F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme

Michael Kifer*
Department of Computer Science
SUNY at Stony Brook, Stony Brook,
NY 11794, U.S.A.

Georg Lausen
Fakultät für Mathematik und Informatik
Universität Mannheim, D-6800 Mannheim,
West Germany

Abstract

We propose a database logic which accounts in a clean declarative fashion for most of the "object-oriented" features such as object identity, complex objects, inheritance, methods, etc. Furthermore, database schema is part of the object language, which allows the user to browse schema and data using the same declarative formalism. The proposed logic has a formal semantics and a sound and complete resolution-based proof procedure, which makes it also computationally attractive.

1 Introduction

Object-oriented approach to databases has generated considerable interest in the community in the past few years. Although the very term "object-oriented" is loosely defined, a number of concepts have been identified as the most salient features of that approach. According to [4,39,9,43] and a number of other surveys, these concepts are: complex objects, object identity, methods, and inheritance.

One of the reasons for the interest in object-oriented database languages is that they show promise in overcoming the, so called, *impedance mismatch* [28,9] between programming and database languages. Meanwhile, a different, *deductive*, approach has gained enormous popularity both in databases and programming languages. Since logic programming can be used as a full-fledged computational formalism as well as a database specification language, proponents of the deductive approach have also argued that it is capable of overcoming the aforesaid mismatch problem. However, in their present form both approaches have shortcomings. The main problem with the object-oriented approach is the lack of formal semantics which is traditionally considered to be very important in databases. On the other hand, deductive databases normally use flat data model and do not support object identity and data abstraction. It therefore can be expected that combining the two paradigms will yield significant benefits.

A number of attempts to combine the two approaches [1,2,5,6,7,13,20,23,22,27,37] have been reported in the literature, but, in our opinion, none of them succeeds in meeting all of the above goals. These approaches either do not support object identity, or restrict the kinds of complex objects and queries one can use, or do not support inheritance, limit deduction, etc.

In this paper we propose a formalism, called *Frame Logic* (abbr. F-logic), which is a full-fledged logic achieving *all* of the goals listed above and, in addition, is suitable for defining and manipulating database schema. Our work has also implications for the frame-based languages in AI [16,33], which are essentially scaled down versions of complex objects with identity, inheritance, and deduction. It is this connection that the name "F-logic" was derived from.

To reason about inheritance and schema we need capabilities of higher-order logics. However, these are usually much too powerful to be useful for our purposes. A number of researchers have suggested that the "useful" parts of higher-order logics can be given a first-order semantics by encoding them in predicate calculus [18,32]. However, this provides only an indirect semantics and is not true to the spirit of object-oriented programming and frame-based languages in AI. In contrast, we propose a logic which has an appearance of a higher-order logic, but, unlike it, is tractable and has a natural di-

*Supported in part by the NSF grant DCR-8603676; Work partially performed while visiting the University of Mannheim, W. Germany

rect first-order semantics. More precisely, according to the classification of [12], F-logic has a higher-order syntax and a first-order semantics. A sound and complete proof procedure for F-logic will be described in the full paper.

This work is an extension of [20], which in turn was based on Maier's O-logic [27]. In [20] a distinction between objects, classes, and relationships is maintained, which is the main reason why inheritance, methods, and schema cannot be reasoned about in these formalisms.

This paper is organized as follows. In Section 2 we present the syntax and semantics of F-logic, and Section 3 presents Skolem and Herbrand theorems; in Section 4 we illustrate the higher-order capabilities of F-logic.

2 Syntax and Semantics of F-logic

In this section we describe the proposed logic by means of an example, followed by a formal account of the syntax and semantics.

2.1 F-logic by Example

In developing F-logic we were motivated by the desire to capture in a logically clean way a number of scenarios whose most salient common features are depicted by way of an example in Figures 1 and 2. Figure 1 shows part of the IS-A hierarchy. All classes and individual objects are taken from the same domain and are organized in a lattice. It asserts that a *faculty* and an *assistant* are employees, *student* and *empl* are persons, "*Mary*" and "*CS*" are strings, *mary* is a *faculty*, while *sally* is a *student*. Notice that the lattice is ordered with respect to the "definedness" ordering of denotational semantics, or, equivalently, with respect to the relative "knowledge content" [17]. For instance, a statement *assistant* : *john* (john is an assistant) is more informative than *empl* : *john* because every assistant is an employee, but not vice versa. Therefore, *assistant* has more knowledge content and is located above *empl*. In general, a class is always located below each of its instances. Furthermore, note that classes are *reified* by being placed in the same domain with their instances. Thus the same object (e.g. *empl*) can be viewed as an instance of its superclass (*person*) and, at the same time, as a class of all objects located above it in the lattice.

Figure 2 presents facts about employees, students, etc. The first clause says that object *bob* is a faculty whose name is "Bob" (notice: *bob* is an id of the object, while "Bob" is a string representing *bob*'s name). Additionally, this clause states that *bob* works in the department cs_1, the department name is "CS" and the manager is an employee denoted by the constant *phil*. Clauses (2) through (4) present similar information about *mary*, *john*, and *sally*. Note that the *friends* attribute in *mary*'s record is *set-valued*, which is syntactically expressed by means of the set constructor { }. Our syntax is that from [27,20] with some embellishments from [13].

Clauses (5), (6), and (7) provide general information about classes *faculty*, *students*, and *empl*, such as that *faculty* are normally supervised by *faculty* and are middle aged, *students* are normally *young*, etc. Clause (8) is a rule stating that employee's supervisor is the manager of the department the employee works in.

Clause (9) is a rule defining a *method*: for each person, X, the method *children* is a function which for a given argument, Y, returns a set of persons containing all the children common to X and Y. Notice that the term $children(Y)$ appears in the same clause (9) in two different roles: in the head of (9) it is in the "attribute" position, where it denotes the aforesaid method, and it is in the "object" position in the body of (9), where it denotes the id of the object whose content is the set of children of Y. Thus, any ground instance of this term, say $children(mary)$, has two different roles: it denotes the object representing all *mary*'s children and also a function which, for each person, returns a (possibly empty) set of all children *mary* has with that person. Thus, in F-logic, the same syntactic term may denote different things, depending on the context it appears in. This feature allows one to pose meta-queries about the objects, such as "retrieve a set of all objects which represent the labels defined for a certain object".

Two applications of method *children* are demonstrated by queries (10) and (11). Query (10) asks for the father of *mary*'s child *sally*, and (11) requests all children *mary* has with *phil*. Query (12) is requesting information about all middle aged employees working for the "CS" departments. Particularly, for each such employee, it is requesting the supervisor, age, and the department. The expected answer to this query is

(13) $bob[supervisor \rightarrow \top,\ age \rightarrow 40,\ works \rightarrow cs_1]$

(14) $mary[supervisor \rightarrow faculty,\ age \rightarrow 30,\ works \rightarrow cs_2]$

To see that *bob* is an answer, first observe that, by (1), *bob* is working for the "*CS*" department and that *bob* has age 40, which is a value belonging to class *midaged* (see Figure 1). The value for *supervisor* is partly inferred from the rule (8) and partly *inherited* from the general description (5) of class *faculty*. Indeed, the rule suggests that *bob*'s supervisor should be *phil*, while on the other hand, being a faculty, *bob*'s supervisor has

