COL : a logic-based language for complex objects

Serge Abiteboul, Stéphane Grumbach

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP105
78153 Le Chesnay Cedex
France
Tél.(1)39635511

Rapports de Recherche

N° 714

COL: A LOGIC-BASED LANGUAGE FOR COMPLEX OBJECTS

Serge ABITEBOUL
Stéphane GRUMBACH

SEPTEMBRE 1987

COL: A LOGIC-BASED LANGUAGE FOR COMPLEX OBJECTS[1]

COL: un langage pour objets complexes
basé sur la logique

Serge Abiteboul Stéphane Grumbach

I.N.R.I.A.
78153 Le Chesnay, FRANCE

August 13, 1987

Abstract: A logic-based language for manipulating complex objects constructed using set and tuple constructors is introduced. Under some stratification restrictions, the semantics of programs is given by a canonical minimal and causal model that can be computed using a finite sequence of fixpoints. Applications of the language to procedural data, semantic database models, heterogeneous databases integration, and Datalog queries evaluation are presented.

Résumé: On introduit un langage basé sur la logique dans le but de manipuler des objets complexes obtenus à partir de constructeurs de nuplets et d'ensembles. Sous certaines restrictions de stratification, la sémantique des programmes est donnée par un modèle canonique, minimal et causal, qui peut être calculé en utilisant une séquence finie de points fixes. Des applications du langage pour les données procédurales, les modèles de données sémantiques, l'intégration de bases de données et l'évaluation de requètes Datalog sont présentées.

1. This research was supported in part by the Projet de Recherches Coordonnées BD3.

INTRODUCTION

Two approaches have been followed for defining manipulation languages for complex objects: (1) an algebraic approach [AB,ABi,FT,SS and many others], and (2) a calculus approach [J,AB,RKS]. Recently, there has been some interest in pursuing a so-called logic programming approach to define languages for complex objects [BK,AG,Be+,K]. This is the approach followed here.

The language COL *(Complex Object Language)* based on recursive rules is presented. This language is an extension of Datalog which permits the manipulation of complex objects obtained using tuple and (heterogeneous) set constructors. The originality of the approach is that besides the base and derived relations, base and derived "data functions" are considered. As we shall see, data functions are multivalued functions defined either extensionally (base data functions) or intensionally (derived data functions). The introduction of these functions permits the manipulation of complex objects. Other advantages of data functions are also discussed.

The semantics for COL programs is based on minimal models. Unfortunately, because of sets and data functions, some programs may have more than one minimal model. A stratification in the spirit of the stratification introduced by [ABW,G,N and others] is used. It is shown that stratified programs do have minimal models. Furthermore, a canonical *causal* [BH] and *minimal* model of a given program is computed using a sequence of fixpoints of operators.

Stratification is used in [ABW,G,N] to handle the presence of negation in the body of the rules. It turns out that negation can be simulated using data functions. Indeed, the stratification for negation thereby obtained corresponds precisely to the stratification imposed by data functions. In the present paper, we also allow negations in the body of the rules. As just mentioned, this does not add any power to the language.

Data functions are natural tools to manipulate complex objects. Data functions present other advantages as well:

(i) Since queries can be viewed as data functions, the inelegant dichotomy between data and queries of the relational model disappear. In particular, queries can be stored in the database as data functions. The model therefore permits the manipulation of procedural data [S].

(ii) In COL, data can be viewed both in a functional and in a relational manner. As a consequence, the language can be used in a heterogeneous databases context (e.g., relational view on a functional data base; integration of a relational database with a functional one).

(iii) COL can also be used as a kernel language for semantic database models like SDM [HM], IFO [AH] or Daplex [Sh].

(iv) Some evaluation techniques for datalog queries like Magic Sets or others [B+,GM] make extensive use of particular functions. These functions can be formalized using our model.

As mentioned above, two other approaches have been independently followed to obtain a rule-based language for complex objects [Be+,K]. In [Be+], they do not insist on a strict typing of objects. In [K], only one level of nesting is tolerated. However, both approaches could easily be adapted to the data structures considered in this paper. Furthermore, in [AB], it is argued that all these approaches yield essentially the same power (i.e., the power of the safe calculus of [AB]). The points (i-iv) above clearly indicate advantages of our approach.

The paper is organized as follows. In the first section, types and typed objects are described, and examples of COL rules given. The second section is devoted to the formal definition of the language. The stratification is introduced in Section 3. In the fourth section, it is shown that each stratified program has a canonical, causal and minimal model which can be computed using a sequence of fixpoints. Advantages of the language are briefly considered in a last section. The proof of key results of Section 4 can be found in an appendix.

I. PRELIMINARIES

In this section, types and typed objects are described, and examples of COL rules given.

The existence of some *atomic types* is assumed. A set of *values* is associated with each type A. This set is called the *domain* of A, and denoted dom(A). More complex types are obtained in the following way.

Definition: if $T_1,...,T_n$ are types ($n \geq 1$), then

(i) $T=[T_1,...,T_n]$ is a (tuple) *type*, and
$$dom(T) = \{ [a_1,...,a_n] \mid \forall\ i,\ 1 \leq i \leq n,\ a_i \in dom(T_i)\}.$$
(ii) $T=\{T_1,...,T_n\}$ is a (set) *type*, and
$$dom(T) = \{ \{a_1,...,a_m\} \mid m \geq 0,\ \forall\ i,\ 1 \leq i \leq m,\ \exists\ j,\ 1 \leq j \leq n,\ a_i \in dom(T_j)\}.$$

An *object of type* T is an element of dom(T).

Note that objects of a given type can be seen as particular trees of bounded depth. Most of the results of the paper would still hold if the strict typing policy is replaced by a weaker condition which guarantees the boundedness of object trees.

Note also that the language allows the manipulation of heterogeneous sets. For instance, if CAR and PLANE are two types, then, for instance, {747, Concorde, Le Car, Mustang} is an object of type {CAR, PLANE}. However, our types are more restricted than types in [AH,HY,KV] which use a "union of type" constructor. For instance, pairs with atomic first coordinate of type CAR, and second coordinate of type PLANE or CAR, are not considered. On the other hand, the type corresponding to sets of such pairs (i.e., {[CAR,PLANE], [CAR,CAR]}) can be used. Since we are mainly interested in sets of objects, this limitation is not too severe. Furthermore, the language COL could be extended in a simple way to handle such types.

For each type T, the existence of an infinite set $\{x_T, y_T...\}$ of *variables* of that type is assumed. When the type of a variable x_T is understood, or when this type is not relevant to the discussion, the variable is simply denoted x.

Various aspects of the language are now illustrated through three examples.

Example 1 (sets):
Sets form an important component of the data structure. The predicates $\in$ and $=$ belong to the language. It is possible to define other predicates like $\subseteq$, disjoint, disjoint-union, union, ..., or functions like $\cup$, $\cap$, ... using rules.

In this example, x is a variable of type integer, and X and Y are variables of type set of integers. The following rules define the functions $\cap$, $\cup$, and Difference:

$$x \in \cap(X,Y) \leftarrow x \in X,\ x \in Y,$$
$$x \in \cup(X,Y) \leftarrow x \in X,$$
$$x \in \cup(X,Y) \leftarrow x \in Y,$$
$$x \in \text{Difference}(X,Y) \leftarrow x \in X,\ \neg(x \in Y).$$

Intuitively, the functions $\cap$, $\cup$ and Difference define sets by stating explicitly what are the elements of each set. Thus the term $\cap(X,Y)$ for instance is interpreted as the set of all the elements x such that $x \in X$, and $x \in Y$. Using these functions, the predicates $\subseteq$, $\subset$, Disjoint, Union, and Disjoint-union are now defined:

$$\subseteq(X,Y) \leftarrow \cup(X,Y) = Y,$$
$$\subset(X,Y) \leftarrow \subseteq(X,Y),\ x \in \text{Difference}(Y, X),$$
$$\text{Disjoint}(X,Y) \leftarrow \cap(X,Y) = \phi,$$
$$\text{Union}(X,Y,\cup(X,Y)) \leftarrow,$$
$$\text{Disjoint-union}(X,Y,\cup(X,Y)) \leftarrow \text{Disjoint}(X,Y).$$

The language allows the manipulation of complex objects, and also of "nested relations" [ABi,FT,JS,...] which are special cases of complex objects.

