ON THE POWER OF LANGUAGES FOR THE MANIPULATION OF COMPLEX OBJECTS*

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

Catriel Beeri
Department of Computer Science
The Hebrew University
Israel
beeri@humus.huji.ac.il

December 31, 1993

Abstract

Various models and languages for describing and manipulating hierarchically structured data have been proposed. Algebraic, calculus-based and logic-programming oriented languages have all been considered. This paper presents a general model for complex objects, and languages for it based on the three paradigms. The algebraic language generalizes those presented in the literature; it is shown to be related to the functional style of programming advocated by Backus. The notion of domain independence familiar from relational databases 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 have equivalent expressive power. 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 functions and predicates are used in the languages.

*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.

1 Introduction

The first normal form restriction forces the components of tuples in relational databases to be atomic [19]. In the last decade 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 [34, 38, 39]. Many models that incorporate more semantics into databases have been introduced and studied. In the mid 80's a variety of models generalized the relational model by allowing hierarchically structured data; these are the *nested relations* and *complex objects* models[1] [2, 25, 35, 44]. Towards the end of the decade the emphasis shifted to semantic and particularly to object-oriented models. These models support features such as object identity, modelling of behaviour, encapsulation, and *isa* relationships. (An extension of our model with object-oriented features can be found in [7].)

A variety of languages were proposed for these models, encompassing all known paradigms of query languages: algebraic, calculus-based, logic-programming oriented, and SQL-extensions. The variety of features and operations found in those languages is quite confusing. It seems that we still do not have a commonly agreed upon approach to the design of query languages, or even to the generalization of known paradigms to new models. One of our goals in this paper is to improve our understanding of this issue.

1.1 Overview of the Results

In this paper, we present and compare query languages for the model of complex objects. These are values 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. This model lacks features such as object identity and behavior modelling. Nevertheless we believe that, since complex structures are an important component of object-oriented and semantic models, languages that allow one to access such structures are important and worthy of study. We consider a calculus-based language (similar to the one proposed by Jacobs [31], and those in [28, 35]), an algebra, and a logic-programming language (similar to those presented in [13, 14, 37, 36, 4]).

[1]Note that the word "object" as used in "complex object", and in this paper, does not refer to the notion of object as known in object-oriented database models. In particular, our objects have no identity that is distinct from their values. They are essentially composite values.

Our main results concerning these languages are:

- We show that the classical equivalence between the domain independent calculus and the algebra is valid in our model as well.
- Domain independence is a semantic, undecidable, property. We therefore consider syntactic restrictions that guarantee domain independence. Syntactically restricted formulas are called here *safe*. 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.
- To make the algebra equivalent to the calculus, we had to include in it a *powerset* operation, that generates all subsets of a given set. This is an expensive operation, as its output size may be exponential in the size of its input. It allows one to express queries that cannot be computed in polynomial time in the size of the database. It is, therefore, interesting to characterize the power of the algebra without the powerset operation. We present a restricted notion of safety, and we prove that the calculus, thus restricted, corresponds to the algebra without powerset, thereby characterizing the power of many algebras found in the literature.
- It is well-known that even simple recursive queries (e.g., transitive closure) cannot be expressed using relational calculus [8]. This does not hold for our languages: it turns out that the algebra, the safe (or domain independent) calculus, and a language for complex objects based on recursive rules are equivalent. This follows from the use of the *powerset* in the algebra, or equivalently the unrestricted use of the calculus. Expressing recursive queries using *powerset* leads to rather un intuitive and resource consuming computations — another indication that the *powerset* should not be included.

In addition to the technical results described above, we have also tried to address the issue of generalizing linguistic paradigms to new models. We believe that our approach to this issue is also a contribution (although it is not a theorem). We describe it briefly.

If one is interested in generalizing the relational model to allow more structure to be represented, then there are several directions to go. First, one may generalize the notion of a tuple, by allowing more complex type constructors to be used in the construction of the elements stored in database collections. These may include *tuple, set, list array* and so on, and the orthogonality principle implies one should be able to use these constructors in any order. Second, one may allow arbitrary atomic domains, with arbitrary collections of functions and predicates. That is, database users are allowed to define their own types, and use them in the database. (Additionally, one can add features such as object identity, behavior, and so on, but these are not considered here.) It is important to consider both directions. In particular, the ability to construct generalized "tuples" is important, since it means that at least some of the structure that the user is interested in can be defined in terms of constructors that are part of the data model, hence are known to the database system. Storage structures and access paths for such structures are in the domain of the database system, hence efficient organization and access optimization are possible. However, since not all needs of users can be anticipated, or captured by a given set of constructors, it is important also to allow arbitrary data types, with arbitrary operations, to be stored and manipulated.

We consider both directions, although the bulk of the paper is concerned with the first. The basic idea is that the classical language paradigms are viewed as linguistic frameworks that can accept arbitrary type systems as parameters. Regarding the first direction above, the complex object model generalizes the relational model by allowing set and tuple constructors to be recursively applied. A type constructor is equipped with operations and predicates specific to it. These typically include constructors and selectors. For example, for tuples (or records) one has tuple construction as a constructor, and selection by attributes as selectors. For sets, the membership predicate plays the role of a selector. Additional useful operations and predicates are normally also included. Our approach is to consider such a collection of operations and predicates to be given for each type constructor. Each of the classical languages is then extended by adding these operations and predicates. This approach emphasizes orthogonality, rather than notions like minimality of a set of operations. Although our model is based on the *set* and *tuple* constructors, the approach clearly applies to other constructors as well, hence the model and the languages could easily be generalized to include such constructors. Furthermore, the languages are designed so that no assumption is made on the underlying atomic type, and functions and predicates of these types can be freely used. Thus, the second direction is also taken care of. The main results of the paper apply in this general setting (with some restrictions, see Section 8). This generalizes, for example, results of [33, 41].

The approach above works smoothly for the calculus-based and logic-programming paradigms. For the algebra more effort is needed. It has to include operations to allow one to describe explicitly quite complicated restructuring of complex objects. These are obtained essentially as compositions of the type-specific operations. It also needs to have operations that manipulate collections of such elements, combine them in various ways, or apply restructuring functions to them. For the relational model, these are the classical relational algebra operations. For our model, some of these can be adapted without change from the relational model but some, in particular those that deal with restructuring, had to be generalized. We also added one operation to help in restructuring[2]. Our generalization emphasizes the view of the algebra as a functional language, and relies on the notion of higher-order operations for the generalization of the classical operations.

Another issue that had to be considered has to do with the dual nature of the set constructor in our model. On one hand, sets are used to organize the database; we traditionally view a database as a vector of named collections. The algebra operations are tailored for the manipulation of such collections. But, *set* can also be used as a constructor in the construction of elements, so there is a need for set operations for the manipulation of elements that contain set components. Our approach is to allow the use of the algebraic operations on any sets. Thus, the algebra is a recursive language in that algebraic operations can be nested. Nesting of operations in queries allows one to deal in a straightforward manner with the nesting of database values.

The emphasis in our algebra on composition, and on a few selected higher-order operations is closely related to recent research on query languages for bulk types that uses a category-theoretic framework, and in particular to the notions of monads and comprehensions, see [45, 50], and the *monadic algebra* of [17]. Comprehensions can be viewed as a pure form of generalized SQL (a paradigm not considered here). See the last section for further discussion.

1.2 Comparison with Previous Work

Our model generalizes the non-first-normal-form relational models [25, 32, 35, 39, 44]. The objects we deal with can also be seen as the objects resulting in semantic database modeling [5, 30, 27] from the use of aggregation (tuple constructors) and classification

[2]Another operation, the *powerset*, was added not because a need for it was felt in the algebra itself, but rather to make the algebra as expressive as the calculus. As already mentioned, we believe that for practical purposes the opposite approach should be adopted: the calculus should be restricted so as to be as expressive as the algebra without *powerset*.

(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 [5, 6, 30], but we believe that our results can easily be extended if heterogeneous sets are allowed. Our types can be described by trees, that is, our objects are actually values. Unlike in [37], cycles are not allowed in type definitions. However, note that even in a model where objects have identities, hence cycles are allowed, a query result is defined by an expression that is applied to each object in a set, and contains conditions that the object and other objects and values that are reachable from it by attribute application must satisfy. The objects and values reachable from an object form a (virtual) tree. Thus, our languages can be applied to such models as well.

Equivalence results of algebraic and calculus-based languages have been given in [37, 41, 42, 35]. The equivalence of relational algebra and calculus with aggregates was considered in [33], and extended in [41] to relations with set-valued attributes. A comparison is proposed in [35] 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. 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 [23]. They allow nested structures in which internal nodes are sets, and the leaves are relations. They extend the results of [20] concerning completeness of query languages to this model. Their techniques can be extended to our model; however, completeness of languages in the sense of [20] is not treated here. For results about completeness for our model, see [3].

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. In particular, the *powerset* operation allows one to increase the set height of values. Hull and Su [29] exhibit a hierarchy of languages, based on restrictions on the types of intermediate results.

Our main technical results have been included in the previous, unpublished, version of this paper [2]. The current paper differs from that version in one important respect, namely our approach described above to extending existing languages to new models. This has consequences in particular for the algebra. Algebras in the literature have a wide variety of operations and differ quite a lot in how the restructuring problem is tackled. Our algebra is simpler than the one in the previous version, and we believe that it provides a good understanding of which operations are the essential operations for a general algebra. As an additional consequence of the simplicity, the proofs of the major results are simpler and better structured. (Minor errors in earlier proofs have been corrected as well.)

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. 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, arbitrary functions and predicates are introduced into both the algebra and the calculus. Section 9 deals with recursive queries. A summary is presented in Section 10.

2 Complex Types and Values

In the relational model, instances are sets of tuples. That is, the basic constructors are the set and the tuple constructors, and in the construction of a type (i.e., a relation schema), each is used precisely once: first the tuple constructor, then the set constructor. In the nested relational model, each of the two constructors can be used more than once, but it is required that they alternate in any given type. We extend the model further by removing the last restriction, and requiring only that the set constructor is the last one used.

We assume the existence of a set of *domain names* $\widehat{D_1}, \widehat{D_2}, \ldots$ and of an infinite set of *names*, also called *attributes*, $A_1, A_2, \ldots$ Types are structure definitions, that use domain names, set and tuple constructors, and attributes.

Assume that the domain names are associated with *domains* $D_1, D_2, \ldots$ The nature of the elements of the domains is irrelevant in this paper. We frequently omit the domains in type and value definitions. In examples, we use integers. Naturally, collections of domains are also equipped with operations, that is functions that map a domain either to itself or to another domain. For now, we disregard such domain-specific functions, and consider only the functions that operate on tuples and sets — the basic building blocks of our model. Domain-specific functions are treated in Section 8.

The elements of the domains are called *atomic* values. Complex values are constructed from the atomic elements, using the constructors. A type is associated with each value, in the obvious way; each value is an *instance* of a type[3]. Formally, *types* and *values* are defined as follows:

1. If $\widehat{D}$ is a domain name, then $\widehat{D}$ is an *atomic* type. For each a in D, a is a value of this type.

2. If $T_1, \ldots, T_n$ are types, and $A_1, \ldots, A_n$ are distinct attributes, then $[A_1{:}\, T_1, \ldots, A_n{:}T_n]$ is a *tuple* type. If $v_1, \ldots, v_n$ are values of types $T_1, \ldots, T_n$, resp., then $[A_1{:}\, v_1, \ldots, A_n{:}\, v_n]$ is a value of the type. We also include $T_{[]}$ as a type. The only value of this type is $[\,]$, the empty tuple.

3. If T is a type, then $\{T\}$ is a *set* type. Any finite set of values of type T is a value of type $\{T\}$.

Types and values can be viewed as trees. In a type tree, leaves are labeled by their atomic types, and in a value tree, leaves are labeled by values. Internal nodes are labeled in both cases by constructors. Since a tuple constructor includes a sequence of attributes, the edges outgoing from a tuple node are labeled by attributes. A type is *tuple*, or *set*, respectively, according to the constructor at the root of its tree, and is *atomic* if its tree consists of a single node.

The tuple types, as we defined them, are actually record types, as each component has an attribute label. We allow also unlabeled tuples, assuming that an unlabeled n-tuple has (implicitly) the labels $1, \ldots, n$. These labels should not be confused with the integers used as an example of a specific domain. This distinction will be clear from the context.

[3]The statement 'v is of type T' in this paper always assumes a given assignment of domains to domain names.

We also assume that an unlabeled tuple of length 1 is the same as the element in it. That is we identify $[v]$ and v. As attributes serve as selectors, that is, as functions used to select components of tuples, this assumption implies that the function 1 is the identity function, defined on each domain. This assumption is commensurate with definitions of tuples in the literature, and simplifies some of our arguments in the sequel[4].

Let T be a type, different from $T_{[\,]}$, using domain names $\widehat{D}_1, \ldots, \widehat{D}_k$, and let domains $D_1, \ldots, D_k$ be given. The set of all values of type T, that contain only values from D_i in the leaves of type $\widehat{D}_i$, denoted $DOM(T, D_1, \ldots, D_k)$, can alternatively be defined as follows:

1. Replace in the definition tree of T each leaf labeled $\widehat{D}_i$ by D_i.
2. Replace the labels of internal nodes as follows: Each tuple constructor is replaced by a labeled cross-product operator, and set constructor is replaced by a finite powerset operator.
3. Evaluate the tree.

Note that, for $T_{[\,]}$, the set of values is $\{[\,]\}$, independently of the domains.

Variations

The principal variation of the above structure is the nested relation which is at the core of the nested relation model. A nested relations type is a complex object set type in which the set and tuple constructions alternate. For instance,

$$\begin{array}{rcll} T_1 & = & \{\,[A, B, C : \{\,[D, E : \{\,[F, G]\,\}]\,\}\,]\,\}, & \text{and} \\ T_2 & = & \{\,[A, B, C : \{\,[E : \{\,[F, G]\,\}\,]\,\}\,]\,\} & \end{array}$$

are types of nested relations whereas

$$\begin{array}{ll} \{\,[A, B, C : [D, E : \{\,[F, G]\,\}\,]]\,\} & \quad \text{and} \\ \{\,[A, B, C : \{\,\{\,[F, G]\,\}\,\}\,]\,\} & \end{array}$$

are not. (For the first, observe two adjacent tuple constructions; and two set constructions for the second.)

[4] However, for readability, we actually use later the notation id for the identity function.

The restriction imposed on the structure of nested relations is mostly cosmetics. A more fundamental limitation is considered in [1] which describes the data structure and the language used in the Verso system. Like for nested relations, set and tuple constructors must alternate. Further, 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 must be atomic*. The type T_1 above would be acceptable for a Verso relation whereas type T_2 would not since the intermediate set construction contains tuples with no atomic attribute.

A further assumption in the Verso model is that in a Verso instance, for each set of tuples the atomic attributes form a key. Observe that this implies that the cardinality of each set in a Verso instance is bounded by a polynomial in the number of atomic elements occurring in the instance. This bound certainly does not apply for a nested relation of type

$$\{\ [A : \{\ [B : \widehat{D}]\ \}]\ \}$$

which is essentially a set of sets.

3 Databases and Queries

A *database scheme* is a pair $\widehat{DB} = \langle [\widehat{D_1}, \ldots, \widehat{D_k}], [\widehat{R_1} : T_1, \ldots, \widehat{R_n} : T_n] \rangle$, where $T_1, \ldots, T_n$ are set types, involving only the domain names $\widehat{D}_1, \ldots, \widehat{D}_k$. An *instance* of $\widehat{DB}$ is a structure $DB = \langle [D_1, \ldots, D_k], [R_1, \ldots, R_n] \rangle$, where the D_i's are domains and each R_i is a value of $DOM(T_i, D_1, \ldots, D_k)$. We also refer to $\widehat{DB}$ as the *database type*, and to DB as the *database value*; each of the R_i's is called a *relation*.

A *query* of signature $\widehat{DB} \to T$ (with T of set type) is a partial function from instances of $\widehat{DB}$ to instances of T. $\widehat{DB}$ and T are the input and output types of the query. When $\widehat{DB}$ is obvious from the context, we may refer to T as the type of the query. We assume that the domain names used in T are among those in $\widehat{DB}$. The assumption that T is a set type follows the accepted convention in database systems and models, that a query returns a set of values. The result of applying a query q to a database instance DB is denoted $q(DB)$.

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. In this paper, we are concerned with the three well-known paradigms of query languages: calculus, algebra and rule-based deduction.

In general, the value of $q(DB)$, where q is a query, may depend on $D_1, \ldots, D_k$, as well as on the relations[5]. 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 the database relations, or the query) does not change the result of the query [24]. It is well known that the expressions of relational algebra (assuming complement is not used) define domain independent queries. The same holds for our algebra. Calculus formulas do not necessarily define domain independent queries. Domain independence is a semantic property, defined in terms of structures, known to be undecidable even for the classical relational calculus [22, 47]. The same holds for our calculus, since it contains the relational calculus. For relational calculus, there exist syntactic restrictions that guarantee domain independence, yet do not limit its expressive power [46, 48]. We also consider syntactic restrictions and prove a similar result.

Before we introduce the languages, we comment on an important issue that needs to be dealt with in each of them. An important class of operations on complex values, that is almost absent from the relational model, is those that perform restructuring, i.e., changing the structure of each member of a set. Such restructuring can be defined by a function to be applied to each member of the set. (The relational projection operation is a restricted special case.) We call queries in which a function is applied to each element of a set *restructuring* filters. The relational selection illustrates another type of filter, a *predicative* filter, where a predicate is applied to each element of a set. The elements for which its value is true are in the output, unchanged; the others are ignored.

In the relational model the only elements are (flat) tuples, hence restructuring is limited to adding or deleting fields of tuples. In our model, the elements in a set may be of a particular type and the result after restructuring may be of a quite different type. Thus, to be able to express interesting queries we need to be able to express functions of possibly complex input and output types. Therefore, when presenting the languages, we place special emphasis on explaining the mechanisms for expressing such functions. Since this is quite a rich class of functions, it seems that a relatively powerful functional language may be needed, possibly based on some version of the *lambda*-calculus. Such an approach is illustrated in [17] We show that in all the languages we consider, the functions can be expressed without resorting to such a formalism, thus preserving the traditional database approach.

[5]Because of that, domain names have to appear in the signature of queries.

4 A Complex Values Calculus

In this section, we present calculus-based query languages. One language is presented in the first subsection. Variations on the language are considered in the second subsection.

4.1 The Calculus

The calculus is essentially a standard, first-order, many-sorted calculus. The sorts are the types, as defined in Section 2. The domain names are the atomic types, and the non-atomic types are constructed from them using the tuple and set constructors.

