On the power of languages for the manipulation of complex objects

Serge Abiteboul, C. Beeri

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N°846

ON THE POWER OF LANGUAGES FOR THE MANIPULATION OF COMPLEX OBJECTS

Serge ABITEBOUL
Catriel BEERI

MAI 1988

RR-0846

ON THE POWER OF LANGUAGES FOR THE MANIPULATION OF COMPLEX OBJECTS[1]

Serge Abiteboul
I.N.R.I.A
78153 Le Chesnay, Cedex
France

Catriel Beeri
Department of Computer Science
The Hebrew University
Israel

ABSTRACT

Various models and languages for describing and manipulating hierarchically structured data have been recently proposed. Although algebraic, calculus based and logic programming oriented languages have all been considered, there is very little work on the expressive power of those languages. This paper presents a general model for complex objects, and languages based on the three paradigms for the model. The algebraic language generalizes those presented in the literature; it is an adaptation of the functional style of programming advocated by Backus. The notion of domain independence is defined, and syntactic restrictions (referred to as safety conditions) on calculus queries are formulated, that guarantee domain independence. The main results are: The domain independent calculus, the safe calculus, the algebra, and the logic programming oriented language are equivalent to each other. In particular, recursive queries, such as the transitive closure, can be expressed in each of the languages. For this result, the algebra needs the powerset operation. A more restricted version of safety is presented, such that the restricted safe calculus is equivalent to the algebra without the powerset. The results are extended to the case where arbitrary interpreted functions and predicates are added to the languages.

[1] This research was supported by a grant 2545 from the National Council for Research and Development of Israel, and the Association Franco- Israelienne pour la Recherche Scientifique et Technologique.

DE LA PUISSANCE DES LANGAGES DE MANIPULATION D'OBJETS COMPLEXES

RESUME

Plusieurs modèles pour données structurées hiérarchiquement ont été proposés récemment. Des algèbres, des calculus, des langages orientés programmation logique ont été considérés. Les puissance de ces langages ont été que peu étudiées. Ce rapport présente un modèle d'objets complexes, et des langages basés sur ces trois paradigmes. Les langages introduits généralisent la plupart des langages préalablement proposés; leurs puissances sont comparées. Ces résultats sont étendus à des langages autorisant l'addition de prédicats et fonctions interprétées.

1. INTRODUCTION

Various database models have been recently proposed to deal with data of hierarchical nature (e.g., non-first-normal form relations or complex objects). Languages in these models are algebraic, calculus based, or logic programming oriented. Surprisingly, few works have tried to compare the power of these languages. The purpose of this paper is to present such a comparison, in the context of a general model for complex objects.

1.1 Overview

In general, current database systems deal exclusively with atomic values, and objects are simple aggregates of such values. For instance, the first normal form condition forces the components of tuples in relational databases to be atomic [C]. Over the last few years, it has been widely recognized that this restriction imposes unacceptable constraints on the use of database technology in a variety of application domains such as engineering, computer aided design, or office systems [Ko, Mac, Mak]. Many models and languages that allow non atomic components have been introduced, and are now the subject of theoretical investigations and implementation efforts. The structures that are allowed in these models are generally referred to as *relations with set-valued attributes* [JS, OO, OOM], *non-first-normal-form relations* [AB, FT, ScSc, KRS, H] or *complex objects* [AH2, BK, H].

In this paper, we present and compare query languages for a general model of complex objects. In our model, complex objects are obtained from atomic ones using *set* and *tuple* constructors. No restrictions are placed on the order of application of the constructors, or on the depth of the constructed objects. There are many ways to define an algebra for complex objects. Our approach adopts the combinator paradigm of [Ba], and can be viewed as a high level functional language for complex objects. The calculus is based on the calculus of Jacobs [J], and is essentially similar to the calculus used in [KRS, H]. The logic programming oriented language is also a straightforward extension of such languages for the relational model.

It is well known that, in the relational model, the calculus is more expressive than the algebra, the reason being that one can express in the calculus queries whose answer depends not only on the database, but also on the domains. However, the calculus, restricted to *domain independent* queries (called safe in [U]) is equivalent to the algebra. We present a natural notion of domain independence for complex object languages. Our first result states the equivalence between the algebra and the domain independent calculus.

Domain independence is a semantic, undecidable, property. We therefore consider

syntactic restrictions that guarantee domain independence. Syntactically restricted formulas are called here *safe*. The idea in syntactic restrictions is to attach a range formula to each variable. Our next result is that the algebra and the safe calculus are equivalent. This implies in particular, that this syntactic restriction "captures" the semantic notion of domain independence.

One of the operations of the algebra is the "powerset," used to generate all subsets of a given set. It is not found in most previously proposed algebras. This operation is necessary in order to obtain the power of the calculus. However, the powerset operation is quite costly. It is, therefore, interesting to characterize the power of the algebra without the powerset operation. We present a restricted (syntactic) notion of safety, and we prove that the calculus, thus restricted, corresponds to the algebra without powerset, thereby characterizing the power of most algebras found in the literature.

Functions and predicates have, in the past, been included both in the algebra and in the calculus. The functions usually used are standard arithmetic, and aggregate functions. The generality of our model, which allows atomic, tuple and set valued domains, permits us to consider arbitrary functions among those domains in a natural setting. We show that the equivalence of the algebra (with/without the powerset operation) and the (syntactically safe/restricted syntactically safe) calculus holds for essentially arbitrary sets of predicates and functions, provided of course, that the same predicates and functions are included in both languages.

Recently, languages using the logic programming paradigm have been proposed for complex objects [BNRST, Ku, AG]. We compare here the expressive power of that approach to the algebraic or calculus approach. It is a well-known result that simple recursive queries (e.g., transitive closure) cannot be expressed using relational calculus [AU]. It turns out that the same queries can be expressed using the domain independent complex object calculus. We present an extremely simple language for complex objects based on recursive rules. We show that each query expressible in this language corresponds to a domain independent calculus (respectively, algebraic) query, and vice versa.

1.2 Comparison with previous works

Our model generalizes the non-first-normal-form relational models [FT, JS, KRS, Mak, ScSc]. The objects we deal with can also be seen as the objects resulting in semantic database modeling [AH1, HY, HM] from the use of aggregation (tuple constructors) and classification (set constructors). Sets of homogeneous objects only are considered. In that respect, the data structure that we study is strictly weaker than those considered in [AH1,

AH2, HY]. We believe that our results can easily be extended if heterogeneous sets are allowed. Our types can be described by trees. Unlike in [KV], cycles are not allowed in type definitions.

Equivalence results of algebraic and calculus based languages have been given in [KV, OO,OOM, KRS]. The equivalence of relational algebra and calculus with aggregates was considered in [Kl], and extended in [OO] to relations with set-valued attributes. A comparison is proposed in [KRS] for a non-first-normal-form relational model. However, they have chosen to restrict their work to the case where the nest and unnest operators commute, and their equivalence proof uses unnest to reduce the problem to the case of flat relations, and nest to restore the original relations. They do not introduce any algebraic operators that can access set-valued components of tuples. Since nest and unnest in general do not commute, there is a need for such operations in the algebra, and a need for a direct proof, without reduction to the classical case. An equivalence result for a different, and in a sense more general, model, in which objects have identities and cycles are allowed, is given in [KV]. Our characterizations of syntactic safety, and the comparison to recursive languages are new. The equivalence results obtained in the presence of interpreted functions and predicates are also new.

A model slightly less general than ours has been presented in [DM]. They allow nested structures in which internal nodes are sets, and the leaves are relations. They extend the results of [CH] concerning completeness of query languages to this model. Their techniques can be extended to our model; however, completeness of languages in the sense of [CH] is not treated here. We note that the proof of their main theorem relies on coding nested structures by flat relations, using generalizations of nest and unnest. However, these constructs require sequencing operators and cannot be expressed in the algebra presented here (and in any of the other algebras presented in the literature). Hence the remark above about the need for a direct proof is still valid.

The languages that we present are strictly more powerful than the relational calculus. Mappings from relations to relations can be defined using those languages, which cannot be expressed in the relational calculus. This is a consequence of the ability to manipulate richer structures in temporary results. Indeed, [HS] exhibit a hierarchy of languages, based on restrictions on the types of intermediate results. Our languages are not restricted, hence are more powerful than those considered there.

1.3 Organization

The paper is organized as follows. In Section 2, we present the data model, and in

Section 3 we define databases, functions and queries. The calculus is introduced in Section 4, and the algebra in Section 5. Additional examples and explanations regarding the two languages are presented in Appendices A and B. Section 6 deals with the equivalence between the algebra and the domain independent calculus. In Section 7, we introduce syntactic safety restrictions on the calculus and compare the power of the resulting languages to the algebra with and without the powerset. In Section 8, built-in functions and predicates are introduced both in the algebra, and the calculus. Section 9 deals with recursive queries. A summary is presented in Section 10.

2. COMPLEX TYPES AND OBJECTS

In the relational model, instances are sets of tuples. That is, the basic constructors are a set constructor and a tuple constructor. In the construction of a type (i.e., a relation schema), each is used precisely once: first the tuple constructor, then the set constructor. We use the same two constructors, but we allow repeated use of both constructors. Furthermore, in contrast to most other proposals for complex objects that require the two constructors to alternate in any given type, we make no such restriction.

We assume the existence of a set of *domain names* $D_1, D_2, \cdots$ and of an infinite set of *names*, also called *attributes*. Types are structure definitions, constructed from domain names, attributes and constructors. Assume that the domain names are associated with *domains* $D_1, D_2, \cdots$. The elements of the domains are called *atomic* values. Objects are constructed from the domain elements, attributes and the constructors. A type is associated with each object, in the obvious way; each object is an *instance* of a type.

Types and *objects* are defined as follows:

(1) If D is a domain name, then D is a type. For each a in D, a is an object of this type.

(2) If $T_1, \ldots, T_n$ are types, the sets of attributes used in them are disjoint, and $A_1, \ldots, A_n$ are attributes not used in any of them, then $[A_1{:}T_1, \ldots, A_n{:}T_n]$ is a *tuple* type. If $O_1, \ldots, O_n$ are objects of types $T_1, \ldots, T_n$, resp., then $[A_1{:}O_1, \ldots, A_n{:}O_n]$ is an object of the type. For generality, we also include $T_{[\,]}$, where [ ] denotes the empty tuple, as a type. The only object of this type is [ ], the empty tuple.

(3) If T is a type, and A is an attribute not used in it, then $\{A{:}T\}$ is a *set* type. If O is a set of objects of type T, then O is an object of that type.

(4) If T is a type, and A is a name not used in it, then $A:T$ is a *named type*, with name A. Any instance of T is an instance of $A:T$.

(Thus, tuple and set types are created by applying the corresponding constructors to named types.)

A comment on the use of names is in order. The components of a tuple type, and also of a tuple object, must be named. The reason is that the name of a tuple component serves to denote that component, and distinguishes it from other components of the type, or the object. For that reason, the name must be unique. It would seem that a name for the members of a set is not useful, since it cannot serve to distinguish the members of a set object from each other. Consider, however, a set of sets. We would like to be able to apply a selection operation to the set. To express the operation, we need to use the name of the member of the set in a selection predicate. Thus, a name for the members of a set type is useful, although it does not serve as a unique identifier.

Types and objects may be described by trees. In a type tree, leaves are labeled by their atomic types, and in an object tree, leaves are labeled by their values. Internal nodes are labeled in both cases by the constructors. In addition, the nodes are labeled by the names. We will refer to types as *atomic*, *set* or *tuple* types, respectively, according to their main constructor, i.e., according to the constructor at the root of the tree.

To illustrate the previous definitions, consider the well known supplier example. The corresponding relational schema may be represented by the type shown in Figure 2. (We use inline description of the tree.)

```
database:
[       supplier:
        {       [ name:[last:string, first:string], sno:int, cities:{city-name:string} ] }
        part:
        {       [ pno:int, name:string, price:int ]     }
        supply:
        {       [ snum:int, pnum:int, quantity:int ]    }       ]
```

Figure 2: A Supplier-Part database

Remarks:

(1) The nature of the elements of the domains is irrelevant in this paper. We frequently omit the domains in type and object definitions. In examples, we use the domain of integers.

(2) The requirement that all names in a type be distinct can be relaxed. Any naming scheme that attaches a unique identifier to each node of a type tree is acceptable. (e.g., using a concatenation of the names on the path to the root). Thus, in the example database above, it is common practice to use the same attribute 'sno' in both the supplier and supply relations, and similarly for 'pno'.

Let T be a type, different from $T_{[]}$, using domain names $D_1, \ldots, D_k$, and let domains $D_1, \ldots, D_k$ be given. We can construct the set of all values of type T that contain only atomic values from D_i in the leaves of type D_i, denoted $DOM(T, D_1, \ldots, D_k)$, as follows:

(1) Replace in the definition tree of T each leaf labeled D_i by D_i.

(2) Replace the labels of internal nodes as follows: If a node is a tuple constructor, replace its label by a cross-product operator. If a node is a set constructor, replace the label by a powerset operator. [2]

(3) Evaluate the tree.

For $T_{[]}$, the set of values is $\{[]\}$, independent of the domains.

If T is a type $[A_1:T_1, \ldots, A_n:T_n]$, and $O = [A_1:O_1, \ldots, A_n:O_n]$ is an object of type T then $T.A_i$ denotes the type T_i, and $O.A_i$ denotes the object O_i. In our model, since objects have more structure than in the relational model, this notation may be generalized: if $T.A_i$ is a tuple of tuples type, and B appears in the subtuple named A_i, then the expression $T.A_i.B$ is well defined (assuming left associativity). Furthermore, since names are unique in a type, we may simply write $T.B$. For a tuple type T, the names that appear in T, and in its subtuples (to any depth) are called the *constant names* of T, denoted by *c_names(T)*. Intuitively, if T is viewed as a tree, a label of a node r is a constant name iff there is a path in T from the root to r which does not cross a set node. For every object of the type T, every such name denotes a unique subobject. Once a set node is reached, however, then the name of a member of that set is not a unique identifier of an object. We allow to use any constant name A of T in a

[2] These operators are introduced formally later. We assume they are known to the reader.

qualification of the form $T.A$, to denote the A component of T. Similarly, for O of type T, $O.A$ denotes the A-component of O. (In particular, for a named type $A:T$, $O.A$ refers to O itself.)

3. DATABASES, QUERIES AND LANGUAGES

A *database scheme* is a pair

$$DB = <[D_1, \ldots, D_k], B:[R_1:T_1, \ldots, R_n:T_n]>,$$

where $T_1, \ldots, T_n$ are types, involving only the domain names $D_1, \ldots, D_k$. An *instance* of DB is a structure

$$DB = <[D_1, \ldots, D_k], B:[R_1, \ldots, R_n]>,$$

where the D_i's are domains and each R_i is of type T_i. Further, the atomic values in R_i, of type D_i are from D_i. Following the accepted conventions in relational databases, we will refer to B as the *database type*, and to B as the *database object*. Elements in them, at any level of nesting, are also referred to as objects.

A *function* of type $DB \rightarrow T$ is a partial function from instances of DB to instances of T.[3] We assume that the domain names used in T are in the list of domain names in DB. We refer to DB and T as the input and output types of the function. When DB is obvious from the context, we may refer to T as the type of the function. The result of applying a function f to a database instance DB is denoted $f(DB)$. If the association of domains with domain names is fixed (or irrelevant), we also write $f(B)$.

A *query* is a function such that the input and output types, $T_1, \ldots, T_n, T$, are set types. (This is usual in databases. Also, it is particularly suited for queries expressed in a calculus-based language, as will be seen below.) A *query language* is a notation for expressing queries, coupled with a mechanism for assigning meaning to the expressions, i.e. for associating queries with expressions. For a discussion of query languages, see [CH]. In this paper, we are concerned with three languages: generalizations of the relational calculus and algebra, and a rule-based language. All three express only total queries.

Our main interest in this paper is in queries. However, to be able to express interesting

[3] It is customary to define the type of a function by the types of its input and output. In our case, a function may, in general, depend not only on the objects in a database instance, but also on the association of domains to domain names defined by it. Hence, we represent its type as $DB \rightarrow T$, and not as $T_1, \ldots, T_n \rightarrow T$. See the discussion of domain independence below.

queries on complex objects, we need the ability to express general functions as well. An important class of operations on complex objects, that is almost absent from the relational model, is those that perform restructuring - changing the structure of each member of a set. Such restructuring is defined by a function to be applied to each member of the set. This is essentially the idea of the *apply to all* operator of FP [Ba], of the *mapcar* operation of LISP, and of the *replacement axiom* of set theory.