Example 2 (nested relations):

Let N denote the set of integers. Consider the predicate R(N,N,N) and the three predicates S(N,{[N,N]}), S'(N,{[N,N]}), S''(N,{[N,N]}). (The first field of S, S' and S'' contains an integer, and the second a binary relation.) Let Z be a variable of type {[N,N]}; and F and TC be functions of the appropriate types.

Unnest:

$$R(x,y,y') \leftarrow S(x,Z),\ [y,y'] \in Z.$$

Nest:

$$[y,y'] \in F(x) \leftarrow R(x,y,y'),$$
$$S'(x,F(x)) \leftarrow R(x,y,y').$$

Transitive closure of the second field of S':

$$[x,z] \in TC(Z) \leftarrow [x,z] \in Z,$$
$$[x,z] \in TC(Z) \leftarrow [x,y] \in Z,\ [y,z] \in TC(Z),$$
$$S''(x,TC(Z)) \leftarrow S'(x,Z).$$

Example 3 (heterogeneous sets):

Let STRING be a type. Consider the following typed symbols:

- P({{N,STRING}}) (i.e., P is a unary predicate, and its unique field contains a set of sets of integers and strings).
- F is a function of type $\{N,STRING\} \rightarrow \{N\}$;

- Q({{N}}); and
- H a function of type {{N,STRING}} $\rightarrow$ {{N}}.
- x_N is a variable of type N;
- Y of type {N,STRING};

The following program filters the integers from P:

$$x_N \in F(Y) \leftarrow P(X),\ Y \in X,\ x_N \in Y,$$
$$F(Y) \in H(X) \leftarrow P(X),\ Y \in X,$$
$$Q(H(X)) \leftarrow P(X).$$

II. THE COL LANGUAGE

In this section, the complex object language is defined.

The *language L* of the underlying logic is first defined. This language is based on a typed alphabet containing:

(a) typed constants and variables;

(b) logical connectors and quantifiers $/\backslash$, $\backslash/$, $\neg$, $\Longrightarrow$, $\exists$, $\forall$;

(c) typed equality ($=_T$), and membership ($\in_{T,S}$) symbols;

(d) typed predicate symbols;

(e) typed function symbols of three kinds:
- data functions,
- tuple functions $[]_{T1,...,Tn}$,
- set functions $\{\}_{T1,...,Tn}$.

Terms of the form $[]_{T1,...,Tn}(a_1,...,a_n)$, will be denoted by $[a_1,...,a_n]$; and terms of the form $\{\}_{T1,...,Tn}(a_1,...,a_m)$ by $\{a_1,...,a_m\}$. The set function has an arbitrary (but finite) number of arguments. Clearly, that function could be replaced by a binary function set-cons and a particular symbol, say φ, for the empty set. For instance, $\{\}_{T1,...,Tn}(a_1,...,a_3)$ would stand for set-cons(a_1,set-cons(a_2,set-cons(a_3,φ))).

In the remainder of the paper, the word "function" will only refer to data functions, and not to tuple or set functions. It is assumed that all the functions that are considered in the following are set-valued, i.e., an image by a data function is always a set. In the last section, this limitation is discussed, and an extension of the language to remove it considered.

Note that $\in_{T,S}$ is a symbol of the language. Clearly, $\in_{T,S}$ is interpreted by the classical membership of set theory. Indeed, when the types are understood, $\in_{T,S}$ is simply denoted by $\in$. A constant of a certain type T is interpreted as an element of dom(T).

The terms of the language are now defined:

Definition: A constant or a variable is a *term.* If $t_1,...,t_n$ are terms and F is an n-ary data, tuple or set function symbol, $F(t_1,...,t_n)$ is a *term.* (The obvious restrictions on types are of course imposed.)

A *closed term* is a term with neither variables, nor data functions.

Example II.1: The term [1,{2,3},{7}] is a closed term. On the other, [1,{2,3},F(2)] is not closed. These two terms are different, but they may have the same interpretation (if F(2) = {7}).

Literals are defined by:

Definition: Let R be an n-ary predicate, and $t_1,...,t_n$ terms. Then (with the obvious typing restrictions) $R(t_1,...,t_n)$, $t_1 = t_2$, and $t_1 \in t_2$ are *positive literals.*

If ψ is a positive literal, $\neg\psi$ is a *negative literal.*

Arbitrary well-formed formulas are defined from literals in the usual way. We have defined here the language of a first order logic. One can define a model theory and a proof theory for this language. This is not in the scope of the present paper. We next introduce a clausal logic based on this first order logic. A key component of that clausal logic is the notion of "atom". An *atom* is a literal of the form $R(t_1,...t_n)$ or $t_1 \in F(t_2,...,t_n)$. If $t_1,...,t_n$ are closed terms, the atom is said to be *closed.*

Now we have:

Definition: A *rule* is an expression of the form $A \leftarrow L_1,...,L_n$ where

- the *body* $L_1,...,L_n$ is a conjunction of literals; and
- the *head* A is an atom.

A *program* is a finite set of rules.

Example II.2: consider the following program P_0:

$$R(1,2,3) \leftarrow$$
$$R(1,3,5) \leftarrow$$
$$[y,y'] \in F(x) \leftarrow R(x,y,y')$$
$$S(x,F(x)) \leftarrow R(x,y,y').$$

The predicate R is extensionaly defined, whereas the function F and the predicate S are intensionaly defined.

In Datalog, rules are used to specify the extension of derived predicates. Consider the third rule above. The predicate in the left hand side of the rule is $\in$ which is interpreted by the set membership. In fact, the rule is used to specify the *data function* F and not the extension of a predicate.

We are interested by Herbrand-like models of our programs. The *universe* U is formed of all the closed terms which can be built from the constants of the language. Let P be a program The *base* B_P of P is the set of all closed atoms formed from the predicate and function symbols appearing in P, and the closed terms of U. An *interpretation* of a program P is a finite subset of the base B_P.

Continuing with the example above, we have:

Example II.2 (continued): An interpretation of the program P_0 is:

$$I = \{ R(1,2,3), R(1,3,5), [2,3] \in F(1), [3,5] \in F(1), S(1,\{[2,3],[3,5]\}) \}.$$

It should be noted that elements in the base (and thus, in the interpretation) have very simple form. In particular, literals like $F(2) = \{3,4,5\}$ or $F(2) \in G(2)$ are not in the base.

In order to define the notion of satisfaction of a rule, and thus of a program, the concept of valuation is introduced. Valuations play here the role of substitution in classical logic programming.

Definition: Let θ be a ground substitution of the variables, and I an interpretation. The corresponding *valuation* θ_I is a function from the set of terms to the set of closed terms defined by[2]:

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

(ii) $\theta_I[t_1,...,t_n] = [\theta_I t_1,...,\theta_I t_n]$, $\theta_I\{t_1,...,t_n\} = \{\theta_I t_1,...,\theta_I t_n\}$, and

(iii) $\theta_I F(t_1,...,t_n) = \{ a \mid [a \in F(\theta_I t_1,...,\theta_I t_n)] \in I \}$.

The function θ_I is extended to literals by:

(iv) $\theta_I P(t_1,...,t_n) = P(\theta_I t_1,...,\theta_I t_n)$,

(v) $\theta_I(t_1=t_2) = (\theta_I t_1 = \theta_I t_2)$, $\theta_I(t_1 \in t_2) = (\theta_I t_1 \in \theta_I t_2)$, and

(vi) $\theta_I(\neg A) = \neg\, \theta_I A$.

A valuation in this context depends on the interpretation that is considered. This comes from the need to assign values to terms built using function symbols. As we shall see, this is a major reason for the non monotonicity of the operators that will be associated to COL programs.

