge \neg d),$$

let F be a set of components, and assume that (h_i, F) is constructive; then $C(h_i, F)$ contains a subgraph $A'(h_i, F)$

—that satisfies the reachability and acyclicity conditions mentioned in the definition of constructibility graphs; and

—for which, for each component of the v variables and each element of F, there exists a unique query or relation from which the component or element can be reached in $A'(h_i, F)$.

To see that this is so, consider $C(h_i, F)$, and let $x(\alpha)$ be a component of either a v variable or an element of F. Since (h_i, F) is constructive, there exists a subgraph $A(h_i, F)$ of $C(h_i, F)$ that satisfies the reachability and acyclicity conditions mentioned in the definition of the constructibility graph. Suppose there exist two different nodes Q_1, Q_2 (each of which is either a query or a relation) in $A(h_i, F)$ from which $x(\alpha)$ is reachable (see Figure 6). Let y be the node where the paths from Q_1 and Q_2 to $x(\alpha)$ join for the first time, and let (z, y) be the edge of the path from Q_2 to $x(\alpha)$ pointing to y. Consider the subgraph of $A(h_i, F)$ where the edge (z, y) is removed. It should be clear that this subgraph still satisfies the reachability and acyclicity conditions mentioned in the definition of constructibility graphs and, furthermore, that in this graph $x(\alpha)$ can be reached via one less path than it could in $A(h_i, F)$. Repeated removal of such edges eventually results in a subgraph $A'(h_i, F)$ with the desired properties. (This establishes a result about constructibility graphs that will be necessary in the rest of the proof about the preservation of constructiveness by query membership elimination.)

Let $Q = \{t_2[\lambda_2] \mid f(t_2)\}$ (remember that $c_j \equiv t_1 \in \{t_2[\lambda_2] \mid f(t_2)\}$ in (T1) and, hence, that $c_j \equiv t_1 \in Q$, let

$$f(t_2) \equiv \big(p_1(t_2) \vee \cdots \vee p_m(t_2) \vee \cdots \vee p_r(t_2)\big),$$

and let

$$p_m(t_2) \equiv \exists w_{m1}[\lambda_{m1}] \cdots \exists w_{ms_m}[\lambda_{ms_m}]\big(q_{m1}(t_2) \wedge \cdots \wedge q_{mu_m}(t_2) \wedge \neg e_m(t_2)\big).$$

Then

$$h'_i \equiv \bigvee_{m=1}^{r} \exists v_1[\lambda_1] \cdots \exists v_n[\lambda_n] \exists w_{m1}[\lambda_{m1}] \cdots \exists w_{ms_m}[\lambda_{ms_m}]$$
$$\big(c_1 \wedge \cdots \wedge c_{j-1} \wedge c_{j+1} \wedge \cdots \wedge c_l \wedge q_{m1}(t_1) \wedge \cdots \wedge q_{mu_m}(t_1)$$
$$\wedge \neg \big(d \vee e_m(t_1)\big)\big).$$

Let h'_{im} be the mth disjunct of h'_i. We have to prove that (h'_{im}, F) is constructive. Consider h_i and $p_m(t_2)$. We know that (h_i, F) and $(p_m(t_2), G)$ (where $G = \{t_2(\beta) \mid t_2(\beta) \in \lambda_2\}$) are constructive. Consider the graphs $A'(h_i, F)$ and $A(p_m(t_2), G)$. There are two cases:

(1) (Q, t_1) is an edge of $A'(h_i, F)$ (Figure 7). Consider what happens in this situation when we "join" the graphs $A'(h_i, F)$ and $A(p_m(t_2), G)$ by applying the query membership elimination rule. Figure 8 shows the resulting graph $AU(h'_{im}, F)$. Clearly, $AU(h'_{im}, F)$ is $A(h'_{im}, F)$ plus