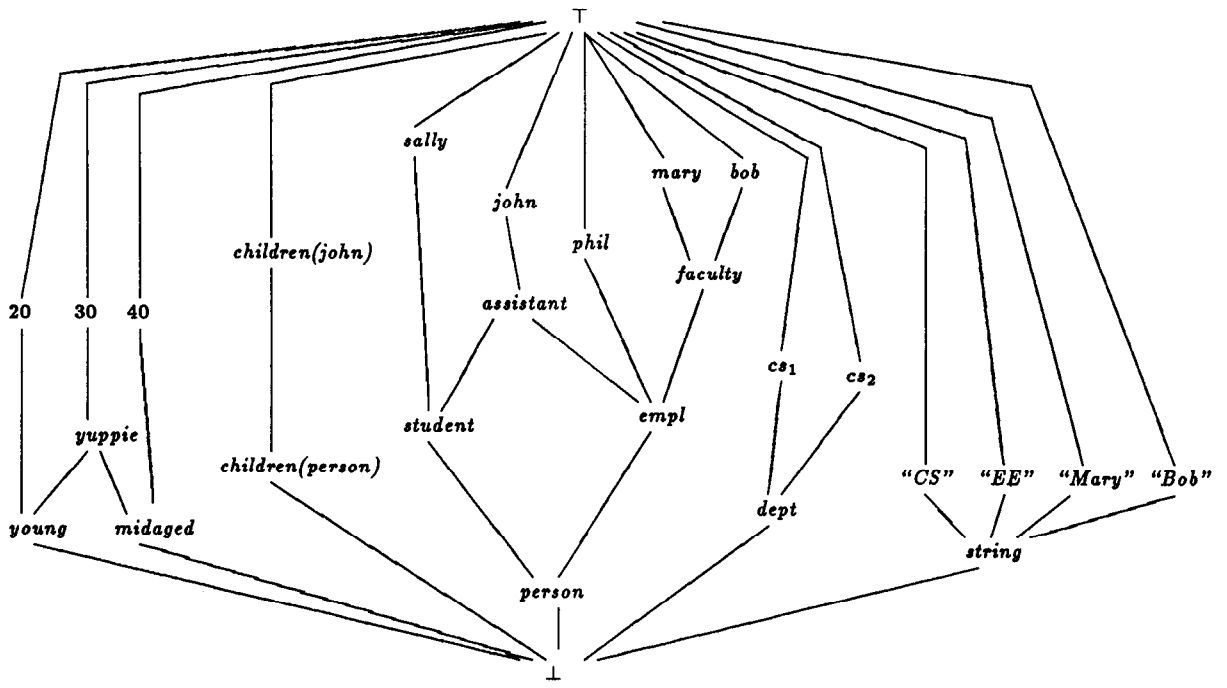

Figure 1: Part of the IS-A hierarchy

Facts:

(1) $faculty : bob[name \rightarrow \text{“}Bob\text{”}, age \rightarrow 40, works \rightarrow dept : cs_1[dname \rightarrow \text{“}CS\text{”}, mngr \rightarrow empl : phil]\ ]$

(2) $faculty : mary[name \rightarrow \text{“}Mary\text{”}, age \rightarrow 30, friends \rightarrow \{bob, sally\}, works \rightarrow dept : cs_2[dname \rightarrow \text{“}CS\text{”}]\ ]$

(3) $assistant : john[name \rightarrow \text{“}John\text{”}, works \rightarrow cs_1[dname \rightarrow \text{“}CS\text{”}]]$

(4) $student : sally[age \rightarrow midaged]$

General Class Information:

(5) $faculty[supervisor \rightarrow faculty, age \rightarrow midaged]$

(6) $student[age \rightarrow young]$

(7) $empl[supervisor \rightarrow empl]$

Rules:

(8) $E[supervisor \rightarrow M] \Leftarrow empl : E[works \rightarrow dept : D[mngr \rightarrow empl : M]]$

(9) $X[children(Y) \rightarrow \{Z\}] \Leftarrow person : Y[children\_obj \rightarrow children(Y)[members \rightarrow \{person : Z\}]],$
$person : X[children\_obj \rightarrow children(X)[members \rightarrow \{person : Z\}]]$

Queries:

(10) $mary[children(Y) \rightarrow \{sally\}]?$

(11) $mary[children(phil) \rightarrow \{Z\}]?$

(12) $empl : X[supervisor \rightarrow Y, age \rightarrow midaged : Z, works \rightarrow D[dname \rightarrow \text{“}CS\text{”}]]?$

Figure 2: A Sample Database

to be in the class *faculty*. However, looking at Figure 1, we observe that *phil* is not an instance of *faculty*. The semantics of F-logic then suggests that the value of *supervisor* for *bob* should be $lub(phil, faculty)$ which is $\top$ (see Figure 1). That is, the information about *phil*'s supervisor is inconsistent.

Clause (14) is retrieved because *mary* is in the *yuppie* age, and thus is also a middle aged person. On the other hand, the object denoted by constant *john* is not retrieved because *john*, being a student, inherits *young* from (6), and according to Figure 1, *young* is not an instance of *midaged*. However, if the restriction $midaged : Z$ in query (10) were dropped, then the following clause will be retrieved as well:

(15) $john[supervisor \rightarrow phil,\ age \rightarrow young,\ works \rightarrow cs_1]$

The *supervisor* of *john* is inferred by rule (8), since *john* is working in cs_1, and, due to clause (1), department cs_1 is managed by *phil*. An interesting point here is that, unlike *bob*, *john* is not a *faculty* and therefore is not subject to the restriction imposed by clause (5) that his supervisor must be a faculty. Therefore, no contradiction with *john*'s supervisor being *phil* arises!

One additional important observation is that disagreements in attribute values need not always result in inconsistency as in the *bob*'s case. For instance, *sally* is *midaged* according to (4), but, being a student, she inherits *young* from (6). In F-logic this logically entails $sally[age \rightarrow lub(young, midaged)]$, which is *yuppie* according to Figure 1.

The example of Figures 1 and 2 alludes to the point that in F-logic we are dealing with higher-order concepts. For instance, in Figure 2, attribute *friends* is essentially a set-valued function which for each person returns a set of persons. Similarly, the user can think of objects of the lattice displayed in Figure 1 as sets ordered by subset relation, and use higher-order intuitions to develop programs and models. However, the underlying semantics formally remains first-order (see [12] for details), which allows us to overcome the difficulties normally associated with truly higher-order theories.

The first-order behaviour with respect to sets is achieved by introducing typing so that single-valued and set-valued functions do not mix. Furthermore, formally, F-logic has only *flat* sets, but we can model arbitrarily deeply nested sets in a very natural way. F-logic shares this behaviour with respect to sets with other two related formalisms [13,20] and more discussion on that can be found in [20]. The first-order behaviour with respect to the class/subclass hierarchy is achieved by defining a lattice structure on the set of objects and classes which allows us to model sets in quite an authentic, yet inexpensive, way.

2.2 Syntax

The alphabet of F-logic consists of (1) a set $\mathcal{O}$ of *basic* objects, (2) a set of *object constructors* $\mathcal{F}$, (3) an infinite set of *variables*, $\mathcal{V}$, and (4) usual logic connectives and quantifiers $\vee$, $\wedge$, $\forall$, $\exists$, $\neg$, $\Longleftarrow$, etc.

The "objects" in F-logic are either complex objects in the usual sense, or classes of objects. Unlike other approaches which treat classes and instances as inherently different entities, we do not emphasize the distinction. For instance the object *student* can be viewed as representing the class of students (each individual student being its instance) and at the same time as an instance of its superclass represented by the object *person*. Thus, in F-logic classes have the same meaning as in frame-based formalisms (e.g. [16,33]) in the sense that a class is viewed as an instance (rather than a subset) of a superclass. This feature is largely responsible for the fact that inheritance is naturally built into the semantics of F-logic, contrasting it to algorithmically defined inheritance in other approaches.

Basic objects (elements of $\mathcal{O}$) are the constants of F-logic. Object constructors (elements of $\mathcal{F}$) are logically function symbols over $\mathcal{O}$ of arity ≥ 1; they are used to construct new objects. Although members of $\mathcal{O}$ can be viewed as 0-ary object constructors, it is convenient to consider them separately, and we assume that $\mathcal{O}$ and $\mathcal{F}$ are disjoint. An *id-term* is a term composed of function symbols (i.e. object constructors), constants (i.e. basic objects), and variables in the usual way. The set of all ground (i.e. variable-free) id-terms is denoted by $\mathcal{O}^*$. In the following we will show that $\mathcal{O}^*$ essentially plays the role of Herbrand universe of F-logic. Conceptually id-terms should be viewed as objects themselves or as object abstractions which are commonly referred to as *object identity* [19]. Objects represented by id-terms other than constants (non-basic objects) are best perceived as being constructed from simpler objects. For instance, the object $university(state)$ can be taken as a class representing universities, while the object $university(nys)$ is the class of universities in New York State (assuming *nys* is an instance of class *state*).

