S-Logic: A Higher-Order Logic Language for Deductive Databases

Mengchi Liu and John Cleary
Department of Computer Science
University of Calgary
Calgary, Alberta Canada T2N 1N4

February 16, 1990

Abstract

Deductive databases based on relational databases and Prolog techniques are not suitable for complex object modeling. The problems result from the underlying relational model and the pure Prolog which use inexpressive flat structures. Semantic data models using data abstractions and inheritance greatly increase expressiveness. So do extended terms with internal structure in logic programming. This paper proposes a higher-order logic language for deductive databases called S-logic which is the combination of the semantic data model and extended term approaches. It supports object identity, semantic data abstractions and inheritance very naturally and allows the definition and manipulation of database schema and data in an integrated framework.

1 Introduction

A primary goal of deductive databases is to express facts, deductive information, meta-information and queries in a uniform way. However, deductive databases based on relational database and pure Prolog techniques [7, 19] cannot really satisfy this goal. They can not represent real world information such as object identity, complex objects and inheritance naturally do not support schemas and higher-order queries in an integrated framework [15]. The problems result from the underlying relational data model and pure Prolog techniques themselves.

The relational data model is record-based and has a number of serious limitations, especially from the user's point of view. In response, a number of semantic data models have been proposed in order to provide mechanisms and constructs that mirror the prevalent kinds of relationships naturally arising between data stored in a database [2, 4, 5, 8, 21]. Four abstraction mechanisms

are found essential to most semantic data models: classification, aggregation, association and generalization [2, 3, 5, 8, 11, 18, 20, 21]. Inherent in abstractions is property inheritance. Generalization and classification support top-down inheritance, while aggregation and association support bottom-up inheritance. Inheritance enables us to reduce the redundancy of the schema specification while maintaining its completeness. Using some of the four abstractions for relating objects and constructing new objects, very complex objects can be easily represented and the advantage of inheritance obtained. This capability greatly increases the expressive power of semantic data models. It is argued in [19] that first-order language could use logical implication to express data abstractions other than association and their inheritances. However as discussed in [10], this approach does not capture what we really mean even though it is semantically satisfactory. The problem is that traditional first-order logic use inexpressive flat structures which can only represent names of object and relationships among those names, rather than the existence and internal structure of objects.

In deductive databases, we have to reason about schema information. Therefore, we need higher-order logics. But higher-order logics have been met with skepticism since the unification problem is undecidable. So normally, a separate language is provided to specify and manipulate the schema information.

A number of attempts have been made to solve the above problems. Among them, ψ-terms [10], O-logic [13, 17], C-logic[6], COL [1] and F-logic [12] are most notable because they can naturally model the existence and internal structure of objects. But they support only some of the abstractions and inheritance rather than all of them. This causes some problems as discussed in detail in [15].

Some work has been done towards the integration of reasoning about schema and data. It was suggested that the useful parts of higher-order logic can be given a first-order semantics by encoding them in predicate calculus [9]. However, this provides only an indirect semantics and cannot really solve the problem. Based upon bottom-up semantics where unification is replaced by matching, [14] proposed a higher-order language for deductive databases which is decidable. However, it does not solve the other problems.

In this paper we propose a deductive database language, called S-logic which extends ψ-terms, O-logic, C-logic, COL and F-logic and is capable of representing the four abstractions and inheritance in general. Besides, S-logic provides the capability of expressing and manipulating schema and higher-order information in an integrated manner. The higher-order semantics of S-logic is decidable based on the replacement semantics proposed in [14].

The organization of the paper is as follows. By means of examples, we introduce S-logic informally in Section 2. Then in Section 3, we formally define

its semantics.