Using valuations, we now define the notion of *satisfaction* of rules and programs:

Definition: The notion of satisfaction (denoted by $\models$) and its negation (denoted by $\not\models$) are defined by:

- For each closed positive literal, $I \models P(b_1,...,b_n)$ iff $P(b_1,...,b_n) \in I$; $I \models b_1 = b_2$ iff $b_1 = b_2$ is a tautology; and $I \models b_1 \in b_2$ iff $b_1 \in b_2$ is a tautology.
- For each closed negative literal $\neg B$, $I \models \neg B$ iff $I \not\models B$.
- Let $r = A \leftarrow L_1,...,L_m$. Then $I \models r$ iff for each valuation θ_I such that for each i, $I \models \theta_I L_i$, then $I \models \theta_I A$.
- For each program P, $I \models P$ iff for each rule r in P, $I \models r$.

[2] The reader has to be aware of a subtlety in (iii). The symbol $\in$ in $[a \in F(\theta_I t_1,...,\theta_I t_n)]$ is a symbol of the language COL, whereas the other occurrence of $\in$ denotes the usual membership of set theory.

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, $N \subseteq M \Longrightarrow N = M$.

Example II.2 (end): The interpretation I_0 is a model of P_0. Furthermore, I_0 is minimal.

A given COL program may not have a minimal model. This of course arises because of the use of negation. However, even positive COL programs may not have a minimal model as illustrated by the following example:

Example II.3: Consider the program:

$$1 \in F,\ p(F),\ q(2),$$
$$q(1) \leftarrow p(\{1\}).$$

Then $\{1 \in F,\ p(\{1\}),\ q(1),\ q(2)\}$ and $\{1 \in F,\ 2 \in F,\ p(\{1,2\}),\ q(2)\}$ are two incomparable minimal models.

III. STRATIFIED PROGRAMS

The notion of stratification has been used by several authors [ABW, G, N,...] to give a semantics to programs with negation in the body of rules. We present a similar notion for programs allowing complex objects and data functions.

Some basic notions are first defined.

In a literal $P(t_1,...,t_n)$, the symbol P is the *defined* symbol. Similarly, in a literal $t_1 \in F(t_2,...,t_n)$, F is the *defined* symbol. The *defined symbol of a rule* is the defined symbol of the head of the rule. (This clearly relates to the fact that a rule "$t_1 \in F(t_2,...,t_n) \leftarrow ...$" does not pertain to the definition of the predicate $\in$, but in that of the function F.)

A symbol which occurs in a rule not as the defined symbol of the head is called *determinant* of the rule.

Consider, for instance, the following two rules:

$$y \in F(x) \leftarrow R(F(x),y)$$
$$S(x,F(x)) \leftarrow P(x,y)$$

The symbol F is the defined symbol of the first rule; and S that of the second. The symbols R

and F are determinants of the two rules.

To define the notion of stratification, we use the auxiliary concepts of "total" and "partial" determinants of a rule. We say that an occurrence of a determinant predicate P is *partial* in a rule if that occurrence arises in a positive literal. Similarly, the occurrence of a determinant function F in a positive literal $t_1 \in F(t_2,...,t_n)$ is said to be *partial*. A determinant is *partial* (in a rule) if all its occurrences are partial; a determinant is *total* otherwise.

For instance, consider the rule:

$$x \in F(G(y)) \leftarrow y \in H(x),\ R(x,y),\ \neg\ S(y,z),\ y \in H'(H'(x))$$

In that rule, F is the defined symbol. The symbols R and H are partial determinants, and the symbols S and G total determinants. The symbol H' has one total and one partial occurrence, and thus is a total determinant.

The distinction between total and partial determininant is quite natural. To derive a new atom using the previous rule it suffices to know some partial information on R and H (i.e., R(x,y) and $y \in H(x)$). On the other hand, S has to be completely known to be able to assert S(y,z). Similarly, H'(x) must be completely known.

Intuitively, if Y is defined by the rule, and X is a total determinant, then X must be "completely defined" before Y. This is denoted by $X < Y$. If X is only a partial determinant, then X must be defined no later than Y. This is denoted by $X \leq Y$. For each program P, a marked graph G_P is constructed as follows:

- the nodes of G_P are the predicate and function symbols of P,
- there is an edge from X to Y if $X \leq Y$, and
- there is a marked edge from X to Y if $X < Y$.

We are now ready to define the condition for stratification:

Definition: A program P is *stratified* iff the associated graph G_P has no cycle with a marked edge.

Remark: We have defined stratification of programs using both negation and data functions. As we shall see, negation van be simulated using data functions. We could have presented stratification only for data functions, and derived that for negation.

The stratification of the program induces an order of evaluation of the predicate and function symbols as follows:

Proposition: Let P be a program, and Q the set of predicate and function symbols of P. Then P is stratified iff there is a partition

$$Q = Q_1 \cup \ldots \cup Q_m$$

of Q such that

$$X \leq Y, X \in Q_i \Longrightarrow \exists\, j\ (i \leq j, Y \in Q_j), \text{ and}$$
$$X < Y, X \in Q_i \Longrightarrow \exists\, j\ (i < j, Y \in Q_j).$$

□

The partition of symbols induces a partitioning of a program in strata. For each $Q = Q_1 \cup \ldots \cup Q_m$, let $P = P_1 \cup \ldots \cup P_m$ where for each i

$$P_i = \{\ r \in P \mid \text{the defined symbol of r is in } Q_i\}.$$

It is assumed in the following that such a partitioning is assigned to each stratified program. Indeed, one can show [ABW] that the choice of that partitioning is not relevant. A program with a single stratum is called *monostratum* program, and a program with several is called *multistrata* program.

To conclude this section, we illustrate the previous definitions with an example:

Example: Consider the following four rules:

$$r_1 = y \in F(x) \leftarrow R(x,y),$$
$$r_2 = S(x,F(x)) \leftarrow R(x,y),$$
$$r_3 = x \in F(y) \leftarrow S(x,Y),\ y \in Y,$$
$$r_4 = x \in F(y) \leftarrow S(y,F(x)).$$

The program $\{r_1,r_2\}$ is stratified. A corresponding partition is $\{R,F\} \cup \{S\}$. Similarly, $\{r_1,r_3\}$ is stratified. A corresponding partition is $\{R,F,S\}$. On the other hand, $\{r_1,r_2,r_3\}$ and $\{r_4\}$ are not stratified.

IV. FIXPOINT SEMANTICS OF STRATIFIED PROGRAMS

In this section, the semantics of stratified programs is defined using a canonical, minimal and causal models.

The following three well-known 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 $T(I) \subseteq I$.

With each program P, we associate an *operator* T_P defined as follows:

Definition: Let P be a program, and I an interpretation of P. Then a closed term A is the *result of applying the rule* $A' \leftarrow L_1,...,L_m$ *with* a valuation θ_I if

- $I \models \theta_I L_i$ 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 $A' = [t_1 \in F(t_2,...,t_n)]$, and $A = [\theta_I t_1 \in F(\theta_I t_2,...,\theta_I t_n)]$.

The operator T_P is defined by:

$$T_P(I) = \{ A \mid A \text{ is the result of applying a rule in P with some } \theta_I \}.$$

For a program P, T_P is not monotonic in general. For instance, consider the program P consisting of the single rule $Q(F) \leftarrow$. Then

$$T_P(\{1 \in F\}) = \{Q(\{1\})\} \not\subseteq \{Q(\{1,2\})\} = T_P(\{1 \in F, 2 \in F\}).$$

The following proposition links the notion of model of P to that of pre-fixpoint of T_P.

Proposition IV.1: Let P be a program, and M an interpretation of P. Then the next two statements are equivalent:

- M is a (minimal) model of P,
- M is a (minimal) pre-fixpoint of T_P.

Proof: It is clearly sufficient to prove that M is a model of P iff M is a pre-fixpoint of T_P.

M is a model of P,
iff for each rule r in P, $M \models r$,
iff for each rule r in P, if A is the result of applying r with θ_M, then A belongs to M,
iff $T_P(M) \subseteq M$. □

The next proposition relates the notion of "causal" model to that of fixpoint of T_P. We first define the concept of causality [BH].

Definition: A model M of P is said to be *causal* if for each $A \in M$, there exist a rule r in P, and a valuation θ_M such that A is the result of applying r with θ_M.

