1995

A Logic For Object-oriented Databases

Machmudin Junus

A LOGIC FOR OBJECT-ORIENTED DATABASES

by

Machmudin **Junus**

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
April 1995

ISBN 0-612-03474-7

CONTENTS

ABSTRACT

When E. F. Codd [Codd70] designed relational databases, he made use of mathematical logic concerning Predicate Calculus. However, when object-oriented databases were designed, there was no such support from any established mathematical logic.

In this thesis, we present a logic suitable for reasoning about object-oriented database systems. This logic is the result of the modification of F-logic [KifLa89, KifLaWu90]. The main difference between our proposed logic and F-logic, is the way our logic treats class objects and instance objects.

Our designed logic has the ability to represent: object identities, attributes, methods, classes, class hierarchies and inheritance. Similar to F-logic, the designed logic also has a sound and complete proof procedure based on resolution. In addition, the logic has advantages over F-logic.

Key Words: object-oriented databases, deductive databases, logic programming, formal logic, proof theory.

For my mother and father

ACKNOWLEDGEMENTS

I am deeply grateful to Dr. James K. Mullin, my advisor, for the many patient hours spent on useful discussions of this research and reading the drafts of this thesis, and for partially funding me in my latest stages. Especially, his understanding and constant encouragement has helped me not to give up on this research. I am also deeply grateful to Dr. A. Nait Abdallah, my advisor, for his useful criticism and suggestions on this thesis. The presentation of this thesis has been much improved with his help. I wish to thank Dr. M. Kifer who sent me some of his papers that helped me a lot in initiating the research on this subject several years ago. I also wish to express my gratitude to the examining board: the external examiner, Dr. M. Ozsoyoglu, the extra-departmental examiner, Dr. A. M. Dawes, and the departmental examiners, Dr. H. Jurgensen and Dr. R. E. Mercer, for their helpful criticism and corrections that have improved the writing quality of this thesis.

I am deeply grateful to The Ministry of Education and Culture of the Republic of Indonesia for providing me a scholarship that made it possible to study here, at U.W.O. I am also deeply grateful to Dr. E. W. Elcock for partially funding me through his research grant, and to The Department of Computer Science and The Faculty of Graduate Studies for partially funding me through teaching assistantships and The Special University Scholarship. Finally, I express my love and gratitude to my mother, S. Zaleha Yunus, my father, H. M. Yunus Umar, my wife, Kasiyah, and my daughter, Inas, for their patient and constant support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Currently, many studies are being conducted on object-oriented database systems, both from a practical and a theoretical point of view. One problem that can be quickly recognized about these studies is the lack of a common data model, which can be traced to the lack of a common formal foundation. According to [Maier89, AtBa92], there is no clear specification of an object-oriented database system comparable with the specification of a relational database system proposed by Codd [Codd70].

In this thesis we propose a logic that is suitable for object-oriented databases. Our motivation is to produce a logic which may be used for object-oriented databases in a manner si· ıilar to the way Predicate Calculus was used as a basis for the specification of relational database systems.

1.1 Object-Oriented Databases

Applications of object-oriented (OO) concepts have been important research topics in several areas such as programming languages, databases, knowledge representations and computer architectures. However, it is still not clear what *object oriented* means in general, especially in the context of object-oriented databases. Won Kim [Kim90] tried to formulate the core concepts of object-oriented databases. The main basis for his view of the core concepts in object-oriented databases comes from his experiences with the ORION series [Ban87, Kim88, Kim89]. According to him, the core concepts of OO databases are:

- *Object and Object Identity.*

  This means that in OO systems and languages, every real-world entity is uniformly modeled as an object. Each object is associated with a unique identifier, which is commonly called object identity.

- *Attributes and Methods.*

  Each object has a *state* and a *behaviour*. An object's state is a set of values for the object's attributes. An object's behaviour is the set of methods (program code) which operate on the object's state.

  The value of an attribute can be a single value or a set of values. Although each element of a set is an object, the set itself is not an object. The state and behaviour of an object can only be accessed from outside the object through explicit message passing or function calls.

- *Class.*

  A class is used for grouping all the objects sharing the same set of attributes and methods. Furthermore, an object is allowed to belong to only one class as that class' instance. A class corresponds to the concept of abstract data types or primitive types. A *primitive class* is a class whose instances have no attributes, for example: integer, string and Boolean. The value of an object's attribute also belongs to some class, which is the domain of that object's attribute.

- *Class Hierarchy and Inheritance.*

  A new class can be derived from an existing class, this new class is called a subclass. A subclass inherits all attributes and methods of the existing class. Additional attributes and methods can also be defined for a subclass. A class may have any number of subclasses.

Some OO systems allow a class to have only one superclass. This is called *single inheritance*, because a class inherits attributes and methods from only a single class. In this single-inheritance system, the classes form a tree-like hierarchy which is called a *class hierarchy*. Other OO systems allow a class to have any number of superclasses. This is called *multiple inheritance*, because a class inherits attributes and methods from more than one class. In this multiple-inheritance system, the classes form a rooted and connected directed acyclic graph (DAG), which is sometimes called a *class lattice*. The hierarchy (or DAG) has only one root, which is a system designed class named CLASS. Since the hierarchy is rooted and connected, there is no isolated class, and every class is reachable from the root. Every class in the class hierarchy (or DAG) has a distinct name. Similarly, every attribute and method in a class has a distinct name.

In this thesis we do not assume the existence of either one root or a connected DAG. We will use the term class hierarchy for either a tree-like hierarchy or the more general directed acyclic graph (DAG).

It is also worth noting here J. D. Ullman's view of the essential features in object-oriented databases [Ullm88], especially because he explicitly includes encapsulation as part of the essential features. According to him the essential features of OO database systems are:

- *Complex Objects.*

  This paradigm means: the system's ability to define data types with a nested structure.

- *Encapsulation.*

  This is the ability to define procedures applying only to objects from a certain type, and the ability to ensure that all access to those objects is by means of application of one of these procedures.

- *Object-Identity.*

  This means the ability to distinguish two objects that "seem" equal, because all their components of primitive type are the same. Examples of primitive types are characters, strings and numbers.

Furthermore, J. D. Ullman states: a system that supports encapsulation and complex objects is said to support abstract data types (ADT's), or *classes.* A *class* or ADT is the definition of some structure together with the operations by which objects of that class can be manipulated. It should be noted here also that the definitions of the words *class* and *type* in [Ullm88] are slightly different from those of [Kim90]. The meaning of the word *type* in [Ullm88] is the same as the meaning of the phrase *primitive class* defined in [Kim90], and the meaning of the phrase *abstract data type* is similar to the meaning of the word *class* described in [Kim90]. In this thesis we use the word *class* as interpreted in [Kim90].

1.2 Importance of These Features for Databases

This section summarizes the importance of the core concepts required in object-oriented databases.

Object Identity. In an object-identity based model, one is able to distinguish between equal objects and identical objects [AtBa92]. This is the direct result of the characteristics of an object identity, because by using an object identity an object has an existence which is independent from its values. Two objects are identical if they are the same object, while two objects are equal if they have the same values. We can distinguish between identical objects and equal objects in two ways: object sharing and object updates [AtBa92]:

- *Object sharing.* In an identity based model, two objects can share the same components. So, the pictorial representation of a complex object is a *graph*; in contrast, in a system without object identity its pictorial representation is limited to a *tree*. For example, John and Mary have a one-year-old child named Jean. In real life, John and Mary may be the parents of the same child, or perhaps there are two children involved. In an identity-less system, John is represented by

  $$(john, 33, \{(jean, 1, \{\})\})$$

  and Mary is represented by

  $$(mary, 31, \{(jean, 1, \{\})\})$$

  In an identity-less system, there is no way of expressing whether or not John and Mary are the parents of the same child. However, in an identity-based model, we can represent whether the two structures share the common part $\{(jean, 1, \{\})\}$ or not; this allows us to distinguish between equal objects and identical objects.

- *Object updates.* Let John and Mary be the parents of the same daughter, Jean. In this case, all updates concerning John's daughter will also apply to Mary's daughter. In a value based system the two updates should be executed separately. On the contrary, in an identity-based model, we only need to do the update once.

Parallel to the above benefit, object identities are useful to prevent giving more than one reference to the same object. For example, a married woman, who has more than one legal name, will not have more than one object identity; on the contrary, in a value based system, the same woman may have more than one reference.

Complex Objects. We build complex objects from simpler ones by applying object constructors. Examples of simplest objects are integers, characters and strings. According to [AtBa92], the minimal set of constructors that the system should have are sets, tuples and lists (or arrays). Sets are important because they are a natural way of representing collections from the real world, for example: a set of graduate students. Tuples are important because they provide a natural way of representing properties of an entity. Lists (or arrays) are important because they capture order which occurs in the real world, for example the need for time-series data. The importance of *attributes* and *methods* is captured by the importance of tuples as object constructors, since attributes and methods are constructed using tuples.

Classes or Types. Classes (or types) are important because they serve as a form of data-structuring mechanism. Thus, the classical notion of a database schema can be replaced by that of a set of classes or a set of types.

There are similarities between classes and types. Both terminologies have been used for the same meanings i.e., to specify the common features of a set of objects. However, the differences can be subtle for some systems [AtBa92]. We can categorize object-oriented systems into those supporting the notion of class and those supporting the notion of type. The following are the concepts of types and classes based on [AtBa92].

A type in an object-oriented system specifies the common features of a set of objects; it corresponds to the notion of an abstract data type. It consists of *the interface part* and *the implementation part.* Users can only see the interface part, whereas the type designer can see the implementation part as well. The *interface part* contains the list of operations together with their signatures. The signatures specify the types of the input parameters and the types of the results of the operations. The *implementation part* consist of a data part and an operation part. The data part describes the internal structure of the object's data. The operation part contains the procedures that implement the operations of the interface part.

In programming languages, types are used to help programmers in ensuring program correctness. By typing, the system can do type checking at compile time. Thus types are mainly used at compile time to check the correctness of the programs.

The specification of a class is the same as that of type i.e., it summarizes the common features of a set of objects. However, the notion of class is different from that of type [AtBa92]. Class is more a run time notion. It contains two aspects, the object factory and an object warehouse. The *object factory* is used to create new objects. The *object warehouse* allows the class to be attached to the set of its instance objects. Then the user can manipulate the warehouse by applying operations on all instances of the class. Classes are not used for checking the correctness of a program but they are used for creating and manipulating objects [AtBa92].

Class Hierarchies. Class hierarchies are important because they give a concise description of the world and they help us in factoring out shared specification and implementations in applications [AtBa92]. For example, consider three kinds of objects: Persons, Employees, and Students. Students and Employees are Persons. Say initially we introduce the class *Person* with attributes **name** and **age** and we write an operation (method) **die** for instances of this class. Then we declare the class *Employee* as a subclass of *Person.* It inherits the attributes and methods from *Person.* In addition, we define an attribute **salary** for the class *Employee.* Similarly, we then declare the class *Student* as a subclass of *Person* and we define the attribute **setOf-**

Grade. The class *Student* will inherit the attributes and methods from *Person*. This illustration shows that we can reduce the number of specifications of attributes and methods for each class because we can *reuse* the specifications of its superclass. Thus we can have a better-structured and more concise description of the schema because we can factor out specifications.

Encapsulation. According to [AtBa92], the idea of encapsulation comes from:

1. The need for a clear distinction between specification and implementation of an operation.
2. The need for modularity. Modularity is required to structure complex applications designed and implemented by a team of programmers. It is also important as a tool for *protection* and *authorization*.

The principle of encapsulation in object-oriented databases is to encapsulate both program and data. In a relational system, the data are stored in a database, but the programs that manipulate the data are stored in an ordinary file (they are not part of the database). Moreover, there are distinctions between the query language, which is usually declarative, and the programming language used for a›)lications programs such as update. On the other hand, in an object-oriented system, both the data and the operations are stored in the database. Thus, there is a single model for data and operations and we can hide both types of information. Relating to this, we specify the interface of an object, so that no operations other than those given in the interface can be performed.

Encapsulation serves as a kind of "logical data independence" [AtBa92]. This means that we can change the implementation of an abstract data type (or a class) without changing any of the programs using that type (or class). This way the application programs are protected from implementation changes in the lower layers of the system. It is argued in [AtBa92] that proper encapsulation is obtained only when the operations are visible, and the data and the implementation of the operations are hidden in the objects. However, there are some cases where encapsulation is not required. for example, ad-hoc queries that do not change the database's contents. In this case, perhaps we want to allow encapsulation to be violated because the issue of maintainability is not important.

1.3 The Goal of This Thesis

The goal of this thesis is to design a logic that can be used as the basis in designing object-oriented databases, similar to Predicate Calculus which was used by Codd as the basis for designing relational databases [Codd70]. We want a logic that can be used for reasoning about object-oriented database systems. The resulting logic will become a logical framework for natural representation and manipulation of complex objects. In other words, we attempt to put in place a formal basis for combining the ideas of deductive database and object-oriented database that can evolve to actual computer software.

1.4 Significance of the Work

When Codd [Codd 70] designed relational databases, he relied on the established mathematical logic concerning Predicate Calculus. In this thesis we design a logic that is appropriate for reasoning about object-oriented database systems. A success of that design would remove some di-advantages of current deductive and object-oriented databases, and provide a strong logical basis for object-oriented database systems. In addition, the logic would be able to serv- as a flexible modeling environment [Page89].

In designing the logic we attempt to overcome the weaknesses of both deductive and object-oriented databases, and to capture most advantages from these two types of databases. The following are the advantageous properties of deductive databases:

- Having non-procedural queries. Many users prefer a declarative style of query language because it allows them to concentrate more on expressing what they want than spending time on how their queries will be answered, or on how to produce answers on their queries algorithmically. This is proven from the success of SQL as a query language for relational database systems.
- Having a uniform language for querying, updating, defining virtual data, and constraining. By having a uniform language, users need to learn only one language for different purposes.

The following are advantageous properties of object-oriented databases:

- having object identities, which will help in determining quickly whether two objects are identical;
- the ability to represent complex objects, methods, inheritance, and classification hierarchies for organizing database schema.

Some problems encountered in each type of these database systems are:

- from deductive databases: the flat database models and problems related to the absence of support for object identities, data abstraction and inheritance;
- from object-oriented databases: the lack of formal semantics, the lack of non-procedural methods. Current existing object-oriented database management systems have different properties, and even the same terminologies have different meanings.

1.5 Organization of the Thesis

Chapter 2 presents related work which also serves as necessary background material and historical overview.

Chapter 3 discusses the design process of the logic, the syntax and semantics of the logic, the properties of the semantic entailment, and the soundness and completeness of the proof procedure.

Chapter 4 discusses the logic as a programming language, several extensions of the logic, and examples.

Finally, in Chapter 5, we summarize what we have done, our contribution, a comparison between our designed logic and F-logic [KifLa89, KifLaWu90] as well as the new F-logic [KifLaWu94], a summary of object-oriented database concepts based on our logic, and possible future work.

CHAPTER 2

Related Work

This chapter introduces several important closely related work on the effort of combining deductive and object-oriented databases. This chapter also serves as historical background to this research area.

2.1 O-Logic [Maier86]

In this section, we present a summary of David Maier's O-logic paper [Maier86] which is an early paper in designing a formal basis for deductive object-oriented databases.

2.1.1 Introduction

Maier's O-logic is considered as the first attempt in the effort of combining logic and object-oriented (OO) programming as it is applied to databases. The goal of O-logic development was to provide a formal basis for the application of this combination. Since [Maier86] contains David Maier's initial ideas presented in a workshop, we found that some terminology and concepts that he used are not clearly defined (respectively, explained) or not defined (respectively, explained) at all. However, some ideas from this paper are used in several papers about object-oriented logic written afterwards.

The O-logic was designed to capture some of the advantageous characteristics from both logic and OO programming. From logic programming, O-logic adapts non-procedural queries, a uniform language for queries, updates, defining virtual data and constraints. On the other hand, from OO programming, O-logic takes the following characteristics: complex objects that can change their internal structures, object identities, and class hierarchies. A database built based on O-logic stores an explicit model of the database, or its portion, which is a departure to storing a theory (a set of formulae) and using its implicit model as in Prolog.

2.1.2 O-Logic Object-Oriented Modeling

Updating in a logic based system is rather awkward, because we must do this by changing some statements about the world rather than by changing the structure of the world (the notion of structure will be explained later in Subsection 2.1.4). As an example in logic programming, two facts cannot share the same sub parts. When an object belongs to two facts, then we should represent these facts as two statements with duplicate information about the object. If the information about the object changes, the two statements must be updated as well. The following example, adapted from [Maier86], illustrates this kind of problem.

dept("Sales", manager(name("Joe", "Doe"), addr("24-455 PlattsLn")).
dept("Marketing", manager(name("Joe", "Doe"), addr("24-455 PlattsLn")).

The two statements talk about the same individual, and when he moves we must update both facts. Moreover, the change is made by retracting both facts and asserting the following new facts.

dept("Sales", manager(name("Joe", "Doe"), addr("22A-939 WesternRd")).

dept("Marketing", manager(name("Joe", "Doe"), addr("22A-939 WesternRd")).