In our model, the elements in a set may be of arbitrary type, and the result after restructuring is also arbitrary. Hence the need for expressing functions with no restrictions on the input and output types. In this paper, queries in which an operation is applied to each element of a set are called *filters*. In the relational model, selection and projection are filters. These two operations illustrate the two general types of filters:

- A *transformational* filter applies a transformation, called the *filter transformation*, to each element of a set.

- A *predicative* filter applies a predicate, called the *filter predicate*, to each element of a set. The elements for which its value is true are in the output, unchanged. The others are ignored.

Both the calculus and the algebra presented in this paper contain mechanisms for expressing general functions and predicates. As will be seen later, in each of the two languages, the mechanisms for expressing functions and queries are slightly different. Even when a function is a query, it has two representations, depending on how it is to be used - as a filter, or as a query.

We have placed no restriction on the association of domain names with domains. Such restrictions may be needed. First, we allow the use of constants in queries, and then the domains are restricted to contain the constants. Second, we may want to fix the interpretation of a domain, e.g., to fix the interpretation of *int* to the integers. The reader is referred to [CH] and [AV] for a discussion of this subject. See also Section 8.

In general, the value of $q(DB)$, where q is a query, may depend on $D_1, \ldots, D_k$, as well as on B. (See the discussion of *safety* in [U], and in [VGT].) We say that a query q is *domain independent* if for any database structure, changing the domains (but keeping them large enough to contain all atomic entries appearing in B or the query) does not change the result of the query [Fa]. It is well known that the expressions of relational algebra define domain independent queries. The same holds for our algebra. Calculus formulas do not necessarily define domain independent queries. Domain independence is a semantic definition, in that it

uses the notion of a structure. Even for the classical relational calculus, domain independence is undecidable [DiP, Va]. It is natural therefore to consider syntactic restrictions. We will consider several syntactic restrictions later. Expressions that are in a restricted class of expressions, that guarantees domain independence, will be called in this paper *safe*.

4. A COMPLEX OBJECT CALCULUS

The calculus is a many-sorted first-order language, based on that presented by Jacobs [J]. The components are: [4]

(1) Domain names, $D_1, D_2, \cdots$.

(2) Attributes.

(3) Types, constructed from the domain names and the attributes (including $T_{[\,]}$).

(4) Constants. (Each constant is assumed to belong to one of the domains, i.e., it is of an atomic type.)

(5) Input parameters, $R_1, R_2, \cdots$. It is assumed that each is associated with a type T_i.

(6) Variables. It is assumed that each variable is associated with a type.

(7) The "dot" selector, ".".

(8) Two particular predicates: $\in_{S,T}$ (membership predicate), and $=_{SS}$ (equality predicate) for all types S, T={S}. [5]

(9) Connectives: $\wedge, \vee, \neg$.

(10) Quantifiers: $\forall, \exists$.

Type declarations of the input parameters and the variables are usually omitted, when they can be inferred from the context. We also omit the types in the equality and membership predicates.

---

[4] To distinguish between equality in the language, in formulas, and equality in the metalanguage, e.g., for expressing syntactic equality of formulas, we use $=$ for the first, and $\equiv$ for the latter, throughout the paper.

[5] We have, for instance, $\{B:5,B:6,B:9\} =_{\{B:D\},\{B:D\}} \{B:5,B:9,B:6\}$.

Let $DB = <[D_1, \ldots, D_k], B:[R_1:T_1, \ldots, R_n:T_n]>$ be a database scheme. We associate with it the sublanguage in which only the domain names $D_1, \ldots, D_k$ and the input parameters $R_1, \ldots, R_n$ are used, and R_i is of type T_i. An *interpretation* is a mapping that assigns domains $D_1, \ldots, D_k$ to the names $D_1, \ldots, D_k$, and objects $R_1, \ldots, R_n$ to $R_1, \ldots, R_n$ (of appropriate types).[6] Just as a scheme uniquely identifies a sublanguage, an interpretation is uniquely defined by a database instance; from now on, we identify scheme and instance with sublanguage and interpretation, respectively.

The roles of variables and input parameters here are essentially the same as their roles in the classical predicate calculus. Like variables, input parameters have associated types, and in an interpretation they denote objects of that type. However, an interpretation fixes the object assigned to each input parameters. The variables are allowed to range over the set of values of their type determined by the interpretation. Quantifiers are applicable to variables only. As a consequence of the fact that both variables and input parameters are associated with complex types, there are some similarities in their use First, components of variables or of input parameters are selected using attribute names and the dot selector. Second, and probably more significant, an occurrence of an input parameter in a formula can be replaced by an occurrence of a variable of the same type (and vice versa). This interchangeability of variables and input parameters makes it easy to combine formulas to represent, e.g., composition of functions, or to create a filter query from a formula representing a filter transformation.

Terms and their types are defined as follows: a constant is an (atomic) term; an input parameter is a term, of the same type as the parameter; a variable is a term, of the same type as the variable; if t is a term of tuple type, and A is a constant name of that type, then $t.A$ is a term, of the type corresponding to A in the type of t.

Predicates applied to terms yield *atomic* formulas. In particular, $t \in R$ (abbreviated $R(t)$), $t \in s.A$, and $t \in s$ are formulas, as are similar formulas constructed with the equality predicate. We assume that terms in an atomic formula are of types that are compatible with the predicates used in it. Finally, we use connectives and quantifiers to create *non-atomic* formulas. (Only variables may be quantified; this distinguishes variables from input parameters.)

Let $DB = <[D_1, \ldots, D_k], B:[R_1, \ldots, R_n]>$, be an instance of DB, i.e., an interpretation. Let T be any type that uses only domain names from $\{D_1, \ldots, D_k\}$. The

[6] Alternatively, we could use B to denote the input. Our choice conforms to the customary notation in database languages.

choice of an interpretation DB implicitly assigns a *range* to each variable t of type T, namely $DOM(T, D_1, \ldots, D_k)$, as defined in Section 2. Thus, an interpretation assigns a meaning to the quantifiers, and truth values for formulas can be defined in the standard way. A truth value is associated with a formula when each of its free variables is assigned a value from its domain.

Note that whereas the R_i's are used in the construction of terms and formulas, the domain names D_j's are used only in the definitions of the types. (The situation in the algebra is similar.) Nevertheless, the truth value of a formula depends not only on the objects $R_1, \ldots, R_n$ assigned to the R_i's, but also on the domains assigned to the D_j's, since the domains determine the ranges for the variables.

The calculus as presented here is quite simple. It is of interest, from both theoretical and practical viewpoints, to consider more powerful features in the language. We present here several such features. However, we show that they can all be considered as abbreviations. Therefore, the use of such terms does not augment the expressive power of the language.

- Non atomic constants, e.g., $[B{:}5, C{:}\{\}]$ or $\{2, 6, 7\}$.

  The first term can be viewed as an abbreviation for the use of a variable x, where each such use is and-ed with $x.B = 5 \wedge x.C = \{\}$, and where x is declared to have type $[B{:}int, C{:}\ \{\ int\ \}]$. A finite enumerated set $\{ a_1, a_2, \cdots \}$ can be represented by a variable z, and-ed with a formula $y \in z \leftrightarrow y = a_1 \vee y = a_2 \cdots$. The empty set can be represented by z and the formula $\forall y\ (y \notin z)$.

- Non atomic terms involving input parameters, variables and constants, e.g., $[B{:}x, C{:}y]$, $\{R_2\}$, $\{t_1, t_3, t_8\}$. The construction here is as in the previous case.

- A term for the projection of a tuple on several components: $t.[A, B, C]$.

- Defining set terms using the classical mathematical notation $\{x \mid \phi\}$ where ϕ is a formula with only free variable x. (The idea of using set terms in a calculus for complex objects seems to have first appeared in [KI]. Of course, set terms have been in use in mathematics for a long time.) The expression $\{x \mid \phi\}$ can be replaced by a new variable y and-ed with the constraint $\forall x\ (x \in y \leftrightarrow \phi)$. As an example, consider the following query on the suppliers and parts database of Fig. 2.

$$\text{snum}{:}x \mid \{\ \text{pno}{:}y \mid \exists p\ (\text{part}(p) \wedge p.\text{pno} = y\ \} =$$
$$\{\ \text{pnum}{:}y \mid \exists sp\ (\text{supply}(sp) \wedge sp.\text{snum} = x \wedge sp.\text{pnum} = y\ \}$$

This query can be written as follows: Introduce a variable z to stand for the two sets.

$$\text{snum}{:}x \mid \exists z\ [\ \forall y(y \in z \leftrightarrow \exists p((\text{part}\ (p) \wedge p.\text{pno} = y) \wedge$$
$$\forall y(y \in z \leftrightarrow \exists\ sp((\text{supply}(sp) \wedge sp.\text{snum} = x \wedge sp.\text{pnum} = y\ )\ ]$$

We will use the notation $\phi(R_1, \ldots, R_n; t_1, \ldots, t_l)$ for a formula ϕ with input parameters $R_1, \ldots, R_n$, and free variables $t_1, \ldots, t_l$. The input parameters or the variables may occasionally be omitted. Given ϕ as above, we use $\phi(x_1, \ldots, x_n; t_1, \ldots, t_l)$ to denote the formula obtained from ϕ by simultaneous substitution of x_i for R_i. The x_i's may be objects in a database, or terms of the language. In either case, they must be of the appropriate types. Similar notation is used for variable substitution.

A *c-query* is an expression $q \equiv \langle A{:}t \mid \phi \rangle$, for some formula $\phi(R_1, \ldots, R_n; t)$. The pair $A{:}t$ is called the *target list* of the c-query. Let the type of t be T. The c-query q *defines* a query, i.e., a mapping from instances of DB to instances of $\{A{:}T\}$, as follows: given DB, $q(DB)$ is defined by

$$q(DB) = \{\ v \text{ of type } A{:}T \mid \phi(R_1, \ldots, R_n; v) \text{ is true in } DB\ \}$$

In the following, we sometimes refer to a formula *expressing* a query, instead of defining a query. We often identify a c-query with the formula defining it. In particular, we write $\phi(DB)$ instead of $q(DB)$. In our examples, we often omit the name attached to the variable in the target list.

The definition of c-query allows us to associate a c-query with any formula that has a single free variable. We can extend the association to arbitrary formulas as follows. If ϕ contains the free variables $t_1, \ldots, t_l$, then the c-query *associated* with ϕ is $q_\phi \equiv \langle A{:}t \mid \exists t_1 \cdots \exists t_l(\ \phi \wedge t = [A_1{:}t_1, \ldots, A_l{:}t_l]\ )\rangle$, where $A, A_1, \ldots, A_l$ are new arbitrary names. The notation $\phi(DB)$ is extended to such formulas: $\phi(DB) = q_\phi(DB)$.[7]

In the following example, we illustrate the expressive power of the calculus. In

[7] Note that if ϕ contains a single free variable of any type T, then $\phi(DB)$ is a set of elements of type T. If ϕ contains more than one free variable, then $\phi(DB)$ is a set of tuples. In principle, we could restrict attention to formulas with one free variable. The need to consider more than one free variable arises when we prove the equivalence of the calculus and the algebra, since there we consider arbitrary subformulas of a given formula.

particular, we show how composition of queries can be expressed in the calculus.

Example 4.1: Consider the schema $[\ R:\{\ [A,\ A']\ \},S:\{\ [B,\ B':\{B''\}\ ]\ \}\ ]$. The queries are:

(1) The union of R and a set of two constant tuples.

$$r \mid R(r) \vee r = [A:3, A':5] \vee r = [A:0, A':0]$$

(2) Select from S the tuples where the first component is a member of the second component.

$$s \mid S(s) \wedge s.B \in s.B'$$

(3) The cross product of R and S.

$$t \mid \exists r,s\ (R(r) \wedge S(s) \wedge t.[A,\ A'] = r.[A,A'] \wedge t.[B,B'] = s.[B,B'])$$

(4) The join of R and S, on $A = B$. We will express this query as a composition of the cross product, which we have expressed previously, with a selection. Denote the formula describing the cross product by ϕ_3. The output variable of ϕ_3 is t, of type T_t:$[A,\ A',\ B,\ B']$. We first write a c-query expressing a selection on input variable R_t, of type $\{T_t\}$.

$$t \mid (R_t(t) \wedge t.A = t.B)$$

Now, we replace $R_t(t)$ in this query by $\phi_3(t)$, and "and" the resulting formula with the formula for the selection. The result is

$$t \mid \exists r,s\ (R(r) \wedge S(s) \wedge t.[A,\ A'] = r.[A,A'] \wedge t.[B,B'] = s.[B,B'])\ \wedge\ t.A = t.B$$

(5) The powerset of the relation R.

$$t \mid \forall u\ (u \in t \rightarrow R(u))$$

(6) The collection of subsets of the second component of tuples of S, which do not contain the values 2, 4, or 5.

$$t \mid \exists s\ (S(s) \wedge \forall w\ (w \in t \rightarrow (w \in s.B' \wedge w \notin \{2,4,5\})))$$

The two last queries can be written more succinctly, using set expressions and the subset comparator, as follows:

(5) $\quad t \mid t \subseteq R$

(6') $\quad t \mid \exists s\ (S(s) \wedge t \subseteq \{w \mid w \in s.B' \wedge w \notin \{2,4,5\}\ )$

If we allow the use of algebraic operations, such as difference, in the formula, then we obtain an even more succinct expression:

(6'') $t \mid \exists s\,(S(s) \wedge t \subseteq (s.B' - \{2,4,5\}))$. □

We have seen in the example how to express query composition, by "connecting" the output of one query to the input of another. The technique is generally applicable. The example also illustrates notation for expressing "set containment", i.e., restricting a variable to be a subset of a set. Similar notation can be used to express "set assignment", i.e. equating a variable with a set. These are useful abbreviations that facilitate the construction of queries.

As mentioned in the Introduction, in the calculus one can express general functions and construct filters using them. This is explained and illustrated in Appendix A.

Recall the definition of domain independence from the previous section. All the queries above are domain independent. The following is a simple example of a query that is not domain independent.

$t \mid \exists r(R(r) \wedge t.A \neq r.A)$

In the following, we will use the term *domain independent calculus*, meaning the calculus, restricted to domain independent formulas and queries.

5. AN ALGEBRA FOR COMPLEX OBJECTS

The algebra is a functional language, based on a set of primitive operations, and on a set of combinators that produce new operations from given operations, such as composition and filter creation. The operations are mostly extensions of the classical operations. However, since we deal with objects that are recursively defined, it is possible to recursively apply filter creation, hence filters are much more general than in the relational algebra. We have made an effort to use simple basic operations, so operations like nest and unnest that are usually used in N1NF algebra are definable rather than basic in our algebra. Queries are expressed by expressions, with set input types, and a set output type. As we have previously explained, to allow restructuring of complex objects, we need to deal with arbitrary functions, to be used as filters. The algebra also has expressions for such functions.

We first present an informal description of the operations. The first three operations are

the set operations of *union, intersection* and *difference*. They can be used when their arguments are of the same set type, and they produce a result of the same type. Another familiar operation is the *cross product*, that can be applied to any n objects, each of a set type. It creates a set of tuples, where each tuple has n components, one for each participant in the product. Note that this is different from the classical definition, where the product of two sets of tuples of lengths m and n is a set of tuples of length $m+n$. Our definition allows us to take the cross product of any sets, even if they are not sets of tuples. A cross product involving a single set, transforms it into a set of tuples of length one.

A condition that is required for the cross product is that no attribute appears in more than one of the operand types. Since we do want to use the product in situations where this condition does not hold, we introduce a *rename* operation, that takes a type and a name that appears in it and creates a new type with the same structure but with the name replaced by another. This is an operation that has many uses in a model that relies heavily on names. A nontrivial, new, operation is the *powerset*, which creates the set of all subsets of a set. We will explain why it is needed, and also discuss important cases where its use is not needed in later sections.

To perform restructuring, we need *filter* operations. Two special filters are the *tuple collapse* operation, that collapses each tuple of tuples in a set into a (flat) tuple, [8] and the *set-collapse* operation that collapses a set of sets into a set. The classical *select* is a predicative filter. The *project* operation is another example of a transformational filter. Yet another example is the *extend* operation [Gr]: Given a set of tuples, it allows the addition of a component to each tuple. The name of the new component is specified in the operation; in addition, the operation specifies an algebraic expression to compute the value of the new component. This expression is applied to each tuple, to produce the value of the new attribute for that tuple. The combination of *project, extend* and *rename* allows one to perform quite general transformations.

Our approach is to define a general *replace* operation that includes *select, project, extend*, and *tuple-collapse* as special cases. It is constructed from an algebraic expression that specifies the filter transformation. It is essentially a combinator, the *apply to all* of FP [Ba].

The language of the algebra contains the following components:

(1) Domain names, $D_1, D_2, \cdots$.

[8] We can generate the result of the classical product of sets by applying the tuple-collapse to their cross product.

(2) Attributes.

