

Inheritance and Rules in Object-Oriented Semantic Web Languages*

Guizhen Yang¹ and Michael Kifer²

¹ Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260, U.S.A.

`gzyang@CSE.Buffalo.EDU`

² Department of Computer Science
Stony Brook University, Stony Brook, NY 11794, U.S.A.

`kifer@CS.StonyBrook.EDU`

Abstract. Rule-based and object-oriented techniques are rapidly making their way into the infrastructure for representing and reasoning about semantic information on the Web. Combining these two paradigms has been an important objective and F-logic is a widely adopted formalism that achieves this goal. However, the original F-logic was lacking the notion of *instance methods* — one of the most common object-oriented modeling tools. Extending F-logic with instance methods poses new, non-trivial problems. It requires a different kind of nonmonotonic inheritance and impacts much of the semantics of the logic. In this paper we incorporate instance methods into F-logic and develop a complete model theory as well as a computation framework for the extended language.

1 Introduction

From its humble beginning in RDF, the Semantic Web effort has progressed to Description Logic based languages such as DAML+OIL and OWL. More recent initiatives, such as RuleML [19] and DAML Rules [7], testify to the intense interest in extending the Semantic Web with rule-based capabilities. Object-oriented modeling of semantic information on the Web has had strong appeal since very early on. This is perhaps most pronounced in the influential OntoBroker project [2] and its subsequent commercialization by `ontoprise.com`, which utilizes F-logic [12] — a logic which extends the classical predicate calculus with object-oriented concepts and meta-data programming facilities, and provides a foundation for modeling ontologies and rules in one natural framework. The object-oriented approach was also prominent in the design of the OIL language [1], although this particular aspect received lower priority in the combined DAML+OIL language [8, 9]. Finally, object-oriented approaches, specifically, F-logic based systems, are very popular with the efforts that combine the Web with rule-based reasoning (cf. `FLORA-2` [23, 24], `TRIPLE` [20], `XPathLog` [14], and `FLORID` [3]).

One major difficulty with rule-based object-oriented languages is the semantics of inheritance, especially the issues related to overriding and conflict resolution [12, 25]. A recent study [13] shows that inheritance — especially multiple

* This work was supported in part by NSF grant IIS-0072927.

inheritance — permeates RDF schemas developed by various communities over the past few years. However, neither RDF nor OWL supports inheritance overriding or conflict resolution (in fact, they support no default reasoning at all, which many feel is a serious limitation). As shown in [12, 25], the difficulty in defining a semantics for inheritance is due to the intricate interaction between inferences made by inheritance and via rules (Section 3 shows a simple yet practical example). Unfortunately, this aspect received no satisfactory solution in the original work on F-logic [12], and subsequent works tried to either justify the original solution or impose unreasonable restrictions on the language [16, 11, 15].

Our earlier work [25] proposed a solution to the above problem by developing a semantics that is both theoretically sound and computationally feasible. However, this semantics (like the one in [12] and most other related works) is restricted to the so called *class methods* [18] (or *static methods* in Java terminology). The notion of *instance methods* — a much more important object-oriented modeling tool — was not supported in the language or its semantics. In this paper we rectify this drawback by introducing instance methods and a new kind of inheritance, called *code inheritance*, into F-logic. Our main contribution is the development of a complete model theory and computation framework for non-monotonic multiple inheritance. Limitation of space does not allow us to define the semantics in full generality, but the reader is referred to [22] for details.

This paper is organized as follows. Section 2 introduces the basic F-logic syntax that is used throughout the paper. Section 3 motivates the research problems concerning inheritance and rules by presenting a motivating example. The new three-valued semantics for F-logic is introduced in Section 4. Section 5 defines the inheritance postulates and Section 6 formalizes the associated notion of object models. Section 7 develops the optimistic object model semantics and discusses its properties and implementation. Finally, Section 8 concludes the paper.

2 Preliminaries

F-logic provides natural and powerful syntax as well as semantics for modeling and reasoning about semantic data. In particular, classes, methods, and values are all treated as objects. For instance, the same object, *e.g.*, *ostrich*, can be viewed as a class in one context (with members such as *tweety* and *fred*) and as a member of another (second-order) class (*e.g.*, *species*) in another context.

To save space, we use a subset of the F-logic syntax in this paper, which includes only three kinds of *atomic* formulas: A formula of the form $o:c$ says that object o is a member of class c ; $s::c$ says that class s is a (not necessarily immediate) subclass of class c ; and $e[m \rightarrow v]$ says that object e has an inheritable set-valued method, m , whose result is a set that contains object v . The symbols o , c , s , e , m , and v here are the usual first-order terms.

Traditional object-oriented languages normally distinguish between *instance methods* and *class methods*. The former characterize all instances of a class while the latter characterize classes themselves as objects [18]. Class methods are analogous to “static” methods in the Java language and instance methods correspond

to the other nonstatic (instance) methods. In object-oriented data modeling especially in the case of semistructured objects on the Web it is also convenient to *explicitly* define *object methods* for individual objects. These explicitly specified methods override the methods inherited from superclasses. Object methods are similar to class methods except that they are not intended for inheritance. In F-logic both instance and class/object methods are specified using rules.

Let A be an atom. A literal of the form A is called a *positive* literal and $\neg A$ is called a *negative* literal. An F-logic program is a finite set of rules where all variables are *universally* quantified. There are two kinds of rules: value-rules and code-rules. Value-rules were introduced in the original F-logic [12] while introduction of code-rules is one of the contributions of this paper. Generally, value-rules define class membership, subclass relationship, and class/object methods. Code-rules represent pieces of *code* that define instance methods.

A value-rule has the form $H \leftarrow L_1, \dots, L_n$, where $n \geq 0$, H is a positive literal, called the rule *head*, and each L_i is a positive or a negative literal. The conjunction of L_i 's is called the *body* of the rule. A code-rule has the following form: `c[m \rightarrow v] \leftarrow L_1, \dots, L_n` , which is similar to a value-rule except that it is prefixed with the special keyword `code` and its head must specify a method (*i.e.*, it cannot be `o:c` or `s::c`). We will also assume that c is a constant³ and will say that such a code-rule defines the instance method m for the class c .

In the rest of this paper, we will use uppercase names to denote variables and lowercase names to denote constants. A rule with an empty body is called a *fact*. We will call value-rules and code-rules with an empty body value-facts and code-facts, respectively; we will omit “ \leftarrow ” when writing down the facts.