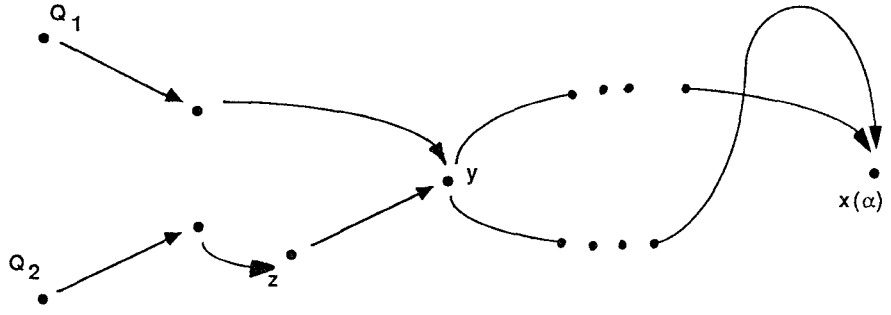

Fig. 6. $x(\alpha)$ is reachable from both Q_1 and Q_2.

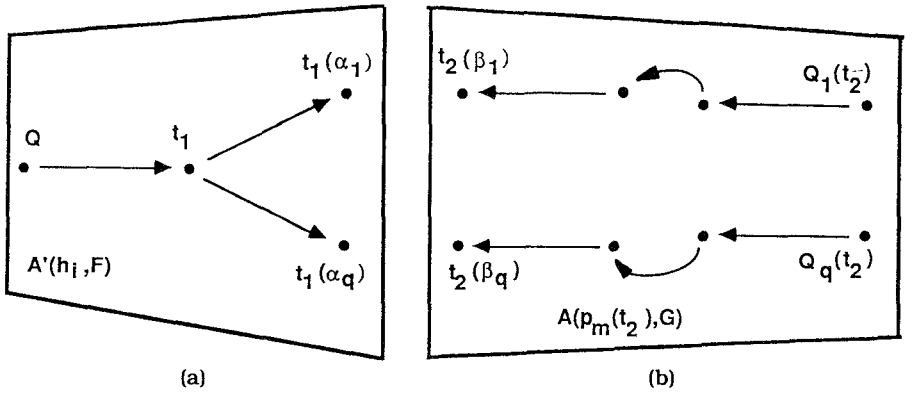

Fig. 7. (a) Graph $A'(h_\iota, F)$; (b) graph $A(p_m(t_2), G)$.

the edges of the form $(z(\gamma), P)$ if $z(\gamma)$ occurs in the query P. (It is important to observe that t_2 is replaced by t_1 in these graphs and that Q has disappeared.)

We have to show the reachability of the components of the v variables, the w variables, and the elements of F in $A(h'_{\iota m}, F)$, and the acyclicity of $AU(h'_{\iota m}, F)$. Let $z(\gamma)$ be a component of a v variable or of an element of F. We consider two cases:

(a) $z(\gamma)$ is not reachable from $t_1(\alpha_v)$. Then the reachability follows from the reachability of $z(\gamma)$ in $A'(h_\iota, F)$.

(b) $z(\gamma)$ is reachable from $t_1(\alpha_v)$. Then the reachability follows from the reachability of $t_2(\beta_v)$ in $A(p_m(t_2), G)$.

Let $z(\gamma)$ be a component of a w variable. The reachability of $z(\gamma)$ follows from the reachability of $z(\gamma)$ in $A(p_m(t_2), G)$.

Next we have to show that $AU(h'_{im}, F)$ is acyclic. To show this, we first identify two parts of the graph $AU(h'_{\iota m}, F)$, called L and R, respectively. L is the subgraph of $AU(h'_{\iota m}, F)$ corresponding to $A'U(h_\iota, F)$, and R is the subgraph of $AU(h'_{im}, F)$ corresponding to $AU(p_m(t_2), G)$ (see

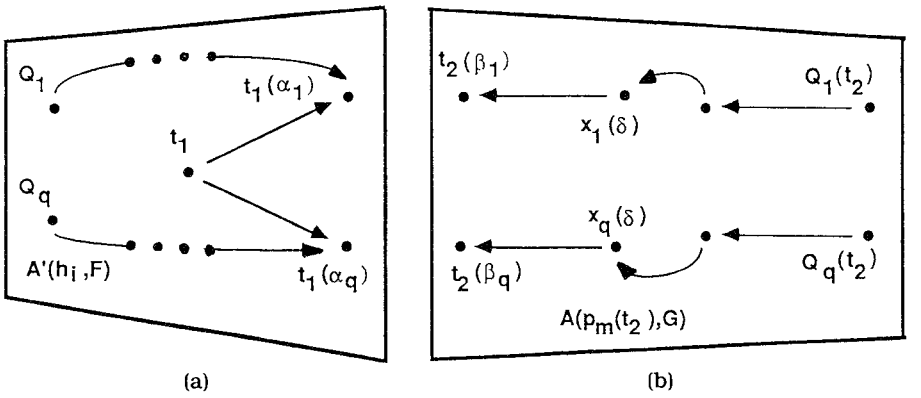

Fig. 8. Graph $AU(h'_{\iota m}, F)$.

