On the Power of Algebras with Recursion *

Catriel Beeri
The Hebrew University †

Tova Milo
I.N.R.I.A ‡

Abstract

We consider the relationship between the deductive and the functional/algebraic query language paradigms. Previous works considered this subject for a non-recursive algebra, or an algebra with a fixed point operation, and the corresponding class of deductive queries is that defined by stratified programs. We consider here algebraic languages extended by general recursive definitions. We also consider languages that allow non-restricted use of negation. It turns out that recursion and negation in the algebraic paradigm need to be studied together. The semantics used for the comparison is the valid semantics, although other well-known declarative semantics can also be used to derive similar results. We show that the class of queries expressed by general deduction with negation can be captured using algebra with recursive definitions.

1 Introduction

Declarative languages for object oriented databases are a central subject in database research [1, 3, 5, 13, 14, 15, 16]. The three language paradigms being considered are algebra, calculus and deduction-based. There has been a lot of work investigating the relationship between those paradigms [1, 5, 18, 22]. Many of these works (including ours) considered the relationship between algebra and deduction, presenting equivalence results about the expressive power of the two paradigms. However, all these works (again, including ours) investigated this relationship only for a rather restricted class of deductive programs: the **stratified** programs, and algebras with recursion in the form of an explicit fixed point operation. It is of interest to extend the algebraic paradigm with a general facility for recursive definitions, and investigate the relationship of the resulting language to deductive languages. This is the main subject of the paper.

Recently, much attention is paid to defining a declarative semantics for negation in non stratified deductive programs [8, 2, 23, 24, 11]. There have been several suggestions for defining semantics for such programs. For example, the *well-founded* semantics[24] and the *stable-model* semantics [11] provide an intuitive semantics for logic programs with negation. A recent proposal, the *valid* semantics, extends the well founded semantics in a natural manner [6]. Surprisingly, the question whether the class of queries defined using such semantics can also be captured in the algebraic paradigm, has not yet been considered. This question is also a main subject of this work.

As in previous works [4, 5], we use the algebraic specification paradigm as a formal framework for the algebraic paradigm. This framework allows one to define a variety of models with rich data structuring facilities [9, 10, 12, 25]. However, as currently used, it is limited since it allows only to use positive facts (when negation is used, the semantics of a specification is no longer well-defined[17, 19]). We explain why negation need to be considered for algebraic query languages even without recursion, certainly with recursion. In order to support negation as well, we extend the paradigm, using the valid model approach. We base the investigation on this paradigm because of its simplicity and generality, The results can be easily adjusted to capture other declarative semantics as well. Using this extended framework, we define two algebraic query languages, that are essentially extensions of the fixed point algebra presented in [5] with recursive definitions. We investigate these algebras and in particular concentrate on the relationship between them and deduction. The same approach to semantics, namely the valid semantics, is used for both paradigms.

We have shown in [5] that the fixed point algebra has the same expressive power as stratified deduction. The two extended algebras presented here are proved to be more expressive than the fixed point algebra. In particular, we show that they both expresses exactly the class of queries captured by general deductive programs, under the valid model semantics.

The main contribution of this paper is in providing for a better understanding of the relationship between the deductive and the functional/algebraic paradigms. The results are potentially of interest to the area of algebraic specifications, by showing that data types can be specified in a language that uses negation. However,

*This research was supported by a grant from the United States-Israel Binational Science Foundation (BSF), and by the Chateaubriand scholarship.

†Department of Computer Science, The Hebrew University, Jerusalem 91904, ISRAEL, beeri@cs.huji.ac.il

‡Currently at University of Toronto, Department of Computer Science, Toronto, Ontario, Canada M5S 1A4, milo@db.toronto.edu

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0377...$1.50

we do not pursue this direction here.

In Section 2, we present the main concepts of algebraic specifications, and the extensions needed to handle negation. In section 3 we present the general functional language, a restricted fixed point algebra, and two versions of algebras with recursive definitions, the $algebra^{=}$ and the $\mathrm{IFP}-algebra^{=}$. The deductive language is introduced in section 4, and its relationship to the above algebras is investigated in the next two sections. Conclusions are presented in section 7.

2 Algebraic Specifications

This section presents some of the main concepts of algebraic specifications [9], and the extensions needed to support negation.

2.1 Basic Concepts

We start by considering specifications without negation. A specification defines a collection of related data types. It contains a list 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.*

A specification defines many models, which are *many-sorted* algebras. Since it is expected to specify a unique meaning, (the equivalence class of) one algebra is selected as its meaning. This is most often the *initial algebra*, and we assume so in this paper. This 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 [9]; in particular, for specifications given by (conditional) equations it exists, and whenever it exists it is unique (up to isomorphism). It structure is also known: The Herbrand universe, the collection of ground terms over OP, can be made an (S, OP)-algebra, and its quotient modulo the invariance relation defined by E, the **quotient term algebra**, is an initial algebra. This approach is closely related to that of *minimal model* used in logic programming. Indeed, we have here the case of a single predicate, namely equality, defined by Horn-clauses (equations and conditional equations). The invariance relation defined by them is the minimal model of the equality predicate.