The next proposition is a straightforward consequence of Proposition IV.1.

Proposition IV.2: Let P be a program, and M an interpretation. Then the next two statements are equivalent:

- M is a (minimal) causal model of P,
- M is a (minimal) fixpoint of T_P.

Proof: It clearly suffices to show that M is a causal model of P iff M is a fixpoint of T_P.

M is a causal model of P,
iff $T_P(M) \subseteq M$ (model of P), and $M \subseteq T_P(M)$ (causality),
iff $M = T_P(M)$, i.e., M is a fixpoint of T_P. □

Monostratum programs are first considered. For these programs, a model can be obtained by repeated application of the corresponding operators. This motivates the use of the classical notion of *powers* of an operator T:

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

We will prove the following result:

Theorem IV.1: Let P be a monostratum program. Then for each I,

- $T_P\uparrow\omega(I)$ is a minimal pre-fixpoint of T_P containing I.
- $T_P\uparrow\omega(\emptyset)$ is a minimal fixpoint of T_P.

This result shows that $T_P\uparrow\omega(\emptyset)$ can be viewed as a canonical model of the monostratum program P since by Proposition VI.2, it is a minimal causal model of P.

To prove that result, we will use three properties of monostratum programs. But, first, we introduce some notation which allows us to consider particular subsets of a given interpretation.

Notation: Let I be an interpretation, and X a set of predicate and data function symbols. We denote by $I|_X$ the following subset of I:

$$I|_X = \{P(a_1,...,a_n) \in I \mid P \in X\} \cup \{[a_1 \in F(a_2,...,a_n)] \in I \mid F \in X\}.$$

To prove Theorem IV.1, we shall show that monostratum programs are "growing", "X-finitary" and "stable on X" for some X.

Definition: Let P be a program and X a set of symbols. Then:

(1) T_P is *growing* [ABW] if for each interpretation I, J and M such that $I \subseteq J \subseteq M \subseteq T_P{\uparrow}\omega(I)$, then $T_P(J) \subseteq T_P(M)$.

(2) T_P is *X-finitary* if for each sequence (I_n) of interpretations such that for each n $(0 \leq n)$, $I_n \subseteq I_{n+1}$, and $I_n|_X = I_0|_X$, then $T_P(\bigcup_{n=0}^{\infty} I_n) \subseteq \bigcup_{n=0}^{\infty} T_P(I_n)$,

(3) T_P is *stable on X* if for each I, $(T_P(I))|_X \subseteq I|_X$.

The proof of Theorem IV.1, can be found in the appendix. Indeed, it is shown there that for some X, monostratum program are X-finitary and stable on X (Lemma A.2), that they are growing (Lemma A.3); and for each operator T with these three properties, and for each interpretation I,

(a) $T(T{\uparrow}\omega(I)) \subseteq T{\uparrow}\omega(I)$, and

(b) $T{\uparrow}\omega(I) \subseteq T(T{\uparrow}\omega(I)) \cup I$ (Proposition A.1).

Theorem IV.1 is then a consequence of these results (see Appendix).

Theorem IV.1 does not hold for multistrata programs. Indeed, the operator corresponding to a multistrata program is, in general, not growing as shown by the following example.

Example IV.1: Consider the program:

$$1 \in F \leftarrow ,$$
$$2 \in F \leftarrow ,$$
$$P(F) \leftarrow .$$

Let $I = \{1 \in F\}$, $J = \{1 \in F, 2 \in F\}$. One can show that $T_P(I) = \{1 \in F, 2 \in F, P(\{1\})\}$, and $T_P(J) = \{1 \in F, 2 \in F, P(\{1,2\})\}$. Thus, $I \subseteq J \subseteq T_P(I)$, and $T_P(I) \not\subseteq T_P(J)$. Therefore T_P is not growing.

To prove Theorem IV.1, the X-finitarity is used. In [ABW], besides the growing property, finitarity is used. Finitarity corresponds here to φ-finitarity. It should be noted that monostratum programs are not, in general, finitary as shown by the example:

Example IV.2: Consider the one-rule program:

$$P(F) \leftarrow$$

where F is a 0-ary function which returns a set of integers. Let (I_n) be the sequence such that $I_n = \{[i \in F] \mid i < n\}$. Then $\bigcup_{n=0}^{\infty} T(I_n) = \{P(\varphi),P(\{0\}),...,P([0..n]),...\}$; and $T(\bigcup_{n=0}^{\infty} I_n) = \{P([0..\infty])\}$. Then $T(\bigcup_{n=0}^{\infty} I_n) \not\subseteq \bigcup_{n=0}^{\infty} T(I_n)$. In fact, the program is {F}-finitary.

Now consider multistrata programs. Intuitively, the stratification guarantees a locality property [ABW] which permits us to view them as a sequence of monostratum ones. Indeed, with each stratified program, $P = P_1 \cup ... \cup P_m$, a sequence $T_1,...,T_m$ of operators is associated. The following construction is used:

- $K_0 = \varphi$, and
- $K_i = T_i \uparrow \omega(K_{i-1})$ for each $i \in [1..m]$.

The sequence $T_1,...,T_m$ of operators has the locality property which allows to conclude:

Theorem IV.2: Let P be a stratified program. Then K_m, defined as above, is a minimal fixpoint of $\bigcup_{i=0}^{m} T_i$. Thus K_m is a minimal causal model of P.

The proof of Theorem IV.2 can also be found in the appendix.

This is the main result for COL programs. It is interesting to note that negation can be simulated using data functions. Let P be a predicate. The following program gives an equivalent form of $\neg$ P.

$$t \in F(t) \leftarrow P(t),$$
$$A(t, F(t)) \leftarrow ,$$

$$Q(t) \leftarrow A(t,\{\}).$$

It is easy to see that Q(t) is equivalent to $\neg$ P(t). Consider the stratification condition imposed by the previous program. From the first rule, $P \leq F$; from the second, $F < A$, and from the third, $A \leq Q$. As a consequence, $P < Q$ which leads to the classical notion of stratification for negation.

V. DISCUSSION

In this section, we briefly consider some applications and extensions of the language. More precisely, we illustrate the following points:

(i) procedural data;
(ii) heterogeneous databases (functional and relational);
(iii) semantic database models; and
(iv) evaluation techniques for datalog queries.

During the presentation, we encounter various extensions of the language which are left for future research.

V.1 Procedural Data

One of the reasons for considering a functional database model versus a relational one is to remove the dichotomy between data and queries. The removal of that dichotomy is also the motivation for introducing procedural fields in Postgres [S]. However, if the procedural fields solution is interesting as being an extension of the popular relational model, it certainly lacks the elegance of the functional solution. We believe that COL presents the advantages of both approaches by first being a relational extension, and also by making explicit use of functions to handle procedural-like data. The purpose of this section is to briefly investigate this issue.

Procedural data is introduced in [S] in order to blur the dichotomy between data and queries. Queries can be stored in the database in particular fields (called procedural fields). When the corresponding data is needed, the queries are activated.

Consider the database schema:

R(employee, manager, {hobby}),
S(employee, {phone}).

Suppose that the company policy is that managers can also be reached at their employees phone

numbers. The relation S can be defined intensionally using a function PHO, the facts:

$$5555 \in \text{PHO(John)}, 6666 \in \text{PHO(Peter)}, 7777 \in \text{PHO(Tom)}...$$

and using the rules:

$$w \in \text{PHO}(z) \leftarrow \text{R}(y,z,X), w \in \text{PHO}(y),$$
$$\text{S}(y,\text{PHO}(y)) \leftarrow.$$

To continue with the same example, some facts are known on relation R:

$$\text{R(John, Peter, \{chess,football\}), R(Peter, Max, \{bridge\})...}$$

Suppose that it is also known that employee Tom is managed by Peter, and does not have any hobby but the ones of his boss. Then one might want to store the fact:

$$\text{R(Tom, Peter, HOB(Peter))}$$

where the HOB function is defined by:

$$x \in \text{HOB}(y) \leftarrow \text{R}(y,z,X), x \in X.$$

The data functions therefore brings a lot of flexibility. The query R(John,Peter,X)? is answered by a simple access to the database, whereas the query R(Tom,Peter,X)? can be translated to the query $x \in$ HOB(Peter)? (if a lazy evaluation strategy is chosen). Furthermore, an update of Peter's hobbies will implicitly modify Tom's ones.

