JOURNAL OF COMPUTER AND SYSTEM SCIENCES **54**, 3–33 (1997)
ARTICLE NO. SS971449

Catriel Beeri[†]

Department of Computer Science, The Hebrew University, Jerusalem 91904, Israel

and

Tova Milo[‡]

Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

Received May 8, 1996

In the last decade many extensions of the relational model were proposed, and basic properties of the relational model were investigated in their contexts. In particular, the equivalence of calculus and algebra, and the relative expressive power of other related languages were explored. This paper investigates this subject in a general framework, independently of any specific data type constructors that may exist in specific models, with the goal of making explicit the conditions that enable translation between query languages. The framework is based on a combination the well-founded approach of deductive programs and the initial algebra approach of algebraic specifications. The latter does not support negation (i.e., disequations); hence our combination contributes to the theory of algebraic specifications. Given the framework, we present the predicative and functional approaches to database description and querying. The first leads to the calculus and deductive approaches, the second to several algebras and, also, to generalizations that allow restricted definition by equations. We extend the notions of domain independence to our framework. We then present various sufficient conditions for the calculus and (some) algebra to be equivalent. We also compare the expressive power of algebras and more general languages to several deductive languages, under stratified and well-founded semantics. Finally, we define safety conditions and prove similar results for safe versions of the languages.

1. INTRODUCTION

In the last decade, the theory of the relational model has been generalized in many ways: to nested relations/complex object models that support the construction of complex values using the *tuple* and *set* constructors, to also allow other type constructors such as *list* and *bag*, to allow abstract data type (ADT) definitions, and to object-oriented data models. [3, 4, 6, 16, 18, 31, 36]. As was done for the relational model, much attention has been focused in this work on declarative languages for these models. The relational model supports three query language paradigms: algebra, calculus, and deductive. These have been extended to the more general models, either by extending operations, or by adding operations and predicates for dealing with sets (such as $x \in S$) and new algebraic operations such as *nest*, *unnest*, *powerset* [1, 9, 21, 28].

A lot of work has been done on analyzing the comparative expressive power of languages for the relational model; the most famous is Codd's result about the equivalence of the algebra and the calculus. Many of these results have been reexamined in the context of complex object models and object-oriented models; in particular, the algebra–calculus equivalence has been reproved for several such models [1, 13, 35, 25]. Although many recent works consider this issue, a general comparison of the three paradigms has not yet appeared.

In this paper, we present and compare languages based on the three paradigms; algebra, calculus, and deduction. We view each of these paradigms as a linguistic *frame*, that can be used with a variety of data models. Thus, rather than comparing these paradigms for one model (general as it may be), our aim is to investigate the fundamental relationships between them and to determine if and how these relationships depend on the particular features or properties of a specific data model and the type constructors it allows.

The framework for this investigation is obtained by combining the relational paradigm with the well known *algebraic specifications* paradigm. The combination allows one to investigate a variety of data models with rich data structuring facilities including, in particular, all those that use constructors such as *tuple* and *set*, or that allow ADT's. The combination also serves here to define the semantics of the languages we consider.

We now briefly describe the issues considered in the paper. First, the current algebraic specification framework

* This research was supported by a grant from the United States–Israel Binational Science Foundation (BSF), Jerusalem, Israel.

[†] E-mail: beeri@cs.huji.ac.il.

[‡] For correspondence contact Tova Milo, Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; E-mail: milo@math.tau.ac.il.

0022-0000/97 $25.00

allows one only to use positive facts (i.e., equations); when negation is used, the semantics of a specification is no longer well-defined [22, 26]. We argue that for our work negation needs to be considered, and we extend the algebraic specifications framework to include negation. This is of interest by itself, independently of the rest of the paper.

Most previous works investigating the relationship between algebra and deduction considered a rather restricted class of deductive programs—the **stratified** programs, and only algebras with recursion in the form of an explicit fixed point (or iteration) operation. It is of interest to extend the algebraic paradigm with a more general facility for recursion, and to investigate the relationship of the resulting language, which is a functional language, to general (not necessarily stratified) deduction. This is one of the subjects considered in this work.

Given our framework—a combination of the relational and algebraic specifications paradigms, we present two basic modes for defining databases and queries, namely predicative and functional. The calculus and deductive languages are naturally associated with the predicative mode and the algebraic language with the functional mode. Then we consider two main questions: (i) whether every database that can be defined using one mode can also be defined in the other mode?, and (ii) for a database that is definable in both ways, what is the relative expressive power of the various languages?

To answer these questions we extend the notions of *domain independence* and *safety* to our framework. We present various results on when these properties are satisfied by the languages and consider the relative expressive power of the languages. In particular we show the equivalence of the domain-independent calculus and a restricted version of the algebra. We also present results that relate the algebra, or the more general functional language, to the safe deductive language. In particular we show that using a restricted kind of recursion one can obtain an algebra with the same expressive power as stratified deduction, while a more general kind of recursion leads to the expressive power of general nonstratified deduction. Our results not only generalize many previous results, but also seem to clarify some of the fundamental requirements and restrictions needed to prove such equivalence results.

The outline of the paper is as follows: In Section 2, we present a brief summary of some of the main concepts of algebraic specifications and the extension needed to support negation. We explain how these concepts are used to specify data types. The predicative and functional approaches to database definition and query languages are presented in Section 3. In Section 4 we present several general algebraic languages. Domain independence, and the domain independent calculus are defined in Section 5. Sections 6 and 7 deal with the equivalence between the domain-independent calculus and the general algebras. The relation between the calculus, the domain independent calculus, and deduction is considered in Section 8. In Section 9, we introduce syntactic safety restrictions on the calculus and deduction, and we compare the power of the resulting languages to the algebras. Conclusions are presented in Section 10.

$\underline{\textbf{TUPLE}} = \textbf{nat} + \textbf{char} +$

sorts : *tuple*

opns : CREATE : $nat, char \rightarrow tuple$ (make tuple)
ATT$_1$: $tuple \rightarrow nat$ (1st attribute)
ATT$_2$: $tuple \rightarrow char$ (2nd attribute)

eqns : $x \in nat$, $y \in char$
ATT$_1$(CREATE$(x, y)) = x$
ATT$_2$(CREATE$(x, y)) = y$

FIG. 1. The specification **TUPLE**.

2. ALGEBRAIC SPECIFICATIONS OF ABSTRACT DATA TYPES

We adopt the algebraic specifications approach (with some extensions) as our formal framework. The algebraic specification approach uses equations for defining data types and their operations. There are other approaches to that, or to the definition of queries, or to query languages. For example, structural recursion is a powerful mechanism for defining functions. However, algebraic specifications have the advantages of being relatively simple and of having a well-defined semantics that is close to the semantics approaches used in the database area.[1] This section presents some of the main concepts of this approach [15, 19], and the extensions needed to support negation. We emphasize the concepts that are used in the sequel.

2.1. *Basic Concepts*

We start by considering the standard approach to specifications that does not use negation. A specification defines a collection of related data types. It contains sets of sort names and operations (i.e., functions) that define a language of many-sorted first-order predicate logic, with equality as the only predicate. Properties of the operations are stated as formulas, typically (conditional) equations.

DEFINITION 2.1. An **abstract data type specification** is a triple $SPEC = (S, OP, E)$ where S is a set of sort names, OP is a set of function symbols with arities in $S^* \rightarrow S$, and E is a set of (conditional) equations over S and OP.

Example specifications are shown in Figs. 1 and 2 and discussed below. The two specifications use only equations,

[1] On can view it as a disciplined way of using structural recursion.

SET = **data** + **bool** +

sorts : *set*

opns :

EMPTY : $\rightarrow$ *set*	(empty set)
INS : *data*, *set* $\rightarrow$ *set*	(insert)
MEM : *data*, *set* $\rightarrow$ *bool*	(member)
RM : *data*, *set* $\rightarrow$ *set*	(remove)

eqns : $d, d' \in data$, $s \in set$

$\mathrm{INS}(d, \mathrm{INS}(d, s)) = \mathrm{INS}(d, s)$	(idempotence)
$\mathrm{INS}(d, \mathrm{INS}(d', s)) = \mathrm{INS}(d', \mathrm{INS}(d, s))$	(commutativity)
$\mathrm{MEM}(d, \mathrm{EMPTY}) = F$	
$\mathrm{MEM}(d, \mathrm{INS}(d', s)) =$ if $EQ(d, d')$ then T else $\mathrm{MEM}(d, s)$	
$\mathrm{RM}(d, \mathrm{EMPTY}) = \mathrm{EMPTY}$	
$\mathrm{RM}(d, \mathrm{INS}(d', s)) =$ if $\mathrm{EQ}(d, d')$ then $\mathrm{RM}(d, s)$ else $\mathrm{INS}(d', \mathrm{RM}(d, s))$	

FIG. 2. The specification **SET**.

and not conditional equations. We shall see an example for a conditional equation later on.

Since the only predicate symbol use is equality, and the standard properties of equality are assumed, a model is a *many-sorted* algebra. A specification, being a set of sentences, has many models. This is a problem, since it is expected to specify a unique meaning. Note that the same problem exists in logic programming: a Horn-rule program, viewed as set of sentences, has many models. The solution is to choose the (unique) minimal model in the Herbrand universe as the semantics of the program. A similar solution is adopted for specifications: (the equivalence class of) one algebra is selected as its meaning. This is most often the *initial algebra* (see below), and we assume so in this paper. The similarity is more than just the choice of a unique semantics. The minimal model of a Horn-rule program is the initial model of the program in the class of all models.

An initial algebra in the class of algebras that are models of *SPEC* is an algebra $A_{SPEC} = (A_S, A_{OP})$, of signature (S, OP) such that there exists a unique homomorphism from it to each of the other algebras of *SPEC*. The desirable properties of the initial algebra are well documented [15, 19, 30]. In particular, whenever it exists it is unique (up to isomorphism). This follows from the fact that any two initial algebras are related by unique homomorphic mappings. For specifications given by (conditional) equations it exists, and, furthermore, its structure is also known. Let the Herbrand universe be the collection of ground terms over *OP*. It is also an algebra, as each of the operations is defined on it: for an operation f and term t, $f(t)$ is defined, as usual, to be the term $f(t)$. The extension of the equality predicate on the set of terms, as defined by the (conditional) equations in E and by its standard properties is an invariance relation on the domain of this algebra. Thus, the quotient of this algebra modulo the invariance relation, the **quotient term algebra**, is also an algebra: every element in this algebra is an equivalence class of ground terms. It can be shown to be an initial algebra. The existence of a homomorphism to any other algebra of *SPEC* is easily established, by induction on the structure of terms. Uniqueness follows from the fact that terms have not been made equal unless it so follows from E. For mode details see [15, 19]. Note the similarity to the construction of the *minimal model* used in logic programming. For algebras, we have here the case of a single predicate, namely equality, defined by conditional equations, and the sentences that define the standard properties of equality. The invariance relation defined by them is the minimal model of the equality predicate. It can similarly be shown for the minimal model in the Herbrand universe that it has a unique homomorphism to every other model.

We use an informal language [15] for writing specifications, as shown in the example in Fig. 1. The notation *nat* + *char* + $\cdots$ means that these previously defined specifications are imported and made part of this specification. The specification defines an algebra with sorts and operations for natural numbers, characters, and tuples consisting of a natural number and a character, with operations for creating tuples and selecting their attributes.

A very useful mechanism for defining standard types like tuples, sets, stacks, and trees, is the use of *parameters* in specifications. Parameterized specifications can be viewed as *type constructors* and have nice properties (see, e.g., [6, 15]). In order to specify a generic *tuple* type, we can replace *nat* and *char* in the above specification by type variables *data* 1 and *data* 2, that can later be instantiated by specific types as actual parameters. As another example, a generic *set* type is defined in Fig. 2. Given that the type **bool** with its operations is included, the **if**...**then**...**else** is a ternary parametric function of signature $\mathbf{bool} \times \mathbf{data}' \times \mathbf{data}' \rightarrow \mathbf{data}'$ in the specification for **bool**. The parametric specification of

bool is imported into the specification of sets, and *data'* is instantiated by bool.

Note that, in general, a specification for sets with element type *type* can contain the MEM "predicate," the conditional function and the RM operation, only if equality is definable on *type* [32]. Many other operations on sets can be defined using equations (some, e.g., union and difference, are considered later).

For brevity, we use in the rest of this paper the notations $x.i$, $[x_1, ..., x_n]$, $x \in s$, $\{x_1, ..., x_n\}$, for representing the expressions $\mathrm{ATT}_i(x)$, $\mathrm{CREATE}(x_1, ..., x_n)$, $\mathrm{MEM}(x, s) = TRUE$, and $\mathrm{INS}(x_1, ..., \mathrm{INS}(x_n, \mathrm{EMPTY}))$, respectively. We also use $[t_1, ..., t_k]$ to represent the type of tuples whose attributes are of types $t_1, ..., t_k$, and $\{t\}$ to represent the type of sets with members of type t.

As noted above, there is a close relationship between the semantics of algebraic specifications and logic programs. There is also, however, a fundamental difference between the two. Logic programs are based on the formalism of predicate calculus. The notion of a predicate is built into this formalism as follows: a language has an arbitrary collection of predicate names (determined by the user of the language); the atomic formulas have the form $p(x)$, where p is a predicate name and x is a tuple term. The notion of a model associates a set (a relation) with each predicate, and $p(x)$ is interpreted as stating that x is a member of the set corresponding to p. The contents of a predicate/relation can be defined by a Horn-clause program. The semantics of the program determines therefore the contents of certain sets. The semantics can be either the minimal model semantics or the "entails" semantics (restricted to ground facts); they are known to be equivalent for Horn-rule programs.

Figuratively, we may say that an external observer sees one box, containing a program and the "entails" relation, and another box, containing models of the program. The notions of being *true* or *false* are outside the boxes—they are part of the observer's universe, not values in the domains of the models. The semantics uses a default assumption (e.g., in the choice of the minimal model as the semantics), so a programmer must only specify the tuples that are in the relations, not those that are not in them. For the relational model, this is often referred to as *the closed world assumption*. The assumption applies to each predicate in the language. Thus, we have a powerful tool to define sets and their contents, and in the typical applications of predicate calculus and logic programs these are the only sets.

We make two observations on this approach. First, note that this default treats *true* differently from *false*. This is due to the fact that these are in the observer's universe, and she chooses to treat them differently. Also note that sets are not values of the domain (or one of the domains, for the many-sorted case), but are treated in a rather special way.

Now, a similar default assumption, namely that of using the initial model, is used in algebraic specifications. However, now we have only one predicate, namely equality. No general notion of predicate is built in, so we have to represent sets as values of a sort in a many-sorted algebra. That also includes sets that we might have represented as predicates in a predicate-based approach, e.g., database relations. However, not every predicate is conveniently represented as a set; sometimes we prefer to represent certain predicates as boolean-valued functions. A case in point is the membership predicate on sets.[2] To help with this task, the booleans are represented as the constants T, F of sort *bool* in the language and as corresponding values in the models. Then we can represent membership and similar predicates as boolean-valued functions, rather than as set values. Note that in terms of the observer above, the booleans have not been completely *internalized* (i.e., pushed into the object level). The observer still has her notion of truth, but this now applies only to the unique predicate that is allowed, namely equality. In particular, the powerful tool—the default assumption that extensions are determined in the minimal model—applies only to extension of equality.

It follows that to define properly a boolean-valued function like membership, it is not sufficient to provide the positive (i.e., *true*) facts, by stating when the function value is T. The negative (i.e., *false*) facts must also be provided, by equating the function value on certain arguments to F, for now T, F are just regular values, and the default mechanism does not apply. If we only have equations that tell us when the function value is equal to T, then in the initial algebra the function value on all other arguments will not be equal to T, but it will not be equal to F either; it will actually be a "new" boolean value. Indeed, recall that the domain for **bool** in the construction of the initial algebra is a set of equivalence classes of terms. If a ground term is not forced by the equations to be in the class of either T or F, then it will be in another class; hence there will be a boolean value that is neither T nor F. Such a situation is referred to as "the sort boolean is not protected," meaning that the booleans fail in this algebra to be what we expect them to be. Thus, to correctly express our intention for the membership predicate, a specification must define a total membership function.

Now, a total specification of membership can be accomplished when finite sets are defined; the specification in Fig. 2 completely determines both positive and negative membership for any finite set, since any such set can be obtained from the empty set by inserting its elements one at a time.[3] The algebraic specification-based approach may,

[2] As a rule of thumb, predicates that are part of the definition of a data type are treated as boolean-valued functions, while database relations are treated as sets.

[3] In other words, the EMPTY constant and the **insert** operation form a *complete presentation* for finite sets; such a presentation allows to define total functions on finite sets by structural recursion.

therefore, be adequate for relational databases with no function symbols, since there both the given and the defined relations are finite. But, in the more general setting of databases with richer structures, in particular in object-oriented models, one often wants to allow functions, and the possibility that logic programs or algebraic expressions define infinite sets must be considered. (A similar observation applies to other extensions of databases, e.g., databases with constraints [23].) Unfortunately, it is not all clear that one can specify infinite sets with a totally defined membership function using the algebraic specification approach.

A solution, using negation in the form of disequations, is presented below. We note that the restriction to positive equations only, adopted in the classical approach, is not without good reason; algebraic specifications are computable, whereas logic programs (and also specifications, as extended below) with negative facts may fail to be so.

2.2. *Negation*

The following example demonstrates how negation can be used in specifications, and in particular its role in defining infinite sets.

EXAMPLE 1. The specification **SET** defines only finite sets. Some infinite sets can also be defined using just equations. An infinite set S can be described by adding a constant S_c to the language, and equations for making this new constant denote S in the initial model. For example, the infinite set S^e of all even natural numbers can be described by stating that it contains every even number. We represent this infinite set using a new constant name s_c^e.

opns: $S_c^e\colon \rightarrow \{nat\}$

eqns: $S_c^e = \mathrm{INS}(2i, S_c^e)$

It is easy to see (using term rewriting) that for every $i \geqslant 1$ holds:

$$S_c^e = \mathrm{INS}(0, \mathrm{INS}(2, ..., \mathrm{INS}(2i, S_c^e), \ldots)).$$

It follows that every even number is "inserted" into S_c^e. Now, suppose we combine this specification with **SET** above and consider the initial algebra. We feel that we have a constant denoting the set of even numbers in this algebra. But in what sense can we say that S_c^e really represents the infinite set $S^e = \{0, 2, 4, \ldots\}$? At the least, we want it to *behave* like that set under certain operations. More formally, this can be stated as follows: Consider a mapping from the initial model of our specification to the universe of integers and "real" integer sets that maps the value corresponding to the given constant to the set we wanted to represent. First, we want to ascertain this is indeed a mapping, that is, it is not the case that a value in the initial model is denoted by two terms that we intend to represent two different sets and that this mapping is one-to-one. In our case, this is rather easy to prove. Two terms in the specification are made equal if and only if they both correspond to the same finite set, or both correspond to the set of even numbers. Next, we ask if this mapping is an isomorphism with respect to certain set operations? For example, suppose we want to check whether the number x belongs to the set S^e. Will $\mathrm{MEM}(x, S_c^e)$ return the correct answer, namely T is x is even and F if x is odd? For a finite set S, $\mathrm{MEM}(x, S)$ as defined in Fig. 2 defines a total boolean-valued function that equals T if x is in S, and F otherwise. But for the infinite set S_c^e MEM can be shown to equal T if x is even, but there is no derivation that produces equality to F for an odd number. (Naively, we may say the computation "does not terminate" because S_c^e is expanded to an infinite series of inserts and EMPTY is never encountered when it is scanned in the search for a member x.) This means, intuitively, that the membership testing function in Fig. 2 is partial; formally "boolean is not protected"—an undesirable situation.