Essentially all known data types, including atomic types like the characters, the integers, the booleans, and structured types like sets, lists, stacks, and so on, can be so defined. The following example specifies finite sets of natural numbers (assuming that natural numbers and booleans have already been defined):

$\underline{\mathbf{SET(nat)}} = \mathbf{nat} + \mathbf{bool} +$

sorts : $set(nat)$

opns :
EMPTY :	$\rightarrow set(nat)$	(empty set)
INS :	$nat, set(nat) \rightarrow set(nat)$	(insert)
MEM :	$nat, set(nat) \rightarrow bool$	(member)

eqns :
$d, d' \in nat$, $s \in set(nat)$
$\mathrm{INS}(d, \mathrm{INS}(a, s)) = \mathrm{INS}(d, s)$
$\mathrm{INS}(d, \mathrm{INS}(d', s)) = \mathrm{INS}(d', \mathrm{INS}(d, s))$
$\mathrm{MEM}(d, \mathrm{EMPTY}) = \mathrm{FALSE}$
$\mathrm{MEM}(d, \mathrm{INS}(d', s)) =$ IF $EQ(d, d')$
THEN TRUE
ELSE $\mathrm{MEM}(d, s)$

The notation $nat{+}bool{+}\ldots$ means that these previously defined specifications are imported. The initial algebra has sorts and operations for natural numbers, booleans, and sets of natural numbers, with a constant denoting the empty set, and operations for insertion, and membership testing. Note that no special property of the natural numbers is used in the specification (except for the fact that equality is defined on them)[1]. By replacing *nat* with a type variable *data*, we obtain a *parameterized specification*, which can be instantiated by substituting a concrete type for *data*. For brevity, we use in the rest of this paper the notation $\{x_1, \ldots, x_n\}$ to represent the expression $\mathrm{INS}(x_1, \ldots, \mathrm{INS}(x_n, \mathrm{EMPTY}))$, and $\{t\}$ to represent the type of sets with members of type t.

Many other operations on sets can be defined using equations, including the familiar relational algebraic operations, and generalizations thereof [5, 7].

The above specification of sets illustrates a weakness of algebraic specifications, compared to logic programs. For the latter, the standard notion of model from logic describes sets in the form of relations (predicates), and membership, as built-in notions. The content of a relation can be defined by a Horn-clause program. The tuples in the relations are those derived from the program. Membership testing for each such tuple relative to the corresponding predicate returns *true* by definition. For a tuple that is not in the relation in the minimal model, the answer is *false*. Note that the programmer must only describe the tuples that are in the relations. The fact that other tuples are not in, follows immediately from the minimal model semantics. This is the only point where a default assumption, namely that of using the minimal model, is used. Now, a similar default assumption, namely that of using the initial model, is used in algebraic specifications. However, it applies only to the extension of the equality predicate. No general notion of predicate is built-in, so we have to represent other predicates as sets, and membership as a boolean-valued function. The booleans are regular values, included in the specification and its model. Hence, to define properly a boolean valued function like membership, it is not sufficient to provide the positive (i.e., *true*) facts. The negative (i.e., *false*) facts must also be provided (since now *true*, *false* are just regular values, and the default mechanisms does not apply.) For this reason, one can define finite sets,

[1] In general, a specification for sets with element type *type* can contain the MEM 'predicate' iff equality is definable on *type* [21].

as above, since both positive and negative membership facts can be specified, but not infinite sets. Note that this restriction has an associated benefit — algebraic specifications are computable, whereas negative facts in logic programs usually are not.

2.2 Negation

As we have explained, the expressive power of the above framework is limited since one can only use positive facts (i..e., equalities), and predicates and membership are not built-in. To define and investigate database languages, we need to have sets, since predicates can be represented as sets. The following example demonstrates how negation can be used in specifications, and in particular its role in defining sets.

Example 1:

The specification above defines only finite sets. We have shown in [5] that some infinite sets can also be defined in this framework. For example, the infinite set S^e of all even natural numbers can be described as an infinite union $S^e = S^e_1 \cup S^e_2 \cup S^e_3 \ldots$, where S^e_i is the set containing all natural numbers smaller than $2i$. We represent this infinite set using a new constant name S^e_c, defined using an auxiliarity function $F : nat \rightarrow \{nat\}$ that for every i returns the set S^e_i :

opns: F : nat $\rightarrow \{nat\}$
$S^{e'}_c$: nat $\rightarrow \{nat\}$
S^e_c : $\rightarrow \{nat\}$

eqns: $F(0) = \text{EMPTY}$
$F(\text{SUCC}(i)) = F(i) \cup \{2i\}$ [2]
$S^{e'}_c(i) = F(i) \cup S^{e'}_c(\text{SUCC}(i))$
$S^e_c = S^{e'}_c(1)$

Actually, an alternative, simpler, definition that does not use the auxiliary functions is available:

eqns: $S^e_c = S^e_c \cup \{2i\}$

The difference between the two alternatives is that in the second we have a declarative description of the set, and we trust the default mechanism to produce the right set. In the first, we specify more explicitly what we want to be in the set, although in the end we use the default mechanism here also. For this specific example, both definitions give the same result, but we will see later examples where this is not the case, and an important use of the first style.

It is easy to see (using term rewriting) that in the first definition for every $i \geq 1$ hold,

$$\begin{aligned} S^e_c &= F(1) \cup F(2) \cup \ldots \cup F(i) \cup S^{e'}_c(\text{SUCC}(i)) \\ &= \{0\} \cup \{0,2\} \cup \ldots \cup \{0,2,\ldots,2i-2)\} \cup \\ &\quad S^{e'}_c(\text{SUCC}(i)) \\ &= \{0,2,\ldots,2i-2\} \cup S^{e'}_c(\text{SUCC}(i))^{3} \end{aligned}$$

It follows that every even number n is inserted into S^e_c. A similar observation applies to the other definition. But, does the constant S^e_c really represent the infinite set $S^e = \{0,2,4,\ldots\}$? For example, suppose we want to check whether the number x belongs to the set S^e. Will MEM(x, S^e_c) return the correct answer? For a finite set S, MEM(x,S) defines a boolean-valued function that returns T if x is in S, and F otherwise, but for the infinite set S^e_c, MEM returns T if x is in S^e_c, but there is no derivation that produces *false* for an odd number (because EMPTY is never encountered when the content of S^e_c is scanned).