It should be noted that the above program is not stratified. Indeed, HOB > R because of the statement on employee "Tom", and R $\geq$ HOB because of the HOB defining rule. However, it is clearly possible to give a semantics to such programs. Intuitively, one has to consider a partial order of a set of atoms and terms. For instance, such an order would impose:

$$\text{R(Peter, Max, \{bridge\})} < \text{HOB(Peter)} < \text{R(Tom, Peter, \{bridge\})}.$$

This extension of the stratification is related to the local stratification in the sense of Przymusinski [P].

To conclude with this example, assume that it is known that Tom always has for hobbies the hobbies of his current boss. Then one might store the fact:

$$\text{R(Tom, Peter, HOB\_BOSS(Tom))}$$

where the HOB_BOSS function is defined by:

$$x \in \text{HOB\_BOSS}(y) \leftarrow \text{R}(y,z,X), x \in \text{HOB}(z).$$

The above program is also not stratified. Indeed, it is not even locally stratified according to [P]. The complex structure of facts should also be taken into account. For instance, two objects, say A and B, may be both intensionally defined with a subobject of each one of them depending on a subobject of the other.

V.2 Heterogeneous databases

We show how to integrate a relational database, and a functional one into a COL database. It is also possible to use a similar approach to define heterogeneous views when relations and functions are considered, and to restructure a relational database into a functional one, or conversely.

The main problem encountered in this context is that functional database models like FQL [BF] or Daplex [Sh] allow monovalued functions. A not too clean solution is to represent them using multivalued ones and enforce a oneness constraint. A more interesting solution is to extend the language with monovalued data functions. Rules like

$$x = F_1(y) \leftarrow R(x,y), \text{ and}$$

$$x = F(y) \leftarrow R(x,y),\ y = H(x)$$

have to be considered. The first rule yields inconsistency if in the extension of R, the first attribute does not functionally determine the second one. This can not be the case in the second rule. In both rules, the derived function may be only partially defined.

We now present an example with multivalued functions only. Consider the following two databases:

(a) *A RELATIONAL DATABASE:*

SHOW(film,theater,time)
PLAYS(actor,film)
LOCATION(theater,address)

(b) *A FUNCTIONAL DATABASE*

CASTING: film $\rightarrow\!\!\rightarrow$ actor
LOCATED: theater $\rightarrow\!\!\rightarrow$ address
EXHIB: film $\rightarrow\!\!\rightarrow$ theater, time

The two database can be integrated, for instance, in a COL database consisting of one function and one predicate:

(c) *THE INTEGRATING COL DATABASE*

The schema consists of the following:
GLOB_THEA(theater, address)
GLOB_INFO: theater $\rightarrow\!\!\rightarrow$ film, time
GLOB_FILM(film,{actor})

The integrating program is as follows:

GLOB_THEA(t,a) ← LOCATION(t,a)
GLOB_THEA(t,a) ← t ∈ theater(), a ∈ LOCATED(t)

a ∈ ACTSIN(f) ← PLAYS(a,f)
a ∈ ACTSIN(f) ← a ∈ CASTING(f)
GLOB_FILM(f,ACTSIN(f)) ← PLAYS(a,f)
GLOB_FILM(f,ACTSIN(f)) ← f ∈ film()

[f,h] ∈ GLOB_INFO(t) ← SHOW(f,t,h)
[f,h] ∈ GLOB_INFO(t) ← [t,h] ∈ EXHIB(f)

V.3 Semantic Database Modelling

The field of semantic database models (see, [HK] for a survey) has been primarily concerned with structures and semantics, and with notable exceptions like Daplex, language aspects have not been studied in depth. The COL language presents the advantages of dealing with complex objects, and of handling both data functions, and data relations. A consequence is that the language is more suited than other languages in the context of semantic database modelling.

In this section, we consider an example taken from the model IFO [AH1], and investigate what is still missing in the COL language to make it a language for the IFO model. The IFO model has been chosen here because it incorporates most structural aspects of semantic database models: it is an object-based model, with aggregation (tuple constructor), classification (set constructor), functions, specialization and generalization. Furthermore, the IFO model has been formally defined, which simplifies the investigation.

A first difficulty that is encountered comes from IFO nested functions. In IFO, the result of a function can itself be a function. Since this is a very peculiar aspect of IFO, we do not consider this feature here. We concentrate on an example given in [AH1] without nested function. The schema is shown in Figure V.1. We present a corresponding COL database, and then discuss the extensions of the language that need to be considered, and the limitations of the COL representation:

ABSTRACT TYPES are represented by basic domains:

hull

car

person

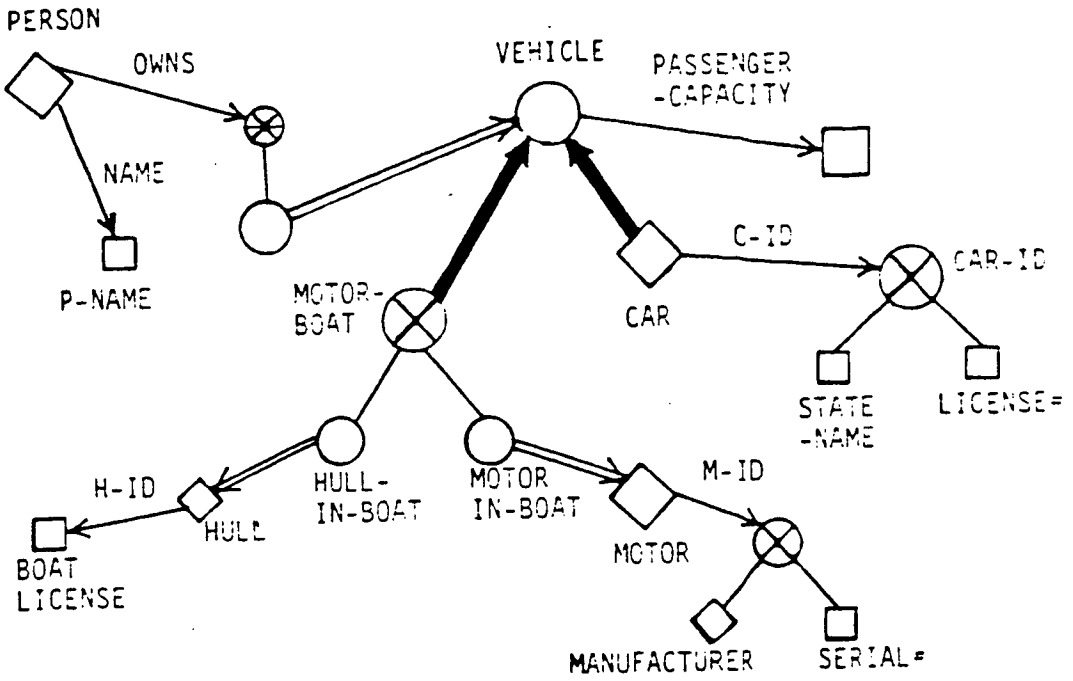

Figure V.1: an IFO schema

motor

manufacturer

CONSTRUCTED TYPES are represented by base objects:

MOTORBOAT(hull,motor)

CAR-ID(string,integer)

FUNCTIONS:

OWNS: person $\rightarrow$ {[hull,motor],car}

C-ID: car $\rightarrow$ [string,integer]

M-ID: motor $\rightarrow$ [manufacturer,string]

H-ID: hull $\rightarrow$ string

PASSENGER-CAPACITY: [hull,motor] $\rightarrow$ integer

PASSENGER-CAPACITY: car $\rightarrow$ integer

NAME: person $\rightarrow$ string

Some problems are posed by limitations of COL that were already mentioned:

- the type VEHICLE (i.e., either a MOTOR-BOAT or a CAR) can not be described, which yields a typing problem for the function PASSENGER-CAPACITY.
- some of the functions are monovalued.