Figure 8). It should be clear that only edges of the form $(z(\gamma), P)$, where $z(\gamma)$ occurs in the query P, might create cycles. We consider two cases:

(a) $z(\gamma)$ is not a component of t_1. We consider two cases that may give rise to cyclicity:
  (i) $z(\gamma)$ occurs in L and appears in P in R. Suppose that there is a path in $AU(h'_{\iota m}, F)$, giving rise to a cycle. Notice that such a path necessarily has to go through a component of t_1. Now, since $z(\gamma)$ occurs in P, it also occurred in Q before the query membership elimination rule was applied. It is easily seen that this would have given rise to a cycle in $A'U(h_\iota, F)$, a contradiction.
  (ii) $z(\gamma)$ occurs in R and appears in P in L. Notice that the only paths from L to R originate in t_1 or in its components. Because of the special characteristics of $A'(h_\iota, F)$, there is no path from P to t_1 or to any components of t_1; hence, the edge $(z(\gamma), P)$ cannot create a cycle in $AU(h'_{\iota m}, F)$.

(b) $z(\gamma)$ is a component of t_1, say, $t_1(\alpha_\upsilon)$. There are two cases:
  (i) Suppose that P is in L. Then the edge $(t_1(\alpha_\upsilon), P)$ was added to $A(h'_{\iota m}, F)$. This edge cannot give rise to a cycle, since in $A'(h_\iota, F)$ there is not a path from P to $t_1(\alpha_\upsilon)$ because of the acyclicity of $A'U(h_\iota, F)$.
  (ii) Suppose that P is in R. Then the edge $(t_1(\alpha_\upsilon), P)$ was added to $A(h'_{\iota m}, F)$. There are two cases to consider:
    (ii.1) $t_1(\alpha_\upsilon)$ occurred in P before the query membership elimination rule was applied. In this case, $t_1(\alpha_\upsilon)$ occurred in Q, giving rise to a cycle in $A'U(h_\iota, F)$, a contradiction.
    (ii.2) $t_2(\beta_\upsilon)$ occurred in P before the query membership elimination rule was applied. In this case, the acyclicity of

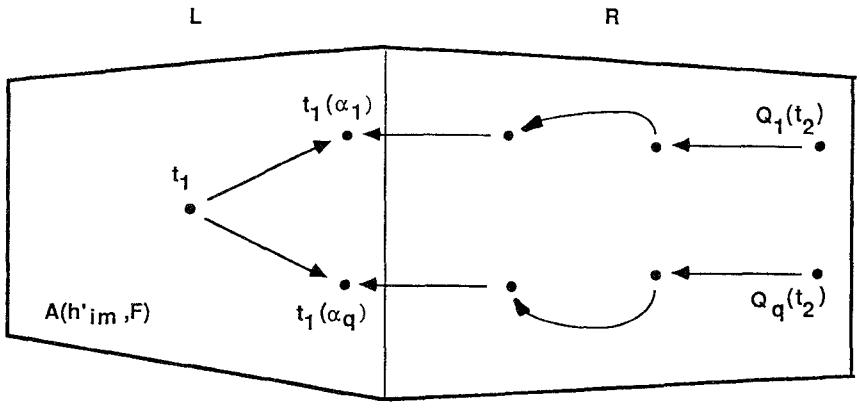

Fig. 9. Graphs $A'(h_i, F)$ and $A(p_m(t_2), G)$.

$(p_m(t_2), G)$ guarantees that there can be no cycle in $AU(h'_{im}, F)$.

We have thus shown that, when (Q, t_1) is an edge of $A'(h_i, F)$, (h'_{im}, F) is acyclic. We now turn to the other case.