In Prolog, we cannot say something like "Everything is still the same except the address". This is why we should do what is described above.

In an O-logic based database, all terms are construed as assertions of existences of objects in the world, and relationships among them. The database admits ***an interpretation*** as a special model for the terms. This interpretation is useful for updating especially as it avoids problems such as illustrated in the above example.

A database never abstracts everything about the world. In other words, we cannot expect that the description about an object to be comprehensive. For example, we may have two people having the same name and they both work in the same department, if the information that we have is only these two attributes, we might think that both people are the same object. For this reason, in a database, it is useful to represent two different objects differently, so that lo not have to investigate or to make up properties for distinguishing them [MacLe83]. This is why O-logic uses object identities to distinguish objects, so that we do not need to check the internal properties of objects to determine whether they are the same or not, and we do not fall into a false conclusion that two objects are the same simply because their explicitly asserted internal properties are the same.

2.1.3 O-Logic Syntax

The alphabet of an O-logic language consists of:

1. a countable set[1] D of basic terms called *data values*;
2. a countable set L of *labels*;

[1]A set is ***denumerable*** if and only if it is one-to-one correspondent with the set of all natural numbers, and a set is ***countable*** if and only if it is either finite or denumerable.

3. a countable set Σ of *class names*;
4. a countable set V of *object variables*, each object variable is denoted by a capital letter;
5. logical connectives: $\vee, \wedge, \Rightarrow, \neg$ and quantifiers $\forall, \exists$;
6. a set of auxiliary symbols: " : ", ")", "(", " $\rightarrow$ ", "$\Leftarrow$ ", ",",.

2.1.3.1 O-Terms (Object Terms)

O-teιms are built recursively from D, Σ, and L. The set of basic terms D is the names of all objects with no further internal structures. For exposition purposes, the set of basic terms, D, is limited to integers and strings. The set of labels L is used to build a set of terms which is disjoint from D. An O-term is recursively defined as follows:

1. a simple O-term,

$$c : d$$

   where $c \in \Sigma$, and $d \in D$ or $d \in V$;

2. a complex O-term

$$c : X(label_1 \rightarrow c_1 : t_1,\ \ldots,\ label_n \rightarrow c_n : t_n)$$

   where c, c_1, $\ldots$, $c_n \in \Sigma$, $X \in V$, $label_1$, $\ldots$, $label_n \in L$, and t_1, $\ldots, t_n$ are O-terms.

The component $label_i \rightarrow c_i : t_i$ is also called a *field*, where t_i is the *field value*.

Let $E, N, Y, Z \in V$. The following are examples of O-terms.

$employee : E$

$string :$ "*Joe*"

$employee$: $E(name \rightarrow personName : N(first \rightarrow string :$ "*Joe*",
$last \rightarrow string :$ "*Doe*"),
$degree \rightarrow degree : Y(level \rightarrow string :$ "*M.Sc.*", $year \rightarrow number : 1990$,
$school \rightarrow university : Z(name \rightarrow string :$ "*U.W.O*",
$prov \rightarrow string :$ "*ON*")))

2.1.3.2 O-Formulae

An O-term is an *atomic formula.* More complex formulae are built from O-terms, logical connectives $(\wedge, \vee, \neg, \Rightarrow)$ and quantifiers $(\forall, \exists)$ in the same way as formulae in Predicate Calculus are built from atomic formulae.

2.1.4 O-Logic Semantics

In O-logic all O-terms are construed as assertions of existences of objects in the world, and relationships among them.

A set of objects may form a class. The notion of class is not defined explicitly in [Maier86]. However, from the examples given, it seems that O-logic uses the common definition of class as given in [Kim90], i.e., a class is a set of objects sharing the same set of attributes and methods. However, unlike the related concept in [Kim90], an object is not restricted to belong to only one class, it is even allowed to have multiple unrelated classes. Classes may form a class hierarchy or a more general directed acyclic graph if multiple inheritance is allowed. Figure 2.1 gives an example of a hierarchy H of classes.

The following defines the semantic structure for a formula f, and the conditions under

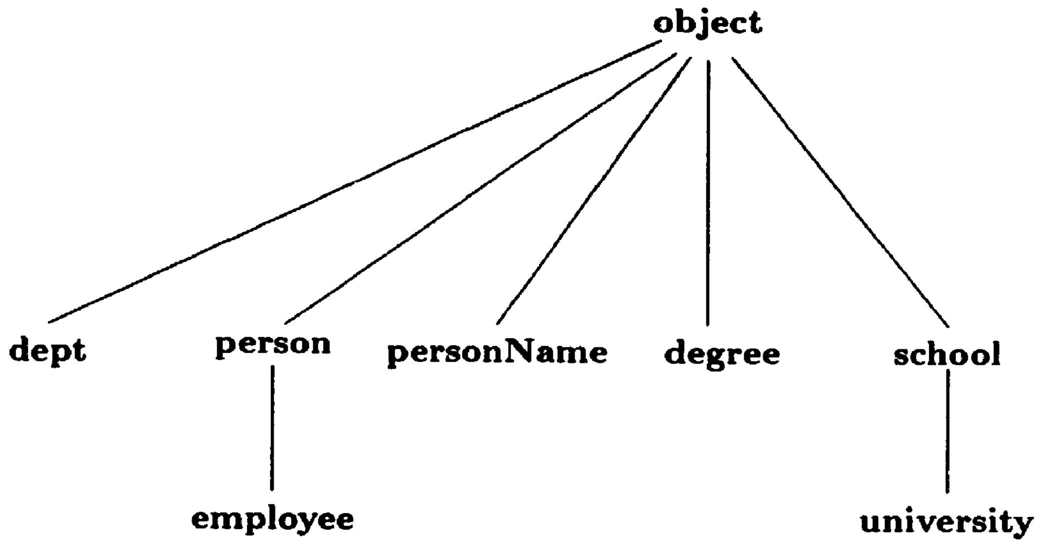


Figure 2.1: An Example of a Class Hierarchy (from [Maier86])

which a formula is true for a structure. Let W be a set of objects (called *entities* in [Maier86]) which is disjoint from D. A *structure* ST for a formula f is a triple $\langle U, g, h\rangle$, where U is the *universe* which is the set $D \cup W$. Thus, U is *the set of all defined objects.* The second component g interprets labels as functions which map labels in L to partial functions from W to U, i.e., $g : L \rightarrow (W \rightarrow U)$ (maps labels to functions from entities to the set of all defined objects). The third component, h, maps the classes to sets of entities, i.e., $h : \Sigma \rightarrow \mathcal{P}(W)$, where $\mathcal{P}$ is the *power set operator.* The classes of basic terms D are mapped in a fixed way. It should be noted that the function h must respect the class hierarchy, so if C_1 is a superclass of C_2, then $h(C_1) \supseteq h(C_2)$.

Variable Assignment

A variable assignment $\mathcal{V}$ is a mapping from variables to U. The variable assignment $\mathcal{V}$ is extended to O-terms as follows:

Given an O-term t, $\mathcal{V}(t)$ is defined as follows:

1. $\mathcal{V}(t) = \mathcal{V}(Y)$ if $t = c : Y$, where c is a class name and Y is an object variable;

2. $\mathcal{V}(t) = d$ if $t = c : d$, where c is a class name and $d \in D$;

3. $\mathcal{V}(t) = \mathcal{V}(Y)$ if $t = c : Y(\ldots)$, where c is a class name and Y is an object variable.

This variable assignment is called *instantiation* in [Maier86]. Notice that the result of the application of $\mathcal{V}$ in item 1 and 3 is similar.

The Meaning of a Formula

Given a structure $ST = \langle U, g, h \rangle$ and variable assignment $\mathcal{V}$. The *meaning structure* $M_{ST,\mathcal{V}}$ for an O-term (an atomic formula) is a constant mapping that is defined as follows:

- For any O-term $c : d$, where $d \in D$, $M_{ST,\mathcal{V}}(c : d) = true$ if $d \in h(c)$ where h maps the class c into a set of entities, otherwise $M_{ST,\mathcal{V}}(c : d) = false$.

- For any O-term $c : X$, where X is an object variable, $M_{ST,\mathcal{V}}(c : X) = true$ if $\mathcal{V}(X) \in h(c)$ (i.e., object $\mathcal{V}(X)$ exists in the right class) otherwise $M_{ST,\mathcal{V}}(c : X) = false$.

- For any complex term $t = c : X(label_1 \rightarrow c_1 : t_1, \ldots, label_n \rightarrow c_n : t_n)$, if:

  1. $\mathcal{V}(X)$ is an entity, not a data value, and $\mathcal{V}(X) \in h(c)$;

  2. for $e = \mathcal{V}(X)$ and $e_i = \mathcal{V}(t_i)$, $g(lab_i)(e) = e_i$, and $e_i \in h(c_i)$;

  3. $M_{ST,\mathcal{V}}(c_i : t_i) = true$;

  then $M_{ST,\mathcal{V}}(t) = true$, otherwise $M_{ST,\mathcal{V}}(t) = false$. Informally, a term t is true if the values of its fields and its subfields correspond to those given by g, and respects their classes which is given by h.

The definition of $M_{ST,\mathcal{V}}$ on an O-term (i.e., an atomic formula) is extended to a more complex formula. If a and b are O-terms then $M_{ST,\mathcal{V}}(a \wedge b)$, $M_{ST,\mathcal{V}}(a \vee b)$, $M_{ST,\mathcal{V}}(a \Rightarrow b)$ and $M_{ST,\mathcal{V}}(\neg a)$ are defined as usual as in Predicate Calculus. For quantifiers, O-logic has

- $M_{ST,\mathcal{V}}(\forall X\ f) = \mathit{true}$ if for every $\mathcal{V}'$ that agrees with $\mathcal{V}$ except possibly on X, $M_{ST,\mathcal{V}'}(f) = \mathit{true}$; and
- $M_{ST,\mathcal{V}}(\exists X\ f) = \mathit{true}$ if for some $\mathcal{V}'$ that agrees with $\mathcal{V}$ except possibly on X, $M_{ST,\mathcal{V}'}(f) = \mathit{true}$.

A structure $ST = \langle U, g, h \rangle$ is called *a model* for a formula f if $M_{ST,\mathcal{V}}(f) = \mathit{true}$. It should be noted that for a closed formula (a formula with no free variables) f, $M_{ST,\mathcal{V}}(f)$ does not depend on the variable assignment $\mathcal{V}$. So, if f is a closed formula and $M_{ST,\mathcal{V}}(f)$ is true, then the structure ST is a model of f.

Example (from [Maier86]):

Suppose we have a structure $ST = \langle U, g, h \rangle$ as follows:

$U = \{e_1, e_2, e_3, e_4, e_5\} \cup D$

$g(\mathit{manager})(e_1) = e_2$, $g(\mathit{manager})(e_3) = e_4$, $g(\mathit{worksIn})(e_2) = e_1$, $g(\mathit{worksIn})(e_4) = e_5$, $g(\mathit{dname})(e_1) =$ "*Sales*", $g(\mathit{dname})(e_3) =$ "*Manuf*", $g(\mathit{dname})(e_5) =$ "*RandD*".

Let h be a mapping such that:

$h(\mathit{object}) = \{e_1, e_2, e_3, e_4, e_5\}$

$h(\mathit{person}) = \{e_2, e_4\} = h(\mathit{employee})$

$h(\mathit{dept}) = \{e_1, e_3, e_5\}$

Assume that:

$t = \ \mathit{dept} : R(\mathit{manager} \rightarrow \mathit{employee} : M)$

Let $\mathcal{V}_1$ be a variable assignment where $\mathcal{V}_1(R) = e_1$ and $\mathcal{V}_1(M) = e_2$, then $M_{ST,\mathcal{V}_1}(t)$

is *true* because $e_1 \in h(dept)$, $g(manager)(e_1) = e_2$ and $e_2 \in h(employee)$. However, if we have another variable assignment $\mathcal{V}_2$, where $\mathcal{V}_2(R) = e_1$ and $\mathcal{V}_2(M) = e_3$, then $M_{ST,\mathcal{V}_1}(t)$ is *false* because although $e_1 \in h(dept)$, but $g(manager)(e_1) = e_2$; furthermore $e_3 \notin h(employee)$.

2.1.5 O-Logic Queries

An existentially-quantified formula can be treated either as a query that needs answers (for the binding of its existentially-quantified variables) or as a query that needs only a yes-no response. For example, formula $\exists X \exists Y \exists Z\, f$ is *true* in a structure ST if there is a variable assignment $\mathcal{V}$ with $M_{ST,\mathcal{V}}(f) = true$. If we are interested in the particular values of X, Y, and Z that make the formula f *true*, the answers to f under ST are defined as

$$\{\mathcal{V}[XYZ] \mid M_{ST,\mathcal{V}}(f) = true\}$$

where $\mathcal{V}[XYZ]$ is the restriction of the variable assignment $\mathcal{V}$ to only a three-element domain of the variables of interest. The expression $\mathcal{V}[XYZ]$ is basically an XYZ-tuple that can have entities and also data values in it. So the answer is actually pointing to objects (elements of U) in the database.

2.1.6 O-Logic Stores Structures as a Database

Let an *entity finite* structure $\langle U, g, h\rangle$, where U, g, and h are defined as before, be a structure in which the set of entities (W) is finite and the set of labels (L), for which g is defined, is finite as well. Then we can store such an entity-finite structure as database in lieu of a set of formulae. The database system manages unique surrogates to identify entities in W. The g and h parts of the structure could be stored as binary and unary relations respectively. Then, O-logic formulae can be evaluated against this structure (see the example in Section 2.1.4).

For a database schema, apart from the class hierarchy, we can add typing constraints. A typing constraint defines what fields every member of a class has, and what classes the values of those fields should belong to. For example:

$$\forall E\, \exists Y\, \exists N\, (employee : E \Rightarrow employee : E(worksIn \rightarrow dept : R,\\ name \rightarrow personName : N))$$

This declaration means that every *employee* entity has a *worksIn* field and a *name* field. This implication can be abstracted into a more familiar schema description:

$$employee(worksIn \rightarrow dept, name \rightarrow personName)$$

This description says that certain fields must exist for an object from the class *employee*. However, an object is allowed to have more fields than what is specified. Since an object is allowed to belong to more than one class, it is possible that an object has more fields than what is specified. Some restrictions must be put on schemata between subclasses and superclasses. They must satisfy the condition that a subclass must have at least the fields of the superclass, and the typing of those fields must be at least as restrictive as in the superclass.

Relations are represented as tuples of objects. For example, a supplier-parts relation may have a schema

$$SP(supplier \rightarrow Company, provides \rightarrow Part)$$

It should be noted that this relation is not in the first normal form (of relational databases) because attribute values may not be basic values i.e., they could be entities.

2.1.7 O-Logic Update Commands

Update commands are used to add entities, to set field values, and to create new entities. Let t be an O-term and f be an O-logic formula. The update-command form is

$$t \Lleftarrow f$$

Formula f is used to provide a sequence of variable bindings to be used for instantiating t. O-logic interprets every variable assignment of t as a property that must be made true in the current structure $ST = \langle U, g, h \rangle$. More precisely, this command means that for all answers to f (i.e., variable assignments that make $M_{ST,\nu}(f) = true$), the current structure ST is modified into a new structure ST' so that $M_{ST',\nu}(t) = true$.

Example (from [Maier86]):

The following statement is intended to express that *Joe Doe* is promoted to the manager of the Purchasing department:

$$\begin{aligned}
&manager : E(manages \rightarrow dept : R) \Lleftarrow \\
&\qquad employee : E(name \rightarrow personName : \\
&\qquad\qquad N(first \rightarrow string : \text{“}Joe\text{”}, last \rightarrow string : \text{“}Doe\text{”})), \\
&\qquad dept : R(dname \rightarrow string : \text{“}Purchasing\text{”})
\end{aligned}$$

2.1.8 O-Logic Limited Negation

Limited forms of structural negation in the head part of update commands are allowed for removing an entity from a class or removing a field (label) from an entity.

Examples (from [Maier86]):

(1) Taking the manager of Purchasing out of the Stockholder class:

$\neg stockholder : M \Leftarrow\!\!\!\Leftarrow$

$\quad dept : R(name \rightarrow string : \text{“}Purchasing\text{”}, manager \rightarrow employee : M)$

(2) Removing the “worksIn” field:

$$E(\neg worksIn) \Leftarrow\!\!\!\Leftarrow employee : E(name \rightarrow personName : N(first \rightarrow string : \text{“}Joe\text{”}, last \rightarrow string : \text{“}Doe\text{”})$$

Unfortunately, in [Maier 86], there is no explanation about what kind of limitation is imposed on the negation of the head part of update commands, or why it is limited.

2.1.9 O-Rules

Similar to Prolog, rules in O-logic are limited to Horn-type rules in order to get a minimal model semantics for a database with rules. These Horn-type rules are called O-rules. The head of a rule is either one O-term or none, and the body of the rule consists of a conjunction of O-terms. The syntax of O-rules is the same as updates except for “$\Leftarrow$” in place of “$\Leftarrow\!\!\!\Leftarrow$.”