3 A Motivating Example: Building a Pricing Agent

We will now use the simplified F-logic language introduced above to specify a pricing agent that helps an fictitious product vender, Acme International, to stay ahead of competition. The job of this pricing agent is to propose discount prices for sale items based on the following attributes: (i) `compPrice`: competitor's price; (ii) `cost`: Acme's cost of procuring a sales item; (iii) `discPrice`: proposed discount price; (iv) `approved`: a boolean attribute indicating whether a discount offer is valid (`yes`) or not (`no`).

<code>code coltem[discPrice \rightarrow P] \leftarrow coltem[approved \rightarrow yes], coltem[compPrice \rightarrow C], $P = C * (1-10\%)$.</code>	(1)
<code>X:coltem \leftarrow X[compPrice \rightarrow C], $C < 50$.</code>	(2)
<code>coltem[approved \rightarrow yes].</code>	(3)

Fig. 1. Discount Offer Rules

The agent has two modules: one includes discount offer rules (Figure 1) and the other contains loss control rules (Figure 2). The first module calculates dis-

³ The semantics does not require this, but this assumption is appropriate in practice.

count prices while the second audits if those prices make economic sense. Let us first consider these two modules separately.

In Figure 1, the name `coltem` represents the class of commodity items and code-rule (1) encodes the pricing policy for all commodity items. It states that *the discount price of a commodity item is 10% off the competitor's price if this discount offer is approved*. Value-rule (2) says that *a sales item is a commodity item if the competitor's price for it drops below 50 dollars*. Finally, value-fact (3) assigns the value `yes` to the class method `approved` of `coltem`. In light of inheritance, this implies that *discount offers of commodity items are valid by default*.

Suppose our knowledge base also contains the fact `item101[compPrice → 30]`. How do we determine the discount price (the value of the method `discPrice`) for `item101`? First, since value-rule (2) is enabled, we can derive `item101:coltem`. Now that `item101` is known to be an instance of the class `coltem`, it can inherit the definition of the instance method `discPrice` (code-rule (1) in Figure 1) from `coltem`. When inherited, this code-rule is instantiated for `item101` as follows:

$$\begin{aligned} \text{item101}[\text{discPrice} \rightarrow P] &\leftarrow \\ \text{item101}[\text{approved} \rightarrow \text{yes}], \text{item101}[\text{compPrice} \rightarrow C], P &= C * (1-10\%). \end{aligned}$$

i.e., the object ID, `item101`, is substituted for the class ID, `coltem`, which is essentially a *placeholder* that stands for all instances of this class. This substitution corresponds to the so called *late binding* in traditional object-oriented languages like C++ and Java. To determine the value of the method `approved` on `item101`, observe that `approved` is defined as a class method of `coltem` in (3) and hence `item101` can inherit its value. Therefore, we can derive `item101[approved → yes]` and then `item101[discPrice → 27]`.

What if our knowledge base contains the fact `item101[approved → no]` in addition to all the previous information? Since now `item101[approved → no]` *explicitly* assigns the value `no` to the method `approved` of the object `item101`, it *overrides* the inherited value `yes` (from `coltem`). Consequently the body of the above instantiated rule is no longer satisfied and the method `discPrice` is not defined on `item101`. This scenario illustrates the *nonmonotonic* aspect of inheritance — addition of a new fact invalidates a previous deduction.

$X:\text{loltem} \leftarrow X[\text{discPrice} \rightarrow P], X[\text{cost} \rightarrow C], P < C.$	(4)
$\text{loltem}[\text{approved} \rightarrow \text{no}] \leftarrow \text{loltem}[\text{totalLoss} \rightarrow T], T > 10000.$	(5)

Fig. 2. Loss Control Rules

Now let us look at the rules in Figure 2, which are intended to control possible losses due to price cuts. Here `loltem` represents the class of loss items whose membership is defined by value-rule (4). It states that *an item is a loss item if its discount price is lower than its cost*. Value-rule (5) defines a *conditional* class method. It implies that *by default loss items are not subject to discount if the aggregate loss on the sales of loss items exceeds 10000*.

Suppose our knowledge base contains rules (1)–(5) and the following facts, `item101[compPrice → 30]`, `item101[cost → 28]`, and `loltem[totalLoss → 20000]`. It is instructive to see how one can determine the value of the method `discPrice` for

`item101`. As before, value-rule (2) is fired immediately and yields `item101 : coltem`. The body of value-rule (5) is also satisfied, which derives `loltem[approved → no]`. At this point, no other deduction via rules is possible and `item101` is known to belong to the class `coltem` but no other classes. Therefore, it *appears* that inheritance from `coltem` to `item101` should take place, *i.e.*, the discount offer of `item101` should be approved (`item101[approved → yes]`).

As before inheritance from `coltem` to `item101` enables deduction of the fact `item101[discPrice → 27]`. Now this newly derived fact enables value-rule (4), which in turn yields `item101 : loltem`. However, this leads to a contradiction: `item101[approved → yes]` was derived via inheritance assuming `item101` is a member of `coltem` and no other inheritance contradicts this derivation; but based on this assumption we have inferred that `item101` also belongs to `loltem`, which offers a different value, `no`, for the same method `approved` through inheritance!

The above scenario illustrates a very interesting problem: deduction via inheritance and via rules can enable each other; moreover, the original reasons for inheritance may be invalidated by subsequent deductions via rules which were enabled by inheritance. In the example above, our solution is to “withdraw” the inheritance from `coltem` to `item101` and make the method `discPrice` *undefined* on `item101`. This is analogous to reasoning by contradiction in which a reasoner rejects assumptions that lead to contradictions.

Observations

Nonmonotonic Inheritance. Overriding in inheritance leads to nonmonotonic reasoning, since more specific definitions take precedence over more general ones. However, overriding is not the only source of nonmonotonicity here. When an object belongs to multiple incomparable classes, inheritance conflicts can arise and their “canceling” effects can lead to nonmonotonic inheritance as well.

Dynamic Class Hierarchy. A class hierarchy becomes *dynamic* when class membership and/or subclass relationship is defined using rules (*e.g.*, value-rules (2) and (4) in Figures 1 and 2, respectively), because it can only be decided at runtime but not compile time due to satisfiability of these rules. In such cases complex interactions can come into play between deduction via inheritance and via rules. In particular, a “bad” inheritance decision may trigger a chain of deductions via rules which eventually violate the assumptions based on the principles of overriding and multiple inheritance conflict.