As is customary in many-sorted calculi, each constant and each variable is associated with a sort, that is a type, and functions and predicates are associated with signatures. For a k-ary predicate, the signature is a k-tuple of types, and for a k-ary function it is a $k+1$ tuple of types. Actually, as in all database languages, the functions and predicates are generic, that is, they are families of functions and predicates indexed by types. Types of constants and variables, and signatures of functions and predicates, are usually omitted, when they are irrelevant or can be inferred from the context.

The *terms* of the language are defined as usual as the smallest set that contains the *atomic constants* and *variables*, and is closed under the application of functions. The functions of the language are: the *tuple* and *set constructors* and the *attributes* (considered as unary functions on tuple values). As just mentioned, these are actually parameterized families of functions. For each attributes $A_1, \ldots, A_n$ and types $T_1, \ldots, T_n$, we have the n-ary constructor $[\,]_{A_1:T_1,\ldots,A_n:T_n}$. The term obtained by applying it to n terms $t_1, \ldots, t_n$, where each t_i is of type T_i, respectively, is $[\,]_{A_1:T_1,\ldots,A_n:T_n}(t_1, \ldots, t_n)$. As customary in the database literature, we denote it $[A_1\colon t_1, \ldots, A_n\colon t_n]$. When the types are irrelevant, or known, we denote the constructor by $[\,]_{A_1,\ldots,A_n}$, and when we refer to the generic constructor, the attributes are omitted as well. Similarly for each n and each type T, we have the constructor $\{\}^n_T$. Here also, following standard notation, the term obtained by applying this constructor to n terms $t_1, \ldots, t_n$ of type T, is denoted $\{t_1, \ldots, t_n\}$, and the indexes are omitted. Thus, the two constructors not only have a type parameter, but are variadic as well. For the case that $n = 0$, we obtain the empty tuple $[\,]$, and for each type T, the emptyset $\{\ \}$ or $\emptyset$ for that type. Finally, if t is a term of a tuple type that has A as a component, then $t.A$ is a term. (The notation should have been $A(t)$, since A is viewed as a unary function. We use instead the notation customary in the database area.)

Since we have in the language atomic constants, and we can use constructors and selectors. we can construct ground terms that denote non-atomic constants, e.g., $[B : 5, C : \emptyset]$ or $\{2, 6, 7\}$. For convenience, we also use standard abbreviations such as $t.[A, B, C]$. for $[t.A, t.B, t.C]$

Predicates applied to terms (with the proper type restrictions) yield atomic formulas. The set of predicate includes $\widehat{R}_1, \widehat{R}_2, \ldots$, the names of the database relations. It also includes the three binary predicates $=$ (equality[6]), $\in$ (membership) and $\subseteq$ (set containment). Finally, formulas are obtained from atomic formulas by applications of the connectives $\wedge, \vee, \neg$, and the quantifiers $\forall, \exists$. It follows that each database scheme

$$\widehat{DB} = \langle\, [\widehat{D}_1, \ldots, \widehat{D}_k], [\widehat{R}_1 : T_1, \ldots, \widehat{R}_n : T_n]\,\rangle$$

defines a language in which only the domain names $\widehat{D}_1, \ldots, \widehat{D}_k$ are used in type expressions, and only the predicates $\widehat{R}_1, \ldots, \widehat{R}_n$ are used as relation names.

We now consider the semantics of the language. An *interpretation* is any database that is an instance of the scheme, as defined in Section 3, i.e., is a mapping that assigns domains $D_1, \ldots, D_k$ to the names $\widehat{D}_1, \ldots, \widehat{D}_k$, and sets of values $R_1, \ldots, R_n$ (of appropriate types) to $\widehat{R}_1, \ldots, \widehat{R}_n$. The domains for the non-atomic types in a given interpretation are defined as in Section 2, namely, the domain of a tuple type is the (labeled) cross product of the domains of the component types, and the domain of a set type is obtained by applying a finitary powerset operation to the domain of the element type. Note that this differs from the usual definition of interpretations for many-sorted calculi, where the domains for *all* the sorts can be arbitrary. We treat the tuple and set type constructors as having a fixed predefined meaning. Hence the domains for non-atomic types are determined by those of the atomic types. Furthermore, the tuple and set constructors are also interpreted as the functions that map elements to the tuple, or set, constructed from them, and attributes are interpreted as selectors over tuples, so these functions also are interpreted in a fixed way. The built-in predicates $=, \in$ and $\subseteq$ are also given their standard interpretation. From now on, we identify scheme and instance with language and interpretation, respectively.

Let DB be an instance of $\widehat{DB}$. Let T be any type that uses only domain names from $\widehat{D}$. The choice of the 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 3. Thus, an interpretation assigns

[6]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.

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 although the $\widehat{D}_j$'s are used only in the definitions of the types, the truth value of a formula depends not only on the values assigned to the $\widehat{R}_i$'s, but also on the domains assigned to the $\widehat{D}_j$'s, since the domains determine the ranges for the variables. This is the standard approach.

A formula defines a query on databases, as follows. A *c-query* q is an expression $\{x_1, \ldots, x_n \mid \varphi\}$, where $x_1, \ldots, x_n$ are the free variables of[7] φ. The list of free variables is called the *target list* of the c-query. Let the types of $x_1, \ldots, x_n$ be $T_1, \ldots, T_n$. Then c-query q *expresses* a mapping from instances of $\widehat{DB}$ to instances of $\{[T_1, \ldots, T_n]\}$, as follows: given DB, $q(DB)$ is defined by[8]:

$$q(DB) = \{[v_1, \ldots, v_n] \mid v_i \text{ of type } T_i, DB \models \varphi(v_1, \ldots, v_n)\}.$$

We can identify a c-query with the formula in it, and write $\varphi(DB)$ instead of $q(DB)$.

In the following example, we illustrate the expressive power of the calculus. In particular, we show how *composition* of queries can be expressed in the calculus, and how to express restructuring filters, that is, queries that transform each element in a set to a value of another type.

Example 4.1 Consider the schema

$$\langle\ \widehat{R}\!:\!\{[A : \widehat{D}, A' : \widehat{D}]\},\ \ \widehat{S}\!:\!\{[B : \widehat{D}, B'\!:\!\{\widehat{D}\}\,]\,\}\ \rangle$$

(or $\langle\ \widehat{R} : \{[A, A']\},\ \ \widehat{S} : \{[B, B' : \{\ \}\,]\,\}\ \rangle$ when the domain is omitted for brevity). The queries are:

1. The union of $\widehat{R}$ and a set of two constant tuples:

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

2. Select from $\widehat{S}$ the tuples where the first component is a member of the second component:

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

[7] Since all the free variables are included in the target list, it suffices in principle to write the formula only. But, this notation can be extended by adding attributes to name the components.

[8] Since tuples are legal types, we could, in principle, restrict attention to formulas with one free variable. Note that if the formula contains only one free variable, then by our convention on one-component tuples, the definition can be written as $q(DB) = \{v \mid v \text{ of type } T, DB \models \varphi(v)\}$.

3. The (classical) cross product of $\widehat{R}$ and $\widehat{S}$:

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

   where t is of type $T_t = [A, A', B, B' : \{\}]$.

4. The join of $\widehat{R}$ and $\widehat{S}$, on $A = B$. We express this query as a composition of the cross product, which we have expressed above, with a selection. Denote the formula describing the cross product by φ_3. We first write a c-query expressing a selection on predicate $\widehat{R}_t$, of type $\{T_t\}$:

$$\{t \mid \widehat{R}_t(t) \wedge t.A = t.B\}.$$

   Now, we replace $\widehat{R}_t(t)$ in this query by $\varphi_3(t)$:

$$\{t \mid \varphi_3(t) \wedge\ t.A = t.B\},$$

   i.e.,

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

5. Flatten $\widehat{S}$, that is produce a set of flat tuples, each of which contains the first component of a tuple of $\widehat{S}$, and one of the elements of the second component. This is (almost) the well-known unnest operation:

$$\{t \mid \exists s\, (\widehat{S}(s) \wedge t.B = s.B \wedge t.C \in s.B')\}$$

   where t is of type $[B, C]$.

   Note that the last two conjuncts of the formula can be viewed as a formula $\psi(s, t)$, with two free variables, that expresses a relation on the domains of s and domain of t. By prefixing with $\exists s \widehat{S}(s)$ we have transformed it into a restructuring filter on $\widehat{S}$. (Note that this filter creates from each tuple of $\widehat{S}$ a set of tuples, and the result is their union. This is why ψ expresses a relation, not a function.)

6. The powerset of the relation $\widehat{R}$:

$$\{t \mid t \subseteq \widehat{R}\}$$

   with t of the same type as $\widehat{R}$

7. The collection of subsets of the second component of tuples of $\widehat{S}$, which do not contain the values 2, 4, or 5:

$$\{t \mid \exists s(\ \widehat{S}(s) \wedge t \subseteq s.B' \wedge 2 \notin t \wedge 4 \notin t \wedge 5 \notin t)\}.$$

   Here also, the last part of the formula expresses a relation between s to t, and composing with the existential quantifier produces the required filter.

In summary, the example shows how to compose queries by "connecting" the output of one query to the input of another. The technique is generally applicable. It is based on the fact that any subformula of the form $\widehat{R}(t)$ can always be replaced by a formula $\varphi(t)$. Another technique illustrated in the example is how to combine a formula with two free variables, representing a relation, with another to create a restructuring filter query. This also is a generally applicable technique. These two techniques indeed allow the calculus to express all the algebra expressible queries. They will be considered again when we prove the equivalence of the languages.

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:

$$\{t \mid \exists r\ (\widehat{R}(r) \wedge t.B \neq r.A)\}$$

where t is of type $[B, C]$. In the following, we will use the term *domain independent calculus*, meaning the calculus, restricted to domain independent formulas and queries.

4.2 Variations

It is illuminating to compare the calculus with relational calculus. The latter can also be considered as a many-sorted language. However, we have lifted the restriction on the use of the tuple and set constructors, so we have a richer type system. With a type system are normally included type-specific functions and predicates. Our calculus differs from the relational calculus precisely in this aspect: it allows unrestricted use of the functions and predicates associated with the tuple and set type constructors. We have chosen a reasonable, actually rather small, set of functions and predicates. These include a constructor and a selector for tuples, and a constructor for sets. There is no functional selector for sets, but the membership predicate can be viewed as a selector. We have added, for convenience, a comparator of sets.

Naturally, one may ask whether the set of functions and predicates we have added is the only one, or the best. We now consider other possibilities.

We first note that although our set of functions and predicates is small, it is not minimal. The tuple and set constructors can be removed. For example, the term $[B : 5, C : 8]$ can be viewed as an abbreviation for the use of a variable x (of the appropriate type), "anded" with $x.B = 5 \wedge x.C = 8$; and the finite enumerated set $\{5, 8\}$ can be represented by a variable z, "anded" with the formula $y \in z \leftrightarrow y = 5 \vee y = 8$. (The empty tuple is simply represented by a variable of type $T_{[]}$ since $[\,]$ is the only instance of that type.)

So, the tuple and set constructors of the language are redundant. The reasons are actually quite simple. First, the definitions of the types and their domains use the constructors, and the fact that variables are typed determines for each variable a range of values of a specific form. Second, the calculus allows to construct formulas that describe relationships between values. Thus, instead of writing a value as an explicit term, obtained by applying a constructor to some arguments, one can represent it by a variable, and describe its relationship with its arguments. This is precisely how we eliminated the tuple constructor above. For sets, the membership predicate serves as a selector for this purpose. Finally, we note that the set comparator $\subseteq$ is also redundant: $y \subseteq z$ can be expressed by $\forall x(x \in y \rightarrow x \in z)$.

Let us now consider the other direction, namely the inclusion of additional functions and predicates, and possibly other features, in the language. We present here several such extensions, and we show that they can all be considered as abbreviations, hence their use does not augment the expressive power of the language.

Since we have sets as a data type, it is natural to consider additional operations on sets, such as union and intersection. In general, the inclusion of additional operations in a language makes it easier to express queries. For example, using difference, the last query in Example 4.1 can be expressed as

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

It is easy to show that these can be defined in our language. For example, the union of sets S_1 and S_2 can be represented by a variable x "anded" with $\forall y(y \in x \leftrightarrow y \in S_1 \vee y \in S_2)$. As a matter of fact, we prove later, in Section 6, much more: we show there that the calculus can express each of the algebraic operations introduced in Section 5. Therefore, we can augment the calculus by introducing in it all algebra operations without changing its expressive power. (This, in contrast to the relational case, is made possible by the fact that we employ orthogonality in the use of the type constructors.)

Another useful feature is to augment the class of terms by allowing the definition of set terms using the classical mathematical notation of *set comprehension*: $\{x \mid \varphi\}$ where φ is a formula with only free variable x. The idea of using set terms in a calculus for complex values seems to have first appeared in [33]. Of course, set terms have been in use in mathematics for a long time. A similar notation for functional languages, that is indeed very close to database algebraic languages has been proposed in [43], and later proposed as a query language formalism is several papers, e.g., [45]. .

The expression $\{x \mid \varphi\}$ can be replaced by a new variable y "anded" with $\forall x(x \in y \leftrightarrow \varphi)$. As an example, consider the query on a standard suppliers-and-parts database asking for suppliers supplying all parts:

$$\{\ x \mid \ \{y \mid \exists p\,(part(p) \wedge p.pno = y\} \ =$$

$$\{y \mid \ \exists sp\,(supply(sp) \wedge sp.snum = x \wedge sp.pnum = y\}\ \}.$$

This query can be written in our calculus as well, by using auxiliary variables, but its formulation is then more difficult to write and understand. Note that if this kind of terms is allowed, the definitions of terms and formulas become mutually dependent. (However, set comprehension can also be viewed as a macro facility with no effect on the formal definition of the language.)

In summary, given a type system of interest, represented as a set of constructors, there is in general quite a lot of flexibility in the choice of functions and predicates that go with it in a language. For a practical language, considerations such as orthogonality and ease of use play a major role in the design. But, we are not concerned here with languages for use in practice. Our choice of functions and predicates was motivated by the following considerations. Extending the calculus with additional functions or features would probably load the proof of equivalence with the algebra with many additional details, having to do with the additional features, without shedding more light. (Of course, if we added to the calculus all the operations of the algebra, then the proof might become trivial.) On the other hand, we have opted against a minimal language, without the constructors, since we felt that this would also make the proof too involved, and hide some of its structure.

5 A Complex Values Algebra

The framework for the algebra is essentially the same as for the calculus, namely, that of many-sorted universes, where the sorts are the types. An essential difference is that there are no predicates. The database relations are simply named sets of values of appropriate types; a database instance is an environment that assigns values to these names[9]. The predicates $=, \in, \subseteq$ are handled as boolean-valued functions, and so are any predicates defined for the base types. A query is a functional expression to be evaluated in the given database. Thus, the algebra is a functional language. Although this viewpoint is not emphasized in the literature, this is also true for the relational algebra. But, while most functional languages are based on the use of variables and lambda abstraction, the relational algebra uses a different paradigm — it is based on a small set of operations, that encapsulate useful iterations over relations, and that can be *composed* to express queries. ([10] is a seminal paper on this paradigm of functional programming.) Our algebra is based on the same approach.

Some of the classical relational operations, like *union, intersection* and *difference* can be used without change in our algebra, since they are generic with respect to the types of the elements in the sets. The *cross product* can also be used, although its definition needs to be slightly changed. We introduce two more operations, whose role will be clarified in the sequel. The major issue that we need to consider is that of how to perform restructuring of complex objects.

The classical projection is the primary means (with cross product) for restructuring in relational algebra. It is sufficient, since restructuring in the relational model can only map flat record structures to flat record structures. Obviously, we need a more general mechanism. We address this issue as follows: First, we have in the algebra a sublanguage whose expressions denote restructuring functions. Thus, our algebra has two classes of expressions: queries and restructuring functions. Second, we introduce an operation for restructuring, which is actually an operation scheme. To use it, one needs to supply it with a function parameter. Given a function parameter, the operation takes a set as input, and produces as output the set of its elements, each restructured by the given function. The use of function parameters in this fashion allows us to deal with substructures of complex, recursively defined, values. This approach, first used in [44] for projection, leads

[9] For additional details on this difference in viewpoint between the calculus and the algebra, see [15].

to a compact yet powerful notation. In functional language terminology, the operation is a *higher-order* function. Given the idea of a function parameter, it is straightforward to allow also the use of functions and predicates of the base type as parameters. This is considered in Section 8.

The construction of algebraic expressions is more elaborate than that of calculus formulas (or that of relational algebra expressions). We first present and explain the meaning of the operations. Then we define the classes of queries and functions, and present examples to illustrate the expressive power and the programming style of the algebra. We also consider some properties of the classes of functions and queries. Finally, we consider, as we did for the calculus, issues of redundancy and possible variations.

5.1 The Operations

We start by listing the operations, and explaining their semantics. All operations have set input and output types. In the discussion below, we make general remarks about the possible types of each operation, but aside from that we do not consider the issue of type checking or inference.

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

Cross product: If, for $i \in [1..n]$, R_i is of type $\{T_i\}$, then $cross_{[A_1,\ldots,A_n]}(R_1, \ldots, R_n)$ is of type $\{[A_1\colon T_1, \ldots, A_n\colon T_n]\}$. The value is the set of n-tuples that have the A_i component from R_i[10]. Note that whereas in the classical relational algebra the product of two sets of tuples of lengths m and n is a set of tuples of length $m + n$, our operation produces a set of pairs of tuples. We show later how to express the classical product. Our definition allows us to take the cross product of any sets, even if they are not sets of tuples.

Powerset: If R is of type $\{T\}$, then $powerset(R)$ is the collection of all subsets of R. This is a new operation, that does not exist in relational algebra. Its role will be clarified in the sequel.

Set-collapse: This operation is simply an extended union operation. The argument must

[10] We also allow the use of *cross*, that creates unlabeled tuples. It follows from our convention regarding one-component tuples, that $cross(R)$ is equal to R.

be a set of sets, and the result is the union of the member sets.

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

This is also a new operation.

Select The classical relational *select* is a predicative filter, but the set of predicates that can be used in it is restricted. We extend it by allowing the use of $=, \in, \subseteq$ as comparators, and by using arbitrary functions (from the set of functions defined below) as comparands. Thus, this operator accepts a function parameter: it is a higher-order operation. We use the notation[11] $\sigma\langle\, p \,\rangle$ for this operation; p is predicate, i.e., a boolean-valued function. Given a set R, and a predicate p that is applicable to values of the type of the element type of R, $\sigma\langle\, p \,\rangle\,(R)$ is the set containing all the elements of R that satisfy p. Our notation was chosen to show clearly that an instance of *select* is obtained from a general scheme by supplying a function argument.

