Semantic Query Optimization in Deductive Object-Oriented Databases

Jong P. Yoon † and *Larry Kerschberg* ‡
Department of Information and Software Systems Engineering
School of Information Technology and Engineering
George Mason University, Fairfax, VA 22030-4444
{†jyoon,‡kersch}@isse.gmu.edu

Abstract. This paper addresses the problem of semantic query reformulation in the context of object-oriented deductive databases. It extends the declarative object-oriented specifications of F-logic proposed by Kifer and Lausen using the semantic query optimization technique developed by Chakravarthy, Grant, and Minker. In general, query processing in object-oriented databases is expensive when a query incorporates declarative rules, methods and inherited properties. We introduce the technique of semantic query reformulation for F-logic queries which transforms the original query into an equivalent, semantically-rich query that is more efficiently processed. We also discuss the issues of conflict resolution strategies and query evaluation priorities for queries involving the upper bounds of objects in the F-logic "type" lattice.

1 Introduction

In traditional database systems, queries are typically optimized by either access method cost models or rule-based methods. Access methods constitute an index structure (or an efficient plan) for executing a user's declarative query. The rule-based methods generate an evaluation plan for a query. Some researchers [3, 4, 10, 13] have used heuristic rules to optimize queries. Rules are used to determine efficient search strategies.

It is difficult to associate rules to a query for efficient processing. To lessen this difficulty, a few researchers have developed so called semantic query reformulation [2, 3, 7]. Constraints are associated with a query which in turn becomes semantically optimized [2]. Integrity constraints may be added to queries as functional equations [3]. Type checking is performed at query compilation time [1]. We apply the query optimization concept [2] to queries expressed in F-logic [6]. We propose a query reformulation scheme by which a user-issued query is reformulated into an equivalent and semantically-rich query; the query incorporating rules and inheritances, and resolves conflicts between inherited and derived values. This paper is extended from the earlier work [5]. In particular, the query reformulation process exploits the partial ordering information for resolution between a query and other available information (e.g., rules) stated above. Using the partial ordering information makes it easier to deal with both single and multiple inheritances.

This paper is organized as follows: First, we review F-logic in Section 2. We examine research issues pertinent to semantic query optimization and develop the steps for query optimization in Section 3, including the aspects relevant

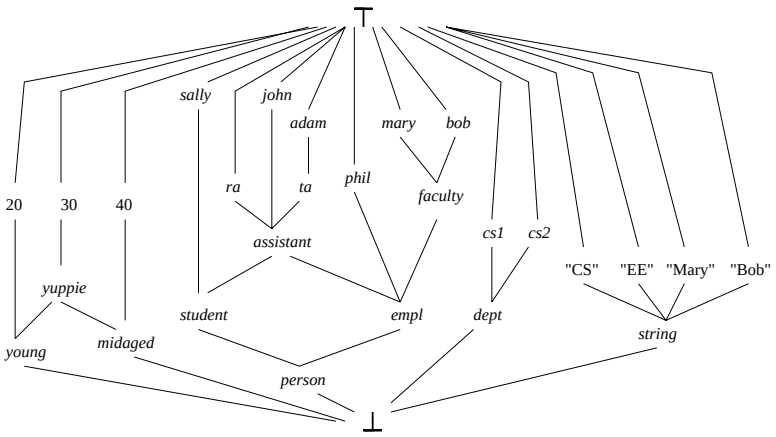

Figure 1: A Lattice of Database Example

to object-oriented databases. The query reformulation process is applied to four examples retaining various object-oriented features in Section 4. Due to several features of the object-oriented paradigm, multiple equivalent queries can be obtained from the original query. Evaluation priority for the reformulated queries is discussed in Section 5. Finally, Section 6 summarizes our work.

2 Preliminaries and Definitions

We introduce the background knowledge used in *F-Logic* in this section, leaving the syntactic and semantic details to the original paper [6]. The unique features of F-logic which are essential to our work will be described. Before describing the features of F-logic, the partial ordering theory [14] is introduced.

Lattice

A *lattice* is a formalism for an ordering of values based on their information content [14]. The lattice in Figure 1 shows part of the IS-A hierarchy. Notice that the IS-A relation in typical object-oriented databases (see Figure 4) is depicted reversely in a lattice. In Figure 1, the information content of *person* is contained in the information content of *student*. Similarly, *empl* contains more (or equal) information than *person*. Both *student* and *empl* are the upper bound of *person*. For example, a statement *student* : *john* (John is a student) is more informative than *person* : *john* because every student is a person, but not vice versa.