In O-logic [27,20], which is a predecessor of F-logic, there was a distinction between objects and *labels* and the latter were viewed exclusively as binary relations among objects. This distinction made it difficult to reason about entities and relationships in a uniform framework; it can be traced to the Entity-Relationship model where entities (objects) and relationships (labels) constitute two disjoint categories.

In F-logic, every object can be viewed as an entity or a relationship depending on the situation (namely,

on its syntactic position in a formula). In its role as a label, each object has a *type*: a label can be either *single-valued* (sometimes also called *functional*) or *set-valued*. Accordingly, we partition $\mathcal{O}^*$ into a pair of disjoint sets $\mathcal{L}_{\#}$ (for objects typed as functional labels) and $\mathcal{L}_*$ (for objects typed as set-valued labels).

We require the aforesaid partition to be *congruent*: if $t_1, s_1, \ldots, t_n, s_n \in \mathcal{O}^*$, where for each i, t_i and s_i are in the same partition, and $f \in \mathcal{F}$ is an n-ary object constructor, then $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ belong to the same element of the partition ($\mathcal{L}_{\#}$ or $\mathcal{L}_*$).

This restriction is needed to be able to specify types on $\mathcal{O}^*$ effectively and check them efficiently. Indeed, under the congruence assumption, assigning types to elements of $\mathcal{O}^*$ involves the following: (1) assigning a type to each element of $\mathcal{O}$; (2) specifying the type of $f(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ for every n-ary function symbol $f \in \mathcal{F}$ and each n-tuple $< \mathcal{L}_1, \ldots, \mathcal{L}_n >$, where each $\mathcal{L}_j$ is either $\mathcal{L}_{\#}$ or $\mathcal{L}_*$. Since the cardinality of $\mathcal{O}$ and $\mathcal{F}$ is finite in practical cases, this procedure is effective. To check the type of a term $t \in \mathcal{O}^*$, one needs to "evaluate" this term by substituting types for subterms, starting with constants. This procedure is linear in the size of the term. Furthermore, we suspect that, in practice, one needs to assign only a single type to the range of each function symbol (regardless of the argument types), which then makes type definition linear in the sizes of $\mathcal{O}$ and $\mathcal{F}$, and type checking can be done in constant time by checking the range type of the outermost function symbol.

The language of F-logic consists of a set of formulae constructed out of the alphabet symbols. Formulae are built from atomic formulae by means of the usual connectives $\neg$, $\vee$, and $\wedge$, and quantifiers $\exists$ and $\forall$. Atomic formulae are, so called, *F-terms* (F-logic terms). Later we will see that the id-terms introduced earlier can be viewed as a special case of F-terms.

For convenience, we will use names starting with lower-case letters to denote ground terms, and names starting with capital letters to denote possibly non-ground terms. An *F-term* (cf. [27,20]) is

(1) a *simple* F-term, $P : Q$, where P and Q are id-terms, or

(2) a *complex* F-term, $P : Q[Flab_1 \rightarrow T_1, \ldots, Flab_m \rightarrow T_m, Slab_1 \rightarrow \{S_{1,1}, \ldots, S_{k_1,1}\}, \ldots, Slab_l \rightarrow \{S_{1,l}, \ldots, S_{k_l,l}\}]$. Here P, Q are id-terms; $Flab_i$ and $Slab_j$ are also id-terms, but we chose to name them differently to indicate that their syntactic position within the F-term emphasizes their role as labels. Furthermore, the appearance of the set construct, { }, indicates that the respective id-terms $Slab_1$, ..., $Slab_l$ are supposed to be typed as set-valued labels; the rest of the labels, $Flab_1$, ..., $Flab_m$, are functional. The order of labels in an F-term is immaterial. Finally, in the above, T_i and $S_{n,j}$ denote F-terms.

It is worth noting here that, since we chose a sortless setting for F-logic, typing of nonground labels is virtually impossible. Consequently, say, $t = a[X \rightarrow b]$ is considered to be a syntactically correct term, even though X may be bound to ground id-terms typed as functional as well as set-valued labels. However, the semantics is set up in such a way that, since the syntax of t calls for a functional label, t will be always false whenever X is universally quantified. Furthermore, even the constructs such as $a[lab \rightarrow c, lab \rightarrow \{d\}]$ are syntactically correct F-terms, despite the fact that *lab* must be a functional label due to one part of the term, and a set-valued label due to the other. However, according to the F-logic semantics, this term is *unsatisfiable*.

Intuitively, the F-term (2) above is a statement about an object, Q, asserting that it is an instance of the class P and has properties specified by the labels. Thus, when no labels are specified we can omit the brackets, thereby reducing a complex F-term, $P : Q[\,]$, to the simple F-term $P : Q$.

To account for the higher-order features of frame-based and object-oriented formalisms without incurring the overhead of the higher-order predicate calculus, we reify classes and model class membership by means of a lattice ordering instead of the true set-theoretic membership. Formally, we assume that the elements of $\mathcal{O}^*$ are organized in a lattice[1] by means of the ordering $\prec_{\mathcal{O}}$. As usual, $\preceq_{\mathcal{O}}$ stands for $\prec_{\mathcal{O}}$ or $=$. We distinguish in $\mathcal{O}^*$ the maximal element, $\top$, and the minimal element, $\bot$. The maximal element, $\top$, can be viewed as a "meaningless" object which represents a class with no instances; $\bot$ can be perceived as the object representing the biggest class (or as the "unknown" object).

The lattice on $\mathcal{O}^*$ is a *static* part of the language[2], and can be viewed as part of schema specification: the lattice represents the transitive closure of the "subclass-of" and the "instance-of" relationships among classes, so that $p \prec_{\mathcal{O}} q$ (e.g. *person* $\prec_{\mathcal{O}}$ *student* or *student* $\prec_{\mathcal{O}}$ *john*) means that q is a (possibly indirect) subclass or instance of p. As mentioned earlier, we do not distinguish between individual objects and classes: any object, d, is treated as a class whose extension contains all objects/classes found above d in the lattice. Therefore, any element $p \in \mathcal{O}^*$ may appear in an F-term in the "instance position", $q : p[...]$, and in

[1] When objects are considered in their role as labels, the lattice structure on labels is ignored.

[2] This assumption is needed for F-logic unification to be decidable.

the "class position", $p : r[...]$. This gives F-logic a "feel" of a higher-order language, although its semantics is essentially first-order [12]. Accordingly, we will use the terms "instance" and "class" to refer to the same object, p, depending on whether we want to emphasize p in its role as a class or as an instance in its respective superclass. Part of a sample lattice structure on $\mathcal{O}^*$ is depicted in Figure 1. We impose the following *monotonicity* restriction on the lattice $\mathcal{O}^*$:

if $t_1 \preceq_{\mathcal{O}} s_1$, ..., $t_n \preceq_{\mathcal{O}} s_n$ then
$f(t_1,...,t_n) \preceq_{\mathcal{O}} f(s_1,...,s_n)$;

This simply means that object constructors are monotonic functions on the lattice: for instance, if $person \prec_{\mathcal{O}} john$, i.e., John is a person, then $car(person) \prec_{\mathcal{O}} car(john)$ meaning that John's cars belong to the class of cars owned by persons.

Monotonicity is necessary for the resolution procedure to be complete, which will be discussed in the full paper. Apart from that, this ensures that the lattice structure on $\mathcal{O}^*$ can be given effectively as part of schema specifications and that the "instance-of" relationship can be verified efficiently. Indeed, in practical cases, $\mathcal{O}$ and $\mathcal{F}$ are finite and $\prec_{\mathcal{O}}$ can be first specified on $\mathcal{O}$. To complete the specification of the lattice order, one only needs to provide typing information for each of the finite number of object constructors by specifying the classes for the range and the arguments. For instance, $cons : edge \times path \rightarrow path$ (meaning $path \prec_{\mathcal{O}} cons(edge, path)$) is an example of such typing information. Additional typing for function symbols can be automatically inferred using the well known type inference techniques (e.g. see [10]). Verification of whether $t \preceq_{\mathcal{O}} s$ holds for a pair of ground id-terms $t, s \in \mathcal{O}^*$ can be done in a way resembling the usual unification algorithm and will be discussed in the full paper.