Semantically, O-rules behave like deferred updates. A query is answered as if all of the rules have been executed before answering the query. This interpretation is similar to the minimal model semantics of Prolog.

In O-logic, since an update can change a field value, we should be more careful than in Prolog. The change of a field value may cause a contradiction to be expressed without negation. For example, given a fixed interpretation of data values in D, we could have the following contradiction:

$$V(field \rightarrow 1) \wedge V(field \rightarrow 2)$$

Thus, we might have a situation where the head contradicts the information already in the database without negation.

An entity-creating rule i.e., a rule that has a variable in the head and not in the body, might cause serious problems. Although this kind of rule will not cause a contradiction, it causes a problem for defining a minimal model semantics. On the other hand, O-rules that add fields or add entities to classes would not be a problem for the minimal model.

Before we define the minimal model in O-logic, we define first the containment property between two structures. Structure $ST' = \langle U', g', h' \rangle$ *contains* structure $ST = \langle U, g, h \rangle$ if the following holds:

1. $U' \supseteq U$,

2. $g'(lab)$ is defined at least where $g(lab)$ is, for every label lab, and

3. $h'(C) \supseteq h(C)$ for every class C.

Let U, g and h be defined as before. *A minimal model* for a structure $ST = \langle U, g, h \rangle$ under a set of rules is the structure ST' such that

1. ST' *contains* ST, and

2. the rules, which are interpreted as updates, will not add anything to ST', and

3. no structure smaller than ST' has these properties.

Applying a rule as an update poses problems. Suppose an entity that satisfies the head of a rule already exists, then the rule will still add another entity to ST'. In principle, we can apply the rule many times, generating more and more entities, because the corresponding update introduces a new entity. As a result there will be no ST' that is unchanged under the rules, a situation that will make U' infinite.

O-logic solves the problem by using meta axioms that put a limit on the applications of an entity-creating rule. By the definition of an update command, an update that creates a new entity gets an answer, A, from the body of the command and extends it to the variables in the head of the command. A meta-axiom says that any answer gets extended *exactly once*. Let's use the previous update's example of adding *Joe Doe* as a manager. However, this time we view it as a rule. Then we have a meta-axiom stating that exactly only one set of bindings of E and N are made for each binding of R.

$$\begin{aligned}
&\mathit{manager} : E(\mathit{manages} \rightarrow \mathit{dept} : R, \mathit{worksIn} \rightarrow \mathit{dept} : R,\\
&\qquad \mathit{name} \rightarrow \mathit{personName} : N(\mathit{first} \rightarrow \mathit{string} : \text{“}\mathit{Joe}\text{”},\\
&\qquad \mathit{last} \rightarrow \mathit{string} : \text{“}\mathit{Doe}\text{”}, \mathit{age} \rightarrow \mathit{integer} : 26) \Leftarrow\\
&\qquad \mathit{dept} : R(\mathit{dname} \rightarrow \mathit{string} : \text{“}\mathit{Purchasing}\text{”})
\end{aligned}$$

The meta axiom will allow us to say when a rule has been applied "enough" times.

2.1.10 Discussion of O-Logic

In this section, we will discuss quantification of object variables in an entity creating rule. The following example from [Maier86] is an entity creating rule to form an interesting pair entity for each *employee* whose *manager* has the same name.

$$\begin{aligned}
\mathit{interestPair} :\ & P(\mathit{emp} \rightarrow \mathit{employee} : E, \mathit{manager} \rightarrow \mathit{employee} : M) \Leftarrow\\
& \mathit{employee} : E(\mathit{name} \rightarrow \mathit{string} : N, \mathit{works} \rightarrow \mathit{dept} :\\
& R(\mathit{manager} \rightarrow \mathit{employee} : M(\mathit{name} \rightarrow \mathit{string} : N)))
\end{aligned}$$

It is argued in [Maier86] that a universal quantification over variable P will not make any sense in this example, because we do not want to express that every *interestPair* object has E in its label *emp* and M in its label *manager*. Instead, we want an existential quantifier that matches the update semantics, i.e., there is some *interestPair* object P for each E and M value that the body matches. Thus, the following quantification is suggested: $(\forall E)(\forall M)(\forall N)(\exists P)$. It is pointed out in [CheWa89] and [KifWu89] that Maier's argument for this choice is not clear and they claim that the correct quantification is $(\forall E)(\forall M)(\exists P)(\forall N)$. It is also said in [KifWu89] that Maier's quantification is true only if E and M functionally determine N. So, if the label *name* is set valued, the two quantifications will yield different results.

It is argued in [KifWu89] that the quantification for the object variable in the head, which does not appear in the body, should be quantified explicitly. Thus, we do not need to determine the quantification in an ad-hoc manner. Later we will find in [CheWa86] and [KifWu89] that this quantification problem is solved by introducing an object identifier as a function of several object variables appearing in the rule's body (see Section 2.2.6 and 2.3.2.1). The idea of this kind of object identifier is similar to the idea of a Skolem function.

2.1.11 Concluding Remarks on O-Logic

O-logic [Maier86] is the first serious attempt we know about which merged ideas from the field of object-oriented databases and logic programming.

Some of Maier's ideas in [Maier86] have been adapted in subsequent papers of logic formalisms for object-oriented databases. Unlike traditional deductive databases, an O-logic database stores a structure rather than a set of formulae. In other words, its extensional database is identical to its interpretation. The word *extensional* here has the same meaning as the one used in Datalog, which is the existing tables (objects).

The space of objects in O-logic is divided into a set of *basic terms* (also called data values) and a set of *entities*. It should be noted that any element of the set of basic objects cannot have attributes.

O-logic possesses the following concepts of object-oriented databases: object, object identity, attributes, class and class hierarchy. The core concepts of object-oriented databases [Kim90] that are still missing are *method definition* and *inheritance*.

2.2 RO-Logic [KifWu89]

This section summarizes a revision and significant extension of Maier's O-logic. Although this revision of O-logic [KifWu89] is still called O-logic, in this thesis we call it RO-logic to avoid confusion with the the original O-logic [Maier86].

2.2.1 Introduction

RO-logic supports *complex objects, object identity, class*, and *class hierarchy*. It contains all of the Predicate-Calculus features as a special case. It is a result of a combination of object-oriented, value-oriented and logic-programming paradigms. It also tolerates inconsistent data.

It was claimed in [Ullm87] that the deductive approach is intrinsically "value-oriented" so that it cannot be combined with the inherent object-oriented features, such as object identities. However, the work presented in [KifWu89] proves that this claim is not really true. The presented logic has a well-defined logical semantics that keeps some significant properties of object-oriented database systems.

When RO-logic was designed, it was intended to have a deductive system that would capture the following features of object-oriented databases: complex objects, typing, and object identity.

RO-logic was designed on the basis of Maier's O-logic [Maier86]. O-logic is claimed to suffer from some semantic problems, especially quantification related problems [KifWu89]. These semantic problems is solved in RO-logic [KifWu89]. The semantics of O-logic is extended to include sets and to handle inconsistent information. In addition, RO-logic also has a resolution-based proof procedure which is sound and complete.

The language of RO-logic is closely related to frame languages [Minski81, FiKe85]). There have been several efforts in combining frames with deduction such as [FiKe85]. It is claimed in [KifWu89] that frame languages can be regarded as a scaled-down version of RO-logic languages plus inheritance. Pointers in frame languages which are used to refer to other frames are essentially the same as object identities.

The next two subsections describes the syntax and semantics of RO-logic.

2.2.2 RO-Logic Syntax

The alphabet of an RO-logic language consists of:

1. a set O of *basic objects* i.e., objects from basic types such as integer and string. This set also includes a pair of distinguished basic objects: $\perp$ (the *nil* object) and $\top$ (the meaningless object).
2. a set F of function symbols, which are called *object constructors*;
3. a countable set $L_{\#}$ of *single-valued (or functional)* labels;
4. a countable set $L_{\bullet}$ of *set-valued labels*;
5. a countable set Σ of *class names*;
6. a countable set V of *object variables*;

7. logical connectives: $\vee, \wedge, \Leftarrow, \neg$ and quantifiers: $\forall, \exists$;
8. a set of auxiliary symbols: " : ", ")", "(", "]", "[", "}", "{", ",", " $\rightarrow$ ".

The set of all labels ($L_{\#} \cup L_{\bullet}$) is denoted by L. Some additional alphabet symbols such as parentheses and arrows are introduced later. It is assumed that the sets $O, F, L_{\#}, L_{\bullet}, V$ and Σ are disjoint.

2.2.2.1 RO-Terms

An id-term is constructed from object constructors (from F), basic objects (from O) and object variables. Similar to the way a functional term is formed in Predicate Calculus, for example: $f(a, p(X, c))$, where f and p are function symbols, a, c are basic objects, and X is an object variable. The set of *all ground id-terms* is denoted by O^* (playing the role analogous to that of Herbrand Universe in classical logic). This set O^* consists of the set of basic objects O and the set of constructed objects $O^* \setminus O$ (the set O^* minus O).

In RO-logic, typing constraints are defined similar to that of Maier's O-logic, except that they are extended to include set-valued labels. An *RO-term* is defined as one of the following:

1. a simple RO-term, $p : T$, where p is a class and T is an id-term;
2. a complex RO-term,

$$p : T[flab_1 \rightarrow t_1,\ \ldots,\ flab_n \rightarrow t_n,$$
$$slab_1 \rightarrow \{s_{1,1},\ \ldots,\ s_{1,m}\},\ \ldots,\ slab_k \rightarrow \{s_{k,1},\ \ldots,\ s_{k,m_k}\}]$$

where p is a class name, T is an id-term, and t_i for $1 \leq i \leq n$ and $s_{j,m}$ for

$1 \leq j \leq k$, $1 \leq m$ are RO-terms; the labels $flab_i$ for $1 \leq i \leq n$ are functional, that is, from $L_{\#}$, and labels $slab_j$ for $1 \leq j \leq k$ are set-valued, that is, from $L_{\bullet}$.

2.2.2.2 RO-Formulae

An RO-term is an *atomic formula.* A more complex formula is constructed from simpler formulae by means of logical connectives $\vee, \wedge, \neg$, and quantifiers $\exists$ and $\forall$ in a similar way to that of classical first-order logic. The implication $f \Leftarrow g$ is equivalent to $f \vee \neg g$.

A Flat Lattice on Ground RO-Terms

A structure of a flat lattice on ground RO-terms is defined as follows. The partial order $<_O$ is defined on O^* by making $\perp$ and $\top$ the minimal and the maximal elements respectively. In addition, it is assumed that the elements in $O^* \setminus \{\perp, \top\}$ are incomparable with respect to $<_O$. This means that we say $a <_O b$ if and only if either $a = \perp$ and $b \in O^* \setminus \{\perp\}$, or $b = \top$ and $a \in O^* \setminus \{\top\}$. As usual, $a \leq_O b$ means $a <_O b$ or $a = b$.

A Complete Lattice on Classes

A complete lattice is formed by classes. The *least restrictive* class is **all** (the class of all objects) and the *most restrictive* class is **none**. Ordering on the classes is denoted by $<_\Sigma$, the more restrictive a class is the bigger it is with respect to $<_\Sigma$, for example **all** $<_\Sigma$ **none**.

Some simplifications used in the syntax are as follows:

- if a functional label is omitted in an object specification, then the label's value is **all:** $\perp$;
- if a set-valued label is omitted in an object specification, then the label's value

is { };

- if the class specification is omitted in an RO-term, then the class **all** is assumed.

As an example, the following two specifications are regarded as the same thing:

$empl : john[name \rightarrow \text{"}john\text{"}]$

$empl : john[name \rightarrow \textbf{all} : \text{"}john\text{"},\ pay \rightarrow \perp,\ children \rightarrow \{\}]$.

2.2.3 RO-Logic Semantics

Before we discuss the semantics itself, we need to introduce a lattice structure on the set of all subsets of a lattice. Let U be a lattice with the ordering $\leq_U$, the maximal element $\top_U$, and the minimal element $\perp_U$. Let $\mathcal{P}$ be as defined before, i.e., the power set operator. An associated ordering $\sqsubseteq_U$ on $\mathcal{P}(U)$ (also called Hoare's ordering [BunOh89]) is defined as follows: for any $X, Y \subseteq U$, $X \sqsubseteq_U Y$ if and only if for each element $x \in X$ there is an element $y \in Y$ such that $x \leq_U y$. This ordering on sets (in $\mathcal{P}(U)$) is only a preorder, because there may be a cycle. For example, $\{e\} \sqsubseteq_U \{e, \perp_U\} \sqsubseteq_U \{e\}$ and $\{\top_U\} \sqsubseteq_U U \sqsubseteq_U \{\top_U\}$. However, this is an acyclic ordering (partial ordering) on $\mathcal{P}(U)$ modulo the equivalence relation $\approx_U$, where $X \approx_U Y$ if and only if $X \sqsubseteq_U Y$ and $Y \sqsubseteq_U X$. Thus $\mathcal{P}(U)$ becomes a lattice modulo $\approx_U$ with the equivalence classes of $\{\top_U\}$ and the empty set $\{\}$ being the largest element and the smallest element respectively.

For a given pair of lattices, U and V, an ordering on the set of mappings $U \rightarrow V$ (denoted $Map(U, V)$) is defined as follows:

$$f \leq_{Map(U,V)} g \text{ iff for each } u \in U,\ f(u) \leq_V g(u).$$

Interpretation

For a language of RO-logic, its interpretation I is a tuple $\langle U, g_O, g_F, g_L, g_\Sigma \rangle$. U is a possibly infinite universe of all objects. The set U is a lattice with $\bot_U$ and $\top_U$ being the minimal and the maximal elements respectively and the ordering $\leq_U$. Recall that in the original O-logic, there is no lattice structure on the universe of objects. Function g_O is a lattice homomorphism $O \rightarrow U$, and g_F interprets each k-ary object constructor as a mapping $U^k \rightarrow U$ ($g_F : F \rightarrow (U^k \rightarrow U)$). The mapping g_L consists of $g_{L_\#}$ and $g_{L_\bullet}$ where $g_{L_\#} : L_\# \rightarrow (U \rightarrow U)$ and $g_{L_\bullet} : L_\bullet \rightarrow (U \rightarrow \mathcal{P}(U))$. Finally, function g_Σ maps each class to a subset of U, i.e., $g_\Sigma : \Sigma \rightarrow \mathcal{P}(U)$. Function g_Σ must respect the class lattice ordering: If $a \leq_\Sigma b$ then $g_\Sigma(b) \subseteq g_\Sigma(a)$ (a higher class is more restrictive) and whenever $u \in g_\Sigma(a) \cap g_\Sigma(b)$ then $u \in g_\Sigma(lub(a, b))$, where *lub* stands for the least upper bound.

Variable Assignment

A variable assignment, $\mathcal{V}$, is a mapping from object variables to U. Its extension to id-terms is defined below.

Let t be an id-term, then $\mathcal{V}(t)$ is defined as follows:

1. $\mathcal{V}(t) = \mathcal{V}(Y)$ if $t = Y$ and Y is a variable;
2. $\mathcal{V}(t) = \mathcal{V}(d) = g_O(d)$ if $t = d$ and $d \in O$;
3. $\mathcal{V}(t) = \mathcal{V}(f(\ldots, T, \ldots)) = g_F(f)(\ldots, \mathcal{V}(T), \ldots)$ if $t = f(\ldots, T, \ldots)$ and f is a function symbol ($f \in F$) and T is an id-term.

We can see that this definition is similar to that of O-logic (in Section 2.1.3.1, except that RO-logic has id-terms which are composed out of function symbols (object constructors), basic objects and variables.

The Meaning of a Formula

For an interpretation I and a variable assignment $\mathcal{V}$, we can define a function $M_{I,\mathcal{V}}$ that assigns meaning to each RO-term. Similarly to O-logic, determining the truth of an atomic formula under an interpretation I is equivalent to establishing the existence of the corresponding object in I, and this object should have the properties specified for it. As a consequence, an atomic formula is always true if no properties are specified and the object is of an appropriate class. Let function $M_{I,\mathcal{V}}$ assign the meaning to each RO-term, then

- For any simple RO-term $t = p : T$, $M_{I,\mathcal{V}}(p : T) = true$ iff $\mathcal{V}(T) \in g_{\Sigma}(p)$.
- For any complex RO-term, $t =$

  $p : T[\ldots,\ flab_i \rightarrow p_i : T_i[\ldots], \ldots,\ slab_j \rightarrow \{q_1 : S_1[\ldots],\ \ldots,\ q_m : S_m[\ldots]\},\ \ldots]$,

  $M_{I,\mathcal{V}}(t) = true$ if and only if

  1. $\mathcal{V}(T) \in g_{\Sigma}(p)$;
  2. for each single-valued label $flab_i$,
     $\mathcal{V}(T_i) \leq_U g_{L_{\#}}(flab_i)(\mathcal{V}(T))$ and $M_{I,\mathcal{V}}(p_i : T_i[\ldots]) = true$;
  3. for each set-valued label $slab_j$, $\{\mathcal{V}(S_1),\ \ldots,\ \mathcal{V}(S_m)\} \sqsubseteq_U g_{L_{\bullet}}(slab_j)(\mathcal{V}(T))$ and $M_{I,\mathcal{V}}(q_j : S_j[\ldots]) = true,\ j = 1,\ \ldots,\ m$.