(2) (Q, t_1) does not appear in $A'(h_i, F)$. Hence, in $A'(h_i, F)$ no edges leave Q, so no reachability of variable components results as a consequence of the existence of Q; in particular, the reachability of the components of t_1 results from other queries or relations (see Figure 9).

For $A(h'_{im}, F)$ we propose using the graph shown in Figure 10. Clearly, $AU(h'_{im}, F)$ is $A(h'_{im}, F)$ plus the edges $(z(\gamma), P)$ if $z(\gamma)$ appears in the query P. Again, we introduce the notation L and R.

It should be clear that the reachability of the components of t_1 and the components of the v variables follows from their reachability in $A'(h_i, F)$. The reachability of the components of the w variables follows from their reachability in $A(p_m(t_2), G)$. The acyclicity of $AU(h'_{im}, F)$ follows because there are no paths from L to R and vice versa (see Figure 10).

Hence, (h'_{im}, F) is acyclic in this case also. Furthermore, since $(d \vee e_j(t_1), \varnothing)$ is constructive, it follows from Lemma 3(C) that (h'_{im}, F) is constructive. Hence, (h'_i, F) and (g', F) are constructive.

(T2) *Query propagation.* We have to prove that (h'_i, F) is constructive. First note that $t_1(\alpha)$ (remember that $t_1(\alpha) = Q$ or $Q = t_1(\alpha)$ in (T2)) can occur within a subquery P of h_i. In P, $t_1(\alpha)$ is certainly a free variable component. This implies that at the query P, when we consider its associated constructibility graphs, we have that no edge leaves $t_1(\alpha)$ (see conditions (1) and (2) in the edge part definition of constructibility graphs) and, therefore, that the substitution of $t_1(\alpha)$ by Q cannot lead to problems of cyclicity in the subquery P. The reachability of the components of the v variables and the elements of F in $A(h'_i, F)$ follows from the reachability of these

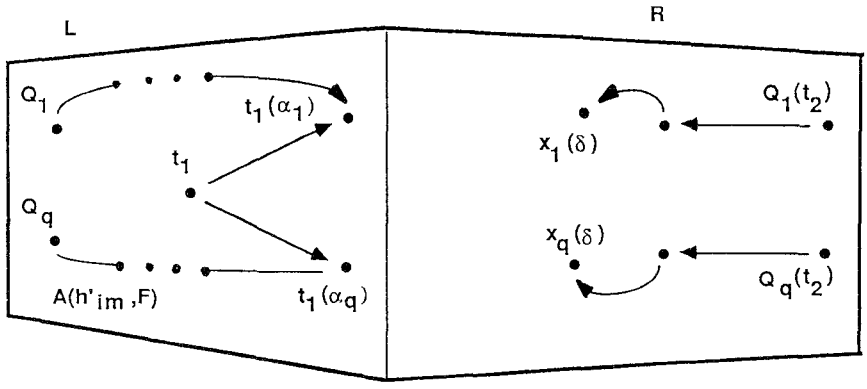

Fig. 10. Graph $A(h'_{im}, F)$.

components and elements in $A(h_i, F)$. The only interesting observation to make is that, when $x(\beta)$ is reachable from a query P and on the path from P to $x(\beta)$ the node $t_1(\alpha)$ occurs, then in $A(h'_i, F)$ the reachability of $x(\beta)$ will result from the fact that there is a path from Q to $x(\beta)$ in $A(h'_i, F)$. We now turn to the acyclicity of (h'_i, F). We first remark that the query Q contains no occurrence of $t_1(\alpha)$; otherwise, there would be a cycle in $AU(h_i, F)$. We consider two cases in which the substitution of $t_1(\alpha)$ may potentially give rise to cyclicity:

(a) Let $x(\beta)$ be reachable from a query P in $AU(h_i, F)$, such that $t_1(\alpha)$ occurs on the path from P to $x(\beta)$. After substituting $t_1(\alpha)$ by Q, there is a path from Q to $x(\beta)$. Suppose that $x(\beta)$ occurs in Q. But then there would be a cycle in $AU(h_i, F)$, namely, the path consisting of the edge going from Q to $t_1(\alpha)$, the path going from $t_1(\alpha)$ to $x(\beta)$, and the edge going from $x(\beta)$ to Q, a contradiction.