Every F-term is also an (atomic) *F-formula*. F-formulae are constructed from other (simpler) F-formulae by means of logical connectives and quantifiers.

To simplify the notation we assume the following convention: if a single-valued label, Lab, in an F-term is omitted then the intention is $Lab \rightarrow \bot : \bot$; similarly, if a set-valued label, Lab' is omitted then we assume $Lab' \rightarrow \{\}$. Furthermore, if a class specification is omitted then $\bot$ is assumed. Thus, for instance, $john[name \rightarrow string : "john"]$ and $\bot : john[name \rightarrow string : "john", pay \rightarrow \bot : \bot, children \rightarrow \{\}]$ are considered to be the same term. This convention allows us to view id-terms as a special case of F-terms by identifying P and $\bot : P[\,]$.

2.3 Semantics

Before presenting the semantics, we will need to introduce an ordering on the powerdomain of a lattice. This ordering was also used in [5] and is sometimes called *Hoare's ordering* [8]. Given a lattice U with the ordering $\preceq_U$ and maximal and minimal elements $\top_U$ and $\bot_U$, the preorder $\sqsubseteq_U$ on the powerset 2^U is defined as follows: for any pair of sets $X, Y \subseteq U$, we write $X \sqsubseteq_U Y$ iff for every element $x \in X$ there is $y \in Y$ such that $x \preceq_U y$.

The preorder $\sqsubseteq_U$ on 2^U is not an order, since it is cyclic. For instance $\{a\} \sqsubseteq_U \{a, \bot_U\} \sqsubseteq_U \{a\}$ and $\{\top_U\} \sqsubseteq_U U \sqsubseteq_U \{\top_U\}$. However, 2^U can be considered a lattice modulo the equivalence relation $\approx_U$, where $X \approx_U Y$ if and only if $X \sqsubseteq_U Y$ and $Y \sqsubseteq_U X$. The maximal and minimal elements in this lattice are the equivalence classes of $\{\}$ (the empty set) and $\{\top_U\}$, respectively. To simplify the language, we will often talk about the lattice structure on the powerset of U disregarding the aforesaid subtlety.

Similarly, given a pair of lattices, U and V, we can define a lattice structure on the set of mappings $U \rightarrow V$, denoted $Map(U,V)$, as follows: $f \preceq_{Map(U,V)} g$ if for every $u \in U$, $f(u) \preceq_V g(u)$. Two kinds of lattice mappings, monotonic (denoted $Mon(U,V)$) and homomorphic[3] (denoted $Hom(U,V)$) are of particular importance. Clearly, $Hom(U,V) \subseteq Mon(U,V)$.

Semantics of F-logic can now be defined as follows. Given a language of F-logic, its interpretation, I, is a tuple $< U, g_{\mathcal{O}}, g_{\mathcal{F}}, \jmath_{\#}, \jmath_{*} >$. Here U is a universe of all objects which is required to have a lattice structure with $\bot_U$ and $\top_U$ being the smallest and the largest elements, respectively, and with the lattice ordering $\preceq_U$; U is partitioned into a pair of subsets $U_{\#}$ and U_{*} to account for the types of elements of $\mathcal{O}^*$. It is useful to think of the elements of $\mathcal{O}^*$ as the *names* of objects, while the elements of U are best thought of as the *objects themselves* in the *possible world* I.

The homomorphism $g_{\mathcal{O}} : \mathcal{O}^* \rightarrow U$ is interpreting objects of $\mathcal{O}^*$ by elements of U, so that $g_{\mathcal{O}}(\mathcal{L}_{\#}) \subseteq U_{\#}$ and $g_{\mathcal{O}}(\mathcal{L}_{*}) \subseteq U_{*}$. The mapping $g_{\mathcal{F}} : \mathcal{F} \rightarrow Mon(U^k, U)$ interprets each k-ary object constructor, $f \in \mathcal{F}$, by a monotonic mapping $U^k \rightarrow U$. Additionally, $g_{\mathcal{O}}$ and $g_{\mathcal{F}}$ are related as follows: if $t = f(s_1,...,s_n) \in \mathcal{O}^*$ then $g_{\mathcal{O}}(t) = g_{\mathcal{F}}(f)(g_{\mathcal{O}}(s_1),...,g_{\mathcal{O}}(s_n))$.

The reader may notice that constants and function symbols are interpreted essentially the same way as in predicate calculus. The only difference is that the set of ground terms, $\mathcal{O}^*$, and the domain, U, now have lattice structures which must be accounted for.

In their role as labels, objects are interpreted

[3] i.e. the ones preserving *lub* and *glb*.

by associating appropriate mappings to each element of U using functions $\jmath_{\#}$ and $\jmath_{*}$. More specifically, $\jmath_{\#} : U_{\#} \rightarrow Mon(U,U)$ associates a monotonic mapping $U \rightarrow U$ with each element of $U_{\#}$ and $\jmath_{*} : U_{*} \rightarrow Mon(U, 2^U)$ associates monotonic mappings $U \rightarrow 2^U$ with elements of U_{*}. Notice that $\jmath_{\#}$ and $\jmath_{*}$ ignore the ordering $\prec_U$ induced on $U_{\#}$ and U_{*} by U.

Thus, set-valued labels are interpreted by monotonic set-valued functions, while functional labels become monotonic single-valued functions. Notice that these functions are associated with the elements of U, not $\mathcal{O}^*$, because, as noted earlier, $\mathcal{O}^*$ is, strictly speaking, only a set of object names; these are interpreted by the "real" objects in a possible world, I, and it should be the objects, not their names, who can assume roles (of labels) [4].

A *variable assignment*, ν, is a mapping from variables, $\mathcal{V}$, to the domain U. We extend it to id-terms in the usual way: $\nu(d) = g_{\mathcal{O}}(d)$ if $d \in \mathcal{O}$ and, recursively, $\nu(f(..., T, ...)) = g_{\mathcal{F}}(f)(..., \nu(T), ...)$. To simplify the notation, we will also extend variable assignments to F-terms as follows: $\nu(P : Q[...]) = \nu(Q)$.

Let I be an interpretation and ν a variable assignment. The *meaning* in I under ν of an F-term T, denoted $\mathcal{M}_{I,\nu}(T)$, is a statement about the existence (true) or nonexistence (false) in I of an object $\nu(T)$ with the properties specified in $\nu(T)$. Consider an F-term, $T = P : Q[..., Flab_i \rightarrow T_i, ..., Slab_j \rightarrow \{S_1, ..., S_m\}, ...]$, where P, Q are id-terms in their role as objects, $Flab_i$, $Slab_j$ are id-terms in their role as functional and set-valued labels, respectively, and T_i, S_k are F-terms. Then $\mathcal{M}_{I,\nu}(T) = true$ iff the following conditions hold:

(1) $\nu(P) \preceq_U \nu(Q)$;

(2) for each id-term $Flab_i$ (intended as a functional label),
- $\nu(Flab_i) \in U_{\#}$,
- $\nu(T_i) \preceq_U \jmath_{\#}(\nu(Flab_i))(\nu(Q))$, and
- $\mathcal{M}_{I,\nu}(T_i) = true$;

(3) for each id-term $Slab_j$ (intended as a set-valued label),
- $\nu(Slab_j) \in U_{*}$,
- $\{\nu(S_1), ..., \nu(S_m)\} \sqsubseteq_U \jmath_{*}(\nu(Slab_j))(\nu(Q))$, and
- $\mathcal{M}_{I,\nu}(S_k) = true$ for $k = 1, ..., m$.

Here (1) simply says that the object $\nu(Q)$ must be in the class $\nu(P)$ in the possible world I. In (2) and (3), the first condition says that id-terms representing the labels must be appropriately typed; the second condition says that for an F-term, T, to be satisfied by a possible world, I, w.r.t. ν, that world must have at least as much information about the object denoted by $\nu(T)$ as the amount of information asserted by T. Finally, the third condition in (2) and (3) simply says that the properties of Q asserted by T (i.e., T_i, S_j, etc.) must also be true in I w.r.t. ν.