Value Inheritance vs. Code Inheritance. Inheritance of instance method definitions (*e.g.*, inheritance of code-rule (1) in Figure 1) is called *code inheritance*, while inheritance of class method definitions (*e.g.*, inheritance of value-fact (3) in Figure 1) is called *value inheritance*. A fundamental difference between value and code inheritance is that the former is *data-dependent* whereas the latter is not. This difference becomes even more apparent when class method definitions are specified using rules. For instance, inheritability of the method `approved` defined by value-rule (5) in Figure 2 hinges on satisfiability of its rule body. Thus, if we had `loltem[totalLoss → 5000]` instead of `loltem[totalLoss → 20000]` then the body of value-rule (5) would not be satisfied, `loltem[approved → no]` would

not be derived, and multiple inheritance conflict would not arise. In contrast, inheritability of instance method definitions is independent of whether a rule body is satisfied or not: it is the code itself that is inherited, not the facts derived through this code. Therefore, if value-rule (5) were a code-rule, multiple inheritance conflict would persist regardless of the actual value of `totalLoss`.

4 Three-Valued Semantics

The motivating example in Section 3 has illustrated the complicated interactions between deduction via inheritance and rules. This suggests the use of the stable model semantics [6] or the well-founded semantics [5] as the basis of a reasoning mechanism. In this paper we adopt the latter. Since well-founded models are three-valued and the original F-logic models were two-valued [12], we define a suitable three-valued semantics for F-logic programs first.

Let P be an F-logic program. The *Herbrand universe* of P , denoted \mathcal{HU}_P , consists of all the *ground* (i.e., variable-free) terms constructed using the function symbols and constants found in the program. The *Herbrand instantiation* of P , denoted $ground(P)$, is the set of rules obtained by consistently substituting all the terms in \mathcal{HU}_P for all variables in every rule of P . The *Herbrand base* of P , denoted \mathcal{HB}_P , consists of the following sorts of atoms: $o:c$, $s::c$, $s[m \rightarrow v]_{ex}^s$, $o[m \rightarrow v]_{va}^c$, and $o[m \rightarrow v]_{co}^c$, where o , c , s , m , and v are terms from \mathcal{HU}_P .

A *three-valued interpretation* \mathcal{I} of an F-logic program P is a pair $\langle T; U \rangle$, where T and U are *disjoint* subsets of \mathcal{HB}_P . The set T contains all atoms that are *true* whereas U contains all atoms that are *undefined*. The set F of all atoms that are *false* is defined as $F = \mathcal{HB}_P - (T \cup U)$.

Following [17], we will define the truth valuation functions for atoms, literals, and value-rules. The atoms in \mathcal{HB}_P can take one of the three values: **t**, **f**, and **u**. Intuitively the truth value **u** means possibly true or possible false and so carries more “truth” than the truth value **f**. Therefore, the ordering among truth values is defined as follows: $\mathbf{f} < \mathbf{u} < \mathbf{t}$. An interpretation $\mathcal{I} = \langle T; U \rangle$ can be defined as a truth valuation function on any atom A from \mathcal{HB}_P as follows: (i) $\mathcal{I}(A) = \mathbf{t}$, if $A \in T$; (ii) $\mathcal{I}(A) = \mathbf{u}$, if $A \in U$; (iii) Otherwise, $\mathcal{I}(A) = \mathbf{f}$. Moreover, for any $A_i \in \mathcal{HB}_P$, $1 \leq i \leq n$: $\mathcal{I}(A_1 \wedge \dots \wedge A_n) = \min\{\mathcal{I}(A_i) \mid 1 \leq i \leq n\}$.

We can extend the truth valuation function \mathcal{I} to all value-rules in $ground(P)$. In an interpretation of an F-logic program, atoms of the form $s[m \rightarrow v]_{ex}^s$ capture the idea that $m \rightarrow v$ is *explicitly defined* at s via a value-rule, while atoms of the forms $o[m \rightarrow v]_{va}^c$ and $o[m \rightarrow v]_{co}^c$, where $o \neq c$, capture the idea that object o inherits $m \rightarrow v$ from class c by value and code inheritance, respectively.

The intuitive reading of a value-rule is as follows: its rule head acts as an *explicit definition* while its rule body as a *query*. In particular, if $s[m \rightarrow v]$ is in the head of a value-rule and the body of this rule is satisfied, then $m \rightarrow v$ is explicitly defined for s . However, in the body of a value-rule the literal $s[m \rightarrow v]$ tests whether s has an explicit definition of $m \rightarrow v$, or s inherits $m \rightarrow v$ from some superclass by either value or code inheritance. Therefore, the truth valua-

tion of a ground F-logic literal depends on whether it appears a rule head or in a rule body. The above discussion is formalized as follows.

Definition 1. *Given an interpretation \mathcal{I} of an F-logic program P , the truth valuation functions, \mathcal{V}_T^h and \mathcal{V}_T^b (h and b stand for head and body, respectively), on ground F-logic literals are defined as follows: (i) $\mathcal{V}_T^h(o:c) = \mathcal{I}(o:c)$; (ii) $\mathcal{V}_T^h(s::c) = \mathcal{I}(s::c)$; (iii) $\mathcal{V}_T^h(s[m \rightarrow v]) = \mathcal{I}(s[m \rightarrow v]_{ex}^s)$; (iv) $\mathcal{V}_T^b(o:c) = \mathcal{I}(o:c)$; (v) $\mathcal{V}_T^b(s::c) = \mathcal{I}(s::c)$; (vi) $\mathcal{V}_T^b(o[m \rightarrow v]) = \max\{\mathcal{I}(o[m \rightarrow v]_{ex}^o), \mathcal{I}(o[m \rightarrow v]_{va}^c), \mathcal{I}(o[m \rightarrow v]_{co}^c) \mid c \in \mathcal{HU}_P\}$. Let L and L_i ($1 \leq i \leq n$) be ground literals. Then: (i) $\mathcal{V}_T^b(\neg L) = \neg \mathcal{V}_T^b(L)$; (ii) $\mathcal{V}_T^b(L_1 \wedge \dots \wedge L_n) = \min\{\mathcal{V}_T^b(L_i) \mid 1 \leq i \leq n\}$; where $\neg \mathbf{f} = \mathbf{t}$, $\neg \mathbf{u} = \mathbf{u}$, and $\neg \mathbf{t} = \mathbf{f}$.*

With the truth valuation functions \mathcal{V}_T^h and \mathcal{V}_T^b for ground literals, we can define the truth valuation function \mathcal{I} on ground value-rules. We should point out that although \mathcal{I} is three-valued when applied to ground atoms, it becomes *two-valued* when applied to ground value-rules. Intuitively, a ground value-rule is evaluated to be true if and only if the truth value of its rule head is greater than or equal to that of its rule body. Formally, we have the following definition.