(3) Types, constructed from the domain names and the attributes.

(4) Constants. (Each constant is assumed to belong to one of the domains, i.e., it is of an atomic type.)

(5) Input parameters, $R_1, R_2, \cdots$. It is assumed that each is associated with a type T_i.

(6) Two particular predicates: $\in_{S,T}$ (membership predicate), and $=_{SS}$ (equality predicate) for all types S, T={S}.

(7) Two *constructors*: A *tuple* constructor, denoted by matched brackets "[", "]", and a *set* constructor, denoted by matched set brackets "{", "}".

(8) Filtering brackets "<", ">".

(9) A set of operations, as listed below. All operations generate output of type set, and all of them, except the *replace*, accept only set arguments. [9]

There are two classes of expressions in the algebra, one for representing queries, the other for representing general functions. Both classes are obtained from suitable base expressions, by applying operations and constructors. Our order of presentation is the following. We first list the operations, and explain the meaning of each one of them. The last operation is the *replace*, and we include in its description a definition of *replace specifications*, which are the representations of general functions. These are constructed using the operations and additional constructors. Finally, we define the class of *a-queries*, used to express queries.

The operations, except the *replace*, have the generic form $op(R_1, \ldots, R_n)$. The format of the replace is more complicated . The semantics of an operation is defined, when needed, by specifying its effect on objects $R_1, \ldots, R_n$, of the appropriate types.

Set operations: $\cup$, $\cap$, and - are binary operations. Their arguments must be of the same set type, and they produce a result of the same type.

[9] Strictly speaking, we need to associate a type with each operation, so e.g., we need a different *cross product* for each possible combination of input types. Since the input and output types are always evident from the expression in which the operation appears, we do not make this distinction. That is, we regard the operations as parametric [CW86].

Cross product: The operation is denoted $cross_A(A_1{:}R_1, \ldots, A_n{:}R_n)$, where R_i is of type T_i. It is assumed that $A, A_1, \ldots, A_n$ are new names not used in T_i's, and that no attribute appears in both T_j and T_k, j ≠ k. If for i in [1..n], R_i is of type $\{T_i\}$, then $cross_A(A_1{:}R_1, \ldots, A_n{:}R_n)$ is of type $\{A{:}[A_1{:}T_1, \ldots, A_n{:}\ T_n]\}$. The value is the set of tuples obtained by considering all combinations of values, one from each participating set.

Rename: If R is of type T, A appears in T, and B does not, then $rename_{A \to B}(R)$ is an expression of type T' where T' is obtained from T by replacing A by B. The operation does not changes the value of its argument, only the names used in it.

Powerset: If R is an type $\{A{:}T\}$ and B does not appear in T, then $powerset_B(R)$ is an expression of type $\{B{:}\{A{:}T\}\}$. Its value, when applied to a set R, is the collection of subsets of R.

Set-collapse: This operation is simply a union operation, where the argument is a set of sets, and the result is the union of the member sets. If R is an expression of type $\{B{:}\{A{:}T\}\}$, then $set\text{-}collapse(R)$ is an expression of type $\{A{:}T\}$.

$$set\text{-}collapse\,(R) = \cup\,\{\,x \mid x \in R\,\}$$

We now present the last operation, the *replace*, and with it we define *replace specifications*. A replace specification is used to denote a transformation to be applied to each member of a set of a given type. It is a specification of a function, which may be partial. (Thus, a predicate may be represented by a partial function that is equal to the identity when it is defined.) There is no restriction on the types of the input and output. Given a replace specification G, a replace operation has the form $replace\ <G>(R)$. Its meaning is that when it is applied to a set R, each element of R is transformed by applying G to it. (A more formal definition is given below.)

Before presenting the formal definition of replace specifications, we discuss some aspects of the format in which they are represented. Consider a relation of type $\{[A, B]\}$, and assume we want to project it on the A-component. In the classical algebra, this is written as $R[A]$, or $\pi_A(R)$. In our formalism, it is $replace\ <[A]>(R)$. The point to notice is that the replace specification is '$[A]$'(denoting a function on inputs of type $[A, B]$) rather than, say, $R.A$. There is no explicit input parameter in the replace specification; the input is *implicit* - it is assumed to be the member of the input parameter of the replace operation in which the specification is used. For that reason, the *dot* selector is not used either - an occurrence of an attribute denotes the selection of the appropriate component of the implicit input. This contrasts with the calculus, where an input to a formula representing a function is always

explicit - it is either a a variable, or an input parameter - and the dot selector is explicitly used. Therefore, in the definition of replace specifications given below, we assume a type is given, and we define the class of replace specifications for that type. The type is assumed to be the implicit input type for associated class.

Consider a set of type $\{A:D\}$. To apply *select* (which is a special case of *replace*) to it, we need to be able to refer to its members. The implicit input in this case is of type $A:D$. In order to be able to refer to the implicit input itself in the specification, we assume that the implicit input type is a named type. The output type is not assumed to be named; it inherits the name from the input. That is, if G is a replace specification from type $A:T$ to type T', and R is of type $\{A:T\}$, then *replace* $<G>(R)$ has output of type $\{A:T'\}$. Thus, we may regard G has a mapping from type $\{A:T\}$ to type $\{A:T'\}$.

Consider Example 4.2(2), in Appendix A. In the formula ψ_5, representing a replace specification on the input parameter $R_{B'}$, we needed to also use an input parameter R_B, since the name B is not known in the context of $R_{B'}$. In the corresponding algebraic expression, $R_{B'}$ is the implicit input. The need to refer to an element B means that an additional, explicit, input parameter R_B needs to be used. Thus, in general, a replace specification has an implicit, named, input parameter, and zero or more explicit input parameters. The general format is $G(-, R_1, \ldots, R_n)$, where the first position is reserved for the implicit input. The parameters may be of any type. We use $G(x, x_1, \ldots, x_n)$ to denote the result of applying such a replace specification G to an implicit input object x, and to additional explicit inputs $x_1, \ldots, x_n$.

Given this format for replace specifications, we now define the *replace*.

Replace: If R is of type $\{A:T\}$, and $G(-, R_1, \ldots, R_n)$ is a *replace-specification* (as defined below) from $A:T$ to $A:T'$, then *replace*$<G>(R)$ is a *replace expression* whose input types are $\{A:T\}, T_1, \ldots, T_n$ and whose output type is $\{A:T'\}$. Thus, it has one input parameter of set type, which we refer to as the *primary* input type, and possibly other input parameters of arbitrary types. Given a set R whose elements are of type T, and objects $R_1, \ldots, R_n$, of the types $T_1, \ldots, T_n$, the effect of the operation is to replace each element r of R for which G is defined by $G(r, R_1, \ldots, R_n)$, of type T'. That is,

$$replace<G>(R) = \{ G(r, R_1, \ldots, R_n) \mid r \in R \wedge G(r, R_1, \ldots, R_n) \text{ is defined} \}$$

In the following, whenever a replace specification G is constructed from replace specifications $G_1, \ldots, G_m$, we assume (unless explicitly stated otherwise) that the G_i's all have the same implicit input. Each may also have a subset of $R_1, \ldots, R_n$ as explicit inputs, hence G has the format $G(-, R_1, \ldots, R_n)$. Let $A:T$ be a named type. The *replace*

specifications for $A{:}T$ are defined as follows.

Basis: c, R_i, and B are replace specifications for $A{:}T$, where c is a constant, R_i is an input parameter, and B is a constant name of $A{:}T$. If x is of type T, and x_i is of type T_i (the type of R_i) then

$$c(x) = c; \quad R_i(x, x_i) = x_i; \quad B(x) = x.B$$

Note that, in particular, A is a basic replace specification for $A{:}T$, and $A(x) = x$. In addition, [ ], the empty tuple, is a replace specification for any type, and

$$[\,](x) = \{[\,]\}.$$

The result of *replace* $< [\,] >(R)$, when R is assigned a set R is $\{[\,]\}$, when R is nonempty, and $\{\}$ when R is empty. Thus, it is a predicate that tests a set for emptiness.

Tuple construction: if $G_1, \ldots, G_m$ are replace-specifications from $A{:}T$ to $T_1, \ldots, T_m$, resp., then $[A_1{:}G_1, \ldots, A_m{:}G_m]$ is a replace-specification from $A{:}T$ to $[A_1{:}T_1, \ldots, A_m{:}T_m]$. Its effect is defined by

$$[A_1{:}G_1, \ldots, A_m{:}G_m](x, x_1, \ldots, x_n) = [A_1{:}G_1(x, x_1, \ldots, x_n), \ldots, A_m{:}G_m(x, x_1, \ldots, x_n)].$$

Set construction: If $G_1, \ldots, G_m$ are replace-specifications from $A{:}T$ to T', then $\{G_1, \ldots, G_m\}$ is a replace-specification from $A{:}T$ to $\{A{:}T'\}$. It effect is defined by

$$\{G_1, \ldots, G_m\}(x, x_1, \ldots, x_n) = \{G_1(x, x_1, \ldots, x_n), \ldots, G_m(x, x_1, \ldots, x_n)\}$$

Conditional: If G, H, K are replace specifications for $A{:}T$, where the output type of K is T_K, then **if** $G\theta H$ **then** K is a replace specification from $A{:}T$ to T_K. The predicate θ is one of the two predicates $=$ and $\in$, and it is assumed that the types of the two arguments are compatible with the predicate being used. The effect is that of a selection: it defines a partial function J, such that

$$J(x, x_1, \ldots, x_n) = \begin{cases} K(x, x_1, \ldots, x_n) & \text{if } G(x, x_1, \ldots, x_n)\theta H(x, x_1, \ldots, x_n) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Application of an operation (except *replace*): If $G_1, \ldots, G_m$ are replace-specifications from $A{:}T$ to types $T_{G_1}, \ldots, T_{G_m}$, respectively, and $op(R_1, \ldots, R_m)$ is an algebraic operation of type $T_{G_1}, \ldots, T_{G_m}$ to T', then $op(G_1, \ldots, G_m)$ is a replace-specification from $A{:}T$ to T'. Its effect is defined by

$$(op(G_1, \ldots, G_m))(x, x_1, \ldots, x_n) = op(G_1(x, x_1, \ldots, x_n), \ldots, G_m(x, x_1, \ldots, x_n)).$$

Application of replace: If G_1 is a replace specification from type $A':T'$ to type $A':T''$, and G_2 is a replace specification from type $A:T$ to type $A:\{A':T'\}$, then $replace<G_1>(G_2)$ is a replace specification from type $A:T$ to type $A:\{A':T''\}$. It effect on a set of elements of type $\{A:T\}$ is to first replace each element by its image under G_2, thus obtaining a set of type $\{A:\{A':T'\}\}$, then apply G_1 to each member of each A'-set.

$$replace< replace < G_1>(G_2) >(x, x_1, \ldots, x_n) =$$

$$\{replace<G_1>(y, x_1, \ldots, x_n) \mid y \in G_2(x, x_1, \ldots, x_n)\}$$

This concludes the definition of replace specifications, and since the replace is the last operation, it concludes also the presentation of the operations. The class of *a-queries* is now defined. All a-queries have inputs and output of set types only.

Basis: For every constant c, $\{c\}$ is an a-query. If R is of set type, then R is an a-query.

Application of an operation (except *replace*): If $Q_1, \ldots, Q_m$ are a-queries, with set output types $T_{Q_1}, \ldots, T_{Q_m}$, respectively, and *op* is an algebraic operation of type $T_{Q_1}, \ldots, T_{Q_m}$ to set type T', then $op(Q_1, \ldots, Q_m)$ is also an a-query, with output type T'.

Application of replace: If Q is an a-query, with set output type $A:T$, and if G is a replace specification for $A:T$ such that all input parameters appearing in G (if any) are of set types, then $replace<G>(Q)$ is an a-query.

(Note that the fact that $replace<G>(R)$ is an a-query, when R is of type $\{A:T\}$ and G contains only parameters of set type is a special case. If $R_1, \ldots, R_n$ are the parameters of G, then this a-query has inputs $R, R_1, \ldots, R_n$.)

The (mutually dependent) definitions of operations, replace specifications, and of a-queries are now complete. Note that the definition of the *replace* contains the following operations as special cases: *project* is obtained by constructing a tuple that contains the names of the components in the projection list; *tuple-collapse*, an operation that collapses a tuple of tuples into a "flat" tuple is obtained similarly. (However, note that we need to know the structure of the tuple being collapsed; the replace specification depends on the structure. In contrast, *tuple-collapse* applies to all tuples, with no regard to their structure.) Finally, the use of conditionals allows us to express selections, using conditions that are quite complex. The classical *select* is obtained as the following special case: the input type is $A:T$, B and C are constant names of $A:T$, R is of type $\{A:T\}$, and the operation is $replace <$ if $B\theta C$ then $A >(R)$. In the following, we continue to use for this case the classical notation, i.e., $select< B\theta C >(R)$. We often use ρ for the replace operation, and σ for the select operation.

While the semantics of the conditional is quite clear when it is the main operator of a replace specification, there is more than one way to define the semantics when a conditional replace specification is combined with other replace specifications in a tuple or set constructor, or in an application of operation. In this paper we assume that if the predicate evaluates to **false**, then the value of the expression in which the conditional is embedded is undefined. This semantics is that of composition of partial functions, and is the simplest to translate into a calculus notation.

The semantics of algebraic expressions is defined in a straightforward manner. An interpretation binds each input parameter R_i to an object R_i of the appropriate type. The value of the expression is then obtained by evaluating the constructors and operations in it in the obvious way.

Before discussing further the properties of algebraic expressions, we present several simple queries corresponding to the calculus queries in Example 4.1. We omit names whenever they are not important for understanding the construction of the query.

Example 5.1: Recall the schema $[R: \{[A,A']\}, S: \{[B,B':\{B''\}]\}$. The queries are:

(1) The union of R and a set of two constant tuples.

$$R \cup \{ [A:3, A':5]\} \cup \{[A:0, A':0] \},$$

(2) Select from S the tuples where the first component is a member of the second component.

$$\sigma<B \in B'>(S),$$

(3) The (classical) cross product of R and S.

$$tuple\text{-}collapse\,(cross\,(R,S)),$$

(4) The join of R and S, on $A = B$. This is easily expressed as a composition:

$$\sigma<A = B>(tuple\text{-}collapse\,(cross\,(R,S))).$$

(5) The powerset of the relation R.

$$powerset(R).$$

(6) The collection of subsets of the second component of tuples of S, which do not contain the values 2,4, or 5.

$$set\text{-}collapse\,(\rho<powerset(B'-\{2,4,5\})>(S)).$$ □

Let us now consider some properties of algebraic expressions. All the operations, except *replace*, are essentially of fixed format. (The *cross* has a variable number of arguments, but its definition is still simple. It is also obvious that a binary *cross* suffices.) Also, they all have set inputs and outputs. The *replace* is by far more complicated. It is essentially a class of operations, obtained by applying the combinator *replace* to a filter and an input. In *replace*$<G>(R)$, G may be an arbitrary complex expression, involving many occurrences of operations and constructors, and containing additional input parameters. Essentially, every replace specification G defines an occurrence of the *replace*. Since G may itself be a replace expression, replace expressions may be nested. Also, note the relationship between the implicit input of G and the input parameter R. By virtue of combining them in the *replace*, the implicit input of G is bound to the members of the value of R. Since G may depend on explicit input parameters, the full form is *replace*$<G(-, R_1, \ldots, R_n)(R)$, and this represents a function of $R, R_1, \ldots, R_n$. By the definition of *replace*, R must be of set type. Not so the other input parameters - they may be of any type. Of course, if any of them is not of set type, then this occurrence of *replace* is not an a-query. It is a replace specification.

In the following, we summarize some of the important properties of a-queries and replace specifications.

Proposition 5.1:

(I) For every a-query E, and for every type $A:T$, E is a replace specification for $A:T$ (whose value does not depend on the implicit input).

(II) The set of replace specifications, for any given type, is closed under composition. That is, if $G(-,R_1, \ldots, R_n)$ and $G_1(-,S_1, \ldots, S_m)$ are replace specifications for the type $A:T$, and the output type of G_1 is the same as the type of R_1, then $G(-,G_1(-,S_1, \ldots, S_m), R_2, \ldots, R_n)$ is also a replace specification for $A:T$.

(III) The set of a-queries is closed under composition. That is, if $E(R_1, \ldots, R_n)$ and $E_1(S_1, \ldots, S_m)$ are a-queries, and the output type of E_1 is the same as the type of R_1, then $E(E_1(S_1, \ldots, S_m), R_2, \ldots, R_n)$ is also an a-query.

(In both (II) and (III), the R_i's and the S_j's need not be different.)