Lattices, therefore, determine the least upper bound (*lub*) of objects, which is the smallest upper bound of the objects. The *least upper bound* function, $lub(e_1, .., e_n)$, in short, returns the least upper bound type for a set of the arguments $e_1, ..e_n$. Our resolution of the type conflict is based on denotational semantics: (1) the *least upper bound* of types is constructed for a single-valued

label; (2) the *union* of the sets of types is constructed for set-valued labels. A single-valued label for more than one object is set by computing the function *lub*. For example, from $lub(student, empl) = assistant$, the single-valued label of the object which denotes both *student* and *empl* is constructed as *assistant* according to the lattice in Figure 1. From $lub(ra, john) = \top$, where $\top$ means the greatest element which is viewed as a "meaningless" (or inconsistent) object, it is likely that an object which denotes both *ra* and *john* is meaningless. Single-valued labels denoting $\top$ will be discussed in more detail in Section 5. Similarly, $\bot$ is the least element, meaning an "unknown" object. On the other hand, a set-valued label is set by union of appropriate object. For example, the type conflict between $mary[friends \rightarrow \{bob, sally\}]$ and the inherited property $faculty[friends \rightarrow \{faculty : department\_faculty\}]$ is resolved by union of those two labels: $mary[friends \rightarrow \{bob, sally, department\_faculty\}$.

F-Logic

Now, we review the unique features of F-logic [6]. A class is viewed as an instance of a super-class. The object *student* can be viewed as representing the class of students and at the same time as an instance of its superclass represented by the object *person*. Each object has an *object identity* (*oid*).

F-Logic defines *terms*. The so called "F-term" $P : Q[label_i \rightarrow T_i]$ is a statement about an object Q asserting that it is an instance of the class P and has properties specified by the labels. Note P, Q and *label* denote "id-term," and T denotes "type constructor" which may be, in turn, an F-term. A term constructed by an F-term is called a *nested* term. The terms can be either constant symbols, variable symbols, or function symbols. T can be either single-valued, function-valued, or set-valued. $P : Q[label_i \rightarrow T_i]$ can be written as $P : Q$ if no labels are specified.

For example, consider the term $student : john[name \rightarrow$ "John"]. The object identity *john* is a student whose name is bound by "John." All terms are constant single-valued in this F-term. However, in

$student : john[works\_for \rightarrow dept : cs[chair \rightarrow$ "Peter"]],

the type constructor *works_for* for *john* is a nested term $dept : cs[chair \rightarrow$ "Peter"]]. John works for the *cs* department where the chair is "Peter." In

$faculty : peter[authorship \rightarrow$ {publication: B}],

the faculty *peter* has an authorship that is a set B which is in publication. The type constructor *authorship* for *peter* is set-valued.

Now, we discuss the decomposition and composition of terms. $X[label_1 \rightarrow a, label_2 \rightarrow Y]$ is equivalent to a conjunction of its atoms $X[label_1 \rightarrow a]$, $X[label_2 \rightarrow Y]$. We call decomposition of the formula $X[label_1 \rightarrow a, label_2 \rightarrow Y]$ into the two atoms $X[label_1 \rightarrow a]$ and $X[label_2 \rightarrow Y]$ "*fission*," while the reverse process (composition of atoms) "*fusion*." For example, the two terms

Facts:
(1) $faculty : bob\ [name \rightarrow$ "Bob", $age \rightarrow 40, works \rightarrow dept :$
$cs_1[dname \rightarrow$ "CS", $mngr \rightarrow empl : phil]]$
(2) $faculty : mary\ [name \rightarrow$ "Mary", $age \rightarrow 30, friends \rightarrow \{bob, sally\},$
$works \rightarrow dept : cs_2[dname \rightarrow$ "CS"]]
(3) $assistant : john[name \rightarrow$ "John", $works \rightarrow cs_1[dname \rightarrow$ "CS"]]
(4) $student : sally[age \rightarrow middleaged]$

General Class Description:
(5) $faculty[supervisor \rightarrow faculty, age \rightarrow middleaged]$
(6) $student[age \rightarrow young]$
(7) $empl[supervisor \rightarrow empl]$

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

Figure 2: F-Logic Example Specification

$student : john[name \rightarrow$ "John"], and
$student : john[works\_for \rightarrow dept : cs[chair \rightarrow$ "Peter"]]
are "fused" into a term
$student : john[name \rightarrow$ "John", $works\_for \rightarrow dept : cs[chair \rightarrow$ "Peter"]].

Inheritance

We assume *monotonic* inheritance. If $t_1 \preceq_{\mathcal{O}} s_1, ...,$ and $t_n \preceq_{\mathcal{O}} s_n$ then $f(t_1, ..., t_n) \preceq_{\mathcal{O}} f(s_1, ..., s_n)$, where $\preceq_{\mathcal{O}}$ denotes the ordering in a lattice, where f is an n-ary function and is used to construct objects. For example, if $person \preceq_{\mathcal{O}} john$ then $car(person) \preceq_{\mathcal{O}} car(john)$.

eval Function

We also introduce an evaluation function, $eval(X, Y)$, a binary function with arguments "object" and "attribute." The function $eval(X, Y)$ returns a set of the values associated with the attribute Y of the object X.