2 Informal Presentation and Examples

An S-logic program consists of four parts: the type system, database, rules, and queries. The type system is the schema of the database and consists of all type definitions, a lattice over all types. A type in S-logic is a name which specifies a set of objects having the same properties. There are five kinds of types in S-logic: basic types, set types, disjunctive types, abstract types and built-in types. The user specifies all the types other than basic types and built-in types explicitly in type definitions or implicitly in rules. The basic types include integer, string and their subsets. A set type specifies a structure of elements of identical type, called set element type. For example, $\{integer\}$ specifies a set type whose set element type is *integer*. An abstract type specifies the set of elements and the properties the elements have as well. A property is described by a function called *isa* or attributes from the specified type to a type. The *isa* label is used for specifying subtyping relationship and performing automatic property inheritance. Figure 1 shows several examples of abstract types. *Student* is a subtype of *person*. Therefore, it inherits (via *isa*) all the properties the *person* has with a restriction on age. Besides, it has its own property of studying a number of courses in type $\{course\}$. There are two built-in types in S-logic. One is called *all* which includes all objects in every type and may has no properties at all. The other is called *none* which has no objects and has all the existing properties. Based on the subtype relationships, S-logic's type system forms a lattice. Types in the lattice are either built-in, user-defined, or inferred from what the user has defined. Figure 2 shows a lattice based on the type definition of Figure 1.

A database in S-logic consists of all objects that satisfy the type system. Every object belonging to an abstract type has all the properties of this type. Since subtype inherits the properties of its supertype, no *isa* label is in the database. For every type other than *none*, there is a biggest element $\top$ which means any unknown object, and a smallest element $\bot$ which means empty element. For the *none*, it has only one element $\bot$. If a property of a object is not specified, S-logic will assume a default value $\top$ of the corresponding type. If a conflict happens, S-logic will infer a value $\bot$ of the corresponding type. Whenever the type information is clear, the type name may be omitted for convenience. Figure 3 shows a database corresponding to the type system of figure 1.

In S-logic, a type itself corresponds to a classification and its elements inherit all the properties of the type. An *isa* label corresponds to a generalization and the properties of super type is automatically inherited by the subtype. A set type corresponds to association which inherit the properties of its set element

type. An abstract type definition with labels other than *isa* corresponds to aggregation.

```
person(name → string,
       age → integer({0..120}),
       sex → string({'Male', 'Female'}),
       father → person,
       mother → person).

student(isa → person(age → integer({0..30}))),
        study → {course}).

employee(isa → person(age → integer({25..65}),
           works_at → dept).

assistant(isa → student, isa → employee).

dept(name → string, head → person).

course(name → string, credit → integer).
```

Figure 1. A sample type system

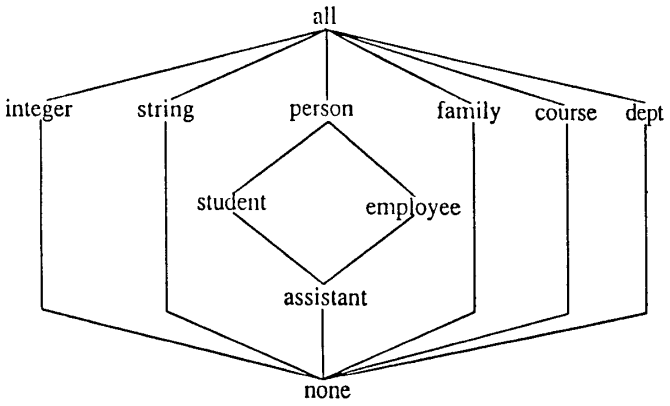

Figure 2. The lattice over the sample type system

Based on the database, deductive information can be defined by using rules in S-logic. We can define a new type by describing how to construct its objects, or we can define a new attribute for an existing type etc. Figure 4 shows several rules defined on the database. The first one defines a *family* type which has a set grouping attribute *children* based on the existing *person* type. The existence of objects of the *family* type depends on the existence of the objects

X and Y. The function *id* is an object constructor. The second rule defines another property of *employee* type from the existing information. For a set type, its elements are sets and a variable can be used to range over the elements. {X} however is used to stand for one specific element which is a set and X ranges over elements of this set. S-logic's objects are determined by functions.