To correct the specification, we add a statement that identifies the result of membership testing with F, whenever *it cannot be proved* to be equal to T:

$$\mathrm{MEM}(x, y) \neq T \rightarrow \mathrm{MEM}(x, y) = F. \qquad \text{(mem-F)}$$

This is a conditional equation with negation. However, the nature of the correction still remains to be made precise.

We will later consider other operations on sets, i.e., those of algebraic languages. We certainly want these operations to interact with those defined above for sets, particularly with membership, in the right way even for infinite sets; hence we take the above rule as a part of the specification of sets, additional to that given in Fig. 2. But, this formula uses a disequation—it uses negation. The standard initial model semantics cannot in general be used for specifications that contain negation, since the existence of an initial model is not guaranteed for such specifications [22, 26]. Thus, an alternative default mechanism for choosing the desired algebra must be provided. A similar problem exists in logic programs with negation—the existence of a unique minimal model is not guaranteed any more. The close relationship between the initial model semantics for data types specification and the minimal model approach for deductive programs indicates that possibly similar default mechanisms should be suitable for handling negation in both paradigms. There has been much recent work on semantics of deductive programs in the presence of negation in the rule bodies [12, 2, 33, 34, 17, 11]. Which of these can be used for our purpose?

Stratification is a syntactic restriction on programs that prevents a predicate from recursing negatively through itself [12, 2, 33]. A stratified program has a well-defined semantics given by taking least fixpoints of successively higher strata. But stratification cannot be used in our case, since a

specification contains only one predicate, namely equality, so any set of conditional equations with negation is not stratified. (Local stratification can sometimes be used, as explained later.)

It turns out that there are nonstratified programs for which an intuitive semantics can be discerned. This observation led to the development of other approaches to providing semantics for logic programs with negation that are not necessarily stratified; The **fixed point** semantics (e.g., inflationary), an essentially operational semantics, and the declarative **well-founded** [34] **stable-model** [17] and **valid** [11] semantics. The inflationary fixed-point semantics is not suitable for handling specifications with negation since it is not based on the idea that inequality should be used only when no more equalities can be derived. Thus, if we consider the disequation (mem-F) used in the definition of membership under inflationary semantics, it will be applied in the first step, when all sets are empty and will make all facts of the form $\mathrm{MEM}(x, S) = F$ true! The declarative semantics mentioned above, on the other hand, are based on the idea of using negation only as a "last resort," hence, seem more suitable for our problem. Of these three, the stable model semantics does not choose a unique model, whereas the well-founded and valid semantics are defined and unique for each program; hence we consider them only.[4]

In general the well-founded semantics is 3-valued: an atom may be true, false, or undefined. The same holds for the valid semantics. Also, if the value of an atom is defined in the well-founded semantics, it is defined and has the same value in the valid semantics. The latter may, however, assign truth values to atoms left undefined by the well-founded semantics. That is, it is a more powerful semantics. In particular, if the well-founded semantics has a two-valued model for a program, then the valid semantics agrees with it on the program [11]. For our development below, of the semantics of algebraic specifications with negation, both approaches can do equally well, and actually they agree on the specifications we need for the sequel. We use the well-founded semantics, as it is better known.[5]

We present below a brief summary of the well-founded model semantics, its properties, and how it is used for defining semantics for algebraic specifications with negation. (For a formal definition of the well-founded model semantics see [34]; for the use of negation in specifications see also [27].) The well-founded model of a deductive program P is a pair $(\mathscr{T}, \mathscr{F})$, representing a 3-valued model with $\mathscr{T}$ the set of true facts, $\mathscr{F}$ the set of false facts (guaranteed to be disjoint to $\mathscr{T}$), and the complement of $\mathscr{T} \cup \mathscr{F}$ in the Herbrand base as the set of undefined facts. The model is defined in stages, as follows: Initially, all the truth values of all facts are undefined: $\mathscr{T}_0 = \mathscr{F}_0 = \varnothing$. At a stage i, we have $\mathscr{T}_i$, $\mathscr{F}_i$. We first apply the rules of the program using positively (to satisfy positive body literals) only facts of $\mathscr{T}_i$, and using negatively (to satisfy negative literals) only facts in $\mathscr{F}_i$. The facts that are derived are considered to be certainly true and are added to $\mathscr{T}_i$ to form $\mathscr{T}_{i+1}$. Next, we consider all the possible derivations using the given program, starting from the current set $\mathscr{T}_{i+1}$, in which *all* facts not in $\mathscr{T}_{i+1}$ are allowed to be used negatively. Facts that are so derived are referred to as *possibly true*, and the set is easily seen to contain $\mathscr{T}_{i+1}$. The complement—the facts that are not derivable in any such derivation—are assumed to be certainly false, and they constitute $\mathscr{F}_{i+1}$. The process is repeated until no more certainly true and certainly false facts can be derived. It is guaranteed to produce sets $\mathscr{T}$, $\mathscr{F}$ that are disjoint. Note that a single stage may consist of an infinite fixpoint computation. Hence the "process" just described is not effective in general.

The following properties of the process and the sets generated in it are used in the sequel. The sequences $\mathscr{T}_i$, $\mathscr{F}_i$ are monotonically increasing to their respective limits $\mathscr{T}$, $\mathscr{F}$. From the description above, it can be seen that $\mathscr{F}_i$ is a function of $\mathscr{T}_i$ only. When $\mathscr{T}_i = \mathscr{T}_{i+1}$ then also $\mathscr{F}_i = \mathscr{F}_{i+1}$. The set of possibly true facts computed for stage i is the complement of $\mathscr{F}_{i+1}$; hence it contains not only $\mathscr{T}_{i+1}$ but also $\mathscr{T}$. Finally, note that the body of each rule of the program, under any substitution, is either true, false, or undefined in the well-founded model, and if it is true then so is the head (for otherwise the head should be added to the final $\mathscr{T}$). Thus, if the model is 2-valued, then it is a model in the standard sense of the program.

A specification *SPEC* can be viewed as a deductive program with "$=$" being the only predicate. Condition equations (and disequations) of *SPEC* are facts (i.e., rules with empty bodies); as seen in rule (mem-F), rules with nonempty bodies may also be used. Also, the standard equality axioms (transitivity, symmetry, reflexivity, and substitution) are rules. Taking a well-founded model approach, the deductive version of *SPEC* has a 3-valued interpretation, say $(\mathscr{T}, \mathscr{F})$. The facts in $\mathscr{T}$ represent pairs of terms that are equal, the facts in $\mathscr{F}$ are pairs of unequal terms, and the rest are equalities whose status is undefined. This is called the **well-founded interpretation** or **model** of *SPEC*. Note that $\mathscr{T}$ satisfies the axioms of equality, so is a congruence relation.

To illustrate the idea, consider the definition of S_c^e, and the membership test as defined in Fig. 2 with the additional rule (mem-F). The equations for MEM in Fig. 2 use no negation, and also no mention of F, so the facts they introduce into $\mathscr{T}$ do not rely on any fact being used negatively. One can easily see that for each even number x, the fact $\mathrm{MEM}(x, S_c^e) = T$ can be derived without using any fact negatively; hence it is in $\mathscr{T}$. Even after all such facts are put

[4] They are also known to be more general than all other known approaches that produce a unique model.

[5] In the preliminary version of the paper [7, 8] we used the valid semantics.

in $\mathscr{T}$, and using any fact not in $\mathscr{T}$ negatively, no fact of this form for any odd x can ever be derived. Hence all the facts $\mathrm{MEM}(x, S_c^e) = T$, where x is odd are put into $\mathscr{F}$ (meaning that certainly $\mathrm{MEM}(x, S_c^e) \neq T$). Similarly, no fact of the form $\mathrm{MEM}(x, S_c^e) = F$, where x is even, can be derived, so these are put into $\mathscr{F}$. Now, we can use the rule with negation to derive that for each odd number x the fact $\mathrm{MEM}(x, S_c^e) = F$ is in $\mathscr{T}$. These are the final values of the two sets (for MEM facts). Thus, for each x, precisely one of $\mathrm{MEM}(x, S_c^e) = T$, $\mathrm{MEM}(x, S_c^e) = F$ is in $\mathscr{T}$, and the other is in $\mathscr{F}$, so MEM is a total function in the well-founded interpretation.

It is easy to see that all equalities of set terms are determined by the rules in Fig. 2. No rules uses negation, so after these equalities are put into $\mathscr{T}$, all other equalities are put into $\mathscr{F}$. Essentially, for the part that uses no negation, the minimal model and the well-founded semantics agree. Note that we have shown that the well-founded interpretation is for this example 2-valued, rather than 3-valued, and that, in this example, negation with the well-founded semantics simulates the standard semantics of the part of the universe of sets we wanted to specify. We show later that this holds for more general cases.

Consider now the general case of specifications with disequations. For an algebraic approach, we need to associate an algebra with the interpretation. As in the case of the initial algebra, we expect such an algebra to be obtained as the quotient of the interpretation modulo some collection of equalities that forms a congruence relation. Naturally, $\mathscr{T}$ should be included in the congruence. That is, we consider the algebras that agree with the well-founded interpretation on the *true* facts to be the "more natural" candidates of being models of *SPEC*. As mentioned above, the set $\mathscr{T}$ in the well-founded interpretation is a congruence relation; hence it gives rise to a candidate algebra. But note that we also allow algebras, where equalities in $\mathscr{F}$ hold, as the falsity of equalities in $\mathscr{F}$ is not a consequence of the rules, but rather it is obtained by some default reasoning from the lack of derivations for these facts. The same approach is used in classical specifications; the models are the algebras that agree with the derivable set of *true* facts. An algebra so obtained is model of *SPEC* if it satisfies all its formulas. We call such algebras **well-founded**.

DEFINITION 2.2. An algebra A_{SPEC} is a **well-founded algebra** of *SPEC* iff it is a model of *SPEC*, and $\exp_1 = \exp_2 \in \mathscr{T}$ implies that $\exp_1 = \exp_2$ holds in A_{SPEC}.

Note that, as seen in Example 2 below, there exists a specification and a congruence relation that contains $\mathscr{T}$, actually $\mathscr{T}$ itself, that gives rise to a quotient algebra that is not a model of the specification.

Since, intuitively, a specification is supposed to specify a unique data type, a **default** mechanism must be provided to choose one well-founded algebra, if more than one exists. Following the traditional initial model approach, we choose the well-founded algebra such that there exists a unique homomorphism from it to each of the other well-founded algebras of *SPEC*. We call this algebra the **initial well-founded model** of *SPEC*.[6]

Note the difference between the traditional *initial model* and the *initial well-founded model*. An initial model must have a unique homomorphism to any of the algebras of *SPEC*, while an initial well-founded model must have unique homomorphisms only to the well-founded algebras. (When negation is not used, the concepts are equivalent.) This allows a well-founded initial model to exist in more cases. For example, in the specification of the set of even numbers above, for every number x, precisely one of $\mathrm{MEM}(x, S_c^e) = T$, $\mathrm{MEM}(x, S_c^e) = F$ holds. It is easy to see that for this specification the quotient term algebra defined by the well-founded interpretation is the initial well-founded algebra.

However, there exist specifications that do not have an initial well-founded model.

EXAMPLE 2. Consider *SPEC* defining a sort s with three constants a, b, and c, using the (generalized conditional) equations:

$$a \neq b \rightarrow a = c$$

$$a \neq c \rightarrow a = b.$$

Starting with $\mathscr{T}$, $\mathscr{F}$ both empty, and assuming that all facts not in $\mathscr{T}$ can be used negatively, we obtain possible derivations for $a = b$, $a = c$. Hence, none of these two facts is certainly false. Thus, we cannot derive any equality as being certainly true, i.e., in $\mathscr{T}$. That is, the well-founded model for the equality predicates is $\langle \mathscr{T} = \varnothing, \mathscr{F} = \varnothing \rangle$. Note that the quotient modulo $\mathscr{T}$ is an algebra that is not a model of *SPEC*. There are three well-founded algebras: an algebra, where $a = b = c$; an algebra, where $a = b \neq c$; and an algebra, where $a = c \neq b$. However, none of these is initial because they do not satisfy the condition of a unique homomorphism to the other two. Indeed, any homomorphism must map the element denoted by a in one algebra to the element denoted by it in the other, and similarly for b, c. It is easy to see that such a function between any two of the algebras does not exist. The symmetry in the two given conditional equations leads to the existence of two different, incompatible algebras (which can be viewed as obtained by nondeterministically choosing one of the two disequations to apply.)

Moreover, it turns out that deciding the existence of an initial well-founded model is undecidable.

[6] Thus, we are proposing a two-part mechanism, where the parts are the construction of the well-founded interpretation and the choice of the well-founded algebra.

PROPOSITION 2.3. (1) *It is undecidable whether a specification with negation has an initial well-founded model.*

(2) *If only* 0*-ary functions are used in the specification* (*i.e., only constants*), *then the problem becomes decidable.*

Proof. We first prove the first. The proof is rather simple and works by reduction to the problem of proving a ground equation from *SPEC*, known to be undecidable [15]. Given a specification $SPEC = (S, OP, E)$, where E does not contain negation, and a ground equation $\exp_1 = \exp_2$ of expressions of sort s_0, we construct a specification $SPEC'$ such that the equation is derivable from *SPEC* iff $SPEC'$ has an initial well-founded algebra.

$\underline{\mathbf{SPEC'}} = \mathbf{SPEC} +$

opns: $a: \rightarrow s_0$

eqns: $\exp_1 \neq \exp_2 \rightarrow \exp_1 = a$
$\exp_1 \neq a \rightarrow \exp_1 = \exp_2$

It is easy to see that if $\exp_1 = \exp_2$ follows from *SPEC*, then $SPEC'$ has an initial well-founded algebra where $\exp_1 = \exp_2 \neq a$, while if $\exp_1 = \exp_2$ does not follow from *SPEC* then $SPEC'$ has three different types of algebras: those where $\exp_1 = \exp_2 = a$, those where $\exp_1 = \exp_2 \neq a$, and those where $\exp_1 = a \neq \exp_2$. But none of these algebras is initial since there is no homomorphism from it to the algebras in the other two groups. It follows that $SPEC'$ has an initial well-founded algebra iff $\exp_1 = \exp_2$ is derivable from *SPEC*.

The decidability in the second part follows from the fact that the number of different terms in such specifications is finite. Thus, $\mathscr{T}$ and $\mathscr{F}$ can be computed in finite time, and so can all the possible well-founded algebras over the language of *SPEC*. Checking whether one of them is initial, can again be done in finite time. ∎

The (un)decidability result follows from the close relationship between logical implication (which is undecidable in general, but decidable for finite domains) and the initial well-founded algebra semantics.

The above result shows that the difficulty of writing correct specifications increases when negation is used. However, we need negation for defining correctly *sets*, and operations on them. Fortunately, it turns out that there exists a large and quite expressive family of specifications that are restricted in a way that assures the existence of initial well-founded model, and the specifications for sets, as we have presented so far and as extended in the sequel, are in this family. We call specifications that have an initial well-founded model **well-defined**.

PROPOSITION 2.4. *If the well-founded interpretation of SPEC is* 2*-valued then the algebra obtained by taking the quotient of the free algebra of SPEC over the congruence relation defined by* $\mathscr{T}$ *is an initial well-founded model of SPEC.*

Proof. We first show that the quotient algebra is a model of *SPEC*. Recall that if the well-founded interpretation is 2-valued, then it is a standard model of *SPEC*. It easily follows that the quotient algebra, where every two terms in $\mathscr{T}$ represent the same element and every two terms in $\mathscr{F}$ represent different elements, is also a model of *SPEC*. It is also well-founded, since the equal terms are exactly the pair of terms in $\mathscr{T}$.

Next, we show that the algebra is initial. We prove that there exists a unique homomorphism from the quotient algebra to any other well-founded model, by presenting the homomorphism and showing its uniqueness. The mapping is the same as the one presented in the literature for the case of specifications without negation [15]. It maps every constant/function in the quotient algebra to the corresponding constant/function in the target algebra. The mapping of more complex terms is defined inductively. The uniqueness proof is identical to the one in the classical case (for details see [15]). Essentially, the classical proof works since we use $\mathscr{T}$ rather than some superset as the congruence relation. ∎

We note that being well-defined is only a condition that allows us to show that a specification has an initial well-founded model. We still need to prove that this model is the one we intended to specify. In particular, all imported types have to be protected. For our specification of sets that means that **int**, **bool** have to be protected and that the algebra is isomorphic to the subalgebra of real sets over the integers containing the finite sets and the set of even numbers. We sketched some details of the proof above. It is quite easy to prove that **int** is protected. As MEM is total and only boolean-valued function, the only possibility for **bool** not to be protected is if $\text{MEM}(t, S_1) = T$ and $\text{MEM}(t, S_2) = F$ hold, and $S_1 = S_2$ is in $\mathscr{T}$. As we know what equalities hold between set terms, we can prove this does not occur. Note that if we add any equality, we make two terms that correspond to distinct sets equal and, via MEM, that will destroy **bool**. Thus, the initial well-founded model is the only well-founded algebra of this specification that protects **bool**.

2.3. *Predicates*

The specification approach presented above can be extended by adding a set P of typed predicate names and using generalized Horn-clauses instead of simple (conditional) equations. Thus, it allows a full language of the predicate calculus. Each clause has the form $Q_1, ..., Q_n \rightarrow Q_0$, where Q_i is an atomic formula of the form $R(x)$, or an equation $\exp_1 = \exp_2$. The structures of this language consist of domains with operations defined on them and of relations that are restricted by signatures specified for them in the language. For a given set of formulas, a unique model

can be selected by using a default mechanism. The simplest approach is to take the minimal model, as for logic programs. Now, restricting attention to the terms and the equalities on them in this model, one takes the quotient to obtain an algebra. It is easy to show that the predicates are well defined in this quotient. This gives the initial model of the specification. This generalizes both the minimal model approach to logic programs and the initial algebra approach to specification of data types, and allows us to specify simultaneously a collection of data types, namely an algebra and a database on top.