As mentioned above, these problems can be overcome by considering simple extensions of the COL language. The introduction of *names* instead of the use of the numbering of tuple fields would be a simple modification of the language that would bring it closer to the IFO model.

Perhaps a more fundamental problem is that there is no explicit way of formulating ISA relationships. An extension of the language in that direction should be considered.

V.4 Evaluation of Datalog Queries

It is not our purpose here to explain another technique for evaluating datalog queries. We only want to hint that the COL language provides a nice formalism for studying such questions. We briefly consider the method of [GM]. Their proposal is to rewrite the relational system of equations used in a datalog query as a functional system of equations. Consider the famous Ancestor example:

$$\mathrm{ANC}(x,y) \leftarrow \mathrm{PAR}(x,y)$$
$$\mathrm{ANC}(x,y) \leftarrow \mathrm{PAR}(x,z),\ \mathrm{ANC}(z,y)$$

Let us introduce the following three rules:

$$x \in F_{\mathrm{PAR}}(y) \leftarrow \mathrm{PAR}(x,y)$$
$$x \in F_{\mathrm{ANC}}(y) \leftarrow x \in F_{\mathrm{PAR}}(y)$$
$$x \in F_{\mathrm{ANC}}(y) \leftarrow x \in F_{\mathrm{PAR}}(z),\ z \in F_{\mathrm{ANC}}(y).$$

Now if the query ANC(x,Tom)? is given, one can compute instead $F_{\mathrm{ANC}}(\mathrm{Tom})$ (i.e., answer the query $x \in F_{\mathrm{ANC}}(\mathrm{Tom})$?). In other words, a relational equation has been transformed into a functional equation:

$$F_{\mathrm{ANC}} = F_{\mathrm{PAR}} + F_{\mathrm{PAR}} \circ F_{\mathrm{ANC}}$$

where "+" stands for union and " $\circ$ " for the composition of multivalued functions.

In another proposal for evaluating datalog queries [B+], namely the magic sets approach, particular terms called "grouping terms" are used. It is easy to see that these terms correspond to particular derived data functions.

VI. CONCLUSION

The paper presents a language to manipulate complex objects based on recursive rules. The novelty is the use of data functions. The semantics of COL programs is defined as a canonical causal and minimal model using a sequence of fixpoint operators. In that sense, the semantics is constructive in nature.

We illustrated the use of the language in various database contexts: heterogeneous databases, semantic modelling, procedural data, and evaluation of datalog queries. This suggested extensions of the language: single-valued functions, explicit union of types constructor, structural stratification. Besides these issues which were just sketched in the present paper, other important questions are raised:

- the role of inheritance in the language, and
- updates for COL databases.

Last but not least remains the issue of an efficient implementation. There has been a lot of work on nested relations and complex objects. Few of them have so far been followed by an efficient implementation (e.g., the Verso system at Inria [V], and the Aim project at IBM Heidelberg [D]). We believe that the fixpoint semantics of COL programs makes such an implementation feasible. Indeed, the operators which are described in Section 4 can all be expressed in the algebra of complex objects of [AB].

REFERENCES

[AB] Abiteboul S., and C. Beeri, On the Manipulation of Complex Objects, abstract in Proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt (1987)

[ABi] Abiteboul, S., and N. Bidoit, "Non first normal form relations: an algebra allowing data restructuring", in *Journal of Computer Systems and Science* (1986),

[AG] Abiteboul, S., and S. Grumbach, COL: a Language for Complex Objects based on Recursive Rules, abstract in Proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt (1987)

[AH1] Abiteboul S., and Hull, R., "IFO: A formal semantic database model," Proc. ACM SIGACT/SIGMOD Symposium on Principle of Database Systems (1984), to appear in *ACM Transactions on Database Systems.*

[AH2] Abiteboul S., and Hull, R., "Object restructuring in semantic database models," Proc. Intern. Conf. on Database Theory, Roma (1986) to appear in *Theoretical Computer Science*

[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)

[B+] Bancilhon F., et al, Magic Sets and Other Strange Ways to Implement Logic Programs, Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems (1986)

[BH] Bidoit, N., R. Hull

[BK] Bancilhon, F., and Khoshafian, S., "A calculus for complex objects, Proc. ACM SIGACT/SIGMOD Symposium on Principle of Database Systems (1985).

[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)

[BH] Bidoit, N., R. Hull, Positivism vs. Minimalism in Deductive Databases, proc. ACM SIGACT-SISMOD Symposium on Principles of Database Systems (1986)

[BF] Buneman, P., R.E. Frankel, FQL - a Functional Query Language (preliminary report) proc. ACM SIGMOD conf. on Management of Data (1979)

[D] Dadam, P., History and Status of the Advanced Information Management Prototype, Proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt (1987)

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

[GM] Gardarin G., C. de Maindreville, Evaluation of Database Recursive Logic Programs as Recurrent Function Series, proc. ACM SIGMOD conf. on Management of Data (1986)

[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)

[HK] Hull, R., R. King, Semantic database modeling: Survey, applications, and research issues. U.S.C. Computer Science Technical Report (1986) to appear in ACM computing surveys

[HY] Hull, R., C.K. Yap, The format model: A theory of database organization. Journal of the ACM 31(3) (1984)

[J] Jacobs, B., on Database Logic, *Journal of the ACM* (1982).

[JS] Jaeschke, B., H.J. Schek, Remarks on the algebra of non first normal form relations, Proc. ACM SIGACT/SIGMOD Symposium on Principle of Database Systems, Los Angeles (1982)

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

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

[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)

[P] Przymusinski, T. C. On the Semantics of Stratified Deductive Databases and Logic Programs, to appear in *Journal of Logic Programming*

[SS] Schek H., and M. Scholl, the Relational Model with relation-valued attributes, in *Information Systems* (1986)

[S] Stonebraker M., Object Management in Postgres using Procedures, in the Postgres Papers, UCB report (1986)

[Sh] Shipman, D., The Functional Data Model and the Data Language Daplex, *ACM Transactions on Database Systems.* (1981)

[V] Verso, J., (pen name for the Verso team), Verso: a Database Machine Based on non-1NF Relations, Inria Internal Report (1986)

APPENDIX

In this appendix, Theorems IV.1 and IV.2 are proven.

To prove Theorem IV.1, we first show that each monostratum program is growing, X-finitary and stable on X, for some X. To do that, we use the following technical lemma:

Lemma A.1: Let J and K be two interpretations such that $J|_X = K|_X$ for a given set X of symbols, and θ_J and θ_K two valuations with $\theta_J x = \theta_K x$ for each variable x. If t is a term such that each function symbol occurring in t belongs to X, then $\theta_J t = \theta_K t$.

Proof: The result is obvious if t contains no function symbols. Now consider $t = F(t_1,...,t_n)$ where F is in X and $t_1,...,t_n$ contain no function symbol. Then

$$\begin{aligned}
\theta_J F(t_1,...,t_n) &= \{x \mid [x \in F(\theta_J t_1,..., \theta_J t_n)] \in J\}, \text{ by definition,} \\
&= \{x \mid [x \in F(\theta_J t_1,..., \theta_J t_n)] \in J|_X\}, \text{ since F is in X,} \\
&= \{x \mid [x \in F(\theta_J t_1,..., \theta_J t_n)] \in K|_X\}, \text{ since } J|_X = K|_X, \\
&= \{x \mid [x \in F(\theta_K t_1,..., \theta_K t_n)] \in K|_X\}, \text{ since } t_1,...,t_n \text{ contain no function symbol,} \\
&= \{x \mid [x \in F(\theta_K t_1,..., \theta_K t_n)] \in K\}, \text{ since F is in X,} \\
&= \theta_K F(t_1,...,t_n).
\end{aligned}$$

By induction of the imbrication of function symbols, $\theta_J t = \theta_K t$ for each term t containing only function symbols in X. □

We now consider X-finitarity and stability.

Lemma A.2: Let P be a monostratum program, and X the set of symbols in P which are not defined in P. Then T_P is X-finitary and stable on X.

Proof: Consider first stability on X. For each interpretation I of P, $T_P(I)$ contains only atoms that are built from a defined symbol. Thus $(T_P(I))|_X = \phi \subseteq I|_X$, so T_P is stable on X.