For a formula, ϕ, that contains logical connectives or quantifications, its meaning (*i.e.*, $M_{I,\mathcal{V}}(\phi)$) is defined as in Predicate Calculus. An interpretation I, with respect to $\mathcal{V}$, is called a model of ϕ if $M_{I,\mathcal{V}}(\phi) = true$.

2.2.4 RO-Logic Databases

A database is defined as a set of formulae. If P is a set of formulae and ϕ is a formula, then $P \models \phi$ if and only if ϕ is true in every model of P (ϕ is logically implied by P).

For a given database which has a set of variables V and a set of basic objects O, a ***substitution*** σ is a mapping $\sigma : V \longrightarrow \{$*id-terms of P*$\}$, where the range are identities outside the domain of σ. The domain of σ is a finite set $dom(\sigma) \subseteq V$. Substitution σ commutes with object constructors (function symbols) as usual. Substitution is also extended to RO-terms and formulae by letting σ commute with logical connectives and assuming:

$$\sigma(p : T[\ldots,\ flab_i, \rightarrow t_i,\ \ldots,\ slab_j \rightarrow \{s_1,\ \ldots,\ s_k\},\ \ldots]) =$$
$$p : \sigma(T)[\ldots,\ flab_i,\ \rightarrow \sigma(t_i),\ \ldots,\ slab_j \rightarrow \{\sigma(s_1),\ \ldots,\ \sigma(s_k)\},\ \ldots].$$

For a formula ϕ and a substitution σ, the formula $\sigma(\phi)$ is an *instance* of ϕ. A substitution σ is called a *ground* substitution if for each $X \in dom(\sigma)$, $\sigma(X)$ is a ground id-term. The composition of substitution θ and μ is defined as usual, i.e., $\theta \circ \mu(X) = \theta(\mu(X))$.

2.2.5 RO-Logic Queries

A *query* is defined as a statement of the form $Q?$, where Q is an RO-term. For a given database, P, the set of *answers* to $Q?$ is the smallest set of ground RO-terms which is closed under the $\models$ defined before and contains all instances of Q logically implied by P.

2.2.6 Examples of RO-Logic Programs

Example 1 (from [KifWu89]):

The following example shows how RO-logic solves the quantification problem found in the original O-logic (see section 2.1.10). The example is about "interesting pair".

$$interestPair : P(\ emp \rightarrow employee : E, manager \rightarrow employee : M) \Leftarrow$$
$$employee : E(name \rightarrow string : N, works \rightarrow dept :$$
$$R(manager \rightarrow employee : M(name \rightarrow string : N)))$$

The rule is intended to say that a pair employee-manager is interesting if the employee's department's manager's name is the same as the employee's name. A universal quantification over variable P will not make any sense in this example. Thus, [Maier86] suggests the following quantification: $(\forall E)(\forall M)(\forall N)(\exists P)$ with the reason that a universal quantification on P would not make sense, and an object P exists for each $\langle E, M, N\rangle$ triple that matches the body of the rule. [KifWu89] argue that Maier's argument for this particular quantification is not quite clear and claim that the correct quantification is $(\forall E)(\forall M)(\exists P)(\forall N)$. [KifWu89] further say that Maier's quantification is true only if E and M functionally determine N. Note that, if the label *name* is set valued, the two quantifications will yield different results. There is no obvious way in choosing between the two quantifications based on only the syntactic structure of the rule above. [KifWu89] argue that the source of the problem is that the object variable P does not appear in the body, so that we cannot determine with certainty the correct quantification solely based on the rule's syntactic structure. Since O-logic does not have object constructors, we cannot connect P with the object variables in the body. This causes us to choose an ad hoc quantification. For this reason, object constructors are needed so that we can present explicitly the set of variables that determine the existential variable in the rule head. It is suggested in [KifWu89] that the correct representation of the problem is:

$$\begin{aligned}&\mathit{interestPair} : \mathit{namesake}(E, M)[\mathit{emp} \rightarrow \mathit{employee} : E, \mathit{manager} \rightarrow \mathit{employee} : M] \Leftarrow\\&\qquad \mathit{employee} : E[\mathit{name} \rightarrow \mathit{string} : N, \mathit{works} \rightarrow \mathit{dept} :\\&\qquad R[\mathit{manager} \rightarrow \mathit{employee} : M[\mathit{name} \rightarrow \mathit{string} : N]]]\end{aligned}$$

where *namesake* is an object constructor. This rule format is considered well defined and it corresponds to the intended meaning. This rule is also domain independent because all its head variables also appear in the body. The explicit use of object constructors is intended to clarify the quantification.

Example 2 (from [KifWu89]):

In order to show the relation between relational style data and object-oriented data of RO-logic, the following example shows how to map a first-order predicate into an RO-logic term.

For a given ground atomic formula of classical first-order logic,

$$p(a_1, \ldots, a_n)$$

the corresponding RO-term is

$$\tilde{p} : d[arg_1 \rightarrow a_1, \ldots, arg_n \rightarrow a_n]$$

where $\tilde{p}$ is the corresponding class for predicate p and d is a ground id-term.

In general, for an atom (not necessarily a ground atom) in Predicate Calculus:

$$p(T_1, \ldots, T_n)$$

the corresponding RO-term is

$$\tilde{p} : f_p(T_1, \ldots, T_n)[arg_1 \rightarrow T_1, \ldots, arg_n \rightarrow T_n]$$

where $\tilde{p}$ is the class corresponding to the predicate p, and f_p is a unique function symbol corresponding to p.

It should be noted here, that every ground fact $p(t_1, \ldots, t_n)$ corresponds to a unique object with the id $f_p(t_1, \ldots, t_n)$ i.e., the id-term depends on the values of all the corresponding object's attributes. This makes the formed (the resulting) RO-term *value oriented.* This example also shows that the mapping to RO-terms *allow us to use object identities, complex objects, and predicates in the same framework.* It also shows that RO-logic classes play a role similar to that of *predicate symbols* in Predicate Calculus.

2.2.7 Discussion of RO-Logic

The main differences between RO-logic and Maier's O-logic [Maier86] are

- In RO-logic, extensional databases are **not** identical to **interpretations**. Recall that in O-logic, a database is not a set of formulae but the structure of an interpretation for a set of formulae. The RO-logic follows the usual way (such as Datalog) i.e., the extensional database is a finite set of RO-terms (atomic formulae).
- The object constructors are explicit parts of RO-logic. Although this difference seems minor, this property is responsible for the elimination of problems related to the semantics of variables that only exist in the head part but not in the body part of an entity-creating rule in the original O-logic (see the example in the previous section);
- Inconsistency is included explicitly in RO-logic language by way of the meaningless object $\top$, which will replace inconsistent values of an attribute. This allows the system to reason about inconsistent data.

There are also some minor differences between RO-logic and the original O-logic as follows:

- In RO-logic, there is no distinction between basic objects (objects from primitive classes, these objects are also called data values in [Maier86]) and non-basic objects (objects that are not from primitive classes, which are called *entities* in O-logic). In O-logic [Maier86], basic objects are not allowed to have internal states. In RO-logic, every basic object, such as a numeral, is both an object and its own object identity, and there is no restriction on the semantics which prohibits a basic object from having internal states. For example, the following basic objects have properties:

$$int : 6[prop \rightarrow div : even,\ type \rightarrow types : integer]$$

$$\text{"}jim\text{"}[type \rightarrow types : charstring,\ length \rightarrow int : 3]$$

The reason for this is that Maier's restriction on atomic-data values is not intrinsic in the semantics of O-logic.

- Set-valued labels are included in RO-logic. A set in RO-logic may contain only id-terms (object identities) as its members but not other sets. However, since a set can be represented by an object identity (strictly speaking, this set appears as a value of that object identity's set-valued label), then a set of an arbitrary depth can be modelled. This is a useful feature in representing complex objects.

- Complex objects are constructed by combining the usual tuple constructors and set constructors which create new kinds of objects. For example (from [KifWu89]):

  $john\text{-}children[father \rightarrow john,\ kids \rightarrow \{bill, mary\}]$

  is structurally different from

  $john[name \rightarrow \text{"}John\text{"},\ kids \rightarrow children\text{-}of\text{-}john[val \rightarrow \{bill, mary\}]]$.

  The difference is that the latter object can be obtained by first applying the set constructor to form $children\text{-}of\text{-}john[val \rightarrow \{bill, mary\}]$ and then the tuple constructor to form the object *john*. This sequence is not applicable to the former one. In the former object the second component, $\{bill, mary\}$, is not an RO-term by itself, because it does not have any associated identity. Therefore, it cannot be constructed independently of the object *john-children*. This component corresponds to *weak entities* in entity-relationship models.

2.2.8 Concluding Remarks on RO-Logic

The proof of the soundness and the completeness of a resolution-based proof procedure in RO-logic is sketchily given in [KifWu89] although some theorems are given without proofs.

The presented logic remedies the quantification problems associated with variables that appear only in the head part but not in in the body part of an entity-creating rule in O-logic (see Section 2.1.10). RO-logic also extends O-logic in several directions such as incorporating sets and inconsistent information.

The concepts of object-oriented databases that RO-logic captures are: object, object-identity, attributes, class and class hierarchy. Similar to O-logic, two important concepts of object-oriented databases are still missing: *methods* and *inheritance*.

2.3 C-Logic [CheWa89]

This section summarizes C-logic [CheWa89]. This logic can be viewed as an extension of O-Logic [Maier86] in the sense that it solved the quantification problems found in entity-creating rules of O-Logic. It should be noted that C-logic [CheWa89] and RO-logic [KifWu89] were published at about the same time.

2.3.1 Introduction

The objective of the development of C-logic is to design a logical framework for natural representation and manipulation of complex objects [CheWa89]. Its design was stimulated by the development of O-logic [Maier86]. C-logic solves the quantification-related problems found in O-logic (see Section 2.1.10). In addition, C-logic has some features such as multi-valued labels, a dynamic notion of types (which will be explained later in subsection 2.3.2.3), and provides a relatively simple framework for exploring efficient logic deduction over complex objects.

Chen and Warren claim that the common characteristic in most previous proposals of reasoning mechanisms on complex objects is that they are language based [CheWa89]. They mean that a language is first presented for describing complex objects, and then the semantics of the language is given. There are two disadvantages to this language-based approach mentioned in [CheWa89]. First, different languages have different features and it is not always clear why those features are needed and how they support complex objects. Moreover, the same feature may have subtle semantic differences in different languages. This situation causes difficulties when comparing the semantics of complex objects across different languages. Second, each language may make seemingly unnecessary constraints over complex object specifications. Such constraints can make the associated language inflexible.

The design of C-logic is based on *what needs to be modeled.* The main purpose of complex object modeling is to capture more of the structure of real world data. The strategy in designing C-logic was to focus on complex object modeling instead of specific languages. The design of C-logic was started with an analysis of semantic modeling of complex objects. Based on this analysis, C-logic was designed to support what the designers [CheWa89] considered to be essential features of complex objects including object identity, multi-valued labels and a dynamic notion of types. Other guiding principles in its design were simplicity and flexibility.

C-logic programs can be translated into first-order logic. Some other higher-order features such as single-valued labels and a static notion of types (which will be explained later in Subsection 2.3.2.3) can be added to C-logic.

2.3.2 C-Logic Complex-Object Modeling

Complex objects are intended to model structured entities in the real world. In this subsection we describe how, within the C-logic framework, we represent entities, their properties, and classes by object identities, labels and typing.

2.3.2.1 Object Identity

In O-logic, object entities are referred to by *object variables* only. This is not adequate, especially when objects are defined by rules (entity creating rules). Consider the following example [CheWa89]:

$$\begin{aligned}
&\mathit{path} : C(\mathit{src} \rightarrow X, \mathit{dest} \rightarrow Y, \mathit{length} \rightarrow 1) \Leftarrow \\
&\qquad \mathit{node} : X(\mathit{linkto} \rightarrow Y) \\
&\mathit{path} : C(\mathit{src} \rightarrow X, \mathit{dest} \rightarrow Y, \mathit{length} \rightarrow L) \Leftarrow \\
&\qquad \mathit{node} : X(\mathit{linkto} \rightarrow Z),\ \mathit{path} : C'(\mathit{src} \rightarrow Z, \mathit{dest} \rightarrow Y, \mathit{length} \rightarrow L'), \\
&\qquad L\ \mathit{is}\ L' + 1
\end{aligned}$$

In O-logic, the above rules would be considered as entity-creating rules. These rules do not determine what entities are to be created, that is how variable C should be quantified with respect to other variables in the rules. It is pointed out in [Maier86] that in a rule such as that above, C should be existentially quantified, but its scope is not specified explicitly. To interpret the rules above, we could have three possible interpretations [CheWa86]: The *path* objects can be determined by either:

1. node objects at both ends only, i.e., X and Y; or
2. node objects at both ends and the length of the *path*, i.e., X, Y and L; or
3. the sequence of node objects along the *path*, i.e., X, ..., Y.

For example, consider the variables in the second *path* rule in the above example. We can have three different interpretations as enumerated above. In the first case, the quantification would be $\forall X \forall Y \exists C$, in the second case: $\forall X \forall Y \forall L \exists C$, and in the third case: $\forall X \forall C' \exists C$. We cannot express these three different interpretations in O-logic, in other words, we cannot write explicitly our intended semantics. C-logic solves the problem by requiring the user to specify the intended semantics i.e., by allowing an explicit construction of an object identity. However, the user needs to specify only the variables on which the object identity is dependent. The Skolem function of the existential variable could be given explicitly by the user, or generated automatically by the system. The following example shows how the previous rules are converted to C-logic when *path* objects are determined by node objects at both ends only, i.e., the first case in the three possible interpretations above.

$$
\begin{aligned}
&path : id(X, Y)[src \rightarrow X, dest \rightarrow Y, length \rightarrow 1] \Leftarrow \\
&\qquad node : X[linktto \rightarrow Y]
\end{aligned}
$$

$$
\begin{aligned}
&path : id(X, Y)[src \rightarrow X, dest \rightarrow Y, length \rightarrow L] \Leftarrow \\
&\qquad node : X[linkto \rightarrow Z], \\
&\qquad path : C'[src \rightarrow Z, dest \rightarrow Y, length \rightarrow L'], \\
&\qquad L \; is \; L' + 1
\end{aligned}
$$

2.3.2.2 Using Labels for Representing Properties

An object's properties are represented as labeled values. In O-logic, labels are interpreted as *partial functions* from objects to objects, so that it has the following built-in functionality constraint: Whenever a partial function is defined it yields a single value. Thus, if a program has a multi-valued label, the program will not have any model. An example is given by the label *name* in the following C-terms:

$j[name \rightarrow \text{“}John\text{”}]$

$j[name \rightarrow \text{“}John\ Doe\text{”}]$

The label *name*, from a semantic point of view should yield two values, thus it cannot be a function. In C-logic, labels are considered as *binary predicates* over objects, so that the logic can model multi-valued labels and simulate sets. Multi-valued labels are very useful in many cases. For example, a person may have several children, or several addresses.

Another implication of viewing labels as binary predicates is as follows. Representation of an object with several labeled values can be viewed as a conjunction of several atomic formulae as illustrated in the following example.

$$j[name \rightarrow \text{“}John\ Doe\text{”},\ age \rightarrow 28]$$

can be expressed as:

$$j[name \rightarrow \text{“}John\ Doe\text{”}] \wedge j[age \rightarrow 28]$$

or in standard first-order logic:

$$name(j, \text{“}John\ Doe\text{”}) \wedge age(j, 28)$$

This property is helpful in translating C-logic formulae into standard first-order logic formulae.

2.3.2.3 Type

In databases, we can divide objects into various classes according to their properties. According to [CheWa89], there are *dynamic* and *static* aspects of a class in databases. Within the dynamic aspect, a class denotes a set of objects in the class, and such memberships may be changed by database updates. Within the static aspect, a class represents the common structural properties of all objects in a class i.e., which properties an object should have in order to belong to a certain class. It is argued in [CheWa89] that corresponding to these two aspects are the dynamic and static notions of types.

In the dynamic notion of types, a type is only a set of object identities, and it is essentially part of the database state. There are no further assumptions about what properties a member object should have. Instead, whenever we add an object to the database, we also must specify its type. In some cases we may use a default type such as *object* that includes all objects.

In the static notion of types, a type denotes a set of properties which must be possessed by objects of that type. For example, let $l_1, \ldots, l_n$ be labels corresponding to all properties indicated by a type τ, then one possible meaning of τ is a set of objects specified as follows:

$$\tau(X) \Leftarrow X[l_1 \rightarrow X_1, \ldots, l_n \rightarrow X_n]$$

By this rule, every object with all properties specified by type τ will *automatically* belong to that type.

We can organize types into hierarchies. In a dynamic notion, the type hierarchy should be specified explicitly. However, in a static notion of types, the hierarchy is implicitly determined by the properties of every type.

C-logic uses the dynamic notion of types. The main reason is to make the framework simpler. It is simpler because the static notion is some kind of constraint which is better specified by the database's schema or by other constraints over the database state, for example, the constraint regarding the functionality of labels [CheWa89].