$eval(X, Y) = \{y \mid y$ is a value of the attribute Y of class X such that $X[Y \rightarrow y]$, or $X[Y \rightarrow \{y\}]\}$

In the above example, $eval(john, works\_for) = cs$ and $eval(cs, chair) =$ "Peter".

With this background of F-Logic, consider the facts and rules [6]. The objects are depicted in a lattice as shown in Figure 1 and described in F-logic in Figure 2. The object *mary*, named "Mary", is 30 years old, has the friends *bob* and *sally*, and works at the department "CS" whose *oid* is cs_2 as in (2) below. The general class descriptions play the role of *typing constraints*. The attribute *supervisor*, not defined in $faculty$, is inherited by $faculty$ as defined in (5). In the rule (8), an employee E's supervisor is M if E works for a department managed by M.

3 Semantic Query Reformulation

This section discusses F-logic query reformulation. Before discussing the subject, we examine the research issues concerning object-oriented databases in F-Logic which we believe have not yet been taken into account.

- Unified Representation. Rule execution is by nature expensive, in contrast to database retrieval. How can rules be added to queries so that only the queries are evaluated?
- Multiple Reformulation Strategies. A query may be semantically reformulated by (1) rule evaluation and (2) property inheritance. Can a query specify both at the same time?
- Activeness. Rules are not actively executed as a query is posed. Conflicting values are not automatically resolved.

Example: Consider the query (12) [6]:

(12) $empl : X[supervisor \rightarrow Y, age \rightarrow middleaged : Z, works \rightarrow D[dname \rightarrow \text{"CS"}]]$

requesting information about all middle aged employees working for the "CS" departments. Suppose the database consists of the rule (8) and the general class description (5) in Figure 2. The issues are how to associate rules and general class descriptions with a query and what to do for conflicting results. □

3.1 Query Reformulation Process

In a database S having rules and/or inherited properties (denoted as K for both) available, a query Q is posed. It is well known that the rules or inherited properties K are executed against the database S to prove that the query Q entails the answer A[1]. Logically speaking, $S, K \vdash A \supset Q$ holds. It can be rewritten as $S \vdash A \supset (\neg K \vee Q)$. Therefore, it is true that $S \vdash A \supset \neg(K \wedge \neg Q)$. That is, the query Q is equivalent to the reformulated query "$\neg(K \wedge \neg Q)$." Thus, the reformulated query is obtained by negating the resolvent of K and the negation of Q. Notice that the resolvent (or *residue* [2]) is the remaining atoms from resolution.

With the above rationale, we propose the process of query reformulation. Resolution of a rule with the negation of a query results in the negation of the reformulated query.

Query Reformulation Process:

1. Negation. A query is negated.
2. Fission. A rule and the negated query are decomposed into a set of atoms.
3. Substitution. To unify two atoms, (1) a class can be substituted with sub-classes since property inheritance is allowed; and then (2) a (query)

[1] That is, $S, K \vdash A \equiv Q$ holds. It is also held that $S, K \vdash (A \supset Q) \wedge (Q \supset A)$. Assuming that both A and Q are true, we can consider only $S, K \vdash A \supset Q$.

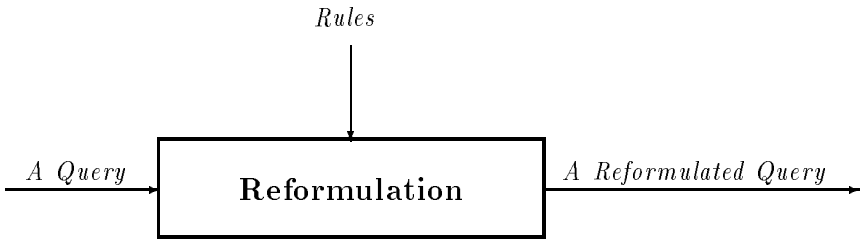

Figure 3: Update Reformulation

variable is substituted with another variable or (3) a variable is substituted with a query value. Note that this step will be discussed in detail in the following sub-section.

4. Resolution. Resolve the above two clauses. The resolvent is obtained.
5. Fusion. We compose the resolvents into a formula.
6. Negation. Finally, the negation of the fused formula is a reformulated query.

The above process is performed within the box in Figure 3. A given query is reformulated into a query with associated rules. The reformulated query is more semantically-rich than the original query. As seen, the resolution step requires substitution between atoms. We will discuss substitution in the following sub-section.