Allowing us to use negation in rule bodies, the **well-founded initial model** in this extended framework generalizes both the initial well-founded model approach to equational specification and the deductive well-founded model approach. In particular, when there are no predicates it is exactly the initial well-founded model and, when there are no equations, it is the well-founded model in the Herbrand universe. In the special (but probably more practical) case that the formulas consist of two separate parts, one for defining data types and the other for defining the predicates, then the initial well-founded model consist of the initial algebra of the types specification and, on top, the well-founded model of the predicates [5].

3. DATABASES AND QUERIES

Let us now consider how a database, and how queries on a database can be specified. To define a data base we need to provide a specification for the underlying data types, and additionally, define the database contents. Query specification uses a similar framework. There are basically two approaches to database and query definition:

3.1. *Predicative Style Definition and Languages*

The first approach is to use predicates. It is the approach commonly assumed in the literature for database specification. For example, a relational database is a collection of formulas in a many-sorted language that contains the following: (i) A specification of certain atomic types (usually called "built-in"), and a specification of certain tuples types over these atomic types (these are the tuples used in relations); (ii) a specification of the database contents in the form of a collection of ground atomic formulas of the form $R_i(t)$, where R_i is a predicate symbol in a given collection of names of database relations. The semantics, the initial model, has the initial algebra of (i) as the domain (with tuple construction and attribute selection operations) and additionally has relations that satisfy the formulas of (ii). In general, a database definition will contain one part (denoted by *SPEC*) that specifies atomic and composite domains, with their operations, and another part (denoted by *DB*) consisting of ground atomic formulas, the database contents.[7] A more elaborate approach to content specification is to use Horn-clauses instead of simple ground formulas. In either case, the database content is normally defined assuming as given the sorts and operations of *SPEC*. Different instances of a given database schema have the same types of specification and different database content specifications. We assume that the data types used in the database are well defined; i.e., their specification has an initial well-founded model. Since our formalism allows us to define domains of sets, or domains of arbitrary ADTs, the nested relations/complex object models and models that allow attribute values to be arbitrary ADTs are special cases.

The query languages for such databases use predicates; these are the (predicate) calculus and various deductive languages. A **calculus query** has the form $q = \{x \mid \phi(x)\}$, where ϕ is a first-order formula, whose atomic formulas are of the form $R_i(x_j)$, and $\exp_1 = \exp_2$.[8] The R_i's denote the database relations. The meaning of a query is defined in the standard way: each R_i is instantiated by the actual database relation, and the formula is then evaluated. The scopes of the quantifiers are the domains of the sorts in the underlying algebra. The result contains all the elements that satisfy the formula. A **deductive query** consists of a set of rules of the form $Q_1, ..., Q_n \rightarrow R_i(x)$, where Q_j is an atomic formula ($R_i(x_j)$, $\exp_1 = \exp_2$) or a negated atomic formula, and a query of the form "$R(x)$?." The answer is obtained by computing a model (e.g., the well-founded model) of the program over the given database.[9] In special cases the answer can be computed by simpler methods. E.g., if the program is stratified, then the answer can obtained by successively computing the minimal model of each stratum, and if it does not contain negation then it suffices to compute the minimal model.

3.2. *Functional Style Definition and Languages*

The alternative approach is to assume that the database contents is a finite collection of named sets. In this approach no predicates (except equality) are used. The sets are values, represented by named constants, and their contents are specified by (conditional) equations, or more generally by rules with negation that involve only equalities. For example, a database relation R_i that contains elements $a_1, ..., a_n$ of type t_i can be represented by a constant $R_i^a: \rightarrow \{t_i\}$ that is defined using the equation $R_i^a = \{a_1, ..., a_n\}$. In general, equations of the form $R_i^a = \exp$ can be used for defining

[7] In practice underlying domains and their operations are not specified but rather are assumed to be provided by the hardware and basic software. Also, tuples are not specified, but rather implemented by the database software.

[8] Note that $R_i(\exp)$ can be described by $R_i(x_j \wedge x_j = \exp)$, so there is no loss of generality.

[9] An alternative is to compute the answer by importing *SPEC* and DB, and computing the initial well-founded model of the resulting program.

the contents of the relations. Again, we assume that the relations are well defined, i.e., that their specifications have an initial well-founded model.

The appropriate query language is a **functional** language. The query result is represented by a constant Q that is defined using an equation of the form $Q = \text{exp}$. The query specification additionally includes definitions for all the types and functions used in exp, that are not defined in *SPEC* and *DB*, and is required to have an initial well-founded model. When a functional query is applied to a database, each relation name R_i^a is instantiated by the actual database relation and the query is then evaluated. This has the same affect as importing the data types and the database contents specifications and computing the initial well-founded model of the combined specification.

In database practice and research so far, more restricted languages have been the focus of attention. The restriction consists of allowing to use in exp above only a specific set of predefined operators $\widehat{OP}$ (mostly operations on sets). For example, in the relational algebra, queries are defined only using the operators *select*, *project*, *join*, etc. We call such a set of operations the $\widehat{OP}$ **algebra** and the queries (databases) defined using the operators the $\widehat{OP}$**-algebra queries (databases)**. The $\widehat{OP}$-algebra is a restricted version of the general functional approach since the user is not allowed to define arbitrary new functions, but rather uses only the predefined operations. It is nevertheless a functional language.

3.3. *On the Two Paradigms*

We have seen that two styles can be used for database definition and for query languages, namely predicative and functional. The first question that arises is whether every database can be expressed in both paradigms. More specifically, can a database that can be defined in one of these styles also be defined in the other? The answer may depend of course on the specific language (including its semantics) used to implement each paradigm. For example, it is clear that the two styles are equivalent for database definition when database contents is specified by enumeration: If predicates are defined by ground atomic formulas $R_i(a_1)$, ..., $R_i(a_n)$ (as in the relational model), they can be represented by sets using equations $R_i^a = \{a_1, ..., a_n\}$, and vice versa. The problem is more difficult when databases can be defined by more general programs. As most research has dealt with relational databases, this issue has not received attention so far.

Another important question is whether for a database that is definable in both ways we can express the same queries in both paradigms. We note that understanding the relative power of query languages in terms of the queries they can express can also shed light on their relative power to define database contents. Hence, our investigation of the two problems is intertwined.

These two questions are a primary subject of this work.

Let us briefly consider how the two paradigms were used in the relational and nested relational models. In the relational model, the calculus is purely predicative; i.e., all given and defined sets are treated as predicates, and the algebra is purely functional and treats all sets as values. Codd's major result states the equivalence of the two approaches. In the nested relational model, the calculus [1, 20] treats outermost sets as predicates, and nested sets as values. However, keeping the predicative spirit, only the membership operation is provided for these values. The ability to generate new set values is provided by having variables of set types, whose values are determined by logical formulas. The algebras, such as the one in [1], treat all sets as values and provide a set of operations for the manipulation and creation of set values. In this model also, the basic equivalence result holds in principle, except that the algebra has two versions of inequivalent power and one has to distinguish two corresponding calculus languages.

4. ALGEBRAS AND GENERALIZATIONS

In this section we present an algebra that generalizes the relational and the complex object algebras and show that the operations in it can be specified using the approach of initial well-founded algebra. Actually, we present several algebras, not one, and also functional languages that generalize them. These are the functional languages that are investigated in this work.

We use parameterized specifications for describing generic operators and concentrate on three kinds of operations:

(i) **manipulation** operations (like $\cup$, $\times$, $-$) that manipulate sets without changing or looking at their elements (except for testing equality);

(ii) **filters** that iterate over one set and change its members or test if certain conditions are satisfied, including:

(a) **local** filters that operate on each element separately, including the operation σ_{test} that selects set elements using some boolean-valued *test*, a $\mathbf{MAP}_f$ operator that restructures each element in a set using some restructuring function f, and a general UNNEST operation;

(b) **semi-local** filters like NEST_{f_1, f_2} that process together several set elements;

(c) **global** filters, like the general aggregator $\text{AGGR}_{g, h, \text{unit}}$ that may use the whole set for producing the result;[10]

(iii) a **fixed point** operator IFP_{exp} that computes the fixed point of an expression exp constructed from the above operators.

[10] This classification is presented for intuition only.

The important point is that these operations can all be specified in the algebraic specification style, using the initial well-founded algebra semantics; thus the algebras based on them [21, 9, 28] are covered by our framework. We start by presenting full specifications of the above operations (followed by a short explanation) and then investigate their properties. In particular we consider the question whether the operations are well defined, i.e., whether the specification has an initial well-founded model. Our specification includes Fig. 2, and the additional rule (mem-F) for MEM; below we only provide the additional operations and their properties.

4.1. *The Manipulation and Filter Operations*

The specifications for the first two kinds of operations are given in Figs. 3 and 4.

Remarks. • Note that the selection criteria *test* used in σ_{test} must be a boolean valued function; i.e., for every element a of the domain test(a) must equal T or F.

• The UNNEST operator operates on a set of tuples where one of the attributes is of type *set* and replaces every original tuple $[a_1, ..., a_{n-1}, \{a_n^1 \cdots a_n^k\}]$ by the tuples $[a_1, ..., a_n^1], ..., [a_1, ..., a_n^k]$. Similar specifications can be used for decomposing tuples having attributes of other complex types, such as trees and stacks.

• The GROUP operator gets as input an element a and a set s, collects all the elements of s having the same result for f_1 as a, and restructures them using f_2. NEST replaces each set member a by a tuple where the first attribute contains $f_1(a)$, and the second attribute contains the group of all elements having the same result for f_1 (restructured by f_2). For defining the standard nesting operation on tuples, we can use $f_1(x) = [x.1, ..., x.n-1]$, and a restructuring function $f_2(x) = x.n$.

• The operator $\text{AGGR}_{\text{unit}, g, h}$ iterates over the elements of the set in some arbitrary order. It applies the function h on each element and aggregates the results using the function h. Note that the specification of the operator is correct only if the function g is commutative. An aggregator that uses noncommutative g can be defined if the members of the sets are ordered [21].

• The above operators are generic and can operate on sets with arbitrary element types (providing that equality is definable on the type of the elements of the sets). However, although it seems that the operators are also generic in the functions *test*, *f*, *g*, *h*, and *unit*, this is not really the case. Our framework is strictly first order, and function variables are **not** available, thus separate specifications must be provided for different functions. The definitions above can be viewed as the macro definition, where the actual function needs to be substituted to obtain a concrete specification for a given function. For the work in this paper, this weak form of genericity is sufficient.

For example, $\sigma_{EQ(x.1, 8)}$ can be used to denote selecting tuples where the first attribute equals 8, $\text{MAP}_{[x.1, x.2]}$ denotes projection of the first and the second attributes, and $\text{AGGR}_{(+, \text{id}, 0)}$ denotes the *sum* function.

4.2. *The Fixed-Point Operator*

In addition to the above operators, we also consider a generic fixed point operator IFP. Let $x_0, x_1, ..., x_n$ be variables of set types $\{s_0\}, \{s_1\}, ..., \{s_n\}$, respectively, and

opns :	$\cup : \{s_1\}, \{s_1\} \to \{s_1\}$	(union)
	$\times : \{s_1\}, \{s_2\} \to \{[s_1, s_2]\}$	(cartesian product)
	$- : \{s_1\}, \{s_1\} \to \{s_1\}$	(subtraction)
	$\sigma_{test} : \{s_1\} \to \{s_1\}$	(selection)
	$\text{MAP}_f : \{s_1\} \to \{s_2\}$	(restructuring)
	$\text{UNNEST} : \{[s_1, \ldots, s_{n-1}, \{s_n\}]\} \to \{[s_1, \ldots, s_n]\}$	(un-nesting)
	$\text{GROUP}_{f_1, f_2} : s, \{s\} \to \{s_2\}$ $\text{NEST} : \{s\} \to \{[s_1, \{s_2\}]\}$	(nesting)
	$\text{AGGR}_{g,h,unit} : \{s_1\} \to s_2$	(aggregation)

FIG. 3. Manipulation and filter operations.

eqns :

$$x \cup y = y \cup x$$
$$\text{EMPTY} \cup y = y$$
$$\text{INS}(a, x) \cup y = \text{INS}(a, (x \cup y))$$
$$---$$
$$x \times y = y \times x$$
$$\text{EMPTY} \times x = \text{EMPTY}$$
$$\text{INS}(a, x) \times \text{INS}(b, y) = \text{INS}([a, b], ((x \times \text{INS}(b, y)) \cup (\text{INS}(a, x) \times y)))$$
$$---$$
$$\text{EMPTY} - y = \text{EMPTY}$$
$$\text{INS}(a, x) - y = \text{if MEM}(a, y) \text{ then } x - y \text{ else INS}(a, (x - y))$$
$$---$$
$$\sigma_{test}(\text{EMPTY}) = \text{EMPTY}$$
$$\sigma_{test}(\text{INS}(a, x)) = \text{if } test(a) \text{ then INS}(a, \sigma_{test}(x)) \text{ else } \sigma_{test}(x)$$
$$---$$
$$\text{MAP}_f(\text{EMPTY}) = \text{EMPTY}$$
$$\text{MAP}_f(\text{INS}(x, s)) = \text{INS}(f(x), \text{MAP}_f(s))$$
$$---$$
$$\text{UNNEST}(\text{EMPTY}) = \text{EMPTY}$$
$$\text{UNNEST}(\text{INS}([x_1, \ldots, x_{n-1}, \text{EMPTY}], y)) = \text{UNNEST}(y)$$
$$\text{UNNEST}(\text{INS}([x_1, \ldots, x_{n-1}, \text{INS}(x_n, x)], y)) = \text{INS}([x_1, \ldots, x_{n-1}, x_n], \text{UNNEST}(\text{INS}([x_1, \ldots, x_{n-1}, x], y))$$
$$---$$
$$\text{GROUP}_{f_1,f_2}(a, \text{EMPTY}) = \text{EMPTY}$$
$$\text{GROUP}_{f_1,f_2}(a, \text{INS}(b, x)) = \text{if EQ}(f_1(a), f_1(b)) \text{ then INS}(f_2(b), \text{GROUP}_{f_1,f_2}(a, x)) \text{ else GROUP}_{f_1,f_2}(a, x)$$
$$\text{NEST}_{f_1,f_2}(\text{EMPTY}) = \text{EMPTY}$$
$$\text{NEST}_{f_1,f_2}(\text{INS}(a, x)) = \text{INS}([f_1(a), \text{GROUP}_{f_1,f_2}(a, \text{INS}(a, x))], \text{NEST}_{f_1,f_2}(x))$$
$$---$$
$$\text{AGGR}_{g,h,unit}(\text{EMPTY}) = unit$$
$$\text{AGGR}_{g,h,unit}(\text{INS}(a, x)) = g(h(a), \text{AGGR}_{g,h,unit}(\text{RM}(a, x)))$$

FIG. 4. Equations for manipulation and filter operations.

let $\text{exp}(x_0, x_1, ..., x_n)$ be some algebra expression of type $\{s_0\}$. We use the notation $\text{exp}(x_0, x_1, ..., x_n)$: $\{s_0\}, \{s_1\}, ..., \{s_n\} \to \{s_0\}$, to denote that fact that for every instantiation $R_0, R_1, ..., R_n$ for the variables $x_0, x_1, ..., x_n$, $\text{exp}(R_0, R_1, ..., R_n)$ is a set of type $\{s_0\}$. The operator $\text{IFP}_{\text{exp}, x_0}$: $\{s_1\}, ..., \{s_n\} \to \{s_0\}$, computes the (inflationary) fixed point of exp with respect to x_0, i.e., for every instantiation $R_1, ..., R_n$ for the variables $x_1, ..., x_n$, respectively, the fixed point is computed by first instantiating x_0 by the empty set EMPTY, applying exp on EMPTY, $R_1, ..., R_n$, and then successively at each step instantiating x_0 by the set obtained in the previous step, applying exp, and accumulating the

$$F_{exp,x_0}(0, x_1, \ldots, x_n) = \text{EMPTY}$$
$$F_{exp,x_0}(\text{SUCC}(i), x_1, \ldots, x_n) = F_{exp,x_0}(i, x_1, \ldots, x_n) \cup exp(F_{exp,x_0}(i, x_1, \ldots, x_n), x_1, \ldots, x_n)$$
$$\text{IFP}'_{exp,x_0}(i, x_1, \ldots, x_n) = F_{exp,x_0}(i, x_1, \ldots, x_n) \cup \text{IFP}'_{exp,x_0}(\text{SUCC}(i), x_1, \ldots, x_n)$$
$$\text{IFP}_{exp,x_0}(x_1, \ldots, x_n) = IFP'_{exp,x_0}(0, x_1, \ldots, x_n)$$
$$F_{exp,x_0}(i, x_1, \ldots, x_n) = F_{exp,x_0}(\text{SUCC}(i), x_1, \ldots, x_n) \to \text{IFP}'_{exp,x_0}(0, x_1, \ldots, x_n) = F_{exp,x_0}(i, x_1, \ldots, x_n)$$

FIG. 5. The IFP operation.

results. The mechanism for defining the IFP operator is given in Fig. 5. We first define an auxiliary function $F_{\exp, x_0}$: nat, $\{s_1\}, ..., \{s_n\} \to \{s_0\}$, that for any given i and any instantiation for $x_1, ..., x_n$, computes the result of i successive applications of exp. Then, the limit of the monotonic sequence $F_{\exp, x_0}(i, ...)$, $F_{\exp, x_0}(i+1, ...)$, ... (i.e., the fixed point of exp) is obtained by taking the union of all the sets in the sequence. For this we define an auxiliary function IFP′ whose value for all i is that limit. (Observe that is defining equation is not a recursive definition.)

Note that the fixpoint operation may produce finite sets; hence the extra rule (mem-F) for MEM is indeed needed.

Finally, since in some cases the result of applying the IFP operation is not an infinite set and we would like to use the regular definition of membership, we also add the last conditional equation that assures that whenever exp has a finite limit, this limit is made equal to a "real" **finite** set, rather then being represented by an infinite union.

For brevity, whenever the same of the variable x_0 is clear from the context (for example, if exp has only one variable) we shall use $\mathrm{IFP}_{\exp}$ to denote $\mathrm{IFP}_{\exp, x_0}$.