It should be noted that there is not any explanation or pointer in [CheWa89] about the distinguishing characteristics between the meaning of the word *class* and the word *type* used in the paper, eventhough the word class is used for introducing the notions of types. Moreover, if we look at the informal definitions of class and type in [CheWa89] as summarized above, they look the same. Furthermore, from the observation of the way types are used in C-logic syntax and semantics, we cannot see its differences from the way classes are used in O-logic or RO-logic.

2.3.3 C-Logic Syntax

The alphabet of a C-logic language consists of:

1. a countable set V of variables;
2. a countable set of function symbols;
3. a countable (possibly empty) set of predicate symbols;
4. a countable (possibly empty) set of labels;
5. a countable set of type symbols, which contains a type symbol *object*. The set of type symbols is partially ordered by a given relation $\leq$ such that for any type τ, $\tau \leq object$,
6. logical connectives: $\wedge$, $\vee$, $\neg$, $\Leftarrow$, $\forall$, $\exists$;
7. auxiliary symbols: "[", "]", "(", ")", "{", "{", "$\rightarrow$", "," , "$\leq$".

It is assumed that these sets of symbols are disjoint.

2.3.3.1 C-Terms

Let τ be any type symbol. A *C-term* is defined as one of the following:

- $\tau : X$

  where X is a variable.

- $\tau : c$

  where c is a constant symbol;

- $\tau : f(t_1, \ldots, t_n)$

  where f is an n-ary function symbol, and t_i, $1 \leq i \leq n$, are C-terms;

- $t[l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n]$

  where $n \geq 1$ and

  1. t is a C-term of one of the three forms: $\tau : X, \tau : c$, or $\tau : f(t_1, \ldots, t_k)$;
  2. l_i, $1 \leq i \leq n$ is a label; and
  3. e_i, $1 \leq i \leq n$ is either a C-term or a set of C-terms of the form $\{t'_1, \ldots, t'_k\}$, $1 \leq k$, in which $t'_1, \ldots, t'_k$ are C-terms.

As an abbreviation, a C-term of the form $object : t$ can be written as t.

The following are examples of C-terms:

$object : X$

$path : p(X, Y)[length \rightarrow 7]$

$person : joe[children \rightarrow \{person : mark, person : mike\}]$

The first example above is read intuitively as a variable X from type *object*. The second example represents $p(X, Y)$ as an object of type *path* with *length* $= 7$.

2.3.3.2 C-Formulae

An ***atomic formula*** is either a C-term or $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are C-terms. As usual, formulae are constructed from atomic formulae by using logical connectives and quantifiers. Recall that O-logic does not have formulae containing predicate symbols.

2.3.4 C-Logic Semantics

Given a language L of C-logic, the semantic structure ST is a pair $\langle U, g\rangle$, where U is a non-empty set of all objects called ***the domain of the structure*** (or the universe) and g is the ***interpretation function*** that assigns values to non-logical symbols in L. The interpretation function g is defined as follows:

- $g(f) \in (U^n \rightarrow U)$ for every n-ary function symbol f;
- $g(p) \in \mathcal{P}(U^n)$ for every n-ary predicate symbol p (where $\mathcal{P}$ is as defined before, i.e., the power set operator);
- $g(l) \in \mathcal{P}(U^2)$ for every label l;
- $g(\tau) \in \mathcal{P}(U)$ for every type symbol τ, such that for any two type symbols τ_1 and τ_2, if $\tau_1 \le \tau_2$, then $g(\tau_1) \subseteq g(\tau_2)$.

Semantically, a label is the same as a binary predicate, and a type is the same as a unary predicate. However, a label and a type are different from predicates in C-logic languages. This is because labels and types can appear inside C-terms, and this is not the case with predicates.

Variable Assignments

Formulae and C-terms may contain free variables, thus their semantics may depend on variable assignments. A C-term could have two meanings because it has two purposes: for denoting an object and for indicating whether the denoted object satisfies certain properties.

Given a language L of C-logic, let ϕ be a formula in L, U be the universe, g be the interpretation function, $ST = \langle U, g \rangle$ be a semantic structure for L, and $\mathcal{V} : V \longrightarrow U$ be a variable assignment, i.e., a function from the set V of all variables into the domain U of ST.

Let *Term* denote the set of all C-terms in L and τ denote a type. The extension $\bar{\mathcal{V}} : Term \longrightarrow U$ of a variable assignment $\mathcal{V}$ to the set of all C-terms is defined recursively as follows:

- For each variable X, $\bar{\mathcal{V}}(\tau : X) = \mathcal{V}(X)$.
- For each constant symbol c, $\bar{\mathcal{V}}(\tau : c) = g(c)$.
- If $t_1, \ldots, t_n$ are C-terms and f is an n-ary function symbol, then

$$\bar{\mathcal{V}}(\tau : f(t_1, \ldots, t_n)) = g(f)(\bar{\mathcal{V}}(t_1), \ldots, \bar{\mathcal{V}}(t_n)).$$

- if t is a C-term of the form $\tau : X$, $\tau : c$, or $\tau : f(t_1, \ldots, t_m)$, and l_i, $1 \leq i \leq n$ is a label, then

$$\bar{\mathcal{V}}(t[l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n]) = \bar{\mathcal{V}}(t)$$

  where e_i, $1 \leq i \leq n$, is a C-term or a set $\{t'_1, \ldots, t'_{n_i}\}$ of C-terms.

Given a semantic structure ST, the definition of the satisfaction of formula ϕ by structure $ST = \langle U, g \rangle$ (where U and g are as before) and variable assignment $\mathcal{V}$, $(ST, \mathcal{V}) \models \phi$ is as follows:

- For every variable X, $(ST, \mathcal{V}) \models \tau : X$ if $\mathcal{V}(X) \in g(\tau)$.

- For every constant symbol c, $(ST, \mathcal{V}) \models \tau : c$ if $g(c) \in g(\tau)$.
- If $t_1, \ldots, t_n$ are C-terms and f is an n-ary function symbol, then $(ST, \mathcal{V}) \models \tau : f(t_1, \ldots, t_n)$ if $\bar{\mathcal{V}}(\tau : f(t_1, \ldots, t_n)) \in g(\tau)$ and $(ST, \mathcal{V}) \models t_i$ for every i, $1 \leq i \leq n$.
- If t is a C-term of the form $i : X, \tau : c$, or $\tau : f(t_1, \ldots, t_m)$, and l_i, $1 \leq i \leq n$, is a label, then $(ST, \mathcal{V}) \models t[l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n]$ if $(ST, \mathcal{V}) \models t$, and for every e_i, is either
  - e_i is a C-term, $(ST, \mathcal{V}) \models e_i$ and $\langle \bar{\mathcal{V}}(t), \bar{\mathcal{V}}(e_i) \rangle \in g(l_i)$; or
  - e_i is a set of C-terms of the form $\{t_1^i, \ldots, t_k^i\}$, $1 \leq k$, and for every j, $1 \leq j \leq k$, $(ST, \mathcal{V}) \models t_j^i$ and $\langle \bar{\mathcal{V}}(t), \bar{\mathcal{V}}(t_j^i) \rangle \in g(l_i)$.
- If $t_1, \ldots, t_n$ are C-terms and p is an n-ary predicate symbol, then $(ST, \mathcal{V}) \models p(t_1, \ldots, t_n)$ if $(ST, \mathcal{V}) \models t_i$ for every i, $1 \leq i \leq n$ and $\langle \bar{\mathcal{V}}(t_1), \ldots, \bar{\mathcal{V}}(t_n) \rangle \in g(p)$.

The meaning of a more general formula, with logical connectives or quantifiers, is defined from its atomic formulae in the usual way.

Similar to O-logic and RO-logic, a complex object description can be decomposed into atomic descriptions and various pieces of descriptions can be combined into a complex one. This is made possible by the following semantic equivalencies. A C-term of the form $t[l_1 \rightarrow t_1, \ldots, l_n \rightarrow t_n]$ is semantically equivalent to $t[l_1 \rightarrow t_1] \wedge \ldots \wedge t[l_n \rightarrow t_n]$; and a C-term of the form $t[l \rightarrow \{t_1, \ldots, t_n\}]$ is semantically equivalent to $t[l \rightarrow t_1] \wedge \ldots \wedge t[l \rightarrow t_n]$.

It is claimed in [CheWa89] that any formula of a C-logic language can be transformed into an equivalent genuine first-order logic formula. This transformation is used to provide the basis for implementing complex object reasoning in first-order logic. Thus, while providing a framework for natural representation and manipulation of complex objects, a language of C-logic still has first-order s[illegible]antics; this fact makes C-logic computationally attractive.

2.3.5 Specification and Computation of Complex Objects in C-Logic

C-logic allows us to use a logic programming approach to complex objects. This means we can specify complex objects by facts and rules in the logic, and reasoning or computation of complex objects can be performed as logical inferences.

For a given language of objects, a clausal subset of formulae is defined as follows:

- A *literal* is either an atomic formula or the negation of an atomic formula.
- A *clause* is a disjunction of literals

  $$L_1 \vee L_2 \vee \ldots \vee L_n$$

  where all variables are implicitly universally quantified at the outer-most level.
- A *definite clause* is a clause that contains only one positive literal, i.e., $A \vee \neg B_1 \vee \ldots \vee \neg B_m$ ($m \geq 0$), which is also written as

  $$A \Leftarrow B_1, \ldots, B_m$$

  where $A, B_1, \ldots, B_m$ are atomic formulae.
- A *negative clause* is a clause that contains no positive literals, that is, $\neg B_1 \vee \ldots \vee \neg B_m$, which is also written as

  $$\Leftarrow B_1, \ldots, B_m$$

  A negative clause is also called *a query* or *a goal*.

It is assumed in the definition of C-logic languages that each language should have a countable partially ordered set of type symbols with the greatest type *object* which is a super type of any other type. The user is expected to specify the ordering among other type symbols. A user may specify a *subtype declaration* as follows

$$\tau_1 \leq \tau_2$$

where τ_1 and τ_2 are type symbols.

A program is a finite set of subtype declarations and definite clauses.

2.3.6 An Example of C-Logic Program

The following example from [CheWa89] shows a program that defines objects of type *nounphrase.*

name : *john*

name : *bob*

determiner : *the* [*num* → { *singular,plural* }, *def* → *definite*]

determiner : *a* [*num* → *singular, def* → *indef*]

determiner : *all* [*num* → *plural, def* → *indef*]

noun : *student* [*num* → *singular*]

noun : *students* [*num* → *plural*]

proper-np : *X*[*pers* → 3, *num* → *singular, def* → *definite*] ⇐ *name* : *X*

common-np : *np*(*Det, Noun*)[*pers* → 3, *num* → *N*, *def* → *T*] ⇐

  determiner:Det [*num* → *N*, *def* → *T*],

  noun: Noun [*num* → *N*]

$\textit{proper-np} \leq \textit{noun-phrase}$

$\textit{common-np} \leq \textit{noun-phrase}$

If we give the following query to find instances of ***noun-phrase*** which have ***plural*** as the value of their ***num*** labels,

$\Leftarrow \textit{noun-phrase} : X[\textit{num} \rightarrow \textit{plural}]$

we will get two answers for X :

$np(the, students)$ and $np(all, students)$.

2.3.7 Multi-Valued Labels and Sets in C-Logic

Although set manipulation is regarded as an important feature for any system of complex objects, it is not clear how this feature can be supported in a simple logical framework. C-logic is still first-order logic, so that there are no set values. However, multi-valued labels allow us to use the concept of sets under the following condition: by accepting the fact that a set value cannot be bound to a variable and the equality among sets cannot be checked.

The following example illustrates the use of sets in C-logic:

the fact:

$person : mike[children \rightarrow \{noah, adam, eve\}]$

and a query written as:

$\Leftarrow person : mike[children \rightarrow \{X, Y\}]$.

Since the semantics interprets labels as binary predicates (see section 2.3.4) the above fact and query can be transformed into the following:

$person : mike[children \rightarrow noah,\ children \rightarrow adam,\ children \rightarrow eve]$

and

$\Leftarrow person : mike[children \rightarrow X, children \rightarrow Y]$

Variables X and Y will be bound to each *noah*, *adam*, *eve* to make the query succeed. Users still can think of a label intuitively as a set-valued function. It is expected that users still can do most set manipulations required. Some manipulations that C-logic cannot do are returning a set value and checking the equality of sets.

2.3.8 Discussion of C-Logic

The main differences between C-logic and O-logic are:

- In O-logic, only object variables are introduced for referring to object identities. In C-logic, similar to RO-logic, an object is denoted by a C-term (called id-term in RO-logic) constructed from a composition of a function symbol, constant symbols, and variables. The idea of using object constructors for forming object identities in C-logic and RO-logic is the same (it should be noted that the two papers appeared at about the same time). As mentioned before, this object-constructor idea eliminates the quantification problems found in O-logic.

- Labels in C-logic are interpreted as binary relations. Each label can be used as single valued or set valued (no syntactic distinction), and basically each label is set valued. This is different from that of O-logic and RO-logic. In O-logic, all labels are single valued, and interpreted as partial functions from objects to objects. In RO-logic, there are two kinds of labels: single-valued labels and set-valued labels.

  We think C-logic will have problems when what a user really wants is a single-valued label, in which case if there is an inconsistency the logic cannot detect it. This is because basically C-logic has only set-valued labels.

2.3.9 Concluding Remarks on C-Logic

C-logic captures several concepts of object-oriented databases, i.e., object-identities, multi-valued attributes (labels), type and type-hierarchy. However, it does not capture two of the object-oriented database core concepts specified in [Kim90]. The missing two concepts are *inheritance* and *methods.* These two concepts are captured by F-logic [KifLa89] which is summarized in the next section.

2.4 F-Logic [KifLa89]

In this section, we present a summary of F-logic paper [KifLa89], a significant improvement over O-logic [Maier86], RO-logic [KifWu89] and C-logic [CheWa89] in terms of object-oriented properties captured.

2.4.1 Introduction

Frame logic [KifLa89], also called F-logic, is a database logic that provides most object-oriented database concepts, such as object identities, complex objects, attributes, methods, typing, and inheritance. The name Frame logic (abbreviated, F-logic) was derived from frame-based languages [Minski81, FiKe85] that are used in Artificial Intelligence. F-logic has implications for the frame-based languages. Essentially, frame languages can be viewed as scaled-down versions of complex objects with identity, inheritance, and deduction [KifLa89].

To make a deduction system able to reason about inheritance and a database schema, some capabilities of higher-order logics are required [KifLa89]. However, there are some practical problems with higher-order logics. F-logic's solution is to have a logic with syntax that has an appearance of a higher-order logic. However, unlike a higher-order logic, F-logic is still tractable and has a natural direct first-order semantics [KifLa89]. This logic is the result of an extension of RO-logic [KifWu89]. It eliminates distinctions among objects, classes and relationships, in order to allow reasoning about inheritance, methods and schemata.

2.4.2 F-Logic Syntax

The alphabet of an F-logic language consists of

1. a set O of basic objects. This set is partially ordered with $\top$ as the unique maximal element, and $\bot$ as the unique minimal element (O is a complete lattice);
2. a set F of object constructors;
3. a countable set V of variables;
4. logical connectives: $\vee, \wedge, \neg, \Leftarrow$, and quantifiers: $\forall, \exists$;
5. a set of auxiliary symbols: " : ", ")", "(", "]", "[", "}", "{", ",", " $\rightarrow$ ".

Note that, in F-logic, function symbols are called object constructors. In F-logic, *objects* are either complex objects or classes of objects. Unlike O-logic [Maier86], RO-logic [KifWu89] and C-logic [CheWa89], F-logic puts classes and instances in the same category. For example, basic object *assistant* $\in O$ can be viewed as representing the class of assistants (i.e., every individual assistant is an instance of basic object *assistant*). Basic object *assistant* can also be viewed as an instance of its superclass *student* $\in O$. Thus, a class is viewed as an instance of its superclass, rather than as its subclass (this view is similar to that of frame languages). According to [KifLa89], this particular feature allows *inheritance* to be built naturally.

The ***constant*** symbols of F-logic are the basic objects (elements of O). The object constructors (i.e., elements of F) are ***function*** symbols of arity ≥ 1 which are used to construct new objects. For convenience, it is assumed that O and F are disjoint, although basic objects can be viewed as 0-ary object constructors.

Id-Term

An *id-term* is a term composed, in the usual way, of function symbols (object constructors), constant symbols (basic objects, i.e., elements of O), and variables. A basic object (an element of O) is also an id-term.

The symbol O^* denotes the set of all *ground (variable free) id-terms.* O^*, essentially, plays the role of Herbrand Universe of F-logic. In F-logic, ground id-terms should be seen as objects themselves or abstractions of objects which are commonly called object identities. Thus object identities are either basic objects or non-basic objects (recall that non-basic objects are denoted by id-terms other than constant symbols). In F-logic, objects (object identities, strictly speaking) that are represented by id-terms other than constant symbols are perceived as being constructed from simpler objects.