Proof: (I) The proof is by induction on the structure of a-queries. Details are left to the reader. The fact that the value of an a-query is independent of the implicit input is obvious - the only way to refer to the implicit input in a replace specification is by using its constant

names. No such names are used in an a-query. [10]

(II) The claim is proved for each fixed G_1, by induction on the structure of G. The base case is when G is R_i, c, or B. Substitution of G_1 makes sense only if $G = R_1$, in which case $G(G_1)$ is simply G_1, which is a replace specification. If G is obtained from some replace specifications by applying a constructor, or an operation (except *replace*), then by induction hypothesis if G_1 is substituted for R_1 in each of them, we obtain replace specifications, and then $G(G_1, \cdots)$ is obtained from them by applying the same constructor or operation. Finally, assume that $G = replace\langle G'\rangle(G'')$, and it was obtained from G') and G''. Then we can substitute $G1$ into G' and into G'', and apply *replace* to obtain G.

(III) The proof is by induction on the structure of E, for each fixed E_1. Details are left to the reader. The only case that deserves special attention is when E is $replace\langle G(-, R_1, \ldots, R_n)\rangle(R)$. By assumption, the types of all input parameters are set types, and we need to consider the case when E_1 is substituted for, say, R_1. (The case of substitution for R is included explicitly in the definition of a-queries.) Since E_1 is also a replace specification, we may use the closure properties proved in (II). □

Corollary 5.2:

(I) If $replace\langle G(-, R_1, \ldots, R_n)\rangle(R)$ is an a-query, and $Q, Q_1, \ldots, Q_n$ are a-queries, with output types corresponding to the types of $R, R_1, \ldots, R_n$, then $replace\langle G(-, Q_1, \ldots, Q_n)\rangle(Q)$ is an a-query.

(II) If $replace\langle G(-, R_1, \ldots, R_n)\rangle(R)$ is a replace specification, where the type of R is $\{A:T\}$, and if $G_1, \ldots, G_n$ are replace specifications for $A:T$, and G is a replace specification from $B:T'$ to $\{A:T\}$, then $replace\langle G(-, G_1, \ldots, G_n)\rangle(G)$ is a replace specification for $B:T'$.

(III) For any R, $\{R\}$ is an a-query.

Proof: We prove only the third claim. We know that R is an a-query. It is also a replace specification. Now, let R' be of any type. Then $replace\langle R\rangle(R')$ is an a-query, equivalent to $\{R\}$. □

The results in the corollary generalize the corresponding parts of the definitions of replace specifications and a-queries.

Proposition 5.3: (Tuple pull-up) Let G be a replace specification for $A:T$, and let $A':T'$ be a

[10] Names may be used in an a-query that contains a *replace*, but then the names refer to the implicit input of that replace specification, which is not the implicit input of the a-query, viewed as a specification.

(tuple) type such that A is a constant name of T', and $T'.A$ is of type T. Then G is also a replace specification for $A':T'$.

Proof: It is easy to see that every constant name of $A:T$ is also a constant name, hence a replace specification, of $A':T'$. Also, every input parameter R and every constant c is a replace specification for $A':T'$. and so is [ ]. Thus, all base replace specifications of $A:T$ are also replace specifications for $A':T'$. The claim now follows easily by induction on the structure of replace specifications. □

We note that the 'application of replace' in the definition of replace specification provides a similar construction for 'pulling-up' through a set type. We will refer to that construction in the following as set pull-up. It is easy to prove that a replace specification of the form $replace<G>(G_1)$ can be obtained only by using set pull-up.

Additional examples, illustrating the usefulness of the propositions, are presented in Appendix B. The appendix also contains a comparison to other algebras presented in the literature.

Our collection of operations is not minimal. We will discuss this issue after showing the equivalence of the algebra and the calculus. We conclude this section with the following observation.

Theorem 5.4: The algebra is a domain independent language. □

6. EQUIVALENCE OF CALCULUS AND ALGEBRA

In this section, we prove that the well-known equivalence holds for our model as well: the algebra and the domain independent calculus have the same expressive power.

Theorem 6.1: The algebra and the domain independent calculus are equivalent. That is, for each a-query there exists an equivalent domain independent c-query, and for each domain independent c-query, there exists an equivalent a-query. □

6.1 From Calculus to Algebra

We follow the lines of the classical proof [U]. We first prove that for each type, there exists an a-query that when applied to a database generates the set of values of that type that can be constructed from atomic values that appear in the database objects. Given that, we

show how to construct for each c-query an a-query that generates on every database the same answer, restricted to the values that appear in the database. The claim then follows.

Given a database instance $DB = <[D_1, \ldots, D_k], B:[R_1, \ldots, R_n]>$, denote by $D_i[B]$ the set of elements of D_i that appear in B.

Claim 6.1: For each domain name D, there exists an a-query, denoted F_D, such that for every database instance DB, $F_D(DB) = D[B]$, where D is the domain associated with D in DB.

Proof: We construct a-queries $F_1(R_1), \ldots, F_n(R_n)$, such that $F_i(DB)$ is the set of elements of D that appear in R_i. Then $F_D \equiv F_1 \cup \cdots \cup F_n$. The a-query $F_i(R_i)$ is constructed by applying the following recursive procedure:

(a) If R_i is of type $\{A:D\}$, then $F_i \equiv R_i$. If R_i is of type $\{A:D'\}$, then $F_i \equiv \emptyset$. (which we may take to be an abbreviation to $R_1 - R_1$).

(b) If R_i is of type $\{A:\{T\}\}$, then apply the procedure to $set\text{-}collapse(R_i)$.

(c) If R_i is of type $\{A:[A_1:T_1, \ldots, A_m:T_m]\}$, then apply the procedure to each of the expressions $replace<A_j>(R_i)$. Then take the union of the a-queries that were generated.

Note that in cases (b) and (c) the procedure is applied to simpler types, hence termination is guaranteed. Also note that if D does not appear in R_i, or in an expression derived by (b) or (c), the procedure can be terminated immediately, and the value returned is $\emptyset$ □

Corollary 6.2: For each domain name D, and for each finite set of constants C, there exists an a-query $F_{D,C}$, such that $F_{D,C}(DB) = D[B] \cup (C \cap D)$.

Proof: $C \cap D$ is a constant set, hence an a-query. The claim follows by the closure of a-queries under union. □

For a type T and a set of constants C, denote by $DOM(T, B, C)$ the set $DOM(T, D_1[B] \cup (C \cap D_1), .., D_k[B] \cup (C \cap D_k))$, that is, the set of values of type T that can be constructed from atomic values that appear in B or in C. (Note that although the D_i's do not appear in the notation, they are implicitly used.)

Claim 6.3: For every type T and a constant set C, there exists an a-query $F_{T,C}$, such that for every database instance DB, $F_{T,C}(DB) = DOM(T, B, C)$.

Proof: We prove the claim using induction on the structure of types.

Basis: T is an atomic type, D. Then $F_{T,C} \equiv F_{D,C}$

Induction: There are two cases to consider.

(a) If $T = \{A{:}S\}$, then $F_{T,C} \equiv powerset(F_{S,C})$.

(b) If $T = [A_1{:}T_1, \ldots, A_n{:}T_n]$, then $F_{T,C} \equiv cross_A(A_1{:}F_{T_1,C}, \ldots, A_n{:}F_{T_n,C})$. □

Note the use of the powerset operation in this proof. It turns out that this is the only place in the translation from calculus to algebra where this operation is used.

In the following, we assume that a c-query $q \equiv \langle l{:}\phi\rangle$ is given. We denote by C_q the set of constants that appear in q, and by $DOM(B;q)$ the set of elements of any type that can be constructed from atomic values that appear in B or in C_q.

Corollary 6.4: Let t be a variable, used in q. There exists an a-query E_t^q, such that for very database instance, $E_t^q(DB)$ is the set of values in $DOM(B;q)$ of the type of t.

Proof: Let T be the type of t, then $E_t^q \equiv F_{T,C_q}$. □

As explained in Section 4, one can associate a c-query with every subformula of ϕ. We refer to such a c-query as a *subquery* of q. We now state and prove the main claim of this subsection.

Proposition 6.5: Let $q \equiv \langle l{:}\phi\rangle$ be a domain independent c-query. Then there exists an a-query E_q, such that for every database DB,

$$E_q(DB) = q(DB) \cap DOM(B;q)$$

Proof: We first prove the claim for databases with minimal domains with respect to B and q. That is, given DB, let DB' be the database obtained from it by restricting the domains to contain only the elements that appear in the B or in q. In other words, each domain D_i is replaced by $D_i \cap DOM(B;q)$. We prove the claim first for DB'.

Let $\phi \equiv \phi(R_1, \ldots, R_n)$. We associate the a-query E_t^q with each variable t, free or bound, that is used in it. The property we need for that a-query is that stated in Corollary 6.4 Let us also associate with R_i the a-query $\bar{R}_i$. The property we need here is that $R_i(DB) = R_i$. Thus, if h is a variable or an input parameter, we have an a-query associated with it, denoted E_h^q. The proof is by induction on the structure of the formula ϕ.

Basis: let ϕ be an atomic formula. It has the form $h\theta k$, where each of h and k is a variable or an input parameter of ϕ, or a variable or an input parameter qualified by a $.A$ selector, and θ is in $\{=, \in\}$. Let A_h, A_k be new names. Then

$$E_\phi \equiv \rho< tl >(\sigma< B_h\ \theta\ B_k >(\ cross(A_h{:}E_h^q,\ A_k{:}E_k^q)\ )\ ),^{11}$$

where tl is a target list constructed according to the structure of ϕ. Thus, if both h and k are variables, then $tl = [A_h,\ A_k]$; if, say, only A_h is a variable, then $tl = A_h$; if both h and k are input parameters, then tl is the empty tuple. (In the last case, ϕ is a predicate. When it evaluates to true, the answer is $\{[\,]\}$, else the answer is $\{\}$, i.e., $\varnothing$.) The names B_h,B_k are chosen according to the specific form of h and k. Thus, if $h = t$, or $h = R_i$, then $B_h = A_h$, and if $h = t.A$, or $h = R_i.A$, then $B_h = A$. For example, if ϕ is $t \in R.A$, then the formula is

$$\rho< A_t >(\ \sigma< A_t \in A_R >(\ cross(\ A_t{:}E_t^q,\ A_R{;}R)\ ),$$

and if ϕ is $R_1 \in R_2$, then the formula is

$$\rho< [\,] >(\ \sigma< A_1 \in A_2 >(\ cross(\ A_1{:}R_1,\ A_2{:}R_2)\ ).$$

Note that in the last case, the a-query is a predicate that tests whether $R_1 \in R_2$ holds in the database. It is easy to see that when ϕ is atomic, for E_q defined as above,

$$E_q(DB) = q(DB) \cap DOM(B;q),$$

as required, whether DB has minimal domains, as assumed above, or not.

Induction: It suffices to consider the cases where ϕ is obtained from simpler formulas by applying $\wedge$, $\neg$, or an existential quantifier. The proof is essentially as in [U]. We use "natural join", that is, *cross product* (preceded by a rename, if necessary), *select* and *project*, in that order, for $\wedge$. Note that the *cross product* here may actually be more than just an application of the *cross* operation. For example, if each of the two formulas has a set of tuples as output, then our *cross* generates a set of tuples, each containing two tuples as components. What we need here is to flatten these tuples, using the *tuple-collapse*, which as we have seen is a version of replace. Thus, the precise form of product to be used depends on the formulas being combined. As each formula may have zero, one or more free variables, there are six cases.

For negation, we take the complement with respect to the product of the E_t^q's, for those t's that appear as free variables in the formula being negated. Finally, we use projection for the existential quantifier.

---

[11] When ϕ is $t \in R$, we can use instead the simpler a-query R.

For each of the three constructors, we assume by induction hypothesis that the claim holds for the subformulas of ϕ, and we prove it for ϕ. The only nontrivial step is the association of projection with the existential quantifier. However, for the minimal-domain database DB', the range of the existentially quantified variable t is $E^?(DB')$, and it follows that projection indeed has the same effect as the existential quantifier. Thus, the claim holds, provided the domains are minimal with respect to B and q, as explained above. Since q is domain independent, the result of applying it to an arbitrary database DB is the same as applying it to the database DB', obtained form DB by restricting the domains to contain only the elements that appear in B or in q. This is also true for every a-query. The claim follows. □

Now, if q is a domain independent c-query, then $q(DB) = q(DB) \cap DOM(B;q)$. This concludes the proof of one direction of Theorem 6.1.

The following points are of interest in the proof. If $T = \{A:S\}$, the set of possible objects of type T is the powerset of the set of objects of type S. Hence we need the *powerset* in the algebra to construct the domains for types and variables (and this is the only place it is used in the proof). Thus, it seems that the algebra must include the *powerset* (or a comparable) operator.

Note also that the powerful *replace* operator is not used in the induction part of the construction, except in a restricted way as a *project* or as a *tuple-collapse*. The same is true of the use of the *replace* in constructing a-queries for the domains of variables. In addition, a simple *select* is used. Thus, the operators that belong to Codd's algebra, combined with rather weak restructuring operations, and with the *powerset* seem to be sufficient to provide the algebra with the necessary expressive power. There is, of course, a price to pay for using only the weak operations. If we construct a-queries as in the proof, then we need first to construct the "domain a-queries" $E^?$. That is, we need to "take the relations apart". Then we need to put the pieces together again, in the desired format. This is a rather akward procedure.

6.2 From Algebra To Calculus

The proof in this direction is more complicated, since we have to account for both queries and replace specifications. We prove the following.

Proposition 6.6:

(q) For every a-query E there exists an equivalent (domain independent) c-query

$q_E \equiv \langle t \mid \phi_E(t) \rangle$.

(r-s) For every replace specification $G(-, R_1, \ldots, R_n)$ there exists a formula $\psi_G(R, R_1, \ldots, R_n; u)$, that represents the same function as G. (R is of the same type as the implicit input of G.)[12]

(Remark: (q) is the claim we need, (r-s) is needed for the induction step of the proof.)

Proof: The claims are proved simultaneously, using induction on the structure of algebraic expressions.

Basis:

(q) The a-queries $\{c\}$, R are equivalent to the c-queries $\langle t \mid t = c \rangle$ and $\langle t \mid t \in R \rangle$, respectively.

(r-s) The replace specifications B, c, R, where B is a constant name of the implicit input type, correspond to the formulas $t = R.B$, $t = c$, $t = R$, respectively. The replace specification [ ] corresponds to $t \in R \wedge u = u$, where the type of u is $T_{[\,]}$, and R corresponds to the implicit input..

Induction:

(q) We consider each of the operations. For most of them, the construction is essentially the same as in the classical proof. We assume that E_1, E_2 are a-queries, and ϕ_{E_1}, ϕ_{E_2} are the formulas in the corresponding c-queries. The a-query E is constructed from E_1 and E_2. We show how to construct ϕ_E from ϕ_{E_1}, ϕ_{E_2}.

- **Set operations:** $E = E_1 \theta E_2$, where θ is one of $\cup$, $\cap$, $-$. We change the two given formulas, if necessary, so that they have the same free variable. Then,

  if $E \equiv E_1 \cup E_2$, then $\phi_E \equiv \phi_{E_1} \vee \phi_{E_2}$
  if $E \equiv E_1 \cap E_2$, then $\phi_E \equiv \phi_{E_1} \wedge \phi_{E_2}$
  if $E \equiv E_1 - E_2$, then $\phi_E \equiv \phi_{E_1} \wedge \neg \phi_{E_2}$

- **Cross product:** $E \equiv cross(E_1, E_2)$. We change the two given formulas so that their free variables are different. Then

  $\phi_E \equiv \phi_{E_1} \wedge \phi_{E_2}$.

- **Rename:** $E \equiv rename_{A \to B}(E_1)$. The attribute A occurs in the type of the free variable

[12] Note that R is used here to refer to the implicit input of G. This differs from its use in the previous section, in *replace*$<G(R)$, where it denotes a set, and the implicit input is a member of the set.

of ϕ_{E_1}, say in t_1. Let t be a new variable whose type is the same as that of t_1, except that A is replaced by B. Then

$$\phi_E \equiv \exists\, t_1(\phi_{E_1}(t_1, \ldots) \wedge t = t_1).$$

- **Powerset:** $E \equiv powerset(E_1)$. The formula ϕ_{E_1} has a single free variable (of set type), say t_1. Let t be a new variable. Then

$$\phi_E \equiv \exists\, t_1\ (\phi_{E_1}(t_1) \wedge\ \forall x(x \in t \rightarrow x \in t))$$

- **Set-collapse:** $E \equiv set\text{-}collapse(E_1)$. As in the previous case, ϕ_{E_1} has a single free variable t, of type set of sets. Then

$$\phi_E \equiv \exists\, t_1\ (\phi_{E_1}(t_1) \wedge t \in t_1)$$

- **Replace:** $E \equiv replace\langle G \rangle(E_1)$. Let the free variable of ϕ_{E_1} be of type $A{:}T$. Then the implicit input of G is of type $A{:}T$. Let $\psi_G(R, R_1, \ldots, R_n; u)$ be the formula that represents the same function as G. (R is the implicit input, of type T, and u is the output, of some type T'. If G is [ ], then u is of type $T_{[\,]}$.) Assume the free variable of ϕ_{E_1} is t. Then

$$\phi_E \equiv \exists\, t\ (\phi_{E_1}(t) \wedge \psi_G(t, R_1, \ldots, R_n; u)).$$

(In ψ_G, R has been replaced by t.)