Notice that according to these definitions, $\mathcal{M}_{I,\nu}(\top) = \mathcal{M}_{I,\nu}(\bot) = \mathcal{M}_{I,\nu}(d) = true$ and $\mathcal{M}_{I,\nu}(\bot : d) = \mathcal{M}_{I,\nu}(d : \top) = true$ for every $d \in \mathcal{O}^*$. Similarly, if $\bot \prec_{\mathcal{O}} d \prec_{\mathcal{O}} \top$ then $\mathcal{M}_{I,\nu}(d : \bot) = \mathcal{M}_{I,\nu}(\top : d) = false$.

Meaning of the formulae $\phi \vee \psi$, $\phi \wedge \psi$, and $\neg\phi$ is defined in the standard way: $\mathcal{M}_{I,\nu}(\phi \vee \psi) = true$ (resp., $\mathcal{M}_{I,\nu}(\phi \wedge \psi)$, resp. $\mathcal{M}_{I,\nu}(\neg\phi)$) iff $\mathcal{M}_{I,\nu}(\phi) = true \vee \mathcal{M}_{I,\nu}(\psi) = true$ (resp. $\mathcal{M}_{I,\nu}(\phi) = true \wedge \mathcal{M}_{I,\nu}(\psi) = true$, resp. $\mathcal{M}_{I,\nu}(\phi) \neq true$). The meaning of quantifiers is also quite standard: $\mathcal{M}_{I,\nu}(\psi) = true$, where $\psi = (\forall X)\phi$ (resp. $\psi = (\exists X)\phi$) if for every (resp. some) μ which agrees with ν everywhere, except possibly on X, $\mathcal{M}_{I,\mu}(\psi) = true$.

Clearly, for a closed formula, ψ, its meaning is independent of a variable assignment, and we can simply write $\mathcal{M}_I(\psi)$. An interpretation I is a *model* of ψ if $\mathcal{M}_I(\psi) = true$.

As an aside, other orderings on powersets over lattices are possible. For instance, according to the *Smyth's ordering* [8], $X \sqsubseteq_U^{Smyth} Y$ iff for every element $y \in Y$ there exists $x \in X$ such that $x \preceq_U y$. Presumably, we could use this ordering instead of Hoare's in our semantics. This would allow us to enforce typing constraints on elements of sets the same way as we do it for functional labels (see Section 4.2). However, switching to Smyth's ordering would permit certain unnatural inferences, such as: from $a[lab \rightarrow \{b\}]$ infer $a[lab \rightarrow \{b, c\}]$ for any c. Ideally, we would like to use the so called Egli-Milner's order, which is Hoare's and Smyth's orderings combined. We could then benefit both from the right semantics of sets achieved through Hoare's ordering, and from type enforcement which Smyth's order has to offer. Unfortunately, in order to be able to group elements into sets under Egli-Milner's ordering, we would have to change F-logic syntax by introducing variables over sets. Particularly, instead of being first-order, as in O-logic [20], set grouping will become second-order, as in LDL [6], which makes handling of sets an expensive proposition.

2.4 Databases and Queries

A *database* is a set of formulae. We distinguish between the *extensional* part of a database (the set of F-terms) and its *intensional* part (the set of formulae "more complex" than F-terms). If S is a set of formulae

[4] There is also a technical reason for that, which becomes apparent when one tries to define formula satisfaction w.r.t. a variable assignment.

and ϕ a formula, we write $\mathbf{S} \models \phi$ (ϕ is logically *implied* or *entailed* by $\mathbf{S}$) iff ϕ is true in every model of $\mathbf{S}$.

Given a language $\mathbf{L}$ with a set of variables $\mathcal{V}$ and a set of basic objects $\mathcal{O}$, a *substitution* is a mapping $\sigma : \mathcal{V} \rightarrow \{\text{id}-\text{terms of } \mathbf{L}\}$ which is identity everywhere outside some finite set $dom(\sigma) \subseteq \mathcal{V}$, called the *domain* of σ. It is extended to id-terms by letting σ to commute with object constructors and, recursively, to F-terms so that $\sigma(P : Q[\ldots, Flab \rightarrow T, \ldots, Slab \rightarrow \{\ldots, S, \ldots\}] = \sigma(P) : \sigma(Q)[\ldots, \sigma(Flab) \rightarrow \sigma(T), \ldots, \sigma(Slab) \rightarrow \{\ldots, \sigma(S), \ldots\}]$. Finally, substitutions are extended to F-formulae by letting them commute with logical connectives. A substitution is *ground* if $\sigma(X) \in \mathcal{O}^*$ for each $X \in dom(\sigma)$. Given a substitution σ and a formula ϕ, $\sigma(\phi)$ is called an *instance* of ϕ. It is a *ground* instance if it contains no variables.

A *query* is a statement of the form $Q?$, where Q is an F-term. The set of *answers* to $Q?$ w.r.t. a database $\mathbf{D}$ is the smallest set of ground F-terms which is (1) closed under $\models$ and (2) contains all instances of Q logically entailed ($\models$) by $\mathbf{D}$.

3 Skolemization and the Herbrand Theorem

Skolemization procedure in F-logic is not any different from that of predicate calculus. As in predicate calculus, we have the following result.

Theorem 1 (cf. the Skolem Theorem) *Let $\mathbf{D}$ be a set of F-formulae and ϕ an F-formula. Let $\mathbf{D}'$, ϕ' denote some skolemization of $\mathbf{D}$ and ϕ, respectively. Then $\mathbf{D} \cup \{\neg\phi\}$ is unsatisfiable (has no model) iff so is $\mathbf{D}' \cup \{\neg\phi'\}$.*

Given a language L of F-logic, its *Herbrand universe* is $\mathcal{O}^*$. A *Herbrand interpretation*, H, is an interpretation whose domain is $\mathcal{O}^*$ together with the lattice ordering $\prec_{\mathcal{O}}$ originally supplied with $\mathcal{O}^*$ (in L). Herbrand interpretations interpret objects and object constructors in the usual way: for $d \in \mathcal{O}$ and $f \in \mathcal{F}$, $g_{\mathcal{O}}(d) = d$ and $g_{\mathcal{F}}(f)(t_1, ..., t_k) = f(t_1, ..., t_k)$.

We can compare interpretations (Herbrand, in particular) as follows: For a pair of interpretations $I = (U, g_{\mathcal{O}}, g_{\mathcal{F}}, \jmath_{\#}, \jmath_{*})$ and $\hat{I} = (U, g_{\mathcal{O}}, g_{\mathcal{F}}, \hat{\jmath}_{\#}, \hat{\jmath}_{*})$ differing only in the way they interpret objects as labels, we write $I \preceq \hat{I}$ iff for every object $d \in \mathcal{O}^*$ typed as a functional label and every set-valued label, $e \in \mathcal{O}^*$, $\jmath_{\#}(g_{\mathcal{O}})(d) \preceq_{Map(U,U)} \hat{\jmath}_{\#}(g_{\mathcal{O}})(d)$ and $\jmath_{*}(g_{\mathcal{O}})(e) \preceq_{Map(U,2^U)} \hat{\jmath}_{*}(g_{\mathcal{O}})(e)$. The ordering on lattice mappings was introduced at the beginning of Section 2.3. To spell it out for this particular case, it means that for all $u \in U$, $\jmath_{\#}(g_{\mathcal{O}})(d)(u) \preceq_U \hat{\jmath}_{\#}(g_{\mathcal{O}})(d)(u)$ and $\jmath_{*}(g_{\mathcal{O}})(e)(u) \sqsubseteq_U \hat{\jmath}_{*}(g_{\mathcal{O}})(e)(u)$.

Because $\sqsubseteq_U$ is, strictly speaking, only a preorder (see Section 2.3), $\preceq$ is too a preorder, but not an order. However, as $\sqsubseteq_U$, it is an order modulo the equivalence relation $\approx$, where $I \approx \hat{I}$ iff $I \preceq \hat{I}$ and $\hat{I} \preceq I$.