In O-logic and RO-logic, *objects* and *labels* belong to distinct categories. Due to this distinction, it is not easy to treat objects and relationships (identified by labels) in a uniform framework [KifLa89]. In contrast, in F-logic. an object identity can be viewed as an entity or a relationship depending on syntactic position of the object identity in a formula.

As a label, every object identity has a *type*. There are two types of labels: *single-valued* (functional) and *set-valued* labels. According to this types, the set O^* of ground id-terms is partitioned into two disjoint sets: $L_{\#}$ (functional labels) and $L_{\bullet}$ (set-valued labels). To make specifying and checking types in O^* easy, this partition is required to be ***congruent*** in the following sense:

$$\text{partition } O^* = L_{\#} \cup L_{\bullet} \text{ is congruent } \textit{if and only if}$$

$$\forall t_1, s_1, \ldots, t_n, s_n \in O^*,\ \forall f \in F,\ \forall i = 1, \ldots, n$$

$$\textit{if either } (\text{both } t_i, s_i \in L_{\#}) \textit{ or } (\text{both } t_i, s_i \in L_{\bullet}) \textit{ then either}$$

$$(\text{both } f(t_i, \ldots, t_n), f(s_i, \ldots, s_n) \in L_{\#}) \textit{ or } (\text{both } f(t_i, \ldots, t_n), f(s_i, \ldots, s_n) \in L_{\bullet})$$

Assigning types to elements of O^* (the set of ground id-terms) consists of the following processes:

1. assigning a type (i.e., single-valued or set-valued type) to each element of O;
2. specifying the type of ground id-term $f(t_1, \ldots, t_n)$ for every n-ary function symbol $f \in F$ and for each n-tuple $\langle t_1, \ldots, t_n \rangle$ where each $t_j\,(1 \le j \le n)$ is either $t_j \in L_{\#}$ or $t_j \in L_{\bullet}$.

In practice, this procedure is considered effective because the cardinalities of O and F are finite [KifLa89]. Checking the type of a term $t \in O^*$, is done by substituting types for subterms, starting with constant symbols. This procedure requires linear time as a function of term sizes. Moreover, in practice, we might only need to assign one type to the range of each function symbol. This makes type specifications linear in the sizes of O and F, and type checking can be executed in constant time by checking the range type of the outermost function symbol.

2.4.2.1 F-Terms

For convenience, ground F-terms are denoted by names starting with lower-case letters, while non-ground F-terms are denoted by names starting with capital letters. *An F-term* , which is similar to [Maier86] and [KifWu89]'s idea, is defined as either

1. a simple F-term,

$$P : Q$$

   where P and Q are id-terms. Intuitively, Q is an instance of class P, for example $student : john$; or

2. a complex F-term,

$$P : Q[Flab_1 \rightarrow T_1, \ldots, Flab_m \rightarrow T_m,$$
$$Slab_1 \rightarrow \{S_{1,1}, \ldots, S_{k_1,1}\}, \ldots, Slab_l \rightarrow \{S_{1,l}, \ldots, S_{k_l,l}\}].$$

   P and Q are id-terms; $Flab_i$ and $Slab_j$ are also id-terms, but they are named differently to emphasize their role as labels. The order of labels in an F-term is immaterial. T_i and $S_{n,j}$ denote *F-terms*.

A complex F-term of the form (2) above can be viewed intuitively as a statement about object Q: This object Q is an instance of class P and has properties specified by the labels $Flab_1, \ldots, Flab_m$, $Slab_1, \ldots, Slab_l$. When no labels are specified for an object Q (i.e., $P : Q[]$), we may omit the brackets, so that it is equivalent to a simple F-term $P : Q$.

In order to have the feature of a higher-order logic without having the overhead of higher order calculus, class membership is modelled by means of a *lattice ordering* instead of a true set-theoretic membership. Formally, the elements of O^* are organized in a complete lattice with ordering $\prec_O$. The ordering $\preceq_O$, as usual, stands for $\prec_O$ or $=$. The lattice has two special elements: *the maximal element* $\top$ and *the minimal element* $\bot$. The maximal element is a meaningless object that represents a class with no instances, while the minimal element represents the biggest class (the unknown object). The lattice is the static part of the language and it can be perceived as a part of a schema definition. It represents the transitive closure of the "subclass of" and the "instance of" relationships among classes. Thus, $p \prec_O q$ (such as ***student*** $\prec_O$ ***assistant***) means that q is a subclass or an instance (possibly indirect, i.e., there may be objects in between) of p. This ordering relation in the lattice is similar to information ordering of denotational semantics. It should be noted again that the same object can have both roles as an instance and as a class depending on its syntactic position. For example, $b \in O^*$ appears in the "instance position" in $a : b[\ldots]$ and in the "class position" in $b : c[\ldots]$.

The following *monotonicity* restriction is imposed on the lattice O^*:

$$\text{if } t_1 \preceq_O s_1, \ \ldots, t_n \preceq s_n \text{ then } f(t_1, \ \ldots, t_n) \preceq f(s_1, \ \ldots, s_n)$$

In other words, object constructors are required to be monotonic functions on the lattice. For example, if $person \prec_O jean$ (jean is is an instance of a person), then $book(person) \prec_O book(jean)$ (Jean's books belong to the class of books owned by persons). This monotonicity is required for two reasons: first, to make the resolution procedure complete (not further explained in [KifLa89]), second, to ensure that the lattice structure in O^* can be given effectively as part of schema specifications and that the "instance-of" relation can be verified efficiently.

2.4.2.2 F-Formulae

In F-logic, an *F-term* is also an *atomic formula.* An F-logic language consists of a set of formulae. A formula is built from atomic formulae by means of logical connectives and quantifiers in the usual way.

The following convention is assumed for the notation. If a single-valued label Lab is omitted in an F-term then it is equivalent to an F-term that has component $Lab \rightarrow \bot : \bot$. Similarly if a set-valued label Lab' is omitted in an F-term, then it is equivalent to an F-term that has component $Lab' \rightarrow \{\}$ (a set valued label whose value is the empty set). Finally, if a class specification in an F-term is omitted then class $\perp$ is assumed. For example:

$csd[name \rightarrow string : \text{“}ComputerScience\text{”}]$, and

$\perp : csd[name \rightarrow string : \text{“}ComputerScience\text{”}, budget \rightarrow \perp : \perp, students \rightarrow \{\}]$

are considered to be the same F-term. Based on this convention, id-terms can be viewed as a special case of F-terms by identifying p and $\perp : p[]$ as the same F-term.

2.4.3 F-Logic Semantics

An ordering, namely Hoare's ordering [BunOh89], is used in F-logic. For a given lattice U with partial ordering $\preceq_U$, the maximal element $\top_U$, and the minimal element $\perp_U$, the preorder $\sqsubseteq_U$ on the power set $\mathcal{P}(U)$ is defined as follows: for any pair of subsets $X, Y \subseteq U$, $X \sqsubseteq_U Y$ iff for every element $x \in X$ there is $y \in Y$ such that $x \preceq_U y$. The power set $\mathcal{P}(U)$ can be considered as a lattice modulo the equivalence relation $\approx_U$, where $X \approx_U Y$ iff $X \sqsubseteq_U Y$ and $Y \sqsubseteq_U X$. In this lattice, the minimal element is the equivalence class of $\{\}$ (the empty set) and the maximal element is the equivalence class of $\{\top_U\}$.

The following is the definition of a lattice structure on the set of mappings. Given a pair of lattices, U and V, then a lattice structure on the set of mappings $U \rightarrow V$, which is denoted $Map(U, V)$, is defined as: $f \preceq_{Map(U,V)} g$ if for every $u \in U$, $f(u) \preceq_V g(u)$. Here, there are two particularly important types of lattice mappings: *monotonic* which is denoted $Mon(U, V)$ and *homomorphic* (the one that preserves *lub* (least upper bound) and *glb* (greatest lower bound)), which is denoted $Hom(U, V)$. This set of homomorphic mappings $Hom(U, V)$ has the following properties: For each

$f \in Hom(U, V)$, we have:

$$f(lub(X)) = lub\{f(x) | x \in X\}, \text{ for each } X \subseteq U, \text{ and}$$

$$f(glb(X)) = glb\{f(x) | x \in X\}, \text{ for each } X \subseteq U.$$

Interpretation

The semantics of F-logic is defined as the following. Given an F-logic language, its interpretation, I, is a tuple $\langle U, g_O, g_F, J_{\#}, J_{\bullet} \rangle$, where U is the universe of all objects, which is required to have a lattice structure with the maximal element $\top_U$, the minimal element $\perp_U$, and with a lattice ordering $\preceq_U$. U is partitioned into a pair of subsets $U_{\#}$ (which is associated with functional labels) and $U_{\bullet}$ (which is associated with set-valued labels) according to the types of elements of O^*. The elements of O^* can be perceived as the ***object identities (or names)*** of objects, and the elements of U can be thought of as the objects themselves (in the possible world I).

The homomorphic mapping, as defined above, $g_O : O^* \longrightarrow U$ is defined as interpreting objects of O^* by elements of U, so that $g_O(L_{\#}) \subseteq U_{\#}$ and $g_O(L_{\bullet}) \subseteq U_{\bullet}$ (see example in Subsection 2.4.7). The mapping $g_F : F \longrightarrow Mon(U^k, U)$ interprets each k-ary object constructor $f \in F$ by a monotonic mapping $U^k \longrightarrow U$. The mappings g_O and g_F are related as follows:

$$\text{if } t = f(s_1, \ldots, s_n) \in O^* \text{ then } g_O(t) = g_F(f)(g_O(s_1), \ldots, g_O(s_n)).$$

Constant symbols (basic objects) and function symbols (object constructors) are essentially interpreted in the same way as in Predicate Calculus. The difference is the existence of lattice structures on the set of ground F-terms O^*, and on the domain U.

Label objects (objects that play roles as labels) are interpreted by associating appropriate mappings to elements of U using functions $J_{\#}$ and $J_{\bullet}$. The function

$J_\# : U_\# \rightarrow Mon(U, U)$ associates a monotonic mapping $U \rightarrow U$ with each element of $U_\#$, and $J_\bullet : U_\bullet \rightarrow Mon(U, \mathcal{P}(U))$ associates a monotonic mapping $U \rightarrow \mathcal{P}(U)$ with each element of $U_\bullet$. The ordering on $\mathcal{P}(U)$ is $\sqsubseteq_U$. It should be noted that $J_\#$ and $J_\bullet$ do not take into account the ordering $\prec_U$ appearing on $U_\#$ and $U_\bullet$ induced by U.

Variable Assignment

A ***variable assignment*** $\mathcal{V}$, is a mapping from a set of variables V to the domain U. Its extension to id-terms is as follows:

$\mathcal{V}(d) = g_O(d)$ if $d \in O$ and,

recursively, $\mathcal{V}(f(\ldots,\ T,\ \ldots)) = g_F(f)(\ldots,\ \mathcal{V}(T),\ \ldots)$.

Its further extension to F-terms is as follows :

$\mathcal{V}(P : Q[\ldots]) = \mathcal{V}(Q)$.

The Meaning of F-Formulae

Recall that in F-logic every F-term is an atomic formula. The *meaning* of an F-term T under interpretation I (which is a tuple $\langle U, g_O, g_F, J_\#, J_\bullet \rangle$) and variable assignments $\mathcal{V}$, denoted $M_{I,\mathcal{V}}(T)$, is a statement about the existence (*true*) or nonexistence (*false*) in I of an object $\mathcal{V}(T)$ with properties specified in T. The following is the formal definition of the *meaning function* $M_{I,\mathcal{V}}(T)$. For an F-term T given by

$$P : Q[\ldots,\ Flab_i \rightarrow R_i,\ \ldots, Slab_j \rightarrow \{S_1,\ \ldots, S_m\}, \ldots]$$

$M_{I,\mathcal{V}}(T) = true$ if and only if the following conditions hold:

1. $\mathcal{V}(P) \preceq_U \mathcal{V}(Q)$

   Intuitively, object $\mathcal{V}(Q)$ must be in class $\mathcal{V}(P)$ of the possible world (interpretation) I.

2. For each id-term $Flab_i$ (a functional label):

   - $\mathcal{V}(Flab_i) \in U_\#$

Intuitively, an id-term representing a label must be appropriately typed.

- $\mathcal{V}(R_i) \preceq_U J_{\#}(\mathcal{V}(Flab_i))(\mathcal{V}(Q))$

  Intuitively, the possible world I, with respect to $\mathcal{V}$, must have information about the object denoted by $\mathcal{V}(T)$ regarding label $Flab_i$, i.e., $J_{\#}(\mathcal{V}(Flab_i))(\mathcal{V}(Q))$, at least as much as the amount of information asserted by T (i.e., $\mathcal{V}(R_i)$). Recall that the information content of an object b is at least as much as that of an object a if $a \preceq_U b$.

- $M_{I,\mathcal{V}}(R_i) = true$

  Intuitively, the property of Q asserted by T (i.e., R_i) must also be true in I with respect to $\mathcal{V}$.

3. For each id-term $Slab_j$ (a set-valued label):

   - $\mathcal{V}(Slab_i) \in U_\bullet$

     Intuitively, an id-term representing a label must be appropriately typed.

   - $\{\mathcal{V}(S_1), \ldots, \mathcal{V}(S_m)\} \sqsubseteq_U J_\bullet(\mathcal{V}(Slab_i))(\mathcal{V}(Q))$

     Intuitively, the possible world I with respect to $\mathcal{V}$, must have information about the object denoted by $\mathcal{V}(T)$, i.e., its $J_\bullet(\mathcal{V}(Slab_i))(\mathcal{V}(Q))$, at least as much as the amount of information asserted by T ,i.e.,, $\{\mathcal{V}(S_1), \ldots, \mathcal{V}(S_m)\}$

   - $M_{I,\mathcal{V}}(S_k) = true$ for $k = 1, \ldots, m$

     Intuitively, the property of Q asserted by T (i.e., S_k) must also be true in I with respect to $\mathcal{V}$.

If the conditions above are not fulfilled then $M_{I,\mathcal{V}}(T) = false$.

For a formula, ϕ, that consists of logical connectives and/or quantifications, its meaning $(M_{I,\mathcal{V}}(\phi))$ is defined in the usual way. An interpretation I is called a *model* of ϕ if $M_{I,\mathcal{V}}(\phi) = true$, briefly denoted as $I \models_{\mathcal{V}} \phi$. Let S be a set of formulae. Then an interpretation I, with respect to a variable assignment $\mathcal{V}$, is called a *model* of S if for every $\phi \in S$, $I \models_{\mathcal{V}} \phi$.

2.4.4 F-Logic Databases

A database is defined as a finite set of formulae. There are two parts in a database, the ***extensional part*** (i.e., the set of F-terms (atomic formulae), this part is also called the set of facts) and the ***intensional part*** (i.e., the set of formulae "more complex" than F-terms). ***Logical implication*** for a set of formulae S is defined as usual:

$S \models \phi$ (the formula ϕ is logically implied by S) iff ϕ is true in every model of S.

Substitution is defined as follows: given a language L with a set of variables V and a set of basic objects O, a substitution σ is a mapping $V \longrightarrow$ {id-terms of L}. The range, i.e., {id-terms of L}, consists of object identities (id-terms) outside the domain of σ (which is some finite set $dom(\sigma) \subseteq V$). This substitution is extended to id-terms, where σ commutes with object constructors, and then extended to F-terms as follows:

- $\sigma(f(\ldots,\ T,\ \ldots)) = f(\ldots,\ \sigma(T), \ldots)$
- $\sigma(P : Q[\ldots,\ Flab \rightarrow R,\ \ldots, Slab \rightarrow \{\ldots,\ S, \ldots\}]) =$
  $\sigma(P) : \sigma(Q)[\ldots,\ \sigma(Flab) \rightarrow \sigma(R),\ \ldots, \sigma(Slab) \rightarrow \{\ldots,\ \sigma(S),\ \ldots\}]$

Substitution σ is further extended to formulae by letting it commute with logical connectives. A ***ground substitution*** is the case when $\sigma(X) \in O^*$ for each $X \in dom(\sigma)$. An instance of a formula ϕ is $\sigma(\phi)$ where σ is a given substitution, and $\sigma(\phi)$ is called a ***ground instance*** if there is no variable.

2.4.5 F-Logic Queries

A ***query*** statement is defined as an expression $Q?$, where Q is an F-term. The ***set of its answers with respect to a database*** D is the smallest set of ground F-terms that is

1. closed under $\models$ and
2. contains all ground instances of Q logically implied by D.

Examples of queries are given in the next subsection.

2.4.6 Examples of F-Logic Databases

Examples given in Figure 2.2 and Figure 2.3, taken from [KifLa89], show significant features of F-logic.

Figure 2.2 depicts a part of an IS-A (subclass) hierarchy (or, more precisely, an IS-A lattice [Kim90]). Classes and individual objects, which are from the same domain, are organized in a lattice. In F-logic, the lattice is ordered according to the "definition" ordering of denotational semantics, or according to the "knowledge content". For example, class ***assistant*** has more knowledge content than class ***empl*** because every individual assistant is an employee but not vice versa. Thus, statement ***assistant : jc*** [illegible] informative than ***empl : john***. We can see in Figure 2.2 that an instance is always located "above" (with respect to the ordering $\prec_O$) its class (or classes). For example, Figure 2.2 asserts that ***assistant, phil*** and ***faculty*** are ***employees***, ***"CS"***, ***"EE"***, ***"Mary"*** and ***"Bob"*** are ***string***, etc.