Definition 2. *Given an interpretation \mathcal{I} of an F-logic program P , the truth valuation function \mathcal{I} on a ground value-rule, $H \leftarrow B$, in $ground(P)$, is defined as follows: $\mathcal{I}(H \leftarrow B) = \mathbf{t}$, if $\mathcal{V}_T^h(H) \geq \mathcal{V}_T^b(B)$; otherwise, $\mathcal{I}(H \leftarrow B) = \mathbf{f}$. Similarly, given a ground value-fact, H , in $ground(P)$: $\mathcal{I}(H) = \mathbf{t}$ iff $\mathcal{V}_T^h(H) = \mathbf{t}$.*

We will say that a three-valued interpretation satisfies the value-rules of an F-logic program, if it satisfies all the ground value-rules of this program.

Definition 3 (Value-Rule Satisfaction). *A three-valued interpretation \mathcal{I} satisfies the value-rules of an F-logic program P , if $\mathcal{I}(R) = \mathbf{t}$ for every value-rule R in $ground(P)$.*

5 Inheritance Postulates

Even if an interpretation \mathcal{I} satisfies all the value-rules of an F-logic program P , it does not necessarily mean that \mathcal{I} is an intended *object model* of P , because \mathcal{I} must also include facts that are derived via inheritance. F-logic programs only specify class hierarchies and method definitions — what needs to be inherited is not explicitly stated. In fact, as we saw in Section 3, defining exactly what should be inherited is a subtle issue. In our framework, it is the job of the *inheritance postulates*, which embody the common intuition behind inheritance.

First we will introduce the notion of inheritance candidacy. The various concepts to be defined in this section come with two flavors: *strong* or *weak*. The “strong” flavor of a concept requires that all relevant facts be positively established while the “weak” flavor allows some or all facts to be undefined.

Definition 4 (Explicit Definition). *Given an interpretation \mathcal{I} of an F-logic program P , $s[m]$ is a strong explicit definition, if $\max\{\mathcal{I}(s[m \rightarrow v]_{ex}^s) \mid v \in \mathcal{HU}_P\} = \mathbf{t}$; $s[m]$ is a weak explicit definition if $\max\{\mathcal{I}(s[m \rightarrow v]_{ex}^s) \mid v \in \mathcal{HU}_P\} = \mathbf{u}$.*

Definition 5 (Value Inheritance Context). Given an interpretation \mathcal{I} of an F -logic program P , $c[m]$ is a strong value inheritance context for o , if $c \neq o$ ⁴ and $\min\{\mathcal{I}(o:c), \max\{c[m] \rightarrow v\}_{\text{ex}}^c | v \in \mathcal{HU}_P\} = \mathbf{t}$; $c[m]$ is a weak value inheritance context for o , if $c \neq o$ and $\min\{\mathcal{I}(o:c), \max\{c[m] \rightarrow v\}_{\text{ex}}^c | v \in \mathcal{HU}_P\} = \mathbf{u}$ (roughly speaking, if o is a proper member of c and $c[m]$ is an explicit definition).

Definition 6 (Code Inheritance Context). Given an interpretation \mathcal{I} of an F -logic program P , $c[m]$ is a strong (weak) code inheritance context for o , if $c \neq o$, $\mathcal{I}(o:c) = \mathbf{t}$ ($\mathcal{I}(o:c) = \mathbf{u}$), and there is a code-rule in P which specifies the instance method m for the class c .

Note that explicit definitions can only be established via value-rules but not code-rules. The difference between a value and a code inheritance context is that the former requires at least one value be established for its class method via a value-rule, whereas the latter only requires the presence of at least one code-rule which specifies its instance method. Generally we will use the term *inheritance context* to refer to either a value or a code inheritance context. In the following definitions we will see that value and code inheritance contexts are treated equally as far as overriding is concerned.

Definition 7 (Overriding). Given an interpretation \mathcal{I} of an F -logic program P , the class s strongly overrides $c[m]$ for o , if $s \neq c$, $\mathcal{I}(s::c) = \mathbf{t}$, and $s[m]$ is either a strong value or a strong code inheritance context for o .

The class s weakly overrides $c[m]$ for o if the above conditions are relaxed by allowing $s::c$ to be undefined and/or allowing $s[m]$ to be a weak inheritance context. Formally this means that either: (i) $\mathcal{I}(s::c) = \mathbf{t}$ and $s[m]$ is a weak inheritance context for o ; or (ii) $\mathcal{I}(s::c) = \mathbf{u}$ and $s[m]$ is either a weak or a strong inheritance context for o .

Definition 8 (Value Inheritance Candidate). Given an interpretation \mathcal{I} of an F -logic program P , $c[m]$ is a strong value inheritance candidate for o , denoted $c[m] \overset{sv}{\rightsquigarrow}_{\mathcal{I}} o$, if $c[m]$ is a strong value inheritance context for o and there is no s that strongly or weakly overrides $c[m]$ for o .

$c[m]$ is a weak value inheritance candidate for o , denoted $c[m] \overset{wv}{\rightsquigarrow}_{\mathcal{I}} o$, if the above conditions are relaxed by allowing $c[m]$ to be a weak value inheritance context and/or allowing weak overriding. Formally, this means that there is no s that strongly overrides $c[m]$ for o and either: (i) $c[m]$ is a weak value inheritance context for o ; or (ii) $c[m]$ is a strong value inheritance context for o and there is s that weakly overrides $c[m]$ for o .

Definition 9 (Code Inheritance Candidate). Given an interpretation \mathcal{I} of an F -logic program P , $c[m]$ is a strong code inheritance candidate for o , denoted $c[m] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} o$, if $c[m]$ is a strong code inheritance context for o and there is no s that strongly or weakly overrides $c[m]$ for o .

$c[m]$ is a weak code inheritance candidate for o , denoted $c[m] \overset{wc}{\rightsquigarrow}_{\mathcal{I}} o$, if the above conditions are relaxed by allowing $c[m]$ to be a weak code inheritance context and/or allowing weak overriding. Formally, this means that there is no s

⁴ $c \neq o$ means that c and o are distinct terms.

that strongly overrides $c[m]$ for o and either: (i) $c[m]$ is a weak code inheritance context for o ; or (ii) $c[m]$ is a strong code inheritance context for o and there is s that weakly overrides $c[m]$ for o .

Example 1. As an example, consider an interpretation $\mathcal{I} = \langle T; U \rangle$ of an F-logic program P , where $T = \{c_1 : c_2, c_1 : c_4, c_1 : c_5, c_2 :: c_4, c_3 :: c_5\} \cup \{c_2[m \rightarrow a]_{ex}^c, c_3[m \rightarrow b]_{ex}^c, c_4[m \rightarrow c]_{ex}^c\}$ and $U = \{c_1 : c_3\}$. \mathcal{I} and P are shown in Figure 3, where solid and dashed arrows represent true and undefined values, respectively.