4.3. *Mutual Fixed Points*

So far we considered only the fixed point of a single expression. However, we can also express in our language **mutual fixed points** of several expressions. Let $\exp_j(x_0^1, ..., x_0^n, x_1, ..., x_k)$, $j = 1 \cdots n$, be algebra expressions of arity $\{s_0^1\}, ..., \{s_0^n\}, \{s_1\}, ..., \{s_k\} \to \{s_0^j\}$, respectively. For every instantiation $R_1, ..., R_k$ for $x_1, ..., x_k$ their mutual fixed points are obtained as follows: First instantiate $x_0^1, ..., x_0^n$ to the empty sets. Then successively at each step apply the expressions on the current values of the variables; then instantiate each x_0^j by adding the result returned by $\exp_j$. The final result is obtained by accumulating the results. The reason we can express mutual fixed points is that we have the product ($\times$) and projection ($\mathrm{MAP}_{x.j}$) operations. Thus, we can "pack" n set arguments into a set of n-tuples using the product, simulate the mutual fixed point by a single fixed point (using projection to access components o the tuples), and finally use projection to decompose the final result. Specifically, we use the expression $\mathrm{IFP}_{\exp, x_0}(x_1, ..., x_k)$, where x_0 is of type $\{[s_0^1, ..., s_0^n]\}$, and

$$\exp = \exp_1([\Pi_1(x_0), ..., \Pi_n(x_0), x_1, ..., x_k]) \times \cdots \times \exp_n([\Pi_1(x_0), ..., \Pi_n(x_0), x_1, ..., x_k]).^{11}$$

The variable x_0, with respect to which the fixed point is computed, represents the cartesian product of the variables $x_0^1, ..., x_0^n$. We denote the jth component of (mutual) fixed point by $\mathrm{IFP}^n_{\exp_j}$. To extract this component from the above cartesian product we have to project the jth attribute of the result. Thus

$$\mathrm{IFP}^n_{\exp_j}(x_1, ..., x_k) = \Pi_j(\mathrm{IFP}_{\exp, x_0}(x_1, ..., x_k)).$$

The IFP operator, and in particular mutual fixed point computation, is very useful for expressing recursive computations. We shall consider its properties in the sequel and show how it can be used for defining computations similar to those expressed by deductive programs.

EXAMPLE 3. Given a specification with sorts $s_1, ..., s_n$, the fixed point operator can be used for describing the (possibly infinite) sets $\mathrm{dom}(s_i)$, $i = 1 \cdots n$, that contain all and only the elements of type s_i, respectively, that is, the domains of the sorts:

$$\exp_i(x_0^1, ..., x_0^n) = \{c \mid c \text{ is a constant function of arity: } \to x_i\} \cup (\bigcup \{\mathrm{MAP}_{f_j}(x_0^{j_1} \times \cdots \times x_0^{j_k}) \mid f_j \text{ is a function of arity } s_{j_1}, ..., s_{j_k} \to s_i\}).$$

$\mathrm{IFP}^n_{\exp_i}$ then represents in the initial well-founded model the (possibly infinite) set $\mathrm{dom}(s_i)$. Note that $\exp_j$ does not contain any variables other then the fixed point variables $x_0^1, ..., x_0^n$. Thus $\mathrm{IFP}^n_{\exp_i}$ is of arity: $\to \{s_i\}$; i.e., it is a *set constant*.

In the following we consider an algebra that does not use IFP, but which contains constants that denote the domains of the sorts. As just shown, this can be viewed as a restricted special use of IFP.

4.4. *Correctness*

It is important to note that, while many works assume a finite database and no functions, we allow functions on the domains, such as addition n numbers; hence the fixed point operator may generate infinite sets. Thus, as already explained, negation needs to be used to correctly define the membership function.

As stated in Section 2, we use the well-founded model approach to define the semantic of the specifications. (Although IFP is an inflationary fixed point operation, using the inflationary fixed point for the semantics of specifications is wrong, since the disequation (mem-F) would then be applied in the first step.) A natural question is whether the above operations, and in particular the fixed-point operator are well defined and correct, i.e., whether the specification of the operations has an initial well-founded model, and if so, does this model reflect the intended universe and its operations. To understand the problem note that the definition of the difference operation uses

[11] For clarity, we use the projection symbol Π_i to denote $\mathrm{MAP}_{x.i}$.

membership. Thus, we need to provide semantics for expressions that may involve several nested applications of difference and IFP and for applications of MEM that involve such expressions. Obviously, the disequation (mem-F) is now used, not only in the last step of a well-founded computation, but in several distinct steps. The well-founded model of the deductive version of a specification still exists and is unique, since it has these properties for all logic programs, but the questions of whether it gives rise to an initial well-founded algebra and whether it is the desired algebra arises.

THEOREM 4.1. *Let $\mathscr{A}$ be an algebra with the (imported) sorts $t_1, ..., t_n$ (with equality) including* **bool**, *and let SPEC be a specification that extends $\mathscr{A}$ that for every type t_i defines a set type $\{t_i\}$, with the operations* EMPTY, INS, RM, MEM, $\cup$, $\times$, $-$, σ, MAP, UNNEST, NEST, AGGR, IFP, IFP′, F, *as defined in Figs.* 2, 3, 4, 5, *and in disequation* (mem-F). *Then the deductive version of SPEC has a* 2*-valued well-founded model*; *hence SPEC is well-defined—it has an initial well-founded algebra. Furthermore*, *this algebra protects the imported types*, *particularly* **bool**, *and is isomorphic to a subalgebra of real sets over the given types.*[12]

The proof of the above theorem is rather technical and not essential for understanding the rest of the paper. It is therefore deferred to the Appendix.

4.5. *The Functional Languages*

We now define the functional languages of interest to us.

4.5.1. *Algebras.* We consider the following algebras.

DEFINITION 4.2. The **IFP-algebra** is the consisting the operations $\cup$, $-$, $\times$, σ, MAP and IFP. The ***S*-algebra** is a restricted version of the IFP-algebra, where the IFP operator is used only to construct sets $\mathrm{dom}(s_i)$ representing the sort domains (and showed in Example 3) and nowhere else. The **algebra** does not contain the IFP operator at all (i.e., not even the domain constants).

Note that the operators UNNEST, NEST, and AGGR are not included in these algebras. We shall consider their properties separately.

4.5.2. *Algebras with Equations.* The above algebraic paradigm can be enriched by allowing the programmer to add new operation names to the language and to use equations to define their properties. However, if no restrictions are posed, then the resulting language is a very general functional language that allows defining any specification. We restrict the language as follows: Given one of the algebras, we allow augmenting it by a set of new operations, but we allow only new operations with input and output parameters of *set* type. To define n new operations, $f_1, ..., f_n$, one writes n equations of the form $f_i(x_1, ..., x_n) = \exp(x_1, ..., x_n)$, where exp is an algebraic expression that contains no variables other than $x_1, ..., x_n$ and uses the algebraic operations of the algebra and, possibly, also some or all of $f_1, ..., f_n$. For a given algebra (i.e., set of the operations) this can be viewed as an extension of the algebra with recursion, although actually it is somewhat more general since the equations are not restricted in any way; i.e., they are not required to be recursion equations. We will refer to such a language as the extension of the algebra with recursion. The restricted framework allows one to define new operations only in terms of a specific set of predefined operators.

Note that this language is not an extension of the algebra with structural recursion. Indeed, a typical use of the latter is to define a function on (finite) sets by structure on their contents. The definition of MEM for finite sets has this form. Our restrictions on equations rule out such definitions.

EXAMPLE 4. The operator $\cap: \{t\}, \{t\} \to \{t\}$ (*intersection*) can be defined using the operator $-$ and the equation $x \cap y = x - (x - y)$. Similarly, an *exclusive-or* operator $\otimes: \{t\}, \{t\} \to \{t\}$ can be defined with the equation $x \otimes y = (x - y) \cup (y - x)$. These do not use the defined operation in the right-hand side of the equation. As an example of a recursive definition, given a binary relation R, we may define its transitive closure by $T = T \cup (T \circ R)$, where $\circ$ is a suitable combination of product, selection, and projection. Finally, recall the definitions of S_c^e. The definition uses an integer variable i which is not of *set* type; hence it is not allowed in our language. Another possible definition for this constant is

$$S_c^e = \{0\} \cup \mathrm{MAP}_{+2}(S_c^e),$$

stating that the set S^e is such that increasing all the members by 2, and adding the number 0 to the set, results in the original set S^e. Since $\{0\}$ is a constant of the algebra, this definition satisfies the restrictions. Given Theorem 4.1, it can be seen that this extension of any of our algebras by S_c^e has the desired semantics.

As another example for a recursive definition, we consider a game that was one of the examples leading to the formalization of the well-founded and stable models' semantics [34]. Consider a game where one wins if the opponent has no moves (as in checkers). Assume the relation MOVE represents the possible moves. The set WIN of all winning positions is defined by the recursive equation

$$\mathrm{WIN} = \pi_1(\mathrm{MOVE} - ((\pi_1 \mathrm{MOVE}) \times \mathrm{WIN}))$$

(where π_i, $i = 1, 2$, is a shorthand for $\mathrm{MAP}_{x.i}$). The equation defines WIN to be the set containing all positions in the first

[12] Subalgebra, since the language has terms that denote all finite sets and some, but not all, infinite sets.

column of MOVE, where the next position is not a winning one. Note that the equation contains subtraction (hence, inversion of T and F for membership). We shall show in the following that equation of this form may not have a well-defined model. (If the MOVE relation is acyclic then the well-founded interpretation is 2-valued, and an initial well-founded model exists. This is not the case, however, for cyclic MOVE.)

We denote the *algebra* and the *IFP-algebra*, augmented with the capability of defining new operations by equations as above, *algebra*$^=$, and *IFP-algebra*$^=$, respectively.

Clearly, if the definitions are restricted to be nonrecursive, i.e., the defined operations cannot be used in the right-hand sides, the expressive power of the languages is not increased (every new operation can be expressed by an algebra expression containing no new operations). The extension is then just a convenience for modular programming, as it allows one to name frequently used expressions and then to use these names instead of the expressions. If recursive definitions are used, then the above extension is no longer just syntactic sugar and, as we show below, it increases the expressive power of the languages.

Whereas Theorem 4.1 guarantees that the algebraic languages are well-defined and the semantics agrees with our intentions, we do not have a corresponding result for algebras with equations. For example consider the constant S, defined by

$$S = \{a\} - S.$$

Recall that the definition of the $-$ operator performs inversion of membership. A well-founded computation starts with $\mathscr{T}$, $\mathscr{F} = \varnothing$. In particular, $\mathrm{MEM}(a, S) = T$ is not in $\mathscr{T}$. Assuming it is in $\mathscr{F}$, we can use the disequation to conclude $\mathrm{MEM}(a, S) = F$. Then using the definition of $-$, will allow us to derive $\mathrm{MEM}(a, S) = T$. So there is a potential derivation of $\mathrm{MEM}(a, S) = T$ but not a sure one. Hence, $\mathrm{MEM}(a, S) = T$ is not put into $\mathscr{F}$, and the conclusion is that both $\mathscr{T}$ and $\mathscr{F}$ retain their initial values (with respect to this fact). It follows that in the well-founded semantics the membership status of a in S is undefined. Now consider the possible well-founded algebras of the specification. Each such algebra must be a model of the specification. The only possible model is where $\mathrm{MEM}(a, S) = T$ and also $\mathrm{MEM}(a, S) = F$. Thus in every well-founded algebra of the specification we have that $T = F$; i.e., bool is not protected. This in particular means that the initial well-founded algebra does not protect *bool* and identifies T and F, which is clearly not something we want to do.

A similar observation holds in some cases for the WIN set defined in the previous example. If the MOVE relation contains, for example, the tuple $[a, a]$, then the membership status of a in WIN will be undefined and in the initial model T and F will be identified.

Clearly, we are only interested in algebras that capture the intuition that the bool type has two distinct elements, T and F. We therefore refine the definition of the initial well-founded model and from now on give that name only to the initial well-founded models that protect the booleans.

In general, it turns out that syntactic analysis is not sufficient for determining whether such a program has such an initial well-founded model.

PROPOSITION 4.3. *It is undecidable whether an algebra*$^=$ (*IFP-algebra*$^=$) *query has an initial well-founded model.* (*We assume an underlying domain with equality and at least one constant.*)

Proof. Given an *algebra*$^=$ program P, we can assume w.l.o.g. that it defines a set S, for otherwise we can simply add such a definition. Given this, and an element a, we construct an *algebra*$^=$ program P' such that P' has an initial well-founded model iff $a \notin S$, as follows: We take P, add to the language a new set constant S', and define it using the equation

$$S' = \sigma_{\mathrm{EQ}(x, a)}(S) - S'.$$

Now, if $a \in S$, then using the same argument as in the previous example, P' does not have an initial well-founded model, but if $a \notin S$ then $a \notin S'$, and an initial well-founded model for P' (in which S' is empty) exists. It follows that P' has an initial well-founded model iff $a \notin S$ in the model of P. But, in Section 8 (Proposition 8.10) we show that the problem whether $a \in S$ for some set S defined by an *algebra*$^=$ program is undecidable. ∎

This result is not surprising. We prove in Section 8 that *algebra*$^=$ has the same expressive power as general deductive programming under the well-founded semantics. Thus clearly it suffers from similar undecidability problems.

Theorem 4.1 states that *IFP-algebra* programs always have initial well-founded models. The above proposition states that this is not the case for *algebra*$^=$ programs. Thus, clearly the expressive power of the two languages is different. An interesting question is whether they are comparable. The only possibility is, of course, that *algebra*$^=$ is more powerful.

Note that the program (equations) defining the IFP operator is **not** an *algebra*$^=$ program (since the auxiliary function $F_{\mathrm{exp}}(i)$ used in the definition has an input variable that is not of *set* type). However, it is well known, and also shown in Example 4, that equations can be used to describe recursive computations. One can expect to be able to describe fixed-point computations by equations without explicitly using a fixed-point operator. In particular, let $\exp(x_0, x_1, ..., x_n)$: $\{s_0\}, \{s_1\}, ..., \{s_n\} \to \{s_0\}$ be an algebra expression, and let S: $\{s_1\}, ..., \{s_n\} \to \{s_0\}$ be a new operation name defined by the equation $S(x_1, ..., x_n) = \exp(S, x_1, ..., x_n)$.

Intuitively, the equation defines S to be a fixed point of exp w.r.t. x_0. The definition of S_c^e in Example 4 is of this kind. While S is a "real" fixed point of exp, $\mathrm{IFP}_{\mathrm{exp},\, x_0}$ computes its inflationary fixed point. How are the two fixed-point definitions related?

Using a monotonicity argument, one can easily verify that the two fixed points coincide for expressions that define monotone mappings on sets.

DEFINITION 4.4. An expression exp is **monotone** iff for every two sets S_1, S_2, if for every a $\mathrm{MEM}(a, S_1) = T$ implies $\mathrm{MEM}(a, S_2) = T$, then for every element a' $\mathrm{MEM}(a', \mathrm{exp}(S_1)) = T$ implies $\mathrm{MEM}(a', \mathrm{exp}(S_2)) = T$.

We now have

PROPOSITION 4.5. *Let exp is monotone, and let* $S(x_1, ..., x_n)$ *be defined by the equation above. Then in the well-founded interpretation, for every element a and every instantiation* $R_1, ..., R_n$ *for the variables* $x_1, ..., x_n$, *respectively, hold*:

$$\mathrm{MEM}(a, S(R_1, ..., R_n)) = T \quad \text{iff} \quad \mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0}(R_1, ..., R_n)) = T,$$

$$\mathrm{MEM}(a, S(R_1, ..., R_n)) = F \quad \text{iff} \quad \mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0}(R_1, ..., R_n)) = F.$$

Proof. We first prove the claim for the case where x_0 is the only variable of exp. From the definition of IFP it follows that $\mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0}) = T$ can be derived in a well-founded computation iff there exists some i such that $\mathrm{MEM}(a, \mathrm{exp}^i(\mathrm{EMPTY})) = T$ can be derived. Since exp is monotone, this implies that $\mathrm{MEM}(a, \mathrm{exp}^i(R)) = T$ can be derived for every set R, and in particular for $R = S$. Since $S = \mathrm{exp}(S)$, we conclude that $\mathrm{MEM}(a, S) = T$ can be derived.

On the other hand, $\mathrm{MEM}(a, S) = T$ iff there exists a well-founded computation deriving $\mathrm{MEM}(a, S) = T$. In this derivation the equation $S = \mathrm{exp}(S)$ is used[13] at most i times (for some finite i). Since there are no other equations that involve S, we can change the order of the derivation sequence such that all the derivations that use the equation $S = \mathrm{exp}(S)$ are performed at the beginning of the computation. Thus if $\mathrm{MEM}(a, S) = T$ is derived using the equation $S = \mathrm{exp}(S)$, i times, then so can $\mathrm{MEM}(a, \mathrm{exp}^j(S)) = T$, for some $j \leqslant i$, without using the equation. But, since no other equation in the specification refers to S explicitly, the same derivation sequence can be used for every $\mathrm{exp}^j(R)$ and in particular for $R = \mathrm{EMPTY}$, and it follows that $\mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0}) = T$ can be derived as well. From the above discussion it follows that $\mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0}) = T$ can be derived iff $\mathrm{MEM}(a, S) = T$ can be derived as well. Thus $\mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0}) = T$ holds in the well-founded interpretation iff $\mathrm{MEM}(a, S) = T$ holds. The argument holds for possible derivations; hence also (because of the disequation) $\mathrm{MEM}(a, \mathrm{IFP}_{\mathrm{exp},\, x_0} = F)$ holds iff $\mathrm{MEM}(a, S) = F$ holds.

For the case where exp has more then one variable, the proof is the same but, instead of representing the ith application of exp on a set R by $\mathrm{exp}^i(R)$, we use $\mathrm{exp}(\cdots \mathrm{exp}(R, R_1, ..., R_n), R_1, ..., R_n)$, where exp is repeated i times. ∎

Note in particular that every expression $\mathrm{exp}(x)$, where x does not appear negatively in exp, i.e., does not appear in any subexpression being subtracted, is monotone. If exp is not monotone (in particular, if x appears negatively in exp), then $\mathrm{IFP}_{\mathrm{exp}}$ and S may **not** have the same behavior. For example, if $\mathrm{exp} = \{a\} - x$, then

$$\mathrm{IFP}_{\{a\}-x} = (\{a\} - \mathrm{EMPTY}) \cup (\{a\} - (\{a\} - \mathrm{EMPTY})) \cup \cdots = \{a\},$$

while for the set S defined by the equation $S = \{a\} - S$, an initial well-founded model (protecting bool) does not exist. The difference between the results reflects the difference in the interpretation of subtraction. The IFP operator computes the fixed point in an inflationary manner. At each step of the computation, the set being subtracted contains only the elements computed so far. This behavior is dictated explicitly in the definition of the operator. But S, as defined by the equation $S = \mathrm{exp}(S)$ in the initial well-founded model, is the "real" fixed point of exp, in the sense that the set being subtracted is assumed to be the set being defined. As the equation is part of the definition of the set, there is cyclicity in the definition, hence, undefinedness.