Figure 2.3 shows some facts about ***faculty, assistant, student***, etc. Clause (1) asserts that object ***bob*** (read as "an object denoted by object identity ***bob***") has name 'Bob", whose ***age*** is 40, works in ***dept*** : cs_1, etc. In clause (2), the attribute (la-

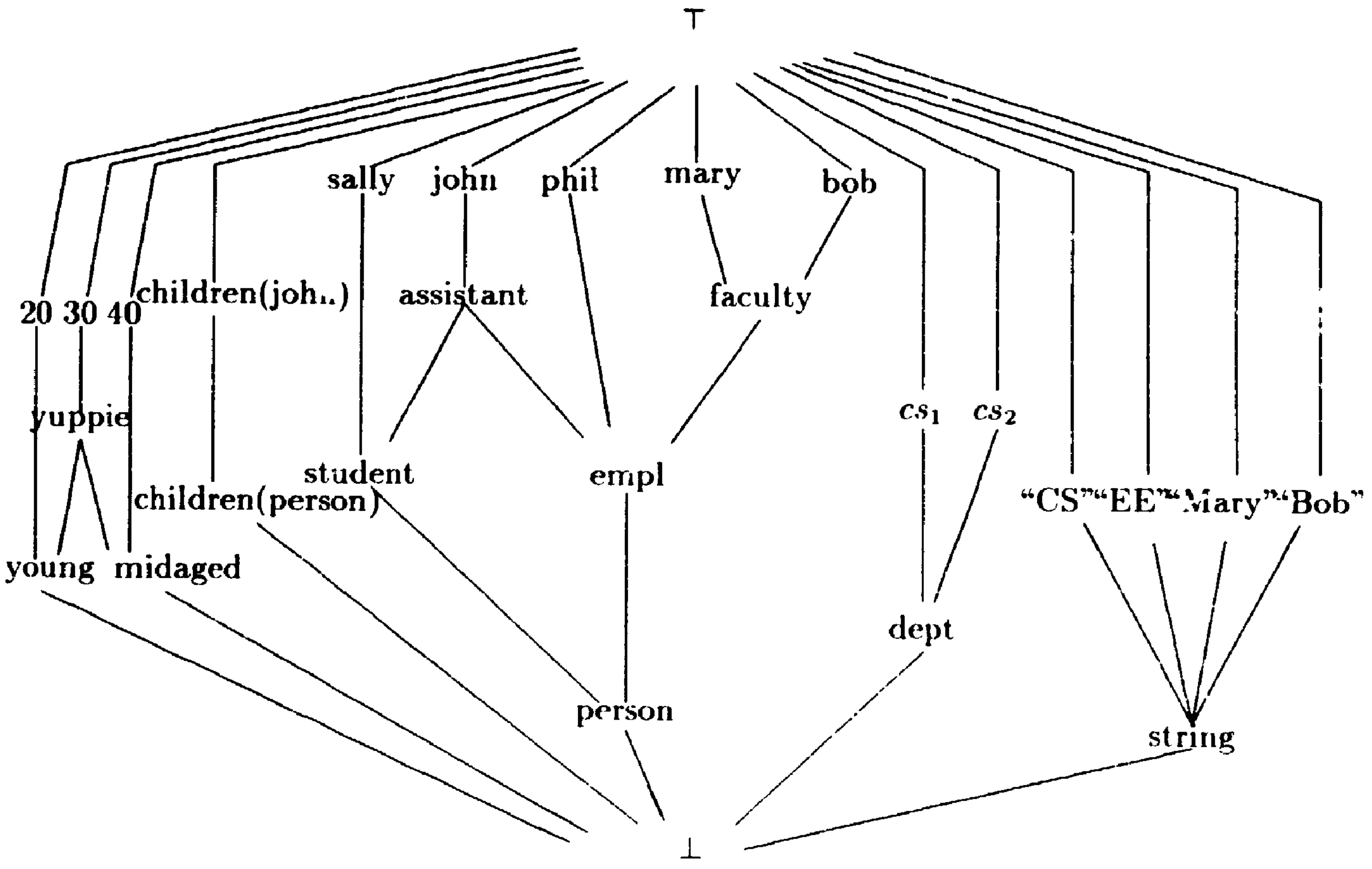


Figure 2.2: Part of the IS-A lattice (from[KifLa89])

bel) *friends* is set valued. This is achieved by using the set constructor "{}", e.g., *friends* $\rightarrow$ {*bob*, *sally*}.

General information about classes is given in clauses (5), (6) and (7). For example, clause (5) says that *faculty* is supervised by a *faculty* and is *midaged*.

Examples of rules are given by clauses (8) and (9). Clause (9) is a rule that defines a *method*. It says that for any person X, the method *children* is a function, which if given an argument Y, will return a set containing all common children of X and Y. In F-logic, the same id-term may denote different things. For example, in clause (9), id-term *children*(Y) appears twice, in the head and in the body. Id-term *children*(Y) that appears in the head is an attribute, while its appearance in the body serves as

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 2.3: A Sample Database (from [KifLa89])

an identity of an object containing the children of Y.

Queries in clause (10) and clause (11) show the application of method *children*. Clause (10) asks the *father* of *mary*'s child *sally*, while clause (11) asks the children of *mary* with *phil*. Clause (12) illustrates a query that has contradicting answers. This query asks information about all middle aged employees working for the *"CS"* department. Note that the attribute *supervisor* is single valued. The answers of these queries are:

$(A1)$ $bob[supervisor \rightarrow \top, age \rightarrow 40, works \rightarrow cs_1]$, and

$(A2)$ $mary[supervisor \rightarrow faculty, age \rightarrow 30, works \rightarrow cs_2]$.

In answer (A1), *bob*'s supervisor is $\top$. This is because based on rule (8), *bob*'s supervisor is *phil*, and based on the general description (5) *bob*'s supervisor is a *faculty*, but *phil* is not an instance of *faculty*, an inconsistency! An inconsistency does not always end up in $\top$, but it depends on the least upper bound of inconsistent values. If the restriction $midaged : Z$ in query (12) is deleted, then the following clause will also be retrieved:

$(A3)$ $john[supervisor \rightarrow phil, age \rightarrow young, works \rightarrow cs_1]$

where *john*'s supervisor, *phil*, is inferred from rule (8) and clause (1). Unlike object *bob*, object *john* is not a faculty so that the restriction due to clause (5) is not imposed. Thus, there is no contradiction.

Figure 2.2 and Figure 2.3 also demonstrate that users can think the objects of the lattice as sets ordered by a subset relation, and they can use higher-order intuitions to develop programs and models. However, the underlying semantics is still first order. This first-order semantics prevents difficulties associated with a real higher-order logic.

The first-order semantics of sets is achieved through *typing* on labels, so that single-valued and set-valued functions do not mix.

2.4.7 F-Logic Inheritance

In F-logic, inheritance (monotonic) is built into the semantics.

Theorem [KifLa89] *Let D be a database, $T = p[lab_1 \rightarrow q, lab_2 \twoheadrightarrow \{r\}]$ be a ground F-term, and $D \models T$. Assume $p' \in O^*$ is an id-term such that $p \preceq_O p'$, i.e., p' is an instance of class p. Then*

$$D \models p'[lab_1 \rightarrow q, lab_2 \twoheadrightarrow \{r\}]$$

Intuitively, this theorem says that if $p \preceq_O p'$, then properties of p also hold for p'; or, we can say that p' inherits properties of p. In the previous example, we can see that because $faculty \preceq_O mary$, $mary$ inherits $supervisor \rightarrow faculty$ from clause (5).

Proof: The proof follows directly from the definitions of interpretation structure and of the meaning of formulae. Let S denote $p'[lab_1 \rightarrow q, lab_2 \twoheadrightarrow \{r\}]$, and $I = \langle U, g_O, g_F, J_\#, J_\bullet \rangle$ be a model of D. Then we conclude $I \models T$ because $D \models T$. F-terms T and S are ground, so that they do not depend on $\mathcal{V}$. Hence, for every variable assignment $\mathcal{V}$, it follows from the meaning of an F-term described in the semantic section (section 2.3.3) that we have for F-term T:

1. $\mathcal{V}(lab_1) \in U_\#$ (the id-term representing the label is appropriately typed)

2. $\mathcal{V}(lab_2) \in U_\bullet$ (the id-term representing the label is appropriately typed)

3. $\mathcal{V}(q) \preceq_U J_\#(\mathcal{V}(lab_1))(\mathcal{V}(p))$

4. $\{\mathcal{V}(r)\} \sqsubseteq J_\bullet(\mathcal{V}(lab_2))(\mathcal{V}(p))$

5. $I \models q \wedge r$

Then, since $\mathcal{V}(p) \preceq_U \mathcal{V}(p')$, by the monotonicity of $J_{\#}$ and $J_{\bullet}$, we have for F-term S:

1. $\mathcal{V}(q) \preceq_U J_{\#}(\mathcal{V}(lab_1))(\mathcal{V}(p'))$, because $J_{\#}(\mathcal{V}(lab_1))(\mathcal{V}(p)) \preceq_U J_{\#}(\mathcal{V}(lab_1))(\mathcal{V}(p'))$ and $\mathcal{V}(q) \preceq_U J_{\#}(\mathcal{V}(lab_1))(\mathcal{V}(p))$.

2. $\{\mathcal{V}(r)\} \sqsubseteq_U J_{\bullet}(\mathcal{V}(lab_2))(\mathcal{V}(p'))$, for a similar reason with above, only this time for $J_{\bullet}$.

3. $I \models q \wedge r$.

Now, $I \models S$ follows from the definition of the truth of an F-term in the semantic section. ∎

The following is an analysis of a more complex case about *sally* (clause (4)) from the previous example. Because *student* $\preceq_O$ *sally*, *sally* inherits *age* $\rightarrow$ *young* from clause (6). However, because clause (4) asserts that sally is *midaged*; in each interpretation where both

$$sally[age \rightarrow young] \text{ and } sally[age \rightarrow midaged]$$

are true, it is necessarily the case that

$$sally[age \rightarrow lub(young, midaged)] = sally[age \rightarrow yuppie]$$

is also true. That is clause (4) and clause (6) logically entail *sally*[*age* $\rightarrow$ *yuppie*]. In every interpretation I the label *age*, which is interpreted as a monotonic single-valued function $J_{\#}(age)$, has to map $g_O(sally)$ into something which is an upper bound of both $g_O(young)$ and $g_O(midaged)$. Because $g_O : O^* \rightarrow U$ is a lattice homomorphism, we have $lub(g_O(young), g_O(midaged)) = g_O(lub(young, midaged)) = g_O(yuppie)$. Thus, for the chosen I, we might derive *sally*[*age* $\rightarrow$ *yuppie*]. Here we see that although the inherited property *age* $\rightarrow$ *young* is still true, in fact we have also: *age* $\rightarrow$ *yuppie*. This effect is called *monotonic overwriting* of inheritance [KifLa89].

The monotonic overwriting property of F-logic inheritance might not satisfy all real world situations. For instance, in the previous example, we might want *sally* to be still *midaged* (based on clause (4)) although based on clause (6) *sally* inherits *young*. This is a disadvantage of the monotonic overwriting property that resolves inconsistent values by their *lub*. We could solve the problem by making a method's value defined on a subclass overwrite its inherited value, but this property, which can be categorized as non-monotonic, is likely to cause problems in the logic.

2.4.8 F-Logic Methods

Methods are considered as the *procedural* aspect of OO paradigms. This is why some researchers argue that methods cannot be expressed in declarative ways [KifLa89]. For example, a formal data model to support a procedural OO language is proposed in [LecRi88, LecRV88],

Kifer and Lausen [KifLa89] argue that impedance mismatch between programs and data should be overcome in a declarative fashion, which requires methods to be defined declaratively. Thus the procedural component should be integrated in a declarative framework in a clean way.

F-logic allows declarative definition of methods because non-ground id-terms are allowed to appear in the label positions of F-terms. An example was given in Figure 2.3 (clause (9)) . The following is another example (from [KifLa89]). This example shows F-logic's ability to inherit methods and build them incrementally. Suppose that $person \preceq_O male$, $person \preceq_O female$, and $person \preceq_O writer$. Although normally the legal name is a person's last name, a maiden name of a married female and a pen name of a writer are also considered to be legal names. The method *legal-names* at any given year (represented by variable Y) is firstly defined for each person as follows:

$$X[\textit{legal-names}(Y) \rightarrow \{N\}] \Leftarrow \textit{year} : Y,\ \textit{person} : X[\textit{last-name}(Y) \rightarrow \textit{string} : N]$$

and then the definitions for married females and writers:

$$X[\textit{legal-names}(Y) \rightarrow \{N\}] \Leftarrow \textit{year} : Y,\ \textit{female} : X[\textit{maiden-name}(Y) \rightarrow \textit{string} : N]$$
$$X[\textit{legal-names}(Y) \rightarrow \{N\}] \Leftarrow \textit{year} : Y,\ \textit{writer} : X[\textit{pen-name}(Y) \rightarrow \textit{string} : N]$$

As an illustration, if *Joe* is a male and not a writer. this method will return only one legal name. On the other hand, if in year 1991 *mary* was a married female, a writer, and uses her husband's last name, she will have three different legal names in that year.

This example also demonstrates the capability of *operator overloading*, which means that the same method name can represent different procedures depending on the class where this name is used.

Essentially, methods in F-logic are "labels with parameters." Thus, plain labels can be viewed as non-parametric methods. It is pointed out by Kifer and Lausen [KifLa89] that this property is a pleasing one and corresponds to situations in abstract data types. We also consider that this is an elegant solution to the problems of expressing variety of methods in a object-oriented logic programming.

2.4.9 Discussion of F-Logic

F-logic has all basic properties required to be called object oriented (based on [Kim 90]). However, the treatment of the same object as a class and an instance object may cause problems. A simple example follows. Suppose we have data *student* : *joe* which means that an object denoted by *joe* is an instance of a class denoted by *student*. F-logic allows us to write something like *joe* : *mary* which asserts that *mary* is an instance of *joe*. This example illustrates how an object which we intended to be an individual object can have an instance object. In other words, F-logic cannot distinguish between class objects and instance objects, and we cannot express an object that is supposed to represent itself (i.e., as an individual object only).

The main differences between F-logic and its predecessors, i.e., O-logic, RO-logic and C-Logic, are:

- In F-logic, objects and labels are put in the same category, they all are represented by id-terms. The reason is to be able to treat them in the same framework.
- In F-logic, instance objects are also class objects, the reason for this is to make the inheritance property natural. It is worth noting that F-logic follows the way frame languages treat a class as an instance of its superclass rather than as its subclass.
- F-logic incorporates some inheritance concept which is not available at all in O-logic, RO-logic or C-logic.

2.4.10 Concluding Remarks on F-Logic

F-logic has all the core concepts of object-oriented databases (based on [Kim90]), i.e., object identity, attribute, method, class, class hierarchy, and inheritance. One concept that seems counter intuitive is that the same object can play a role as a class or as an individual object depending on its syntactic position (occurrence) in an F-term. In consequence, we cannot express an object that represents only itself (i.e., an individual object).

2.5 Comparison Study and Weaknesses to be Overcome

Table 2.1 shows the comparison among logics that we have discussed.

Table 2.1: Comparison Table I

Language	Complex Object	Typing	Object identity	Inheritance	Method	Encapsulation
O-logic [Maier86]	no sets	yes	yes	no	no	no
C-logic [CheWa89]	sets only	yes	yes	no	no	no
RO-logic [KifWu89]	yes	yes	yes	no	no	no
F-logic [KifLa89], [KifLaWu90]	yes	yes	yes	yes	yes	no

Table 2.2: Comparison Table II

Language	Deduction	Soundness	Completeness
O-logic [Maier86]	not well defined	yes	?
C-logic [CheWa89]	yes	yes	yes
RO-logic [KifWu89]	yes	yes	yes
F-logic [KifLa89], [KifLaWu90]	yes	yes	yes

Based on the discussed logic systems, we try to extract some reasonable properties that an OO logic should have. The following is the list of the properties that an "ideal" logic for OO databases should have:

- well-defined formal semantics;
- a clean declarative fashion for most OO concepts, such as object identity, complex objects, inheritance, methods;
- a well defined deduction;
- a sound and complete proof procedure;
- a uniform language for queries, updates, defining virtual data, and defining constraints;
- an ability to reason about inheritance and database schema. Usually, for reasoning about these, we need a higher order logic. However, as we know, a higher-order logic would be impractical if it is applied to this problem. One possible solution is what is done in F-logic, a logic that has an appearance of higher-order logic but has a natural first order semantics.
- an ability to support encapsulation;
- an ability to allow browsing schema and data using the same declarative formalism, such as the one demonstrated in F-logic;
- an ability to define complex objects in "natural" ways
- an ability to allow a user to program in a procedural way at some points, if necessary;
- an ability to reason about inconsistent data, such as what has been demonstrated by RO-logic;

In designing our logic for OO databases, we tried to accommodate the ideal properties above as much as we could.