jenny:student(name → 'Jenny', age → 24, sex → 'Female',
father → john, study → {c213,m251}).

john:employee(name → 'John', age → 56, sex → 'Male',
works_at → cpsc).

henry:employee(name → 'Henry', age → 37, sex → 'Male',
work_at → math).

smith:assistant(name → 'Smith', age → 29, sex → 'Male',
works_at → cpsc, study → {c671, c657}).

dennis:person(name → 'Dennis', age → 5, sex → 'Male',
father → smith, mother → jenny).

cpsc:dept(name → 'Computer Science', head → john).

math:dept(name → 'Mathematics', head → henry).

c211:course(name → 'Programming Language', credit → 3).

m251:course(name → 'Calculus', credit → 6).

c657:course(name → 'Distributed System', credit → 3).

c671:course(name → 'Artificial Intelligence', credit → 3).

Figure 3: A sample database

Queries can be defined over the database, rules and the type system in a uniform way. Figure 4 shows several query examples. The first query asks for all the employees' name who are female and work at the computer science department and whose ages are not greater than 30. The only answer is X = 'Smith', Y = 29. Note that object *smith* belongs to type *assistant* and *assistant* is a subtype of *employee*. The second query asks for all of jenny's children. Here {X} is a set grouping variable similar to [22]. The answer is X = dennis The third query asks for john's type information, properties and the corresponding values. The type *employee* and *person* will be given as answers to X, and *name*, *age*, etc. will be given to L, and *'John'*, 56, etc. will be given to Y as corresponding values. The fourth query asks for henry's type information. Except *string*, *integer*, *dept*, etc. will be given to y, the rest are the same as the third query. The last query asks for all information about the type system.

id(X,Y):family(father → X, mother → Y, children → {Z})
⇐ Z:person(father → X, mother → Y).

E:employee(supervisor → M) ⇐ E(works_at → D:dept),
D(head → M).

(1) ? :employee(name → X, sex → 'Female', age → Y,
works_at → cpsc), Y ≤ 30.

(2) ? :family(mother → jenny, children → {X}).

(3) ? john:X(L → Y).

(4) ? henry:X(L → :Y).

(5) ? X(L → Y).

Figure 4: Rules and Queries

3 A Formal Presentation

In this section, we introduce the syntax and semantics of the basic S-logic. First, we define types, database, rules and queries. Then we define its semantics.

3.1 Syntax

The alphabet of S-logic consists of (1) the set $A = \{all, none, \top, \bot\}$, (2) the set $B = \{integer, string\}$, (3) a set D which is the union of the set Z of integers, the set S of strings. (4) a countably infinite set O of object identifiers, (5) a finite set C of type constructors, (6) a finite set L of labels which are of three kinds: single-valued, set-valued and *isa* labels. (7) a countably infinite set F of function symbols, which are called object constructors, (8) an infinite set V of variables, and (9) logical connective $\Leftarrow$, (10) parentheses, arrows, :, etc. Here we assume that the sets A, B, D, O, C, F, L, V are disjoint.

TYPES and SUBTYPES:(1) Every element of B is a type, called a basic type. If s is a basic type, then $s(\{a_1, ..., a_n\}), (n \geq 1)$ is also a basic type and $a_1, ..., a_n$ are called elements of the type. If s is an *integer*, then $s(\{lb..rb\})$ is also a basic type and lb and rb are called the range of this type. (2) If s is a type, then $\{s\}$ is a type called a set type. s itself is called a set element type. (3) If $s_1, ..., s_n$ are types $(n \geq 2)$, then $s_1|...|s_n$ is a type, called a disjunctive type. Each s_i is a subtype of $s_1|...|s_n$. (4) If $p_1, ..., p_n$ belong to C or B, p belongs to C, $l_1, ..., l_n$ are labels, and we have the definition $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$ then p is a type, called an abstract type. If l_i is an *isa* label, then p is a subtype of p_i. If l_i is other than *isa* label, then $l_i \rightarrow p_i$ is called the property of p, l_i is called the attribute and p_i is called the component type of p. If p_i is a set type, then l_i is a set-valued label, otherwise l_i is a single-valued label. (5) *all* and *none* are

types, called built-in types, which represent the biggest type and the smallest type respectively. Every type is a subtype of *all* and every type has *none* as a subtype. All the types are partially ordered by the subtype relationship. We use '$\leq$' to represent subtype relationship between types.