Replace: This is the main tool for performing restructuring of complex values, which may include the application of functions to substructures of the values. It is a higher-order operation, with a function parameter that describes the restructuring. If f is a function from the set of functions defined below, then[12] $\rho\langle\, f \,\rangle$ is a *replace* operation. If R is a set value, compatible with f in the sense defined below, then:

$$\rho\langle\, f \,\rangle\,(R) = \{f(r) \mid r \in R\}$$

This operation appears under various names in functional languages; for example, *apply-to-all* in FP [10], *map* in many others. It embodies the idea of set construction expressed by the replacement axiom of set theory.

Why is it justified to consider this operation an algebraic operation? Let a relation R of type $\{[A, B, C]\}$ be given, and assume we want to project it on the A, B-components, written as $R[A, B]$, or $\pi_{A,B}(R)$. In addition to the value argument R, this expression has attribute parameters that determine the structure (i.e., type) of the result. It is a simple conceptual step to regard the projection list as a function that transforms each member of R to the desired format. More precisely, each of the attributes is regarded as a unary function; the attribute are combined by using the tuple constructor to the restructuring

[11]Occasionally we use explicitly *select*, for readability.
[12]Occasionally, we use explicitly *replace*.

function $[A, B]$. In our algebra, this would be written as $\rho\langle\, [A, B]\,\rangle\,(R)$ (see below). Thus, the *replace* is a generalization of the classical *project*.

Another example of a restructuring operation is the *extend* operation of [26]: Given a relation, it allows the addition of a component to each tuple. The name of the new component, and the expression defining the function used to compute it are specified in the operation. This function is applied to each tuple to produce the value of the new attribute for that tuple. Obviously, both *project* and *extend* are special cases of the general concept: to apply some function to each element of a set. The *replace* has the additional advantage of generalizing to our more general type system (and to others) with no change.

5.2 Queries and Functions

Both queries and functions are obtained from the same building blocks. Before presenting the definitions, we note that although intuitively queries are also functions, there is a fundamental difference between them. A query is an expression in which relation names appear. In a given database, the query denotes a constant. Using the functional language viewpoint, a query is an expression in which the relational names are free variables; an association of these names to given database relations is an environment, or a substitution, that binds these variables to values. Under this binding, the query expression denotes a constant. It is a function only when we consider all possible environments, that is, it is a function only when we consider all the databases to which it can be applied. Hence, a query is an expression whose type is that of a 0-ary function. In contrast, a function parameter used in a *select* or *replace*, represents a function even when a database is given. Consider the query $Q = \rho\langle\, f\,\rangle(\widehat{R})$. When $\widehat{R}$ is bound to a relation R, Q denotes a value, but in order for this expression to be well defined, f must denote a unary function, that can be applied to each element of R. Note that this is true even when a function expression contains relation names, which are bound in a given environment; the expression must still denote a unary function under this binding. Thus, the type of a function expression is a unary function. Similarly, in $\sigma\langle\, p\rangle$, p must denote a boolean-valued function, that is a unary function from some type to *bool*. This difference between queries and functions entails somewhat different constructions for queries and functions.

We now define the class of algebraic query expressions, or *a-queries*. (But note that a complete definition must also include the definition of functions, given below.) The base queries are: each set constant $\{c\}$ is an a-query, and relation name $\widehat{R}$ is an a-query. The class of a-queries is the least class of 0-ary expressions that contains these, and is closed under *application* of the operations listed above. That is, if $Q_1, \ldots, Q_m$ are a-queries, and op is an m-ary operation then $op(Q_1, \ldots, Q_m)$ is also an a-query[13]. In particular, if $\rho\langle\, f\rangle$ is a replace operation, and E and f are compatible, then $\rho\langle\, f\rangle(E)$ is an a-query, and similarly for $\sigma\langle\, p\rangle(E)$.

We emphasize that although we have higher-order operations, our definition disallows higher-order queries; each query expression must have a value type. Thus, while $R \cup R'$ is a valid a-query, $R \cup$ (i.e. $\lambda S(R \cup S)$) is *not*.

As noted above, the expressions denoting functions and predicates that are used in *replace* and *select* have types of unary functions. Let us now consider the notation for expressing functions. As an example, consider $\pi_{A,B}(R)$, which in our notation is written as $\rho\langle\,[A, B]\,\rangle\,(R)$. In either case, there are no variables; rather, basic functions are combined into more complex functions. Also, there is no explicit notation for the input. It is assumed to be a tuple of the relation to which this function is applied. That is, in the algebraic approach we are given a base set of function names, and we obtain additional functions by combining them. No variables are used. Input is determined from the context in which the function is used.

We define now the set of function expressions. Although we only need unary functions, it includes also non-unary functions. The base functions are: For each constant c, the constant function $\lambda x.c$ is a function expression, and similarly, for each database relation name $\widehat{R}$, $\lambda x.\widehat{R}$ is a function expression. Abusing the notation, we use instead c and $\widehat{R}$, respectively. Additionally, each attribute A is a function expression. We also use id, denoting the identity function[14]. The tuple and set constructors are also included; these are variadic function expressions. The algebraic operations, except *select* and *replace* are also function expressions. (*Select* and *replace* have been instantiated by appropriate functions parameters, hence are not included in the basis; see the closure clause below.) Finally, we have the boolean-valued $=, \in, \subseteq$, and also $\vee, \wedge, \neg$.

The class of function expressions is the least class that contains the base functions above and is closed under:

1. Composition.

[13]This definition emphasizes application. We present later a viewpoint centered on composition.

[14]In the calculus, id is obviously redundant: a term $id(t)$ can always be replaced by t. In the algebra, this function turns out to be useful, as illustrated below.

2. Function construction, using *select* and *replace*.

By function construction we mean the following: If p is a unary predicate, then $\sigma\langle\, p \,\rangle$ is a function expression, and if f is a unary function then $\rho\langle\, f \,\rangle$ is a function expression. Note that $\rho\langle\, f \,\rangle$ is not obtained by using composition. It is an application of a higher-order operation to a function, that produces another function. *Replace* can also be used in compositions as, for example, in $\rho\langle\, g \,\rangle \circ f$, and that differs from its use for function construction. The same applies to *select*.

Although the set of function expressions includes also expressions denoting non-unary functions, only unary functions can be used as parameters for *replace*; these are called in this paper *replace specifications*. Similarly, unary predicates are *selection specifications*.

There is one detail regarding notation that we still need to consider. For unary functions, $\circ$ is sufficient for expression compositions. But, we have also non-unary functions. Hence, we use a notation like that of application. For example, if h is a binary function, and f, g are unary functions, then we denote by $h(f, g)$ the function $\lambda x.h(f(x), g(x))$[15].

Let us give a few examples: A, $A(B), A(id), [\,]_{A,B}(A, C), cross_{A,B}(C, D)$ are all function expressions. Note that in the second expression B must be a tuple type with an A-component, and in the last expression C and D must be set-valued. That is, type compatibility of functions and arguments must be enforced. Obviously, the expressions $A, A(id)$ denote the same function, and so do c and $c(A)$. In particular, composing any function with c or $\widehat{R}$ (on the left) is redundant; it can be simplified to c and $\widehat{R}$, respectively. As another example, $\rho\langle\, [\,] \,\rangle\,(\widehat{R})$ is $\{[\,]\}$, when the associated relation R is nonempty, and $\emptyset$ otherwise. Thus, we can test a set for emptiness[16]. In $\rho\langle\, \{id\} \,\rangle\,(\widehat{R})$ the replace operation adds one additional level of set nesting to the relation. Note the use of id here as a means for handling the implicit input. As another example involving id, $\rho\langle\, \widehat{R} \cup id \,\rangle\,(\widehat{S})$, when $\widehat{S}$ is a set of sets, adds the contents of $\widehat{R}$ to each of $\widehat{S}$'s elements. (The use of id here is needed, since $R \cup$ is illegal.)

Although our notation is standard in both mathematics and functional languages, it is in many cases cumbersome, and the database area uses different, often simpler notations. Let

[15] An alternative approach is to use $[\,]$ as a higher-order operation that creates tuples of functions, and regard non-unary functions as unary functions with tuple inputs. Then all functions would be unary, and $\circ$ would suffice. Tuple construction would then appear under construction, rather than under composition.

[16] More generally, we could use $\{\,\}$ and $\{[\,]\}$, the two values of type $\{[\,]\}$, as representing the two truth value, as is done in [17].

us consider some examples. The expression $[\,]_{C,D}$ denotes a binary function; composing it with the two unary functions A and B, we obtain $[\,]_{C,D}(A,B)$. When applied to a tuple with A, B attributes, it generates a tuple with attributes C, D, where the A attribute of the input is the value of the new C attribute, and similarly for B and D. This is essentially a projection with renaming. A more convenient notation for such an expression is $[C = A, D = B]$. That is, a tuple constructor with a list of component names, and a function to define each of them (in the style of target lists of relational languages). Applying this expression to $[A\colon a, B\colon b]$, we obtain $[C\colon a, D\colon b]$. When the new attribute and the function name are the same, an even more compact notation can be used: Instead of $[A = A, \ldots]$, we simply write $[A, \ldots]$. The expression $\rho\langle\,[A,B]\,\rangle\,(\widehat{R})$, denoting the classical projection of $\widehat{R}$ on the attributes A and B, is thus an abbreviation for $\rho\langle\,[\,]_{A,B}(A,B)\,\rangle\,(\widehat{R})$. Similar notation is used with the set constructor. Finally, instead of $A(f)$ we use dot notation: $f.A$.

The definitions of functions and a-queries are now (almost) complete. As we have remarked, not every expression $\rho\langle\, f\rangle\,(\widehat{R})$ is a legal query. Type compatibility of f and $\widehat{R}$ is required, and there is an additional condition, considered below. We first present a few examples of a-queries, starting with those presented for the calculus in Example 4.1

Example 5.1 Recall the schema $\langle\ \widehat{R} : \{[A, A']\},\quad \widehat{S} : \{[B, B'\colon \{\}]\}\ \rangle$ of Example 4.1. The queries are:

1. The union of $\widehat{R}$ and a set of two constant tuples:
$$\widehat{R} \cup \{[A\colon 3, A'\colon 5], [A\colon 0, A'\colon 0]\}.$$
2. Select from $\widehat{S}$ the tuples where the first component is a member of the second component:
$$\sigma\langle\, B \in B'\,\rangle\,(\widehat{S}).$$
3. The (classical) cross product of $\widehat{R}$ and $\widehat{S}$:
$$\rho\langle\,[C.A, C.A', D.B, D.B']\,\rangle\,(\,(cross_{[C,D]}(\widehat{R}, \widehat{S})\,).$$
4. The join of $\widehat{R}$ and $\widehat{S}$, on $A = B$. This is easily expressed as a composition:
$$\sigma\langle\, A = B\,\rangle\,(\,\rho\langle\,[A = C.A, A' = C.A', B = D.B, B' = D.B']\,\rangle\,((cross_{[C,D]}(\widehat{R}, \widehat{S})\,)\,).$$

5. Flatten $\widehat{S}$, that is produce a set of flat tuples, each of which contains the first component of a tuple of $\widehat{S}$, and one of the elements of the second component:

$$\textit{set-collapse}(\ \rho\langle\ \textit{cross}_{[B,C]}(\{B\}, B')\ \rangle\ (\widehat{S})\ ).$$

   In this expression, the *replace* transforms each tuple of $\widehat{S}$ into a set of pairs. (Note that *cross* operates on sets, hence we need to transform B to $\{B\}$.) The first component is called B and its value is the B-value of the given tuple. The second component, called C, has a value that is a member of the set B' in the original tuple. This has the effect of pushing B into B', and making each element of B' a pair. The *set-collapse* is needed to remove the extra set brackets from the result.

6. The powerset of the relation $\widehat{R}$: $\textit{powerset}(\widehat{R})$.

7. The collection of subsets of the second component of tuples of $\widehat{S}$, which do not contain the values 2,4, or 5:

$$\textit{set-collapse}(\ \rho\langle\ \textit{powerset}(B' - \{2,4,5\})\ \rangle\ (\widehat{S})\ ).$$

The next examples illustrate complex queries and, in particular, the use of functions in *replace* and *select* expressions for the manipulation of deeply nested components of complex values.

Example 5.2 1. Recall the relation $\widehat{S} : \{[B, B' : \{\}]\}$ of Example 4.1. The relation of type $\{[B' : \{\}]\}$ obtained by adding the value of the B component to the B' component and deleting the B component is given by:

$$\rho\langle\ B' \cup \{B\}\ \rangle\ (\widehat{S}).$$

2. We present a more complex query, in which a construction by stages is helpful. Let $\widehat{R} : \{\ [A, B : \{\ \{\ [C, D]\ \}\ \}\ ]\ \}$ be a scheme. The query is to add a third attribute E, with the value 5, to each of the tuples in each member of B and to eliminate the A-component.

   The replace operation for a set X of type $\{[C, D]\}$ is

$$\rho_1 = \rho\langle\ [C = C, D = D, E = 5]\ \rangle\ .$$

Or, using our abbreviations,

$$\rho_1 = \rho\langle\, [C, D, E = 5]\,\rangle\,.$$

Now for Y of type $\{\{[C, D]\}\}$, we want to perform this restructuring on each set member of Y, so we use

$$\rho_2 = \rho\langle\, \rho\langle\, [C, D, E = 5]\,\rangle\,(id)\,\rangle.$$

Observe the necessary use of the *id* function here. Since the B component of $\widehat{R}$ has this type, this is a valid replace specification for it. To obtain a replace for $\widehat{R}$ we have to add one more level:

$$\rho_3 = replace\langle\, \rho\langle\, \rho\langle\, [C, D, E = 5]\,\rangle\,(id)\,\rangle\,(B)\,\rangle(\widehat{R}).$$

Now, suppose that instead of 5 we want to use the A-component. This is done using:

$$\rho_3 = replace\langle\, \rho\langle\, \rho\langle\, [C, D, E = A]\,\rangle\,(id)\rangle\,(B)\rangle\,(\widehat{R}).$$

Observe that in this last expression C, D apply to tuples at the inner level, where such attributes indeed exist. A is also used at the inner *replace*, but its meaning is the A of the outer level tuples. In particular, observe that in the inner replace specification, $[C, D, E = A]$, A has no meaning in the strict context of the type $\{[C, D]\}$. Essentially, in this expression A is *free*. It remains free on the next level, and it becomes bound (i.e., is given a meaning) only in the larger context of the complete query.

By the definitions of a-queries and functions, if f is a replace specification and Q a query, then $\rho\langle\, f\,\rangle\,(Q)$ is also a query, and if g is another replace specification, then $\rho\langle\, f\,\rangle\,(g)$ is also a replace specification. Both forms use the *replace* operation $\rho\langle\, f\,\rangle$. So far, we have not placed any restrictions on these constructions. As we have seen in the example, in the replace specification $\rho\langle\, f\,\rangle\,(g)$, it makes sense to allow the use in f of an attribute that does not appear in the operand g; such an attribute presumably is given a meaning when the replace specification is used in a larger context. But, it does not make sense to do so in an a-query $\rho\langle\, f\,\rangle\,(Q)$. For example, if $\widehat{R}$ is not of a set-of-tuple type, or if it is of such a type, but A is not used in the tuple type, then what could be the meaning of $\rho\langle\, A\,\rangle\,(\widehat{R})$ as a query? A related problem is the following: suppose A is used in f, and we have $\rho\langle f \rangle(\widehat{R})$, where $\widehat{R}$ is a tuple type that contains A. Does A necessarily refer to the A component of $\widehat{R}$? Assume $f = \rho\langle A \rangle(\widehat{S})$, and $\widehat{S}$ also has A as an attribute. Then presumably A refers to the A function on elements of $\widehat{S}$. Thus, we have here an issue of bindings of names, according to scopes where the names are known. This intuition is now formalized.

For each function expression, we divide the set of attributes that are used in it as functions[17] to *bound* and *free*. The definition uses induction on the structure of function expressions. In the expression A, A is free. The other basic functions do not include attributes. A function obtained by composition has the form $f(f_1, \ldots, f_k)$, where f is a basic function or an algebraic operation, and the f_i's are functional expressions. The free and bound attributes of each of the f_i remain free and bound, respectively, in this expression. If f does not contain attributes, then we know all the free and bound attributes of the expression. There are two cases to consider when f contains attributes: Either f is an attribute, so the expression has the form $f_1.A$ (i.e., $A(f_1)$). For this expression to be legal, f_1 must have a tuple type with an A component. Hence, A is bound in this expression. The second case is the application of $\rho\langle f \rangle$ to g, to obtain $\rho\langle f \rangle(g)$. (Selection is treated similarly.) The bound and free attributes of g remain bound and free, respectively, in this expression, and the bound attributes of f also remain bound. The free attributes of f remain free except that if g is a set of tuples of type T, and A is an attribute of T, and A was free in f, then it now becomes bound. The two cases can be summarized as follows: when a function/operation is applied to an argument, and a free attribute in it is meaningful in the argument, that attribute becomes bound. The last case to consider is function construction. Let f be a function expression, and let $\rho\langle f \rangle$ be a function expression constructed form it. Then the bound/free attributes in f remain bound/free in this expression.

In summary, binding of function attributes to their meaning is done from the inside out; an attribute is bound in the smallest subexpression where a meaning for it exists. Note that in some cases the user's intention may be different. For example, in the expression $\rho\langle \rho\langle \ldots A \ldots \rangle(\widehat{S}) \rangle(\widehat{R})$, if both relations contain A, the user might want to refer to the A-component of $\widehat{R}$. To do that, it is necessary first to rename the attribute A in $\widehat{S}$ to a new attribute (see below how to rename), then construct the required expression. Similar problems may arise in the use of the identity function, and solving them may complicate

[17]Note: in $[A_1 : f_1, \ldots, A_n : f_n]$, the attributes $A_1, \ldots, A_n$ are *not* used as functions!

the expression of queries.

Now, we summarize the conditions in terms of bound and free attributes on the construction of replace specifications and a-queries that use *replace*. There is no restriction on replace specifications. But, an expression $\rho\langle\, f \,\rangle\,(Q)$ is a query only if any free attributes of f (if any) must appear in the type of the members of Q, which must be a tuple.