(b) Let $x(\beta)$ be reachable from a query P in $AU(h_i, F)$, and suppose that P contains an occurrence of $t_1(\alpha)$. Hence, after the substitution, P is transformed into a query P' in which $t_1(\alpha)$ is replaced by Q. Suppose that Q contains an occurrence of $x(\beta)$. This would introduce a cycle in $AU(h'_i, F)$. This cannot happen, however, because there would be a cycle in $AU(h_i, F)$, namely, the path consisting of the edge going from Q to $t_1(\alpha)$, the edge going from $t_1(\alpha)$ to P, the path going from P to $x(\beta)$, and the edge going from $x(\beta)$ to Q.

Hence, (h'_i, F) is acyclic. Also, it can easily be seen that the substitution of Q for $t_1(\alpha)$ in d does not affect the constructiveness of $(d', \varnothing)$. Finally, constructive subqueries are not affected by the substitution of Q for $t_1(\alpha)$. Hence, (h'_i, F) is constructive, and therefore, so is (g', F).

(T3) *Structured component elimination.* This is obvious since we only eliminate terms, attributes, and, eventually, quantifiers.

(T4) *Query equality elimination.* We have to show that (h'_i, F) is constructive. We first show that (h'_i, F) is acyclic. This follows because we can choose for $A(h'_i, F)$ the graph $A(h_i, F)$ without the nodes corresponding to the queries $\{t_1[\gamma_1] \mid f_1(t_1)\}$ and $\{t_2[\gamma_2] \mid f_2(t_2)\}$. Clearly, constructive subqueries are preserved by the query equality elimination rule. We only need to show that

$$\left(\left(d \stackrel{\vee}{-} \exists t_1[\gamma_1]\left(f_1(t_1) \stackrel{\wedge}{-} \neg f_2(t_1)\right) \stackrel{\vee}{-} \exists t_2[\gamma_2]\left(f_2(t_2) \stackrel{\wedge}{-} \neg f_1(t_2)\right)\right), \varnothing\right)$$

is constructive. This follows from Lemmas 3(C), 3(D), and 3(A). □

LEMMA 8. *A constructive query in the flat calculus is safe.*

PROOF. According to Ullman [44], a query $\{t[\lambda] \mid f\}$ of the flat calculus is *safe* iff

(1) whenever t satisfies f each component of t occurs somewhere in a relation of the database; and
(2) whenever $\exists v[\gamma]f_1$ occurs in the query, if f_1 is satisfied by v for any values of the other free variables in f_1, then each component of v occurs somewhere in a relation of the database.

Let $\{t[\lambda] \mid f\}$ be a constructive query in the flat calculus. This implies that (f, F) is constructive, where $F = \{t(\alpha) \mid \alpha \in \lambda\}$. In particular,

(1) all of the elements of F, hence, all the components of t, are reachable in (f, F), which implies item (1) above, since t only has atomic attribute components; and
(2) in a subformula $\exists v[\gamma]f_1$ all of the components of v are reachable in (f_1, F_1), $F_1 = \{v(\alpha) \mid \alpha \in \gamma\}$, which implies item (2) above, since all of the components of the v variables are atomic attribute components. □

4.4 Algorithm *Translate*

We now present an algorithm to construct a query in the flat algebra that is logically equivalent to a formula of the nested algebra with flat operands and a flat result.

Algorithm translate

Input. An ff-expression of the nested algebra.

Output. An expression of the flat algebra.

Method

(1) Transform the given ff-expression into a constructive calculus query, using Lemma 6.
(2) Repeat steps (2.1) and (2.2) until no longer possible.
  (2.1) Repeat query membership elimination until no longer possible.
  (2.2) Repeat query propagation immediately followed by structured component elimination until no longer possible.

(3) Repeat query equality elimination until no longer possible.
(4) Apply Codd's theorem for translating a safe calculus query into an equivalent flat algebra expression.