All types except basic types and built-in types are user-defined types. If a user defined type does not contain a nested type definition, then it is called a flat type. Without losing generality, we restrict ourselves to flat types later on. If all user-defined types are specified in a top-down fashion, i.e. the present types are defined based on the existing types, then all the types form a lattice under $\leq$ with *all* as a top element and *none* as a bottom element.

TYPE SYSTEM: A type system is a set of type definitions which forms a lattice under the subtype relationship. If a type p is a subtype of two supertypes q, r, then there should be a type other than *all* such that q, r are its subtypes under the transferrable relationship[1].

VALUES, VARIABLES and OBJECT IDENTITIES: Every element of D and O is a value, called a basic value. If $v_1, ..., v_p$ are basic values, then $\{v_i, ..., v_p\}$ is a set value. $\{\}$ denotes the empty set value. Every type is a value, called a type value. Every label is a value, called a label value. Every element of V is a variable. There are four kinds of variables in V: single-valued variables, set-valued variables, type variables and label variables. If $X_1, ..., X_q$, $(q \geq 1)$ are single-valued variables, then $\{X_1, ..., X_q\}$ is a set grouping variable. If $Y_1, ..., Y_n, (n \geq 1)$ are single-valued variables or basic values, and f belongs to F, then $f(Y_1, ..., Y_n)$ is an object identity.

TYPE S-TERMS, BASIC S-TERM and S-TERM: (1) If P is a type variable, then P is a type S-term (2) If P is an abstract type or type variable, $P_1, ..., P_n$ are types or type variables, and $L_1, ..., L_n (n \geq 1)$ are labels or label variables, then $P(L_1 \rightarrow P_1, ..., L_n \rightarrow P_n)$ is a type S-term. (3) If p is a type, X is either a basic variable, a basic value, or object identity, then $X : p$ is a basic S-term. (4) If p is an abstract type with properties $l_i \rightarrow p_i, (1 \leq i \leq n)$ and $X, X_1, ..., X_n$ are either basic variables, basic values or object identities, then $X : p(l_1 \rightarrow X_1 : p_1, ..., l_n \rightarrow X_n : p_n)$ is a basic S-term. (5) Type S-terms and basic S-terms are S-terms. (6) If $P, P_1, ..., P_n (n \geq 0)$ are types or type variables, $L_1, ..., L_n$ are labels or label variables other than *isa*, and $X, X_1, ..., X_n$ are either basic variables, basic values and object identities, then $X : P(l_1 \rightarrow X_1 : P_1, ..., l_n \rightarrow X_n : P_n)$ is an S-term.

If X is a single-valued or set-valued variable and its name is not of interest

[1] This definition gives a restriction on subtype relationship which is similar to the global restrictions discussed in [11]

then $X : P$ can be replaced by $: P$ for convenience in S-terms. According to the above definition, there should not be any *isa* labels in S-terms other than type S-terms. To simplify the notation we assume the following convention: if an attribute l and the corresponding type p is omitted in an S-term then the intention is $l \rightarrow \top : p$. Furthermore, if a class specification is omitted and cannot be inferred, then *all* is assumed.

DATABASES: A database is the set of all ground basic S-terms (or base relations).

LITERALS: (1) An S-term is a positive literal. (2) $\psi_1 = \psi_2$, $\psi_1 \leq \psi_2$, $\psi_1 \geq \psi_2$, $\psi_1 < \psi_2$, $\psi_1 > \psi_2$ $\psi_1 \neq \psi_2$, are all positive literals, where ψ_1, ψ_2 are S-terms. (3) If p is a positive literal, ¬p is a negative literal.