Having defined $\preceq$, we can now talk about minimal models: I is *minimal* if there is no J s.t. $J \preceq I$ but not $I \preceq J$. The reader can notice that our definitions of minimality and order are in the spirit of the corresponding classic notions for Herbrand interpretations, where "smaller" means "less defined".

In classic logic programming [26], Herbrand interpretations are defined as subsets of the Herbrand base, where the latter is just the set of all ground atomic formulae. In F-logic, the analogue of Herbrand base is the set of all ground F-terms with the class information stripped off [5]. The class information is superfluous here since the validity of the "instance-of" statement of the form $p : q$, where $p, q \in \mathcal{O}^*$, is a consequence of the lattice structure (which is part of the language), and is therefore independent of the program. Let us call such F-terms *bare*.

As in classic logic, in F-logic every subset of the Herbrand base can be associated with a unique Herbrand interpretation.

Proposition 1 *For any subset S of Herbrand base there is a Herbrand interpretation H such that:*

- *H satisfies every F-term in S;*
- *H is a minimal interpretation w.r.t. the preorder $\preceq$;*

Furthermore, such interpretation H is unique modulo the equivalence relation $\approx$ defined earlier.

However, this relationship between sets of bare F-terms and the corresponding Herbrand interpretations is less obvious than in the classic case. For instance, the set $S = \{d[lab \rightarrow a, lab' \rightarrow \{c\}], d[lab \rightarrow b, lab' \rightarrow \{e\}]\}$ corresponds to the interpretation in which *lab* maps d into $lub(a, b)$ and lab' maps it into the set $\{c, e\}$. The rest of the labels map everything to $\bot$ or $\{\ \}$, depending on the type.

Theorem 2 (cf. the Herbrand Theorem) *A finite set of formulae, $\mathbf{S}$, is inconsistent iff so is some finite subset of its ground instances.*

Herbrand Theorem is a basis for the resolution based semi-decision procedure in predicate calculus [11]. In the full paper we will show that, extending the result of [20], a sound and complete resolution-based proof procedure can be defined for F-logic. This, in turn, provides a firm basis for the theory of logic programming. Particularly, model-theoretic semantics of logic

[5] Recall that according to Section 2.2, a term without the class information, e.g. $d[lab \rightarrow c]$, is an abbreviation of the term $\bot : d[lab \rightarrow \bot : c]$.

programs (e.g. *perfect model* semantics [36]) can be extended to F-logic. This will be discussed in a companion paper.

4 Inheritance, Methods, and Higher-Order Queries

In this section we discuss some salient features of the proposed semantics and illustrate them by a number of examples.

4.1 Inheritance

The notion of inheritance is fundamental in AI and object-oriented programming, and a number of researchers have worked on incorporating it into programming languages. Cardelli [10] considered inheritance in the framework of functional programming. He described a type inference procedure which is sound with respect to the denotational semantics of his system. In contrast, we have devised a *logic* in which inheritance is built into the semantics and the proof procedure (in the full paper) is sound and *complete*.

Ait-Kaci and Nasr [3] incorporate inheritance into logic programs by means of a unification algorithm. Although intuitively appealing, this algorithm was not given any semantically sound justification. In addition, Maier [27] has pointed out that their algorithm may be correct for type inferencing, but not for querying databases. Later, Smolka and Ait-Kaci [38] presented a semantics to the unification algorithm of [3] using equational logic. However, it is not clear how to extend this semantics to a full-fledged logic in such a way that the resolution procedure based on the proposed unification algorithm will be sound and complete. Even if it is possible, this still does not make this system applicable to database querying.

There is also ample literature on, so called, *nonmonotonic* inheritance (e.g. [40,41,15]), which is different from the monotonic inheritance of F-logic (see later). Furthermore, in these works inheritance is defined algorithmically and is not built into the semantics, which we consider to be inappropriate for a logic for object-oriented programming. In contrast, *F*–logic inheritance *is* built into the semantics, as follows from the next theorem:

Theorem 3 *Let* $\mathbf{D}$ *be a database,* $T = p[Lab \rightarrow Q,\ Lab' \rightarrow \{R\}]$ *be an F-term and* $\mathbf{D} \models T$. *(Since* Q, R *and* Lab, Lab' *can be nonground F-terms and id-terms, respectively,* T *should be viewed as a universally quantified F-term.) Suppose* $v \in \mathcal{O}^*$ *is an id-term s.t.* $p \preceq_{\mathcal{O}} v$, *i.e.* v *is an instance of class* p. *Then*

$$\mathbf{D} \models v[Lab \rightarrow Q,\ Lab' \rightarrow \{R\}].$$

Thus, whenever $p \preceq_{\mathcal{O}} v$, properties of p also hold for v. In other words, v *inherits* properties of p. Theorem 3 justifies our intuitions about the example of Section 2.1. For instance, since $faculty \preceq_{\mathcal{O}} mary$ (Figure 1), *mary* inherits $supervisor \rightarrow faculty$ from clause (5) of Figure 2.

Sally (clause (4)) provides a more sophisticated example. Since $student \preceq_{\mathcal{O}} sally$, *sally* inherits $age \rightarrow young$ from clause (6) of Figure 2. However, since clause (4) states that *sally* is *midaged*, in every interpretation in which both $sally[age \rightarrow young]$ and $sally[age \rightarrow midaged]$ are true, necessarily it is the case that $sally[age \rightarrow lub(young, midaged)]$ ($\equiv sally[age \rightarrow yuppie]$) is also true, i.e. clauses (4) and (6) logically entail $sally[age \rightarrow yuppie]$. Indeed, in every interpretation $I = < U, g_{\mathcal{O}}, g_{\mathcal{F}}, \jmath_{\#}, \jmath_{*} >$ the label *age*, being interpreted as a monotonic single-valued function $\jmath_{\#}(age)$, has to map $g_{\mathcal{O}}(sally)$ into something which is above both $g_{\mathcal{O}}(young)$ and $g_{\mathcal{O}}(midaged)$. Since $g_{\mathcal{O}} : \mathcal{O} \rightarrow U$ is a lattice homomorphism, we have $lub(\, g_{\mathcal{O}}(young),\, g_{\mathcal{O}}(midaged)\,) = g_{\mathcal{O}}(\, lub(young,\, midaged)\,) = g_{\mathcal{O}}(yuppie)$. Since I is an arbitrarily chosen interpretation, we derive $sally[age \rightarrow yuppie]$. Thus, although the inherited property $age \rightarrow young$ is still true, in fact, we have more: $age \rightarrow yuppie$. This effect can be called *monotonic overwriting* of inheritance.

It is arguable whether monotonic overwriting suffices for all the needs of real world modeling. It is not difficult to think of a situations when, in the above example, one would want *sally* to be *midaged*, completely disregarding the inherited age *young*. Furthermore, in some cases (recall the paragraph discussing *bob*'s supervisor in Section 2.1) inheritance contradicts the other information to such an extent that we have to declare local inconsistency (cf. $bob[supervisor \rightarrow \top]$). Although in some cases this indeed may be the intention, in other situations it may be not, and one needs a formal account for the latter case. Such complete overwriting of inheritance is a (rather simple) instance of nonmonotonic inheritance mentioned earlier, and it is desirable to have it built into the logic the same way as its monotonic counterpart. However, this raises a host of difficult problems and is an issue for future research.

4.2 Browsing Database Schema

As explained earlier, although F-logic formally has a first-order semantics, it is capable of modeling certain higher-order features such as sets, class/subclass hierar-

chy, and scheme quite naturally. The first two are modeled by means of set-valued functions and the lattice structure on $\mathcal{O}^*$, while schema can be reasoned about because labels (which correspond to attributes of the relational model) are represented by id-terms which are virtually indistinguishable from objects.

In this section we discuss the browsing capabilities of F-logic. Some of the higher-order capabilities described in this section were also discussed in [21] in the context of deductive databases. However, the treatment in [21] is not as general and integrated as in the present paper.