This concludes the part of the proof dealing with queries.

(r-s) Assume $G_1, \ldots, G_m$ are replace specifications for the type $A{:}T$. Let $\psi_{G_1}, \ldots, \psi_{G_m}$) be the corresponding formulas. Let G be a replace specification. We consider each constructor and operation that can be use to construct G from $G_1, \ldots, G_m$.

- **tuple constructor:** $G \equiv [A_1{:}G_1, \ldots, A_m{:}G_m]$. Then

$$\psi_G \equiv \exists\, t_1 \cdots \exists\, t_m(\psi_{G_1}(t_1) \wedge \cdots \wedge \psi_{G_m}(t_m) \wedge t.A_1 = t_1 \wedge \cdots \wedge t.A_m = t_m).$$

- **Set constructor:** $G \equiv \{A{:}G_1, \ldots, A{:}G_m\}$. Then

$$\psi_G \equiv \exists\, t_1 \cdots \exists\, t_m(\psi_{G_1}(t_1) \wedge \cdots \wedge \psi_{G_m}(t_m) \wedge t = \{t_1, \ldots, t_m\})$$

This can be abbreviated to

$$\psi_G \equiv \psi_{G_1}(t) \vee \cdots \vee \psi_{G_m}(t).$$

- **Conditional:** $G \equiv$ **if** $G_1 \theta G_2$ **then** G_3. Then

$$\psi_G \equiv \exists t_1 \exists t_2 \exists t_3 (\psi_{G_1}(t_1) \wedge \psi_{G_2}(t_2) \wedge \psi_{G_3}(t_3) \wedge t_1 \theta t_2 \wedge t = t_3).$$

This can be abbreviated to

$$\psi_G \equiv \exists t_1 \exists t_2 (\psi_{G_1}(t_1) \wedge \psi_{G_2}(t_2) \wedge \psi_{G_3}(t) \wedge t_1 \theta t_2).$$

- **Application** (except for replace): G is obtained from $G_1, \ldots, G_m$ by the application of an operation. The constructions here are similar to those in the previous part of the proof. We consider some of the cases.

  If $G \equiv G_1 \cup G_2$, then

  $$\psi_G \equiv \psi_{G_1}(t) \vee \psi_{G_2}(t)$$

  Intersection and *difference* are treated similarly, using $\wedge$, $\wedge\neg$, instead of $\vee$.

  If $G \equiv cross_A(A_1{:}G_1, A_1{:}G_2)$, then

  $$\psi_G \equiv \exists t_1 \exists t_2 (\psi_{G_1}(t_1) \wedge (\psi_{G_2}(t_2) \wedge t.A_1 = t_1 \wedge t.A_2 = t_2).$$

  Rename, powerset and *set-collapse* are handled similarly.

- **Application** (replace): Let G_1 be a replace specification from $A'{:}T'$ to $A'{:}T''$, let G_2 be a replace specification from $A{:}T$ to $A{:}\{A'{:}T'\}$, and let G be *replace* $<G_1>(G_2)$. Then

  $$\psi_G \equiv \exists t_2 [\psi_{G_2}(R, \cdots; t_2) \wedge t = \{ t_1 \mid \psi_{G_1}(z, \cdots; t_1), z \in t_2 \}]$$

  $$\equiv \exists t_2 [\psi_{G_2}(R, \cdots; t_2) \wedge \forall t_1 (t_1 \in t \leftrightarrow \exists z (\psi_{G_1}(z, \cdots; t_1) \wedge z \in t_2))]$$

7. SYNTACTIC SAFETY AND THE POWERSET OPERATOR

We have proved the existence of reductions from the domain independent calculus to the algebra and vice versa. Our definition of domain independence was semantic in nature. In this section, we present syntactic restrictions that guarantee domain independence. We will call formulas that are syntactically restricted, by a condition that guarantees domain independence, *safe*. The syntactic restrictions are easily checked, whereas domain independence, even for the relational model, is undecidable [Va]. Thus, not every domain independent formula is safe. However, the definitions seem to be general enough for many practical purposes. In particular, for the more general notion of safety introduced below, we show that the domain independent calculus and the safe calculus are equivalent. Thus, although not every domain independent formula is safe, for every domain independent formula there exists an equivalent safe formula. Furthermore, to our knowledge, all algebraic languages proposed so far [AB,FT,SS...] can only express queries that are expressible in the

restricted class of safe formulas presented here.

The accepted approach to making a formula safe is to require that each variable is attached to a range formula. To define safety for our model, we need to indicate what are legal range formulas for variables. The simplest is, of course, $t \in R$, where R is a name of one of the database objects. However, since we allow nested structures, other possibilities exist. Consider, for example, the scheme R:{ [A, B: { [...] }, C] }. We may want to use a variable whose range is the collection of B-sets in tuples of R. To restrict the range of a variable to this collection, we need first to have a variable, say t, that is range restricted to R. Then we can use another variable, say s, with its range restricted by $s \in t.B$.

In the following, we define range formulas, range restricted variables, and safe formulas. The definitions are mutually recursive.

A *range formula* for a variable t is a formula in which t is the only free variable, having the structure as defined below.

Basis:

(1) $t = R$,

(2) $t = c$, where c is a constant,

are range formulas for t.

Construction: If $\phi, \phi_1, \ldots, \phi_k$ are range formulas for variables $s, s_1, \ldots, s_k$, then the following are range formulas for t.(For each of the following formulas, there is a corresponding implicit assumption about t's type.):

(3) $\exists s\ (\phi(s) \wedge t = s.A\ )$

(4) $\exists s\ (\phi(s) \wedge t \in s\ )$.

(5) $\exists s_1 \cdots \exists s_k\ (\phi_1(s_1) \wedge \cdots \wedge \phi_k(s_k) \wedge t.A_1 = s_1 \wedge \cdots \wedge t.A_k = s_k\ )$.

(6) $\forall s(\ s \in t \leftrightarrow \psi(s)\ )$,

where ψ is a safe formula, as defined below, with s being its only free variable. This can be abbreviated to

$t = \{\ s \mid \psi(s)\ \}$.

(7) $\exists s\ (\phi(s) \wedge \forall x\ (x \in t \rightarrow x \in s\ )\ )$,

which can be abbreviated to

$\exists s\ (\ \phi(s) \wedge t \subseteq s)\ )$,

Closure:

(8) If ϕ_1, ϕ_2 are range formulas for t, then so is $\phi_1 \vee \phi_2$

A variable t is *range restricted* in a formula α, if its type is the empty tuple type, or if the following holds.

(1) If t is free in α, then α has the form $\phi(t) \wedge \psi$, where ϕ is a range formula for t.

(2) If t is existentially quantified in α, then the subformula of α that contains the occurrences of t has the form $\exists t\ (\phi(t) \wedge \psi)$, where ϕ is a range formula for t.

(3) If t is universally quantified in α, then the subformula of α that contains the occurrences of t has the form $\forall t\ (\phi(t) \rightarrow \psi)$, where ϕ is a range formula for t.

A formula is *safe*, if all the variables in it are range restricted, except that if t appears as an auxiliary variable in a formula of the form (6) above, i.e., it appears in a formula of the form $\forall t(t \in t' \leftrightarrow \psi(t))$,then it suffices that t is range restricted in ψ.

The quantifiers in the formats $\exists t\ (\phi(t) \wedge \psi$ and $\forall t\ (\phi(t) \rightarrow \psi)$ are customarily called *bounded* quantifiers, and the formulas are often written as $(\exists\, t \phi(t))\ \psi$ and $(\ \forall\, t \phi(t))\ \psi$, respectively. The exception on universally quantified variables can best be unsdersood by considering the second form of (6) above. It is clear that in that formula t is range restricted if all the variables in ψ are; the auxiliary, universally quantified variable, does not appear in the formula at all, and obviously values for it that do not satisfy ψ need not be considered. Note that the variable with the empty tuple type needs no range restriction - its only value is [ ], anyhow.

By combining the base clauses for range formulas with (4) and with (8), we obtain that formulas that are equivalent to $R(t)$ and $t \in C$, where C is a constant set, are range formulas. In the following, "range formula" means any formula obtained using (1-8) above, or a formula that is obtained from such a formula by simple syntactic simplification.

We refer to the calculus, restricted to safe formulas, as the *safe calculus*. In the following, we prove that the safe calculus is equivalent to the domain independent calculus, and to the algebra. We will prove that by showing that the algebra is at least as expressive as the safe calculus, and the safe calculus is at least as expressive as the algebra.

Proposition 7.1:

(I) If $\phi(t)$ is a range formula for t, then there exists an a-query H_ϕ such that for each database DB, $\phi(DB) = H_\phi(DB)$

(II) If α is a safe formula, then there exists an a-query G_α such that for every database DB, $G_\alpha(DB) = \alpha(DB)$.

Proof : We first present a special form for safe formulas, and we show that every safe formula can be converted to an equivalent formula in this form. We prove the proposition for safe formulas (including those that appear in range formulas, as per (6)) in this form only.

The formula $\forall t\,(\,\phi(t) \wedge \psi\,)$ is equivalent tp $\neg\exists t\,(\,\phi(t) \wedge \neg\,\psi\,)$. That is, the universal quantifier can be eliminated. Similarly, the $\vee$ can be eliminated, using DeMorgan's laws. Hence we assume that only $\wedge$, $\neg$ and $\exists$ are used.

Next, we claim that range formulas can be "pushed in", and attached to the atomic subformulas, using the following equivalences.

(a) $\phi(t) \wedge (\psi_1 \wedge \psi_2) \;\equiv\; \phi(t) \wedge (\phi(t) \wedge \psi_1) \wedge (\phi(t) \wedge \psi_2)$

(b) $\phi(t) \wedge \neg\psi \;\equiv\; \phi(t) \wedge \neg(\phi(t) \wedge \psi)$

(c) $\phi(t) \wedge \exists\, s(\phi'(s) \wedge \psi) \;\equiv\; \phi(t) \wedge \exists\, s(\phi(t) \wedge \phi'(s) \wedge \psi)$

In case (a), if, say, ψ_1 does not contain t, then ϕ is pushed in only to ψ_2.

Using the three equivalences, we can push all range formulas in a given safe formula, including those of the bound quantifiers, to the level of the atomic formulas. Recall that atomic formulas have the form $h\theta k$, where each of h and k is either a variable or an input parameter. Thus, an atomic formula has zero, one or two variables. We define a *safe atomic formula* to have one of the forms β, $\phi(t) \wedge \beta(t)$, or $\phi(t) \wedge \phi'(s) \wedge \beta(t, s)$, where ϕ, ϕ' are range formulas, and β is an atomic formula, with zero, one or two variables, respectively. We say that a safe formula is in *full form*, if it is a safe atomic formula, or it is constructed from safe atomic formulas using the following constructions (in which ϕ_i are used to denote range formulas):

- If ψ_1, ψ_2 are safe formulas in full form, with free variables $t_1, \ldots, t_l$, then so is

  $$\phi_1(t_1) \wedge \cdots \wedge \phi_l(t_l) \wedge \psi_1 \wedge \psi_2$$

- If ψ is a safe formula in full form, with free variables $t_1, \ldots, t_l$, then so is

$$\phi_1(t_1) \wedge \cdots \wedge \phi_l(t_l) \wedge \neg\psi$$

- If ψ is a safe formula in full form, with free variables $t_1, \ldots, t_l$, then so is

$$\exists t_i(\phi_1(t_1) \wedge \cdots \wedge \phi_i(t_i) \wedge \cdots \wedge \phi_l(t_l) \wedge \psi)$$

By the discussion above, for every safe formula, there exists an equivalent safe formula in full form. In the rest of the proof, we assume that all safe formulas are in full form. We now prove the two claims of the proposition, using structural induction on formulas.

(I) The base cases are given by clauses (1) and (2) in the definition of a range formula. The a-query for $t = c$ is $\{c\}$. The a-query for $t = R$ is $\rho<R>(\{c\})$, where c is an arbitrary constant. [13] We also note that there exists a formula that generates the set of values of the empty type, that is, the set $\{[]\}$. Next, let ϕ be a range formula such that, for each subformula of ϕ that is a range formula, (I) holds, and for each safe subformula (II) holds. We consider each of the constructions (3-8) in turn.

Consider (3). Assume $\phi \equiv \exists s\ (\phi'(s) \wedge t{=}s.A\ )$, where ϕ' is a range formula for s. By induction hypothesis, there exists an expression $H_{\phi'}$ corresponding to ϕ'. Then H_ϕ is $\rho<A>(H_{\phi'})$. The cases (4) and (5) are treated similarly - the algebraic operations used are *set-collapse* and *cross*, respectively.

For case (6), assume $\phi \equiv t = \{s \mid \psi(s)\}$), where ψ is a safe formula. By induction hypothesis, there exists an a-query G_ψ corresponding to ψ. Its output on a given database is precisely the value of t for that database. The a-query we need is $\{G_\psi\}$. That it is indeed an a-query follows from Corollary 5.2(III) and the closure of a-queries under composition.

For case (7), we use the *powerset* operation. The a-query is *powerset* $(H_{\psi'})$. For case (8) we use union. This concludes the proof of (I)

(II) Consider a safe formula α in full form. By induction hypothesis, for each range formula $\phi(t)$ in α, there exists an equivalent a-query, H_ϕ. The construction of G_α is almost the same as the construction of the a-query for a given c-query in the previous section. However, instead of starting from atomic formulas, we now start from the safe atomic formulas. That is, we treat β, $\phi(t) \wedge \beta(t)$, or $\phi(t) \wedge \phi'(s) \wedge \beta(s, t)$ as indivisible. Instead of using the domain a-query E_1^q for the domain of the variable t, we use H_ϕ. It is easy to see that now we have

[13] Recall that the result of the application of $t = R$ to a database DB, is $\{R\}$. This is not the result of the a-query R, and this is why we use a more complicated expression.

that

$$E_q(DB) = q(DB)$$

for every safe atomic formula.

For the induction, we have to consider the same three cases. For negation, where α is $\phi_1(t_1) \wedge \cdots \wedge \phi_l(t_l) \wedge \neg\psi$, we use complement w.r.t. the cross product of the H_{ϕ_i}'s, $i = 1, \cdots l$. When α is $\phi_1(t_1) \wedge \cdots \wedge \phi_l(t_l) \wedge \psi_1 \wedge \psi_2$, we take the natural join of the a-queries G_{ψ_1}, G_{ψ_2} and of the H_{ϕ_i}'s. When α is $\exists\, t_i(\phi_1 \wedge \cdots \wedge \phi_i \wedge \cdots \wedge \phi_l \wedge \psi)$, we use projection. Correctness of the construction follows by the same reasoning as presented in Section 6. □

Proposition 7.2:

(q) For every a-query E, there exists an equivalent safe c-query q_E.

(r-s) For every replace specification $G(-, R_1, \ldots, R_n)$, there exists a safe formula $\psi_G(R, R_1, \ldots, R_n)$, that represents the same function as G.

Proof: The proof follows the proof of Proposition 6.6. in the previous section.

(q) It is easy to check that there exist safe c-queries for the base a-queries. For the induction, assume that E is obtained from E_1, E_2 by applying some operation. We assume that we have the safe c-queries $\phi_{E_1} \equiv \phi_1(t) \wedge \psi_1(t)$, $\phi_{E_2} \equiv \phi_2(t) \wedge \psi_2(t)$, corresponding to the a-queries E_1, E_2, respectively, and we use the same constructions as in the proof in Section 6.2. We note that the variables that are bound in these formulas remain bound in the same way, i.e., by bounded quantifiers, in the formulas that we construct from them. We need to consider only what happens to the free variables, and also whether the new variables, if any, are range restricted.

For $E_1 \cup E_2$ we have $(\phi_1(t) \wedge \psi_1(t)) \vee (\phi_2 \wedge \psi_2(t))$. Reordering, we obtain the equivalent formula $(\phi_1(t) \vee \phi_2(t)) \wedge [(\phi_1 \vee \psi_2) \wedge (\phi_2 \vee \psi_1) \wedge (\psi_1 \vee \psi_2)]$. The variable t is range restricted in it by $\phi_1 \vee \phi_2$. Similarly, it is easy to see that t is range restricted in the Boolean combinations corresponding to $E_1 \cap E_2$ and $E_1 - E_2$. The arguments for *rename, cross, set-collapse* are also easy, and are left to the reader. For the *powerset*, i.e., *powerset*(E_1), we have the formula $\exists t\, (\phi_1(t) \wedge \psi(t) \wedge s \subseteq t)$. Obviously t is range restricted, and s is range restricted by clause (7) in the definition of range formulas.