RULES and QUERIES: A *rule* is an expression of the form $p \Leftarrow p_1, ..., p_n$ where the body $p_1, .., p_n$ is a conjunction of literals; and the head p is an S-term. Every fact in the database is a rule with an empty body. A *query* is a rule with an empty head and is denoted as ? p1,...,pn.

PROGRAMS: A *program* is a quadruple $\langle S, DB, R, Q \rangle$, where S is a type system, DB is a database, R is a finite collection of rules and Q is a query. A flat program is a program whose types, database, rules and query are flat, i.e. without any nested values or variables.

3.2 Semantics

INTERPRETATION: Given a language P of S-logic, its interpretation I is a tuple $\langle U, \Sigma, \Lambda, \pi, g_C, g_L, g_O, g_F, \rangle$ where U is a possibly infinite universe of all objects, values in the real world, Σ is a finite set of semantic types which stand for subsets of U under the mapping π. Λ is a finite set of semantic labels, which is a sum of three other mappings: $\Lambda_\#, \Lambda_*, \Lambda_{isa}$, where $\Lambda_\#$ is a single-valued mapping $U \rightarrow U$, Λ_* is a a set-valued mapping $U \rightarrow 2^U$, and Λ_{isa} is an identical mapping $U \rightarrow U$. π is a mapping from Σ to 2^U. Function g_C is a homomorphism which interprets each syntax type in C as a semantic type in Σ i.e. $g_C : C \rightarrow \Sigma$. Function g_L interprets each label as a semantic label in Λ, i.e., $g_L : L \rightarrow \Lambda$. Function g_O is a homomorphism $O \cup D \rightarrow U$. Function g_F interprets each k-ary object constructor as a mapping $U^k \rightarrow U$.

SEMANTICS OF TYPES: Given an interpretation I, the intended semantics of types is given by $\pi \circ g_C$ as follows:
(1) For the basic types, $\pi(g_C(integer)) = \mathbf{Z} \subseteq U$; $\pi(g_C(string)) = \mathbf{S} \subseteq U$;if $s = c(\{a_1, ..., a_n\})$, c is either *integer*, or *string*, then $\pi(g_C(s)) = \{g_O(a_1), ..., g_O(a_n)\} \subseteq \pi(g_C(c))$; if $s = integer(\{lb..rb\})$, then $\pi(g_C(s)) = \{x | g_O(lb) \leq x \leq g_O(rb)\} \subseteq \mathbf{Z}$.

(2) For a set type $\{s\}$, $\pi(g_C(\{s\})) = 2^{\pi(g_C(s))} \subseteq 2^U$.
(3) For a disjunctive type $s_1|...|s_n$, $\pi(g_C(s_1|...|s_n)) = \pi(g_C(s_1)) \cup ... \cup \pi(g_C(s_n))$.
(4) For an abstract object p with definition $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$. $\pi(g_C(p)) = \{x | g_L(l_i)(g_O(x)) = g_O(x_i) \in \pi(g_C(p_i)), 1 \leq i \leq n\}$. If $l_i = isa$, then for each $x \in \pi(g_C(p))$, $g_L(l_i)(x) = x \in \pi(g_C(p_i))$, so $g_L(l_i)(\pi(g_C(p)) = \pi(g_C(p)) \subseteq \pi(g_C(p_i))$.
(5) For built-in types, $\pi(g_C(all)) = U$; $\pi(g_C(none)) = \{\}$.

Obviously, under the semantics given above, the subtype relationship is interpreted as set inclusion under the mapping $\pi \circ g_C$, i.e. if $p \leq_C q$ then $\pi(g_C(p)) \subseteq \pi(g_C(q))$.