To correct the specification, we can add an equation that identifies the result of membership testing with F, whenever it can not proved to be equal T.

$$\text{MEM}(x,y) \neq T \rightarrow \text{MEM}(x,y) = F$$

□

We will later consider other operations on sets, i.e., those of algebraic languages. We certainly want these operations to interact with membership in the right way, so we take the above conditional equation as a fixed part of the specification of sets and set operations. But, this conditional equation is really a disequation — it uses negation. The standard initial model semantics can not in general be used for specifications that contain negation, since the existence of an initial model is not guaranteed for such specifications [17, 19]. Thus, an alternative default mechanism for choosing the desired algebra must be provided. 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. As already stated, there have been several suggestions for defining declarative semantics for deductive programs with negation [8, 2, 23, 24, 11, 6]. We use the **valid** model approach for developing the semantics of algebraic specifications with negation. (Similar development can be done using the other declarative approaches, as well.)

We present below a brief summary of the main ideas of the valid model semantics, and how it is used for defining semantics for algebraic specifications with negation. (For a formal definition and full investigation, see [6, 20]). The valid model of a deductive program P is a 3-valued model with a set $\mathcal{T}$ of true facts, and set $\mathcal{F}$ of false facts, and a set of undefined facts. We consider in the following only ground facts. The model is computed as follows: Initially, all the facts are undefined. At each step of the computation, we look at all the possible derivations starting from the current set $\mathcal{T}$ of true facts, where only facts not in $\mathcal{T}$ are allowed to be used negatively. The facts that are not derivable in any such computation, are assumed to be certainly false, and are therefore added to $\mathcal{F}$. The false facts in $\mathcal{F}$ and the true facts in $\mathcal{T}$ are then used to derive new true facts, that are added to $\mathcal{T}$. In this derivation, we use negatively only facts from $\mathcal{F}$. The process is repeated (possibly transfinitely) until no more true facts can be derived. The sets $\mathcal{T}$ and $\mathcal{F}$ of true and false facts obtained at the end of the process define a 3-valued model. This model is called **valid** model, and assumed to be the semantics of the program P (and since it is unique [6], the semantics is well defined.)

[2] For clarity, we use union to denote INS$(2i, F(i))$.

[3] This denotes INS$(0, \text{INS}(2, \ldots, 2i-2, S^{e'}_c(\text{SUCC}(i))))$.

A specification $SPEC$ can be viewed as a deductive program with '=' being the only predicate. The rules in the 'deductive version' of $SPEC$ are the conditional equations of $SPEC$, and the standard equality axioms (transitivity, symmetry, reflexivity, and substitution). Taking a valid model approach, the deductive version of $SPEC$ has a 3-valued valid model. Let $\mathcal{T}$ and $\mathcal{F}$ be the sets of true and false ground facts in the valid model of the 'deductive version' of $SPEC$. The facts in $\mathcal{T}$ represent terms that are (certainly) equal, the facts in $\mathcal{F}$ are (certainly) unequal terms, and the rest are equalities whose status is undefined. This is called the **valid interpretation** of $SPEC$. To illustrate the idea, consider the definition of S_c^e. One can easily see that for each even number x, the fact $\text{MEM}(x, S_c^e) = T$ can be derived immediately as being in $\mathcal{T}$, without using any negative facts. After that, no additional facts of this form (for other values of x) can ever be derived. Hence all the facts $\text{MEM}(x, S_c^e) = T$ where x is odd are put into $\mathcal{F}$ (meaning that $\text{MEM}(x, S_c^e) \neq T$). Now, we can use the rule with negation to derive that for each odd number x, the fact $\text{MEM}(x, S_c^e) = F$ is in $\mathcal{T}$. As no more derivations of true facts are possible, we can put $\text{MEM}(x, S_c^e) = T$ in $\mathcal{F}$ for each odd x, and similarly for $\text{MEM}(x, S_c^e) = F$, where x is even. Thus, for each x, precisely one of $\text{MEM}(x, S_c^e) = T$,$\text{MEM}(x, S_c^e) = F$ is in $\mathcal{T}$. Thus, in this example, negation is used essentially to implement the standard default mechanism of logic programming for MEM. More general cases will be considered later.

As mentioned above, the interpretation of $SPEC$ may be 3-valued. However, for an algebraic approach, we need to consider *total* algebras, (i.e 2-valued models). We consider the (total) algebras that agree with the valid interpretation on the *true* facts $\mathcal{T}$ to be the 'more natural' models of $SPEC$. (The same approach is used in classical specifications: the models are the algebras that agree with the derivable set of *true* facts.) We call such algebras **valid**, and define them formally as follows:

Definition 2.2 *A (total) algebra A_{SPEC} is a* **valid algebra** *of SPEC iff it is a model of SPEC, and $exp_1 = exp_2 \in \mathcal{T}$ implies that $exp_1 = exp_2$ holds in A_{SPEC}.*

Since, intuitively, a specification is supposed to specify a unique data type, a **default** mechanism must be provided to choose one valid algebra from the multitude. Following the traditional initial model approach, we choose the valid algebra such that there exists a unique homomorphism from it to each of the other valid algebras of $SPEC$. We call this algebra the **initial valid model** of $SPEC$.

Note the difference between the traditional *initial model* and the *initial valid model*. An initial model must have a unique homomorphism to any of the algebras of $SPEC$, while an initial valid model must have unique homomorphisms only to the valid algebras. (Note that when negation is not used, every algebra is a valid algebra, thus the concepts are equivalent.) This allows a valid initial model to exist in more cases.

However, there exist specifications that do not have an initial valid model.

Example 2:
Consider $SPEC$ defining a sort s with three constants a,b, and c, using the (generalized conditional) equations:
$a \neq b \rightarrow a = c$
$a \neq c \rightarrow a = b$
All the models of $SPEC$ are valid, since no equalities can be derived in a valid manner. $SPEC$ has three such models: a model where $a = b = c$, a model where $a = b \neq c$, and a model where $a = c \neq b$. However, none of these are initial. The symmetry in the two given conditional equations leads a non deterministic choice between two different, non compatible, algebras. □

Moreover, it turns out that

Proposition 2.3 .
(1) It is undecidable whether a specification with negation has an initial valid model.
(2) If only 0-ary functions are used in the specification (i.e. only constants), then the problem becomes decidable.