Recall that the explicit definition of IFP is not legal in the *algebra*$^=$ language. It turns out, however, that, using a more complex translation technique, $\mathrm{IFP}_{\mathrm{exp}}$ can be represented in *algebra*$^=$ for every exp (even nonmonotone). We first translate $\mathrm{IFP}_{\mathrm{exp}}$ into a deductive program (proved to be possible in Proposition 7.5). Then we translate the deductive program into an *algebra*$^=$ program (proved to be possible in Proposition 8.6). It follows that

THEOREM 4.6. *IFP − algebra* $\subset$ *algebra*$^=$.

COROLLARY 4.7. *IFP − algebra*$^=$ = *algebra*$^=$.

Thus, when the ability to use recursion is added, even in the restricted manner above, a specific fixed-point operator like IFP becomes redundant.

5. DOMAIN INDEPENDENCE

For the relational and the complex objects models, every domain-independent calculus query can be represented by

[13] An equation $t = s$ is used, e.g., in another equation that was derived, say $\mathrm{MEM}(a, t) = T$, by using the equality laws and substituting s for t to obtain $\mathrm{MEM}(a, s) = T$.

an algebra query and vice versa [35, 1]. In order to extend these results to our framework, we first consider the meaning of domain independence in our context. The intended meaning of a specification is taken to be its initial (well-founded) model, and a query result is computed relative to this fixed model. It may seem, therefore, that domain independence is irrelevant. However, there are quite a few good reasons for discussing this subject. First, even when the domain is well known, a query like $\{x \mid \neg R(x)\}$ is undesirable since the answer may be very big, often infinite, and thus not very useful for the user. Second, we want the result of a query to be not only finite, but also efficiently computable. For that, it might be useful to compute the query relative to a model smaller than the initial model. For example consider a simple query without functions. Such a query refers to the elements of the database as constants with no specific interpretation, and any model containing corresponding constants suffices for computing the query (and will produce the same result). This property if often called *genericity*. Instead of computing such a generic query in the initial model which may be very large, it is enough to consider a small model that contains only the required elements. In summary, it is interesting to ask whether we get the same answer if we change the model, but still "preserve" a sufficient part of it.

A full investigation of domain independence for our general framework is presented in [27]. In the following we present some of the main ideas and the concepts that are used in the sequel. We are going to explore what it means to have the same result for a query in the initial model and in some other, preferably smaller, model. The main issue is to define how different models can share parts. We also consider how such models may be constructed.

5.1. *General Domain Independence*

Let $L(S, OP)$, $L' = (S', OP')$ be two signatures such that $S \cap S' \neq \emptyset$, $OP \cap OP' \neq \emptyset$, and let $A_L = (A_s, A_{op})$, $A_{L'} = (A_{s'}, A_{op'})$ be two models for the signatures, respectively. Every element in A_s (A'_s) may have several names in L (L'), thus can be viewed as an equivalence class of ground terms (possibly empty).

DEFINITION 5.1. Let $W \subseteq A_s$.[14] We say that the model $A_{L'}$ **preserves** W iff there exists an injective mapping h from W to $A_{s'}$ s.t. every element e is mapped to an element e' that represents an equivalence class containing all the terms represented by e. We call such W a **window**.

Window preservation formalizes the idea of "preserving" part of a model. Since the mapping is injective and the images of the window elements preserve the equivalence classes of their sources, the model $A_{L'}$ neither identifies nor splits elements of the window. Thus, $A_{L'}$ contains a subset of the elements of A_L and possibly more elements. Also, not only L and L' have *some* sorts and operations in common, but they both contain *all* the sorts and operations used for describing the window members in L (i.e., all the terms in the equivalence classes of the elements of W). As a simple example let A_L be the integers with the operations SUCC and PRED, and let $A_{L'}$ be its extension by the operations $+$, $-$. Every equivalence class in A_L is a set of expressions in SUCC and PRED. Its image in the extended model represents the same integer, but in addition to the terms using PRED and SUCC it also contains terms that use $+$ and $-$. In this example, the extended model preserves all the given model. In the general case we require preservations only for the elements of the window and allow deformations outside the window.

To define domain independence, we compare the result of a query in the initial model A_{SPEC}, with the results obtained by evaluation it in other models. We denote by $Q(DB_A)$ the result of Q when evaluated on a database DB in a model A.

DEFINITION 5.2. Let A be a model that preserves a window W of A_{SPEC}, and let DB be a database instance defined using the common operations of A and A_{SPEC}. We say that Q has the **same answer** for DB in A_{SPEC} and A, iff $Q(DB_{A_{SPEC}}) \subseteq W$, $Q(DB_A) \subseteq h(W)$, and $a \in Q(DB_{A_{SPEC}})$ iff $h(a) \in Q(DB_A)$.

DEFINITION 5.3. Q is **(finite) domain independent** iff for every database DB there exists a (finite) window W of A_{SPEC}, such that Q has the same answer for DB in all the models that preserve W.

Note that if a query is domain independent with respect to some window W then it is domain independent with respect to bigger windows and, in particular, with respect to the whole initial model. However, not every query is domain independent, not even with respect to the domain of the whole initial model. The query $\{x \mid \neg R(x)\}$ is an example.

5.2. *Interpreted Functions and Types*

Typically, queries functions that are either built-in, or user-defined. In either case, the implicit assumption is that these functions behave as specified; that is, their interpretation is fixed. We say that such functions are *interpreted*. In the presence of such functions, domain independence must be tested only relative to the models where the functions behave as in the initial model.

Standard data types, like tuples and sets, are also normally considered as types with fixed interpretation; i.e., the query result is tested relative to models where the atomic element domain may change, but tuples and sets are always interpreted in the same way; tuples are uniquely defined by their attribute values, and sets are defined by their members. We explain in [27] the importance of interpretation

[14] The meaning here (as usual in many-sorted algebras) is a collection of subsets, one for each sort.

assumptions and show that "strange" results are obtained if such assumptions are not explicitly stated. The definitions of *window preservation* and *domain independence* are refined so as to capture the notion of interpreted functions and types: domain independence is tested only relative to the models where the functions and types behave as in the initial model.

5.3. *Database Domain Independence*

In this section we consider **natural** windows with respect to which domain independence can be tested. We restrict our attention to a class of specifications (that is, we feel, quite general). Often, a type system is presented by first defining atomic types and then by defining complex types in some order; i.e., the specification of every complex type uses the type itself and previously defined (atomic and complex) types. We call such a specification a **hierarchical** type system and consider it in the following domain independence in such systems.

The atomic elements used for constructing a complex element are its **components**. We assume that for every complex type s_i, and its component atomic type s_j, *SPEC* contains a decomposition function $\mathrm{DEC}_{i,j}$ that computes these components. For example, if *SPEC* contains nested tuples and sets, then a decomposition function that generates a set containing all the atomic components of a complex object can be defined using projection, union, and unnest operators [1]. The components of a database *DB* and a query *Q* (denoted by comps(*DB*, *Q*)), are the components of all members of *DB* and the values that appear in *Q*. Following previous works, we use the concepts of components and construction to define a *database window* (or active domain).

DEFINITION 5.4. Let *DB* be a database, let *Q* be a query, and let *T* be the set of complex types in *Q*. The ***DB*-window**, of *DB* and *Q*, contains comps(*DB*, *Q*) and all the complex elements (of types in *T*) that can be constructed using elements in comps(*DB*, *Q*).[15]

Since the query may use functions for manipulating the database members, the *DB*-window must be extended and closed under function applications and their inverses [27].

DEFINITION 5.5. The **1-closure** of a window *W* w.r.t. $F = \{f_1, ..., f_m\}$, denoted by W^1_F, is the smallest set such that

1. $W \subseteq W^1_F$
2. if $a_1, ..., a_n \in W$ then $f_i(a_1, ..., a_n) \in W^1_F$
3. if $a \in W$, $a = f_i(a_1, ..., a_n)$, then $a_1, ..., a_n \in W^1_F$.

[15] Note that the DB-window is not always finite. For example, a database that contains one stack with only one atomic element *a*, has an infinite DB-window that contains all the stacks constructed by pushing *a*. Properties of such infinite windows are considered in [27].

W^1_F is constructed by closing *W* under one application (and the inverse application) of $f_1, ..., f_m$. The ***k*-closure** of *W*, denoted by W^k_F, is constructed by repeating the above process *k* times. For $k = \infty$, W^k_F is the infinite union over all the finite *k*'s (clearly, $W^0_F = W$).

DEFINITION 5.6. *Q* is ***DB*-domain independent** iff there exists some *k* (finite or infinite) s.t. for every database *DB*, *Q* has the same answer for *DB* in all the models that preserve the *k*-closure of the *DB*-window (and interpret the complex types and functions in *Q* as expected). *Q* is **strict** *DB*-domain independent if *k* is finite.

The difference between the notions of domain independence defined in Definition 5.3 and here is that, there any window will do, but here only a specific kind of window is considered.

For calculus queries it seems natural to consider *k*-closure, where *k* is the number of function applications in *Q*. For deductive queries *k* may be infinite, but syntactic restrictions (such as functions stratification [1]) may assure its finiteness. We consider this subject further in Section 8.

This definition of domain independence is very general and applies to queries and databases with quite general type structures. In particular it applies to the relational and the complex object [1] models.

THEOREM 5.7. *A relational query with no interpreted functions is domain independent by the traditional definition, iff it is DB-domain independent with respect to the interpreted tuple sort(s). The same holds if* "*relational*" *is replaced by* "*complex objects*," *and* "*tuple*" *is replaced by* "*tuple and set*."

Proof. The proof follows immediately from the fact that for type systems that contain only tuples (and sets), the DB-window contains all the tuples (and sets) that can be constructed from database members. Thus, if the query contains no functions, then our definition of domain independence is identical to the traditional one. ∎

In [27] we consider properties of these concepts. In particular we show that DB-domain independence implies general domain independence but not vice versa, and that, as in the classical definition, (DB-)domain independence is undecidable for the general predicative and deductive query languages. We conclude this section with the following observation.

THEOREM 5.8. *The functional query language presented in Section* 3.2 *is domain independent.*

Proof. The proof follows from the definitions of the language and (DB-)domain independence. Recall that in the functional paradigm in its most general form considered in this paper, as described in Section 3.2, the database relations and the query result are all represented by constants named in a specification. That is, the domains and their

operations, and the database contents are described in some specification, and a query extends it by definitions of additional operations and by an equation that defines the result, of the form $Q = \text{exp}$, where Q is a constant and exp may use both the given and the additional operations. Thus, if a functional query is defined by an equation $Q = \text{exp}$, and the window contains the constant Q and all its atomic components, then every model that preserves W and interprets sets and the functions used in exp in the expected way has the same answer for Q. ▌

It follows that the algebras, being sublanguages of the general functional language, are also domain independent. Note however, that the operator *complement* in its standard definition (i.e., complement with respect to the given model) does not have an algebraic specification. The *complement* as we defined it has a slightly different meaning, it computes the complement with respect to the **initial model**, and this is also its value when used in a different model. That is, its result is independent of the model. Otherwise, it would not be domain independent. This subject will be further discussed in the next section.

6. THE CALCULUS AND THE ALGEBRA

We now proceed to compare the expressive power of the predicative and functional languages. Specifically, we compare the calculus and the algebra. The equivalence of relational algebra and calculus is well known and has been generalized to models with complex objects and with object identity [1, 25]. We consider the two directions of the equivalence proof and present general conditions on the languages that allow the proofs to go through.

6.1. *From Calculus to Algebra*

The notion of a domain for the variables is built into the calculus, but the algebra has no variables, just operations that take database objects as inputs. Thus, a crucial part of the proof of this direction is to show that expressions that represent the domain over which variables are quantified can be written in the algebra. The precise nature of these expressions depends on the calculus. If there is no restriction on ranges of the variables, then expressions representing the full domains are needed. But if the calculus is, for example, restricted so that each variable ranges only on a database relation, then the algebra expressions describing the range of quantifications are simply the relation names. These two cases are examined next.

6.1.1. *Unrestricted calculus.* We start by investigating the relationship between the unrestricted (i.e., not necessarily domain independent) calculus, and the S-algebra (containing the operators $\cup$, $-$, $\times$, σ, MAP, and domain constants). We restrict our attention to type systems where equality is definable for all the (data) types. Let *SPEC* be such a type system and let DB^p and DB^a be predicative and functional database specifications (respectively) over *SPEC* that describe the same database. (Assume for now that such descriptions exist.) The **intended meaning** of a calculus query is defined by evaluating it in the initial model of DB^p, and that of an algebraic query, by evaluating it in the initial model of DB^a. (Note that since DB^p and DB^a represent the same database, their models are isomorphic.)

THEOREM 6.1. *For each calculus query Q^c, there exists an S-algebra query Q^a s.t. Q^c and Q^a have the same result when computed relative to the initial (well-founded) models of DB^p, and DB^a; i.e., $Q^c(x)$ holds if* $\text{MEM}(x, Q^a) = T$, *and* $\neg Q^c(x)$ *holds iff* $\text{MEM}(x, Q^a) = F$.

Proof. We prove the theorem by following the lines of the classical translation [35, 1] and use induction on the structure of the calculus query. We present here only elements which are special to our generalized model. We rely on the fact that domain constants are available.

Basis. An expression of the form $R_i(x)$ is translated to R_i^a. An expression of the form $\text{exp}_1 = \text{exp}_2$ with free variables $x_1, ..., x_n$ from types $s_1, ..., s_n$ is translated to the expression $\sigma_{\text{EQ}, (\widehat{\text{exp}_1}, \widehat{\text{exp}_2})}(\text{dom}(s_1) \times \cdots \times \text{dom}(s_n))$, where $\text{dom}(s_j)$ are sets representing the sort domain (and constructed as in Example 3), and $\widehat{\text{exp}_i}$ is constructed from exp_i by replacing every occurrence of variable x_j in exp_i by the projection of the jth attribute of tuples in $\text{dom}(x_1) \times \cdots \times \text{dom}(s_n)$. For example, the expression $f(x_1, x_2) = g(x_3, x_1, x_4)$, where x_i is of sort s_i, is translated to $\sigma_{\text{EQ}(f(x.1, x.2), g(x.3, x.1, x.4))}(\text{dom}(s_1) \times \text{dom}(s_2) \times \text{dom}(s_3) \times \text{dom}(s_4))$.

Induction. For the induction step it suffices to consider cases where ϕ is constructed from atomic formulas using $\wedge$, $\neg$, and the existential quantifier. For $\wedge$ we use cartesian product and selection by equality for variables with the same names, and then use projection (using MAP) for omitting multiple occurrences of the same variables. For negation we take the complement with respect to the cartesian product of the sets $\text{dom}(s_i)$ that represent the domains of the free variables in the negated formula. Finally, we use projection (using MAP) for the existential quantifier.

Since the sets $\text{dom}(s_i)$ used in the translation process may be infinite, a pure initial model approach may not be sufficient for defining the semantics of the query, but rather an initial well-founded model semantics must be used. Since the S-algebra is well defined, by Theorem 4.1, every S-algebra query, and in particular the query Q^a constructed by the above process, has an initial well-founded model, where MEM is a total boolean-valued function. For every x, $\text{MEM}(x, Q^a)$ equals either T or F, and in particular $\text{MEM}(x, Q^a)$ equals T iff $Q^c(x)$ holds, and equals F otherwise. ▌

Note that the statement "$Q^c(x)$ holds iff $\mathrm{MEM}(x, Q^a) = T$" does not imply computability. In the predicative approach, the answer is *defined* to be the x's for which $Q^c(x)$ is true; for the algebraic approach the answer is the x's for which $\mathrm{MEM}(x, Q^a) = T$. The statement only tells us that the same answer is obtained, but it is possible that no algorithm for computing it exists.

Also note that since Q^c is not necessarily domain independent, Q^c and Q^a have the same result only when both are evaluated relative to the respective initial models. For example, if R is a database relation that contains elements of type s_i, then the queries $Q^c = \{x \mid \neg R(x))\}$ and $Q^a = \mathrm{dom}(s_i) - R^a$ have the same result in the initial model. But while Q^a has the same result in all models (hence is domain independent), Q^c is not domain independent and its result changes if the domain is replaced. The query Q^c computes the complement with respect to the current model, while Q^a always computes the complement with respect to the initial model.

6.1.2. *DB-Domain Independent Queries.* The translation process above is, of course, valid for domain independent queries as well. However, since the computation of such queries uses only part of the initial model (the window), it is interesting to search for a more efficient translation that uses smaller sets that contain only the relevant window elements.

In turns out that for DB-domain independent queries, expressions that denote sets that can replace the sets $\mathrm{dom}(s_i)$ can be constructed. Recall that the window used for computing DB-domain independent queries contains all the database (and query) components, all the complex elements that can be constructed using these components, and is further closed under k-function applications and their inverses for some k. Thus, we look for conditions on the algebra that guarantee the existence of algebraic expressions that denote the sets of elements of a window.

The construction of such a window consists of three parts: (i) For each atomic type s_i we construct a set $\hat{S}_i$ that contains all the atomic components of type s_i, of elements in the database or the query. (ii) Then, for each complex type s_i, a set $\hat{S}_i$ containing all the complex elements that can be constructed from these components is generated. (iii) Finally, the sets are closed under function applications/inverses as many times as needed.

Let $R_1^a, ..., R_n^a$ be the names of the database relations, and let $c_1, ..., c_l$ be the names of the constants in Q^c. The first part is performed as follows: The sets $\hat{S}_i(R_1^a, ..., R_n^a, c_1, ..., c_l)$ of all the atomic components of types s_i in the database and query can be constructed as follows:

$$\hat{S}_i(R_1^a, ..., R_n^a, c_1, ..., c_l) = \left(\bigcup_{k=1}^{n} \mathrm{DEC}_{j_k,\, i}(R_k^a)\right) \cup \left(\bigcup_{h=1}^{l} \mathrm{DEC}_{j_{n+h},\, i}(c_h)\right),$$

where s_{j_k}, $s_{j_{n+h}}$ are the types of the relations R_k^a and the constants c_h, and $\mathrm{DEC}_{j_k,\, i}$ are decomposition functions that decompose the relations (and constants) into their atomic components.[16]

For the second part, the sets $\hat{S}_i(R_1^a, ..., R_n^a, c_1, ..., c_l)$, $i = 1 \cdots m$, containing the elements of complex type s_i, constructed from database and query components, are generated (using the IFP operator) by computing the mutual fixed point of the expressions

$$\begin{aligned}
&\exp_i(x_0^1, ..., x_0^m, R_1^a, ..., R_k^a, c_1, ..., c_l) \\
&\quad = \{c \mid c \text{ is a constant function of arity } c{:}\to s_i\} \\
&\quad \cup \bigcup \{\mathrm{MAP}_{f_j}(x_0^{j_1} \times \cdots \times x_0^{j_m} \times \hat{S}_{j_{m+1}}(R_1^a, ..., R_n^a, c_1, ..., c_l) \\
&\qquad \times \cdots \times \hat{S}_{j_k}(R_1^a, ..., R_n^a, c_1, ..., c_l)), \\
&\qquad \text{where } f_j \text{ is of arity } s_{j_1}, ..., s_{j_k} \to s_i, \\
&\qquad \text{and } s_{j_1}, ..., s_{j_m}\ (s_{j_{m+1}}, ..., s_{j_k}) \\
&\qquad \text{are complex types (atomic types) of } SPEC\}.
\end{aligned}$$

The first step of the mutual fixed-point computation constructs sets containing the constants of the complex types. Then (iteratively) more complex elements are generated using the database components and previously constructed elements. The value of $\exp_i$ in the (mutual) fixed point, denoted by $\mathrm{IFP}^m_{\exp_i}(R_1^a, ..., R_n^a, c_1, ..., c_l)$, is a set containing all the complex elements of type s_i that can be constructed from the components of $R^1, ..., R^n, c_1, ..., c_n$. Thus, when the relations (and constant) names are instantiated by the actual database relations (query constants), the resulting sets contain exactly the members of the DB-window.