We illustrate step (2) of the algorithm with an example. Consider the following constructive query:

$$\{t[A] \mid \exists t_1[B(C)] \exists t_2[D(E(F))] (t \in t_1(B(C)) \wedge t_1 \in t_2(D(E(F))) \wedge t_2(D(E(F))) = \{t_3[G(H)] \mid t_3(G(H)) = \{t_4[I] \mid t_4 \in R\}\})\}.$$

Step (2.1) is not applicable, so we apply step (2.2). This yields the query

$$\{t[A] \mid \exists t_1[B(C)] (t \in t_1(B(C)) \wedge t_1 \in \{t_3[G(H)] \mid t_3(G(H)) = \{t_4[I] \mid t_4 \in R\}\})\}.$$

Now we can apply step (2.1), and we get the query

$$\{t[A] \mid \exists t_1[B(C)] (t \in t_1(B(C)) \wedge t_1(B(C)) = \{t_4[I] \mid t_4 \in R\})\}.$$

Now we apply step (2.2), and we get

$$\{t[A] \mid t \in \{t_4[I] \mid t_4 \in R\}\}.$$

We finally apply step (2.1) to get the query

$$\{t[A] \mid t \in R\}.$$

LEMMA 9. *Algorithm Translate, taking as input an ff-expression of the nested algebra, stops and produces an expression of the flat algebra that is equivalent to the given ff-expression.*

PROOF. First we prove that the algorithm is correct, if it ends. Then we prove that it ends. Lemma 7 indicates how steps (2) and (3) have to be performed. The result is a constructive query. Since every structured component of every variable that occurs in the query is reachable, no structured component can be left after step (2). Nor are any formulas of the form $t(\alpha) \in Q$, $t(\alpha) = Q$ or $Q = t(\alpha)$ left. Since all of the formulas of the form $Q_1 = Q_2$ are eliminated by step (3), and since no new query is created by this step, all of the queries are eliminated after step (3). Hence, the result after step (3) is a constructive query in the flat calculus. This query is safe by Lemma 8 and can be transformed into an equivalent expression in the flat algebra. In order to prove that the algorithm ends, we call n_Q the number of queries and n_S the number of occurrences of structured components in the query. Suppose that we apply step (2) each time on subqueries without occurrences of structured components. Each time step (2.1) or step (2.2) is executed, the value $n_Q + n_S$ becomes smaller, since every (h_t, F) is acyclic. Each time step (3) is applied, n_Q becomes smaller. □

We are now able to formulate our main result:

THEOREM. *For every ff-expression in the nested algebra, there is an equivalent expression in the flat algebra.*

COROLLARY. *The transitive closure is not expressible in the nested algebra.*

5. CONCLUSION

Our main result states that the flat algebra is as expressive as the nested algebra operating on a flat database and with a flat relation as a result. A simple consequence is, therefore, that recursive queries, such as the transitive closure, are not expressible in the nested algebra. This result clearly separates the nested algebra and the powerset algebra since it is well known that recursive queries are indeed expressible in the powerset algebra. This again suggests that the nested algebra is a conservative extension of the flat algebra, whereas the powerset algebra is not.

Although we did not consider selection operations involving constants (either atomic or structured) or selection operations involving subset conditions, it is straightforward to generalize our results to include these options. This generalization notably expands the range of practical queries that are affected by our results.

We believe that our results can contribute to (flat) relational query optimization. This will clearly depend on how efficiently flat-to-flat nested relational queries can be processed in relational database implementations. Related to this issue, it would be interesting to obtain a complete algebraic proof of our main result, that is, a proof without having to introduce an intermediate calculus.

Finally, algebraic query languages have also been introduced for object-oriented databases [42]. It would be interesting to investigate if and how our translation strategies and results generalize to such databases.

ACKNOWLEDGMENTS

The authors are grateful to the referees for their many excellent comments and suggestions.

REFERENCES

1. ABITEBOUL, S., AND BEERI, S. On the power of languages for the manipulation of complex objects. INRIA Tech. Rep. 846. 1988.
2. ABITEBOUL, S., AND BIDOIT, N. Non first normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci. 33*, 3 (Dec. 1986), 361-393.
3. AHO, A. V., AND ULLMAN, J. D. Universality of data retrieval languages. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages* (Jan. 1979, San Antonio, Tex.). ACM, New York, 1979, pp. 110-117.
4. ARISAWA, H., MORIYA, K., AND MIURA, T. Operations and the properties on non-first-normal-form relational databases. In *Proceedings of the 9th VLDB* (Florence). 1983, pp. 197-205.
5. BANCILHON, F. A logic-programming/object-oriented cocktail. *ACM SIGMOD Rec. 15*, 3 (Sept. 1986), 11-20.