The *replace* is more complicated. Assume E = *replace*$<G>(E_1)$. Let the free variable

of ϕ_1 be of type $A:T$. The implicit input of G is of type $A:T$. Let $\psi_G(R, R_1, \ldots, R_n; u)$ be the formula that represents G, where R corresponds to the implicit input, and u is the output variable. The c-query is

$$\phi_E \equiv \exists t \, (\phi_{E_1}(t) \wedge \psi_G(t, R_1, \ldots, R_n; u)).$$

By (r-s) and the induction hypothesis, $\psi_G(R, R_1, \cdots, R_n)$ is safe. We need to show that the variables in ϕ_E above are safe. Note that t is range restricted, and the version of ψ_G used in ϕ_E was obtained by substituting t for an input parameter in a safe formula.

Lemma 7.3: Let $\psi(R, R_1, \ldots, R_n)$ be a formula, and let x be a variable that is range restricted in it. If $\phi(t)$ is a range formula for t, then x is also range restricted in $\psi' \equiv \exists t \, (\phi(t) \wedge \psi(t, R_1, \ldots, R_n))$.

Proof: Obviously, if x is range restricted in ψ by a formula that does not contain R, then it remains range restricted by the same formula in ψ'. We show, using induction, that if R is replaced by t in the range restriction for x, then the resulting formula, and-ed with $\exists t \, (\phi(t))$, is a range restriction for x in ψ'. For the base cases, we need to consider only $x = R$. Then the modified formula is $\exists t \, (\phi(t) \wedge x=t)$, which is equivalent to $\phi(x)$. For cases (3-7), the range formula for x contains range formulas for variables s, $s_1, \ldots, s_k$, and the range formula for x is constructed from them according to one of these cases. By induction hypothesis, ψ' contains range formulas for these variables, hence x is still range restricted. The final case to consider is when x is range restricted by $\phi_1' \vee \phi_2'$. After replacing R by t, and and-ing with $\exists t \, (\phi(t))$, we can distribute the existential quantifier across the $\vee$, and then use induction hypothesis for each of the resulting disjuncts. □

This concludes the proof of (q).

(r-s) We prove the claim using induction on the structure of replace specifications. We follow the construction of formulas corresponding to replace specifications, as presented in Section 6.2.

The claim trivially holds for base replace specifications. It also holds for the case that the replace specification is [ ], the empty tuple. We consider next the constructors. When a constructor or an operation is applied, the variables in the component formulas are known to be range restricted (with the exception noted in the definition of safety), and if they are not free, then they are in the scope of bounded quantifiers, hence they remain range restricted in the constructed formulas. We need therefore to consider only the variables that were free, and the new variable that is introduced.

For the tuple constructor, we have one new variable, t, and the claim that it is range restricted follows from (5) in the definition of a range restriction. For the set constructor, the second form presented in Section 6, provides for range restriction of t using (8). A similar observation holds for the conditional. Applications of operations, except the replace, are handled as in the case of a-queries above. Note that here also, for the *powerset*, we need clause (7) in the definition of range formulas. It remains to consider the use of the *replace* in the construction of replace specifications. If $G = replace\langle G_1\rangle(G_2)$, then we have by induction hypothesis that ψ_{G_1} and ψ_{G_2} are safe. If we consider the formula ψ_G given in Section 6, we see that t_2 that was free in ψ_{G_2} becomes existentially bound, and is therfore range restricted. The variable z is also range restricted, hence so is t, by (6) in the definition of a range formula. Note that we only need t_1 to be range restricted in ψ_{G_1}, by the exception in the definition of safety. □

We have thus proved the following:

Theorem 7.4: The algebra, the safe calculus and the domain independent calculus are equivalent. □

An alternative, more restricted notion of safety is obtained if in the definition of range formulas, clause (7) is dropped. We call formulas that are safe according to this definition *strictly safe*.

Theorem 7.5: The following are equivalent for a query q:

(i) q is expressible by a strictly safe c-query.

(ii) q is expressible by an a-query in which *powerset* is not used.

Proof: Let us reexamine the proofs of equivalence of the algebra and the safe calculus. In the translation from the calculus to the algebra, the *powerset* was used, in Section 6, only in the construction of a-queries, E^t_q, to represent the domains of types (Claim 6.3). However, when the calculus is safe, these a-queries are not used; instead, the a-queries H_ϕ, corresponding to range formulas ϕ are used. In the construction of H_ϕ, the *powerset* is needed only for the simulation of (7). Hence, if the calculus formula is strictly safe, the *powerset* is not needed.

For the other direction, it can be seen in the proof of Proposition 7.2 that clause (7) is used only in the parts of the construction (in (q) and in (r-s)) dealing with *powerset*. Hence, if the algebra does not contain that operation, (7) is not needed. □

8. INTERPRETED FUNCTIONS AND PREDICATES

In this section we consider the use of arbitrary interpreted functions and predicates in our query languages. In a practical language, we need to use predefined built-in functions (like *sum, average*), and predicates (like *even*), or let a user use his/her own. We call such functions and predicates *interpreted*, since their interpretations (including the interpretation of their input and output domains) are fixed. This is in contrast to the use of function symbols in Logic Programming, where the functions are assigned a meaning in the Herbrand Universe, hence are not interpreted. [2] While the inclusion of arbitrary function and predicate symbols in the (syntax of) the languages is easy, this is not the case when translation between the two languages is concerned. We can translate between the algebra and the domain independent calculus. However, even defining the notion of domain independence in the presence of interpreted functions is a problem. We consider here one approach to deal with this issue, and in particular show the equivalence of the enhanced algebra with a suitably restricted version of the enhanced calculus (provided of course, that the same predicates and functions are included in both languages).

Assume that a set Δ_f of functions, and a set Δ_p of predicates are given. Functions in Δ_f and predicates in Δ_p are typed. Furthermore, the domain names used in those types are associated with fixed domains. These will serve both in the algebra, and the calculus.

To extend the calculus, we introduce terms of the form $f(t_1, \ldots, t_n)$, where $f \in \Delta_f$, and allow their use in formulas. We also extend the set of atomic formulas by adding atomic formulas of the form $P(t_1, \ldots, t_n)$, for $p \in \Delta_p$. In both cases, we assume that the t_i's have the correct types. Examples of c-queries are

$x \mid \exists y\,(R(y) \wedge x.A = y.A \wedge x.B = \mathbf{count}(y.B))$

$x \mid S(x) \wedge x.B \leq 5 \wedge \mathbf{even}(x.B)$

To extend the algebra, we allow any predicate in Δ_p to be used in a conditional replace specification. We also allow using functions from Δ_f in replace-specifications. That is, if G is a replace specification, and f is a function , whose input type matches that of G, then so is $f(G)$. For instance, the following are a-queries.

$replace < [A, \mathbf{count}(B)] > (R)$

---

[2] Note that in our languages we use the predicates of equality and membership, and these are interpreted, since they have fixed meanings. However, these two predicates are part of the mathematical basis of Computer Science. Their use does not imply, in particular, any constraints on the assignment of domains to domain names.

$select_{< B \leq 5 \wedge \text{even}(B) >}(S)$

We now need to reconsider the definitions of a database scheme and instance. In the database scheme, some of the domain names will now be attached to fixed domains. E.g., we may use the name *int* in the scheme, and its interpretation is fixed to be the integers. In addition, the sets Δ_p, Δ_f are included in the scheme, with their fixed interpretations. A database instance is constrained in that the interpretations for some of the domains are fixed in the scheme. It also contains the interpreted functions and predicates. We denote such a database structure by $DBI = <[D_1, \ldots, D_k], B, I>$, where I is the interpretations of the fixed domains and of the additional functions and predicates.

When functions are used, it is difficult to use a semantic notion of domain independence. Given a query, it is not enough to consider for $DOM(B,q)$ the values that appear in the database or in the query; it must be closed under applications of functions and their inverses. For example if the database contains 1, and the functions include +, the domain should contain all integers. This extension of $DOM(B,q)$ seems to defeat our intention in the definition of domain independence. In order to be able to compare the algebra and the calculus, we present now a restricted (semantic) notion of domain independence. The intuition behind the following definition is that in any given algebraic expression, there is a bound on the number of times functions are applied.

Given a database DBI, and a set of constants C, let $ATOM(B, C)$ the set of atomic values of any atomic type that appear in B and C, and for any object x, let $ATOM(x)$ be the set of atomic values that appear in x. The *n-closure* of $ATOM(B, C)$, denoted $close^n(B, C)$, is defined as follows.

- $close^0(B, C) = ATOM(B, C)$
- $close^{n+1}(B, C) = close^n(B, C) \cup (\cup \{ATOM(I(f)(x_1, \ldots, x_l)) \mid f \in \Delta_f,$
  $\forall i,\ 1 \leq i \leq l,\ ATOM(x_i) \subset close^n(B, C)\})$[3]

We now want to define a query q to be n-depth domain independent if it depends only on the values of the n-closure of $ATOM(B,C_q)$, where C_q is the set of constants used in the query. To make this notion precise, we have to relax our restrictions on the interpretations of the functions and predicates in $\Delta_f \cup \Delta_p$ and their domains. For n-depth independence, we

[3] Note that if we have functions with a variable number of arguments, then in the definition we have to apply such a function to every set of arguments that it accepts. Thus, if the integers originally given are 1 and 3, and SUM is a function, then two closure steps generate the set {1, 3, 4, 5, 7, 8}.

allow to use domains that are fixed only on $close^n(B, C_q)$, but their part outside of that set is arbitrary. Note that it is not enough to fix the domains on the atomic elements in B and in C, since other atomic values may be included in $close^n(B, C_q)$ as values of functions. Similarly, the interpretations of the interpreted functions and predicates have to be equal to the fixed interpretations only on the set of values that can be constructed using only atomic values from $close^n(B, C_q)$, but can be arbitrarily chosen outside of that set. Assuming this relaxation, a query is *n-depth domain independent* if $q(DBI) = q(DBI')$, for any DBI' that agrees with DBI on $close^n(B,C_q)$. A query is *bounded depth domain independent* if it is n-depth domain independent, for some n.

Theorem 8.1: Let Δ_f and Δ_p be given. The following are equivalent, for a query q:

(a) q is expressible by an a-query,

(b) q is expressible by a bounded-depth domain independent c-query

Proof: We first show how to associate a *depth* with each a-query. For a-queries and replace specifications that contain no functions, the depth is 0. For a replace specification that is constructed using *tuple construction, set construction,* and *conditional*, the depth is the maximum of the depths of the argument replace specifications. The same holds for *application*, except in the case of applying a replace operation. Assume we have constructed the replace specification $G = replace\langle G_1\rangle(G_2)$, and let the depths of G_1, G_2 be n_1, n_2, respectively, then the depth of G is n_1+n_2. Finally, if the depth of G is n, and $f\in\Delta_f$, then the depth of $f(G)$ is $n+1$. For queries, the depth of $op(Q_1,\ldots,Q_m)$ is the maximum of the depths of the Q_i's, except when op is *replace*. For $replace\langle G\rangle(E)$, the depth is the sum of the depths of G and E.

It is easy to see that if the depth of an a-query E is n, then it is n-depth domain independent. Further, we can use the construction of Section 6 to construct an equivalent c-query. The only extensions to the constructions are in the treatment of replace specifications: in a conditional, θ may be any of the predicates in Δ_p, not just one of $\in$, $=$, and we may apply a function from Δ_f to replace specifications, in addition to being able to apply algebraic operations. The details are left to the reader.

Assume now that $q \equiv \langle t \mid \phi\rangle$ is an n-depth domain independent c-query. We translate it into an a-query following the construction of Section 6.1. However, after constructing E_t^q that, for each database DBI, expresses the set of values that t can take on, that are constructed from $DOM(B,C_q) = close^0(B,C_q)$ (cf. Cor. 6.4), we proceed to construct an a-query to represent the set of values for t taken from $close^n(B, C_q)$. Given an a-query, say E, representing a subset S of a domain of a type $A:T$ the a-query $replace\langle f(A)\rangle(E)$ represents

the set $\{f(x) \mid x \in S\}$. By repeated use of this construction and the union operation, we can construct the required "domain a-queries". These are used in the rest of the construction.

In the construction itself, predicates are treated using selection, just like $\in$, =. In Section 6.1, the terms were only variables and input parameters; each term h was represented by an associated a-query E_h^q. Now, a term may also have the form $f(t_1, ..., t_n)$. We can define E_h for an arbitrary term h, using induction, where

$$E_{f(t_1, ..., t_n)} \equiv replace\langle f(A_1, \ldots, A_n)\rangle(cross(A_1{:}E_{t_1}^q, \cdots, A_n{:}E_{t_n}^q)).$$

Using these as "domain a-queries", we obtain the a-query as in Section 6.1. It is easy to see that, since q is n-depth domain independent, the translation generates an a-query that is equivalent to q. □

We can also generalize the notions of range formulas and safety. Let us extend the definition of range formula by adding the following to (1-8):

(func) $\exists s_1 \cdots \exists s_k(\phi_1(s_1) \wedge \cdots \wedge \phi_k(s_k) \wedge t = f(s_1, \ldots, s_k))$

A formula is now *safe* if it is safe using this extended notion of range restriction, and in its "body" (i.e., the part outside the range formulas) it contains no terms of the form $f(...)$. An immediate consequence of the definition is the following.

Fact: If a formula is safe, then it is bounded depth domain independent.

Consider now the translation of the algebra to the safe calculus presented in Section 7. We augment the translation of replace specifications to take care of the addition of function application as a constructor. If $G' \equiv f(G)$, and we have the formula ψ_G, with free variable t, then the formula for G' is $\psi_{G'} \equiv \exists t\,(\psi_G(t) \wedge s = f(t))$. Since, by induction hypothesis, t is range restricted, so is s using (func) above. Thus, we have,

Theorem 8.2:

(I) The extended safe calculus, with Δ_f and Δ_p, is equivalent to the extended algebra, with Δ_f and Δ_p.

(II) The extended strictly safe calculus, with Δ_f and Δ_p, is equivalent to the extended algebra, with Δ_f and Δ_p, without the *powerset* □

9. RECURSIVE QUERIES

In this section, we show that the domain independent calculus and the algebra for complex objects permits the specification of queries which would require a fixpoint in a more classical approach. In particular, we show that it has the same power as a language based on recursive rules. Our presentation is brief.

The transitive closure of a binary relation cannot be expressed using relational calculus [AU]. We present an example which shows that this operation corresponds to a safe calculus* query in the world of complex objects.

Example 9.1: Consider the relational schema R:{[A, B] }, where A and B have the same type. The transitive closure of R can be computed in the following way:

- A first query ψ_1 gives the set R_1 of tuples [A, B] built using values in R (The variable x is of type [A,B]):

$$\psi_1 \equiv x \mid \exists y,z\ (R(y) \wedge R(z) \wedge (x.A = y.A \vee x.A = y.B) \wedge (x.B = z.A \vee x.B = z.B)).$$

- A query ψ_2 gives the set R_2 of subsets of R_1 (i.e., the powerset of R_1).

$$\psi_2 \equiv x \mid \exists t\ (\psi(t) \wedge \forall z(z \in x \rightarrow z \in t))$$

- Note that R_2 is a set of relations. The query ψ_3 gives the set R_3 of elements in R_2 containing R:

$$\psi_3 \equiv x \mid \psi_2(x) \wedge \forall z\ (R(z) \rightarrow z \in x).$$

- The query ψ_4 gives the set R_4 of elements in R_2 containing R and transitively closed:

$$\psi_4 \equiv x \mid \psi_3(x) \wedge \forall u,v\ (u \in x \wedge v \in x \wedge u.B = v.A \rightarrow [u.A,\ u.B] \in x)$$

- Finally, the transitive closure of R is obtained by intersecting the elements of R_4:

$$x \mid \forall z\ (\psi_4(z) \rightarrow x \in z).$$ □

We now present a simple language based on recursive rules. We handle negation using the concept of "layers" as in [ABW, AG, BNRST, N, VG] and others.