We call the IFP operator, when restricted to computing only mutual fixed points of expressions of the above form the DB–IFP operator.

Finally, in the third part, the DB-window must be closed under functions applications and their inverses. In order to close the window under function applications, we apply the functions on the members of the window and add the result to the window. This can be done by applying MAP and $\cup$ on the sets that represent the window. To close the window under inverse of functions we must be able to compute the *sources* of the window elements with respect to those functions; i.e., for every window element a we have to compute the vectors $\langle a_1, ..., a_n \rangle$ such that $a = f(a_1, ..., a_n)$ and add the components of these vectors to the window. We say that a function f in *SPEC* is **invertible** iff *SPEC* contains a function $\hat{f}$ that for every element a computes the set of sources of a with respect to f. If the relevant functions are invertible, then the k-closure of the DB-window can be constructed by k applications of f and $\hat{f}$ using MAP and $\cup$ (and if the

[16] Recall from Section 5 that we consider hierarchical type system where all the complex types have decomposition functions.

window elements have several sources with respect to f then UNNEST is used for decomposing the sets of sources).

Assume $W_1^k, ..., W_n^k$ are the sets that contain the elements of type s_i in the k-closure of the DB-window (clearly $W_i^0 = \hat{S}_i$). Then the sets W_i^{k+1} that represent the $k+1$ closure are constructed as follows:

$$W_i^{k+1} = W_i^k \cup \Big(\bigcup \{ \mathrm{MAP}_{f_j}(W_{j_1}^k \times \cdots \times W_{j_m}^k), \text{ where } f_j \text{ is of arity } x_{j_1}, ..., s_{j_m} \to s_i \} \Big)$$
$$\cup \Big(\bigcup \{ \mathrm{MAP}_{x.i}(\mathrm{UNNEST}(\mathrm{MAP}_{\hat{f}_j}(W_l^k))), \text{ where } \hat{f}_j \text{ is the inverse of some } f_j \text{ of arity } s_{j_1}, ..., s_i, ..., s_{j_m} \to s_l \} \Big).$$

In summary, it is clear that for every finite k, the k-closure of the DB-window can be constructed using the DB–IFP operator and the MAP and UNNEST functions.

We call the algebra consisting of the operations $\cup$, $-$, $\times$, σ, MAP, UNNEST, and DB–IFP, the **DB-algebra**. Then we claim:

THEOREM 6.2. *For every strict DB-domain independent calculus query Q^c, if all the interpreted functions in Q are invertible, there exists a DB-algebra query Q^a, such that Q^c and Q^a have the same result; i.e., $Q^c(x)$ holds iff* $\mathrm{MEM}(x, Q^a) = T$ *and* $\neg Q^c(x)$ *holds iff* $\mathrm{MEM}(x, Q^a) = F$. *Also, if the DB-window is finite, then negation is not needed to define Q^a.*

Proof. The proof follows from the fact that for strict domain-independent queries it suffices to close the DB-window under k function applications, where k is finite. Thus the sets that represent the part of the domain that is essential for the computation can be constructed (from the DB-window) using a finite number operator applications. If the DB-window is finite, then the k-closure is represented by finite sets. The translated query then uses only finite sets, and negation is not needed for defining the MEM function. ∎

Note that the UNNEST operator is only used for closing the DB-window under inverses of interpreted functions. Thus, if no interpreted functions are used in the query, the operator is not needed. Also note that in the relational (and complex object) models, the sets that represent the DB-window can be constructed using projection, cartesian product (and powerset). For such models, the pure algebra (augmented with a power set operator) is sufficient for constructing Q^a [1], and DB–IFP is not needed. The above theorem claims that whenever the algebra is rich enough for constructing the DB-window of all the types used in queries, all the strict DB-domain independent calculus queries can be described. By that it generalizes the results shown by previous works.

6.2. *From Algebra to Calculus*

We start by considering the DB-algebra. If *SPEC* is an hierarchical specification that has a decomposition function $\mathrm{DEC}_{i,j}$ for every complex type s_i and atomic type s_j, then

THEOREM 6.3. *Every DB-algebra query Q^a can be expressed by a strict DB-domain independent calculus query Q^c (where domain independence includes the assumption that the complex types and the functions of SPEC that appear in Q^a are interpreted).*

Proof. We prove the theorem by induction on the structure of the algebraic expression (by presenting a translation for every algebra operator).

Basis. The algebra queries $\{\mathrm{exp}\}$ (where exp is a ground term), and R_i are translated to $Q = \{x \mid x = \mathrm{exp}\}$ and $Q = \{x \mid R_i(x)\}$, respectively.

Induction. We assume E_1, E_2 are algebra queries and ϕ_1, ϕ_2 are the formulas in the corresponding calculus queries. Let E be an algebra query that is constructed from E_1, E_2. We show how to construct ϕ from ϕ_1, ϕ_2. For $\cup$, $-$, and $\times$, the construction is as in the relational case. Thus, we present here only translation for the rest of the operators:

σ_{test}, if $E = \sigma_{\mathrm{test}}(E_1)$ then $\phi = \phi_1(x) \wedge \mathrm{test}(x) = T$.

MAP_f, if $E = \mathrm{MAP}_f(E_1)$ then $\phi = \exists x(\phi_1(x) \wedge y = f(x))$.

UNNEST, if $E = \mathrm{UNNEST}(E_1)$, and the members of E_1 are of type $[s_1, ..., s_{n-1}, \{s_n\}]$, then

$$\phi = \exists x, x_1, ..., x_n, z(\phi_1(x) \wedge x_1 = x.1 \wedge \cdots \wedge x_n = x.n \wedge z \in x_n \wedge y = [x_1, ..., x_{n-1}, z]).$$

DB–IFP, the DB–IFP is a restricted version of the IFP operator. Thus the expressions we have to translate have the form $E = \mathrm{IFP}^m_{\mathrm{exp}_i}(R_1^a, ..., R_n^a, c_1, ..., c_l)$, where exp_i is the expression used for generating all the elements of the complex type s_i that can be constructed from components of elements in the sets $R_1^a, ..., R_n^a$ and the constants $c_1, ..., c_l$.

The corresponding calculus formula has the form: $\phi = \psi_1 \wedge \cdots \wedge \psi_k$, such that

$$\psi_j = \forall y \Bigg(y \in \mathrm{DEC}_{i,j}(x) \rightarrow \Bigg(\Bigg(\bigvee_{p=1\cdots n} \exists z (y \in \mathrm{DEC}_{p,j}(z) \wedge R_p(z)) \Bigg) \vee \Bigg(\bigvee_{q=1\cdots l} y \in \mathrm{DEC}_{n+q,j}(c_q) \Bigg) \Bigg) \Bigg),$$

where $s_1, ..., s_k$ are the atomic types in *SPEC*; s_p is the type of the members of the predicate R_p (representing the set R_p^a), and s_{n+q} is the type of the constant c_q.

This complex calculus formula simply states that all the components of the elements in the result are also components of some relation or constant.

A careful examination of the calculus formula Q^c constructed by the above process, shows that all the domain elements used in the computation of Q^c, are either database members, complex elements constructed from component of database members, or elements constructed by applying functions on these elements. Thus Q^c is strict DB-domain independent. ▮

Note that both the relational and complex object models have hierarchical type systems, where *tuples* and *sets* are interpreted types. Nested tuples and sets can be decomposed into atomic components using projection, union, and unnest. Thus, from the above theorem it follows that algebra queries over nested tuples and sets can be represented by calculus queries. Moreover, whenever the type system is powerful enough so that the complex values used in the queries can be decomposed, the translation is possible.

Now, consider the *S-algebra*. Unlike the DB–IFP operator that can be described by a domain-independent calculus query, this is not the case for the more general S–IFP operator. For example, we show

THEOREM 6.4. *The set* S_{nat}, *containing all the natural numbers, cannot be expressed by a domain-independent calculus query, unless the domain of natural numbers is interpreted.*

Proof. We prove the theorem by showing that the result of every domain-independent query (over the natural numbers) is **finite**; thus it cannot contain all the natural numbers. This is done by considering the model of natural numbers and another model that also contains other elements, that we refer to for simplicity as nonstandard numbers. In addition to $0, \mathrm{SUCC}(0), \mathrm{SUCC}^2(0), ...,$ it contains the nonstandard numbers $0'$, $\mathrm{SUCC}(0')$, $\mathrm{SUCC}^2(0'), ...$ We show that whenever the result of Q^c is infinite, Q^c has a different result in the standard and nonstandard models; thus it is not domain independent. For brevity, we denote in the following a standard number $\mathrm{SUCC}^i(0)$ by i, a nonstandard number $\mathrm{SUCC}^i(0')$ by i', and $\mathrm{SUCC}^i(x)$ by $x+i$. We use the following proposition.

PROPOSITION 6.5. *Let ϕ be a calculus formula over the natural numbers, and let n be the number of applications of* SUCC *in ϕ. ϕ is satisfiable by a vector $v = \langle i_1, ..., i_m \rangle$ iff it is satisfiable by a vector v' that is constructed from v by replacing every $i_j > n$ by i'_j. (Recall that i_j and i'_j denote* $\mathrm{SUCC}^{i_j}(0)$ *and* $\mathrm{SUCC}^{i_j}(0')$, *respectively. Thus, we replace in v every component* $\mathrm{SUCC}^{i_j}(0)$, $i_j > n$, *by* $\mathrm{SUCC}^{i_j}(0')$).

Proof. We prove the proposition by induction on the structure of ϕ.

Basis. An atomic formula may have the form $x+k=j$, $x+k=y+j$, or $x+k=x+j$ (where x, y are variables and k, j are constants). If ϕ is of the form $x+k=j$, then clearly ϕ is satisfied by at most one number $j-k \leqslant n$. If ϕ is of the form $x+k=y+j$, $k>j$ (or $k \leqslant j$) then ϕ is satisfied by all the vectors $\langle i, i+k-j \rangle$ (or $\langle i+j-k, i \rangle$), and all the vectors $\langle i', i'+k-j \rangle$ (or $\langle i'+j-k, i' \rangle$). Finally, if ϕ is of the form $x+i=x+j$, and $i=j$, then ϕ is satisfiable by all the standard and nonstandard numbers. Else, if $i \neq j$, then the result if empty.

Induction. For the induction step it suffices to consider cases where ϕ is constructed from simpler formulas using $\wedge$, $\neg$ and existential quantifier. If ϕ is satisfied only by vectors where all the components are smaller than n then we are done. Else ϕ is satisfied by a vector $\langle i_1, ..., i_m \rangle$, where $i_j > n$ for some $1 \leqslant j \leqslant m$.

Consider first $\phi = \phi_1 \wedge \phi_2$. ϕ is satisfied by a vector v, iff v it satisfies both ϕ_1 and ϕ_2. From the induction assumption we have that v satisfies ϕ_i iff v' satisfies it as well and, thus, satisfies ϕ. Similar arguments hold for $\neg$ and $\exists$.

From the above proposition it follows that $Q^c = \{ x \mid \phi \}$ is domain independent iff ϕ is satisfied only by vectors where all the components are not bigger than n. It follows that every domain independent query is satisfied by a finite number of natural numbers. Thus there is no domain independent calculus query whose result contains all the natural numbers. ▮

The *DB-algebra* and the *S-algebra* do not contain the operators NEST and AGGR. Thus, we consider the translation of these operators to the calculus separately.

The expression $E = \mathrm{NEST}_{f_1, f_2}(E_1)$ can be represented by the calculus formula:

$$\phi = \{ y \mid \exists x, s(\phi_1(x) \wedge y = [f_1(x), s] \wedge \forall z (z \in s \leftrightarrow \exists w(\phi_1(w) \wedge f_1(w) = f_1(x) \wedge f_2(w) = z))) \}.$$

But, the operator $\mathrm{AGGR}_{g,h,\mathrm{unit}}$ cannot always be expressed by the calculus. For example, the query *even* that checks

whether a database relation has even or odd cardinality, can be easily defined using the AGGR operator. But it is well known that such a query is not expressible by the relational calculus. Thus, if *tuples* are the only complex type in *SPEC*, then *even* cannot be expressed by the calculus. In general, the existence of a translation for AGGR depends on the nature of g, h, and *unit*.

7. DEDUCTION

We now consider the relative power of deductive languages compared to that of algebras with some form of fixpoint operation.

7.1. *Stratified Deduction*

Consider a domain **dependent** calculus query Q. One may try to find a domain-independent calculus query Q_1 that represents the *intended meaning* of Q; i.e., when both Q and Q_1 are computed relative to the initial model they have the same result, but while Q's answer changes when the domain is replaced, the answer of Q_1 remains the same. From the last result of the previous section it follows that such a calculus query Q_1 does not always exist. However, we show in the following that a stratified deductive query Q_1 always exists. (Moreover, in the next section we show that the deductive program has "nice" safety property.)

PROPOSITION 7.1. *Every S-algebra query has an equivalent domain independent stratified deductive query.*

Proof. The proof is by induction on the structure of the algebraic expression. For each subquery, a **new** predicate name is introduced, and a derived relation is defined. It is then quite easy to construct a rule that expresses how a subquery is constructed from its component subqueries. The translation technique we present here is more general then actually needed to prove the proposition, but it will be useful later for another result.

Basis. $\{a_1, ..., a_n\}$. Every set occurrence $\{a_1, ..., a_n\}$ is translated into an deductive program by introducing a new predicate name R and adding the rules $R(a_1)$, ..., $R(a_n)$.

R_i^a, The expression R_i that represents in the algebra a database collection is translated to corresponding database predicate, i.e., to the rule $R_i(x) \to R(x)$.

$\mathrm{dom}(s_i)$, Recall that domain constants can generated using the IFP operator. Thus the expressions we have to translate have the form $E = \mathrm{IFP}^n_{\exp_i}$, where $\exp_i$ is the expression used for generating $\mathrm{dom}(s_i)$. In the deductive program we represent every such sort s_i by a corresponding predicate S_i. To define these predicates, we generate for every algebra expression $E_i = \exp(S_1, ..., S_n)$ a deductive program that defines the predicate R_i, and then we add the rules $R_i(x) \to S_i(x)$. Finally, we add a rule $S_i(x) \to R(x)$.

Note that in the last case by adding the last rules the program becomes recursive. But since the expressions $\exp_i$ do not use subtraction $(-)$, the program is stratified.

Induction. Let E_1, E_2 be some algebra queries and let ϕ_1, ϕ_2 be the corresponding deductive programs such that R_1, R_2 are the derived predicates which corresponds to E_1, E_2. Let E be an algebra query constructed from E_1, E_2. We show, in the following, how to construct the corresponding deductive program. In general, this is done by adding a new derived predicate R and adding clauses to $\phi_1 \cup \phi_2$ for defining R:

$\cup$, if $E = E_1 \cup E_2$, we add the rules $R_1(x) \to R(x)$, and $R_2(x) \to R(x)$

$\times$, if $E = E_1 \times E_2$, then we add the rule $R_1(x) \wedge R_2(y) \wedge z = [x, y] \to R(z)$

σ_{test}, if $E = \sigma_{\mathrm{test}}(E_1)$, we add the rule $R_1(x) \wedge \mathrm{test}(x) = T \to R(x)$

$-$, if $E = E_1 - E_2$, then we add $R_1(x) \wedge \neg R_2(x) \to R(x)$

MAP_f, if $E = \mathrm{MAP}_f(E_1)$, we add the rule $R_1(x) \wedge y = f(x) \to R(y)$.

All the elements used in the computation of the deductive query Q^c generated by the above process are database members, elements that have names in *SPEC*, or elements constructed from these elements by function applications. Thus, all are elements of the initial model. It follows that in any model that preserves the initial model, the query has the same result; thus Q^c is domain independent. ∎

From the above propositions it follows that:

COROLLARY 7.2. *For every calculus query Q^c there exists a stratified domain independent deductive query that computes the intended result of Q^c.*

7.2. *Inflationary Semantics Deduction*

S-algebra queries are translated to stratified domain independent deductive programs using induction on the structure of the query. A similar translation can be used for general IFP-algebra queries. However, since the IFP operator can be applied to expressions that include the operator "$-$," the corresponding deductive program may not be stratified. We call the fixed-point operator when restricted to expressions where x_0 does not appear to be negatively **positive IFP**, and the corresponding algebra is not a **positive IFP-algebra**. Queries in the positive IFP-algebra are called **positive queries**. It is easy to see that if the IFP operator is *positive*, the resulting program is stratified. This is not the case, however, when the translation technique is applied to nonpositive queries.

EXAMPLE 5. The general fixed-point expression $Q = \mathrm{IFP}_{\{a\}-x}$ is translated to the deductive program

$R(a)$

$R(x) \wedge \neg Q(x) \rightarrow Q(x)$

which clearly is not stratified. ▮

Recall that the fixed-point operator has an inflationary semantics, where subtraction of a variable is interpreted as "was not derived so far." If the resulting deductive program is not stratified, then its result is the same as that of the original algebra query, only if it is computed using inflationary semantics as well. If a different semantics is used, e.g., well-founded model semantics, then the algebra query and the deductive queries may have different results. This is because the well-founded model semantics interprets negation as "cannot be derived at all" (vs "was not derived so far" in inflationary semantics).

EXAMPLE 5 (Continued). Since $Q = \text{IFP}_{\{a\}-x}(\{a\} - \text{EMPTY}) \cup \cdots = \{a\}$, the element a belongs to Q. When the corresponding deductive program is evaluated using inflationary fixed-point semantics, the computation proceeds as follows: First iteration; the fact $R(a)$ is derived. Second iteration; since no facts have been derived so far for Q, $\neg Q(a)$ is assumed and $Q(a)$ is derived. Third iteration; no more facts are derived and the computation terminates. Thus the query result is the same as that of the original algebra query.