6. BANCILHON, F., AND KHOSHAFIAN, S. A calculus for complex objects. In *Proceedings of the 5th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Mar 1986, Cambridge, Mass.). ACM, New York, 1986, pp. 53-59.
7 BANCILHON, F., RICHARD, P., AND SCHOLL, M On line processing of compacted relations In *Proceedings of the 8th VLDB* (Sept. 1982, Mexico City). VLDB Endowment, 1982, pp. 263-269.
8. BEERI, C., AND KORNATZKY, Y. The many faces of query monotonicity. In *Proceedings of Advances in Database Technology—EDBT '90* (Mar. 1990, Venice) Lecture Notes in Computer Science, vol. 416. Springer-Verlag, New York, 1990, pp. 120-135
9. BEERI, C., NAQVI, S., RAMAKRISHNAN, R., SHMUELI, O., AND TSUR, S. Sets and negation in a logic database language (LDL1). In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, Calif.) ACM, New York, 1987, pp. 21-37.
10. BIDOIT, N. The Verso algebra and how to answer queries without joins. *J. Comput. Syst. Sci. 35*, 3 (Dec. 1987), 321-364.
11. CODD, E. F. Relational completeness of database sublanguages. In *Database Systems*, R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J , 1972, pp. 65-98.
12 COLBY, L. A recursive algebra and query optimization for nested relations. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (May 1-June 2, 1989, Portland, Ore.). ACM, New York, 1989, pp 273-283.
13 DADAM, P. Advanced information management (AIM): Research in extended nested relations. *IEEE Data Eng 11*, 3 (Dec. 1988), 4-14
14. DADAM, P , KUESPERT, K., ANDERSEN, F., BLANKEN, H., ERBE, R , GUENAUER, J , LUM, V., PISTOR, P., AND WALCH, G A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (May 1986, Austin, Tex.). ACM, New York, 1986, pp. 356-366.
15. DEPPISCH, U., PAUL, H. B., AND SCHEK, H. J A storage system for complex objects. In *Proceedings of the International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif.). 1986, pp. 183-195
16. DESHPANDE, A., AND VAN GUCHT, D. A storage structure for unnormalized relations. In *Proceedings of the GI Conference on Database Systems for Office Automation, Engineering and Scientific Applications* (Apr. 1987, Darmstadt, Germany) *Inf Fachberichte 136* (1987), 481-486.
17. DESHPANDE, A., AND VAN GUCHT, D. An implementation for nested relational databases In *Proceedings of the 14th VLDB* (Los Angeles, Calif.). 1988, pp 76-88.
18. DESHPANDE, V., AND LARSON, P. A. An algebra for nested relations. Res. Rep CS-87-65, Dept. of Computer Science, Univ. of Waterloo, Ontario, 1987.
19. GYSSENS, M., AND VAN GUCHT, D. The powerset algebra as a result of adding programming constructs to the nested relational algebra In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (June 1988, Chicago, Ill.). ACM, New York, 1988, pp. 225-232
20. GYSSENS, M., AND VAN GUCHT, D. The powerset algebra as a natural tool to handle nested database relations. *J. Comput. Syst. Sci.* To be published.
21. GYSSENS, M., PAREDAENS, J., AND VAN GUCHT, D. A uniform approach towards handling atomic and structured information in the nested relational database. *J. ACM 36*, 4 (Oct. 1989), 790-825.
22. HOUBEN, G., AND PAREDAENS, J. The R^2-algebra: An extension of an algebra for nested relations. Tech. Rep., Computer Science Dept., Technical Univ., Eindhoven, The Netherlands, 1987.
23. HULL, R., AND SU, J. On the expressive power of database queries with intermediate types. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, Tex.). ACM, New York, 1988, pp. 39-51.
24. JAESCHKE, G., AND SCHEK, H. J. Remarks on the algebra on non first normal form

relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Los Angeles, Calif.). ACM, New York, 1982, pp. 124-138.
25. KUPER, G. M. Logic programming with sets. *J. Comput. Syst. Sci. 41*, 1 (Aug. 1990), 44-64.
26. KUPER, G. M., AND VARDI, M. On the complexity of queries in the logical data model. In *Proceedings of the 2nd International Conference on Database Theory* (Bruges, Belgium). Lecture Notes in Computer Science, vol. 326. Springer-Verlag, New York, 1988, pp. 267-280.
27. LARSON, P. A. The data model and query language of Laurel. *IEEE Data Eng. 11*, 3 (Sept. 1988), 23-30.
28. LINNEMANN, V. Non first normal form relations and recursive queries: An SQL-based approach. In *Proceedings of the 3rd IEEE International Conference on Data Engineering* (Los Angeles, Calif.). IEEE, New York, 1987, pp. 591-598.
29. MAKINOUCHI, A. A consideration of normal form of not-necessarily-normalized relations in the relational data model. In *Proceedings of the 3rd VLDB* (Oct. 1977, Tokyo). VLDB Endowment, 1977, pp. 447-453.
30. NAQVI, S., AND TSUR, S. *A Logical Language for Data and Knowledge Databases*. Freeman, San Francisco, Calif., 1989.
31. OZSOYOGLU, G., OZSOYOGLU, Z. M., AND MATOS, V. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987), 566-593.
32. PAREDAENS, J., DE BRA, P., GYSSENS, M., AND VAN GUCHT, D. *The Structure of the Relational Model. EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, 1989.
33. PAUL, H. B., SCHEK, H. J., SCHOLL, M. H., WEIKUM, G., AND DEPPISCH, U. Architecture and implementation of the Darmstadt database kernel system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (May 1987, San Francisco, Calif.) ACM, New York, 1987, pp. 196-207.
34. PISTOR, P., AND ANDERSEN, F. Designing a generalized NF^2 model with an SQL-type language interface. In *Proceedings of the 12th VLDB* (Aug. 1986, Kyoto). VLDB Endowment, 1986, pp. 278-288.
35. PISTOR, P., AND TRAUNMUELLER, R. A database language for sets, lists and tables. *Inf. Syst. 11*, 4 (1986), 323-336.
36. ROTH, M. A., KORTH, H. F., AND BATORY, D. S. SQL/NF: A query language for $\neg$ 1NF relational databases. *Inf. Syst. 12*, 1 (1987), 99-114.
37. ROTH, M. A., KORTH, H. F., AND SILBERSCHATZ, A. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst. 13*, 4 (Dec. 1988), 389-417.
38. SCHEK, H. J., AND SCHOLL, M. H. The relational model with relation-valued attributes. *Inf. Syst. 11*, 2 (1986), 137-147.
39. SCHEK, H. J., PAUL, H. B., SCHOLL, M. H., AND WEIKUM, G. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Trans. Knowledge Data Eng. 2*, 1 (June 1990), 25-43.
40. SCHOLL, M. H. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proceedings of the 1st International Conference on Database Theory* (Aug. 1986, Rome). Lecture Notes in Computer Science, vol. 243. Springer-Verlag, New York, pp. 380-396.
41. SCHOLL, M. H., PAUL, H. B., AND SCHEK, H. J. Supporting flat relations by a nested relational kernel. In *Proceedings of the 13th VLDB* (Sept. 1987, Brighton, Great Britain). VLDB Endowment, 1987, pp. 137-146.
42. SHAW, G., AND ZDONIK, S. A query algebra for object-oriented databases. Tech. Rep. CS-89-19, Dept. of Computer Science, Brown Univ., Providence, R.I., 1989.
43. THOMAS, S. J., AND FISCHER, P. C. Nested relational structures. In *Advances in Computing Research III, The Theory of Databases*, P. C. Kanellakis, Ed. JAI Press, Greenwich, Conn., 1986, pp. 269-307.
44. ULLMAN, J. D. *Principles of Database Systems*. 2nd ed. Computer Science Press, Rockville, Md., 1982.

Received July 1989; revised February 1991; accepted March 1991