Proposition 1: Given a type system the types of which form a lattice under the subtype relationship, its interpretation based on the above semantics also forms a lattice based on set-inclusion relationship. □

VARIABLE ASSIGNMENT: A variable assignment, ν, is a ground substitution which assigns an element in U to a single-valued variable, a subset of U to a set-valued variable, a type in Σ to a type variable, and a label in Λ to a label variable. Besides, we extend it to non-variable elements in the usual way: if $d \in O \cup D$, then $\nu(d) = g_O(d)$; if $l \in L$, then $\nu(l) = g_L(l)$; if $c \in C$, then $\nu(c) = g_C(c)$; if $f \in F$, then $\nu(f) = g_F(f)$; and $\nu(f(..., X, ...)) = g_F(f)(..., \nu(X), ...)$.

SEMANTICS OF S-TERMS: Given an interpretation I and a variable assignment ν, function $M_{I,\nu}$ defines the semantics of S-terms as follows:
(1) For a type S-term P, $M_{I,\nu}(P) = true$ iff $\nu(P) \in \Sigma$.
(2) For a type S-term $t = P(L_1 \rightarrow P_1, ..., L_n \rightarrow P_n)$, $M_{I,\nu}(t) = true$ iff $\nu(P), \nu(P_1), ..., \nu(P_n) \in \Sigma$, $\nu(L_i) \in \Lambda$, and $\nu(L_i)(\pi(\nu(P)) \subseteq \nu(\pi(P_i)), 1 \leq i \leq n$.
(3) For a basic S-term $X : p$, $M_{I,\nu}(X : p) = true$ iff $\nu(X) \in \pi(g_C(p))$.
(4) For a basic S-term $t = X : p(l_1 \rightarrow X_1 : p_1, ..., l_n \rightarrow X_n : p_n)$, $M_{I,\nu}(t) = true$ iff $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$ is a true type S-term; $\nu(X) \in \pi(g_C(p)), \nu(X_i) \in \pi(g_C(p_i)), (1 \leq i \leq n)$ and $g_L(l_i)(\nu(X)) = \nu(X_i)$. If X_i is a set grouping variable and $X_i = \{Y_1, ..., Y_q\}$, then $\nu(Y_j) \in g_L(l_i)\nu(X), (1 \leq j \leq q)$.
(5) For an S-term $t = X : P(L_1 \rightarrow X_1 : P_1, ..., L_n \rightarrow X_n : P_n)$, $M_{I,\nu}(t) = true$ iff $P(L_1 \rightarrow P_1, ..., L_n \rightarrow P_n)$ is a true type S-term under the assignment $\nu(P), \nu(P_i), \nu(L_i), 1 \leq i \leq n$; $\nu(X) \in \pi(\nu(P)), \nu(X_i) \in \pi(\nu(P_i))$, and $\nu(L_i)(\nu(X)) = \nu(X_i), 1 \leq i \leq n$. If X_i is a set grouping variable and $X_i = \{Y_1, ..., Y_q\}$, then $\nu(Y_j) \in g_L(l_i)\nu(X), (1 \leq j \leq q)$.

SEMANTICS OF LITERALS: Given an interpretation I and a variable assignment ν, $M_{I,\nu}$ defines the semantics of literals other than S-terms as follows: $M_{I,\nu}(\psi_1 = \psi_2) = true$ iff $\nu(\psi_1) = \nu(\psi_2)$. $M_{I,\nu}(\psi_1 \neq \psi_2) = true$ iff $\nu(\psi_1) \neq \nu(\psi_2)$. $M_{I,\nu}(\psi_1 \leq \psi_2) = true$ iff $\nu(\psi_1) \leq \nu(\psi_2), \nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$. $M_{I,\nu}(\psi_1 \geq \psi_2) = true$ iff $\nu(\psi_1) \geq \nu(\psi_2), \nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$. $M_{I,\nu}(\psi_1 < q\psi_2) = true$

iff $\nu(\psi_1) < \nu(\psi_2), \nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$. $M_{I,\nu}(\psi_1 < \psi_2) = true$ iff $\nu(\psi_1) > \nu(\psi_2), \nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$. If ψ is an S-term, $M_{I,\nu}(\neg\psi) = true$, iff $M_{I,\nu}(\psi) = false$.