When the deductive program is evaluated using well-founded model semantics, $Q(a)$ is neither true nor false. This is because in this approach, a fact can be assumed to be false only if there is no possible derivation for it. Thus, $Q(a)$ cannot be assumed to be false. But $Q(a)$ cannot be derived unless $\neg Q(a)$ is assumed. Thus neither $Q(a)$ nor $\neg Q(a)$ hold in the well-founded model. It follows that the result is different than that of the original algebra query.

Thus, we have:

PROPOSITION 7.3. *Every IFP-algebra expression has a logic program that is equivalent to it under the inflationary semantics.*

7.3. *Well-Founded Semantics Deduction*

Now, what can we say about translation to deductive programs with well-founded semantics? It turns out that

PROPOSITION 7.4. *For every deductive program P there exists a deductive program P' such that for every predicate R in P and every element a, $R(a)$ holds when P is evaluated using inflationary fixed point semantic, iff $R(a)$ holds when P' is evaluated using well-founded model semantics.*

Proof. We first assume that *SPEC* contains the natural numbers (the general case is considered later). The program P' is constructed by modifying P as follows:

(i) For every predicate name R we add a new predicate name R'.

(ii) Every ground fact $R(a)$ is replaced by $R'(0, a)$.

(iii) Every rule $\cdots(\neg)\, Q(x) \cdots \rightarrow R(y)$, is replaced by $\cdots(\neg)\, Q'(i, x) \cdots \rightarrow R'(i+1, y)$.

(iv) Finally, for every R' we add the rules $R'(i, x) \rightarrow R'(i+1, x)$ and $R'(i, x) \rightarrow R(x)$.

The program P' simulates the inflationary computations of P. At each step of the derivation, new facts can only be derived using facts with smaller indexes. Thus the result obtained using well-founded semantics is the same as the one obtained by the inflationary computation of Q'.

Now, if *SPEC* does not contain a natural number but it contains some other infinite domain, then the constants and constructors of that type can be used to simulate the 0 and SUCC functions. If all the domains in *SPEC* are finite, then the number of steps in the computation is bounded by the number of constants. Thus, we can simply enumerate the constants and use them for counting (i.e., derive new facts from facts where constants with smaller indices are attached). ▮

It follows that a (nonpositive) IFP-*algebra* query Q can be translated into an equivalent deductive program as follows: Q is first translated into a deductive program P such that Q and P are equivalent when P is evaluated using inflationary fixed-point semantics. Next, P is transformed into a program P' that has the same result under well-founded model interpretation. It follows that

PROPOSITION 7.5. *Every* IFP-*algebra query has an equivalent (under well-founded semantics) domain-independent deductive query. Moreover, if the algebra query is positive, the corresponding deductive program is stratified.*

Theorem 4.6 (whose proof actually uses results of the next section) states that the IFP-*algebra* is properly included in the *algebra*$^=$. Thus, an interesting question is whether general *algebra*$^=$ queries can also be expressed by domain-independent deduction. It turns out that the answer is positive.

PROPOSITION 7.6. *Every algebra*$^=$ *query has an equivalent (under well-founded semantics) domain-independent deductive query.*

Proof. The translation technique is similar to that of the positive IFP-*algebra* queries presented above. In a way, it is simpler, since we have no IFP operation to translate. The only interesting part is the translation of equations. Every expression $f_i(e_1, ..., e_n)$, where f_i is defined by an equation $f_i(x_1, ..., x_n) = \exp_i(x_1, ..., x_n)$ is represented in the deductive program by a corresponding predicate R_i. To define this predicate, we generate for the algebra expression $\exp_i(e_1, ..., e_n)$ a deductive program with result predicate S_i

and then add the rule $S_i(x) \to R_i(x)$. Note that the *algebra*$^=$ query Q and its corresponding deductive program Q' both interpret subtraction and negation (respectively) using well-founded semantics. Thus we have the same result. ▮

Based on the above discussion we can now consider the question of whether every database can be specified in each of the two modes. A partial answer is:

PROPOSITION 7.7. *Every functional style database definition DB^a defined using algebra$^=$ has an equivalent domain independent deductive style database definition DB^p. If the expression is defined in positive* IFP-*algebra, then the deductive program is stratified.*

8. SAFETY

Our definition of domain independence is semantic in nature. In this section we present syntactic restrictions that guarantee domain independence. We will call formulas that are syntactically restricted, by a condition that guarantees domain independence, *safe*. The restrictions we present below are similar to those presented in the literature [35, 1], but are more general; they assure that all the elements needed for computing the query either appear in the database or are obtained from them by decomposition, construction, and function applications.

The **safe calculus queries** use only elements of the k-closure of the database window (where k is finite) and thus are strict DB-domain independent, while the **safe deductive queries** use only elements of the initial model and thus are general domain independent. Moreover, we show that the restrictions on the calculus/deductive queries are general enough so that all the strict DB-domain independent and domain independent queries (respectively) can be expressed.

8.1. *The Safe Calculus*

We first consider safe calculus queries. The accepted approach to making a formula safe is to restrict the range of the variables in the formula. We define below range formulas, and restricted variables, and later use them for defining safe calculus formulas.

DEFINITION 8.1. A range formula restricting the variables $x_1, ..., x_n$ is a formula where $x_1, ..., x_n$ are free, having the structure defined below:

basis. a. $R(x_1)$ is a range formula restricting x_1.

b. $x_1 = \text{exp}$, where exp is a ground expression, is a range formula restricting x_1.

construction. If ϕ_1, ϕ_2 are range formulas restricting $x_1, ..., x_n$ and $y_1, ..., y_m$, respectively, then:

1. $\phi_1 \vee \phi_2$, where ϕ_1 and ϕ_2 have the same free variables $(x_1, ..., x_n)$, is a range formula restricting $x_1, ..., x_n$.

2. $\phi_1 \wedge \phi_2$ is a range formula restricting both $x_1, ..., x_n$ and $y_1, ..., y_m$.

3. $\phi_1 \wedge (\text{exp}_1 = \text{exp}_2)$, where all the variables in exp_1, exp_2 are restricted by ϕ_1, is a range formula restricting $x_1, ..., x_n$.

4. $\phi_1 \wedge \neg\phi_2$, where all the free variables in ϕ_2 are restricted by ϕ_1 is a range formula restricting $x_1, ..., x_n$.

5. $\phi_1 \wedge y = \text{exp}$, where all the variables of exp are restricted by ϕ_1, is a range formula restricting $y, x_1, ..., x_n$.

6. $y \in \text{DEC}_{i,j}(x_i) \wedge \phi_1$, where y is a variables of the atomic type s_j and x_i is a variable of the complex type s_i that is restricted by ϕ_1, is a range formula restricting $y, x_1, ..., x_n$.

7. $\psi_1 \wedge \cdots \wedge \psi_k$, such that $\psi_j = \forall x_j (x_j \in \text{DEC}_{i,j}(y) \to \phi_1)$, where $s_1, ..., s_k$ are the atomic types in *SPEC*, x_j is a variable of the atomic type s_j that is restricted by ϕ_1, and y is a variable of the complex type s_i, is a range formula restricting $y, x_{k+1}, ..., x_n$.

8. $\exists x_i(\phi_1)$, where x_i is restricted by ϕ_1, is a range formula restricting the variables, $x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$.

We call a calculus query $Q = \{x \mid \phi\}$ **safe**, iff ϕ is a range formula that restricts x.[17] We refer to the calculus, restricted to safe queries as the **safe calculus**, and show:

PROPOSITION 8.2. *Every safe calculus query can be expressed by a DB-algebra query and vice versa.*

Proof. The proof of the second direction follows immediately from the proof of Theorem 6.3. To prove the first direction we, again, follow the lines of the classical translation [35, 1] and use induction on the structure of the calculus query.

Basis. A subexpression of the form $R_i(x)$ is translated to R_i^a. A subexpression of the form $x = \text{exp}$ is represented by the set $\{\text{exp}\}$.

Induction. For 1 above we use union. For 2 we use the Cartesian product and selection by equality for variables with the same names, and then we use projection (using MAP) for deleting multiple occurrences of the same variables. Formula 3 is expressed using selection by equality of $\widehat{\text{exp}_1}$ and $\widehat{\text{exp}_2}$, where exp_i is constructed from $\widehat{\text{exp}_i}$ by replacing every occurrence of the variable x_j in exp_i by $x.j$. For 4 we use subtraction, and 5 is expressed by applying $\text{MAP}_{\widehat{\text{exp}}}$, where exp is constructed from $\widehat{\text{exp}}$ as above. For 6 we use $\text{UNNEST}(\text{MAP}_{\text{DEC}_{k,j}}(E))$, and 7 can be expressed by applying the DB–IFP operator. Finally, projection (using MAP) is used for 8. ▮

[17] Recall from the definition of calculus queries that x is the **only** free variable in ϕ.

It follows that

THEOREM 8.3. *If all the interpreted functions are invertible then the strict domain independent calculus, the safe calculus, and the DB-algebra are equivalent.*

Note that if rule 7 above is not used, then the query can be expressed by the pure algebra alone. Again, this observation generalizes the one presented in [1] where it was shown that for nested relations and sets, if 7 is not used, then the powerset operator is not needed.

8.2. *Safe Deduction*

We now consider deduction. We say that a horn clause is **safe** if it is of the form $\phi \to R_i(x)$, where ϕ is a range formula restricting x.[18] Bodies of safe horn clauses are constructed using rules 2–6 of Definition 8.1 above.

DEFINITION 8.4. We say that a deductive program P is **safe**, iff all its clauses are safe. We say that P is **strictly-safe** iff it is safe and, also, it is possible to divide its rules into layers, such that a Horn-clause H of the form

$$\ldots, y = \exp(x_1, \ldots, x_k), \ldots \to R_i(y)$$

(where exp contains some function that is not a constructor of a complex type), is in layer i iff all the rules that define the predicates in the body of H are in layers j, where $j < i$.

Safe deductive queries are domain independent with respect to the whole initial model. Strict safety implies that the only elements used in the query computation are elements from the database window, or elements constructed from those using at most k function applications, (where k is finite and bounded by the number of function applications in P). Thus, strictly safe queries are strict DB-domain independent.

It turns out that the safety syntactic restrictions are general enough to capture all (strict) domain-independent deductive queries.

PROPOSITION 8.5. *Every* (*strict DB-*)*domain independent deductive query has an equivalent* (*strictly*) *safe query. Moreover, if the first query is stratified, then so is the equivalent query.*

Proof. The proof follows from the observation that restricting the variables of a (strict DB domain-independent query to range only over elements from the (DB-window) initial model, does not change the query result. To convert a domain-independent deductive query into a safe one, we restrict all the variables in the body of the rules. We start by considering general domain-independent queries. Let P be such a program and let $s_1, \ldots, s_n$ be the types of the free variables in P. We first define for every sort s_i a predicate S_i that is satisfied by all elements of the initial model of type s_i. (The deductive program defining such predicates is presented in the proof of Theorem 7.1.) Then, we replace every rule $\phi \to R(x_i)$ of P with variables $x_1, \ldots, x_n$, by the rule $S_1(x_1) \wedge , \ldots, S_n(x_n) \wedge \phi \to R(x_i)$. Clearly the new program is safe, and since the original program is domain independent, the two programs are equivalent.

Now consider strict DB-domain-independent queries. Such queries use only elements from the k-closure of the database window. Thus, instead of restricting the variables in P by predicates that are satisfied by the whole initial model, it is sufficient to restrict them by predicates that are only satisfied by the k-closure. The predicates are defined as follows: The predicates that represent the relevant part of the domain of the atomic types are defined using rules of the form $R(y) \wedge x \in \mathrm{DEC}_{ji}(y) \to S_i(x)$. The predicates of the complex types are defined as before. Now, to close the sets under k function applications, we introduce for each sort s_i, k new predicates $S_i^1, \ldots, S_i^k$ and define their content using rules of the form:

$$S_i^j(x) \to S_i^{j+1}(x)$$

$$S_1^j(x_1) \wedge \cdots \wedge S_n^j(x_n) \wedge y = f(x_{i_1}, \ldots, x_{i_m}) \to S_i^{j+1}(y)$$

$$S_i^j(x) \wedge x = f(y_1, \ldots, y_m) \to S_k^{j+1}(y_l).$$

Finally, we modify the rules of P as above, but instead of using the S_i's we use the S_i^k's. Note that negation is not used in the translation; thus stratification is preserved. ∎

8.3. *Deduction and Algebra*

We have shown in the previous section that *S-algebra*, *IFP-algebra*, and *algebra*$^=$ queries have equivalent deductive queries. A careful examination of the translation process shows that these deductive queries are safe. A similar translation technique can be applied to DB-algebra queries for generating deductive queries that are strictly safe. Thus, (*DB-algebra*) *S-algebra*, *IFP-algebra*, *algebra*$^=$ query has an equivalent (strictly) safe deductive query. But what about the other direction? Do (strictly) safe deductive queries have equivalent algebra queries?

It was shown in [1] that every stratified safe deductive query using tuples and sets with nesting height i has an equivalent algebra query with nesting height $i+1$. The construction is based on generating for each type a set containing all the subsets of elements of this type. Then, all the subsets satisfying the program rules are selected and intersected in order to get the minimal answer. A similar construction can be done in our case: The sets containing subsets of the relevant part of the domain can be generated using the S–IFP operator (for safe queries) or the DB–IFP

[18] Note that a ground formula of the form $R(\exp\ )$ can be represented by the safe rule $(x = \exp\ ) \to R(x)$, thus there is no loss of generality.

operator (for strictly safe queries). The rest of the construction is the same as above. Thus, every (strictly) safe deductive query can be represented by a (DB)S–IFP algebra query.

Note, however, that since the S–IFP and DB–IFP operators may generate infinite sets, the resulting algebra may have to manipulate infinite sets, even if the result of the deductive query is finite. Thus, an interesting question is whether a more efficient translation exists. Also note that while the calculus queries use only the types in *SPEC*, the equivalent algebra queries use sets of those types, thus they may have to extend the type system. Therefore, another question is whether equivalence can be achieved using only the types of *SPEC*. These questions are the subject of the rest of this section.

We show in the following that for every (stratified) safe deductive query there exists an (IFP-algebra) *algebra*$^=$ query that satisfies both the above requirements. The translation process is based on defining for every predicate in the program a *simulation function* that computes the extension of the predicate obtained by a single derivation of the program rules and then using this function to simulate the full derivation of the predicate.

Let $R_1, ..., R_m$ be the names of the database relations, and let P be some a safe deductive program that uses the database relations for defining the derived predicates $P_1, ..., P_n$. Let $\phi_1 \to P_i(x), ..., \phi_m \to P_i(x)$ be the rules used in P for defining the derived predicate P_i. A single derivation of the rules of P_i can be represented by the calculus query $Q^c = \{x \mid \exists y_1, ..., y_k(\phi_1 \vee \cdots \vee \phi_m)\}$ (where $y_1 \cdots y_k$ are the free variables, other than x, in $\phi_1, ..., \phi_m$). Let $Q^a = \exp_i(P_1^a, ..., P_n^a, R_1^a, ..., R_m^a)$ be the corresponding *algebra* query. (From the proof of Proposition 8.2 it is clear that such an algebra query exists). We call $\exp_i$ the **simulation function** of P_i.

Clearly, $\exp_1, ..., \exp_n$ simulate one step in the (simultaneous) derivation of $P_1, ..., P_n$, respectively. To simulate the complete valid computation, we define each set P_i^a to be the fixed point of its corresponding simulation function $\exp_i$, i.e.,

$$P_i^a = \exp_i(P_1^a, ..., P_n^a, R_1^a, ..., R_m^a).$$

Note that all the operators in $\exp_i$ are algebra operators; thus the program defining the sets $P_1^a, ..., P_n^a$ is an *algebra*$^=$ program. For every instantiation for the database relations, the sets P_i^a defined by the above equations contain exactly the elements satisfying the corresponding predicates P_i. This follows from the fact that $\exp_i$ simulates the derivations of the corresponding predicate P_i in P. Thus every well-founded computation of P deriving $P_i(a)$ can be simulated by a well-founded computation of the corresponding algebra program deriving $\text{MEM}(a, P_i^a) = T$. (The proof is by induction of the length of the well-founded computation. It is tedious and not deep, and it is left to the reader). It follows that $P_i(a)$ hold in the well-founded model of the deductive program iff $\text{MEM}(a, P_i^a) = T$ in the well-founded interpretation of the algebra program, and also $\text{MEM}(a, P_i^a) = F$ iff $\neg R_a(a)$ hold. If follows that

THEOREM 8.6. *Every safe deductive program has an equivalent algebra*$^=$ *program.*

8.3.1. *Programs without negation.* We next consider deductive programs without negation. The above translation process is, of course, valid for such programs. However, since the well-founded model of such programs is the same as the model obtained by an inflationary fixed-point computation, we can also simulate the programs computation using the IFP operator (i.e., use the IFP-*algebra* instead of the more powerful *algebra*$^=$).

The expressions $\exp_i(x_1, ..., x_n, R_1^a, ..., R_m^a)$, $i = 1 \cdots n$, represent one step in the derivation of the predicate P_i. To simulate the fixed-point derivation if P_i we compute the mutual fixed points of the $\exp_i$'s w.r.t. the variables $x_1, ..., x_n$. The set computed by the expression $\text{IFP}^n_{\exp_i}(R_1^a, ..., R_m^a)$, $i = 1 \cdots n$, then contain exactly the elements which satisfy the corresponding P_i, i.e., for every instantiation of the database relations, $\text{MEM}(a, \text{IFP}^n_{\exp_i}(R_1^a, ..., R_m^a)) = T$ iff $P_i(a)$ is satisfied and $\text{MEM}(a, \text{IFP}^n_{\exp_i}(R_1^a, ..., R_m^a)) = F$ iff $\neg P_i(a)$. Moreover, if P has a finite fixed point, then the sets are "real" finite sets. Thus, the negation is not needed in their specification and an initial model semantics alone is sufficient.

8.3.2. *Stratified programs.* Consider stratified programs. The predicates in such programs can be divided into groups $G_1, ..., G_k$, s.t. (i) a predicate P_i is defined by a rule, where P_j is negated in the body, iff P_j belongs to a group with index smaller than that of P_i, and (ii) is defined by a rule, where P_j appears positively in the body, iff P_j belongs to a group with index smaller than or equal to that of P_i.

The well-founded model of a stratified program P is the same as the model obtained by successively computing the fixed point of the rules defining the predicates in the first group, then the second, etc. Thus, again, instead of using the *algebra*$^=$ program to simulate the well-founded computation of P, we can simulate the fixed-point derivation using IFP.