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 valid model semantics. The undecidability proof is rather simple and employs a reduction to the problem of proving a ground equation from $SPEC$, known to be undecidable [9].

The above result shows that the difficulty of writing specifications increases when negation is used. However, as we illustrated above (and will further explain in the next section), negation is needed for defining correctly *sets*, and *set* operators. Fortunately, it turns out that there exists a large and quite expressive family of specifications that are syntactically restricted in a way that assures the existence of initial valid model. We consider these specifications in the following section. We call specifications that have an initial valid model **well-defined**.

3 Algebras

We next present the algebraic query languages. We start by explaining how databases are defined in this framework. Then, we describe the languages.

To define a database, one has to specify the data types used in it, and then describe its content. We assume in the following that all the data types used in the database are well-defined, i.e. their specification has an initial valid model.

A database is a collection of named sets (every set is a database 'relation'). Each set is represented by a named constant, and its content is specified by (generalized conditional) equations. For example, a database relation R_i that contains elements $a_1, \ldots, a_n$ of type t_i can be represented by a constant $R_i^a :\rightarrow \{t_i\}$ that is defined using the equation $R_i^a = \{a_1, \ldots, a_n\}$. In general, equations of the form $R_i^a = exp$ can be used for defining the contents of the relations. Again, we assume that the specification of the database is well-defined. A query is represented by a constant Q that is defined using an equation of the form $Q = exp$. The query specification additionally includes definitions for

all the types and functions used in *exp*. Thus, the query language is a **functional** language.

Sometimes it is desirable to restrict the queries and use in *exp* only a specific set of predefined operators OP. For example, in the relational algebra, queries are defined only using the operators *select*, *project*, *join*, etc. We call such set of operators an *algebra*, and the queries defined using these operators are algebraic queries.

Recursion can be expressed in an algebra either by an explicit fixed point operation, or by allowing recursive algebraic definitions. Both are considered here.

3.1 The Fixed Point Algebra

In [5] we presented an algebra that generalizes the relational and complex object algebras. (See also [7].) All the operators in this algebra are generic, in the sense that they can be applied to sets containing elements of any arbitrary type (assuming that equality is definable for the type). The operators of the algebra are: $\cup$ (set union), $-$ (difference), $\times$ (cartesian product), σ_{test} (selection using a boolean valued selection function *test*), MAP_f (an operator that restructures each element of a set using restructuring function f), and an inflationary fixed point operator IFP_{exp}. The operation IFP_{exp} computes the inflationary fixed point of the algebraic expression *exp* as follows: Starting with the empty set, at each step, *exp* is applied on the result obtained in the previous step, and the result is accumulated.

All the operations are defined in [5] using parameterized specifications. Note that, though it seems that the operators σ, MAP, and IFP are generic in the functions *test*, f, and *exp*, this is misleading. Our framework is strictly first order, and function variables are **not** available, thus a special specification must be provided for every specific function. In practice, this limitation can be overcome by syntactically allowing fully (second order) genericity, but interpreting functions instantiation as a macro, i.e. a code duplication will take place.

The mechanism for defining the fixed point operator IFP is very similar to the first one used in the previous section for defining the infinite set of even numbers, i.e., the computation of the inflationary fixpoint is explicitly spelled out. We use an auxiliarity function $F_{exp}(i)$ that for every i computes the result of i successive applications of *exp*. Then we define IFP_{exp} to be the infinite union of all the $F_{exp}(i)$'s.

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 on numbers, hence the fixed point operator may generate infinite sets. Thus, as already explained, negation needs to be used in the specification of IFP to define correctly the membership function.

We use the valid model approach to define the semantic of the specifications. Since IFP is an inflationary fixed point operation, it is tempting to try to use use the inflationary fixed point semantics for handling the negation in the algebraic specification of the algebraic operations, particularly the IFP. However, this semantics is not suitable for handling specifications with negation, since it does not capture the idea that inequality should be used only when no more equalities can be derived. Thus, if we consider the disequation used in the definition of membership under inflationary semantics, it will be applied in the first step, when all sets are empty, and will put all facts of the form $\text{MEM}(x, S) = F$ into $\mathcal{T}$!

A natural question is whether the above operations, and in particular the fixed point operator are well-defined, i.e. whether the specification of the operations has an initial valid model.

Theorem 3.1 *Let SPEC be some well defined specification, defining the types $t_1 \ldots, t_n$ (with equality). There exists a a well-defined specification SPEC' that extends SPEC, that for every type t_i defines a set type $\{t_i\}$, with the operations* EMPTY, INS, MEM, $\cup$,$\times$, $-$, σ, MAP, IFP.

Intuitively, the above theorem claims that for every set S constructed using the above operations, the membership function is totally defined, (i.e. in the initial valid model, for every element a, $\text{MEM}(a, S)$ either equals T or F). The specification $SPEC'$ is the one presented in [5]. The theorem is proved by induction on the size of the expressions, based on a "local stratification" argument. We show that the membership function of sets constructed by complex expressions is totally defined in terms of membership in less complex expressions.

The algebra containing all the above operators is called the IFP–*algebra*. A restricted version, where the IFP operator is not used, is simply called the *algebra*. Since we assumed that the data types used in the database are well defined, from the above theorem it follows immediately that every IFP$-$*algebra* query on the database is well-defined.