From the examples and the discussion above it follows that in an a-query $\rho\langle\, f \,\rangle\,(Q)$, if f contains a *replace* operation (i.e., the query has a nested *replace*), then this nested *replace* may contain free attributes. Such attributes necessarily are given meaning in the context of (the type of) Q, since they cannot remain free. This was illustrated in the last example. Is it possible to characterize the attributes that can thus be used in a nested *replace*? The answer was presented in [44]. Let T be the tree representing some set type, and let x be a node in its tree that is also of a set type. The idea is that an attribute A can be used in a replace specification on x if it either appears in the type of x, or if it appears somewhere else in T. But, in the second case we must have that, given a value v_T of type T, and a node (i.e. a value) v_x of type x in it, A can be interpreted as denoting a unique node of v_T relative for this given v_x. In both cases, A plays essentially the same role as a constant, relative to a value v_x. The set of such attributes is those attributes in the nodes that can be reached from v by going up, and possibly also sideways and down, without crossing a set constructor on the way down. If x has a tuple of tuples type, then it also includes those attributes reachable from v by going down, again without crossing a set constructor. This is captured in the following definition.

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

- If T' is a tuple type, $T' = [A_1 : T_1, \ldots, A_n : T_n]$, then $dyn_T(T') = dyn_T(T_1) = \ldots dyn_T(T_n)$, and for each i, A_i is in $dyn_T(T')$;
- If T' is a set type $T' = \{T_1\}$, then $dyn_T(T') \subseteq dyn_T(T_1)$.

For the relation $\widehat{R}$ of Example 5.2, the dynamic constants of the inner tuple node inside B are C, D, B, A. These are the attributes that can be used in a replace specification for this node in an a-query on $\widehat{R}$.

We have illustrated bottom-up construction of queries. The next example illustrates top-down construction of an a-query, containing replace specifications with dynamic constants.

Example 5.3 Consider the relation scheme $\widehat{R} : \{ [A:\{\}, B:\{ [C, D:\{\}, E] \}] \}$. We want to restructure the elements of $\widehat{R}$, by leaving only B, itself transformed by dropping the C component from each of its tuples, and leaving in the D component only those elements that are in A. Furthermore, we want to delete from B those tuples in which the new D component does not contain 0 and the value in the E component.

We start with B as a replace specification for $\widehat{R}$. This is transformed to $\rho\langle [D, E] \rangle (B)$. Next, we replace D by $\sigma\langle id \in A \rangle (D)$, to obtain the replace specification $\rho\langle [\sigma\langle id \in A \rangle (D), E] \rangle (B)$. We now add a selection on the condition $\{0, E\} \subseteq D$, and apply the resulting expression as a replace specification to $\widehat{R}$ to obtain:

$$\rho\langle \sigma\langle \{0, E\} \subseteq D \rangle (\rho\langle [\sigma\langle id \in A \rangle (D), E] \rangle (B)) \rangle (\widehat{R}).$$

To conclude, we state some properties of a-queries and function expressions.

Proposition 5.1 *Every a-query is a also a function expression. (Note that the value of this function does not depend on the implicit input.)*

Proof: As we have already remarked, an a-query q is interpreted in any given database as denoting a constant. It can be viewed as the unary constant function $\lambda x.q$. The proof uses induction on the structure of a-queries. Details are left to the reader. □

An a-query containing relation names $\widehat{R}_1, \ldots, \widehat{R}_n$ defines a function on databases, in which these relation names are treated as variables. We can compose a-queries, by replacing a relation name in a query by another query. Similarly, if a replace specification contains relation names, we may replace some of them by replace specifications of the same types.

Proposition 5.2 *(i) The class of a-queries is closed under composition. That is, if E is an a-query that contains $\widehat{R}$, and E' is another a-query whose output type is the type of $\widehat{R}$, then the expression obtained from E by replacing an occurrence of $\widehat{R}$ in it by E' is an a-query.*
(ii) The class of replace specifications is closed under replacement of relation names by replace specifications. That is, if f contains $\widehat{R}$, and g is another replace specification of the same output type as $\widehat{R}$, then the expression obtained by replacing an occurrence of $\widehat{R}$ in f by g is a replace specification.

Proof: (i) The proof uses induction on the structure of E, for each fixed E'. The case that deserves attention is when $E = \rho\langle f \rangle (Q)$, and an occurrence of $\widehat{R}$ in f is replaced by E'. For this case we use (ii).
(ii) This is proved in a similar manner. Details are left to the reader. Note that the closure of replace specifications under composition is part of their definition. □

We conclude this subsection with the following observation.

Theorem 5.3 *The algebra is a domain independent language.* □

5.3 Variations

We now make some observations about the expressive power of the algebra. In particular, we consider additional operations which can be expressed in it. We also consider whether certain operations can be removed, or restricted, without changing the expressive power. Naturally, we are most interested in what can be done using *replace* with nested expressions, since it is the nesting of structures in the model and of expressions in the language, that distinguishes our model and language from the relational analogues.

What kinds of restructuring can we perform on a relation? We can increase the level of nesting by adding set or tuple constructors: using the set constructor, we can transform each element into a singleton set; using the tuple constructor, we can transform each element into a single-component tuple, with any attribute as a label[18]. In the other direction, that of decreasing the level of nesting, we have several cases to consider. First, if we have a set of sets of T-values, we can use *set-collapse* to transform it into a set of T-values. In the special case that each of the sets is a singleton, this has the same effect as the operation *the* used in the query language of O_2 [11]. Similarly, given a tuple, we can select one of its components, drop the others, and leave this component without the tuple constructor. For example, if we have $\widehat{R} : \{[A{:}\,T, \ldots]\}$, then $\rho\langle A \rangle (\widehat{R})$ has the type $\{T\}$. Another related operation is the following: if we have a set of tuples, where one of the components is a tuple, we can transform it into a set of flat tuples. For example, if we have $\widehat{R} : \{ [A, B{:}[C, D]] \}$, then $\rho\langle [A, C, D] \rangle (\widehat{R})$ has the type $\{[A, C, D]\}$. This derived operation is called *tuple-collapse*. It is an analogue of *set-collapse*. Using this operation, it is easy to obtain the classical cross product of two sets of tuples: we perform a *cross*,

[18]When no label is desired, this is an identity transformation.

followed by a *tuple-collapse*. From that we obtain the various joins by composing with selections.

Assume we have $\widehat{R} : \{[A, \ldots]\}$. We can change the name of the first component to A' by $\rho\langle\, [A' = A, \ldots]\,\rangle\,(\widehat{R})$. Similarly, we can change any attribute in a substructure by using nested *replace*. These expressions can be quite cumbersome, since one has to repeat all the attributes that need not be changed. Since renaming is a useful operation, it is useful in practice to add an operation *rename* for this purpose. We use the notation $rename_{A \to A'}(\widehat{R})$ for changing the attribute A to A' uniformly in $\widehat{R}$.

From the previous discussion, it follows that we can express all the relational algebra operations. Let us now consider the nested relational model.

Two well known operations of that model are the *nest* and *unnest* [32]. Assume we have the scheme $\widehat{R} : \{[A, B, C]\}$. Nesting on the attribute C is accomplished as follows. First, we delete from each tuple of $\widehat{R}$ the C-component, and extend it with a copy of $\widehat{R}$:

$$\widehat{R}_1 = \rho\langle\, [A, B, D = rename_{A \to A', B \to B'}(\widehat{R})]\,\rangle\,(\widehat{R}).$$

The first two attributes in the copy of $\widehat{R}$ are renamed to prevent ambiguity in the expression below. Now, we perform a selection on the embedded copies, so that in a tuple $[a, b, d]$ the d component will contain only the tuples that have a and b in the first two positions. Then we project the resulting d on its C attribute. This is accomplished by

$$\widehat{R}_2 = \rho\langle\, [A, B, \rho\langle\, [C]\,\rangle\,(\sigma\langle\, A = A' \wedge B = B'\,\rangle\,(D)\,)\,]\,\rangle\,(\widehat{R}_1).$$

These two expressions can of course be collected into a single algebraic expression. As a a matter of fact, the projection and selection can be pushed inside, so the final expression is

$$\rho\langle\, [A, B, D = \rho\langle\, [C]\,\rangle\,(\sigma\langle\, A = A' \wedge B = B'\,\rangle\,(rename_{A \to A', B \to B'}(\widehat{R})\,)\,)\,]\,\rangle\,(\widehat{R}).$$

This expression is a good description of how this nest is often computed in practice.

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

$$set\text{-}collapse(\rho\langle\, \rho\langle\, [A, B, D]\,\rangle\,(C)\,\rangle\,(\widehat{R}).$$

We have stated that all the operations of relational or nested relational algebra can be expressed in our algebra. The reason why some of these are not basic, but rather derived, operations, and that some of the expressions are non-trivial is that we have un-bundled the *set* and the *tuple* constructors, and our operations and basic functions treat each one separately. In the relational and nested algebras, the operations deal with the type constructor *set-of-tuple*.

Many algebras for nested relations in the literature do not include a facility such as *replace* nesting, that is, for applying algebraic operations to substructures, like the one we have presented following [44]. Is this facility essential? We argue informally that, theoretically, it is not. First, we show that nested *replace*, that is a *replace* within another *replace* can be eliminated. Consider $\widehat{R} : \{ [A, B, C \colon \{ [D, E] \}] \}$, and a query $\rho\langle [\ldots, \rho\langle f \rangle (C)] \rangle (\widehat{R})$. An equivalent way to express it is as follows. Extend $\widehat{R}$ with an attribute C' that is a copy of C, then unnest on C', calling the new attributes, say D', E'. Then perform a *replace* where the function f is applied to D', E'. Then nest the result and project C out. Note that since we have kept in the unnest all the original attributes, the subsequent nest reconstructs the original relation (except that the copy C' of C has been replaced by $\rho\langle f \rangle (C')$, as required).

We still have that inside a *replace* all algebraic operations, except a *replace*, can be used. Consider the relation $\widehat{R} : \{ [A, B : \{ \}, C : \{ \}] \}$, and a query $\rho\langle [\ldots, op(B, C)] \rangle (\widehat{R})$, where *op* is an algebraic binary operation. We consider *cross* as a concrete example. An equivalent expression is obtained thus: project the relation on B, and perform *set-collapse*, to have the union of the B-values. Do the same for the C-values. Now perform *cross* on the results and extend each tuple of $\widehat{R}$ with the result. Call the new attribute D. The problem is that the D-value in a tuple contains in addition to the pairs in $cross(B, C)$ many other pairs. But, a simple selection eliminates these extra pairs. Similar arguments apply to other operations.

In summary, we could define our algebra so that in the construction of replace specifications, algebraic operations cannot be used. The analogy to the calculus would then be complete. However, note that in eliminating nesting, we are forced to express functions, that can be expressed naturally with nesting, in a very complicated manner. In a practical language for a model with nested structures, nesting is an extremely useful tool.

In [1], the language contains operators which are defined recursively on Verso relations. For example, assume that E and F are two instances of

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

Recall that by definition of Verso instances, $\{A, B\}$ and $\{D\}$ are keys at their respective levels. We can define union of instances of type $\{[F, G]\}$. Using it, we can define the union of two instances I, J of type $\{[D, E : \{[F, G]\}]\}$:

$$I \cup J = \{u \mid \exists v \in I, w \in J, u.D = v.D = w.D, u.E = v.E \cup w.E\}.$$

More generally, one can define recursively union of Verso instances.

In this operation, it seems that adding levels of nesting to the relations makes the expressions more complicated (in the algebra presented here). We conjecture that no single expression of our algebra can express the Verso union (and some of their other operations) for arbitrary relations.

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. As we have seen, for each of the two language paradigms, there is a variety of possible dialects. The details of the proof vary with the choice of a dialect for each paradigm. We present the proof for the languages presented in Sections 4.1 and 5.1, respectively.

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]. The modifications have to do mainly with the richer structure of terms. 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 or in the query. Given that, we show how to construct for each c-query an a-query that generates the same answer on a database in which the domains are the so-called active domains. The claim then follows.

Given a database instance $DB = \langle\, [D_1, \ldots, D_k], [R_1, \ldots, R_n]\, \rangle$, denote by $adom(DB, \widehat{D_i})$ the set of elements of D_i that appear in DB (including those that appear as components of complex values[19]), and by $adom(DB)$ the vector of sets $adom(DB, \widehat{D_1}), \ldots, adom(DB, \widehat{D_k})$. These are the atomic values that appear in the database. A given query q may include constants that do not appear in the database. Hence we extend the notation and use $adom(DB, q, \widehat{D_i}), adom(DB, q)$ to denote the appropriate set, or vector of sets, of atomic values that appear in the database or in the query. In the discussion below, we assume q is given. We also assume that the database schema $\widehat{DB}$ is fixed.

Claim 6.2 *For each domain name $\widehat{D}$ occurring in $\widehat{DB}$, there exists an a-query, denoted $F_{\widehat{D}}$, such that for every database DB, $F_{\widehat{D}}(DB) = adom(DB, \widehat{D})$.*

Proof: For each $\widehat{R}_i$, we construct an a-query F_i, such that $F_i(DB)$ is the set of elements of D, the domain in the database for $\widehat{D}$, that appear in R_i. Then $F_{\widehat{D}} \equiv F_1 \cup \ldots \cup F_n$. The a-query F_i is constructed by applying the following recursive procedure:

- If $\widehat{R}_i$ is a set of atoms of type $\widehat{D}$, then $F_i \equiv \widehat{R}_i$. If $\widehat{R}_i$ is of of atoms from another atomic domain, then $F_i \equiv \emptyset$.
- If $\widehat{R}_i$ is of type $\{\{T\}\}$, for any T, then apply the procedure to *set-collapse*$(\widehat{R}_i)$.
- If $\widehat{R}_i$ is of a tuple type $\{[A_1 : T_1, \ldots, A_m : T_m]\}$, then apply the procedure to each of the expressions $\rho\langle A_j\rangle(\widehat{R}_i)$. Then take the union of the a-queries that were generated.

Note that in the last two cases the procedure is applied to simpler types, hence termination is guaranteed. Also note that if $\widehat{D}$ is not used in $\widehat{R}_i$, or in an expression generated in the last two cases, the procedure can be terminated immediately, and the value returned is $\emptyset$. $\Box$

Since for each finite set of constants there is an a-query that returns it on every database, we can add the constants of the given query to the a-queries of the claim.

Corollary 6.3 *For each query q, and for each domain name $\widehat{D}$, there exists an a-query $F_{\widehat{D},q}$, such that $F_{\widehat{D},q}(DB) = adom(DB, q, \widehat{D})$.* $\Box$

[19] *adom* stands for *active domain*.

Recall from Section 2 that for a type T and a vector $\bar{D} = D_1, \ldots, D_k$ of sets of atomic values, corresponding to the domain names $\widehat{D}_1, \ldots, \widehat{D}_k$, $DOM(T, \bar{D})$ is the set of values of type T that can be constructed from the atomic values in $\bar{D}$. In particular, $DOM(T, adom(DB, q))$ is the set of values of type T that can be constructed from atomic values that appear in DB or in q.

Claim 6.4 *For every type T, there exists an a-query $F_{T,q}$, such that for every database DB, $F_{T,q}(DB) = DOM(T, adom(DB, q))$.*

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

Basis: If T is an atomic type, $\widehat{D}$, then $F_{T,q} \equiv F_{\widehat{D},q}$.

Induction: There are two cases to consider.

- If $T = \{S\}$, then $F_{T,q} \equiv powerset(F_{S,q})$.
- If $T = [A_1 : T_1, \ldots, A_n : T_n]$, then $F_{T,q} \equiv cross_{[A_1,\ldots,A_n]}(F_{T_1,q}, \ldots, F_{T_n,q})$.

□

We have now finished the first stage in the proof, namely showing that the *active domain* of each type can be expressed by an a-query. It follows that for each variable x, there is an a-query, which we denote by E_x^q, that computes the set of values, constructed from atomic values that appear in the database or in q that this variable can take. Indeed, this query is simply $F_{T,q}$, where T is the type of x. Note the use of the *set-collapse* and *powerset* operations in this proof. This is the only place in the translation from calculus to algebra where these operations are used.

We now state and prove the main claim of this subsection, from which this part of the theorem follows. For a given c-query as above, and for a database DB, denote by DB_q the database that has the same relations as DB, but in which the domain for $\widehat{D}_i$ is $adom(DB, q, \widehat{D}_i)$. That each, the database relations are preserved, but the domains are the smallest domains that contain the values used in the database or in the query. From the results above, we have that for each variable x of type T, and for each database DB,

$$E_x^q(DB) = DOM(T, adom(DB, q)) = E_x^q(DB_q).$$

That is, the a-query E^q_x computes on DB_q precisely the set of values on which the variable x ranges in this database, namely the domain of the type of x in this database.

We next show that the algebra is as expressive as the domain independent calculus.

Proposition 6.5 *Let q be a domain independent c-query. Then there exists an a-query E_q, such that for every database DB,*

$$E_q(DB) = E_q(DB_q) = q(DB_q) = q(DB).$$

Proof: It suffices to show that for some E_q, $E_q(DB_q) = q(DB_q)$ (for $q(DB) = q(DB_q)$ since q is domain independent and $E_q(DB) = E_q(DB_q)$ since the algebra is domain independent).

Let φ be the formula in q. The proof is by induction on the structure of φ. As explained in Section 4, one can associate a c-query with every formula, with any number of variables, including 0. We refer to the c-queries that correspond to subformulas of φ, as *subqueries* of q. We construct a-queries for all subqueries of φ, using structural induction.

Let us consider the structure of atomic formulas. They may be of the forms $\widehat{R}_i(t)$, or $t_1\theta t_2$, where t, t_1, t_2 are terms, and θ is one of the comparators $=, \in, \subseteq$. We note that for each term there is a set of values that it can take in the database DB_q. Before we can deal with the atomic formulas, we need to show the existence of a-queries that compute this set of values for each term. We already know that such a-queries exist for variables, and they surely exist for constants. The existence for arbitrary terms can therefore be proven by induction on the structure of terms. However, we actually need to prove a slightly stronger claim. If we consider an atomic formula as a query, then the target list of this query is not the term or terms in the formula, but rather the list of variables in the term(s). When the answer is computed, whenever the term(s) satisfy the formula, the assignment to the variables is output, rather than the value of the term(s). Hence, what we really need is to construct an a-query that computes the set of possible assignments for the variables in the term, and for each of them the value of the term(s). Then, when we find that the formula is satisfied by an assignment, we can use the assignment for the output.