Let P_i be a predicate in the jth group. Since the program is stratified, all the predicates in the body, the rules defining P_i belong to groups with indexes smaller or equal to j. Thus, the algebra expression that simulates one step in the derivation of R_i has the form $\exp_i(x_1, ..., x_l, R_1^a, ..., R_m^a)$, where $x_1, ..., x_l$ correspond to predicates in groups with indices $\leqslant j$. To capture the idea that derivations of predicates in the jth group are computed based on the fixed points of previous groups, we replace all the variables that correspond to predicates in previous groups by sets

representing their fixed points, i.e., the simulation function of P_i is changed to

$$\exp_i(\mathrm{IFP}^{n_1}_{\exp_1}(R^a_1, ..., R^a_m), ..., \mathrm{IFP}^{n_q}_{\exp_q}(R^a_1, ..., R^a_m), x_{q+1}, ..., x_l, R^a_1, ..., R^a_m),$$

where $P_1, ..., P_q$ are predicates in groups with indices $< j$, $x_{q+1}, ..., x_l$ are variables that correspond to predicates in the jth group, and $R^a_1, ..., R^a_m$ are the names of the database relations. To simulate the fixed point derivation of the P_i's in the jth group we compute the mutual fixed points of the corresponding (updated) $\exp_i$'s. Note that, although subtraction may appear in $\exp_i$ (for handling the negated predicates), no variables are being subtracted. This follows from the fact that the program is stratified, and the only sets being subtracted are fixed points of previous groups. It follows that

THEOREM 8.7. *Every stratified safe deductive program has an equivalent* **positive IFP-algebra** *program. Moreover, if all the layers of the deductive program have finite fixed points, then all the sets used in the computation of the algebra program are finite.*

From the above discussion, and from Propositions 7.5, 7.6, 8.5, and Theorem 8.6 we have that

THEOREM 8.8. (i) *The stratified d.i. deductive language, the stratified safe deduction, and the positive* IFP-*algebra are equivalent.*

(ii) *The domain-independent deductive language, the safe deduction, the algebra*$^=$*, and the* IFP-*algebra*$^=$ *are equivalent.*

We can now further discuss database definitions. The above translation process can be applied to deductive programs defining database contents as well. In such programs all the relations are derived; thus the equivalent algebra expressions have no input parameters.

COROLLARY 8.9. *Every* (*stratified*) *domain-independent deductive style database definition* DB^p *has an equivalent functional style definition* DB^a *that contains only* (*positive* IFP-*algebra*) *algebra*$^=$ *operations.*

The close relationship between the various algebras and deductive programming indicates that algebra queries are as hard as deductive ones. Given an element a and a program P defining a set P^a, we call the problem of checking whether $\mathrm{MEM}(a, P^a)$ equals T or F the *membership testing problem.*

PROPOSITION 8.10. *The* **membership testing problem** *is undecidable for the positive* IFP-*algebra, and for the algebra*$^=$*. Moreover, it remains undecidable even if we consider only algebra*$^=$ *programs that have a well-founded model.*

Proof. The proof of the first claim follows immediately from Theorem 8.8 and from the fact that the problem of checking whether $P(a)$ holds for some predicate P defined by a deductive program using tuples and functions is undecidable. For the second claim, recall that every IFP-*algebra* program can be expressed by an *algebra*$^=$ program. Since IFP-*algebra* program have 2-valued interpretation, the undecidability result holds even if we consider only the 2-valued portion of the *algebra*$^=$. ∎

9. CONCLUSIONS

This paper deals with the relative expressive power of functional, algebraic, calculus, and deductive languages. The fundamental relationships between these paradigms was investigated, without depending on the particular features or properties of specific data types. The main results are the following:

We considered a basic algebra with the operations $\cup$, $-$, $\times$, σ, MAP, and several possible extensions. We showed that domain-independent calculus queries are expressible in the algebra, providing that it is augmented with operations that enable generating the part of the domain needed for the computation. We also showed that this augmented algebra can be expressed by the domain-independent calculus, providing the complex types used in queries have well-defined structure and the type system is rich enough so that every complex complex can be decomposed into its atomic elements.

We next considered recursive computations. We showed that the algebra, augmented with a positive fixed-point operator, has the same expressive power as stratified deduction. We do not know whether using a general (not necessarily positive) fixed point further increases the expressive power. An alternative approach to recursion is augmenting the algebra with the capability of using (possibly recursive) equations. This extended language was proved to be richer than the fixed point algebra since it can express the IFP operator, but while the IFP-algebra can only describe 2-valued computations, the *algebra*$^=$ can define partially specified (3-valued) sets. We also proved that this extended language has the same expressive power as general (not necessarily stratified) deduction.

The results presented in this work show what components of the classical proofs are essential for proving equivalence and by that some of the fundamental requirements and restrictions needed to proved such equivalence results are clarified.

APPENDIX

We present here a proof for Theorem 4.1.

Proof. We first define the notion of subtraction height and then use it to prove that the deductive version of *SPEC*

has a 2-valued well-founded model. Informally, the subtraction height of an expression exp is the number of subtractions of (potentially) infinite sets needed in computing its value.

DEFINITION 10.1. Let EXP denote the set of all ground terms of set type in the language of *SPEC*; The **subtraction height** of exp $\in$ EXP, denoted by $h(\text{exp})$ is defined as follows:

- $h(\text{EMPTY}) = 0$
- $h(f(\text{exp}_1, ..., \text{exp}_n)) = \max(h(\text{exp}_{i_1}), ..., h(\text{exp}_{i_k}))$ if f is an operator other than $-$, IFP, IFP′, F, and exp_{i_j} are the parameters of set type of f.
- $h(\text{exp}_1 - \text{exp}_2)) = \max(h(\text{exp}_1), 1 + h(\text{exp}_2))$ if IFP, or IFP′ appear in exp_2. $= h(\text{exp}_1)$ Otherwise.
- $h(F_{\text{exp}, x_0}(0, ...)) = 0$
- $h(F_{\text{exp}, x_0}(\text{SUCC}(i), ...) = h(F_{\text{exp}, x_0}(i, ...) \cup \text{exp}(F_{\text{exp}, x_0}(i, ...), ...))$
- $h(\text{IFP}_{\text{exp}, x_0}(...)) = h(\text{IFP}'_{\text{exp}, x_0}(i, ...)) = \lim(h(F_{\text{exp}, x_0}(i, ...)).$

Note that if IFP does not appear in exp then $h(\text{exp}) = 0$. If IFP is subtracted once, the subtraction height is 1. If such an expression is being subtracted, the height becomes 2, etc. If IFP is applied on some exp not containing IFP, then $h(\text{IFP}_{\text{exp}}) = 0$, but if IFP appears in exp (even only once), the height may become ω. Subtracting such a fixed-point expression increases the height to $\omega + 1$. Applying IFP on exp with height of some ordinal, may result in ω + that ordinal.

We use h to show that *SPEC* has a well-founded initial model. We first consider the "deductive" version of *SPEC*. As explained in Section 2.2 we call the well-founded model of this deductive program the *well-founded interpretation* of *SPEC*. We first show that the well-founded interpretation is 2-valued; i.e., every pair of terms of the same sort is either in $\mathscr{T}$ or in $\mathscr{F}$. This implies that *SPEC* has an initial well-founded model.

Let the well-founded interpretation of *SPEC* be $\mathscr{T}$, $\mathscr{F}$, and assume it is not 2-valued. Since the only rule using negation is (mem-F), the only expressions for which the equality state may be undefined are of the form $\text{MEM}(e, s) = a$, where a is either T or F. Note that all the expressions in which IFP is not used represent finite sets; thus membership is fully defined for them. Thus, all the expressions for which equality is not fully defined contain IFP. The equality status of $\text{MEM}(e, s)$ (i.e., equality to T or F) is undefined iff starting from $\mathscr{T}$ and using the rules of *SPEC* and the equality axioms (transitivity, commutativity, and substitution), it is possible to derive $\text{MEM}(e, s) = T$ using facts in the complement of $\mathscr{T}$ negatively (thus $\text{MEM}(e, s) = T$ cannot be added to $\mathscr{F}$), but it is not possible to derive this fact by using only facts from $\mathscr{F}$ negatively (thus $\text{MEM}(e, s) = T$ cannot be added to $\mathscr{T}$).

Let us consider the structure of a derivation of $\text{MEM}(e, s) = T$ that uses some fact not in $\mathscr{F}$ negatively. Suppose we assume that $\text{MEM}(e, s) \neq T$ for some s with subtraction height k, where $\mathscr{T}$, $\mathscr{F}$ make no claim about the value of $\text{MEM}(e, s)$. The only equation that can use this fact to derive a new fact is (mem-F), and using it $\text{MEM}(e, s) = F$ can be derived. There are two possible ways to proceed. We can either use the equality axioms and derive new equalities of the form $\text{MEM}(e, s) = \text{MEM}(e', s')$, or we can use the equations of *SPEC*. The only equation in which the value of $\text{MEM}(x, y)$ is used is

$$\text{INS}(a, x) - y = \text{if MEM}(a, y) \text{ then } x - y \text{ else INS}(a, (x - y)).$$

Thus, by having $\text{MEM}(e, s) = F$ we can derive new facts of the form $\text{INS}(e, s') - s = \text{INS}(e, (s' - s))$. Note that since s contains IFP, the subtraction height of the expressions on the two sides of equation is $k + 1$.

Now, what else can be derived using the above equalities? First consider expressions of *set* type. Again, we can either use the equality axioms, or the equations of *SPEC*. A careful examination of the axioms and the equations of *SPEC* shows that all the new facts that can be derived have the form $\text{exp}_1 = \text{exp}_2$, where the subtraction height of at least on of the two expressions is $\geqslant k + 1$. For example using the union equation $\text{INS}(a, x) \cup y = \text{INS}(a, (x \cup y))$, and the substitution axiom, we can derive new facts

$$\text{INS}(a, s'') \cup (\text{INS}(e, s') - s) = \text{INS}(a, (s'' \cup \text{INS}(e, (s' - s)))).$$

The above claim can be proved by induction on the size of the expressions. The proof is tedious, is not too deep, and is left to the reader.

Another type of new facts that can be derived using the above facts are equalities between terms of the form $\text{MEM}(a, b)$ and T or F. The only equations that can be used to derive such facts are $\text{MEM}(d, \text{EMPTY}) = F$ and $\text{MEM}(d, \text{INS}(d', x)) =$ if $\text{EQ}(d, d')$ then T else $\text{MEM}(d, x)$. The new equalities derivable using these two equations and the facts derived so far, are of the form $\text{MEM}(a, b) = T$, or $\text{MEM}(a, b) = F$, where $h(b) \geqslant k + 1$. From similar arguments as above, these new facts can again be used to derive equalities among *set* terms of height $\geqslant k + 2$, and $\text{MEM}(a, b) = T/F$ for b's with height $\geqslant k + 2$, and so on.

From the above discussion it follows that by assuming $\text{MEM}(e, s) \neq T$ for some s with subtraction height k, all the new facts $\text{MEM}(a, b) = T$ that can be derived have b of height $> k$.

Now, let us consider again the well-founded interpretation of *SPEC*. Let s be a *set* expression of minimal height (which may be an infinite ordinal), such that for some e, MEM(e, s) is undefined, and let $h(s) = k$. From the fact that by assuming MEM$(e, s') \neq T$ for any s' with height $\geqslant k$ only equalities for terms with height $> k$ can be derived, it follows that MEM$(e, s) = T$ cannot be derived from $\mathscr{T}$, even if facts not in $\mathscr{T}$ are used negatively. Thus MEM$(e, s) = T$ can be added to $\mathscr{F}$, and an immediate consequence MEM$(e, s) = F$ can be added to $\mathscr{T}$. This contradicts the assumption that the value of MEM(e, s) is undefined. Thus no such set s exists, MEM is a total function, and the well-founded interpretation is 2-valued.

It still remains to show correctness. For that, we need to show that **bool** is protected and that all operations interact in the expected way. Note that as MEM is total, we need to show that it does not have both the values T, F on some element of the algebra. Once this is proved, showing that the operations interact as expected is straightforward. The proof follows the lines above and is left to the reader. ▮

REFERENCES

1. S. Abiteboul and C. Beeri, On the power of languages for the manipulation of complex objects, INRIA research report, 1988; *VLDB J.*, to appear.
2. K. R. Apt, H. Blair, and A. Walker, Towards a theory of declarative knowledge, *in* "Foundations of Deductive Databases and Logic Programming" (J. Minker, Ed.), Morgan Kaufmann, Los Altos, CA, 1988.
3. M. P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, The object-oriented database system manifest, *in* "First Int'l Conf. on Deductive and Object Oriented Databases" (W. Kim, J.-M. Nicolas, and S. Nishio, Eds.), pp. 40–57, 1989.
4. C. Beeri, Formal models for object oriented database systems, *in* "First Int'l Conf. on Deductive and Object Oriented Databases, 1989" (W. Kim, J.-M. Nicolas, and S. Nishio, Eds.), pp. 370–395.
5. C. Beeri, New data models and languages, *in* "Proceedings, 11th Symp. on Principles of Database Systems-PODS, San Diego, California, 1992," pp. 1–15.
6. C. Beeri and T. Milo, Subtyping in OODB's, *in* "Proceedings, 10th PODS, Denver, CO, 1991," *J. Comput. System Sci.* **51**, No. 2 (1995), 223–243.
7. C. Beeri and T. Milo, Functional and predicative programming in OODB's, *in* "Proceedings, 11th PODS, San Diego, CA, 1992."
8. C. Beeri and T. Milo, On the power of algebras with recursion, *in* "SIGMOD 93, Washington, DC, 1993."
9. V. Breazu-Tannen, P. Buneman, and S. Naqvi, Structural recursion as a query language, *in* "Conf. on Database Programming Languages, DBPL, 1991."
10. V. Breazu-Tannen, P. Buneman, and L. Wong, Naturally embedded query languages, *in* "Proceedings, 4th Int'l Conf. on Database Theory (ICDT), Berlin, Germany, 1992," pp. 140–154.
11. C. Beeri, R. Ramamkrishnan, D. Srivastava, and S. Sudarshan, The valid model semantics for logic programs, *in* "Proceedings, 11th Symp. on Principles of Database Systems—PODS, San Diego, 1992," pp. 91–104.
12. A. Chandra and D. Harel, Horn clause queries and generalizations, *J. Logic Programming* **2**, No. 1 (1985), 1–15.
13. S. Grumbach and V. Vianu, Tractable query languages for complex object databases, *in* "Proceedings, 10th PODS, Denver, CO, 1991."
14. M. Escobar-Molano, R. Hull, and D. Jacobs, Safety and translation of calculus queries with scalar functions, *in* "Proceedings, 12th Symp. on Principles of Database Systems—PODS, Washington, DC, 1993," pp. 253–264.
15. H. Ehrig and B. Mahr, "Fundamentals of Algebraic Specifications I, Equations and Initial Semantics," EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer-Verlag, Berlin, 1985.
16. H. Ehrig, A. Sernadas, and C. Sernadas, Objects, object types, and object identity, *in* "Categorical Methods in Computer Science with Aspects from Topology" (H. Ehrig *et al.*, Eds.), pp. 142–156, Springer-Verlag, New York/Berlin, 1989.
17. M. Gelfhond and V. Lifschitz, The stable model semantics for logic programming, *in* "Proceedings, 5th Int. Conf. and Symp. on Logic Programming, 1988."
18. J. Gougen and J. Meseguer, Unifying functional, object-oriented and relational programming with logical semantics, *in* "Research Directions in Object-Oriented Programming" (B. Shriver and P. Wegner, Eds.), pp. 417–479, MIT Press, Cambridge, MA, 1987.
19. J. Gougen, J. Tatcher, and E. Wagner, An initial algebra approach to the specification correctness and implementation of abstract data type, *in* "Current Trends in programming Methodology" (R. T. Yeh, Ed.), Vol. 4, pp. 80–150, Prentice–Hall, Englewood Cliffs, NJ, 1978.
20. R. Hull and J. Su, On the expressive power of database queries with intermediate types, *J. Comput. System Sci.* **43**, No. 1 (1991).
21. N. Immerman, S. Pantaik, and D. Stemple, The expressiveness of family of finite set languages, *in* "Proceedings, 10th PODS, Denver, CO, 1991."
22. S. Kaplan, Positive/negative conditional rewriting, *in* "Proceedings, 1st Int'l Workshop on Conditional term rewriting Systems, Orsay," Lect. Notes in Comput. Sci., Vol. 308, pp. 129–143, Springer-Verlag, New York/Berlin, 1988.
23. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, Constraint query languages, *in* "Proceedings, 9th ACM PODS, 1990," pp. 299–313.
24. M. Kifer, On safety, domain independence, and capturability, *in* "3rd Int. Conf. On Data and Knowledge Bases: Improving usability and Responsiveness, Jerusalem, Israel, 1988."
25. G. Kuper and M. Y. Vardi, The logical data model, *ACM Trans. Database Systems (3)* **18** (1993), 379–413.
26. C. K. Mohan and M. K. Srivas, Conditional specifications with inequational assumptions, *in* "Proceedings, 1st Int'l Workshop on Conditional term Rewriting Systems, Orsay," Lect. Notes in Comput. Sci., Vol. 308, pp. 161–178, Springer-Verlag, New York/Berlin, 1988.
27. T. Milo, "Formalisms for describing OODB's," Ph.D. thesis, Hebrew University, Jerusalem, 1992.
28. A. Poulovassilis and C. Small, A functional programming approach to deductive databases, *in* "Proceedings, VLDB 91, Barcelona, Spain, 1991."
29. K. Ross, Relations with relation names as arguments: Algebra and calculus, *in* "Proceedings, 11th Symp. on Principles of Database Systems—PODS, San Diego, CA, 1992," pp. 1–15.
30. A. Selman, Completeness of calculi for axiomatically defined class of algebras, *Algebra Universalis* **2** (1972), 20–32.
31. A. Sernadas, C. Sernadas, and H. Ehrig, Object oriented specification of databases: An algebraic approach, *in* "13th VLDB, 1987" (P. Hammersley, Ed.), pp. 107–116.
32. J. W. Tatcher, E. G. Wagner, and J. B. Wright, Data type specification: Parameterization and the power of specification techniques, *ACM Trans. Programming Languages Systems* **4**, No. 4 (1982).
33. A. Van-Gelder, Negation as failure using tight derivation for general logic programming, *in* "Proceedings, 3rd IEEE Symp. on Logic Programming, Salt Lake City, Utah, Sept. 1986."

34. A. Van-Gelder, K. A. Ross, and J. S. Shlipf, Unfounded sets and well founded semantics for general logic programming, *in* "Proceedings, 7th ACM Symp. on Principles of Database Systems, 1988."
35. J. D. Ullman, "Principles of Database Systems," 2nd ed., Computer Sci. Press, New York, 1982.
36. R. Wieringa and R. Van de Riet, Algebraic specification of object dynamics in knowledge base domains, *in* "The Role of Artificial Intelligence in Databases and Information Systems" (R. Meersman, Z. Shi, and C. Kung, Eds.), pp. 411–436, Amsterdam, North-Holland, 1990.