In the interpretation \mathcal{I} , $c_2[m]$ and $c_4[m]$ are strong value inheritance contexts for c_1 . $c_5[m]$ is a strong code inheritance context for c_1 . $c_3[m]$ is a weak value inheritance context for c_1 . The class c_2 strongly overrides $c_4[m]$ for c_1 , while c_3 weakly overrides $c_5[m]$ for c_1 . The context $c_2[m]$ is a strong value inheritance candidate for c_1 , while $c_3[m]$ is a weak value inheritance candidate and $c_5[m]$ is a weak code inheritance candidate for c_1 . Finally, $c_4[m]$ is neither a strong nor a weak value inheritance candidate for c_1 .

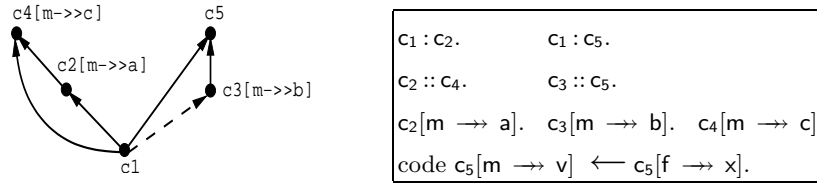


Fig. 3. Inheritance Context, Overriding, and Inheritance Candidate

For convenience, we will simply write $c[m] \rightsquigarrow_{\mathcal{I}} o$ when it does not matter whether $c[m]$ is a strong or a weak value or code inheritance candidate. Now we are ready to introduce the postulates for nonmonotonic multiple value and code inheritance. The inheritance postulates consist of two parts: core inheritance postulates and optimistic inheritance postulates. We formalize the core inheritance postulates first.

Definition 10 (Positive ISA Transitivity). *An interpretation \mathcal{I} of an F-logic program P satisfies the positive ISA transitivity constraint if the positive part of the class hierarchy is transitively closed, formally, if the following two conditions hold: (1) for all s, c : if there is x such that $\mathcal{I}(s :: x) = t$ and $\mathcal{I}(x :: c) = t$, then $\mathcal{I}(s :: c) = t$; (2) for all o, c : if there is x such that $\mathcal{I}(o : x) = t$ and $\mathcal{I}(x :: c) = t$, then $\mathcal{I}(o : c) = t$.*

Definition 11 (Context Consistency). *An interpretation \mathcal{I} of an F-logic program P satisfies the context consistency constraint, if the following conditions hold: (1) for all o, m, v : $\mathcal{I}(o[m \rightarrow v]_{va}^o) = f$ and $\mathcal{I}(o[m \rightarrow v]_{co}^o) = f$; (2) for all c, m, v : if $\mathcal{I}(c[m \rightarrow v]_{ex}^c) = f$, then $\mathcal{I}(o[m \rightarrow v]_{va}^c) = f$ for all o ; (3) for all c, m : if there is no code-rule in $\text{ground}(P)$ which specifies the instance method m for the class c , then $\mathcal{I}(o[m \rightarrow v]_{co}^c) = f$ for all o, v ; (4) for all o, m : if $o[m]$ is a strong explicit definition, then $\mathcal{I}(o[m \rightarrow v]_{va}^c) = f$ and $\mathcal{I}(o[m \rightarrow v]_{co}^c) = f$ for all v, c .*

The context consistency constraint captures the implications of specificity. The first condition rules out self inheritance. The second condition states that if $m \rightarrow v$ is not explicitly defined at c , then no class should inherit $m \rightarrow v$ from c by value inheritance. The third condition states that if a class c does not specify an instance method m , then no object should inherit any value of m from c by code inheritance. The fourth condition states that if $m \rightarrow v$ is explicitly defined at o , then this definition should prevent o from inheriting any value of m from other classes (by either value or code inheritance).

The following constraint captures the semantics of nonmonotonic multiple inheritance.

Definition 12 (Unique Source Inheritance). *An interpretation \mathcal{I} of an F -logic program P satisfies the unique source inheritance constraint, if all of the following three conditions hold: (1) for all o, m, v, c : if $\mathcal{I}(o[m \rightarrow v]_{va}^c) = \mathbf{t}$ or $\mathcal{I}(o[m \rightarrow v]_{co}^c) = \mathbf{t}$, then $\mathcal{I}(o[m \rightarrow z]_{va}^x) = \mathbf{f}$ and $\mathcal{I}(o[m \rightarrow z]_{co}^x) = \mathbf{f}$ for all z, x where $x \neq c$; (2) for all c, m, o : if $c[m] \overset{sv}{\rightsquigarrow}_{\mathcal{I}} o$ or $c[m] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} o$, then $\mathcal{I}(o[m \rightarrow v]_{va}^x) = \mathbf{f}$ and $\mathcal{I}(o[m \rightarrow v]_{co}^x) = \mathbf{f}$ for all v, x such that $x \neq c$; (3) for all o, m, v, c : $\mathcal{I}(o[m \rightarrow v]_{va}^c) = \mathbf{t}$ iff (i) $o[m]$ is neither a strong nor a weak explicit definition; and (ii) $c[m] \overset{sv}{\rightsquigarrow}_{\mathcal{I}} o$; and (iii) $\mathcal{I}(c[m \rightarrow v]_{ex}^c) = \mathbf{t}$; and (iv) there is no x such that $x \neq c$ and $x[m] \rightsquigarrow_{\mathcal{I}} o$.*

Intuitively, we want our semantics for inheritance to have such a property that if inheritance is allowed, then it should take place from a *unique* source. The first condition above implies that positive value or code inheritance from a class should prevent any inheritance from other classes. The second condition states that if a strong value or code inheritance candidate, $c[m]$, exists, then inheritance of the method m cannot take place from any other source (because there would be a multiple inheritance conflict). The third condition specifies when “positive” value inheritance takes place. An object o inherits $m \rightarrow v$ from a class c by value inheritance iff: (i) no value is explicitly defined for the method m at o ; (ii) $c[m]$ is a strong value inheritance candidate for o ; (iii) $m \rightarrow v$ is explicitly defined at c and is positive; and (iv) there are no other inheritance candidates — weak or strong — from which o could inherit the method m .