Claim 6.6 *(i) Let t be a term in q with l variables $x_1, \ldots, x_l$ (l may be 0); then there exists an a-query G^q_t such that $G^q_t(DB_q)$ is a set of $(l+1)$-tuples, the first l of which are* *all combinations of values from* $E^q_{x_i}(DB_q), i = 1, \ldots, l$ *(that is, possible assignments to the variables in the database* DB_q*), and the last component in each tuple is the value of* t *for the assignment represented by the first* l *components.*

(ii) Let t_1, t_2 *be two terms in* q*, with* l *variables* $x_1, \ldots, x_l$ *(*l *may be* 0*); then there exists an a-query* $G^q_{t_1,t_2}$ *such that* $G^q_{t_1,t_2}(DB_q)$ *is a set of* $(l+2)$*-tuples, the first* l *of which represent assignments to the variables as above, and the last two are the values of the two terms for the assignments.*

Proof: (i) The proof uses induction on the structure of terms. If t is a variable x, then we have E^q_x, that computes the domain for x in DB_q. To obtain G^q_x, we take

$$\rho\langle\, [1 = id, 2 = id] \,\rangle E^q_x.$$

If t is a constant c, then $l = 0$ and G^q_t is simply $\{c\}$. Note that this works for non-atomic constants and relation names as well, so if t contains no variables, the construction for t is complete.

If t contains variables, and is not a simple variable, then it is obtained by applying a function to some terms. We consider each of the functions that can be applied. The first case is that $t = t_1.A$, where t_1 is of a tuple type on which A is defined. By induction hypothesis, we have $G^q_{t_1}$. The l variables of t are those of t_1. Hence

$$G^q_t \equiv \rho\langle\, [1, \ldots, l, l+1 = (l+1).A] \,\rangle\, (G^q_{t_1}).$$

The second case is when $t = [\,]_{A_1,\ldots,A_n}(t_1, \ldots, t_n)$, or using our alternative notation, $t = [A_1{:}\,t_1, \ldots, A_n{:}\,t_n]$. Here the set of variables of t is the union of the sets of variables of the t_i's. Hence

$$G^q_t \equiv \rho\langle\, [1, \ldots, l, l+1 = [\,]_{A_1,\ldots,A_n}(l+1, \ldots, l+n) \,\rangle\, (H^q_t),$$

where

$$H^q_t \equiv \textit{permute}(\textit{join on common variables}(\textit{cross}(E^q_{t_1}, \ldots, E^q_{t_n}))).$$

In this expression, each $E^q_{t_i}$ computes tuples of length $l_i + 1$ of which the first l_i are variables values. In the cross product, a variable that appears in more than one t_i has more than one components corresponding to it. The *join* contains a selection that forces, for each variable, all its values to be the same, followed by a projection that leaves just one copy of the value. The cross product also has one position for each of the terms.

After the join, the *permute*[20] moves these to the positions $l+1,\ldots,l+n$; then the tuple constructor in the final *replace* collects them into a tuple, as required.

The last case is that t is obtained by application of the set constructor. The construction of G_t^q for this case is similar, and is omitted.

(ii) This case is treated essentially as in the application of a constructor in the previous case. Given $G_{t_i}^q$, for $i=1,2$, we perform a natural join on the common attributes, and move the two terms to the last two positions. $\square$

We now consider atomic formulas. The first case is a formula $\widehat{R}(t)$. Assume t has l variables. We want to construct an a-query equivalent to $\{\, x_1,\ldots,x_l \mid \widehat{R}(t)\,\}$. We have, by the construction above an algebraic expression G_t^q that computes all possible values in the database for $[x_1,\ldots,x_l,t]$. The required a-query is

$$\rho\langle\,[1,\ldots,l]\,\rangle\,(\sigma\langle\,(l+1)\in\widehat{R}\,\rangle\,(G_t^q)).$$

Note that for the case $l=0$, the selection outputs an answer of type $\{[1\colon]\}$, that is a set of tuples of arity one. The result of the *replace* is then either $\{[\,]\}$ if some instantiation of t is in R, or $\emptyset$ otherwise.

The other case for atomic formulas is $t_1\theta t_2$, where θ is one of $=,\in,\subseteq$. The a-query is

$$\rho\langle\,[1,\ldots,l]\,\rangle\,(\sigma\langle\,(l+1)\theta(l+2)\,\rangle\,(G_{t_1,t_2}^q)).$$

The remark above about the case $l=0$ holds here also. For example, if the query is $\widehat{R}(x.A)$, then the formula is $\rho\langle\,1\,\rangle\,(\sigma\langle\,2\in\widehat{R}\,\rangle\,(\rho\langle\,[1:id,2:A]\,\rangle\,(E_x^q)))$, and if the formula is $a\in\widehat{R}$ (for some element a), then the formula is $\rho\langle\,[\,]\,\rangle\,(\sigma\langle\,1\in\widehat{R}\,\rangle\,(\,\{\,[a]\,\}\,))$.

It is easy to see that when a subformula ψ of φ is atomic, if E_ψ denotes the formula we have constructed for it, then

$$E_\psi(DB_q)=\psi(DB_q).$$

Now we consider induction on the structure of formulas. It suffices to consider the cases where a formula is obtained from simpler formulas by applying $\wedge,\neg$, or $\exists$. The proof is essentially as in [46]. We use natural join for $\wedge$. For negation, we take the complement

[20]Of course, for any given t, instead of doing this permutation, we could use the proper column numbers in the final *replace*.

with respect to the cross product of the E_x^q's, for those x's that appear as free variables in the formula being negated[21]. Finally, we use projection for the existential quantifier.

For each of the three cases, we assume by induction hypothesis that the a-queries equivalent to the c-queries that correspond to the subformulas of the given formula are given, and we prove that such an a-query exists also for the formula. The only nontrivial step is the association of projection with the existential quantifier. However, for the active-domain database DB_q, the range of an existentially quantified variable x is $E_x^q(DB_q)$, and it follows that projection indeed has the same effect as the existential quantifier.

When the construction described above terminates, we have for the given formula $\varphi(x)$ an a-query, E_φ, such that $E_\varphi(DB_q) = \varphi(DB_q)$. This is the required E_q. $\square$

This concludes the proof of one direction of the equivalence theorem.

The following observations are of interest. In the proof, *set-collapse* is used only in the construction of the queries for the active atomic domains. If $T = \{S\}$ is a set type, 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 non-atomic types and variables, and this is the only place it is used in the proof. This seems to imply that these operators are auxiliary, and one could do away with them, if another approach to computing the domains is found. The *powerset*, in particular, is an expensive operator that does not seem to be really necessary in a query language. This issue is considered in the sequel.

Note also that nesting of operations in *replace* operations is not used in the main part of the proof. Essentially, the operations of the relational algebra combined with rather weak restructuring facilities, and with the *powerset* and *set-collapse* are sufficient to provide the algebra with expressive power comparable to that of the calculus. This is another demonstration that nesting is not crucial for obtaining the required expressive power. But, the idea here is different from that used in Section 5. There, we showed that given a description of a query as a function on databases, we can express the function in another, although rather convoluted, way, that does not require nesting. By "expressing a query as a function" we mean that the query actually describes how the components of the answer are computed from those of the input. The alternative formulation simulates this "procedural" approach. In the calculus, this procedural component is much weaker. The

[21]If there are no free variables, we take the complement with respect to $\{[\,]\}$.

existence of typed variables allows one to specify the *form* of the result without using the input in any way; the actual results is then related to the input by a set of conditions. Therefore, the algebraic simulation constructs the domains of certain types, including the type of the result, then simulates the conditions that restrict the domains, until the result only is left. Quite obviously, although this proves the equivalence with the calculus, it is not the direction to go in designing practical languages.

6.2 From Algebra To Calculus

Although we could, in principle, assume that no algebraic operations are used inside *replace*, we deal in the proof with both a-queries and with arbitrary replace specifications. It suffices to prove the following.

Proposition 6.7 (q) *For every a-query E, there exists an equivalent (domain independent) c-query q_E, with a single free variable in its formula.*

(r-s) *For every replace specification f, there exists a formula $\psi_f(u,v)$ that represents it. That is, u, v have the types of f's input and output, and $\psi_f(u,v)$ holds iff $v = f(u)$. Also, for each select specification (i.e., unary predicate) p, there exists a formula $\psi_p(u)$ that represents it, that is, u has the type of p's input, and $\psi_p(u)$ holds iff $p(u) =$ TRUE.*

Proof: (Remark: (q) is the claim we need, (r-s) is needed for the induction step of the proof.) The claims are proved simultaneously, using induction on the structure of algebraic expressions.

Basis:

(q) The a-queries $\{c\}$ and $\widehat{R}$ are represented by the c-queries $\{\, x \mid x = c \,\}$, and $\{\, x \mid \widehat{R}(x) \,\}$, respectively.

(r-s) The replace specifications $c, \widehat{R}$, are represented by the formulas $v = c$ and $\forall y(y \in v \leftrightarrow y \in \widehat{R})$, respectively. (Note that since these functions ignore their input, the variable u is missing. One can of course add $u = u$ to them.) This representation works for non-atomic constants as well, and also for $[\,]$ and $\emptyset$. The function A is also present in the calculus. Its corresponding formula is $v = u.A$. As for *id*, it is represented by $v = u$. The tuple and set constructors, and the binary algebraic operations are not, by themselves, replace specifications, hence need not be considered now. We have two unary algebraic operations among the base replace specifications: *Set-collapse* is represented by $\forall x(x \in v \leftrightarrow \exists y(y \in u \wedge x \in y))$; *powerset* is represented by $\forall x(x \in v \leftrightarrow x \subseteq u)$. Finally, all six boolean functions are binary, so there are no base select specifications.

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 $\varphi_{E_1}, \varphi_{E_2}$ are the formulas in the corresponding c-queries, each with a single free variable[22]. The a-query E is constructed from E_1 and E_2. We show how to construct φ_E from $\varphi_{E_1}, \varphi_{E_2}$.

- **Set operations**: $E = E_1 \theta E_2$, where θ is one of $\cup, \cap, \setminus$. 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 $\varphi_E \equiv \varphi_{E_1} \vee \varphi_{E_2}$.
  - if $E \equiv E_1 \cap E_2$, then $\varphi_E \equiv \varphi_{E_1} \wedge \varphi_{E_2}$.
  - if $E \equiv E_1 \setminus E_2$, then $\varphi_E \equiv \varphi_{E_1} \wedge \neg\varphi_{E_2}$.
- **Cross product**: $E \equiv \mathit{cross}_{[A_1,A_2]}(E_1, E_2)$. We change the two given formulas so that their free variables are different. Then
$$\varphi_E \equiv \exists x_1 \exists x_2 \, (\varphi_{E_1}(x_1) \wedge \varphi_{E_2}(x_2) \wedge x = [A_1 : x_1, A_2 : x_2]\,).$$
- **Powerset**: $E \equiv \mathit{powerset}(E_1)$. The formula φ_{E_1} has a single free variable of type T, say x_1. Let x be a new variable of type $\{T\}$. Then
$$\varphi_E \equiv \forall x_1 \, (x_1 \in x \rightarrow \varphi_E(x_1)).$$
(Note: we could write instead $x \subseteq \{x_1 \mid \varphi_E(x_1)\}$, then use the reduction described in Section 4.)
- **Set-collapse**: $E \equiv \mathit{set\text{-}collapse}(E_1)$, where φ_{E_1} has a single free variable of a set type. Then $\varphi_E \equiv \exists x_1(\varphi_{E_1}(x_1) \wedge x \in x_1)$.
- **Select**: $E \equiv \sigma\langle\, p \,\rangle\,(E_1)$. Assume that $\psi_p(u)$ represents p. Then
$$\varphi_E \equiv \varphi_{E_1}(x) \wedge \psi_p(x).$$

[22]Since we have tuple types, there is no real distinction between one or more variables; it is more convenient to present the proof for a single variable.

- **Replace**: $E \equiv \rho\langle f \rangle(E_1)$. Let $\psi_f(u, v)$ be the formula that represents the function f. Then
$$\varphi_E \equiv \exists u\, (\varphi_{E_1}(u) \wedge \psi_f(u, v)).$$

We consider next replace and select specifications.

(r-s) Replace specifications can be obtained by using either composition or construction. Select specifications can be obtained only by using composition. We first consider the use of composition. We treat both cases together, since they are both unary functions expressions. Since composition is associative, we may assume that when it is used to construct a new function, a function or operation is composed (on the left) with one or more, possibly complex, function expressions. Assume that $f, f_1, \ldots, f_m$ are unary function expressions, represented by $\psi_f, \psi_{f_1}, \ldots, \psi_{f_m}$, that p is a unary predicate, represented by ψ_p and that g is a function, not necessarily unary. We consider how to represent the result of "applying" g to (i.e., composing it on the left with) some of these functions. We consider the possible cases of g.

Base functions:

- **The attributes, the identity:** The function $f.A$ is represented by $\exists x\, (\psi_f(u, x) \wedge v = x.A)$. The function $id(f)$ is represented by ψ_f, since $id(f) = f$.

- **tuple constructor**: The function $[A_1{:}\, f_1, \ldots, A_m{:}\, f_m]$ is represented by
$$\exists v_1 \ldots \exists v_m\, (\psi_{f_1}(u, v_1) \wedge \ldots \wedge \psi_{f_m}(u, v_m) \wedge v = [A_1{:}\, v_1, \ldots, A_m{:}\, v_m]\,).$$

- **Set constructor**: $\{f_1, \ldots, f_m\}$ is represented by:
$$\exists v_1 \ldots \exists v_m\, (\psi_{f_1}(u, v_1) \wedge \ldots \wedge \psi_{f_m}(u, v_m) \wedge v = \{v_1, \ldots, v_m\}).$$

Algebraic operations:

- **Set operations**: We show only the union. Intersection and difference are treated similarly. The function $f_1 \cup f_2$ is represented by:
$$\exists v_1 \exists v_2\, (\psi_{f_1}(u, v_1) \wedge \psi_{f_2}(u, v_2) \wedge \forall x\, (x \in v \leftrightarrow (x \in v_1 \vee x \in v_2))).$$

- **Cross**: This is treated similarly to the union: For example, the function $cross_{[A_1,A_2]}(f_1, f_2)$ is represented by

$$\exists v_1 \exists v_2\, (\psi_{f_1}(u, v_1) \wedge \psi_{f_2}(u, v_2) \wedge v = [A_1 : v_1, A_2 : v_2]).$$

- **Powerset**: The function $powerset(f)$ is represented by

$$\exists v_1\, (\psi_f(u, v_1) \wedge \forall y\, (y \in v \leftrightarrow y \subseteq v_1)).$$

  This extends in the obvious way the treatment of this operation as a base function.

- **Set-collapse**: The function *set-collapse*(f) is represented by

$$\exists v_1\, (\psi_f(u, v_1) \wedge \forall x\, (x \in v \leftrightarrow \exists y\, (y \in v_1 \wedge x \in y))).$$

  This also is the obvious extension of the treatment of this operation as a base function.

- **Select**: The function $\sigma\langle\, p\,\rangle\,(f)$ is represented by

$$\psi_f(u, v_1) \wedge \forall x(x \in v \leftrightarrow \in v_1 \wedge \psi_p(x)\,)\}$$

- **Replace**: The function $\rho\langle\, f\,\rangle\,(f_1)$ is represented by

$$\exists v_1\, (\psi_{f_1}(u, v_1) \wedge \forall x\, (x \in v \leftrightarrow \exists y\, (y \in v_1 \wedge \psi_f(y, x)\ )\,)\,).$$

Boolean functions: We illustrate only two cases.

- **Membership**: The function $f_1 \in f_2$ is represented by

$$\exists v_1 \exists v_2\, (\psi_{f_1}(u, v_1) \wedge \psi_{f_2}(u, v_2) \wedge v_1 \in v_2)).$$

- **Negation**: $\neg p$ is represented by $\neg \psi_p$. (representing the boolean connectives in the calculus is easy, since they are built-in.)

Next, we consider construction.

- **Replace**. Given the formula for f, the formula for $\rho\langle\, f\,\rangle$ is

$$\forall x(x \in v \leftrightarrow \exists y(y \in u \wedge \psi_f(x, y))).$$

- **Select**: The formula for $\sigma\langle\, p \,\rangle$ is

$$\forall x(x \in v \leftrightarrow x \in u \wedge \psi_p(x)).$$

The fact that all the calculus formulas obtained in the (q) part are domain independent follows immediately from their equivalence to a-queries. $\Box$
This concludes the proof of the proposition, and also of the equivalence of the algebra and the domain independent calculus.

7 Syntactic Safety and Powerset

Domain independence is a semantic concept, that is undecidable even for the relational model [Va]. In this section, we present syntactic restrictions on formulas that guarantee domain independence. The formulas that are thereby restricted are called *safe*. In contrast to domain independence, these restrictions are easily checked. Furthermore, although not every domain independent formula is safe, we show that the domain independent calculus, the safe calculus, and the algebra are equivalent. In particular, for every domain independent formula there exists an equivalent safe formula. Finally, we show that by strengthening the restrictions, one obtains a calculus that is equivalent to the algebra without *powerset*.

7.1 Safety

The accepted approach to making a formula safe is to require that each variable is attached to a range formula. The simplest forms of range formulas are $\widehat{R}(x)$, where $\widehat{R}$ is a name of one of the database objects, and certain boolean combinations thereof. These are the forms used in the relational model. However, since we allow nested structures, other possibilities exist. Consider, for example, the scheme $R : \{[A, B : \{[\ldots]\}, C]\}$. We may want to use a variable x whose range is the set of tuples occurring in B-sets in tuples of R. To restrict the range of x to this collection, we need first to have a variable, say y, that is range restricted to R. Then we can range restrict x by $x \in y.B$.

In the following, we define range formulas, range restricted variables, and safe formulas.

Let α be a formula, and assume a partial ordering on its variables is given. We assume that the free and bound variables are distinct. We say that variable x is *(range) restricted* in α, relative to the given ordering, if the type of x is the empty tuple type, or one of the following holds.

Basis: α is one of

(B1) $\widehat{R}(x)$,

(B2) $x \in t$,

(B3) $x = t$,

(B4) $x \subseteq t$,

where, in the above, t is a term that may contain constants or variables that precede x in the ordering. Recall that the functions that can be used to construct terms are tuple and set constructors, and attribute application.

(B5) $\forall y(y \in x \rightarrow \varphi(y))$, where all variables that are used in φ (including y) precede x in the ordering. Note the similarity to the construction used for *powerset* in the translation from algebra to calculus. This is equivalent to $x \subseteq \{y \mid \varphi(y)\}$.

Closure: α is constructed from subformulas such that one of the following holds:

(CL1) α is a conjunct of two formulas and x is restricted in at least one of them.

(CL2) α is a disjunct of two formulas, and one of the following holds: either x is free in it, in which case it must appear and be restricted in both disjuncts, or x is bound in it, in which case it appears in precisely one of the disjuncts, and is restricted in the disjunct where it appears.

(CL3) α is obtained from a formula in which x is restricted by adding an existential quantifier (on x or on another variable).

(CL4) α is obtained from a formula, in which x is both bound and restricted, by adding negation.

A formula is *safe* relative to a given partial ordering, if all the variables are restricted in it. (This includes those in a subformula of the form (B5).) It is safe, if there exists an ordering, such that it is safe relative to it.