Typically, queries in database systems are specified with respect to an existing scheme, which is assumed to be known to the user. The practice shows, however, that this assumption is unrealistic and some kind of *browsing* of the database is necessary. This means that the user has to apply intuitive or exploratory search through the structure of the scheme and the database at the same time and even in the same query (cf. [34]). Many user interfaces to commercial databases support browsing to different extents. The purpose of the following examples is to demonstrate that F–logic provides a unifying framework both for data and schema exploration. We again refer to the example of Section 2.1.

The following pair of rules collects for each instance of class *faculty* all labels which are "more defined" than *person* or {*person*} (including the inherited labels):

$$defined\_labels(X)\,[labels \rightarrow \{L\}] \Leftarrow faculty : X[L \rightarrow person : Z]$$
$$defined\_labels(X)\,[labels \rightarrow \{L\}] \Leftarrow faculty : X[L \rightarrow \{person : Z\}]$$

For the example of Figures 1 and 2, we have: $defined\_labels(mary) = \{friends, supervisor\}$. Replacing *person* by $\bot$ and adding $Z \neq \bot$ in the bodies of the above rules yields the set of all labels which are strictly more defined than $\bot$.

Another example of browsing is retrieval of all objects which mention, say, "CS" directly or indirectly (through other objects). This can be specified as follows:

$$finder(\text{“}CS\text{”})[content \rightarrow \{X\}] \Leftarrow X[Y \rightarrow \text{“}CS\text{”}]$$
$$finder(\text{“}CS\text{”})[content \rightarrow \{X\}] \Leftarrow X[Y \rightarrow \{\text{“}CS\text{”}\}]$$
$$finder(\text{“}CS\text{”})[content \rightarrow \{X\}] \Leftarrow X[Y \rightarrow Z], finder(\text{“}CS\text{”})[content \rightarrow \{Z\}]$$
$$finder(\text{“}CS\text{”})[content \rightarrow \{X\}] \Leftarrow X[Y \rightarrow \{Z\}], finder(\text{“}CS\text{”})[content \rightarrow \{Z\}]$$

For our running example, the query $finder(\text{“}CS\text{”})[content \rightarrow \{X\}]$? will return the set $\{cs_1, cs_2, bob, mary, john\}$.

The inheritance mechanism discussed in the previous section can be also used to enforce the domains of labels. For instance, specifying $person[name \rightarrow string]$ will cause every instance of *person* to inherit the domain *string* for the label *name*. Now, if an F-term specifies a value for *name*, e.g. $john[name \rightarrow \text{“}John\text{”}]$, and this value is an instance of *string*, then everything goes well and, as explained in Section 4.1, "John" overwrites *string*. However, if the specified value is incomparable with *string*, e.g. $john[name \rightarrow 20]$, then, since $lub(20, string) = \top$, $john[name \rightarrow \top]$ is derived.

In relational model, relation schema is usually fixed (e.g., $supplier(sno, sname)$) so that the tuples in a relation are defined over the schema attributes only. In contrast, in object-oriented languages (F-logic in particular), attribute set may vary from object to object. In fact, the general class information (cf. clauses (5) to (7) in Figure 2) limits the schema "from below" by specifying what is generally true about the class, while relational model limits schemes "from above" by specifying the only set of meaningful attributes for a relation. We do not take a stand on whether the notion of schema in relational databases is a modeling necessity or merely an implementational convenience. The following example shows that, if desired, schema restriction in the relational sense can be imposed in F-logic:

$$supplier[X \rightarrow \top] \Leftarrow supplier[X \rightarrow \bot], \quad X \neq sno,\ X \neq sname$$
$$supplier[X \rightarrow \{\top\}] \Leftarrow supplier[X \rightarrow \{\,\}]$$

The first clause states that every label other than *sno* and *sname* maps *supplier* to $\top$; the second clause says that every set-valued label maps *supplier* to the top set. Now, every individual supplier s will inherit these restrictions and therefore every label outside the supplier's scheme will yield meaningless information regarding s.

4.3 Methods

Methods are the means of incorporating data abstraction into object-oriented programming. Since they embody the procedural aspect of the object-oriented paradigm, many researchers believed that methods cannot be cast into a declarative setting. For instance, [24,25] propose a formal data model intended to support a *procedural* object-oriented language. Similarly, in [7] methods written in a procedural language are integrated into a declarative setting.

We believe that the infamous impedance mismatch between programs and data should be overcome in a declarative fashion, which requires methods to be defined declaratively. This is not to say that procedural languages are of no use. However, our contention is that the procedural component should be integrated in a declarative framework in a clean way, e.g. the one alluded to at the end of this section.

In F-logic, declarative definitions of methods are possible because nonground id-terms are allowed to appear in label positions in F-terms. One example to this effect was given in Figure 2. For another one, consider the rule

$$person : X[graduation\_date(I) \rightarrow Y] \Longleftarrow univ : I[alumni \rightarrow \{alumn\_rec : G[stud \rightarrow person : X,\ date \rightarrow year : Y]\}]$$

For each person, the *graduation_date* method is a function from universities to years, and one can ask queries such as

$$X[graduation\_date(I) \rightarrow 1987]?$$
$$john[graduation\_date(I) \rightarrow Y]?$$

to find all persons who graduated from anywhere in 1987, or to find dates and universities *john* graduated from.

By modifying the browsing example of the previous section, we can define a method which returns the set of all objects directly or indirectly referring to the object passed to the method as an argument (the *Stuff* variable):

$$finder[find(Stuff) \rightarrow \{X\}] \Longleftarrow X[Y \rightarrow Stuff]$$
$$finder[find(Stuff) \rightarrow \{X\}] \Longleftarrow X[Y \rightarrow \{Stuff\}]$$
$$finder[find(Stuff) \rightarrow \{X\}] \Longleftarrow X[Y \rightarrow Z],\ finder[find(Stuff) \rightarrow \{Z\}]$$
$$finder[find(Stuff) \rightarrow \{X\}] \Longleftarrow X[Y \rightarrow \{Z\}],\ finder[find(Stuff) \rightarrow \{Z\}]$$

In object-oriented languages, the ability to inherit methods and build them incrementally is responsible for much of the success of this approach in human interfaces and graphics. The following example illustrates how these phenomena can be accounted for in F-logic.

Suppose that $person \preceq_O male, female, writer$. We can define the method *legal_name* as follows. Normally, the legal name is the last name of a person. However, maiden name of a married female as well as a pen-name of a writer is also considered to be a legal name. We can first define this method for each person:

$$X[legal\_names(Y) \rightarrow \{N\}] \Longleftarrow year : Y,\ person : X[last\_name(Y) \rightarrow string : N]$$

and then refine it for females and writers:

$$X[legal\_names(Y) \rightarrow \{N\}] \Longleftarrow year : Y,\ female : X[maiden\_name(Y) \rightarrow string : N]$$

$$X[legal\_names(Y) \rightarrow \{N\}] \Longleftarrow year : Y,\ writer : X[pen\_name(Y) \rightarrow string : N]$$

Thus, if in 1988 *mary* was a married female, a writer, and uses her husband's last name, she will have three different legal names in that year. On the other hand, for a *joe* who is a male and not a writer, this method will return only one legal name.

This example is also an instance of *operator overloading* – another feature attributed to object-oriented programming. This means that the same method name can be used to denote quite different procedures, depending on the class where this name is used. Another instance of overloading can be obtained by modifying the previous example to include the class *company* which is incomparable to *person*. Since companies have a completely different set of rules regulating their legal names, the definition of *legal_name* for class *company* may have little resemblance of the definition of this method for classes *person*, *female*, and *writer*, yet syntactically the name is the same.

Note that, in F-logic, methods are essentially "labels with parameters" and therefore plain labels can be viewed as parameterless methods. This uniformity is rather pleasing and corresponds to the situation in abstract data types. The technique described above allows one to define arbitrarily complex methods, since the full power of logic programming is at our disposal. Alternatively, we could incorporate procedures written in a nonlogic language, such as C or SmallTalk, by considering nonground labels as "computed functions" and adapting the ideas from [30,31].