We should point out that the constraints introduced so far only capture the intuition behind the “definite” part of an object model (the notion of an object model is formalized in Section 6), *i.e.*, the true and the false components. We view them as *core inheritance postulates* that any reasonable object model must obey. However, we still need to assign a meaning to the undefined part of an object model. Since “undefined” means possibly true or possibly false, intuitively we want the conclusions drawn from undefined facts to remain undefined, *i.e.*, the semantics should be “closed” with respect to undefined facts. Therefore, although it might seem tempting to “jump” to negative conclusions from undefined facts in some cases (*e.g.*, if there are multiple weak inheritance candidates), our semantics is biased towards undefined conclusions, which is why we call it “optimistic”. Due to want of space, we will omit the optimistic inheritance postulates here. Their definitions can be found in [22].

6 Object Models

A model of an F-logic program should satisfy all the rules in it. In Section 4 we have formalized the notion of value-rule satisfaction. Here we will extend this notion to code-rules. First note that when an object inherits an instance method definition, *i.e.*, a code-rule, it will be evaluated in the context of this object. This corresponds to the idea of *late binding* in imperative object-oriented languages like C++ and Java.

Definition 13 (Binding). *Let $R \equiv (\text{code } c[m \rightarrow v] \leftarrow B)$ be a ground code-rule which specifies the instance method m for the class c . The binding of R with respect to an object o , denoted $R||o$, is obtained from R by substituting o for every occurrence of c in R . We will use $X_{c \setminus o}$ to represent the term that is obtained from X by substituting o for every occurrence of c in X .*

Therefore, the truth valuation function will be defined on *bindings* of ground code-rules instead of on code-rules directly. When an object inherits code-rules from a class, the bindings of these code-rules with respect to this object should be satisfied similarly to value-rules. However, because only those code-rules which are inherited need to be satisfied, satisfaction of code-rules depends on how they are inherited: strongly or weakly.

Definition 14 (Strong Code Inheritance). *Let \mathcal{I} be an interpretation of an F-logic program P and $R \equiv (\text{code } c[m \rightarrow v] \leftarrow B)$ a code-rule in $\text{ground}(P)$. An object o strongly inherits R , if the following conditions hold: (1) $c[m] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} o$; (2) $o[m]$ is neither a strong nor a weak explicit definition; (3) there is no $x \neq c$ such that $x[m] \rightsquigarrow_{\mathcal{I}} o$.*

Definition 15 (Weak Code Inheritance). *Let \mathcal{I} be an interpretation of an F-logic program P and $R \equiv (\text{code } c[m \rightarrow v] \leftarrow B)$ a code-rule in $\text{ground}(P)$. An object o weakly inherits R , if the following conditions hold: (1) $c[m] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} o$ or $c[m] \overset{uc}{\rightsquigarrow}_{\mathcal{I}} o$; (2) $o[m]$ is not a strong explicit definition; (3) there is no $x \neq c$ such that $x[m] \overset{su}{\rightsquigarrow}_{\mathcal{I}} o$ or $x[m] \overset{sc}{\rightsquigarrow}_{\mathcal{I}} o$; (4) o does not strongly inherit R .*

We can define a function, $\text{imode}_{\mathcal{I}}$, on bindings of ground code-rules, which returns the “inheritance mode” of a binding: (i) $\text{imode}_{\mathcal{I}}(R||o) = \mathbf{t}$, if o strongly inherits R ; (ii) $\text{imode}_{\mathcal{I}}(R||o) = \mathbf{u}$, if o weakly inherits R ; (iii) $\text{imode}_{\mathcal{I}}(R||o) = \mathbf{f}$, otherwise. Now we can extend the truth valuation function to ground code-rules as follows.

Definition 16. *Let \mathcal{I} be an interpretation, $R \equiv (\text{code } c[m \rightarrow v] \leftarrow B)$ be a ground code-rule and $F \equiv (\text{code } c[m \rightarrow v])$ a ground C-fact. The truth valuation function \mathcal{I} on $R||o$ and $F||o$ (the bindings of R and F with respect to o , respectively) is defined as follows:*

$$\mathcal{I}(R||o) = \begin{cases} \mathbf{t}, & \text{if } \text{imode}_{\mathcal{I}}(R||o) \geq \mathbf{u} \text{ and} \\ & \mathcal{I}(o[m \rightarrow v]_{c \setminus o}) \geq \min\{\mathcal{V}_{\mathcal{I}}^b(B_{c \setminus o}), \text{imode}_{\mathcal{I}}(R||o)\}; \\ \mathbf{t}, & \text{if } \text{imode}_{\mathcal{I}}(R||o) = \mathbf{f} \text{ and } \mathcal{I}(o[m \rightarrow v]_{c \setminus o}) = \mathbf{f}; \\ \mathbf{f}, & \text{otherwise.} \end{cases}$$

$$\mathcal{I}(F||\mathfrak{o}) = \begin{cases} \mathbf{t}, & \text{if } \text{imode}_{\mathcal{I}}(R||\mathfrak{o}) \geq \mathbf{u} \text{ and } \mathcal{I}(\mathfrak{o}[m \rightarrow v]_{\text{co}}^c) \geq \text{imode}_{\mathcal{I}}(R||\mathfrak{o}); \\ \mathbf{t}, & \text{if } \text{imode}_{\mathcal{I}}(R||\mathfrak{o}) = \mathbf{f} \text{ and } \mathcal{I}(\mathfrak{o}[m \rightarrow v]_{\text{co}}^c) = \mathbf{f}; \\ \mathbf{f}, & \text{otherwise.} \end{cases}$$

Recall that atoms of the form $\mathfrak{o}[m \rightarrow v]_{\text{co}}^c$ represent those facts that are derived via code inheritance. Moreover, observe that in the case of strong code inheritance, the truth valuation function on code-rules will be defined essentially the same way as on value-rules. Now the idea of code-rule satisfaction and the notion of an *object model* can be formalized as follows.

Definition 17 (Code-Rule Satisfaction). *A three-valued interpretation \mathcal{I} satisfies the code-rules of an F-logic program P , if $\mathcal{I}(R||\mathfrak{o}) = \mathbf{t}$ for all code-rule $R \in \text{ground}(P)$ and all $\mathfrak{o} \in \mathcal{H}\mathcal{U}_P$.*

Definition 18 (Object Model). *An interpretation \mathcal{I} is called an object model of an F-logic program P , if \mathcal{I} satisfies both the value-rules and the code-rules in P , plus the core inheritance postulates: the positive ISA transitivity constraint, the context consistency constraint, and the unique source inheritance constraint.*

7 Optimistic Object Models

In this section we introduce a particular object model, called *optimistic object model*, which is *uniquely* defined for *any* F-logic program and satisfies all the inheritance postulates defined in Section 5, including the optimistic inheritance postulates. First we will present a *procedural* characterization of optimistic object models. Due to space limitation, here we will only sketch the main concepts. The reader is referred to [22] for more details.