We next prove that T_P is X-finitary. Let (I_n) be a growing sequence of interpretations such that $I_n|_X = I_0|_X$ for all n. Let $J = \bigcup_{n=0}^{\infty} I_n$ and let $A \in T_P(J)$. To conclude the proof, it suffices to

show that $A \in T_P(I_k)$ for some k. Since $A \in T_P(J)$, A is the result of applying a rule $r : A' \leftarrow L_1,...,L_m$ in P with a valuation θ_J. For each k, let $\theta_{Ik}x = \theta_J x$ for each variable x. We shall prove that A is the result of applying r with θ_{Ik} for some k.

Let X be the set of symbols that are not defined in P. Clearly, $J|_X = I_k|_X$ for each k. Let $t_1 \in F(t_2,...,t_n)$ or $P(t_1,...,t_n)$ be an atom in rule r, and let $i \in [1..n]$. Each function symbol G appearing in t_i is a total determinant, and thus is not defined since P is monostratum. Since $J|_X = I_k|_X$, $\theta_{Ik}t_i = \theta_J t_i$ by Lemma A.1. Thus

(+) $\theta_{Ik}t_i = \theta_J t_i$, for each atom $t_1 \in F(t_2,...,t_n)$ or $P(t_1,...,t_n)$ in rule r, and each $i \in [1..n]$.

Let $j \in [1..m]$. Since A is the result of applying r with θ_J, $J \models \theta_J L_j$. We prove that for k large enough, $I_k \models \theta_{I_k} L_j$. We distinguish four cases:

(1) $L_j = P(t_1,...,t_n)$. Then, $\theta_J L_j \in J$. Thus there exists an integer k(j) such that for all $k \geq k(j)$, $\theta_J L_j \in I_k$. By (+), $\theta_J L_j = \theta_{Ik} L_j$ for all $k \geq k(j)$. Thus for all $k \geq k(j)$, $\theta_{Ik} L_j \in I_k$, i.e., $I_k \models \theta_{Ik} L_j$.

(2) $L_j = [t_1 \in F(t_2,...,t_n)]$. Then $[\theta_J t_1 \in F(\theta_J t_2,..., \theta_J t_n)] \in J$. Then there exists an integer k(j), such that for all $k \geq k(j)$, $[\theta_J t_1 \in F(\theta_J t_2,..., \theta_J t_n)] \in I_k$. By (+), for all $k \geq k(j)$,
$[\theta_{Ik} t_1 \in F(\theta_{Ik} t_2,...,\theta_{Ik} t_n)] \in I_k$, i.e., $I_k \models \theta_{Ik} L_j$.

(3) $L_j = \neg P(t_1,...,t_n)$. Then $\theta_J P(t_1,..., t_n) \notin J$. Then there exists an integer k(j) (k(j) = 0), such that for all $k \geq k(j)$, $\theta_J P(t_1,...,t_n) \notin I_k$. By (+), $\theta_J P(t_1,...,t_n) = \theta_{Ik} P(t_1,...,t_n)$, for all k. Thus for all $k \geq k(j)$, $\theta_{Ik} P(t_1,...,t_n) \notin I_k$, i.e., $I_k \models \theta_{Ik} L_j$.

(4) The last case is treated similarly.

For each j in [1..m], and each $k \geq k(j)$, $I_k \models \theta_{Ik} L_j$. Let $k = \sup(k(j))$, then $I_k \models \theta_{Ik} L_1 \wedge \cdots \wedge \theta_{Ik} L_n$. Let A_k be the result of applying the rule r with θ_{Ik}. By (+), $A = A_k$. Thus $A \in T_P(I_k)$. □

We also have:

Lemma A.3: If P is monostratum, T_P is growing.

Proof: Let P be a monostratum program. Let I, J, M be interpretations such that $I \subseteq J \subseteq M \subseteq T\uparrow\omega(I)$. We prove that if $A \in T_p(J)$, then $A \in T_p(M)$.

Suppose that $A \in T_p(J)$. Then A is the result of applying the rule $r : A' \leftarrow L_1,...,L_m$ in P with a valuation θ_J. Let θ_M be a valuation such that $\theta_M x = \theta_J x$ for all variables x.

Let X be the set of symbols that are not defined in P. Clearly, $I|_X \subseteq J|_X \subseteq M|_X \subseteq (T_P\uparrow\omega(I))|_X = I|_X$. Let $t_i \in F(t_2,...,t_n)$ or $P(t_1,...,t_n)$ be an atom in rule r, and let $i \in [1..n]$. Each function symbol G appearing in t_i is a total determinant, and thus is not defined since P is monostratum. Since $J|_X = M_X$, $\theta_M t_i = \theta_J t_i$ by Lemma A.1. Thus

(+) $\theta_M t_i = \theta_J t_i$, for each atom $t_i \in F(t_2,...,t_n)$ or $P(t_1,...,t_n)$ in rule r, and each $i \in [1..n]$.

We prove that $M \models \theta_M L_i$ for each i. Like in the previous lemma, there are four cases. We consider here the last case only. The others are left to the reader.

(4) Let $L_i = \neg [t_1 \in F(t_2,...,t_n)]$. Since A is the result of applying the rule with θ_J, $J \models \theta_J L_i$. Thus $[\theta_J t_1 \in F(\theta_J t_2,...,\theta_J t_n)] \notin J$. Thus, by (+), $[\theta_M t_1 \in F(\theta_M t_2,..., \theta_M t_n)] = [\theta_J t_1 \in F(\theta_J t_2,...,\theta_J t_n)] \notin J$. Let $B = [\theta_M t_1 \in F(\theta_M t_2,..., \theta_M t_n)]$. Since $B \notin J$, $B \notin J|_X = M|_X$. Since the literal is negative, F is a total determinant of P. Thus F is not a defined symbol of P (P is monostratum), i.e., $F \in X$. Hence $B \notin M$. Therefore, $[\theta_M t_1 \in F(\theta_M t_2,..., \theta_M t_n)] \notin M$, i.e., $M \models L_i$.

In each case, $M \models \theta_M L_i$. Let A" be the result of applying rule r with θ_M. By (+), A" = A. Thus $A \in T_P(M)$. □

The following proposition will be essential in the proof of Theorem IV.1.

Proposition A.1: Let T be an X-finitary, stable on X, and growing operator. Then for all I,

(a) $T(T\uparrow\omega(I)) \subseteq T\uparrow\omega(I)$, and

(b) $T\uparrow\omega(I) \subseteq T(T\uparrow\omega(I)) \cup I$.

Proof: First consider (a). Since T is stable on X,

$$(T\uparrow(n+1)(I))|_X = (T\uparrow n(I))|_X = (T\uparrow 0(I))|_X.$$

Thus the sequence $(T\uparrow n(I))$ is growing and $(T\uparrow n(I))|_X = (T\uparrow 0(I))|_X$. By the X-finitarity of T,

$$(+)\ T(\bigcup_{n=0}^{\infty} T\uparrow n(I)) \subseteq \bigcup_{n=0}^{\infty} T(T\uparrow n(I)).$$

Thus, $T(T\uparrow\omega(I)) = T(\bigcup_{n=0}^{\infty} T\uparrow n(I))$

$\subseteq \bigcup_{n=0}^{\infty} T(T\uparrow n(I))$ by (+)

$\subseteq \bigcup_{n=1}^{\infty} \mathbf{T}\uparrow n(I) = T\uparrow\omega(I).$

Now consider (b). Let $A \in T\uparrow\omega(I)$. Then either $A \in I$, or there exists $n \geq 1$ such that $A \in T\uparrow n(I)$. Thus either $A \in I$, or there exists $n \geq 0$ such that $A \in T(T\uparrow n(I))$. Since $I \subseteq T\uparrow n(I) \subseteq T\uparrow\omega(I)$, and since T is growing, $T(T\uparrow n(I)) \subseteq T(T\uparrow\omega(I))$. Thus $A \in T(T\uparrow\omega(I)) \cup I$. □

Theorem IV.1, which exhibits a minimal (pre)-fixpoint of T_P, is a straightforward consequence of Lemmas A.2, A.3 and Proposition A.1.