3.2 Operations and Equations

The above algebraic paradigm can be enriched by allowing the programmer to add new operation names to the language, and use equations to define their properties. However, if no restrictions are posed, then the resulting language is essentially the general functional language. We restrict the language by allowing only operations with input and output parameters of *set* type to be defined, where for each new operation name f_i we have only one equation $f_i(x_1, \ldots, x_n) = exp(x_1, \ldots, x_n)$, and where *exp* is an algebraic expression that contains no variables other then $x_1, \ldots, x_n$. We do allow recursion: the defining expressions may contain the names of the new, defined, operations. For a given algebra (i.e., set of operations), this is an extension of the algebra with recursion.[4] The restricted framework allows one to define new operations only in terms of a specific set of predefined operators.

Example 3:
The operator $\cap : \{t\}, \{t\} \rightarrow \{t\}$ (*intersection*) can be defined using the operator $-$, and the equation $x \cap y = x - (x - y)$. Similarly, an *exclusive-or* operator $\otimes : \{t\}, \{t\} \rightarrow \{t\}$ can be defined with the equation $x \otimes y = (x - y) \cup (y - x)$. We have seen in the previous section some examples of recursive definitions, namely the definitions of S_c^e. Note that the first definition is

[4] Actually, this is slightly more general than just recursion, since the equations are not restricted in any way.

not in our restricted language (since it uses auxiliary functions on the integers). The second is closer to what we allow, but it still uses an integer variable i which is not of *set* type. Another possible definition for this constant is

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

stating that the set S_c^e is such that increasing all the members by 2, and adding the number 0 to the set, results in the original set S_c^e. Since $\{0\}$ is a constant of the algebra, this definition satisfies the restrictions. Since the definition of MAP is well-defined, we obtain again the same model, as in the previous definitions, again using the rule with negation for MEM.

As another example for recursive definition, we consider a game that was one of the examples leading to the formalism of the well-founded and stable models semantics [24]. 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 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 equations of this form may not have a well-defined model. (If the MOVE relation is acyclic then the valid interpretation is 2-valued, and an initial valid 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 recursion, *algebra*$^=$, and IFP$-$*algebra*$^=$ resp.

Clearly, if the definitions are restricted to be not recursive, 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 sub expressions, then use these names instead of the full sub expressions. If recursive definitions are used, then the above extension is no longer just syntactic sugar, it is a significant extension, and as we show below it increases the expressive power of the languages.

We first consider well-definedness (i.e. the existence of an initial valid model). It turns out that well definedness is no longer guaranteed for the extended languages. For example consider the constant S, defined by the recursive equation

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

The valid computation cannot infer that $\mathrm{MEM}(a, S) = T$ is in T. The definition of the $-$ operator (as presented in [5]) performs inversion of membership. Thus, assuming $\mathrm{MEM}(a, S) = F$, and using the definition of $-$, will allow us to derive $\mathrm{MEM}(a, S) = T$. So, there is a potential derivation of T but not a sure one. Thus, $\mathrm{MEM}(a, S) = F$ can not be inferred too. It follows that the membership status of a is S is undefined, and there is no initial valid model. (Intuitively, we would not like it to be defined: Identifying $\mathrm{MEM}(a, S)$ with T (or F) will cause the identifications of 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.

In general, a syntactic analysis is not sufficient for determining if such a program has an initial valid model.

Proposition 3.2 *It is undecidable whether an algebra*$^=$ *(*IFP$-$*algebra*$^=$*) program has an initial valid model.*

Proof: Given an *algebra*$^=$ program P, a set S defined by the program, and an element a, we construct an *algebra*$^=$ program P' such that P' has an initial valid 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 valid model, but if $a \notin S$ then $a \notin S'$, and an initial valid model for P' in, which S' is empty, exists. If follows that P' has an initial valid model iff $a \notin S$ in the model of$P$. But, in section 6 (proposition 6.3) 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 6 that *algebra*$^=$ has the same expressive power as general deductive programming under the valid semantics. Thus clearly it suffers from similar problems.

Theorem 3.1 states that IFP $-$ *algebra* programs always have initial valid 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.

Note that the program (equations) defining the IFP operator is **not** an *algebra*$^=$ program (since the auxiliarity function $F_{exp}(i)$ used in the definition has an input variable that is not of set type). However, as showed in examples 2 and 3, equations can be used to describe recursive computations. One can describe fixed point computations without explicitly using a fixed point operator. In particular, let $exp(x) : \{s\} \rightarrow \{s\}$ be an algebra expression, and let $S :\rightarrow \{s\}$ be a new operation name, defined by the equation $S = exp(S)$. Intuitively, the equation defines S to be a fixed point of exp. The definition of S_c^e in example 3 is of this kind. How is this fixed point related to IFP_{exp} ?

While S is a 'real' fixed point of exp, IFP_{exp} computes its inflationary fixed point. Using monotonicity argument, one can easily verify that the two fixed points coincide for expressions that define monotone mappings on sets.

Definition 3.3 *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' $\mathrm{MEM}(a', exp(S_1)) = T$ implies $\mathrm{MEM}(a', exp(S_2)) = T$.*

Proposition 3.4 *If exp is monotone, then in the valid interpretation, for every element a hold:*
$\mathrm{MEM}(a,S) = T$ *iff* $\mathrm{MEM}(a,\mathrm{IFP}_{exp}) = T$,
$\mathrm{MEM}(a,S) = F$ *iff* $\mathrm{MEM}(a,\mathrm{IFP}_{exp}) = F$.

However, if *exp* is not monotone, then IFP_{exp} and S may **not** have the same behavior. For example, if $exp = \{a\} - x$, then
$\mathrm{IFP}_{\{a\}-x} =$

$(\{a\}-\mathrm{EMPTY}) \cup (\{a\}-(\{a\}-\mathrm{EMPTY})) \cup \ldots = \{a\}$
while for the set S defined by the equation $S = \{a\} - S$, the membership status of a in S in the valid model in undefined. The difference between the results reflects 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 $S = exp(S)$ in the initial valid 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 that, using a more complex translation technique, IFP_{exp} can be represented in $algebra^{=}$ for every *exp*. We first translate IFP_{exp} into a deductive program (proved to be possible in proposition 5.3). Then we translate the deductive program into an $algebra^{=}$ program (proved to be possible in proposition 6.1). It follows that

Theorem 3.5 $\mathrm{IFP}\text{-}algebra \subset algebra^{=}$

Corollary 3.6 $\mathrm{IFP}\text{-}algebra^{=} = algebra^{=}$

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

4 Deduction

We next consider the relationship between IFP–*algebra*, $algebra^{=}$ and deduction. We start by presenting deductive database and query specifications.

In a deductive database, the database relations are represented by predicates. To define a database, one specifies the data types used in the database, and then describes the database content. The data types used in the database are defined using a specification *SPEC*. Here again we assume that all the data types are well defined. The database content is defined using Horn clauses of the form $Q_1, \ldots, Q_n \rightarrow R_i(x)$ where Q_j is an atomic formula ($R_i(x_j)$, $exp_1 = exp_2$) or a negated atomic formula. One can use in the definition all the types and operations from *SPEC*. Since our formalism allows us to define domains of arbitrary ADT's, the nested relations/complex object models, and models that allow attribute values to be arbitrary ADT's are special cases.

A **deductive query** consists of a set of rules, and a query of the form "$R(x)$? ". Its answer is obtained by importing data base specification and computing the valid model of program, starting from the initial valid model of the data types. The semantics, the model, has the initial valid algebra of the types specifications as the domain, and additionally has relations that satisfy the formulas of the database and query specifications. Even if no negation is used, using the default mechanism of the minimal model guarantees that the result, including predicates and membership, is well-defined (although membership may not be recursively computable). If the program is stratified, then the answer can obtained by successively computing the minimal model of each stratum.

Note that the result of a deductive query depends on the domain of the data types. For example, the answer to a query of the form $Q(x)$?, where Q is defined by the rule $\neg R(x) \rightarrow Q(x)$, changes if the domain of x is changed. The (in)sensitivity of queries to the domain in which they are computed is often called *domain (in)dependence* (abbrv. d.i.) [1, 5, 18, 22]. Intuitively, domain independent queries use in the computation only a part, a "window", of the initial model, and are insensitive to the properties of elements outside this window. Domain independence is a desirable property, since the initial model may be very large, often infinite. Thus, the ability to ignore parts of it increases the efficiency of the computation. For lack of space we shall not consider the subject further here. For a formal definition of domain independence in our framework, and full discussion of the subject, see [5, 20]. For our needs here, it suffices to observe that if a query is d.i. then in particular one can consider only its result in the initial valid model, since any other (reasonable) model for the domains provides the same answer.

Domain independence (d.i.) is a semantic concept. However, it is possible to syntactically restrict queries by a condition that guarantees domain independence. We present such restrictions below.

The accepted approach to making a formula safe is to restrict the range of the variables in it [22, 1]. We restrict the range of the variables in the formula such that all the elements used in the computation, either appear in the database, are components of database members, or obtained from them by function applications. We define below range formulas, and restricted variables, and later use them for defining safe deduction.

Definition 4.1 *A range formula restricting the variables $x_1, \ldots, x_n$ is a formula with precisely $x_1, \ldots, x_n$ free, having the structure defined below:*
basis:

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

b. $x_1 = exp$, where exp is a ground expression, is a range formula restricting x_1.

construction: *if ϕ_1, ϕ_2 are range formulas restricting $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$ resp., then:*

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

2. $\phi_1 \wedge (exp_1 = exp_2)$, *where all the variables in* exp_1, exp_2 *are restricted by* ϕ_1, *is a range formula restricting* $x_1, \ldots, x_n$

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

4. $\phi_1 \wedge y = exp$, *where all the variables of* exp *are restricted by* ϕ_1, *is a range formula restricting* $y, x_1, \ldots, x_n$

We say that a horn clause is **safe** if it is of the form $\phi \rightarrow R_i(\bar{x})$ where ϕ is a range formula restricting $\bar{x}$.[5] A deductive program P is **safe**, iff all its clauses are safe. Note that the restrictions do not deal with quantifiers (as, e.g, in [5]), since the only quantifiers in logic programs are those that are implicit in the quantification of rules.

Since algebraic queries are domain independent [5], we compare in the following the expressive power of the algebra to the domain independent part of the deductive language. First we note that the safety syntactic restrictions are general enough to capture all d.i. deductive queries.

Proposition 4.2 *Every d.i. deductive query has an equivalent safe query, under the valid semantics. Moreover, if the first query is stratified, then so is the equivalent query.*

Proof: (Sketch) The proof follows from the observation that restricting the variables of a d.i. query to range only over elements from the initial model, does not change the query result. To convert a d.i. deductive query into a safe one, we restrict all the variables in the body of the rules. We first define for every type s_i a unary predicate S_i that contains all the elements in the initial valid model elements of the domain of s_i. Since the elements are constructed from constants, by applying functions, we can write safe rules defining S_i. Next we replace every rule $\phi \rightarrow R(x_i)$ of Q with variables $x_1, \ldots, x_n$ by $S_1(x_1) \wedge \ldots \wedge S_n(x_n) \wedge \phi \rightarrow R(x_i)$. The new program is safe, and since the original one is d.i., the two programs are equal. □

Thus in the following we only consider safe deductive programs.

In [5] we considered a restricted part of **IFP-algebra**, where fixed point operator is applied only to expressions where the variable does not appear negatively, e.i. does not appear in a sub-expression being subtracted. (Such expressions are certainly monotone.) We called this algebra the **positive IFP-algebra** and showed that

Theorem 4.3 *[5] The stratified d.i. deductive language, the stratified safe deductive language, and the positive* IFP$-$*algebra are equivalent.*

In the following sections we extend this result, by considering the relationship between general (i.e., not necessarily positive) IFP$-$*algebra*, *algebra*$^=$, and non stratified deduction.

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

5 From Algebra to Deduction

We start by considering general IFP$-$*algebra* queries.

The naive (and quite well-know) algorithm used in the proof of theorem 4.3, for transforming positive IFP$-$*algebra* queries into deductive queries, works as follows: For every sub expression in the query a new predicate name is introduced, and a derived relation is defined. Then, every sub query is constructed from it sub-query components. For example, $E_1 \cup E_2$ is represented by two rules of the form $R_1(x) \rightarrow R(x)$, $R_2(x) \rightarrow R(x)$, where R_1, R_2 are derived predicates corresponding to E_1, E_2 resp., $E_1 - E_2$ is represented by a rule $R_1(x), \neg R_2(x) \rightarrow R(x)$, and a fixed point expression IFP_{exp} is translated by first translating exp and then introducing recursion in the deduction. The rest of the operations are translated similarly.

Note that subtraction is translated to a negation of the corresponding predicate. When this translation technique is applied to positive IFP queries, the resulting program is stratified [5]. This is not the case, however, when the translation technique is applied to non positive queries.

Example 4:
When the above translation technique is applied to the (non positive) fixed point expression $Q = \text{IFP}_{\{a\}-x}$, the resulting deductive program

$R(a)$

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

is clearly 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. valid model semantics, the algebra query and the deductive queries may have different results. This is because the valid model semantics interprets negation as "can not be derived at all" (vs. "was not derived so far" in inflationary semantics).

Example 4: (cont'd)
Since $Q = \text{IFP}_{\{a\}-x} = (\{a\} - \text{EMPTY}) \cup \ldots = \{a\}$, 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 valid model semantics, $Q(a)$ is neither true nor false. This is because in the valid model approach, a fact can be assumed to be false only if there is no possible derivation for it. Thus, $Q(a)$ can not be assumed to be false. But $Q(a)$ can not be derived unless $\neg Q(a)$ is assumed. Thus neither $Q(a)$ nor $\neg Q(a)$ hold in the valid model. It follows that the result is different then that of the original algebra query. □

Thus, we have:

Proposition 5.1 *Every IFP-algebra program has a deductive program that is equivalent to it, when the IFP-algebra program is evaluated using valid model semantics, and the deductive program is evaluated under the inflationary semantics.*

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

Proposition 5.2 *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 when P is evaluated using fixed point semantic, iff $R(a)$ holds when P' is evaluated using valid model semantics.*

Proof: We present the proof for the case where the initial model contains the natural numbers. The cases where the initial model contains some other type with infinite domain, or where all types have finite domain, are handled similarly. 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 $\ldots(\neg)Q(x)\ldots \rightarrow R(y)$, is replaced by $\ldots(\neg)Q'(i,x)\ldots \rightarrow R'(i+1,y)$.
(iv) Finally, for every R' we add two new rules:
$R'(i,x) \rightarrow R'(i+1,x)$ and $R'(i,x) \rightarrow R(x)$.
The program P' simulates the inflationary computation of P. At each step of the derivation, new facts can only be derived using facts with smaller indexes. Thus the result obtained using valid semantics is the same as the one obtained by the inflationary computation of Q'. □

It follows that a (non positive) 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 valid model interpretation. It follows that

Proposition 5.3 *Every* IFP $-$ *algebra query has an equivalent domain independent deductive query.*

Theorem 3.5 (whose proof actually uses results of the next section) states that the IFP $-$ *algrbra* is properly included in the *algebra*$^=$. Thus, an interesting question is whether general *algebra*$^=$ queries can also be expressed by d.i. deduction. It turns out that the answer is positive.

Proposition 5.4 *Every algebra*$^=$ *query has an equivalent d.i. deductive query.*

Proof: (Sketch) 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 $f_i(e_1,\ldots,e_n)$ were f_i is defined by an equation $f_i(x_1,\ldots,x_n) = exp_i(x_1,\ldots,x_n)$ is represented in the deductive program by a corresponding predicate R_i. To define this predicate, we generate for the algebra expression $exp_i(e_1,\ldots,e_n)$ a deductive program with result predicate S_i, and then add the rule $S_i(x) \rightarrow R_i(x)$. Note that the *algebra*$^=$ query Q and its corresponding deductive program Q' both interpret subtraction and negation (resp.) using valid semantics. Thus have the same result. □

6 From Deduction to Algebra

We have shown in the previous section that *IFP – algebra*, and *algebra*$^=$ queries have equivalent deductive queries. But what about the other direction?

We show in the following that for every safe deductive query there exists an equivalent *algebra*$^=$ query. Each predicate R_i in the deductive program is represented by a corresponding set constant R_i^a. The translation process is based on defining for each such predicate a *simulation function* simulating the derivation of the predicate, and then defining the corresponding constant to be the fixed point of the function.

Let $R_1,\ldots,R_m$ 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 $P_1,\ldots,P_n$. Let $\phi_1 \rightarrow P_i(x),\ldots,\phi_m \rightarrow P_i(x)$ 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 $Q^c = \{x|\exists y_1,\ldots y_k(\phi_1 \vee \ldots \vee \phi_m)\}$, (where $y_1 \ldots y_k$ are the free variables,other then x, in $\phi_1,\ldots,\phi_m$). Since every calculus query can be expressed by the *algebra* [5], Q^c has an equivalent *algebra* query $Q^a = exp_i(P_1^a,\ldots,P_n^a,R_1^a,\ldots,R_m^a)$. We call exp_i the **simulation function** of P_i.

Clearly, $exp_1,\ldots,exp_n$ simulate one step in the (simultaneous) derivation of $P_1,\ldots,P_n$ resp. 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.

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

Note that exp_i is defined by the *algebra*, thus the program defining the sets $P_1^a,\ldots,P_n^a$ 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. It follows that

Proposition 6.1 *Every safe deductive program has an equivalent algebra*$^=$ *program.*

From the above discussion, and from propositions (5.4), (6.1), (4.2), and corollary (3.6) we have that

Theorem 6.2 *The d.i. deductive language , the safe deductive language, the algebra*$^=$*, and the* IFP*–algebra*$^=$ *are equivalent.*

The close relationship between the algebras and deductive programming indicates that algebraic 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 MEM(a,P^a) equals T or F the *membership testing problem.*

Proposition 6.3 *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 valid models.*

Proof: The proof of the first claim follows immediately from theorems 4.3 and 6.2 and from the fact that the problem of checking whether $P(a)$ holds for some predicate P defined by a d.i. 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* programs have valid models, the un-decidability result holds even if we consider only the portion of the *algebra*$^=$ that has valid model. □

7 Conclusions

This paper deals with the question whether the class of queries expressed by deductive programs with negation can be expressed in algebraic tools. We present a language *algebra*$^=$ that has the same expressive power as general deduction under valid model semantics. Thus answer the question positively. The results of this work can be easily adjusted to capture other semantics for negation, e.g. the well-founded or the stable-model semantics, by modifying the definition of the initial valid model accordingly.

Acknowledgments: The authors wish to thank Vassos Hadzilacos and Sam Toueg for technical help.

References

[1] S. Abiteboul, C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. *Proc. Int. Workshop on Theory and Applications of Nested Relations and Complex Objects.* Darmstadt, 1987

[2] K. R. Arp, H. Blair, A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor *Foundations of Deductive Databases and Logic Programming,* Los Altos, CA, 1988. Morgan Kaufmann, 98-184

[3] C. Beeri. New Data models and Languages. *Proc. 11th Symp. on Principles of Database Systems - PODS,* San Diego, California 1992,1-15

[4] C. Beeri and T. Milo. Subtyping in OODB's. *Proc. 10th Symp. on Principles of Database Systems - PODS,* Denver, Colorado 1991,300-314

[5] C. Beeri and T. Milo. Functional and Predicative Programming in OODb's. *Proc. 11th Symp. on Principles of Database Systems - PODS,* San-Diego 1992,176-190

[6] C. Beeri, R. Rmamkrishnan, D. Srivastava, S Sudarshan. The Valid Model Semantics for Logic Programs. *Proc. 11th Symp. on Principles of Database Systems - PODS,* San-Diego 1992,91-104

[7] V. Breazu-Tannen, P. Buneman and Limsoon Wong. Naturally embedded query languages. *Proc. 4th Int'l Conf. on Database Theory (ICDT),* Berlin, Germany, 1992, pp. 140–154.

[8] A. Chandra, D. Harel. *Horn clause queries and generalizations* Journal of logic programming, 2(1):1-15,1985.

[9] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications I, Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science 6, Springer, Berlin, 1985.

[10] H. Ehrig, A. Sernadas, C. Sernadas. Objects,Object Types, and Object Identity. *Categorial Methods in Computer Science with Aspects from Topology,* H. Ehrig et al, Springer Verlag, 1989, 142-156

[11] M. Gelfhond, V. Lifschitz. The stable model semantics for logic programming. *Proc. 5th Int. Conf. and Symp. on Logic Programming,* 1988

[12] J. Goguen, J. Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. *Research Directions in Object-Oriented Programming,* B. Shriver, P. Wegner. MIT Press 1987, 417-479

[13] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags *Proc. 12th Symp. on Principles of Database Systems - PODS,* Washington, D.C., 1993

[14] S. Grumbach and V. Vianu. Tractable Query Languages for Complex Object Databases. *Proc. 10th Symp. on Principles of Database Systems - PODS,* Denver, Colorado 1991,315-327

[15] R.Hull, J.Su On the Expressive Power of Database Queries with Intermediate Types. *Journal of Computer and System Sciences,* 43(1), 1991

[16] N. Immerman, S. Pantaik, D. Stemple. The Expressiveness of Family of Finite Set Languages. *Proc. 10th Symp. on Principles of Database Systems - PODS,* Denver, Colorado 1991,37-52

[17] S. Kaplan. Positive/Negative Conditional Rewriting. *Proc. 1st Int'l Workshop on Conditional Term Rewriting Systems,* Orsay, Springer-Verlag LNCS 308, 1988,129-143

[18] M. Kifer. On Safety, Domain Independence, and Capturability. *3rd Int. Conf. On Data and Knowledge Bases: Improving usability and Responsiveness,* Jerusalem, Israel, 1988

[19] C.K. Mohan, M.K. Srivas Conditional Specifications with Inequational Assumptions. *Proc. 1st Int'l Workshop on Conditional Term Rewriting Systems,* Orsay, Springer-Verlag LNCS 308, 1988,161-178

[20] T. Milo Formalisms for Describing OODB's. *Ph.D. Thesis,* Hebrew University, Jerusalem, 1992

[21] J.W. Thacher, E.G. Wagner, J.B.Wright. Data Type Specification: Parameterization and the Power of Specification Techniques. *ACM Transactions on Programming Languages and Systems.,* Oct. 1982, Volume 4, Number 4.

[22] J. D. Ullman. *Database and Knowledge Base Systems.* Computer Science Press, 1988.

[23] A. Van-Gelder. Negation as failure using tight derivation for general logic programming. *Proc. 3rd IEEE symp. on Logic Programming,* Salt Lake City, Utah, Sept. 1986.

[24] A. Van-Gelder, K. A. Ross, J. S. Shlipf. Unfounded sets and well founded semantics for general logic programming. *Proc. 7th ACM Symp. on Principles of Database Systems,* 1988.

[25] Wieringa, R. Van de Riet, R. Algebraic Specification of Object Dynamics in Knowledge Base Domains. R. Meersman, Z. Shi, C. Kung(eds), *The Role of Artificial Intelligence in Databases and Information Systems* North Holland 1990 411-436