The rules above define the notion of "range restricted" for both free and bound variables. Clearly, the empty tuple type needs no range restriction: its only value is [], anyhow. The five base cases are formulas where x is directly restricted in α either because it is required to be in some database relation, or because it is required to be equal to/a member of/a subset of a value obtained from constants and variables that precede x by the construction and decomposition functions of the calculus, or (in (B5)) because it is required to be a subset of the result of a query expressed by a formula, where all the variables in that formula precede[23] x. Note that if the term t in (B2) - (B4) contains variables, then the range restriction of x depends on these variables, in the sense that it restricts x only if these variables are properly restricted. If α is safe, then these variables have their own range restrictions, and further, their restricting formulas do not depend on x directly, or even indirectly through other variables. This is guaranteed by the assumption that the range restriction for a variable depends only on variables that precede it in the ordering. A similar remark applies to (B5).

We will call the formulas in (B1) - (B5), and formulas obtained from them by using $\wedge$ and $\vee$, *range formulas*.

We have included in our definition enough constructions so that it is possible to simulate the algebra in the resulting calculus. Additional forms of safety can, of course, be added, and all that is needed (given our proof of equivalence to the domain independent calculus) is to show that they preserve domain independence.

Note that we use only the existential quantifier in the closure rules. Indeed, if x is free and restricted in a formula, it is not necessarily restricted when a universal quantifier on x is appended. Since we have negation, there is no loss of generality in formulating a closure rule only for the existential case. In particular, we will have later formulas of the form $\forall x(\varphi(x) \to \psi)$, where x is restricted in φ. This can be taken as a short notation for $\neg\exists x(\varphi(x) \wedge \neg\psi)$, in which x is indeed restricted[24] We can also view this form as a derived restriction rule for universal quantification. The universal quantifier in (B5) is used not for constructing a formula from a subformula, but rather to create a new atomic range restriction. To avoid confusion resulting from the use of $\forall$ in this construction, we shall use from now the equivalent notation $x \subseteq \{y \mid \varphi(y)\}$.

[23](B5) is a more powerful construction than (B2) - (B4); its role is clarified later.

[24]The quantifiers in $\exists x(\varphi(x) \wedge \psi)$ and $\forall x(\varphi(x) \to \psi)$ occur often in safe formulas. They are customarily called *bounded* quantifiers, and the formulas are often written as $(\exists x \varphi(x))\psi$ and $(\forall x \varphi(x))\psi$, respectively.

We refer to the calculus, restricted to safe formulas, as the *safe calculus*.

Theorem 7.1 *The safe calculus and the algebra are equivalent. That is, for each c-query with a safe formula there exists an equivalent a-query, and for each a-query there exists an equivalent c-query with a safe formula.*

Corollary 7.2 *The safe calculus is equivalent to the domain independent calculus.*

One direction of the Theorem follows from the following proposition:

Proposition 7.3 *The safe calculus is domain independent.*

Proof: We say that a formula, or a query, is domain independent in some variable, if for every database, changing the domain of this variable only, but requiring that it still contains the elements of the type of that variable in the active domain, does not change the result of the query. Thus, a formula is domain independent if it is domain independent in each of its variables.

Now, assume given a formula that is safe with respect to a given ordering of the variables, $x_1, \ldots, x_n$. We prove, that it is domain independent in each of $x_1, \ldots, x_n$. This is proved using induction on the number of variables, that is we prove that it is domain independent in each of $x_1, \ldots, x_i$, for $i = 1, \ldots, n$. For a given number of variables, we use induction on the structure of the formula.

For the first variable, x_1, we have only (B1)-(B4) as the base cases. Further, the terms in the last three cases must be ground. It is immediate that these formulas define domain independent queries. It is also easy to see that domain independence for x_1 is preserved under the four closure rules.

Now assume the claim was proved for i variables, and consider x_{i+1}. Consider one of the base formulas (B1)-(B5) for x_{i+1}. Here again, the formula of (B1) is domain independent. In (B5) the formula $\varphi(y)$ does not contain x, and further, it contains only variables that precede it in the ordering. By the induction assumption, the domains for these variables can be restricted to the active domain. It follows that this subformula is domain independent for x_{i+1}, i.e., it can only be assigned values from its active domain. Similarly, in (B2)-(B4), the terms may only contain variables that precede x_{i+1}, so the same reasoning applies. To conclude, we use induction on the closure rules. $\Box$

It follows from the proposition that for every safe c-query there is an equivalent a-query. To conclude the proof of the theorem, we just need to show a translation from the algebra to the safe calculus. Thus, a direct translation from the safe calculus to the algebra is not necessary. Nevertheless, there are good reasons to present such a direct translation. First, such a translation constructs a-queries for the domains of variables that do not need to construct the active domain of the database. Rather, these queries use the range formulas for each variable. This translation therefore provides useful insight on the translation from calculus to algebra. Second, we can see precisely where the powerset operation is used in this translation. Hence, we can derive a version of the calculus that is equivalent to the algebra without powerset. The translation is presented in the appendix.

7.2 From Algebra to Calculus

We prove the following

Proposition 7.4 *(q) For every a-query E, there exists an equivalent safe c-query q_{E},. (r-s) For every replace specification f, there exists a formula $\psi_f(u, v)$, that represents it, in which all variables, except possibly u, are restricted (relative to some ordering such that u is first and v is last in it). Similarly, for each select specification p, there exists a formula that represents it, such that all variables, except possibly u are restricted in it (relative to some ordering such that u is first in it).*

Proof: We follow the translation of the algebra to the calculus in the previous section, and we show that the formulas constructed there satisfy the proposition. We just need to check that the constructed formulas are safe (for the first part), or that all the variables except possibly u are restricted (for the second part).

(q): The base a-queries are $\{c\}, \widehat{R}$. The corresponding c-queries are defined by $x = c$ and $\widehat{R}(x)$, which are safe (by (B1), (B3)). For the induction, assume that E is obtained from E_1, E_2 by applying some operation, and that the corresponding c-queries $\varphi_{E_1}, \varphi_{E_2}$ are safe. We note that the variables that are bound and restricted in these formulas remain bound and restricted 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 restricted. In each case, the ordering of the variables is such that the new variables, if any, are last.

If θ is one of the set operations of *union, intersection* and *difference*, then the c-queries for $E_1\theta E_2$ are $\varphi_{E_1}(x)\widehat{\theta}\varphi_{E_2}(x)$, where $\widehat{\theta}$ is $\vee, \wedge, \wedge\neg$, respectively. Safety follows from (CL2) for disjunction, and from (CL1) and (CL4) for the two cases of conjunction.

For $cross_{[A_1,A_2]}$, the translation is

$$\exists x_1 \exists x_2\, (\varphi_{E_1}(x_1) \wedge \varphi_{E_2}(x_2) \wedge x = [A_1 : x_1, A_2 : x_2]).$$

In this formula, x_1, x_2 are obviously restricted, and remain restricted when the quantifiers are applied by (CL3). Since x follows all other variables, x is also restricted (by (B3)), so the formula is safe. For *powerset*, the formula is $\forall x_1\, (x_1 \in x \rightarrow \varphi_E(x_1))$, or $x \subseteq \{x_1 \mid \varphi(x_1)\}$. All variables (except x) are known to be restricted, and x is restricted by (B5). For *set-collapse*, the formula is $\exists x_1(\varphi_{E_1}(x_1) \wedge x \in x_1)$. It is clearly safe (by (B2) and (CL3)).

We are left with *select* and *replace*. These are more complex, since they contain function parameters. Consider the *replace*. Let the a-query be $\rho\langle\, f\,\rangle\,(E_1)$. Let $\psi_f(u, v)$ be the formula that represents the function f. Then $\varphi_E \equiv \exists u\, (\varphi_{E_1}(u) \wedge \psi_f(u, v))$. We note that by induction hypothesis φ_{E_1} restricts u, and all variables in ψ_f, except possibly u, are restricted. Hence (by (CL1)), this formula is safe. *Select* is treated similarly.

(r-s) The formulas for the replace specifications $c, \widehat{R}, A, id$ are $u = u \wedge v = c$, $u = u \wedge \forall y(y \in v \leftrightarrow \widehat{R}(y)$, and $v = u.A$, respectively. In the first and third, v is restricted by (B2). The second can be rewritten to the form $v \subseteq \{y \mid \widehat{R}(y)\} \wedge \forall z(\widehat{R}(z) \rightarrow z \in v)$. Now, v is restricted (by (B5)), and z by the derived closure rule for universal quantification. *Set-collapse* is represented by $\forall x(x \in v \leftrightarrow \exists y(y \in u \wedge x \in y))$; denoting this formula by φ, it is equivalent to $v \subseteq \{x \mid \exists y(y \in u \wedge x \in y\}) \wedge \varphi$, in which v is restricted by (B5). It is easy to see that both y and x are restricted (the order is u, y, x, v). Hence this formula satisfies the claim. *Powerset* is represented by $\forall x(x \in v \leftrightarrow x \subseteq u)$, and this can be similarly converted to a formula that satisfies the claim.

For specifications, we have the same situation as above, namely that we need to check only what happens to free or new variables. Checking that all variables except possibly u are restricted is in most cases straightforward, so we only illustrate some cases.

If A is an attribute, then the formula for $f.A$ is $\exists x(\psi_f(u, x) \wedge v = x.A)$. We have that v is restricted (by (B3)), and we already know that all other variables, except possibly u

are restricted.

For *powerset*, the translation for $powerset(f)$ is $\exists v_1(\psi_f(u, v_1) \wedge \forall y(y \in v \leftrightarrow y \subseteq v_1))$. As above, the second part can be rewritten as $\forall y(y \subseteq v_1 \rightarrow y \in v) \wedge v \subseteq \{z \mid z \subseteq v_1\}$. The last conjunct restricts v, whereas in the previous conjunct y is bound by a bounded quantifier, hence is restricted, and we already know that v_1 is restricted.

For the *replace*, we have a similar construction: The formula for $\rho\langle f\rangle(f_1)$ is $\exists v_1(\psi_{f_1}(u, v_1) \wedge \forall y(y \in v \leftrightarrow \exists z(z \in v_1 \wedge \psi_f(z, y))))$. Rewriting as in the previous case, we obtain

$$\exists v_1(\psi_{f_1}(u, v_1) \wedge \forall y(\exists z(z \in v_1 \wedge \psi_f(z, y)) \rightarrow y \in v)$$

$$\wedge v \subseteq \{s \mid \exists z(z \in v_1 \wedge \psi_f(z, s))\}.$$

Since v_1 is restricted, the last part restricts v (by (B5)), whereas z is restricted in the part before last, hence so is y since it is bound by a bounded quantifier. All the other variables are restricted by induction hypothesis.

The cases of other base functions, algebraic operations, and the boolean functions are similar, or simpler, and are left to the reader.

Next, we consider construction. Given the formula for f, the formula for $\rho\langle\, f\,\rangle$ is

$$\forall x(x \in v \leftrightarrow \exists y(y \in u \wedge \psi_f(x, y)))$$

We do the same transformation as in previous cases: We "and" the formula with $v \subseteq \{y \mid y \in u \wedge \psi_f(x, y)$. This restricts v, and after we eliminate the existential quantifier and the implication, we have that the other variables are restricted as well. The *select* is treated similarly. □

7.3 Strictly Safe Calculus

The powerset operator of the algebra is unique among the algebraic operations in that one may argue that it is not directly useful for expressing real-life queries. Additionally it may cause exponential cost, since it increases the size of its argument exponentially. It is of interest, therefore, to consider the algebra without the *powerset*. We have seen in the previous section that it is used in the proof of equivalence to the domain-independent calculus only in one place, in the construction of a-queries for the domains. Now that we have the notion of safety, and we have other constructions of a-queries for the domains, we can possibly do without this operator.

When we consider the equivalence proof of the algebra and the safe calculus, we observe the following: In the translation from the domain independent calculus to the algebra (given in the appendix), we used the *powerset* only for the translation of (B4) and (B5). Only in these two forms is there a use of the subset predicate. Thus, we conclude that if we remove the *powerset* from the algebra, we need to remove (B4) and (B5) from the safety definition. In the other direction, the translation from the algebra to the calculus, the situation is a bit more complex. We have indeed used (B5) only in the translation of the *powerset* in the (q) part. Similarly, we used (B4) for the translation of *powerset* in the (r-s) part. However, (B5) is used in some of the other translations, e.g., in translating $\widehat{R}$ and $\rho\langle f\rangle(f_1)$. We observe, though, that in these cases the relevant part of the formula has the form $v = \{y \mid \varphi\}$. We claimed safety for this formula by breaking it into two components, one of which has the form suitable for (B5). The same holds for the other cases where (B5) is used. What we offer to do now, instead, is to replace (B5) by a weaker condition:

(B5') $x = \{y \mid \varphi\}$.

This is weaker than (B5) since it can not yield any exponential growth. It allows to translate all the cases where (B5) was used in (r-s), except the translations of *powerset*. Note in particular the translation of $\rho\langle f\rangle(f_1)$: $\exists v_1(\psi_{f_1}(u, v_1) \wedge v = \{y \mid \exists z(z \in v_1 \wedge \psi_f(z, y))\})$. This simply states that v is obtained from v_1 by applying f to each of its elements, which is just what is needed. Note the similarity to the replacement axiom of set theory.

We call the safe calculus, without (B4) and with (B5) replaced by (B5'), the strictly safe calculus. We now have:

Theorem 7.5 *The strictly safe calculus and the algebra without* powerset *are equivalent.*

Proof: We have already considered the translation from the algebra to the calculus, and have argued that if the algebra does not contain *powerset*, then it can be carried out, provided we have (B5') among the clauses defining range restrictions. We need only to consider the other direction, and particularly, since we have removed (B4) and (B5), we do not need the *powerset* in the algebra. However, now we cannot use the claim that this calculus is included in the domain independent calculus to complete the proof, as we did in the beginning of the section. Rather, we need to transform the the translation from the safe calculus to the algebra given in the appendix, so that the translation for (B5') does not use *powerset*. That is easy: For $x = \{y \mid \varphi\}$, we have an a-query that returns a set of y values Applying *nest* we have an a-query that returns a set of pairs of the form $\{[v, \{y \mid \varphi\}]\}$. So far, these are the steps as performed in the proof in the appendix. The last step there is to apply a *powerset*, and since we are dealing with $x = \{y \mid \varphi(y)\}$, rather than with $x \subseteq \{y \mid \varphi(y)\}$, we simply do not need to apply *powerset* to the second component. $\Box$

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[25]. The inclusion of arbitrary function and predicate symbols in the languages is easy. The calculus is a paradigm that allows the use of any function or predicate name. In the algebra, operations like *replace* and *select* were defined explicitly so that any function or predicate parameter of the appropriate types can be used. Thus, we can add arbitrary functions and predicates to the collection of base functions, and there is no need to change any of the other definitions[26].

We do have a problem, however, to consider domain independence and related notions, since 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

[25] Note that we use the predicates of equality and membership in our languages and these are interpreted, since they have fixed meanings.

[26] Note, however, that aggregates such as average or count may require the use of bags rather than sets, for producing correct results. The problem can be addressed either by including *bag* as a type constructor, or using, e.g., Klug's approach [33]. We do not consider the issue further in the paper.

of the enhanced calculus (provided of course, that the same predicates and functions are included in both languages).

Assume that a set Δ_{fn} of functions, and a set Δ_{pr} of predicates are given. Functions in Δ_{fn} and predicates in Δ_{pr} 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. We assume that:

($\star$) for each function f in Δ_{fn}, f^{-1} is also in Δ_{fn}.

Note that when a function is not 1-1, its inverse is set-valued; since we have sets in the model, that is not a problem. Our assumption implies that for each x, the set $f^{-1}(x)$ is finite.

This is a reasonable assumption. For instance, consider the difference function over the integers which does not satisfy this property (e.g., $f^{-1}(0)$ contains the pairs $[1,1],\ldots,[n,n]\ldots$). We prohibit such functions since they may cause problems as in following query:

$$\{x \mid \exists z \exists y(\widehat{R}(y) \wedge y = z - x)\}.$$

We cannot expect to compute the answer to this query in finite time. Such a query will not be considered domain-independent under the definitions below. (We will briefly consider below an extension which allows the difference function in certain contexts.)

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

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

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

To extend the algebra, we allow any predicate in Δ_{pr} to be used in a *select* operation. We also allow using functions from Δ_{fn} in replace-specifications. For instance, the following are a-queries, expressing the same queries as the c-queries above:

$$\mathit{replace}\langle\ [A, \mathbf{count}(B)]\ \rangle(\widehat{R}),$$

$select\langle\ B \geq 5 \wedge \mathbf{even}(B)\ \rangle(\widehat{S})$.

We need to reconsider the definitions of a database scheme and instance. In a database scheme, some of the domain names are now 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_{pr}, Δ_{fn} 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 as described in the scheme. It also "contains" the interpreted functions and predicates. We denote such a database structure by $DBI = \langle [D_1, \ldots, D_k], \bar{R}, I \rangle$, where $\bar{R}$ is the vector of relations and I is the interpretations of the fixed domains and of the additional functions and predicates.

Now consider domain independence. Given a query, it is not enough to consider for the active domain 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 $+$, then queries can ask for each of the integers, hence the domain should contain all integers. On the other hand, having infinite sets included in the active domain seems to defeat our intention in the definition of domain independence. In order to have a useful notion of domain independence, 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 (and their inverses) are applied.

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

- $close^0(\bar{R}, C) = \mathrm{ATOM}(\bar{R}, C)$
- $close^{n+1}(\bar{R}, C) =$

$$\begin{aligned} & close^n(\bar{R}, C) \cup \bigcup\{\mathrm{ATOM}(I(f)(x_1, \ldots, x_l)) \mid f \in \Delta_{fn}, \\ & \qquad \forall i = 1, \ldots, l, \mathrm{ATOM}(x_i) \subseteq close^n(\bar{R}, C)\} \\ & \quad \cup \{\mathrm{ATOM}(x) \mid \exists x_1, \ldots x_{i-1}, x_{1+1}, \ldots, x_l, \\ & \qquad (\mathrm{ATOM}(f(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots) \in close^n(\bar{R}, C))\} \end{aligned}$$

Thus, if we have the natural numbers with *succ* and $succ^{-1}$, and we are initially given the numbers 1 and 8, then two closure steps generate the set $\{0, 1, 2, 3, 6, 7, 8, 9, 10\}$.