3.2 Object Type Unification

Unification is the process of determining whether two expressions can be made identical by appropriate substitutions for their variables. As we shall see, making this determination is an essential part of our semantic query optimization. A *substitution* is any finite set of bindings between atoms[2], the atoms in a query and the atoms in a rule.

Recall that to unify two atoms, (1) a class can be substituted with sub-classes since property inheritance is allowed; and then (2) a variable is substituted with another (query) variable or (3) a variable is substituted with a (query) value. In this section, we emphasize on how to obtain a substitution between classes. Since the information content of a super class is contained in the information content of a sub-class, variables of a super class are substituted with values (or variables) of a sub-class. More generally speaking, in the term $x : Q$, x are substituted with $lub(x, y)$ where $y : Q$ for an object Q. Of course, if x is a super class of y, x is substituted with y by means of inheritance. Consider the typical IS-A hierarchy example in Figure 4.

[2]Notice that resolution for semantic query processing does not require substitution of variables with expressions.

Figure 4: IS-A Hierarchy Example

In the figure, (a) depicts that O_1 is the super class of O_2. The object type O_2 is unified with O_1 because every object of O_2 is also in O_1 but not vice versa. The substitution is $\{O_1/lub(O_1, O_2)\} \equiv \{O_1/O_2\}$. Similarly, in the figure, (b) depicts that O_3 and O_4 are the super classes of O_5. Due to multiple inheritance, O_3 and O_4 can be unified through the their sub-class. The substitution is, then, $\{O_3/lub(O_3, O_4)\} \equiv \{O_3/O_5\}$, and $\{O_4/lub(O_3, O_4)\} \equiv \{O_4/O_5\}$. For example, the constitution $\{person : Q/student : Q\}$ is possible, but $\{student : Q/person : Q\}$ is not, because $lub(person, student) = student$. It is true that not all persons are students but all students are persons. Substitution of lattice information is very useful. Although x and y are not in a super or sub-class relation, both x and y can be substituted with $lub(x, y)$. For example, the substitution $\{student : Q/assistant : Q;\ empl : Q/assistant : Q\}$ are possible because $lub(student, empl) = assistant$. Using lattice information makes it easier to deal with unification under multiple inheritance.

Moreover, if an atom contains a nested term, in order to unify those two nested terms, an *eval* function is used so that the *lub* is obtained. For example, consider

(8) $E[supervisor \rightarrow M] \Leftarrow empl : E[works \rightarrow dept : D[mngr \rightarrow empl : M]]$
(13) $E[supervisor \rightarrow N] \Leftarrow student : E[grades \rightarrow course : C[instructor \rightarrow faculty : N]]$

The left-hand-side of two rules (8) and (13) can be unified by the substitution

$\{M/lub(eval(D, mngr), eval(C, instructor));\ N/lub(eval(D, mngr), eval(C, instructor))\}$. As discussed in Section 2, $eval(D, mngr)$ returns an object which is an employee in this case. Clearly, the left-hand-side is bound by a least upper bound between a manager employee and a faculty as instructor.

4 Examples

We analyze the four possible cases of semantic query reformulation. (1) Query with rules, where deductive rules are associated with queries, (2) Query together with inheritance (single and multiple inheritance), where the structural properties such as relationships of super- and sub- objects are associated with queries, and (3) Query with methods (*Functions*), where methods which are functional operations are associated with queries (see [5]). Of course, these three cases can be mixed. In this paper, we demonstrate only two cases (1) and (2) above.