Given a database scheme, the relation names in it, $R_1, \ldots, R_n$, are called *base* relations. The language will use, in addition, names of additional relations, called *derived* relations. The language is based on the calculus. Thus, we define atomic formulas as before, except that derived relation names may be used as well. A *literal* is an atomic formula or a negated

atomic formula. A *rule* is an expression of the form

$$P(t) :- Q_1(t_1), \ldots, Q_n(t_n),$$

where P is a derived predicate, and each Q_i is a literal (i.e., a positive or negative atomic formula). A rule is interpreted as the formula

$$\forall x_1 \cdots \forall x_m (Q_1 \wedge \cdots \wedge Q_n \rightarrow P),$$

where $x_1, \ldots, x_m$ are all the variables appearing in it. A recursive query is a pair $<program, Q>$ where *program* is a finite set of rules, and Q is a positive literal, in which the predicate is a derived relation.

Rules, programs and queries also need to be domain independent. For an extended discussion see [VGT]. For now, the following definition should suffice. Let us call a variable that appears in the body *range restricted* if it appears in a positive literal that is not one of the interpreted predicates $\in$, $=$. The definition is extended as follows: If the body contains $x \in y$, and y is range restricted, then so is x. The set of variables that are range restricted in the body is the smallest set that satisfies the definition above. We will assume that each variable that appears in a head of a rule or in a negative literal, is range restricted in the rule.

We assume familiarity with how to apply a program to a database, i.e., to a collection of facts, when the rules do not contain negative literals. It is well known that the fixed point of the program, viewed as an operator on collections of facts, is the minimal model of the program, viewed as a formula. Negation poses difficulties - it is not always possible to assign a meaning to a program with negation. Layering (also called stratification) was suggested as a solution. We now briefly present the concept of layer [ABW,N, VG]. Since this is not central to the present paper, our presentation will be brief.

A *layering* of a program *program* is a partition $\{prog_1, \ldots, prog_n\}$ of the program (i.e., of the set of rules) such that the following hold. Each element of the partition is called a a *layer*.

- All the rules defining a derived relation are contained in a single layer.
- If the rule $P(x) :- \ldots, Q(y), \cdots$ is in layer i, then either Q is a base relation, or the rules defining it are in some layer j, for $j \leq i$.
- If the rule $P(x) :- \ldots, \neg Q(y), \cdots$ is in layer i, then either Q is a base relation, or the rules defining it are in layer j, for some $j < i$.

A program is *layered*, if there exists a layering for it. Note that a layering for a program

induces a layering for the predicates appearing in it, in which the base predicates belong to layer 0.

One can show that layered programs have nice properties. In particular, a minimal model semantics can be given for such programs. In this semantics, one first computes the fixed point of the rules of the first layer, applied to the database, then the fixed point of the rules of the second layer, applied to the result of the first stage, and so on. It is known [ABW] that the final result is independent of the specific layering chosen for the program. The semantics of layered recursive queries is defined in the obvious way: Compute the extensions of all derived relations, then perform a selection according to the query literal.

To illustrate the previous definition, we now give an example of layered recursive query.

Example 9.2: The database object R is of type R:{[A,B:{[C,C']}]}, and the query corresponds to the derived relation T. The query computes for each tuple of R, the transitive closure of the B-set. To simplify the presentation, we assume that in addition to the predicates used so far, that is, equality and membership, there is a predicate *union* which will allow us to do "grouping" in the flavor of [BNRST]. A literal *union*(x,y,z) is interpreted as "z is the union of y and x".

$S(x, y) :- R(x, y)$

$S(x, z') :- union(z', \{ [u.A, v.B] \}, z), S(x, z), u \in z, v \in z, u.B = v.A$

$S'(x, z') :- S(x, z), S(x, z'), u \in z, \neg u \in z'$

$T(x, z) :- S(x, z), \neg S'(x, z)$

The first and second rule computes in S pairs such that the second component contains the corresponding component that appears in R, and possibly additional elements derived by transitivity. To answer the query, we need to select for each x the unique tuple (x, z) of S where z is maximal. The third rule puts into S' tuples (x, z') such that z' is not maximal for that x. The last rule then selects those that are maximal, using negation. □

We have used the *union* predicate, which is not part of the language. It turns out that it can be defined in terms of membership by a layered program. For a given type T, the program that defines *union* for sets of type {T} is (the variables are all of type {T}.):

$N\text{–}INC\,1(x, z) :- \neg y \in z, y \in x$

Thus, $N\text{–}INC\,1(x, z)$ holds if there is an element in x that is not in z.

$N\text{–}INC\,2(x, y, z) :- N\text{–}INC\,1(x, z)$

$N\text{–}INC\,2(X, y, z) :- N\text{–}INC\,1(y, z)$

$N\text{-}INC2(x, y, z)$ holds if there is an element either in x or in y that is not in z, that is if z does not contain the union of x and y. The sets that contain the union of x and y can now be defined:

$$INC(x, y, z) :- \neg N\text{-}INC2(x, y, z)$$

All that remains is to apply intersection, as in Example 9.1.

$$strict(x, y, z) :- INC(x, y, z), INC(x, y, z'), w \in z, \neg w \in z'$$

$$union_T(x, y, z) :- INC(x, y, z), \neg strict(x, y, z)$$

Note that the program is type specific only through its dependence on the types of the variables. The same program computes the union for another type T', if we assume that the variables are of that type. □

We now have the following result:

Theorem 9.1: A query is expressible as a layered recursive query, if and only if it is expressible in the safe calculus.

Proof: (Sketch) **From recursive queries to safe calculus:** First consider a positive program, i.e., a program without negation. We use a language that contains an input parameter for each of the base relations. We also use a variable that corresponds to the cross product of the derived relations. We include in the formulas developed below a subformula that states that the value for this variable is a cross product of its projections corresponding to the individual relations. We mimic the steps of Example 9.1. For each derived relation, we can construct in the calculus a c-query that when applied to a database generates the set of all values of the type of that relation, that can be constructed from the atomic values that appear in the database or in the given program. (This corresponds to ψ_1 in the example.) The *powerset* then generates all subsets of this "domain", i.e., all possible relations of that type whose entries can be constructed from values in the database or the program. (This corresponds to ψ_2.) By viewing the program as a formula (that is taking the conjunction of the rules), we can select the vectors of relations that satisfy the program. (This corresponds to ψ_3.) Note that by our assumption above that variables in the program are range restricted, the formula corresponding to the program is domain independent. The minimal model is obtained by "intersection". That is, we select the tuples of the vector of derived relations that must be there for any selection of values for the derived relations that are constructed as above. (This corresponds to ψ_4.)

For a general program, we do a similar construction for each layer. That is, after all the layers up to and including layer i have been defined, we treat all the derived relations of those

layers as base relations in the construction of the next layer. Finally after we have a formula defining the values of all derived relations, it is easy to select the values for the query.

From algebra to recursive queries: For each subquery, a derived relation is introduced. It is quite easy to construct a rule that expresses how a subquery is constructed from component subqueries. For example, the powerset operation is treated in the following way: Assume $E_1 \equiv powerset(E_2)$.

$R_1(u) :- R_2(u)$

$R_1(z) :- R_1(u), x \in u, \ union(z, \{x\}, u))$,

where *union* is defined as in Example 9.2. □

To conclude this section, we introduce interpreted functions and predicates into the rule-based language. A similar problem has been independently studied in [Ch], where aggregate functions are introduced in a language resembling that of [AG]. The use of interpreted functions leads to the following two problems:

- Interpreted functions may introduce new values in the database, and then the finiteness of the results of the application of programs is not guaranteed any more.
- Programs with interpreted functions, like programs with negation, may not have a unique minimal model.

As for negation, layering provides a solution to the second problem. It turns out that layering can also provide a solution to the first problem. Indeed, an appropriate layering of programs with interpreted functions (and predicates) leads to a language that has exactly the power of the algebra or the safe calculus. The layering is defined as above, with the additional following constraint:

- if $P(...) :- \cdots Q(...) \cdots$ is in layer i, and an interpreted function occurs in the rule, then either Q is a base relation, or the rules defining it are in some layer j, $j<i$.

To complete the informal description of the language, we make two remarks.

(a) We need to extend the notion of safety as follows: if the body contains $y = f(x)$, and x is range restricted, then y also is considered to be range restricted.

(b) the condition of layering in the context of aggregate functions and predicates could be relaxed. For instance, in the rule

$S(x, f(x)) :- R(x, f(x))$

the presence of R and S in the same layer would not cause any problem, since the rule uses R positively, and does not introduce new values. We chose here to use a strong layering condition to simplify the presentation. Intuitively, the layering should be restrictive enough to guarantee that if an object with potentially new atomic components is constructed, then it appears in a predicate at a strictly higher layer.

Theorem 9.2: Let Δ_f and Δ_p be given. A query is expressible as a layered recursive program, using functions and predicates from these sets, if and only if it is expressible as a bounded depth safe c-query with the same functions and predicates.

Proof: from recursive queries to safe calculus: The proof is like in Theorem 9.1. Interpreted predicates and functions are treated in the obvious way. the bounded depth is a consequence of the layering.

From algebra to recursive queries: Again this is like in Theorem 9.1, with the obvious treatment of interpreted functions and predicates. If the a-query is represented by a tree of depth k, the program has at most k layers. □

10. CONCLUSIONS

We have presented in this paper an algebra and a calculus for complex objects. We extended the notion of domain independence to our model, and proved that the algebra and the domain independent calculus are equivalent. We have also provided a syntactic notion of domain independence, and we have shown that it is powerful enough to capture the semantic notion, thus extending the well known results of the relational model. Perhaps the most interesting result, from a practical viewpoint, is a characterization of the algebra without powerset using a syntactic restriction on the calculus This restricted algebra is important (1) because it has the same power as other algebras already introduced, and (2) because the *powerset* is an operation not likely to be used in practice.

Our results indicate that the equivalence between the calculus and the algebra is fundamental - it is not peculiar to the "flat" relational model. (This by itself is not a new idea, but rather one that keeps being discovered.). We have also shown the equivalence with a calculus does not depend on the auxiliary functions and predicates that are added to the languages. The algebra presents certain methods for transforming sets of objects - mainly vertical filtering (projection) and horizontal, tuple by tuple, processing (selection, extension of tuples), and these can be applied recursively inside objects. The safe calculus presents the same power, although in quite a different form. The functions/predicates used are, in a sense, plug-in routines. The translation between the algebra and the calculus does not depend on the functions and predicates being used.

The results showing that our languages are also equivalent to a recursive language (with/without additional functions and predicates) are also of interest. The algebra as presented here is a functional language, so we have here an equivalence between a functional language and a predicate based, Logic programming oriented language.

Some questions remain. Our translation from the calculus to the algebra seems to be quite tedious and unenlightening. Since this is the direction that may be needed in practice, this translation needs to be improved. One factor that greatly influences this translation is the specific versions of the calculus and of the algebra being used. So far, the published works that describe general implementation efforts do not display a tendency to use the calculus more than the algebra. The complexity of the objects allows for a large variety of algebraic operators and no trend towards a unified language is now seen. Obviously an algebra suitable for end users may (or may not) be quite different from that used for internal processing. More effort needs to be invested in studies that clarify which versions of the language are best suited for which purpose. The results of this paper can be used to help in this effort, and also to help in designing translations for the versions chosen.

Another point that is of interest concerns whether to use parts of the languages, or the full languages. In the relational model, a lot of attention has been devoted to project-select-join queries in the algebra, corresponding to existential conjunctive queries in the calculus. For these subsets of the algebra and calculus, translation is easier than for the full languages. It is not clear that this will be the case in the extended model. While the universal quantifier (or the corresponding division operation) may be expensive for large relations, its cost may be reasonable when it is applied to small subrelations of an object. The use of special set constructors and other abbreviations in the calculus, or of additional more powerful but specialized operations in the algebra, may be a solution. Such special operations and notations are also important for ease of use of the languages. Hence, it is interesting to study such notations, and their influence on the translations between the languages.

A last subject that we mention is that of the expressive power of the languages. Our results indicate that since the structures in the model are richer, the languages have a power that in the relational model cannot be obtained in a first order language. It is interesting to examine which results about expressive power that have been obtained for the classical languages carry over into more complicated models, such as ours, or that of [KV].

REFERENCES

[AB] Abiteboul, S., and Bidoit, N., "Non first normal form relations: an algebra allowing data restructuring", *JCSS* (1986),

[AG] Abiteboul, S., and S. Grumbach, A logical approach to the manipulation of Complex Objects, *Proc. EDBT* (March 1988), Springer Verlag Lecture Notes in Computer Science 303.

[AH1] Abiteboul S., and R. Hull, "IFO: A formal semantic database model," *TODS* (1988).

[AH2] Abiteboul S., and R. Hull, "Object restructuring in semantic database models," *Proc. ICDT*, Roma (1986)

[AU] Aho, A.V., J.D. Ullman, "Universality of data retrieval languages," *Proc. POPL* (1979).

[ABW] Apt, K., H. Blair, and A. Walker, "Toward a Theory of Declarative Knowledge," In *Foundations of deductive databases and logic programming*, J. Minker (ed.), Morgan Kaufmann publishers (1988)

[Ba] Backus, J., "Can programming be liberated from the von Neuman style? A functional style of programming and its algebra of programs," 1977 Turing Award Lecture, *CACM* 21:8 (Aug. 1978).

[BK] Bancilhon, F., and S. Khoshafian, "A calculus for complex objects, *Proc. 4th PODS* (March 1985).

[BNRST] Beeri, C.,S. Naqvi, R. Ramakrishnan, O. Shmueli, and S.Tsur, "Sets and Negation in a Logic Database Language," *Proc. 6th PODS* (March 1987)

[Ch] Chen, L., "Extension of datalog with aggregation functions," *IV journees bases de Donnees Avancees* (March 1988).

[C] Codd, E.F., "A relational model for large shared data banks, " *CACM* 13:6 (June 1970).

[CH] Chandra, A.K., and D.Harel, "Computable queries for relational database systems," *JCSS* 21:2 (1980).

[DM] Dalhaus E., and J. Makowski, "Computable directory queries," manuscript, the Technion (August 1985).

[Fa] Fagin, R., "Horn clauses and database dependencies," *JACM* 29:4 (Oct. 1982).

[FT] Fischer, P., and S. Thomas, "Operators for non-first-normal-form relations," *Proc. 7th COMPSAC*, Chicago,(Nov. 1983).

[Gr] Grey, P., *Logic Algebra and databases*, Ellis Norwood Series on Computers and their applications. (1984)

[HM] Hammer, M., and D. McLeod, "Data description with SDM: a semantic database model," *TODS* 6:3 (1981).

[H] Hull, R., "A Survey of Theoretical Research on Typed Complex Database Objects," manuscript, USC (1986).

[HS] Hull, R. and J. Su, " On the expressive power of database queries with intermediate types," *Proc. 7th PODS* (March 1988).

[HY] Hull, R., and C. Yap, "The format model: A theory of database organization," *JACM* 31:3 (July 1984).

[J] Jacobs, B., "On database logic," *JACM* 29:2 (April 1982).

[JS] Jaeschke, B., and H.-J. Schek, "Remarks on the algebra of non first normal form relations," *Proc. 1st PODS* (March 1982).

[Kl] Klug, A., "Equivalence of relational algebra and calculus query languages having aggregate functions," *JACM* 29:3 (July 1982).

[Ko] Kobayashi, I. "An overview of database management technology," TR CS-4-1, Sanno College, Kanagawa 259-11, Japan, (1980).

[KRS] Korth, H.F., M.A. Roth, and A. Silberschatz , "Extended algebra and calculus for ¬NF relational databases," manuscript, TR-84-36, Department of Computer Science, University of Texas at Austin, 1984, revised, 1985, to appear, *TODS*.

[Ku] Kuper, G.M.,"Logic Programming with Sets," *Proc.6th PODS* (1987)

[KV] Kuper, G.M., M.Y. Vardi, "A new approach to database logic," *Proc. 3rd PODS* (1984)

[Mac] Macleod, I.A., "A database management system for document retrieval applications, *Information Systems* 6:2 (1981).

[Mak] Makinouchi, A., "A consideration on normal form of not-necessarily normalized relations in the relational model," *Proc. 3rd VLDB* (Oct. 1977).

[N] Naqvi, S.A., "A Logic for Negation in Database Systems," *Proc. Foundations of Deductive Databases and Logic Programming* (1986)

[OO] Ozsoyoglu, G. Z. and Ozsoyoglu, "An extension of relational algebra for summary tables," *Proc of 2nd Intl (LBL) Conf. on Statistical Database Management* (Sept.

1983).

[OOM] Ozsoyoglu, G, Z.M. Ozsoyoglu, and V. Matos, "Extending relational algebra and relational calculus with set-valued attrributes and aggregate functions," *TODS* 12:4 (Dec. 1987).

[SS] Schek H.-J, and M. Scholl, "An algebra for the relational model with relation-valued attributes," *Information Systems*. 11:2 (1986).

[U] Ullman,J.D., *Principles of database systems*, Computer Science Press, 2nd ed. (1982).