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$(\bar{R}, C_q)$, where C_q is the set of constants used in the query and this set is finite. That implies that we should be able to evaluate the query on any database where the interpretations of the interpreted domains agree with their fixed interpretations on this closure, but may be quite different outside it. Necessarily, the interpreted functions and predicates must be given an interpretation on those new domains. All that we can require is that these interpretations agree with the fixed interpretations on the given closure. Thus, to make this notion of n-depth domain independence precise, we have to relax our restriction on the interpretations of the functions and predicates in $\Delta_{fn} \cup \Delta_{pr}$ and their domains. Formally, for n-depth independence, we allow to use any domains that include $close^n(\bar{R}, C_q)$, and such that the computation of the n-closure in these domains gives the same results as in the original domains. The last requirement ensures, in particular, that if we apply an inverse function to any elements in $close^{i-1}(\bar{R}, C_q)$, we obtain only elements that are in $close^i(\bar{R}, C_q)$, for all $i = 1, \ldots, n$, even though the functions may have arbitrary behavior outside this set. Assuming this relaxation, a query is *n-depth domain independent* if $q(DBI) = q(DBI')$, for any DBI' that agrees with DBI on $close_n(\bar{R}, C_q)$, as described above. A query is *bounded depth domain independent* if it is n-depth domain independent, for some n.

Theorem 8.1 *Let Δ_{fn} and Δ_{pr} 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 *attribute selection*, the depth is the maximum of the depths of the argument replace specifications. The same holds when a function or operator is applied to replace specifications, except in the case of applying a *replace*. Assume we have constructed the replace specification $f = \rho\langle g_1 \rangle(g_2)$, and let the depths of g_1, g_2 be n_1, n_2, respectively, then the depth of f is $n_1 + n_2$. Finally, if the depth of g is n, and $f \in \Delta_{fn}$, 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 $\rho\langle f \rangle(E)$, the depth is the sum of the depths of f 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, which will also be n-depth independent. The only extensions to the constructions are in the treatment of replace specifications: in a *select*, θ may be any of the predicates in Δ_{pr}, not just one of the three built-in predicates $\in, =, \subseteq$, and we may apply a function from Δ_{fn} to replace specifications, in addition to being able to apply algebraic operations. For example, if we have the formula ψ_g for the replace specification g, then the formula for $f(g)$ is $\exists v_1(\psi_g(u, v_1) \wedge v = f(v_1))$. The details are left to the reader.

In the opposite direction, assume we have an n-depth domain independent c-query. We translate it into an a-query following the construction of Section 6, with the following extension. Having constructed a-queries for the sets of atomic values of each atomic type that appear in the database or the query (the atomic part of the active domain), we need to perform n closure steps before we construct domains for non-atomic types. That is, given a-queries for $close^0(\bar{R}, C_q)$, we need to generate a-queries for $close^n(\bar{R}, C_q)$. Given an a-query, say E, representing a subset S of domain of a function f the a-query $\rho\langle f\rangle(E)$ represents the set $\{f(x) \mid x \in S\}$, i.e., a forward closure step. Similarly, since inverse functions are represented as functions as well, we can perform a backward closure step. (This is where $(\star)$ is used.) The *cross, powerset, projection, set-collapse* and *union* can be used to create sets of more complex types, or to decompose elements of sets into components, as needed to allow us to apply the functions, and to collect the new atomic elements.

In the original construction, only the predicates $\in, =, \subseteq$ were used. Clearly, there is no problem in using any of the predicates in Δ_{pr} as well. Finally, interpreted functions may be used in the construction of terms. Incorporating this fact into the proof creates no problem either. It is easy to see that, if the given query is n-depth domain independent, this translation generates an equivalent a-query. $\Box$

We can also generalize the notions of range formulas and safety, by adding the following clause:

(Func) $x = f(s_1, \ldots, s_k)$ (where the s_i precede x in the ordering).

An immediate consequence of this extension 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 6, augmented as described above. It is easy to see that the claim that all variables in $\psi_f(u, v)$, except possibly u, are restricted remains valid. For the case where an interpreted function is applied we use (Func) above. Thus, we have,

Theorem 8.2 *(I) The extended safe calculus, with Δ_{fn} and Δ_{pr}, is equivalent to the extended algebra, with Δ_{fn} and Δ_{pr}.*
(II) The extended strictly safe calculus, with Δ_{fn} and Δ_{pr}, is equivalent to the extended algebra, with Δ_{fn} and Δ_{pr}, without the powerset

The restrictions that we impose are quite brutal. For instance, one could allow functions such as integer difference in a limited manner. For instance, consider query:

$$\{x \mid \exists y, z(R(y, z) \wedge x = y - z)\}$$

or

$$\{x \mid \exists y, z(R(y, z) \wedge y = x - z)\}.$$

Although difference has no inverse, one may argue that such constructions are very "safe" and should be allowed. This clearly can be done, see [15].

9 Recursive Queries

In this section, we show that the domain independent calculus (hence also the algebra) permits the specification of queries which would require a fixpoint in relational calculus. 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 [8]. 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 $\widehat{R} : \{[A, B]\}$, where A and B have the same type. The transitive closure of $\widehat{R}$ can be computed in the following way:

- A first formula ψ_1 is used to obtain the set $\widehat{R}_1$ of tuples $[A, B]$ built using values in $\widehat{R}$ (the variable x is of type $[A, B]$):

$$\psi_1(x_1) \equiv \exists y, z(\widehat{R}(y) \wedge \widehat{R}(z) \wedge (x_1.A = y.A \vee x_1.A = y.B)$$
$$\wedge(x_1.B = z.A \vee x_1.B = z.B)).$$

- A formula ψ_2 gives the set $\widehat{R}_2$ of subsets of $\widehat{R}_1$ (i.e., the powerset of $\widehat{R}_1$).
  $\psi_2(x) \equiv x \subseteq \{x_1 \mid \psi_1(x_1)\}$ where x is of type $\{[A, B]\}$.
- Formula ψ_3 gives the set $\widehat{R}_3$ of elements in $\widehat{R}_2$ containing $\widehat{R}$:
  $\psi_3(x) \equiv \psi_2(x) \wedge \forall z(\widehat{R}(z) \rightarrow z \in x)$.
- Formula ψ_4 gives the set $\widehat{R}_4$ of elements in $\widehat{R}_3$ that are transitively closed:
  $\psi_4(x) \equiv \psi_3(x) \wedge \forall u, v(u \in x \wedge v \in x \wedge u.B = v.A \rightarrow [u.A, v.B] \in x)$.
- Finally, the transitive closure of $\widehat{R}$ is obtained by intersecting the elements of $\widehat{R}_4$:
  $q \equiv \{x' \mid \forall x(\psi_4(x) \rightarrow x' \in x)\}$.

We could extend the calculus with a fixpoint operator as in the language *fixpoint* of [20]. However, the technique used in the previous example can be generalized to demonstrate that this would result in no gain of expressivity, i.e., *complex-value-fixpoint* is no more expressive than (complex value) calculus.

In the same spirit of introducing recursion, we now present a simple language based on recursive rules. We handle negation using the concept of "layers" as in [9, 4, 13, 40, 49] and others.

Given a database scheme, the relation names in it, $\widehat{R}_1, \ldots, \widehat{R}_n$, are called *base* relations. The language will use 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. However, it is important to note that derived predicates also have a given signature. The language is typed. The signature for a predicate specifies the element type, so in particular, all elements have the same form. A *literal* is an atomic formula or a negated atomic formula. A *rule* is an expression of the form

$$P(t) \leftarrow L_1, \ldots, L_n,$$

where P is a derived predicate, and each L_i is a literal. A rule is interpreted as the formula

$$\forall x_1 \ldots \forall x_m(L_1 \wedge \ldots \wedge L_n \rightarrow P(t)),$$

where $x_1, \ldots, x_m$ are all the variables appearing in it. A recursive query is a pair $\langle \mathcal{P}, Q\rangle$ where $\mathcal{P}$ is a finite set of rules, and Q is a derived relation.

Rules, programs and queries also need to be domain independent. For an extended discussion see [48]. For our purposes, the following should suffice. As in Section 7, let us consider an ordering on the variables appearing in a rule. We say that a variable is restricted in the body (relative to the ordering), if it follows from rules (B1) – (B4), (CL1) and (CL4) of Section 7. The body is *safe* if for some ordering of its variables, all its variables are restricted. A rule is *safe* if each variable that appears in the head also appears in the body, and the body is safe. We will assume henceforth that rules are safe.

We assume familiarity with the semantics of programs without negation. Negation poses difficulties: it is not always possible to assign a meaning to a program with negation. This subject has received considerable attention in recent years. We consider stratification, as suggested as in [9, 20, 40, 49].

A *stratification* of a program $\mathcal{P}$ is a partition $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of the program (i.e., of the set of rules) such that the following hold:

1. All the rules defining a derived relation are contained in a single stratum. The facts defining the base relations are all in $\mathcal{P}_1$.

2. If the rule $P(x) \leftarrow \ \ldots, Q(y), \ldots$ is in $\mathcal{P}_i$, then the rules defining Q are in some $\mathcal{P}_j$, for $j \leq i$.

3. If the rule $P(x) \leftarrow \ \ldots \neg Q(y), \ldots$ is in $\mathcal{P}_i$, then the rules defining Q are in some $\mathcal{P}_j$, for $j < i$.

Each element of the partition is called a a *stratum*.

A program is *stratified*, if there exists a stratification for it. Note that a stratification for a program induces a partition for the predicates appearing in it, in which a predicate P is associated with the stratum where the rules defining it appear.

The semantics for stratified programs is a simple extension of the semantics of programs without negation: one first computes the fixpoint of the rules of the first stratum, applied to the database, then the fixpoint of the rules of the second stratum, applied to the result of the first stage, and so on. It is known [9] that the final result is independent of the specific layering chosen for the program. The semantics of stratified recursive queries is defined in the obvious way: compute the extensions of all derived relations; the result is the extension of the selected derived relation.

Example 9.2 The database object $\widehat{R}$ is of type $\{[A, B : \{[C, C']\}]\}$, and the query defines a derived relation Q, which contains the tuples of $\widehat{R}$, with the B-component replaced by its transitive closure. Let us assume that we have a ternary predicate ins, where $ins(x, y, z)$ is interpreted as "z is obtained by inserting x into y". We show later how to express it in the language.

r1: $\widehat{S}(x, y) \leftarrow\ \widehat{R}(x, y)$

r2: $\widehat{S}(x, z) \leftarrow\ \widehat{S}(x, y), u \in y, v \in y, u.B = v.A, ins(z, [u.A, v.B], y),$

r3: $\widehat{S}'(x, z) \leftarrow\ \widehat{S}(x, z), \widehat{S}(x, z'), z' \not\subseteq z$

r4: $\widehat{Q}(x, z) \leftarrow\ \widehat{S}(x, z), \neg\widehat{S}'(x, z).$

The first two rules compute in S pairs corresponding to pairs from $\widehat{R}$, such that the second component of a pair contains the corresponding component from the pair in $\widehat{R}$, and possibly additional elements derived by transitivity. Obviously, for each pair $[x, y]$ of $\widehat{R}$, there is a pair $[x, z]$ is $\widehat{S}$, such that z is the transitive closure of y, but there are other tuples as well. To answer the query, we need to select for each x the unique tuple (x, z) of $\widehat{S}$ where z is maximal[27]. The third rule puts into $\widehat{S}'$ tuples (x, z) such that z is not maximal for that x. The last rule then selects those that are maximal, using negation.

We now show, for a given type T, the program that defines ins for sets of type $\{T\}$ (the variables are all of type $\{T\}$):

$super(z, x, y) \leftarrow\ x \in z, y \subseteq z$

$not\text{-}min\text{-}super(z, x, y) \leftarrow\ super(z, x, y), super(z', x, y), z' \subseteq z$

$ins(z, x, y) \leftarrow\ super(z, x, y), \neg not\text{-}min\text{-}super(z, x, y)$

Note that the program is type specific only through its dependence on the types of the variables. The same program computes ins for another type T', if we assume that the variables are of that type. Note also that the above program is not safe. To make it safe, one would have to use derived relations to range restrict the various variables.

[27] We assume, for simplicity, that the first column of $\widehat{R}$ is a key. It is easy to change the rules for the case when this does not hold.

We note that although we used $\subseteq$ in the example as a built-in predicate, it can be expressed using membership and stratified negation. Also, a *union* predicate can be defined, by a program similar to that used for *ins*.

Our main result is the following:

Theorem 9.1 *A query is expressible as a (safe) stratified 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 assume given base relation schemes. We also use variables whose type is the type of the cross product of all the relations, both base and derived. Obviously, we can express in the calculus the requirement that the value for such a variable is a cross product of its projections corresponding to the individual relations. We can also restrict such a variable such that each atomic component of its elements is an atomic value that appears in the database or in the given program. (This corresponds to ψ_1 in Example 9.1.) Note that this guarantees that the variable is restricted, and consequently the safety of our query. Now, the body of a safe rule is a safe query, and the head of the rule can be obtained by "projection". If we have several rules defining a predicate, we can combine them using *or*. Thus, for a *product* variable v, we can express the requirement that the components of v corresponding to the base relations are equal to the corresponding database relations, and each component corresponding to a derived relation satisfies the corresponding rules of the program. That is, we can express the requirement that v contains the least fixpoint of the program. This construction can actually be carried out for the relational calculus as well. The difficulty is to force the value to be precisely the least fixpoint. Assume $\varphi(v)$ is the formula described so far. For the rest, we mimic the last steps of Example 9.1, where we showed how to select the smallest set in a collection of sets.

For a general stratified program, we do a similar construction for each stratum. That is, after all the strata up to and including $\mathcal{P}_i$ have been "applied", we treat all the derived relations of those strata as base relations in the construction for the next stratum. Finally after we have a formula defining the values of all derived relations, it is easy to select the values for the query.

From safe calculus to recursive queries: Given a c-query, we transform it as described in Section 7, so that each subquery has an associated range restriction for its free variables. (As assumed there, we eliminate first all universal quantifiers.) Now, we construct the recursive program using induction on the structure of the query. Each subquery is treated together with its associated range restriction, which guarantees safety of the rules, and we use a (new) derived predicate for each subquery. Conjunction, of course, poses no problem; disjunction is modeled by having one rule for each disjunct. Negation leads one step up the stratification hierarchy, and the existential quantifier is simulated by projection, i.e., if $\varphi(x_1, \ldots) \equiv \exists x \psi(x, x_1 \ldots)$, then we have a rule $P_\varphi(x_1, \ldots) \leftarrow\ P_\psi(x, x_1, \ldots)$. $\Box$

We note that this claim is not true for a non-typed language. If we allow relations to be heterogeneous, then we could write the following simple rule:

$$\textit{too-large}(\{x\}) \leftarrow\ \textit{too-large}(x)$$

Adding an appropriate exit rule, we have a program that computes an infinite relation, *without* using an external function. This program cannot be simulated in the algebra or the calculus.

To conclude this section, we consider interpreted functions and predicates into the rule-based language. A similar problem has been independently studied in [18], where aggregate functions are introduced in a language resembling that of [4]. 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.

The second problem arises, for example, when aggregates are applied to sets; only when the set if fully computed, it makes sense to apply the function. As for negation, stratification provides a solution to the second problem. It turns out that it can also provide a solution to the first problem. Indeed, an appropriate stratification of programs with interpreted functions (and predicates) leads to a language that has exactly the power of the algebra or the safe calculus. The stratification is defined as above, with the additional following constraint:

> if $P(..) \leftarrow\ \ldots Q() \ldots$ is in $\mathcal{P}_i$, and f, an interpreted function, is used in the rule, then the rules defining Q are in some $\mathcal{P}_j, j < i$.

The condition could be relaxed. For instance, in the rule $\widehat{S}(x, f(x)) \leftarrow \widehat{R}(x, f(x))$, the presence of $\widehat{R}$ and $\widehat{S}$ in the same stratum would not cause any problem, since the rule uses $\widehat{R}$ positively, and does not introduce new values. Intuitively, the stratification should be restrictive enough to guarantee that if a function is applied, the result should be in a new stratum.

To complete the informal description of the language, we remark that we need to extend the notion of safety as follows: if the body contains $y = f(x)$, and x is range restricted and precedes y, then y also is considered to be range restricted.

Theorem 9.2 *Let Δ_{fn} and Δ_{pr} 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.*

As we have noted, the stratification condition can be relaxed. But it cannot be dropped. For example, the rule $\widehat{R}(f(x)) \leftarrow R(x)$ applies f an unbounded number of times, hence it cannot be expressed in the calculus. That is, although the rule is a calculus formula, it is not safe, and there is no safe formula that expresses this query.

10 Conclusions

We have presented in this paper an approach to the generalization of query language paradigms to models that allow more structure than the relational model. We have considered a calculus-based, an algebraic and a logic-programming based paradigms. We have generalized each of the paradigms to the complex object model, and have considered the validity of the relationships known to hold between these paradigms in the relational setting. We have found that the equivalence of the algebra with the domain-independent calculus and a safe calculus hold, but the distinction between the calculus/algebra and a language that allows recursion no longer holds. Both these results depend on the inclusion of a *powerset* operation in the algebra, or equivalently on the unrestricted use of the calculus. We have therefore considered also the algebra without this operation, and presented a restricted version of the calculus equivalent to this algebra. We believe that this is the right algebra for complex objects; indeed the *monadic algebra* of [17] is essentially our algebra without the powerset, and they present arguments to support this claim.

Our algebra is based on the following principles: the use of type-specific operations given with type constructors, in particular the constructors and selectors, and of user-supplied functions on base types, as the main ingredients of restructuring functions; the use of composition as a major tool in the construction of both queries and restructuring functions; the use of a small number of higher-order operations to generate set functions from element functions. The concept of composition exists also in the relational model, in the form of the closure requirement for query expressions. The more complex structure of values, and particularly the possibility to recursively use the *set* constructor, require the inclusion of higher-order operations. We believe that this is the approach that should be taken to the design of generalized algebras. Our emphasis on composition brings to mind category theory, where composition is a central concept. It turns out that the semantic domain for the algebra is indeed a category, that *replace* is a functor in it, and *set-collapse* and *single* (the construction of singleton sets) are natural transformations, and furthermore, they satisfy the axioms of a *monad*. With the empty set and union, we have a *ringad*. For definitions and related work, see for instance [45, 50].

In addition to the general framework and specific results described above, it is of interest to consider which of those languages, if any, is best for practical use. In this context, it is important to note that we have not considered SQL extensions in this paper. Most implemented systems actually use languages based on this paradigm, see e.g., [21]. Comprehensions, considered in [45, 50], can be viewed as a pure form of generalized SQL. We conjecture that language paradigms based on comprehensions or similar ideas are more suitable for user-interfaces, whereas the algebra may be more suitable as an internal-representation language. However, these issues are outside the scope of this paper and deserve further study.