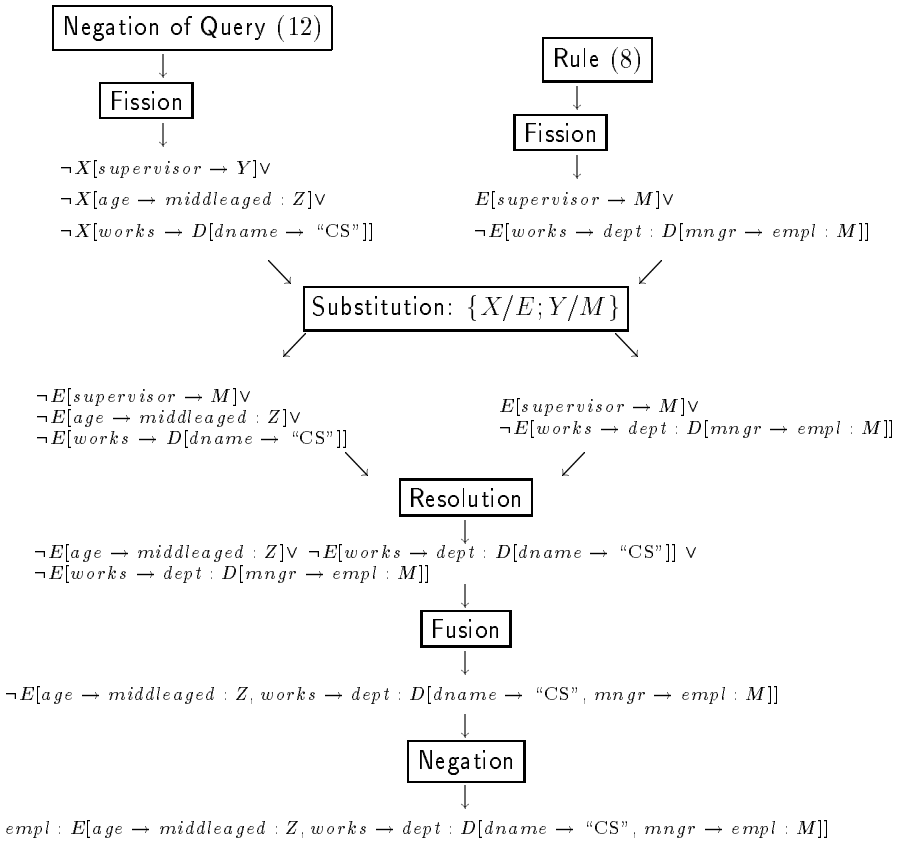

Figure 5: Query Reformulation with a Rule

4.1 Query Reformulation with a Rule

Consider the query (12) and the rule (8).

(12) $empl : X[supervisor \rightarrow Y, age \rightarrow middleaged : Z,\ works \rightarrow D[dname \rightarrow \text{"CS"}]]$
(8) $E[supervisor \rightarrow M] \Longleftarrow empl : E[works \rightarrow dept : D[mngr \rightarrow empl : M]]$

The negations of (12) and (8) are arranged into two columns and the query reformulation process is applied as shown in Figure 5.

The atoms obtained from the negation of (12) and (8) are unified by substituting X with E and Y with M. Then, a pair of the two atoms, $E[supervisor \rightarrow M]$ and $\neg E[supervisor \rightarrow M]$, is removed. The reformulated query returns values for E, Z, D and M as requested by the original query. It turns out that this query specification incorporates the rule evaluation.

4.2 Query Reformulation with Inheritance

The property inherited from a super-class is associated with a query as shown in Figure 6. Consider the same query (12)

(12) $faculty : X[supervisor \rightarrow Y, age \rightarrow middleaged : Z,\ works \rightarrow D[dname \rightarrow$ "CS"]]

and both the rule (8) and the general description (5). The general description (5) is:

$faculty[supervisor \rightarrow faculty, age \rightarrow middleaged]$.

Before resolution with the query, rules and inherited properties are resolved. The resolvent from this resolution is, then, resolved with the query. The class *faculty* of the general class description (5) is used for substitution with the rule (8) and then with the query (12). This substitution preserves property inheritance.

Notice that the label "*supervisor*" bound by both *faculty* in the atom $faculty[$ $supervisor \rightarrow faculty]$ and M in the atom $faculty : E[works \rightarrow dept : D[mngr \rightarrow$ $empl : M]]$ causes the conflicting *oid*s. The two *oid*s, *faculty* and M, are bounded by the $\top$ that is returned from $lub(faculty, M)$ as shown in the lattice of Figure 1. This reformulated query expresses more semantics than the original query (12) by specifying *lub* functions.

4.3 Query Reformulation with Multiple Inheritance

Consider the following rules:

(8) $E[supervisor \rightarrow M] \Leftarrow empl : E[works \rightarrow dept : D[mngr \rightarrow empl : M]]$
(13) $E[supervisor \rightarrow N] \Leftarrow student : E[grades \rightarrow course : C[instructor \rightarrow faculty : N]]$

The above two rules are available in the class *assistant*. The rule (8) is inherited from *empl*, while the rule (13) is inherited from *student*. Suppose the following query is posed to list the supervisors for *assistant*:

(12) $assistant : X[supervisor \rightarrow Y, age \rightarrow middleaged : Z,\ works \rightarrow D[dname \rightarrow$ "CS"]]

Two inherited rules and the query are arranged into three columns and the query reformulation process is applied as shown in Figure 7. Notice that not all type constructors have to be unified. The type constructors are unified if they bind the same attribute. For example, $empl : M$ and $empl : N$ are not unified because their attributes are not the same but *mngr* and *instructor*, respectively. However, in $E[supervisor \rightarrow M]$ and $E[supervisor \rightarrow N]$, M and N are unified because their attributes are binding the same *supervisor*.