Computation of optimistic object models extends the alternating fixpoint process in [4]. The new element here is the book-keeping mechanism for recording inheritance information. This book-keeping information will be projected out when the final object model is generated. The main component of the computation is an *antimonotonic* operator, Ψ_P , which takes an F-logic program P as input and performs implicit deduction via value and code inheritance as well as explicit deduction via rules. Based on the operator Ψ_P we define another operator, $\mathbf{F}_P \stackrel{\text{def}}{=} \Psi_P \cdot \Psi_P$, which is *monotonic* and hence has a unique least fixpoint, denoted $\text{lfp}(\mathbf{F}_P)$. The definition of optimistic object models is stated as follows.

Definition 19 (Optimistic Object Model). *The optimistic object model, \mathcal{M} , of an F-logic program P is defined as follows: $\mathcal{M} = \langle \mathbb{T}; \mathbb{U} \rangle$, where $\mathbb{T} = \text{lfp}(\mathbf{F}_P)$ and $\mathbb{U} = \Psi_P(\text{lfp}(\mathbf{F}_P)) - \text{lfp}(\mathbf{F}_P)$.*

Theorem 1. *The optimistic object model \mathcal{M} of an F-logic program P is an object model of P . Moreover, it satisfies the optimistic inheritance postulates.*

It turns out that the (unique) optimistic object model of an F-logic program P can be computed as the well-founded model of a certain general logic program with negation, which is obtained from P by rewriting. Before describing the rewriting procedure we first define a rewriting function that applies to all value-rules and code-rules.

Definition 20. Given an F-logic program P and a literal L in P , the functions ρ^h and ρ^b that rewrite head and body literals in P are defined as follows:

$$\rho^h(L) = \begin{cases} isa(o, c), & \text{if } L = o : c \\ sub(s, c), & \text{if } L = s :: c \\ exmv(s, m, v), & \text{if } L = s[m \rightarrow v] \end{cases} \quad \rho^b(L) = \begin{cases} isa(o, c), & \text{if } L = o : c \\ sub(s, c), & \text{if } L = s :: c \\ mv(o, m, v), & \text{if } L = o[m \rightarrow v] \\ \neg(\rho^b(G)), & \text{if } L = \neg G \end{cases}$$

The rewriting function ρ on value-rules and code-rules in P is defined as follows:

$$\rho(H \leftarrow L_1, \dots, L_n) = \rho^h(H) \leftarrow \rho^b(L_1), \dots, \rho^b(L_n)$$

$$\rho(\text{code } c[m \rightarrow v] \leftarrow L_1, \dots, L_n) = \text{ins}(O, m, v, c) \leftarrow \rho^b(B_1), \dots, \rho^b(B_n)$$

where O is a new variable that does not appear in P and each $B_i = (L_i)_{c \setminus O}$ B_i is obtained from L_i by substituting O for all occurrences of c . The predicates *isa*, *sub*, *exmv*, *mv*, and *ins* are auxiliary predicates introduced by the rewriting.

Note that because literals in rule heads and bodies have different meanings, they are rewritten differently. Moreover, literals in the heads of value-rules and in the heads of code-rules are also rewritten differently. The rewriting procedure that transforms F-logic programs into general logic programs is defined below.

Definition 21 (Well-Founded Rewriting). The well-founded rewriting of an F-logic program P , denoted P^{wf} , is a general logic program constructed by the following steps: (1) For every value-rule R in P , add its rewriting $\rho(R)$ into P^{wf} ; (2) For every code-rule R in P , which specifies an instance method m for a class c , add its rewriting $\rho(R)$ into P^{wf} . Moreover, add a fact *codedef*(c, m) into P^{wf} ; (3) Include the trailer rules shown in Figure 4 to P^{wf} (note that uppercase letters denote variables in these trailer rules).

$mv(O, M, V) \leftarrow exmv(O, M, V).$ $mv(O, M, V) \leftarrow vamv(O, M, V, C).$ $mv(O, M, V) \leftarrow comv(O, M, V, C).$ $sub(S, C) \leftarrow sub(S, X), sub(X, C).$ $isa(O, C) \leftarrow isa(O, S), sub(S, C).$ $vamv(O, M, V, C) \leftarrow vacan(C, M, O), exmv(C, M, V), \neg ex(O, M), \neg multi(C, M, O).$ $comv(O, M, V, C) \leftarrow cocan(C, M, O), ins(O, M, V, C), \neg ex(O, M), \neg multi(C, M, O).$ $vacan(C, M, O) \leftarrow isa(O, C), exmv(C, M, V), C \neq O, \neg override(C, M, O).$ $cocan(C, M, O) \leftarrow isa(O, C), codedef(C, M), C \neq O, \neg override(C, M, O).$ $ex(O, M) \leftarrow exmv(O, M, V).$ $multi(C, M, O) \leftarrow vacan(X, M, O), X \neq C.$ $multi(C, M, O) \leftarrow cocan(X, M, O), X \neq C.$ $override(C, M, O) \leftarrow sub(X, C), isa(O, X), exmv(X, M, V), X \neq C, X \neq O.$ $override(C, M, O) \leftarrow sub(X, C), isa(O, X), codedef(X, M), X \neq C, X \neq O.$
--

Fig. 4. Trailer Rules for Well-Founded Rewriting

Note that while rewriting an F-logic program into a general logic program, we need to output facts of the form *codedef*(c, m) to remember that there is a code-rule specifying the instance method m for the class c . Such facts are used to derive overriding and code inheritance candidacy information.

There is a unique *well-founded model* for any general logic program [5]. Let P^{wf} be the well-founded rewriting of an F-logic program P . We can define an *isomorphism* between the well-founded model of P^{wf} and the optimistic object model of P based on an one-to-one mapping between the following atoms: (i) $isa(o, c)$ and $o : c$; (ii) $sub(s, c)$ and $s :: c$; (iii) $exmv(s, m, v)$ and $s[m \rightarrow v]_{ex}^s$; (iv) $vamv(o, m, v, c)$ and $o[m \rightarrow v]_{va}^c$; (v) $comv(o, m, v, c)$ and $o[m \rightarrow v]_{co}^c$. Note that while defining the isomorphism we have projected out other book-keeping information for inheritance. As stated in the following theorem, We have proved that these two models are indeed isomorphic.

Theorem 2. *Given the well-founded rewriting P^{wf} of an F-logic program P , the well-founded model of P^{wf} is isomorphic to the optimistic object model of P .*

Clearly, given an F-logic program P , generation of P^{wf} takes time linear in the size of P . Note that the trailer rules in Figure 4 are fixed for an given F-logic program. Therefore, the size of P^{wf} is also linear in the size of the original F-logic program P . This observation combined with Theorem 2 essentially leads to the following claim about the data complexity [21] of our inheritance semantics.