[VGT] Van Gelder, A., and R. Topor, "Safety and correct translation of relational calculus queries," *Proc. 6th PODS* (1987).

[VG] Van Gelder, A., "Negation as Failure Using Tight Derivations for General Logic Programs," In *Foundations of deductive databases and logic programming,* J. Minker (ed.), Morgan Kaufmann publishers (1988)

[Z] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects," *Proc.11th VLDB*, Stockholm (1985).

Appendix A

We show here how formulas may be used to represent functions and how a filter query can be constructed from a formula representing the filter transformation.

In the formula $\phi(R_1, \ldots, R_n; t_1, \ldots, t_l)$, the R_i's may be regarded as *input*, and the t_i's as *output*. A substitution of database objects for the input parameters defines values for the output variables. In general, a formula ϕ defines a relation on its inputs and outputs - for each assignment of objects to the R_i's, there is a set of values assigned to the t_i's. If the relation is a function, say f, then we say that ϕ is *functional*, and that it *represents* the function f. [16] The function represented by a formula may be partial. If a functional formula contains no input parameters, then it represents a constant function. If a formula contains more than one free variable, then its output is a set of tuples. It is always possible to introduce an additional variable of tuple type, and make it the only free variable, so that its values correspond to the output values of the given formula. Hence we assume, without loss of generality, that functions are represented by formulas with a single free variable.

One should keep in mind the distinction between "a formula representing a function", and "a formula defining a query." A formula ϕ defines the query described, informally, by "all t's such that ϕ." This query is always well defined, irrespective of whether the formula is functional or not. Further, if ϕ is functional, representing, say, the function f, then the query it defines is not f but $\{f\}$, mapping each input to a singleton set. If a query q is defined by ϕ, then there exists a formula representing the function q but this formula is not ϕ (see Lemma 4.1, below). Our choice to associate a query with a formula in this way is consistent with the way queries are represented in the relational calculus. Formulas representing functions are used in the construction of filters.

Lemma 4.1: Let $<t \mid \phi(t)>$ be a c-query. Then there exists a formula $\psi(w)$ that represents the function defined by the c-query.

Proof: The formula $\psi(\,; w)$ is: $\forall t(t \in w \leftrightarrow \phi(t))$. □

We now consider the creation of filters queries. Assume we have an input parameter R of type $\{T\}$. To express a transformational filter operation on R, we first represent the desired function on elements of type T by a formula $\psi(R'; t)$, where R' is of type T. The query is

[16] Strictly speaking, the domains should be considered as part of the input, since they also determine the truth value of the formula. However, a formula is functional with *DB* considered as its input, iff for fixed domains it is functional with the R's considered as its input. Since only the R's are explicit in the formula, and also since we are mainly interested in domain independent formulas, as defined below, we disregard the domains in these definitions.

now expressed by $<t \mid \exists u\ (u \in R \wedge \psi(u; t)>$. The construction of a predicative filter query is similar. The formula ψ representing the predicate has no free variables, i.e., it has the form $\psi(R')$. The query is expressed by $<t \mid t \in R \wedge \psi(t)>$. (The formula expressing the *select* in Example 4.1 illustrates this type of filter query.) The construction of a formula that represents a filter transformation is easy when the structure of the object is simple. It may be quite involved when the structure of the object is complex. In the following example, we show how to construct such formulas *bottom-up*. That is, we show how, given formulas that represent filter transformations on objects of types $T_1, \ldots, T_k$, one may construct formulas to represent filter transformations for types $\{T_1\}$ and $[T_1, \ldots, T_k]$.

Example 4.2: We use the relation S defined in the previous example.

(1) The relation S, with the value 3 substructed from the second component of each tuple. This is a transformational filter.

$$z \mid \exists s\ (\ S(s) \wedge z.B = s.B \wedge \forall w\ (\ w \in z.B' \leftrightarrow (\ w \in s.B' \wedge w \neq 3\ )\ )\ )$$

This can be written as

$$z \mid \exists s\ (\ S(s) \wedge z.B = s.B \wedge z.B' = \{w \mid w \in s.B' \wedge w \neq 3\}\ )$$

Although it is not difficult to construct the formula directly, it is of interest to see how it can be constructed incrementally. We first write a formula ψ_1, representing the predicate on elements of the type of B'' in S. let $R_{B''}$ denote an input parameter of that type.

$$\psi_1(R_{B''}) \equiv R_{B''} \neq 3.$$

From ψ_1 we construct a formula representing a selection on input of type $R_{B'}$, of the type of B' in S.

$$\psi_2(R_{B'}; t) \equiv \forall w\ (\ w \in t \leftrightarrow w \in R_{B'} \wedge w \neq 3\ ).$$

or,

$$t = \{w \mid w \in R_{B'} \wedge w \neq 3\}.$$

Note that although this is a query, we need to use it as a filter in another query. Hence we use the formula that represents it, rather than the formula that expresses it as a query. We now construct a formula representing the desired filter on the type $[B, B']$, of a tuple of S. Let R be an input parameter of that type. One component of the filter is represented by ψ_3, obtained from ψ_2 by replacing $R_{B'}$ by $R.B'$.

$$\psi_3(R; t) \equiv \forall w\ (w \in t \leftrightarrow w \in R.B' \wedge w \neq 3).$$ [17]

The second component is simply $R.B$. The required formula ψ_4 is obtained by "tupling" the two formulas:

$$\psi_4(R;t) \equiv t.B = R.B \wedge t.B' = \{w \mid w \in R.B \wedge w \neq 3\}$$

Finally the query on S is obtained by replacing R by s, where s is a new variable, and adding a "bounded quantifier", $\exists s\ S(s)$, to bound the range of s.

(2) The relation S, with the value of the B component added to the set value of the B' component, and the B component eliminated. The result is of type $\{B':\{B''\}\}$. We first construct a formula to represent the transformation of the set B' to $B' \cup \{B\}$. Here we have a problem: assuming the input parameter of the formula is $R_{B'}$, of the type of B', the name B is not a constant name of that type, and is therefore meaningless for $R_{B'}$. We could directly construct a formula with input parameter of type $[B, B']$, but this approach will not work for more complex queries, where filters on deeply nested objects are used. Our solution is to use an additional input parameter R_B. The formula is

$$\psi_5(R_B, R_{B'}; t) \equiv \forall w\ (w \in t \leftrightarrow w \in R_{B'} \vee w = R_B)$$

or,

$$t = R_{B'} \cup \{R_B\}.$$

This is now converted to a formula representing a filter on R of type $[B, B']$.

$$\psi_6(R;t) \equiv t = R.B' \cup \{R.B\}.$$

The query is now easily obtained.

$$t \mid \exists s\ (S(s) \wedge t = s.B' \cup \{s.B\}).$$ □

[17] This actually is a special case of composition. The function that selects the B' component of R is represented by $t=R.B'$. Replacing $R_{B'}$ by t in ψ_2 and and-ing the two formulas, we obtain ψ_3.

Appendix B

We present here addiotional examples, illustrate how complex a-queries may be expressed in the algebra. We also informally compare the algebra to some of the algebraic languages ikn the literature.

Example 5.2: We first consider the two queries of Example 4.2, and we construct replace specifications corresponding to the ψ formulas there.

(1) The relation S, with the value 3 substructed from the second component of each tuple. We construct replace specifications corresponding to the formulas ψ_1, ψ_2, ψ_4. The implicit input of G_1 is of the type of B'', with name B'', and that of G_2 is of the type of B', with name B'.

$G_1 \equiv$ if $B''\neq 3$ then B''

Using proposition 5.2(II), we obtain

$G_2 \equiv \rho<$ if $B''\neq 3$ then $B'' >(B')$

(Note that the a-query with the same effect has a slightly different form: $\rho<$ if $B''\neq 3$ then $B'' >(R_{B'})$.) Using *select*, we can write G_2 as

$G_2' \equiv \sigma<B''\neq 3>(B')$.

This is also a replace specification for implicit input of type $[B, B']$, by Proposition 5.2(I). Since B is also a replace specification for that type, we obtain G_4, corresponding to ψ_4:

$G_4 \equiv [B, \sigma<B''\neq 3>(B')]$

Finally, the query is expressed by the a-query

$\rho< [B, \sigma<B''\neq 3>(B')] >(S)$.

(2) The relation S, with the value of the B component added to the B' component, and the B component deleted. Here again, we construct the replace specifications bottom up, starting with G_5, corresponding to ψ_5, that has implicit input of the type of B', with that name, and an additional explicit input R_B.

$G_5 \equiv B' \cup \{ R_B \}$.

This is also a replace specification for the type $[B, B']$. Replacing R_B by B (using proposition 5.1(III)), we obtain,

$G_6 \equiv B' \cup \{B\}$.

The a-query is:

$\rho< B' \cup \{B\} >(S)$.

(3) The construction of the previous example may seem to be unnecessarily complicated. We present a more complex query, in which a construction by stages is natural. Let $R:\{A:[B, C:\{E:\{F\}\}]\}$ be a scheme. The query is to add the B-component of each tuple to each E-set in the tuple, and to eliminate the B-component. The replace specification for E-sets is :

$E \cup \{R_B\}$.

This is converted to a replace specification for C-sets:

$\rho< E \cup \{R_B\} >(C)$.

By proposition 5.3, this is also a replace specification for an A-tuple. Replacing R_B by B, we obtain

$\rho< E \cup \{B\} >(C)$.

The query is now obtained as:

$\rho< \rho< E \cup \{B\} >(C) >(R)$. □

As the previous examples demonstrate, names of a type may be used in replace specifications that apply to objects of a nested sub-type. However, not every name can be so used. In general, any name that, given a node x in an object tree can be interpreted as denoting a unique object relative to x, can be used as a constant in a replace specification on x. The set of such names are the labels of nodes that can be reached from x by going up, and possibly also sideways and down, without crossing a set node on the way down. This idea was first presented in [ScSc]. The key observation in this characterization is that a replace specification G on a subtype can be embedded in a replace specification G' for a type only by using tuple pull-up, or set pull-up. Explicit input parameters of G may then be substituted by constant names of the implicit input of G'.

Let $A:T$ be a type. We associate a set of *dynamic constants* with each subtype $A':T'$ of T, relative to T, as follows. The set $dyn_T(A':T')$ is the smallest set of names that satisfies the following conditions.

- The set of constant names of the parent of $A':T'$, except for those that are constant names of $A':T'$, are in $dyn_T(A':T')$.
- $dyn_T(parent(A':T')) \subseteq dyn_T(A':T')$.

It can be easily seen that the dynamic constants of $A':T'$ are the constant names of the nodes above A' in the tree, from the first set type node above A'. For the relation R of Example 5.2(3), the dynamic constants of a node labeled E are C, B, A, R. Given that dynamic constants may be used in embedded replace specifications, we may also construct a-queries in a top-down manner. This is illustrated below.

Example 5.3 : To further illustrate operations on nested structures, and the use of dynamic constants in replace specifications, consider the relation scheme

$R:\{A:[B,\ C:\{D:[E,\ F:\{\ G\ \},H]\ \}\ ]\ \}.$

(Wherever types are omitted, the names denote atomic types.) We will construct a sequence of progressively more complex expressions. First, C is a replace specification for R. The a-query obtained by using it, and the result type are:

(i) $\rho< C >(R);$ $\{\ C:\{D:[E,\ F:\{\ G\ \}\ ]\ \}\ \}$

Next, we may in each tuple of C drop the E component, before performing the replace. The resulting a-query and its type are:

(ii) $\rho< \rho< F >(C) >(R);$ $\{\ C:\{\ F:\{\ G\ \}\}\}.$

Now, assume we want also to transform the set F, by selecting only some of its elements, namely those that are less than or equal to B. We then have

(iii) $\rho< \rho< \sigma< G\leq B >(F) >(C) >(R);$ $\{\ C:\{\ F:\{\ G\ \}\}\}.$

The second *replace* in the last expression replaces each F-set in C by a new F-set, containing only the elements that are $\leq B$. Now, we want to apply a *select* to the new C, using a condition on F. A set F is selected if it contains both 0 and H. We obtain:

(iv) $\rho< < \sigma< \{0,\ 3\} \subseteq F >(\rho< \sigma< G\leq B >(F) >(C)\ ) >(R);$ $\{\ C:\{\ F:\{\ G\ \}\}\}.$

The type is the same as in (iii). The subset predicate is not in our language. However, as for the calculus, we use abbreviations. The subset can be represented by a *select* with the condition $0\in F$, followed by a *select* with the condition $H\in F$. Note that the following expression is wrong:

$\rho< \sigma< \{o,\ H\}\subseteq(\sigma< G<=B >(F) >(C)) >(R).$

Also note that the required effect cannot be achieved by a selection on D. A condition on D has to specified in a selection applied to a parent of D. □

The concept of dynamic constants is useful for queries that perform limited restructuring, and in particular do not change the relative positions of set and tuple nodes. Such queries, as in the examples above, simply prune some paths of an input tree. The situation is more complicated for queries that perform more complex restructuring. For example, in $\rho< \rho<G>(cross(B, C))>(fatR)$, where R has the type $\{A:[B:\{T_1\}, C:\{T_2\}]\}$, a replace specification embedded in G may use both B and C. Indeed, to compute the result of such a query, we first replace each tuple of the input by the cross product of the B and C components, which is a set of tuples element. Then we apply G to the elements of each such set. Obviously, both B and C are constant names for these elements. In general, a query involving embedded replace specifications is evaluated 'top-down', and the meanings of names in each specification are determined only when it is applied.

As additional examples, we show how to simulate operations introduced in the literature in our algebra. Two well known operations are the *nest* and *unnest* [JaSc].

Example 5.4:

(I) Let the scheme be $R:\{A:[B, C, D, E\ ]\ \}$. Nesting on the attributes D,E is accomplished as follows.

$R_1 \leftarrow \rho< [B, C, F:rename_{B\rightarrow B', C\rightarrow C'}(R)\ ]>(R)$.

In this expression, each tuple of R is replaced by a tuple containing the first two components, and a third component which is a copy of R, with the first two attributes renamed. (Thus, R is used both as the argument of the *replace*, whose elements are the implicit inputs of the replace specification, and as an explicit input in the replace specification.

$R_2 \leftarrow \rho< [A, B, G:\sigma< B=B', C=C'>(F)>](R_1)$.

Thus, the copy of R in each tuple is replaced by the subset of tuples in which the B', C' components are equal to those that appear in the B, C component of the parent tuple. The a-query denoting the *nest* is now obtained by projecting out the B', C' attributes.

$R_3 \leftarrow \rho< [A, B, H:\rho< [C, D]>(G)\ ]>(R_2)$.

These three expressions can of course be collected into a single algebraic expression.

(II) For unnest, consider the relation $S:\{A:[B, C, D:\{\ E\ \}\ ]\ \}$. Unnesting on E is accomplished by

$set\text{-}collapse(\rho< [A, B, E]>(D))>(R))$ □

Algebras have been described in [AB, ScSc] where selections and projections can be

applied arbitrarily deep in the database structure. We have seen how to perform a projection in which changes, including selection, are applied to substructures. Another possibility described in [ScSc] is applying a selection in which conditions on transformed subobjects are used, but the result is the members of the set that qualify, without changes.

Example 5.5: Let R:{A:[B:{ C }, D] }, and consider selecting the tuples of R in which a given selection of the B-component is non-empty.

$\sigma < \sigma < C \leq 3 > (B) \neq \emptyset > (R)$ □

It is not difficult to see that we can express in our algebra every query expressible in the algebra of [ScSc]. In [AB], describing the language used in the VERSO system, a relation is defined recursively to be a set of tuples, such that each component may itself be a relation, but at least one of them is atomic. In particular, set and tuple constructors must alternate. In general, a relation has the form R: { [X, R_1, . . . , R_n] }, where X is a non-empty list of atomic attributes, and each of R_i is a relation. Operators are defined recursively on such relations. For example, assume that E and F are two instances of such a relation. Their intersection is defined by

$$E \cap F = \{x(E_1 \cap F_1), ..., (E_n \cap F_n) \mid xE_1 \cdots E_n \in E, xF_1 \cdots F_n \in F \}$$

Since the effect of the operation is defined recursively, the effect depends on the structure of the relations. The operation does not assume its arguments to be of any specific types.

Example 5.6: Consider relations R_1, S_1, with scheme { [A, B] }, and relations R_2, S_2 with scheme {[A, B:{[C, D] }] }. The expressions in our algebra for the VERSO intersections for these two pairs are:

$R_1 \cap S_1$, and

$\rho < [A, B \cap B'] > (\sigma < A = A' > (cross(R_2, rename_{A',B' \leftarrow A,B}(S_2))))$.

Thus, it seems that adding levels of nesting to the relations makes the expressions more complicated. We conjecture that no single expression of our algebra can express the VERSO intersection (and some of their other operations), for arbitrary relations. □

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique