DATA FUNCTIONS, DATALOG and NEGATION

(Extended Abstract)

Serge Abiteboul[1]
Institute National de Recherche
en Informatique et en Automatique
Rocquencourt, 78153
France

Richard Hull[2]
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
USA

Abstract

Datalog is extended to incoporate single-valued "data functions", which correspond to attributes in semantic models, and which may be base (user-specified) or derived (computed). Both conventional and stratified datalog are considered. Under the extension, a datalog program may not be consistent, because a derived function symbol may evaluate to something which is not a function. Consistency is shown to be undecidable, and is decidable in a number of restricted cases. A syntactic restriction, *pairwise consistency*, is shown to guarentee consistency. The framework developed here can also be used to incorporate single-valued data functions into the Complex Object Language (COL), which supports deductive capabilities, complex database objects, and set-valued data functions.

There is a natural correspondence between the extended datalog introduced here, and the usual datalog with functional dependencies. For families Φ and Γ of dependencies and a family of datalog programs Λ, the $\Phi-\Gamma$ *implication problem* for Λ asks, given sets $F \subseteq \Phi$ and $G \subseteq \Gamma$ and a program P in Λ, whether for all inputs I, $I \models F$ implies $P(I) \models G$. The FD-FD implication problem is undecidable for datalog, and the TGD-EGD implication problem is decidable for stratified datalog. Also, the $\emptyset$-MVD problem is undecidable (and hence also the MVD-preservation problem).

I. INTRODUCTION

Deductive databases provide one approach to incorporating reasoning capabilities into database schemas, and iterative computations in database queries. Two prominent frameworks for deductive databases are *datalog* (Horn clauses with no function symbols), and *stratified datalog*$^{\neg}$ (which permits negation in the premises of clauses, but still has no function symbols). Functional relationships have been studied in both relational dependency theory and semantic data modeling. In the relational model they are studied using functional dependencies, and in semantic models using *data functions* or attributes. We study in this paper the interaction between (stratified) datalog$^{(\neg)}$ and functional relationships.

We use the term "data function" to encompass both *base* (or user-specified) functions, such as person-birthdate, and *derived* functions, such as person-age (which might be computed from person-birthdate and the current date). These are distinct from the "functors" which arise in logic programming [L,ABW], because functors are *freely* interpreted (i.e., ground terms t_1 and t_2 are equal in a model iff they are syntactially identical).

Semantic database languages such as DAPLEX [Sh] attest to the advantages of supporting the explicit use of data functions in the specification of schemas, queries and updates. In semantic models, both single-valued and set-valued data functions are supported. The Complex Object Language (COL) of [AG] provides a coherent framework which combines stratified datalog$^{\neg}$ with *set-valued* data functions. (COL also supports complex object types, e.g., both flat relations and nested relations.) The prime

[1] Work supported in part by PRC BD Avancee.

[2] Work supported in part by NSF grant IST-85-11541.

objective of the current paper is to incorporate *single-valued* data functions into datalog$^{(\neg)}$ and COL.

We develop a syntax which supports single-valued data functions within deductive databases. Derived single-valued data functions may lead to *inconsistency* (because in the result a given value is mapped to two or more values). This leads to a study of the decidability of consistency. We show that in general, consistency is undecidable, and present several cases where consistency is decidable. A syntactic restriction called *pairwise consistency* is introduced, and shown to imply consistency. Pairwise consistency is related to Sagiv's notion of "uniform equivalence" [Sa], and is decidable in (stratified) datalog$^{(\neg)}$.

A primary contribution of the paper is the incorporation of single-valued data functions in the language COL. For reasons of simplicity, single-valued data functions were not considered in the original COL proposal [AG]. We show here how to incorporate such data functions. It appears that the extended COL that we obtain subsumes the expressive power of conventional semantic database query languages such as DAPLEX [Sh], and also the database component of the language FQL [BFN], which is based on the functional programming paradigm. The extended COL thereby provides the groundwork for incorporating deductive capabilities into query languages for semantic database models, such as the Functional Data Model [KP,Sh] or IFO [AH1].

While the focus of the paper is on single-valued data functions, it makes contributions of a more general nature. The study of the consistency issues raised by the presence of derived single-valued data functions leads naturally to the study of the interaction of functional dependencies with (stratified) datalog$^{(\neg)}$. Indeed, we reduce the *consistency* problem for single-valued data functions to the *FD implication problem* for datalog programs (does satisfaction of some FDs by the input imply satisfaction of FDs by the output?). The FD implication problem is undecidable for datalog programs, and for non-recursive stratified datalog$^{\neg}$ programs. Using the techniques developed here we also show that the implication problem for datalog and multi-valued dependencies (MVDs) is undecidable. This implies that the MVD-preservation problem is undecidable, resolving an open problem of [Sa].

A strong analogy exists between the research presented here and other attempts to combine functional and logic programming in languages (see [DL]). Our approach is different from those because our execution model is based on a fixpoint operation rather than on resolution. Importantly, current work on the LDL language [TZ,Be+] indicates that fixpoint semantics can be implemented in an efficient fashion in the context of databases.

The paper is organized as follows. Preliminary concepts and notations, including the relational model, dependencies, datalog and stratified datalog$^{\neg}$, are reviewed in Section II. Single-valued data functions are introduced and motivated in Section III. A semantics for (stratified) datalog$^{(\neg)}$ programs with single-valued data functions is presented, and a reduction to conventional (stratified) datalog$^{(\neg)}$ (with FDs) is described. Section IV presents our results on decidability and undecidability. Section V introduces and studies the notion of pairwise consistency. In Section VI, the incorporation of single-valued data functions into COL is discussed.

Due to space limitations, this report does not include proofs of the results stated. These proofs, and further discussion and examples, may be found in the full report [AH2] which is in preparation. The material of Section VI is developed in [AH3].

II. PRELIMINARIES

We assume basic knowledge of relational databases such as in [U], and familiarity with basic fixpoint techniques as in logic programming or deductive databases (e.g., [ABW, L]). In this section, we establish the basic notation used for relational dependencies and for (stratified) datalog$^{(\neg)}$.

We assume the existence of three countably infinite and pairwise disjoint sets of symbols: the set **att** of *attributes*, the set **dom** of *constants*, and the set **var** of *variables*. A *relational schema* is a finite set of attributes. A *tuple* over a relational schema R is a mapping from R into **dom** $\cup$ **var**. A *constant tuple* over a relational schema R is a mapping from R into **dom**. We assume an implicit ordering between attributes of a relational schema, and denote a tuple over a relational schema R of arity n by $[x_1,...x_n]$. An *instance* over a schema R is a finite set of constant tuples over R. A *database schema* is a finite set of relational schemas. An *instance* I over a database schema **R** is a mapping from **R** such that for each R in **R**, **I**(R) is an instance over R.

Let **R** be a database schema. A *functional dependency* (FD) over **R** is an expression R:X→Y where X and Y are subsets of a relational schema R in **R**. We assume familiarity with FDs, *multivalued dependencies* (MVDs), *total tuple generating dependencies* (TGD) [BV], and *equality generating dependencies* (EGD) [BV].

In the course of this paper we will consider several variants of datalog and stratified datalog$^{\neg}$ (denoted here by datalog$^{\neg}$). We briefly indicate some of the notational conventions used for datalog$^{(\neg)}$ in this section. Other conventions shall become apparent in Section III.

We assume the existence of a countably infinite set of *predicate symbols* (whose arity is implied by the context). A *term* is a constant or a variable. A *positive (negative) predicate literal* is an expression of the form $R(t_1,...,t_n)$ ($\neg R(t_1,...,t_n)$, resp.) where R is an n-ary predicate symbol and the t_i are terms. An *equality literal* is an expression of the form $t_1=t_2$ or $t_1 \neq t_2$ for terms t_1 and t_2. An *atom* is a positive predicate literal.

In datalog, a *rule* is an expression of the form

$$A \leftarrow L_1,...,L_n$$

where the *body* $L_1,...,L_n$ is a (possible empty) sequence of (positive or negative) equality literals and/or positive predicate literals (viewed as a conjunction); and the *head* A is an *atom*. Rules are viewed to be universally quantified. In datalog$^{\neg}$, both positive and negative predicate literals are permitted in rule bodies. A *program* is a finite set of rules.

Because we permit = and ≠ to appear in datalog rule bodies, the expressive power of our datalog is different than some versions studied elsewhere (e.g., [Sa]).

For both datalog and datalog$^{\neg}$, the *(Herbrand) base* B_P of a program P is the set of all *ground atoms* formed from the predicate symbols and constants appearing in P. If an instance I is present in the context, then the Herbrand base B_P also includes ground atoms formed using the constants occurring in I. The notions of *interpretation, (minimal) model,* and *relation instance associated* with an interpretation are defined in the usual manner.

The semantics of datalog$^{(\neg)}$ programs is defined in terms of mappings T which operate on interpretations. The following concepts are used:

- an operator T is *monotonic* if $I \subseteq J$ implies that $T(I) \subseteq T(J)$;
- I is a *fixpoint* of T, if $T(I) = I$; and
- I is a *pre-fixpoint* of T, if $T(I) \subseteq I$.

In the following, part (a) is well-known, and the proof of (b) is straightforward.

Proposition II.1:
(a) The following have equivalent expressive power: non-recursive datalog, the positive calculus, and the positive algebra.
(b) The following have equivalent expressive power: non-recursive datalog$^{\neg}$, the calculus, and the algebra. □

III. SINGLE-VALUED DATA FUNCTIONS

In this section, we motivate and formally introduce single-valued data functions into datalog. Further motivation for introducing data functions into datalog is presented in [AG]. We present the syntax and semantics of languages datalog+sdf and datalog$^{\neg}$+sdf, which extend datalog$^{(\neg)}$ with single-valued data functions. We conclude this section by describing the close relationship between datalog$^{(\neg)}$+sdf and datalog$^{(\neg)}$.

III.1 Motivation

This subsection presents several examples illustrating the power and convenience of using single-valued data functions in datalog$^{(\neg)}$.

Example III.1.1: Consider a database with relations

COURSES(CNAME, INSTRUCTOR),
WORKS_FOR(INSTRUCTOR,
DEPARTMENT), and
ENROLLMENT(STUDENT, CNAME),

where COURSES satisfies

CNAME → INSTRUCTOR,

and WORKS_FOR satisfies

INSTRUCTOR → DEPARTMENT.

Consider the query "What instructors are teaching Mary?" In conventional datalog, this is expressed by:

ANS(y) ← ENROLLMENT('Mary',x),
COURSES(x,y).

If the information on teaching is stored in a base

function teaches, this query can be expressed as
ANS(teaches(x)) $\leftarrow$
ENROLLMENT('Mary',x).

Consider the query "What students are taught by an instructor from Math?" In conventional datalog this is expressed by:
ANS(x) $\leftarrow$ ENROLLMENT(x,y),
COURSES(y,z), WORKS_FOR(z,'Math').
If works_for is a base function holding the data of WORK_FOR, this can be expressed by
ANS(x) $\leftarrow$ ENROLLMENT(x,y),
'Math' = works_for(teaches(y)).
This shows how function composition can sometimes be used to replace relational joins. □

Example III.1.2: Consider the relation
S(MOTHER, CHILD, NUMBER)
which lists women and their children, along with a number telling which child is oldest, second-oldest, etc. Suppose that $<$ is a base predicate with its usual meaning. The following program uses negation to define the function youngest_sibling:
NOT_YOUNGEST(w,p) $\leftarrow$ S(w,p,z),
S(w,p',z'), $z < z'$
youngest_sibling(p) = q $\leftarrow$ S(w,p,z),
S(w,q,z'), $\neg$ NOT_YOUNGEST(w,q) □

Example III.1.3: Consider the relations S(MOTHER, CHILD, NUMBER) as above, and R(PERSON, SEX), T(PERSON, SPOUSE). and suppose that the ordering $<$ is given as a base predicate. Consider the query "who is the father of my youngest sibling's wife?". It is straightforward to build temporary predicates FATHER_OF and WIFE_OF in datalog; and using negation the predicate YOUNGEST_SIBLING is easily built. The query can now be expressed in datalog$^{\neg}$ as
ANS(x) $\leftarrow$ FATHER_OF(x,y), WIFE(y,z),
YOUNGEST_SIBLING(x,'myself').
If single-valued data functions are supported, this becomes much easier to express and conceptualize. In particular, data functions father_of, wife, and youngest_sibling can be derived, and finally the answer can be expressed using the rule:
ANS(father_of(wife(
youngest_sibling('myself')))) $\leftarrow$.
Alternatively, the answer can be expressed simply as
father_of(wife(youngest_sibling('myself'))). □

The examples above show that using data functions in datalog facilitates the formulation and understanding of queries from a conceptual standpoint. Explicit data functions may be useful in the computation of answers. This is because their presence provides semantic information to the datalog compiler. For example, if f is a derived function (which is known to be single-valued) and one value for f(a) has been determined, then other values for f(a) do not have be considered.

To conclude this section, we present two more involved examples of programs with data functions.

Example III.1.4: This example uses a base function g. The program is:

$$f(x) = x \leftarrow h(x) = x$$
$$f(x) = f(g(x)) \leftarrow \neg\, h(x) \neq x.$$

Intuitively, this defines f(x) to be $g^i(x)$ for the smallest $i \geq 0$ (if it exists) such that $h(g^i(x)) = g^i(x)$. The formal semantics we introduce below will yield this meaning. □

Example III.1.5: Single-valued data functions can also be used to define more conventional operators. In this example, we assume that succ is a base function giving successor.

$$+(x,0) = x \leftarrow$$
$$+(x,succ(y)) = succ(+(x,y)) \leftarrow$$
$$*(x,0) = 0 \leftarrow$$
$$*(x,succ(y)) = +(*(x,y),x) \leftarrow$$
$$divides(x,y) = z \leftarrow x = *(y,z),\ y \neq 0.$$

(Technically, we assume that the underlying domain of constants is finite, and so the succ function and the other functions defined here are not total.) □

III.2 Syntax and Semantics

In this subsection, we present the syntax and semantics of datalog$^{(\neg)}$ with single-valued data functions (denoted by datalog+ sdf and datalog$^{\neg}$+ sdf). A key difference between the conventional and new frameworks is that when derived single-valued data functions are present, the issue of *consistency* of interpretations and programs must be considered.

We first develop datalog+ sdf, and then extend it to include negation.

alphabet, term: We assume the existence of two countably infinite disjoint sets, one of *predicate symbols* and one of (*data*) *function symbols* (whose arity is implied by the context). A *term* is a constant, a variable, or an expression $f(t_1,...,t_n)$ where f is a function symbol, and t_i is a term for i in [1..n]. A *ground term* is a term in which no variables occur.

literal: As before, we have positive and negative predicate and equality literals. A literal is *ground* if no variable occurs in it.

atom: An *atom* is a literal of the form $R(t_1,...t_n)$ or $f(t_1,...,t_{n-1}) = t_n$, where R is a predicate and f is a single-valued data function. If $t_1,...,t_n$ are constants, the atom is said to be *closed.*

Atoms will be used as rule heads. In this context, the equality symbol '=' is not symmetric: a rule of the form $f(x) = g(y) \leftarrow R(x,y)$ will define new values for f in terms of values of g. Intuitively, the '=' symbol here has a meaning related to the assignment symbol ':='.

rule and **program:** A *rule* is an expression of the form $A \leftarrow L_1,...,L_n$ where the *body* $L_1,...,L_n$ is a sequence of equality literals and/or positive predicate literals (viewed as a conjunction); and the *head* A is an *atom.* Rules are viewed to be universally quantified. A *program* is a finite set of rules.

Herbrand base, consistency, interpretation: The (*Herbrand*) *base* B_P of a program P is the set of all *closed atoms* formed from the predicate symbols, function symbols and constants appearing in P. (If an instance I is present in the context, then the Herbrand base also includes atoms formed using the constants occurring in I.) A subset of B_P is *consistent* if it does not contain a pair of atoms $[f(a_1,...,a_{n-1})=a]$ and $[f(a_1,...,a_{n-1})=b]$ where $a \neq b$. An *interpretation* of a program is a **consistent** subset of B_P.

The issue of consistency does not arise in datalog, because every subset of the Herbrand base of a datalog program is consistent. Because of function symbols occurring in terms, we introduce the notion of *valuation*, which will be used in datalog+ sdf in a manner analogous to how 'ground substitutions' are used in datalog.

valuation: Let θ be a ground substitution of the variables, and I a interpretation. The corresponding *valuation* θ_I is a partial function from the set of terms to the set of closed terms defined by:

(i) θ_I is the identity for constants, and $\theta_I x = \theta x$ for each variable,

(ii) if $\theta_I t_1,...,\theta_I t_n$ are defined, and $[f(\theta_I t_1,...,\theta_I t_n) = b] \in I$, then $\theta_I f(t_1,...,t_n) = b$.

satisfaction, (minimal) model: The notion of satisfaction (denoted by $\models$) and its negation (denoted by $\not\models$) are defined by: Let θ be a ground substitution and I an interpretation. If $\theta_I(t_i)$ is defined for $i \in [1..n]$ then

- $I \models P(t_1,...,t_n)[\theta]$ if $P(\theta_I t_1,...,\theta_I t_n) \in I$; and
- $I \models \neg P(t_1,...,t_n)[\theta]$ if $P(\theta_I t_1,...,\theta_I t_n) \notin I$.

If $\theta_I(t_i)$ is defined for $i \in [1,2]$ then

- $I \models t_1 = t_2[\theta]$ if $\theta_I(t_1) = \theta_I(t_2)$.
- $I \models t_1 \neq t_2[\theta]$ if $\theta_I(t_1) \neq \theta_I(t_2)$.

We now have:

- Let γ be the rule $A \leftarrow L_1,...,L_m$. Then $I \models \gamma$ iff for each ground substitution θ, $I \models L_i[\theta]$ for each $i \in [1..m] \implies I \models A[\theta]$.
- For each program P, $I \models P$ iff for each rule γ in P, $I \models \gamma$.

A *model* M of P is an interpretation which satisfies P.

A model M of P is *minimal* iff for each model N of P, $N \subseteq M \implies N = M$.

Example III.2.1: Consider the instance $I = \{ f(a) = b, f(b) = c, R(b)\}$.

- If $\theta_1(x) = a$ then $I \models R(f(x))[\theta_1]$.
- If $\theta_2(x) = b$ then $I \models \neg R(f(x))[\theta_2]$.
- If $\theta_3(x) = c$ then
  $I \not\models R(f(x))[\theta_3]$ and $I \not\models \neg R(f(x))[\theta_3]$.
  This is because f(c) is not defined in I. □

relational+ sdf instance associated with interpretation: With each program P, we associate the *database schema* $\mathbf{R}_P$ of P, which consists of (a) each function symbol occuring in P, and (b) a distinct relation schema R (of appropriate arity) for each predicate symbol R of P. An interpretation I of P can alternatively be viewed as an instance of $\mathbf{R}_P$, where

- for each predicate R of P, $I(R) = \{[a_1,...,a_n] \mid R(a_1,...,a_n) \in I\}$, and
- for each function f of P, I(f) is the (partial) function defined by $I(f)(a_1,...,a_{n-1}) = a_n$ if $[f(a_1,...,a_{n-1}) = a_n] \in I$, and is undefined otherwise.

rule application: Let P be a program, and I an interpretation of P. Then a closed atom A is the *result of applying* the rule $A' \leftarrow L_1,...,L_m$ *on* I *with* a ground substitution θ if

- $I \models L_i[\theta]$ for each $i \in [1..m]$, and
- **either** $A' = P(t_1,...,t_n)$ and $A = P(\theta_I t_1,...,\theta_I t_n)$, **or** $A' = [f(t_1,...,t_{n-1}) = t_n]$, and $A = \theta_I[f(t_1,...,t_{n-1}) = t_n]$.

Example III.2.2: Consider the instance I = { f(a) = b, f(b) = c} and the rules

γ_1: S(x,y) ← f(x) = y

γ_2: S(x,y) ← f(x) ≠ y

Then S(a,b) is a result of applying the rule γ_1 on I, and S(a,c) is a result of applying γ_2 on I. Intuitively, γ_2 says that S(x,y) holds if (a) f(x) is defined, and (b) f(x) ≠ y. Thus, none of S(c,a), S(c,b), nor S(c,c) are results of applying γ_2, because f(c) is not defined in I. □

operator T_P: For a program P, the operator T_P is a *partial* mapping from interpretations of P to interpretations of P. Specifically, for an interpretation I,

$T_P(I)$ = { A |A is the result of applying a rule in P with some ground substitution θ}

if this set is consistent; otherwise $T_P(I)$ is undefined. (In datalog, T_P is a total mapping.)

It is shown in the full paper that T_P (where defined) is monotonic in datalog+ sdf. Also, an interpretation I is a model of P iff it is a prefixpoint of T_P. A *justified* model [BH] (this is called *supported* in [ABW]) of P is a model M of P such that for each atom A in M, A is the result of applying some rule in P on M. In analogy with datalog, it is shown in the full paper that I is a fixpoint of T_P iff I is a justified model of P.

semantics of a program: As in datalog, the semantics of a datalog+ sdf program P is defined using the *powers* of T_P: If P is a datalog+ sdf program and I an interpretation for P, then

$$T\uparrow 0(I) = I,$$
$$T\uparrow(n+1)(I) = T(T\uparrow n(I)) \cup T\uparrow n(I) \text{ (if defined)}$$
$$T\uparrow\omega(I) = \bigcup_{n=0}^{\infty} T\uparrow n(I) \text{ (if defined).}$$

It is shown in the full paper that:

Proposition III.3.1: Let P be a datalog+ sdf program and I an interpretation of P. The following statements are equivalent:

(i) $T\uparrow\omega(I)$ is defined.

(ii) $T\uparrow\omega(I)$ is the minimal model of P containing I.

(iii) P has a model containing I. □

We now present the extension of datalog+ sdf to datalog¬+ sdf. This closely parallels the analogous extension of datalog; only the salient differences are mentioned here. A datalog¬+ sdf *rule* can have both positive and negative predicate literals in its body. The notions of *satisfaction, model,* and *rule application* carry over from datalog+ sdf.

When negative predicate literals are present in rule bodies in a program P, the operator T_P is not monotonic. We thus restrict our attention to a certain class of datalog¬+ sdf programs. This requires a number of preliminary notions.

defined symbol: R is the *defined symbol* of the atom $R(t_1,...,t_n)$, and f is the *defined symbol* of the atom $f(t_1,...,t_{n-1}) = t_n$. The *defined symbol* of a rule γ is the defined symbol of the head of γ. (This corresponds to the fact that a rule "$f(t_1,...,t_{n-1}) = t_n \leftarrow ...$" is used to give information about f rather than =.)

partial and total determinant, stratified program: Let X be the defined symbol of a rule γ. A symbol which occurs in a rule not as the defined symbol is called a *determinant* of X in γ. We distinguish between partial and total determinants: An occurrence of a determinant is *partial* in γ iff in this occurrence, either (a) the determinant is the defined symbol of a positive predicate literal, or (b) the determinant is a function symbol. An occurrence of a determinant is *total* if it is not partial (in particular, if it is a predicate symbol which occurs in a negative literal.) In a program P, the symbol X is a *total determinant* of Y *in* P if it occurs as a total determinant of Y in some rule of P; and X is a *partial determinant* of Y *in* P if it occurs as a partial determinant of Y in some rule of P, and does not occur as a total determinant of Y in any rule of P.

Example III.2.3: Consider the rule γ

f(x,g(y)) = h(y,f(y,x)) ← R(f(x)), ¬S(k(y,z)), y ≠ l(z,x)

(where upper case letters are predicates and lower case letters are function symbols). Then R, f, g, h, and k are partial determinants of f in γ, and S is a total determinant of f in γ. □

Note that single-valued data functions are never total determinants (see Example III.3.2).

definition of stratified, stratification: A program P is *stratified* is there is is no sequence $R_1,...,R_n = R_1$ (n > 1) where R_i is a determinant of R_{i+1} for each i in [1..n-1], and R_i is a total determinant of R_{i+1} for at least one i in [1..n-1]. In what follows, all datalog$^\neg$(+sdf) programs considered are assumed to be stratified. Let P be a datalog$^\neg$+sdf program. Then there is a partition $P = P_1 \cup \ldots \cup P_m$ of P such that:

- For each predicate or function symbol X of P there is a unique $i \in [1..m]$, such that each rule which defines X is in P_i. In this case we say that X is *defined by* P_i.
- If X is a partial determinant of Y and Y is defined by P_i, then X is defined by P_j for some $j \leq i$.
- If X is a total determinant of Y and Y is defined by P_i, then X is defined by P_j for some $j < i$.

Such a partition is called a *stratification* of P, and each P_i is called a *stratum* of P.

semantics of program: The semantics of a datalog$^\neg$+sdf program $P = P_1 \cup \ldots \cup P_m$ is defined using the operators $T_{P_i}\uparrow\omega$ as in datalog$^\neg$. The program P defines a (partial) mapping from interpretations to interpretations as follows: For an interpretation I,

- let $K_0 = I$, and
- let $K_i = T_{P_i}\uparrow\omega(K_{i-1})$ for each $i \in [1..m]$ (if defined).
- If K_m is defined then P(I) is defined to be K_m; it is undefined otherwise.

In analogy with datalog$^\neg$, we have:

Proposition III.3.2: Let P be a datalog$^\neg$+sdf program and I an interpretation. Then P(I) is a model of P. Furthermore (a) P(I) is defined for one stratification of P iff it is defined for all stratifications of P, and (b) if P(I) is defined, it is independent of the stratification used. □

Proposition III.3.2 allows us to give a semantics to a stratified datalog$^\neg$+sdf program. We choose $P(\emptyset)$, i.e., the image of the empty interpretation by P, as the semantics of P.

III.3 Reduction to datalog$^{(\neg)}$

In this section, we exhibit a strong analogy between datalog$^{(\neg)}$+sdf programs and datalog$^{(\neg)}$ constrained by certain FDs.

To each datalog$^{(\neg)}$+sdf program P, we will associate: (a) a relational database schema $\sigma[P]$; (b) a mapping from interpretations of P to instances over $\sigma[P]$ also denoted σ; (c) a set of FDs, $\delta[P]$; and (d) a datalog program $\pi[P]$ over $\sigma[P]$.

In particular, for each function f of arity n-1, let $R_f = \{A_{f,1} \ldots A_{f,n}\}$ where $A_{f,1} \ldots A_{f,n}$ are new attributes. Then $\sigma[P]$ is the schema consisting of the relations of P, together with relations $\{R_f \mid f$ function symbol occurring in P$\}$. We also denote by σ the mapping from instances over the relational+sdf schema of P to instances over $\sigma[P]$ defined by: for each I,

$$\sigma(I) = \{R(b_1,\ldots,b_n) \in I\} \cup$$
$$\{R_f(b_1,\ldots,b_n) \mid [f(b_1,\ldots,b_{n-1}) = b_n] \in I\}.$$

Let $\delta[P]$ be the set $\{ R_f{:}A_{f,1} \ldots A_{f,n-1} \rightarrow A_{f,n} \mid f$ is a function symbol in P $\}$.

We next present a mapping π from datalog$^{(\neg)}$+sdf programs to datalog$^{(\neg)}$ programs. To do this, we need the following normal form:

Definition: A datalog$^{(\neg)}$+sdf program P is in *flat normal form* iff the only literals occurring in P are of the forms: $P(t_1,\ldots,t_n)$, $\neg P(t_1,\ldots,t_n)$, $f(t_1,\ldots,t_{n-1}) = t_n$, $t_1 = t_2$ or $t_1 \neq t_2$, where $t_1,\ldots,t_n$ are free of data function symbols.

Example III.3.1: Consider the rule:

$$f(x) = f(g(x)) \leftarrow R(h(y,x)),\ h(x,y) \neq g(y)$$

An equivalent rule in flat normal form is given by:

$$f(x) = w \leftarrow g(x)=z,\ f(z)=w,\ R(v),$$
$$h(y,x)=v,\ h(x,y)=u,\ g(y)=t,\ u \neq t.$$ □

There is an obvious algorithm for transforming datalog$^{(\neg)}$+sdf programs into flat normal form which does not alter the semantics of the program [AH2]. Now for each flat normal form program P, let $\pi[P]$ be the datalog program obtained by replacing, for each f, each term of the form $f(t_1,\ldots,t_{n-1}) = t_n$ by $R_f(t_1,\ldots,t_n)$.

Example III.3.2: The datalog rule corresponding to the second rule of Example III.3.1 is given by:

$$R_f(x,w) \leftarrow R_g(x,z);\ R_f(z,w),\ R(v),\ R_h(y,x,v),$$
$$R_h(x,y,u),\ R_g(y,t),\ u \neq t.$$

In the original formula we had the negative equality literal $h(x,y) \neq g(y)$, but this was ultimately "replaced" by the literals $R_h(x,y,u)$, $R_g(y,t)$, and $u \neq t$. Note that R_h and R_g are both partial determinants of R_f. On an intuitive level, this indicates why function symbols in datalog$^\neg$+sdf are never total determinants. □

The following natural relation between datalog$^{(\neg)}$ and datalog$^{(\neg)}$+sdf holds.

Theorem III.3.1: For each datalog$^{(\neg)}$+ sdf program P in flat normal form and each I,

(a) P(I) is defined iff $\pi[P](\sigma(I)) \models \delta[P]$, and

(b) if P(I) is defined, then $P(I) = \sigma^{-1}(\pi[P](\sigma(I)))$. □

We also have:

Corollary III.3.2: For a datalog+ sdf program P in flat normal form, the following statements are equivalent: (a) P has a model; (b) $\pi[P](\emptyset) = T_{\pi(P)}\uparrow\omega(\emptyset)$ satisfies $\delta(P)$; and (c) $\sigma^{-1}(T_{\pi(P)}\uparrow\omega(\emptyset))$ is a model of P. □

This provides a method for implementing a datalog$^\neg$+ sdf system based on a datalog$^\neg$ system and a module for checking FD satisfaction.

IV. DECIDABILITY AND UNDECIDABILITY

In this section, we study the implication problem for datalog$^\neg$ programs and the consistency problem for datalog$^\neg$+ sdf programs in parallel. More precisely, we develop results for the implication problem, and derive from them results for the consistency problem.

Let P be a datalog$^{(\neg)}$+ sdf program over the database schema **R**. A predicate or function is *base* if it does not occur in the head of any rule of P, and it is *derived* otherwise. $\mathbf{R}_b$ denotes the set of base predicates and functions. In this section we generally view a program P as (partial) mapping from instances I of $\mathbf{R}_b$ to instances P(I) of $\mathbf{R}_P$. We now have:

Definition: A datalog$^{(\neg)}$+ sdf program P is *consistent,* iff the mapping P is total, i.e., P(I) is defined for each interpretation I over $\mathbf{R}_b$.

We use the following terminology:

Φ-Γ Implication Problem for (relational) language Λ: Let Φ and Γ be classes of relational dependencies. Given $F \subseteq \Phi$, $G \subseteq \Gamma$, and P a program in the language Λ, is it true that for all I over $\mathbf{R}_b$,

$$I \models F \text{ implies } P(I) \models G?$$

Example IV.1: An instance of the FD-FD implication problem for datalog is as follows: Consider the schema R(AB), S(BC), T(AC) and the program P consisting of the single rule:

$T(x,y) \leftarrow R(x,y), S(y,z)$.

Let F = {R:A → B, S:B → C}, and G = {T:A → C}. In this simple case, the answer to the problem is yes, i.e., for all I over {R,S}, I $\models$ {R:A → B, S:B → C} implies P(I) $\models$ {T:A → C}.

We begin with the positive results. The first is demonstrated by looking at "large" Cartesian product instances, and using a technique resembling the "hyperplanes" of [AV]; the second is proved using a chase procedure [MMS].

Proposition IV.1:

(a) The TGD-EGD implication problem for datalog programs is decidable.

(b) It is decidable, given a P datalog+ sdf program with no base data functions, whether P is consistent. □

Theorem IV.2:

(a) The FD-FD implication problem is decidable over the language of datalog programs without recursion.

(b) The FD-FD implication problem is decidable over the positive relational algebra.

(c) It is decidable, given a datalog+ sdf program P without recursion, whether P is consistent. □

Thus, consistency is decidable (a) in the absence of both base functions and negation; and (b) in the absence of both recursion and negation. The following two theorems show that weakening any of these conditions makes consistency undecidable. They are proved using a reduction to the Post Correspondence Problem.

Theorem IV.3:

(a) The FD-FD implication problem for datalog programs (without = and $\neq$) is undecidable.

(b) It is not decidable, given a datalog+ sdf program P with one base function and one derived function, whether P is consistent.

Theorem IV.4:

(a) The $\emptyset$-FD implication problem is not decidable for the family of non-recursive datalog$^\neg$ programs

(b) The $\emptyset$-FD implication problem is not decidable for the relational algebra.

(c) It is not decidable, given a datalog$^\neg$+ sdf program P without recursion or base functions, whether P is consistent. □

Although tangential to the main focus of the paper, we also mention the following result which can be proved using the same techniques as above. The following result implies that the MVD-preservation problem is undecidable, thus answering an open problem raised in [Sa].

Proposition IV.6: The $\emptyset$-MVD implication problem for datalog programs (without $=$ and $\neq$) is undecidable. $\Box$

VI. PAIRWISE CONSISTENCY

The previous section showed that it is undecidable whether a datalog$^{(\neg)}$+ sdf program is consistent. In this section we present a syntactic restriction on programs, called *pairwise consistency,* which guarentees consistency. We show that it is decidable whether a datalog$^{(\neg)}$+ sdf program is pairwise consistent. To do this, we first show that the analogous property of datalog$^{(\neg)}$ programs, here called *FD-uniform implication,* is decidable.

To define the notion of pairwise consistency a second normal form for programs is needed:

Definition: A datalog$^{(\neg)}$+ sdf program P is in *uniform normal form* iff it is in flat normal form, and for each pair of rules in P of the form

$f(t_1,...t_n) = t \leftarrow body_1$, and

$f(s_1,...s_n) = s \leftarrow body_2$,

t_i and s_i are identical for $i \in [1..n]$, and the only variables in common between $body_1$ and $body_2$ are $\{t_1,...,t_n\}$.

It is easy to see that each datalog$^{(\neg)}$+ sdf program can be transformed into an equivalent uniform normal form program.

Definition: A uniform normal form program P is *pairwise consistent* iff the following condition holds for each derived function f. Let

$f(t_1,...t_n) = t \leftarrow body_1$

$f(t_1,...t_n) = s \leftarrow body_2$

be a pair of rules defining f. (We include here the case where these come from the same rule by changing the variables other than $t_1,...,t_n$.) Then for each interpretation I, and each ground substitution θ of the variables, if

$I \models \theta_I body_1 \wedge \theta_I body_2$,

then $\theta_I t = \theta_I s$. (Intuitively, this means that $(body_1 \wedge body_2 \rightarrow t = s)$ is a tautology.)

It is straightforward to verify that the program of Example III.1.4 is pairwise consistent, but that the program of Example III.1.5 is not. (However, in the latter example, if succ is 1-1 in all input instances considered, then + will produce a well-defined function.) It is straightforward to show that:

Theorem V.1: For datalog$^{(\neg)}$+ sdf programs, pairwise consistency implies consistency. $\Box$

The converse of this theorem is not true [AH2].

We now consider the issue of testing for pairwise consistency. To do that, we first reduce the pairwise consistency problem to the problem of *FD-uniform implication* in the relational model. Uniform implication is related to implication in a manner analogous to the relationship between uniform equivalence and equivalence as defined in [Sa]. Also, the link between pairwise consistency and uniform implication is intuitively like the link between consistency and implication.

Φ Uniform Implication Problem for language Λ: Let P be a program in the language Λ for which the operator T_P is defined; and let Φ be a class of relational dependencies. For F in Φ, is it true that for all I *over the schema of P,*

$I \models F$ implies $T_P(I) \models F$?

Note that if an instance <P,F> of the uniform implication problem is answered positively, the instance <P,F,F> of the implication problem is also answered positively. The converse of this is not true.

An algorithm is presented in the full paper for testing uniform implication of FDs. This algorithm is an extension of the chase of tableaux [MMS] or conditional tables [IL,Gr,AGr] by FDs. It is also shown how to reduce the problem of determining pairwise consistency to the FD uniform implication problem. To summarize,

Theorem V.2:

(a) FD Uniform Implication is decidable in polynomial time for datalog and datalog$^{\neg}$ programs.

(b) Given a uniform normal form datalog$^{(\neg)}$+ sdf program P, it is decidable in polynomial time whether P is pairwise consistent. $\Box$

As discussed in the full paper, the notion of pairwise consistency can be used to ensure the consistency of data functions in a context where some other consistent data functions are not pairwise consistent (e.g., youngest-sibling in Example III.1.2).

VI. COL + SINGLE-VALUED DATA FUNCTIONS

The paper [AG] introduced the language COL, a formalism for deductive databases which incorporates complex objects [BK,H], set-valued data functions, and negation. (Other research efforts focussed on incorporting complex objects into deductive databases include [Be+ ,TZ] and [K].)

For reasons of simplicity, single-valued data functions were not considered in [AG]. Single-valued data functions in COL raise consistency problems similar to those studied in the previous sections. Indeed, their introduction in COL was the prime motivation for the research that is reported in the present paper.

In this section, we use results from the previous sections to incorporate single-valued data functions into COL and obtain COL+ sdf. The development here will be brief, as it is a relatively straightforward combination of ideas presented above and in [AG]. A detailed presentation of COL can be found in [AG], a formal presentation of COL+ sdf can be found in [AH3].

Intuitively, COL is a datalog extension which permits the manipulation of complex objects using multivalued data functions. The objects in COL are finite and typed objects obtained using a tuple and (heterogeneous) set constructors. An example of object manipulated in COL is given by:

[math,{[jeremie,2yrs],[charlotte,3yrs],
[maude,2yrs]}]

We next present an example to illustrate the use of multivalued data functions in COL:

Example VI.1: Consider the predicate CLASS(name,student). In the following, S presents the data in CLASS, with students grouped by class.

$F(x) \ni y \leftarrow \text{CLASS}(x,y);$
$S(x,F(x)) \leftarrow \text{CLASS}(x,y);$ □

The following example shows how single-valued functions can be used in COL+ sdf:

Example VI.1 (continued): Suppose that we have a function father_of and a function count. The following programs compute for each class the set of fathers of students in the class, and the number of fathers for each class:

$G(x) \ni \text{father\_of}(y) \leftarrow \text{CLASS}(x,y);$
$T(x,G(x),\text{count}(G(x))) \leftarrow \text{CLASS}(x,y).$ □

Intuitively, the stratification imposed on COL is based on the simple principle that a set can be used in its totality only when completely computed; elements in a set can be used along the way of their derivation. For instance in the rule defining S, F is a total determinant. The semantics of a COL+ sdf program can be defined using a fixpoint technique under some stratification conditions that combine the stratification presented above for single-valued data functions with the stratification of [AG] for multivalued one.

A key issue is consistency checking. We generalize in [AH3] the notion of pairwise consistency to COL+ sdf programs. We show that for COL+ sdf programs, pairwise consistency implies consistency. At the present time it remains open whether pairwise consistency of COL+ sdf programs is decidable, although we conjecture that it is. We show in [AH3] a weaker result resolving the issue for COL+ sdf programs without the ownership predicate. The problem is then co-NP-hard, the increased complexity resulting for the presence of complex objects in the underlying models.

REFERENCES

[AG] Abiteboul, S., and S. Grumbach, COL: a Language for Complex Objects based on Recursive Rules, abstract in Proc. Workshop on Database Programming languages, Roskoff (1987)

[AGr] Abiteboul, S. and G. Grahne, Update Semantics for Incomplete Databases, Proc. 11th VLDB, Stockholm, 1985.

[AKGr] Abiteboul, S., P. Kanellakis and G. Grahne, On the Representation and Querying of Sets of Possible Worlds, Proc. ACM SIGMOD Conf. on Management of Data, (May, 1987), pp. 34-48.

[AH1] Abiteboul, S. and R. Hull, IFO: A Formal Semantic Database Model, ACM Trans. on Database Systems 12:4 (Dec. 1987), pp. 525-565.

[AH2] Abiteboul, S. and R. Hull, Data Functions, Datalog and Negation, Technical report in preparation.

[AH3] Abiteboul, S. and R. Hull, Extending COL to support single-valued data functions. Technical report in preparation.

[AV] Abiteboul, S. and V. Vianu, Equivalence and Optimization of Relational Transactions, to appear, Journal of the ACM (preliminary report in Proc. Intl. Conf. on VLDB, 1984).

[ABW] Apt, K., H. Blair, A. Walker, Toward a Theory of Declarative Knowledge, Proc. of Workshop on Foundations of Deductive Database and Logic Programming (1986)

[BK] Bancilhon, F. and S. Khoshafian, A Calculus for Complex Objects, Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems (1986).

[Be+] Beeri, C., et al., Sets and Negation in a Logic Database Language (LDL1), Proc. ACM SIGACT-SIGMOD Symposium on Principle of Database Systems (1987)

[BV] Beeri, C. and M.Y. Vardi, Formal Systems for Tuple and Equality Generating Dependencies. SIAM J. Computing 13:1 (Feb. 1984), pp. 76-98.

[BH] Bidoit, N., R. Hull, Minimalism, Justification and Non-Monotonicity in Deductive Databases, to appear, Journal of Computer and System Sciences (based on: Positivism vs. Minimalism in Deductive Databases, Proc. ACM SIGACT-SISMOD Symposium on Principles of Database Systems, 1986).

[BFN] Buneman, P., R.E. Frankel, and R. Nikhil, An Implementation Technique for Database Query Languages, ACM Trans. on Database Systems 7:2 (1982), pp. 164-186.

[CFP] Casanova, M.A., R. Fagin and C.H. Papadimitriou, Inclusion Dependencies and their Interaction with Functional Dependencies, Journal of Computer and Systems Science 28:1 (1984), pp. 29-59.

[DL] Logic Programming, Functions, Relations and Equations, edited by D. DeGroot and G. Lindstrom, Prentice-Hall, 1986.

[GJ] Garey, M.R. and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979.

[G] Van Gelder, A., Negation as Failure Using Tight Derivations for General Logic Programs, Proc. of Workshop on Foundations of Deductive Database and Logic Programming (1986).

[Gr] Grahne, G., Dependency Satisfaction in Databases with Incomplete Information, Proc. of teh 10th VLDB, Singapore, 1984.

[HM] Hammer, M. and D. McLeod, Database Description with SDM: A Semantic Database Model, ACM Strans. on Database Systems 6,3 (1981), pp. 351-386.

[H] Hull, R., A Survey of Theoretical Research on Typed Complex Database Objects, in *Databases*, ed. by J. Paredaens, Academic Press (London), 1987, pp. 193-256.

[HK] Hull, R. and R. King, Semantic Data Modeling: Survey, Applications, and Research Directions, U.S.C. Computer Science Technical Report (1986); to appear in ACM Computing Surveys (Sept. 1987).

[IL] Imielinski, T. and W. Lipski, Jr., Incomplete Information in Relational Databases, Journal of the ACM 31:4 (October 1984), pp. 761-791.

[KP] Kerschberg, L. and J.E.S. Pacheco, A Functional Data Base Model, Technical Report, Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, Brazil, February, 1976.

[K] Kuper, G.M., Logic Programming with Sets, Proc. ACM SIGACT/SIGMOD/SIGART Symposium on Principle of Database Systems (1987), pp. 11-20.

[L] Lloyd, J., Foundations of Logic Programming, Springer-Verlag, 1984.

[MMS] Maier, D., A.O. Mendelzon, and Y. Sagiv, Testing Implications of Data Dependencies, ACM Trans. on Database Systems 4:4 (Dec. 1979), pp. 455-469.

[N] Naqvi, S.A., A Logic for Negation in Database Systems, Proc. Workshop on Foundations of Deductive Databases and Logic Programming ed. J. Minker (1986)

[Sa] Sagiv, Y., Optimizing Datalog Programs, Proc. ACM SIGACT/SIGMOD/SIGART Symposium on Principle of Database Systems (1987), pp. 349-362.

[Sc] Schmueli, O., Decidability and Expressiveness Aspects of Logic Queries, Proc. ACM SIGACT/SIGMOD/SIGART Symposium on Principle of Database Systems (1987), pp. 237-249.

[Sh] Shipman, D., The Functional Data Model and the Data Language DAPLEX, ACM Trans. on Database Systems 6:1, (1981), pp. 140-173.

[TZ] Tsur, S. and C. Zaniolo, LDL: A Logic-Based Data-Language, Proc. 12th Intl. Conf. on Very Large Data Bases, Kyoto, Japan, 1986.

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