5 Conclusions

Unlike the relational approach to databases which was initiated by Codd [14] and was based on firm theoretical grounds, object-oriented databases were dominated by "grass-roots" activity where several implementations have been done [44,42,29] without the accompanying theoretical progress. As a result, many researchers had a feeling that the whole area of object-oriented databases is misguided, lacking direction and needing a spokesman, like Codd, who could "*coerce the researchers in this area into using common set of terms and defining a common goal that they are hoping to achieve* [35]".

Our contention is that the problem lies much deeper. When Codd made his influential proposal, he relied on a large body of knowledge in Mathematical Logic concerning predicate calculus. Essentially, he merely applied (in different terms) what logicians had already known for several decades. Logical foundations for object-oriented databases that are parallel to those that underly the relational theory were lacking and, in our opinion, this was a major factor for the uneasy feeling. In his pioneering work [27], Maier proposed a framework for defining model-theoretic semantics for object-oriented logics. However, he encountered certain semantic difficulties with his approach and subsequently abandoned this direction. As it turned out, the difficulties were not fatal, and the theory was repaired and significantly extended in [13,20].

In the present paper, we presented a novel logic which takes the C- and O-logics of [13,20] into a new dimension: F-logic is capable of representing most of what is known as the object-oriented paradigm. We provided a formal semantics for that logic and showed that it embodies in a natural way the notions of complex object, object identity, inheritance, methods, and schema. Although not presented in this paper, we note that F-logic has a sound and complete resolution-based proof procedure which makes it also computationally attractive and renders it a suitable basis for developing a theory of object-oriented logic programming. This issue will be discussed in a companion paper.

Acknowledgements: We have obviously benefited from Dave Maier's original paper on O-logic [27], as well as from his valuable suggestions. Discussions with Dave Warren have been extremely helpful; in particular, Dave helped us understand the "orderness" of the semantics we have developed. We also thank James Wu for finding several inaccuracies in the earlier draft of this paper.

References

[1] S. Abiteboul and C. Beeri. On the Power of Languages for Manipulation of Complex Objects. 1987. manuscript.

[2] S. Abiteboul and S. Grumbach. COL: A Logic-Based Language for Complex Objects. In *Workshop on Database Programming Languages*, pages 253–276, Roscoff, France, September 1987.

[3] H. Ait-Kaci and R. Nasr. LOGIN: A Logic Programming Language With Built-in Inheritance. *J. Logic Programming*, 3:185–215, 1986.

[4] F. Bancilhon. Object-Oriented Database Systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database Systems*, pages 152–162, 1988.

[5] F. Bancilhon and S.N. Khoshafian. A Calculus of Complex Objects. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database Systems*, pages 53–59, March 1986.

[6] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. *Sets and Negation in a Logic Database Language (LDL)*. Technical Report, MCC, 1987.

[7] C. Beeri, R. Nasr, and S. Tsur. Embedding ψ-terms in a Horn-clause Logic Language. In *Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 347–359, Morgan Kaufmann, 1988.

[8] P. Buneman and A. Ohori. Using Powerdomains to Generalize Relational Databases. *Theoretical Computer Science*, 1989. to appear.

[9] H. Ait-Kaci C. Zaniolo, D. Beech, S. Cammarata, L. Kerschberg, and D. Maier. *Object-Oriented Database and Knowledge Systems*. Technical Report DB-038-85, MCC, 1985.

[10] L. Cardelli. A Semantics of Multiple Inheritance. In *Int. Symp. on Semantics of data Types, LNCS 173*, pages 51–67, June 1984.

[11] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[12] W. Chen, M. Kifer, and D.S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *2-nd Intl. Workshop on Database Programming Languages*, Morgan-Kaufmann, June 1989.

[13] W. Chen and D.S. Warren. C-logic for Complex Objects. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database Systems*, March 1989. to appear.

[14] E.F. Codd. A Relational Model For Large Shared Data Banks. *Communications of ACM*, 13(6):377–387, 1970.

[15] D.W. Etherington and R. Reiter. On Inheritance Hierarchies with Exceptions. In *AAAI-83*, pages 104–108, Washington, D.C., 1983.

[16] R. Fikes and T. Kehler. The Role of Frame-Based Representation in Reasoning. *Commun. ACM*, 28(9):904–920, 1985.

[17] M.L. Ginsberg. Multivalued Logics. In M.L. Ginsberg, editor, *Readings in Non-Monotonic Reasoning*, pages 251–255, Morgan-Kaufmann, 1987.

[18] P.J. Hayes. The Logic for Frames. In D. Metzing, editor, *Frame Conception and Text Understanding*, pages 46–61, Walter de Gruyter and Co., 1979.

[19] S.N. Khoshafian and G.P. Copeland. Object Identity. In *OOPSLA-86*, pages 406–416, 1986.

[20] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited). In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database Systems*, March 1989. to appear.

[21] R. Krishnamurthy and S. Naqvi. Towards a Real Horn Clause Language. In *Proceedings of the Intl. Conference on Very Large Data Bases*, 1988.

[22] G. Kuper and M.Y. Vardi. A New Approach to Database Logic. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database Systems*, 1984.

[23] G.M. Kuper. *An Extension of LPS to Arbitrary Sets.* Technical Report, IBM, Yorktown Heights, 1987.

[24] C. Lecluse and P. Richard. Modeling Inheritance and Genericity in Object-Oriented Databases. In *2-nd Int. Conf. on Database Theory (ICDT), LNCS 326*, pages 223–238, Springer Verlag, Bruges, Belgium, 1988.

[25] C. Lecluse, P. Richard, and F. Veles. O_2, an Object-Oriented Data Model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 424–433, 1988.

[26] J.W. Lloyd. *Foundations of Logic Programming (Second Edition).* Springer Verlag, 1987.

[27] D. Maier. A Logic for Objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.

[28] D. Maier. Why Database Languages are a Bad Idea (position paper). In *Proc. of the Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[29] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of OOPSLA-86*, pages 472–482, 1986.

[30] D. Maier and D.S. Warren. *A Theory of Computed Relations.* Technical Report 80/12, Department of Computer Science, SUNY at Stony Brook, November 1980.

[31] D. Maier and D.S. Warren. *Incorporation Computed Relations in Relational Databases.* Technical Report 80/17, Department of Computer Science, SUNY at Stony Brook, December 1980.

[32] J. McCarthy. First Order Theories of Individual Concepts and Propositions. In J.E. Hayes and D. Michie, editors, *Machine Inteligence*, pages 129–147, Edinburgh University Press, 1979.

[33] M. Minsky. A Framework for Representing Knowledge. In J. Haugeland, editor, *Mind design*, pages 95–128, MIT Press, Cambridge, MA, 1981.

[34] A. Motro. BAROQUE: A Browser for Relational Databases. *ACM Trans. on Office Information Systems*, 4(2):164–181, 1986.

[35] E. Neuhold and M. Stonebraker. Future Directions in DBMS Research. 1988. The Laguna Beech Report.

[36] T.C. Przymusinski. On The Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, Morgan Kaufmann, Los Altos, CA, 1988.

[37] M.A. Roth, H.F. Korth, and A. Silberschatz. *Extended Algebra and Calculus for ¬1NF Relational Databases.* Technical Report 84-36, Univ. of Texas at Austin, 1985.

[38] G. Smolka and H. Ait-Kaci. *Inheritance Hierarchies: Semantics and Unification.* Technical Report AI-057-87, MCC, May 1987.

[39] M. Stefik and D.G. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 40–62, January 1986.

[40] D.S. Touretzky. *The Mathematics of Inheritance.* Morgan-Kaufmann, Los Altos, CA, 1986.

[41] D.S. Touretzky, J.F. Horty, and R.H. Thomason. A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems. In *IJCAI-87*, pages 476–482, 1987.

[42] *Vbase Object Manager.* Ontologic, Inc., 1986. User Manual.

[43] P. Wegner. The Object-Oriented Classification Paradigm. 1987. manuscript.

[44] D. Woelk, W. Kim, and W. Luther. Multimedia Information Management in an Object-Oriented Database System. In *Proceedings of the Intl. Conference on Very Large Data Bases*, 1987.