Theorem IV.1: Let P be a monostratum program. Then for each I,

- $T_P\uparrow\omega(I)$ is a minimal pre-fixpoint of T_P containing I.
- $T_P\uparrow\omega(\varphi)$ is a minimal fixpoint of T_P.

Proof: By Lemmas A.2 and A.3, T_P is stable on X, X-finitary, and growing. By definition, $T_P\uparrow\omega(I)$ contains I. Thus, by Proposition A.1 (a), $T_P\uparrow\omega(I)$ is a pre-fixpoint of T_P containing I. By Proposition A.1 (b), $T_P\uparrow\omega(\phi)$ is therefore a fixpoint of T_P.

Now consider the minimality. Suppose that there exists an interpretation J which is a pre-fixpoint of T_p such that $I \subseteq J \subseteq T_p\uparrow\omega(I)$. To conclude the proof, it suffices to show that $T_P\uparrow\omega(I) \subseteq J$.

First, $T_P\uparrow 0(I) = I \subseteq J$. Suppose that $T_P\uparrow n(I) \subseteq J$ for some n. Then $I \subseteq T_P\uparrow n(I) \subseteq J \subseteq T_P\uparrow\omega(I)$. Since T_P is growing, $T_P(T_P\uparrow n(I)) \subseteq T_P(J) \subseteq J$. Thus $T_P\uparrow(n+1)(I) = T_P(T_P\uparrow n(I)) \cup T_P\uparrow n(I) \subseteq J$. By induction, $T_P\uparrow\omega(I) = \bigcup_{n=0}^{\infty} T_P\uparrow n(I) \subseteq J$. □

Arbitrary stratified programs are now considered. First recall the notion of iterative powers

of a sequence of operators, and the locality property [ABW].

Definition: Let $T_1,...,T_m$ be a sequence of operators. The **iterative powers** of that sequence **w.r.t.** an interpretation I are defined by:

- $K_0 = I$, and
- $K_i = T_i \uparrow \omega(K_{i-1})$ for each $i \in [1..m]$.

The sequence of operators $T_1,...,T_m$ is **local**, if for each I and J such that $I \subseteq J \subseteq K_m$, $T_i(J) = T_i(J \cap K_i)$.

Let $P = P_1 \cup ... \cup P_m$ be a stratified program. With the first stratum, we associate an operator T_1; with the second one, an operator T_2; and so on. Then we have:

Lemma A.4: Let $T_1,...,T_m$ be the sequence of operators corresponding to a stratified program $P = P_1 \cup ... \cup P_m$. This sequence is local.

Proof: First suppose that $T_i(J) \not\subseteq T_i(J \cap K_i)$ for some i. Let A be in $T_i(J) - T_i(J \cap K_i)$. Then A is the result of applying some rule r in P_i. Since $J \cap K_i \subseteq J$, and $A \notin T_i(J \cap K_i)$, the application of the rule uses a fact B not in $J \cap K_i$. Suppose that $B = [b_1 \in F(b_2,...,b_n)]$. (The case $B = P(b_1,...,b_n)$ is similar). Since B is used in the application of r,

(i) F is a determinant of r in P_i.

Since B is in $K_m - K_i$, B is the result of the application of a rule r' in P_j for some $j > i$. Thus

(ii) F is the defined symbol of a rule r' in P_j for $j > i$.

Clearly, (i) and (ii) together contradict the stratification condition on $\mathbf{P}_1 \cup ... \cup \mathbf{P}_{i-1}$. Hence, $T_i(J) \subseteq T_i(J \cap K_i)$. The reverse inclusion is proved in a similar way. □

Theorem IV.2 will be a straightforward consequence of the following proposition:

Proposition A.2: Let $T_1,...,T_m$ be a local sequence of operators such that for each $i \in [1..m]$, T_i is growing, X_i-finitary and stable on X_i, for some X_i. For each instance I, let (K_i) be the iterative powers of $T_1,...,T_n$ w.r.t. I. Then[3]

[3] By definition, $(\bigcup_{i=1}^{m} T_i)J = \bigcup_{i=1}^{m} (T_iJ)$.

(1) $(\bigcup_{i=1}^{m} T_i)K_m \subseteq K_m$, and

(2) $K_m \subseteq (\bigcup_{i=1}^{m} T_i)K_m \cup I$.

Proof: Let I be an interpretation. Recall the result of the Proposition A.1: if T is growing, X-finitary and X-stable for some X,

(a) $T(T\uparrow\omega(I)) \subseteq T\uparrow\omega(I)$, and

(b) $T\uparrow\omega(I) \subseteq T(T\uparrow\omega(I)) \cup I$.

(1) $(\bigcup_{i=1}^{m} T_i)K_m = \bigcup_{i=1}^{m} T_iK_m$, by definition,

$\subseteq \bigcup_{i=1}^{m} T_iK_i$, by locality,

$\subseteq \bigcup_{i=1}^{m} K_i$, by (a),

$\subseteq K_m$.

(2) Conversely,

$K_m = T_m\uparrow\omega(K_{m-1})$, by definition,

$\subseteq T_mK_m \cup K_{m-1}$, by (b),

$\subseteq (\bigcup_{i=1}^{m} T_iK_i) \cup I$, by induction,

$\subseteq (\bigcup_{i=1}^{m} T_iK_m) \cup I$, by locality,

$= (\bigcup_{i=1}^{m} T_i)K_m \cup I$, by definition. □

We now conclude:

Theorem IV.2: Let $P = P_1 \cup \ldots \cup P_m$ be a stratified program, and $T_1,\ldots,T_m$ be the corresponding operators,

- $K_0 = \varphi$, and
- $K_i = T_i\uparrow\omega(K_{i-1})$ for each $i \in [1..m]$.

Then K_m is a minimal fixpoint of $\bigcup_{i=0}^{m} T_i$. Thus K_m is a minimal causal model of P.

Proof: By Proposition IV.2, it suffices to show that K_m is a minimal fixpoint of $\bigcup_{i=0}^{m} T_i$. By Lemma A.4, the sequence of operators is local. Thus, by Proposition A.2,

(1) $(\bigcup_{i=1}^{m} T_i)K_m \subseteq K_m$,

(2) $K_m \subseteq (\bigcup_{i=1}^{m} T_i)K_m$.

Therefore, K_m is a fixpoint of $\bigcup_{i=0}^{m} T_i$. It remains to show the minimality.

Let J be a pre-fixpoint of $\bigcup_{i=1}^{m} T_i$. We prove by induction on k that

(*) if $J \subseteq K_k$, then $K_k \subseteq J$.

For $k = 0$, $K_0 = \phi \subseteq J$. Suppose (*) is true for a certain k (first induction hypothesis). We prove by induction that :

(**) $T_{k+1} \uparrow j(K_k) \subseteq J$,

For $j = 0$, it is by hypothesis. Suppose it is true for a certain j (second induction hypothesis). By (**), $K_k \subseteq T_{k+1}\uparrow j(K_k) \subseteq J \cap K_{k+1} \subseteq T_{k+1}\uparrow\omega(K_k)$. Since T_{k+1} is growing,

(+) $T_{k+1}(T_{k+1}\uparrow j(K_k)) \subseteq T_{k+1}(J \cap K_{k+1})$.

Hence, $T_{k+1} \uparrow (j+1)(K_k) = T_{k+1}(T_{k+1}\uparrow j(K_k)) \cup T_{k+1}\uparrow j(K_k)$, by definition,

$\subseteq T_{k+1}(T_{k+1}\uparrow j(K_k)) \cup J$, by second induction hypothesis,

$\subseteq T_{k+1}(J \cap K_{k+1}) \cup J$, by (+),

$= T_{k+1}(J) \cup J$, by locality,

$\subseteq J$, since J is a pre-fixpoint of T_{k+1}.

Thus (**) holds for all j. By induction, (*) holds for all k. In particular, for $k = m$, if J is a pre-fixpoint of $\bigcup_{i=1}^{m} T_i$ such that $J \subseteq K_m$, then $K_m \subseteq J$ which concludes the proof. □

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