Corollary 1. *The data complexity of the optimistic object model semantics for function-free F-logic programs is polynomial time.*

Theorems 1 and 2 give procedural characterization of the optimistic object model as the least fixpoint of extended alternating fixpoint computation. Next we will present a different characterization of the optimistic object model semantics based on the so called *truth ordering* among object models.⁵

It is common to compare different models of a program based on the amount of “truth” contained in the models. Typically, the true component of a model is minimized and the false component maximized. However, in F-logic we also need to deal with inheritance, which complicates the matters somewhat, because a fact may be derived via inheritance. As a consequence, there exist object models that look similar but actually are incomparable. This leads to the following definition of truth ordering among object models, which minimizes not only the set of true atoms of an object model, but also the amount of positive inheritance information implied by the object model.

Definition 22 (Truth Ordering). *Let $\mathcal{I}_1 = \langle P_1; Q_1 \rangle$ and $\mathcal{I}_2 = \langle P_2; Q_2 \rangle$ be two object models of an F-logic program P . We write $\mathcal{I}_1 \leq \mathcal{I}_2$ iff (i) $P_1 \subseteq P_2$; and (ii) $P_1 \cup Q_1 \subseteq P_2 \cup Q_2$; and (iii) for all c, m, o : $c[m] \overset{sv}{\sim}_{\mathcal{I}_1} o$ implies $c[m] \overset{sv}{\sim}_{\mathcal{I}_2} o$; and (iv) for all c, m, o : $c[m] \overset{sc}{\sim}_{\mathcal{I}_1} o$ implies $c[m] \overset{sc}{\sim}_{\mathcal{I}_2} o$.*

Definition 23 (Minimal Object Model). *An object model \mathcal{I} is minimal iff there exists no object model \mathcal{J} such that $\mathcal{J} \leq \mathcal{I}$ and $\mathcal{J} \neq \mathcal{I}$.*

⁵ We can also define *stable* object models and a different ordering, called *information ordering*, among object models. We have also shown that the optimistic object model of an F-logic program is the *least* stable object model of this program with respect to information ordering. See [22] for more details.

The above definitions minimize the number of strong inheritance candidates implied by an object model *in addition to* the usual minimization of truth and maximization of falsehood. This is needed because increasing the number of false facts might inflate the number of strong inheritance candidates (due to nonmonotonic inheritance), which in turn might inflate the number of facts that are derived by inheritance. Nevertheless it turns out that the optimistic object models are minimal.

Theorem 3. *The optimistic object model of an F-logic program P is minimal among the object models of P that satisfy the optimistic inheritance constraints.*

8 Conclusion and Future Work

We presented a new and natural model theory for nonmonotonic multiple value and code inheritance and its implementation using a deductive engine that supports well-founded semantics [5]. The problem described in this paper also arises in the design of rule-based object-oriented languages for the Semantic Web. We illustrated the practical nature of the problem as well as the difficulties in finding a proper solution.

Our model-theoretic approach points to several interesting research directions. First, the proposed semantics for inheritance is *source-based*, since in determining multiple inheritance conflict the semantics only considers whether the same method is *defined* at different inheritance sources. A conflict is declared even if the *set* of return values (*i.e.*, *content*) of this method is equivalent at these sources. Intuitively content-based inheritance has a “higher order” flavor and we plan to study its impact on computational efficiency in the future.

Second, it has been argued that inheritance-like phenomena arise in many applications such as discretionary access control and trust management [10], but they cannot be formalized using a single semantics. We are working on extensions to our framework, which allow users to specify their own *ad hoc* inheritance policies in a programmable, yet declarative, way.

References

1. S. Decker, S. Melnik, F. V. Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The Semantic Web: The roles of XML and RDF. *IEEE Internet Computing*, 15(3):63–74, October 2000.
2. D. Fensel, S. Decker, M. Erdmann, and R. Studer. Ontobroker: Or how to enable intelligent access to the WWW. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1998.
3. J. Frohn, R. Himmeröder, G. Lausen, W. May, and C. Schleppehorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998.
4. A. V. Gelder. The alternating fixpoint of logic programs with negation. In *ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

5. A. V. Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, July 1991.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
7. B. N. Grosz, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *International World Wide Web Conference*, 2003.
8. I. Horrocks. DAML+OIL: A description logic for the Semantic Web. *IEEE Bulletin of the Technical Committee on Data Engineering*, 25(1), March 2002.
9. I. Horrocks and S. Tessaris. Querying the semantic web: A formal approach. In *International Semantic Web Conference*, 2002.
10. S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, June 2001.
11. H. M. Jamil. Implementing abstract objects with inheritance in Datalog^{neg}. In *International Conference on Very Large Data Bases*, pages 56–65, 1997.
12. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.
13. A. Magkanaraki, S. Alexaki, V. Christophides, and D. Plexousakis. Benchmarking RDF schemas for the semantic web. In *International Semantic Web Conference*, 2002.
14. W. May. A rule-based querying and updating language for XML. In *International Workshop on Database Programming Languages (DBPL)*, pages 165–181, 2001.
15. W. May and P. Kandzia. Nonmonotonic inheritance in object-oriented deductive database languages. *Journal of Logic and Computation*, 11(4), 2001.
16. W. May, B. Ludäscher, and G. Lausen. Well-founded semantics for deductive object-oriented database languages. In *International Conference on Deductive and Object-Oriented Databases*, pages 320–336. Springer Verlag LNCS, 1997.
17. T. C. Przymusiński. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM Symposium on Principles of Database Systems*, pages 11–21, 1989.
18. F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.
19. The rule markup initiative. <http://www.dfki.uni-kl.de/ruleml/>.
20. M. Sintek and S. Decker. TRIPLE – a query, inference, and transformation language for the semantic web. In *International Semantic Web Conference*, 2002.
21. M. Vardi. The complexity of relational query languages. In *ACM Symposium on Theory of Computing*, pages 137–145, 1982.
22. G. Yang. *A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases*. PhD thesis, SUNY at Stony Brook, December 2002. <http://www.cs.sunysb.edu/~guizyang/>.
23. G. Yang and M. Kifer. Implementing an efficient DOOD system using a tabling logic engine. In *First International Conference on Computational Logic, DOOD'2000 Stream*, July 2000.
24. G. Yang and M. Kifer. FLORA-2: User's Manual. <http://flora.sourceforge.net/>, June 2002.
25. G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *International Conference on Ontologies, DataBases, and Applications of Semantics*, October 2002.