Cleary, for a closed an S-term ψ, its meaning is independent of a variable assignment, and we can simply write $M_I(\psi)$.

SATISFACTION: Given a program $P = \langle S, DB, R, Q\rangle$ and an interpretation I we define the notion of satisfaction(denoted by $\models_I$) as follows: If I obeys proposition 1, then we say the type system is satisfied by I and we write $\models_I S$. If $M_I(\psi) = true$ for each ground S-term ψ in the database DB, then the database is satisfied by I, and we write $\models_I DB$. If $\models_I S, DB$, ϕ is a literal and $M_I(\phi) = true$, then we say I satisfies ψ based on S, DB, and we write $S, DB \models_I \phi$. Let $r = p \Leftarrow p_1, ..., p_n$, if for every variable assignment ν, such that $S, DB \models_I \nu(p_i)$, for each i, and $S, DB \models_I \nu(p)$, then r is satisfied by I, and we write $S, DB \models_I r$. For the program P, if its type system, database, rules are satisfied by I, then the program is satisfied by I, and we write $\models_I P$. If $\models_I P$, ψ is a ground literal and $\models_I \psi$, than we say ψ logically implied by P, and we write $P \models_I \psi$.

PARTIAL ORDER OVER INTERPRETATIONS: An interpretation $I_1 = \langle U_1, \Sigma_1, \Lambda_1, \pi_1, g_{C_1}, g_{L_1}, g_{O_1}, g_{F_1},\rangle$ is a subset of an interpretation $I_2 = \langle U_2, \Sigma_2, \Lambda_2, \pi_2, g_{C_2}, g_{L_2}, g_{O_2}, g_{F_2},\rangle$ iff the following conditions hold: (1) $U_1 \subset U_2$. (2) $\Sigma_1 \subset \Sigma_2$. (3) $\Lambda_1 \subset \Lambda_2$. (4) $\pi_1(g_{C_1}(d)) \subset \pi_2(g_{C_2}(d))$, for every $d \in C$. (5) $g_{L_1} = g_{l_2}$. (6) $g_{O_1} = g_{O_2}$. (7) $g_{F_1} = g_{F_2}$.

MODELS: A model of a program P is an interpretation which satisfies P. A model M of P is minimal if no proper subset of M exists such that it is also a model of P.

ANSWERS: Given a program $P = \langle S, DB, R, Q\rangle$, and an interpretation I such that $\models_I P$, the set of answers to Q is the smallest set of variable assignments such that for each assignment ν, $P \models \nu(Q)$.

Propositions 2: If a program has a model, then it has a unique minimal model. □.

We define an operator T as follows: Given a set of rules R and a set I
$T_R(I) = \{\nu(p) | p \Leftarrow p_2, ..., p_n \in R, \exists\nu$ satisfying $p_i, ..., p_n\}$.
We define the powers of the operator T as follows:
$T \uparrow 0(I) = T(I)$
$T \uparrow n(I) = T(T \uparrow n-2(I)) \cup T \uparrow n-2(I), (n \geq 2)$
We define the least fixpoint of the program $P = \langle S, DB, R, Q\rangle$ to be the set of

objects $T_R \uparrow \omega(\phi)$ where ω is the first limit ordinal number.

Proposition 3: $T_R \uparrow (\phi)$ exists and is equivalent to the unique minimal model of a given program $P = \langle S, DB, R, Q \rangle$. □

We leave the discussion of this aspect as a full paper.

4 Conclusion

Deductive databases are torn by two opposing forces. On one side there are the stringent real-world requirements of actual databases. The requirements include efficient processing as well as the ability to express complex and subtle real relationships. On the other side are the simple and clear semantics of logic programming and its deductive power. The need of expressiveness have forced the deductive models away from their simple roots in logic programming.