Appendix: From Calculus to Algebra

We now show that for each safe c-query, there exists an equivalent a-query. We follow the translation of the calculus to the algebra presented in Section 6.1, with one major difference. Recall that the translation there is based on a-queries for the domains of the variables, that are constructed from a-queries for the active domains of the database. Since now variables are range restricted, we would like to use a-queries that express these range restrictions as the domain expressions for variables. Thus, whereas the a-query constructed in the previous section for a given c-query q is equivalent to it (in the general case) only on DB_q, the a-query we construct below will be equivalent to q on all databases.

This raises a technical problem, namely that in general range formulas are associated with a query as a whole, or only with some of its subqueries, rather than with each of its subqueries. To prove our claims by induction on the structure of formulas, we need to associate, with each subformula, range restrictions for the variables that are free in it. We therefore show that every safe formula can be converted to an equivalent safe formula that satisfies this requirement, and we prove the claim for such formulas.

The transformation works as follows: First, we push the existential quantifiers to the outside, as much as possible. Recall that $\exists$ commutes with $\wedge, \vee$ (assuming variables do occur bound in a subformula, and also free outside it), but not with $\neg$. Thus, after the transformation, the formula α and each of its negated subformulas start with a (possibly empty) string of existential quantifiers, and there are no other quantifiers. This transformation leaves all variables restricted, so the formula is still safe. For a variable x that appears in the formula, we have the following cases: if x appears only in a negated subformula, it cannot be free there, since otherwise it would not be restricted in α. Hence, it is bound there by a quantifier $\exists x$. If x is free there, then it must also appear outside that subformula, so either it is bound in some enclosing negated subformula, or it is bound in the prefix of α, or it is free in α.

Think of α as a tree, where the leaves are atomic formulas, and the internal nodes are labeled by one of $\wedge, \vee, \neg, \exists x$, or by[28] B5. Each internal node corresponds to a subformula of α. The nodes just below the existential quantifier prefixes of α and its negated subformulas are distinguished in the following sense: for each x there is precisely one such

[28]The last case corresponds to a range formula of the form $x \subseteq \{y \mid \varphi\}$. This is not an atomic formula; it has φ as a subformula.

node, denoted n_x, such that all occurrences of x (except for the one in $\exists x$, if x is bound) occur below that node, and either the associated prefix contains $\exists x$, or x is free in α and the node is just below the prefix of α. Our transformation will associate a range formula for x with n_x, and each of the nodes below it corresponding to subformulas that contain x. We work on one variable at a time, starting from the one that is **last** in the order. We first describe the process for the case that x is this last variable, then describe briefly the changes needed for the other variables.

Going from n_x down, we mark each internal node by x_f if x is free in the corresponding subformula. Clearly, n_x is marked with x_f. Now, since x is restricted in α, there exists atomic subformulas that are range restrictions for x according to (B1) - (B5). But some of those, namely those that appear in the scope of negation, do not contribute to making x restricted in α. For example, in $S(x) \wedge \neg R(x)$, the subformula $R(x)$ by itself is a range formula for x, but it is not the subformula that restricts x in the whole formula. Going from n_x down, we replace in certain nodes the mark x_f by x_r, to signify that x is restricted in the subformula corresponding to that node (when it is considered by itself). Thus, when finish, we have unmarked nodes (x does not appear at all), nodes marked by x_f (x is free but not restricted), and nodes marked x_r (x is free and has a range restriction).

This marking process is done as follows: The node n_x is marked by x_r, since x must be restricted in the corresponding subformula. Given a marked node, we compute the mark(s) for its child(ren) as follows: First, if a node has the mark x_f, the marks of its descendants are not changed. Thus, we need only describe the "inheritance" rules for marks of the form x_r. For a node labeled $\vee$, x occurs and is restricted in both its two children. Thus, each child inherits the mark x_r. For a node labeled $\wedge$ we have a similar case, except that x may be free in a child, yet not restricted in the corresponding subformula. (Recall that if x is restricted in φ, it is also restricted in $\varphi \wedge \psi$, regardless of the structure of ψ.) Each child that has x free and restricted in the corresponding subformula inherits x_r; there is at least one such child. (In the example above, $S(x)$ would be marked by x_r, but $\neg R(x)$ would not; similarly in $S(x) \wedge x < 3$, only $S(x)$ would be marked by x_r.) A child that has x free but not restricted remains marked by x_f. Finally, if a node is labeled $\neg$, then in the corresponding subformula x cannot be restricted. Such a node may have the mark x_f, or no mark, and in either case, this is not changed. Finally, since the marking process stops at negations, and quantifiers occur only below negations, we do not need to consider quantifiers.

When the marking process terminates, there must be range formulas for x that are marked by x_r (otherwise the given formula is not safe). These are the "useful" range formulas. In the next stage, we push these range formulas upward, and also sideways into subformulas that are marked by x_f. We stop when we reach n_x. Before we begin, conjunctions and disjunctions of range formulas are merged, in the sense that from now on such a conjunction/disjunction is considered as an indivisible subformula. As we proceed, we continue to merge range formulas whenever possible.

Denote a range formula for x by $r(x)$. We use the following steps:

- A subformula $(r(x) \wedge \varphi) \ \wedge \ \psi$ (where ψ does not contain a range formula for x) is transformed to $r(x) \wedge (\varphi' \wedge \psi')$, where φ' is $r(x) \wedge \varphi$, if φ contains x free, and is just φ otherwise; ψ' is $r(x) \wedge \psi$ if ψ contains x (is marked x_f), and is just ψ otherwise. If the subformula has the form $(r(x) \wedge \varphi) \wedge (r'(x) \wedge \psi)$, then we merge range formulas: let $r''(x) = r(x) \wedge r'(x)$, then we obtain $r''(x) \wedge (r''(x) \wedge \varphi) \wedge r''(x) \wedge \psi)$. (When the second conjunct has only $r'(x)$, a simpler form is obtained.)
- A subformula $(r(x) \wedge \varphi) \ \vee \ (r'(x) \wedge \psi)$, is transformed to $(r(x) \ \vee \ r'(x)) \ \wedge \ (r(x) \wedge \varphi) \vee (r'(x) \wedge \psi)$, Note that if x appears in a disjunction and is restricted there, then it must be restricted in both disjuncts.

Note that we do not treat negation. As already stated, there are no "useful" range restrictions for a variable x inside a negated subformula, unless it is bound there, in which case n_x is below the quantifier prefix of that subformula. For essentially the same reason, we never have to go through an existential quantifier. Indeed, for each x, n_x is below the quantifier prefix of α, so we never have to go through that prefix. Other quantifier prefixes always occur just below negations, hence for such a prefix, either n_x is below it, or else it occurs under n_x in a negated formula and has no range restriction for x. Finally, we note that for the current case, that x is the last variable in the ordering, x is not used in a range restriction for another variable of the form (B5), hence we do not need to go through such a formula either.

To finish our transformation, we need to describe how a range restriction $r(x)$ that is associated with a subformula ψ that is marked x_f, is pushed down to its subformulas. We have the following cases:

- A formula $r(x) \wedge (\psi_1 \, \theta \, \psi_2)$ is transformed to $r(x) \ \wedge \ (r(x) \wedge \psi_1) \, \theta \, (r(x) \wedge \psi_2)$ where θ is $\wedge$ or $\vee$. Clearly, if, say, ψ_1 does not contain x, then r is pushed in only to ψ_2.

- A formula $r(x) \wedge \neg\psi$ is transformed to $r(x) \wedge \neg(r(x) \wedge \psi)$.
- A formula $r(x) \wedge \exists y(\varphi(y))$ is transformed to $r(x) \wedge \exists y(r(x) \wedge \varphi(y))$.

Here again, we do not need to push down through a formula of the form (B5), since x is not used there. We may need to push down through an existential quantifier.

The result of the transformation can be viewed as follows: Each subtree of the formula that contains only range formulas for x is now considered indivisible; its inner structure is of no further concern to us. Every other subformula below n_x, whether atomic or not, is now "anded" with a range formula for x (unless it does not contain x). It is convenient to carry this structure to the following stages. For example, a subformula $r(x) \wedge (\varphi \vee \psi)$ will be considered as a disjunction. But note that when we perform the transformation for y, it may appear in $r(x)$, since the range restriction for x uses it, or in φ or ψ. We consider y as appearing in the subformula if any of these three cases occurs. Further, and most important, when we associate a range formula $r'(y)$ for y with the subformula, we transform it to $r(x) \wedge r'(y) \wedge (\varphi \vee \psi)$. That is, we collect all range formulas associated with a node of the tree together. This is more economical than pushing the range formulas for y separately into each of them.

When the transformation above is applied to the next variables, there are some differences. First, when we mark nodes by y_r, going down from n_y, we may have to go into a subformula φ appearing in a range restriction of the form $x \subseteq \{z \mid \varphi(z)\}$. Another new case is that we may have a restriction $z \subseteq \{y \mid \varphi(y\}$. Since y precedes z in the ordering, when we treat y, we have already performed the transformation for z (possibly creating many occurrences of this range formula). Anyway, we start the transformation for y in φ, since n_y occurs there, and we do not need to consider (for y) anything outside this subformula.

Second, when going up, we may have to push a range formula out of a subformula of the form (B5). For example, in $\exists z \ldots \wedge x \subseteq \{y \mid \widehat{R}(z) \wedge y \in z\}$, we can push $\widehat{R}(z)$ outside, to obtain $\exists z \ldots \wedge \widehat{R}(z) \wedge x \subseteq \{y \mid \widehat{R}(z) \wedge y \in z\}$. Finally, when pushing down, we may have to push through a subformula of the form (B5). We use the equivalence of $r(x) \wedge z \subseteq \{y \mid \varphi\}$ and $r(x) \wedge z \subseteq \{y \mid r(x) \wedge \varphi\}$

Call a conjunction $\bigwedge r_i(x_i)$ of range formulas *closed*, if for each i, if r_i uses a variable y, then for some $j, y = x_j$. Since a range formula for a variable can only use variables that precede it, a closed conjunction can be ordered such that each conjunct uses only variables from preceding conjuncts. This implies, in particular, that at least the first conjunct has one of the forms (B1) - (B4) (or Boolean combinations thereof), and it uses no variables.

When the transformation terminates, we have a formula where each subformula is either a closed conjunction of range formulas, or the conjunction of some formula φ and a closed conjunction that contains range formulas for the free variables of φ. We now prove that it can be translated to an a-query, using induction on the number of variables in it.

First, we claim that given a closed conjunction of range formulas $\bigwedge r_i(x_i), i = 1, \ldots n$, we construct an equivalent a-query. This query produces a set of n-tuples. Recall that it is not a cross product. For each value of x_1 there is a set of values of x_2, for each pair of values for x_1, x_2 there is a set of values for x_3 and so on. That such an a-query exists is proved by induction on the number of variables in the conjunction. (Thus, the proof uses double induction: on the number of variables in the formula, and for a given number, on the number of variables in the range formulas.) The basis is the construction for the first variable, and we use the fact that its range restriction uses no variables. It has one of the forms (B1) - (B4), or combinations thereof that use only $\wedge$ and $\vee$. Since the combinations can be taken care of by intersection and union, respectively, we consider only the basic forms (and similarly for the other variables). For (B1), the a-query is $\widehat{R}$. For the other three cases, since t is a constant, the a-queries are $t, \{t\}$, and *powerset*(t), respectively. Note that the last case, since we are dealing with the first variable, is not a real use of *powerset*; since t is a constant set, its powerset is just another constant set. But, when used for variables that do depend on previous variables, it is a real use of *powerset*.

If the formula has just one variable, then this part is finished, and we skip to the second part of the proof. If there are more variables in the given closed conjunction, let the next one be x_i. We have three cases. For a restriction of the form (B1), the domain of x_i is independent of those of the other variables, its construction is as above, and we take its cross product with the a-query for the preceding variables. For restrictions of the form (B2) - (B4), we use the same proof as in Claim 6.6, where it is proved that if a term t contains variables $x_1, \ldots x_l$, then one can construct an a-query that produces an $(l+1)$-relation, where the first l columns contain values for the l variables, and the last column contains the corresponding value for t. (In our case, $l = i - 1$.) All we need to do is to apply some operator to the last column, in a *replace* operation, according to the form of the range restriction for x_i. For (B2) we use *set-collapse*, for (B3) the a-query we have is the one we need, and for (B4) we use *powerset* (and this is a real use of this operator).

The last case to consider is (B5), namely $x_i \subseteq \{y \mid \varphi(y)\}$. However, φ does not contain x_i, so it has a smaller number of variables than the formula we are considering. By induction hypothesis, there is an a-query for φ. We have to keep in mind that φ may contain other free variables except y. Let us first illustrate the construction by an example. Assume the query is $\exists v(\widehat{R}(v) \wedge x \subseteq \{y \mid y \in v\})$. After the transformation, the range formula for x will have the form $x \subseteq \{y \mid \widehat{R}(v) \wedge y \in v\}$. The formula φ contains the free variables v, y so the type of its result, that is also the type of the result of the corresponding a-query, is a set of pairs, of (v, y)-values. We need to transform it to have a set of y-values for each possible v-value, since each x-value must be included in one of these y-sets. Thus, we need to simulate a *nest* operator. We have seen how to do that in Section 5. Thus, we can obtain an a-query that will return a set of pairs of the form $\{[v, \{y \mid \varphi(v, y)\}]\}$. Since x is supposed to be a subset of one of the sets in the second column, we apply *powerset* to this column (i.e., using *powerset* inside a *replace*). Note that the domain expression for x also has a position for the parameter v. If we project out the first column, we loose the connection of each set in the second column to a v-value. To translate the full query, we have $\widehat{R}$ for the $\widehat{R}(v)$ part, we join it with the a-query we obtained for the range formula for x (and the variable v that it depends upon) on the common v-column, and finally we perform projection for the existential quantifier.

In the general case, if the free variables in φ, in addition to y are $v_1, \ldots v_j$, then the a-query produces a set of $(j+1)$-tuples, and we perform nesting on the last one, then apply *powerset* to it. These free variables must, however, appear somewhere else in our formula, and by the effect of the transformation, the subformula we are dealing with has the form $\bigwedge r_j(v_j) \wedge x \subseteq \{y \mid \varphi\}$, where each of the free variables of φ except y appears among the v_j's. We therefore join the result of this range formula for x_i with the a-queries for the domain of these variables (using appropriate equalities).

Given these a-queries, for the closed conjunctions of range formulas that appear in the given formula, we proceed to construct the a-query for the given formula using induction on its structure. For $\bigwedge r_i(x_i) \wedge \varphi$, where φ is atomic, we use the same construction as in the preceding section, except that we use the a-query of the closed conjunction as the domain expression. For $\wedge, \neg, \exists$, we use the same constructions as there, namely join, complement and projection. Note that we treat $r(x) \wedge (\varphi \wedge \psi)$ as a single construction, and similarly for $\vee, \neg, \exists$. Thus, when we have a negated subformula, since there is a closed conjunction of range formulas attached to it, that contains all the variables that are free in it, we can use complement. Finally note that we now also have $\vee$, which was not treated in the previous section. It is translated to union. (Each free variable that appears in one disjunct also appears in the other, because of safety.)

References

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

[2] Abiteboul, S., and C. Beeri, On the power of languages for the manipulation of complex objects. INRIA TR 846, May 1988.

[3] S. Abiteboul, C. Beeri, D. van Gucht, and M. Gyssens, An introduction to the completeness of languages for Complex Objects and Nested Relations, in "Nested Relations and Complex Objects", *Lecture Notes in Computer Science*, 361, *Springer-Verlag*, 1989.

[4] 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.

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

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

[7] Abiteboul S., and P.C. Kanellakis, Object Identity as a Query Language Primitive, proc. of ACM SIGMOD conf., 159–173, 1989, full paper to appear in *J. ACM*.

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

[9] 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)

[10] 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).

[11] Bancilhon F., and C. Delobel and P. Kanellakis, Object-Oriented Database System: the Story of O_2, Morgan Kaufmann San Mateo, California, 1992.

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

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

[14] Beeri, C., S. Naqvi, O. Shmueli, and S.Tsur, Set constructors in a logic data language, *Journal of Logic Programming* 10, 1991, 181-232.

[15] Beeri, C., and T. Milo, Functional and Predicative Database Programming . *Proc. 11th PODS* (March 1992).

[16] Breazu-Tannen, V., P. Buneman, and S. Naqvi, Structural recursion as a query language primitive. *Proc. 3rd DBPL workshop,* Napflion, Greece, August 1991, Morgan Kaufmann (1992).

[17] Breazu-Tannen, V., P. Buneman, and L. Wong, Naturally embedded query languages. ICDT (1992).

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

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

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

[21] Cluet, S., C. Delobel, C. Lecluse, and P. Richard, RELOOP, an algebra based query language for an object-oriented database system. *Data and Knowledge Engineering* 5 (1990), pp. 333-352.

[22] R.A. DiPaola, The Recursive Unsolvability of the Decision Problem for a Class of Definite Formulas, *Journal of the Association of Computing Machinery*, 16(2), 324–327, 1969.

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

[24] Fagin, R.. Horn clauses and database dependencies. jp JACM 29:4 (Oct. 1982).

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

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

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

[28] Hull, R., A Survey of theoretical research on typed complex database objects. manuscript, USC (1986).

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

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

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

[32] Jaeschke, B., and H.-J. Schek. Remarks on the Algebra on Non First Normal Form Relations", *Proc. 1st PODS* 124–138, 1982.

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

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

[35] Korth, H.F., M.A. Roth, and A. Silberschatz. Extended algebra and calculus for *not*NF relational databases. *TODS*.

[36] Kuper, G.M., Logic Programming with sets. *Proc.6th PODS* (1987)

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

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

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

[40] Naqvi, S.A., A Logic for negation in database systems. *Proc. Foundations of Deductive Databases and Logic Programming* (1986).

[41] 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).

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

[43] Peyton Jones S.L., *The implementation of functional programming languages*, Prentice Hall (1987).

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

[45] Trinder, P.W., Comprehensions: A query notation for DBPLs. *Proc. 3rd DBPL workshop,* Napflion, Greece, August 1991, Morgan Kaufmann (1992).

[46] Ullman, J.D., *Principles of Database Systems*, Computer Science Press, 2nd ed. (1982).

[47] M.Y. Vardi, The Decision Problem for Database Dependencies, *Information Processing Letter*, 12(5), 251–254, 1981.

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

[49] 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 (1988)

[50] Wadler, P., Comprehending monads. *Proc. Conf. on Lisp and Functional Programming*, 1990.

[51] Zaniolo, C., The representation and deductive retrieval of complex objects. *Proc.11th VLDB*, Stockholm (1985).