5 Evaluation of Reformulated Queries

It is shown that upper bound objects contain more (specific) information than lower bound objects [14]. If more specific objects are considered in queries, more specific information can be obtained. We believe that *queries specified by the upper bound of objects should be evaluated first* similar to the "specificity"

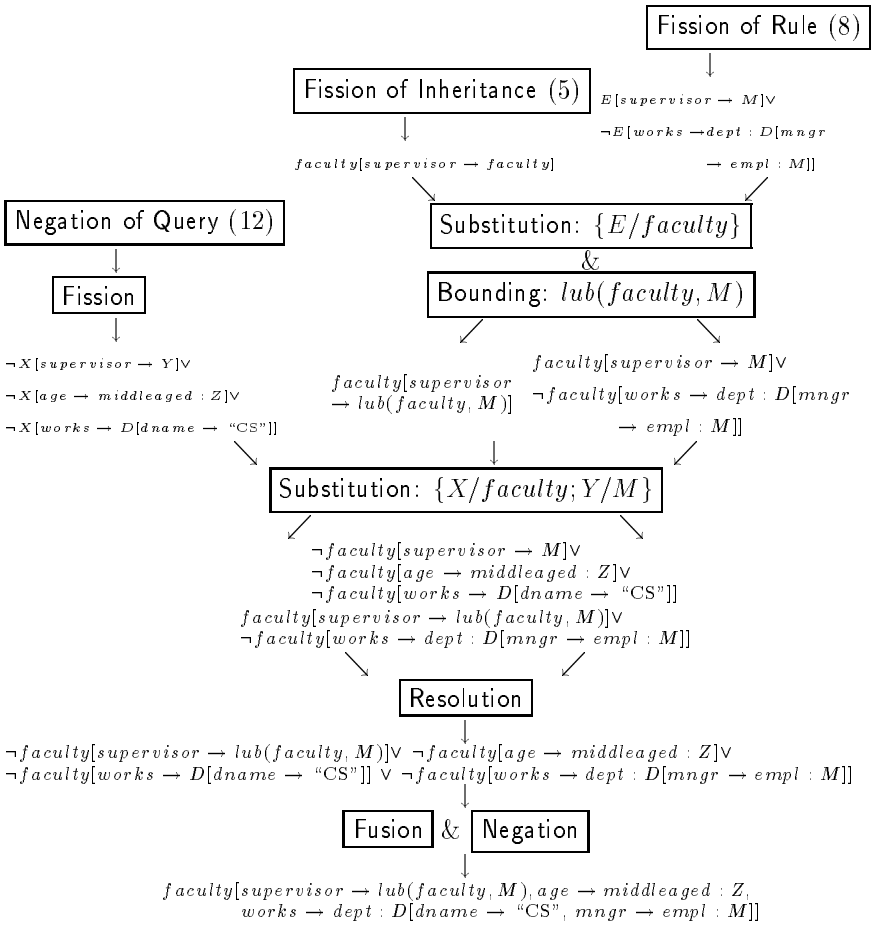

Figure 6: Query Reformulation with Inheritance

precedence in production systems. If two queries are posed at objects in two different levels of specificity (i.e., one object in a super class and the other in a sub-class), the query posed to objects at the more specific level is evaluated first. That is, a query posed to sub-objects is evaluated first.

Regarding the lattice information, if more than one object are denoted by a single-valued label, $[X \rightarrow lub(e_1, e_2, ..., e_n)]$, the label is, if not $\top$, set by computing the least upper bound of those objects. For example, consider $[instructor \rightarrow lub(student, empl)]$. The object *instructor* which is both *student* and *empl* is *assistant* due to $lub(student, empl) = assistant$. However, if $lub(e_1, .., e_n) = \top$, then the least upper bound may not be an acceptable solution. In this case, the *lub* function itself can be shown so that the single-valued label can denote either e_1, e_2, or e_n. Of course, interpretations of the *lub* can be leave to users. For example, in the case that $[good\_student \rightarrow lub(ra, john)] = [good\_student \rightarrow \top]$,

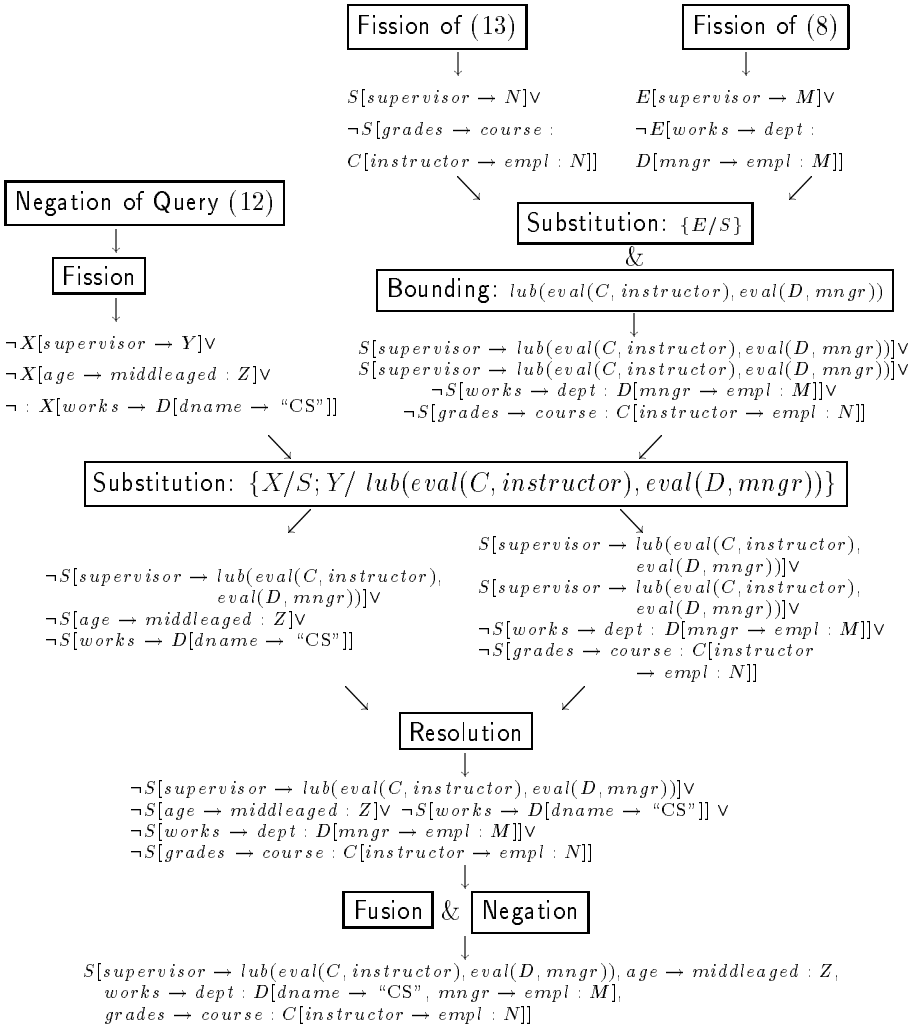

Figure 7: Query Reformulation with Multiple Inheritance

the label *good_student* may be either *ra* or *john* rather than concluding an "inconsistent" object.

Example 1. Consider the following two queries, Q1 and Q2, reformulated in Section 4.1 and 4.2, and the facts (1) and (2):

(Q1) $empl : E[age \rightarrow middleaged : Z, works \rightarrow D[dname \rightarrow$ "CS", $mngr \rightarrow empl : M]]$

(Q2) $faculty : E[supervisor \rightarrow lub(faculty, M), age \rightarrow middleaged : Z,$
$works \rightarrow dept : D[dname \rightarrow$ "CS", $mngr \rightarrow empl : M]]$

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

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

Because Q2 is posed at *faculty* which is sub-class of *empl*, it is evaluated first. With the facts (1) binding M by *phil*, the answer (14) is obtained because $lub(faculty, phil) = \top$. With (2), the variable M is unknown or the least element, then, $lub(faculty, \bot) \equiv faculty$ and the answer is (15).

(14) $bob[supervisor \rightarrow lub(faculty, phil), age \rightarrow 40, works \rightarrow cs_1]$
(15) $mary[supervisor \rightarrow faculty, age \rightarrow 30, works \rightarrow cs_2]$

If (14) were used to evaluate Q1 first, the value for *supervisor* would be *phil* which causes a conflict with the value *faculty* inherited from the general description (5). These answers may resolve the conflict between *phil* and *faculty* by obtaining $\top$, "inconsistency," from $lub(faculty, phil)$ in Q2. However, if the value "inconsistency" is not acceptable to the attribute *supervisor*, the *lub* function may be shown for potential answers, say, either *faculty* or *phil*. □

If two queries are posed to objects in a same level of the specificity, the query specified by the upper bound of objects is evaluated first. That is, a query specified by objects at the more specific level is evaluated first.

Example 2. Section 4.4 illustrated how the query Q3 is reformulated into the query Q4. This example shows that the reformulated query Q4 has expressive power and is semantically richer than Q3.