In this paper, we present a higher-order logic which is based on semantic data models and extended logic term approaches. S-logic is capable of representing object identity, data abstraction and inheritance naturally, and reasoning about schema information and data uniformly. Although not presented in this paper, S-logic has a sound and complete resolution-based proof procedure based on its bottom-up computation feature. We will discuss this issue in another paper.

References

[1] Abiteboul, S., Grumbach, S. "COL: A Logic-Based Language for Complex Objects," Proc. Inter. Conf. on Extending Database Technology, Venice, Italy, 1988, pp 271-293.

[2] Abiteboul, S., Hull, R. "IFO: A Formal Semantic Database Model," ACM Trans. Database Systems, Vol. 12, No. 4, (Dec. 1987), pp 525-565. pp 271-293.

[3] Albano, A., Cardelli, L., and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," ACM Trans. Database Systems, Vol. 10, No. 2 (June 1985), pp 230-260.

[4] Borkin, S. A., Data Models: A Semantic Approach For Database Systems The MIT Press. 1980

[5] Brodie, M. L., and Ridjanovic, D., "On the Design and Specification of Database Transactions," in On Conceptual Modelling M. L. Brodie, J, Mylopoulos, and J. W. Schmidt Eds. Springer-Verlag, New York. 1984, pp. 276-232.

[6] Chen, W. and Warren, D.S., "C-Logic for Complex Objects," ACM PODS, 1989, pp 369-378.

[7] Gallaire, H., Minker, J. and Nicolas, J. M., "Logic and Databases: A Deductive Approach," ACM Computing Surveys, Vol. 16, No. 2 (June 1984), pp 153-186.

[8] Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Database Model.," ACM Trans. Database System, Vol. 6, No. 3 (Sept. 1981), pp 351-386.

[9] Hayes, P.J., "The Logic for Frames," in Frame Conception and Text Understanding, D. Metzing eds, Walter de Gruyter and Co.,1979, pp. 406-416.

[10] Hassan A. K. and Nasr R., "LOGIN: A Logic Programming Language with built-in Inheritance," Journal of Logic Programming, Vol. 3, No. 3 (Oct. 1986), pp 185-215.

[11] Hull, R. and King, R., "Semantic Database Modeling: Survey, Applications, and Research issues," ACM Computing Surveys, Vol. 19, No. 3 (Sept. 1987), pp 201-260.

[12] Kifer, M. and Lausen, G., "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme," Proc. ACM SIGMOD int. Conf. on Management of Data, 1989, pp 134-146.

[13] Kifer, M. and Wu, J., "A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited)," Proc. ACM PODS, 1989, pp 379-393.

[14] Krishnamurthy, R. and Naqvi, S., "Towards a Real Horn Clause Language," Proc. 14th VLDB Los Angeles, USA, 1988, pp 252-263.

[15] Liu, M. and Cleary, J.G., "Deductive Databases - Where to now," To appear, 1990.

[16] Lloyd, J.W., "Foundations of Logic Programming," , Springer Verlag, 2nd edition, 1987.

[17] Maier, D., "A Logic for Objects," Proc. the Workshop on Deductive Databases and Logic Programming, 1986

[18] Peckham, J. and Maryanski,F., "Semantic Database Models," ACM Computing Surveys, Vol. 20, No. 3 (Sept. 1988), pp 153-189.

[19] Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory," in On Conceptual Modelling. M. L. Brodie, J, Mylopoulos, and J. W. Schmidt Eds. Springer-Verlag, New York. 1984, pp. 19-48.

[20] Smith, J. M. and Smith D. C. P., "Database Abstractions: Aggregation and Generalization," ACM Trans. on Database Systems, Vol. 2, No. 2 (June. 1977), pp 105-133.

[21] Su, S. Y. W., "Modeling Integrated Manufacturing Data with SAM*," IEEE Computer Society, Vol. 19, No. 1 (Jan. 1986), pp 34-49.

[22] Tsur, S. and Zaniolo, C., "LDL: A Logic-Based Data-Language," Proc. 12th VLDB Kyoto, Japan, 1986, pp 33-41.