(Q3) $assistant : X[supervisor \rightarrow Y, age \rightarrow middleaged : Z, works \rightarrow D[dname \rightarrow$ "CS"$]]$
(Q4) $assistant : E[supervisor \rightarrow lub(eval(C, instructor), eval(D, mngr)),$
$age \rightarrow middleaged : Z, works \rightarrow dept : D[dname \rightarrow$ "CS",
$mngr \rightarrow empl : M], grades \rightarrow course : C[instructor \rightarrow empl : N]]$

Consider the following facts:

(1) $faculty : bob\ [name \rightarrow$ "Bob", $age \rightarrow 40, works \rightarrow dept :$
$cs_1[dname \rightarrow$ "CS", $mngr \rightarrow empl : phil]]$
(3) $assistant : john[name \rightarrow$ "John", $works \rightarrow cs_1[dname \rightarrow$ "CS"$]]$
(4) $student : sally[age \rightarrow middleaged]$
(16) $ta : adam[age \rightarrow 30, works \rightarrow dept : cs_2[dname \rightarrow$ "CS"$],$
$grades \rightarrow course : os_1[instructor \rightarrow$ "Peter"$]]$

Consider the reformulated query Q4 first. The object in (1) is not applicable for this query. For the fact (3), the attribute *mngr* is bound by *phil* because he works for cs_1 whose manager is *phil*, and the attribute *instructor* is $\bot$ "unknown." That is,

$lub(eval(cs_1, mngr), \perp) = lub(phil, \perp) = phil.$

john's supervisor is *phil*. For *sally*, $lub(\perp, \perp) = \perp$, so her supervisor is unknown. However, in the same manner, *adam*'s supervisor is "Peter." Hence, the answer to the above reformulated query includes:

(17) $john[supervisor \rightarrow phil, age \rightarrow young]$
(18) $sally[supervisor \rightarrow \perp, age \rightarrow middleaged]$
(19) $adam[supervisor \rightarrow$ "Peter", $age \rightarrow 30]$

Note that *john* is young because of the inherited property (6) $student[age \rightarrow young]$.

Without query reformulation, it is not possible to obtain sound answers from Q3. That is, although Q3 may produce answers by means rules and inherited properties, it does not deal with conflicting answers. The conflicting answers can be handled by Q4 efficiently as seen above. □

6 Conclusion

Simple F-logic queries can be reformulated into equivalent and semantically-richer queries that incorporate a rule, a more general description, or a method. We develop the query reformulation process. Reformulated queries may consist of the least upper bound *lub* of the conflicting values, thereby resolving conflicts. Queries with associated rules and inherited properties are semantically rich in that types can be checked by the associated rules specifying that type, query processing can be limited within a range if restricted by constraints, and answers may be intensional [8, 11]. Due to the several features of the object-oriented paradigm available and one or more rules considered, several queries can be generated for an original query. To resolve the conflicting answers, the evaluation precedence of the reformulated queries is also discussed. We believe that queries specified by upper bound objects should be evaluated first.

The unique contributions of this paper are:

- Semantic query optimization techniques have been extended to apply to object-oriented declarative databases, as those expressed in F-logic [6].
- Object-oriented query reformulation techniques extended the notions of semantic query optimization [2] to include active rules and inherited properties.
- Conflict resolution and query evaluation strategies are proposed to ensure the proper evaluation of multiple, semantically-equivalent queries derived from the initial F-logic expression.

Association of *recursive rules* [12] with a query is a topic for future research. Magic-set theory [9] may be also employed in our query reformulation process.

Acknowledgement

This research is supported in part by an ARPA grant, administered by the Office of Naval Research under grant number N0014-92-J-4038.

References

[1] A. Borgida. Type systems for query class hierarchies with non-strict inheritance. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 394–400, 1989.

[2] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):163–207, June 1990.

[3] Georges Gardarin and Rosana S. Lanzelotte. Optimizing object-oriented database queries using cost-controlled rewriting. In *Proc. of 3rd Int'l Conf. on Extending Database Technology*, pages 534–549, Vienna, Austria, 1992.

[4] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 290–301, Brisbane, Australia, 1990.

[5] Larry Kerschberg and Jong P. Yoon. Semantic query reformulation in object-oriented databases. In *Proc. of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 73–85, Washington, D.C., 1993.

[6] Michael Kifer and George Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 134–146, Portland, Oregon, 1989.

[7] Sanggoo Lee, Lawrence J. Henschen, and Ghassan Z. Qadah. Semantic query reformulation in deductive databases. In *Intl. Conf. on Data Engineering*, pages 232–239, 1991.

[8] A. Motro. Using integrity constraints to provide intensional answers to relational queries. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 237–246, Amsterdam, 1989.

[9] Inderpal S. Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 247–258. 1990.

[10] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 39–48, 1992.

[11] A. Pirotte and D. Roelants. Constraints for improving the generation of intensional answers in a deductive database. In *5th Int. Conf. on Data Engineering*, pages 652–659, LA, 1989.

[12] Kenneth A. Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 92–101. 1990.

[13] P. Griffiths Selinger and et al. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979.

[14] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, MA, 1977.