There are other researches related to the the integration of deductive and object-oriented databases that we do not use as the basis in designing our logic. These include [AbiKan89], [HulYos90], and [LouOzs91].

Besides the direct problem of designing a better logic for OO databases, which is to make the logic fulfill all OO properties required but still be practical, there are still some problems awaiting associated with the heterogeneous structure of complex objects. Examples are: how to store complex objects, how to cluster the components of complex objects together, how to store shared information, and how to reason efficiently about complex objects. These are open problems for future research.

CHAPTER 3

Object-Oriented Logic

3.1 Introduction

Based on a comparison study of existing logics for object-oriented databases in the previous chapter, we found that F-logic ([KifLa89], [KifLaWu90]) has most object-oriented (OO) concepts, i.e., object identity, attribute, method, class, class hierarchy (or class lattice), and inheritance.

We have tried different approaches from that of F-logic in order to design a logic that has all the core concepts of OO databases. What is presented here is an improvement to F-logic's characteristic regarding class objects and instance (individual) objects. In F-logic, a class object is also an instance object, depending on where it is used. For example, $c : p$ asserts that p is an instance of class c. However, in another place of an F-logic program we may find $p : r$ which asserts r as an instance of class p. So, in the first sentence, p is an instance object, while in the second sentence p is a class object. We consider this counter-intuitive. The concept of a class object that is also an instance object may well cause confusion. A simple example follows. Suppose we have the datum *student* : *joe* which means that an object denoted by *joe* is an instance of a class object denoted by *student*. Assume that we intend to define object *joe* as an object that represents itself (as an individual object). F-logic allows us to write something like *joe* : *mary* which asserts that *mary* is an instance of *joe*. This example illustrates how an object which is supposed to be an individual object can have an instance. In other words, F-logic has problems in distinguishing class objects and instance objects.

In general, the argument above is for situations similar to the following. Suppose that a system designer of an F-logic program has defined an object intended to be only an instance (individual) object. Later, other users may misuse that object by treating it as a class. Unfortunately, it is not possible for the system designer to prevent such misuse in F-logic, because only *instance-class* objects (instance objects that are also class objects) exist. In our logic this kind of problem can be avoided. We changed the instance-class concept of F-logic. We do this by making a clear distinction syntactically and semantically between class objects and instance objects. An instance object can only be an instance object, it cannot be categorized or treated as a class as well. Conversely, a class object can only be a class object and it cannot be an instance (individual) object. Note that this distinction is in the same spirit as that of O-logic [Maier86] , $\mathcal{L}$O-logic [KifWu89] and C-logic [CheWa89]. Further advantages of our approach will be shown when we discuss *class methods* and *shared methods* in Chapter 4.

3.2 Design Considerations

In this section we will discuss the reasons behind the concepts that we have chosen. In order to clarify the ideas, we will give several simple examples.

3.2.1 Object Identity

We do not see any problem in implementing unique object identities on extensional databases, because this is just like non-deductive OO databases. However, some problems might be met in assigning object identities on intensional data as pointed out by Ullman [Ullm91]. He showed a problem on intensional data, where it is possible to create an infinite number of object identities (OIDs) for a certain type of object, while a non-object-oriented deductive system will create only a finite number of objects of the same type (see [Ullm91]). A similar problem is pointed out by Maier [Maier86] (see Section 2.1.9 on the discussion about applying a rule as an update). One solution to this problem is to follow the strategy used in C-logic [CheWa89], i.e., making the logic allow only *explicit* construction of object identities, in order to limit the number of objects created. We consider this solution to be better than that of [Grec92] which does not even provide OIDs to intensional data, with the reasoning that we can re-create those data by using available rules. We do not like the solution in [Grec92] because if we have two kinds of data, one with object identities and the other without object identities, then we will not be able to manipulate the two kinds of data in a uniform way.

We have chosen that our logic should allow *explicit construction* of object identities. This concept is useful to support Skolemization of existentially quantified object variables, so that there is no ambiguity in what objects determine the object to be created, i.e., the kind of ambiguity found in O-logic [Maier86] (see Section 2.1.10 Discussion of O-logic). In our logic, the user is only required to determine "which variables participate in determining the object being created," and one does not need to worry about supplying a unique identity since it can be generated by the system.

The following example about path rules illustrates the above idea. Note that, in this example, the path objects are determined by only the node objects at both ends (recall the discussion about path objects in Section 2.3.2.1).

1. $pathID\langle X,Y\rangle([instOf \rightarrow path], [src \rightarrow X], [dest \rightarrow Y])$
   $:- X([instOf \rightarrow node], [linkto \rightarrow Y])$
   /*Note that symbol :- is used as in Prolog*/
2. $pathID\langle X,Y\rangle([instOf \rightarrow path], [src \rightarrow X], [dest \rightarrow Y])$
   $:- X([instOf \rightarrow node], [linkto \rightarrow Z]),$
   $C([instOf \rightarrow path], [src \rightarrow Z], [dest \rightarrow Y])$

Here, *pathID*$\langle X,Y\rangle$ is an id-term similarly defined to the definition of an id-term in F-logic [KifLa89]. It represents an object identity that is constructed from function *pathID* and variables X and Y. The notation ":-" is defined as what is usually found in Prolog, Thus, an expression $\alpha : -\beta$ is equivalent to $\neg\beta \vee \alpha$. The identity is dependent on the values of X and Y. Although one might argue that an object identity should not depend on any value, this treatment makes sense here, because the nature of the objects from class *path* are actually depending on the nodes at both ends.

3.2.1.1 An Alternative Solution

The weakness of the above solution is that object identities become value dependent. The common interpretation of an object identity (OID) is that it should not depend on any value, otherwise the system would become value based. This is one major point why Ullman argues that OO concepts are not compatible with logic programming [Ullm91]. One possible solution that we are still investigating is to make an OID consist of two parts: a unique OID (could be generated by the system or supplied by the user), and a value based OID (functional OID) which is in the form similar to

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

pathID$\langle X, Y\rangle$ above. If we apply this idea to the same example as above, the entity creating rule would become:

1'. *PathID*$([instOf \rightarrow path],$ /* *PathID* represents a unique OID*/
$\quad [pathFuncID\ \langle X, Y\rangle],$ /* functional OID*/
$\quad [src \rightarrow X], [dest \rightarrow Y])$
$\quad$:- $X([instOf \rightarrow node], [linkto \rightarrow Y])$

2'. *PathID*$([instOf \rightarrow path],$ /* *PathID* represents a unique OID*/
$\quad [pathFuncID\langle X, Y\rangle],$ /* functional OID*/
$\quad [src \rightarrow X], [dest \rightarrow Y])$
$\quad$:- $X([instOf \rightarrow node], [linkto \rightarrow Z]),$
$\quad\quad C([instOf \rightarrow path], [pathFuncID\langle Z, Y\rangle], [src \rightarrow Z], [dest \rightarrow Y])$

We consider this approach superior to that of C-logic because it has an identity that is not dependent on any other value, yet it still has the information indicating the variables considered significant in determining the rule-created object. This functional OID would be useful when we design a procedure for determining whether two objects are actually identical. In this thesis we do not discuss this alternative solution further.

3.2.2 Class

A class denotes a set of objects that share a common set of attributes (also called properties) and a common set of methods (also called behaviour). Unlike F-logic [KifLa89, KifLaWu90], we distinguish the relation between a class and its subclasses and the relation between a class and its instances. We distinguish it syntactically and semantically. Hopefully, with this concept we will overcome the ambiguity encountered in F-logic where an object identity can represent an instance of a class (individual object) in one place, and the very same object identity could be in the "syntactic position" (the position of an object identity's occurrence in a sentence representing an instance-of relationship) of representing a class object in another place.

The following is an example of how we define classes. Here we use a similar example to the one used in [KifLaWu90] for easy comparison. This example shows how to define the classes of *person, empl, faculty* and *dept.* In this example, the symbol ⇒ (respectively, ⇒⇒) is used to indicate a typing declaration of a single-valued (respectively, set-valued) attribute.

```
1. person([isa → object],                   /*class definition of person*/
                                            /* person is a subclass of object*/
          [name ⇒ string],                  /*typing, single-valued attribute*/
          [friends⇒⇒ {person}],             /*typing, set valued attribute*/
                                            /*person: a constant denotes a class*/
          [children⇒⇒ person]               /*we may remove curly brackets if*/
          )                                 /* only one class is involved*/
2. empl ([isa → person],                    /*class definition of empl */
                                            /*empl is a subclass of person*/
         [works ⇒ dept])
3. faculty ([isa → empl],                   /* class definition of faculty*/
           [boss ⇒ {faculty, hi-paid-empl }], /*typing, single-valued attribute,*/
                                            /*multiple classes on range*/
           [age ⇒ midaged],
           [degrees⇒⇒ degrees]              /*set-valued attribute, this example
           )                                shows name-overloading capability,
                                            as an attribute and as a class name*/
4. dept ([isa → object],                    /*class definition of dept */
         [assistants⇒⇒ {student, empl}])
```

The following is another example adapted from [Page89] about antenna. Here we give an example of how generic methods "gain" and "capture-area" are defined in a class definition by using a rule.

```
1. antenna([isa → object],                                  /*class antenna defn.*/
          [wave ⇒ real], [capture-area ⇒ real],
          [gain(real, real) ⇒ real]                         /*signature for */
          )                                                 /*method gain */
2. par-antenna([isa → antenna],                             /*parabolic ant. defn */
             [efficiency ⇒ integer],                        /*typing, funct. attrib. */
             [diameter ⇒ integer],                          /*typing, funct. attrib.*/
             [capture-area(integer, integer) ⇒ real]        /*typing, 2-ary meth.*/
             )
3. Ant([gain(W, A) → div⟨mult⟨4, mult⟨3.14, A⟩⟩, mult⟨W, W⟩⟩]) :-
      Ant([instOf → antenna],                               /*deductive rule for */
          [wave → W],                                       /*generic meth. gain*/
          [capture-area → A]
          )
4. Par([capture-area(E,D) → div⟨mult⟨3.14, mult⟨D, mult⟨D, E⟩⟩⟩, 4⟩]) :-
      Par([instOf → par-antenna ],                          /*deduct. rule for gen.*/
          [efficiency → E],                                 /*meth. capture-area*/
          [diameter → D]
          )
```

The following is an example of an instance object's definition:

antennaID([*instOf* → *par-antenna*], [*diameter* → 5], [*efficiency* → 0.6])

In the above example, *gain*(W, A) and *capture-area*(E, D) are the generic methods for instances of the class ***antenna*** and the class ***par-antenna*** respectively.

3.3 The Syntax of Languages of Object-Oriented Logic (OOL)

The alphabet of an OOL language consists of:

- a set of logical symbols:
  - a countable set V of variables;
  - the usual logical connectives: $\wedge$ (and), $\vee$(or), $\neg$ (not), :- (implication), $\equiv$ (equivalence), $\perp$ (falsehood); quantifiers: $\forall$ (for all), $\exists$ (there exists);
  - a set of auxiliary symbols: "(", ")", "[", "]", "$\langle$", "$\rangle$", "{", "}", "$\rightarrow$", "$\Rightarrow$", "$\rightarrowtail$", "$\Rrightarrow$", "***isa***", "***instOf***";
- a set of non-logical symbols:
  - a countable set F of function symbols (possibly 0-ary).

In RO-logic [KifWu89] and F-logic [KifLa89, KifLaWu90], function symbols are called ***object constructors.*** Every function symbol has an arity $n \geq 0$. As in Predicate Calculus, a 0-ary function symbol corresponds to a constant symbol.

As in F-logic [KifLa89], the name "object" is used either for ***an individual object*** (which is called an "instance object") or a *class* of objects (which is called a "class object"). However, unlike the F-logic approach ([KifLa89], [KifLaWu90]), OOL treats classes c.ˆ individual objects as different entities (different categories) from instance (individual) objects. By treating instance objects and class objects as different categories, we eliminate a possible source of ambiguity in determining whether an object denoted by an OID is a *class* or an *instance* object. They are represented differently and have a clearly different semantics.

Definition An *id-term* is defined inductively as follows:

1. a variable is an id-term;
2. a 0-ary function symbol is an id-term;
3. if f is an n-ary function symbol and $t_1, \ldots, t_n$ are id-terms then $f\langle t_1, \ldots, t_n\rangle$ is an id-term.
4. There are no other id-terms. ■

Hence, an id-term is defined similarly to that of F-logic [KifLa89, KifLaWu90] and the usual terms found in Predicate Calculus. Notice that we use angle brackets "$\langle$" and "$\rangle$" to enclose arguments of a function, to prevent a confusion with round brackets used for enclosing a method's arguments. The following is an example of an id-term (which is taken from the previous example about path program): *pathID*$\langle X, Y\rangle$, where *pathID* is a function symbol and X, Y are variables.

A *method name* (or method denotation) is an id-term (the role of an id-term as a method name is indicated by its syntactic position). A 0-ary method (a method without any argument) is also called an *attribute* (or a *label*), to give it a more familiar terminology in OO concepts.

Definition A *ground id-term* is an id-term that contains no variables. ■

A ground id-term is regarded as a logical abstraction corresponding to the concept of object identity in object-oriented databases.

In OOL, *terms* are classified into *data* terms and *class* terms as defined below. We follow the notation conv tion in standard Prolog programs, i.e., names for ground id-terms start with lower-case letters and names for non-ground id-terms start with upper-case letters.

The following definitions are the result of modifying the F-logic term syntax [KifLaWu90] to suit our logic. The main differences between OOL syntax and F-logic syntax [KifLaWu90] are:

- a component of a data term cannot be a component of any class term, and vice versa;
- we include the special "*instOf*" attrib te (label) intended for asserting the instance-of relation between instance objects and their classes;
- "empty" data terms or class terms, such as $D()$, are not allowed.

Definition A *data term* has the following structure:

$$
\begin{aligned}
&D([instOf \rightarrow ClassOfD],\\
&\quad [FMeth_1(A_{1,1}, \ldots, A_{1,n_1}) \rightarrow V_1], \ldots, [FMeth_k(A_{k,1}, \ldots, A_{k,n_k}) \rightarrow V_k],\\
&\quad [SMeth_1(S_{1,1}, \ldots, S_{1,o_1}) \twoheadrightarrow \{SV_{1,1}, \ldots SV_{1,p_1}\}], \ldots,\\
&\quad [SMeth_l(S_{l,1}, \ldots, S_{l,o_l}) \twoheadrightarrow \{SV_{l,1}, \ldots, SV_{l,p_l}\}])
\end{aligned}
$$

where:

- D is an id-term.

  Intuitively, the symbol D represents an id-term denoting an instance object. The instance object denoted by D is also called *the context object*, i.e., the context object of each method specified inside the opening round bracket following D.

- *The instance-of-assertion component* [*instOf* $\rightarrow$ *ClassOfD*], where *instOf* is an auxiliary symbol, and *ClassOfD* is an id-term.

  Intuitively, this component is specially used for defining the relationship between an instance object and its class by using the special attribute "*instOf*". The symbol *ClassOfD* represents the id-term of the class to which the instance object denoted by D belongs.

- *The single-valued-method component* $[FMeth_i(A_{i,1}, \ldots, A_{i,n_i}) \rightarrow V_i]$, where

  - $1 \leq i \leq k$; $n_i \geq 0$.
  - $FMeth_i$ is an id-term.

    Intuitively, the symbol $FMeth_i$ represents an id-term denoting a single-valued (functional) method. As mentioned before, a method name is also an id-term. The syntactic position of $FMeth_i$ determines its role as a single-valued method, which is indicated by a single-headed arrow ("$\rightarrow$") following its arguments.
  - $A_{i,1}, \ldots, A_{i,n_i}$ are id-terms.

    Intuitively, the symbols $A_{i,1}, \ldots, A_{i,n_i}$ represent id-terms of instance objects as the proper arguments of $FMeth_i$.
  - V_i is either an id-term or a data term.

    Intuitively, the symbol V_i is a meta variable representing either

    * a data term as the output of method $FMeth_i$; or
    * the id-term of an instance object as the output of method $FMeth_i$.

- *The set-valued method component* $[SMeth_j(S_{j,1}, \ldots, S_{j,o_j}) \twoheadrightarrow \{SV_{j,1}, \ldots, SV_{j,p_j}\}]$, where:

  - $1 \leq j \leq l$; $o_j \geq 0$; $p_j \geq 0$.

- $SMeth_j$ is an id-term.

  Intuitively, the symbol $SMeth_j$ represents an id-term denoting a set-valued method. A set-valued method name is indicated by the double-headed arrow ("$\twoheadrightarrow$") following its arguments, and the curly brackets ("{" and "}") enclosing its output of type set.

- $S_{j,1}, \ldots, S_{j,o_j}$ are id-terms.

  Intuitively, the symbols $S_{j,1}, \ldots, S_{j,o_j}$ represent id-terms of instance objects as the proper arguments of $SMeth_j$.

- $SV_{j,1}, \ldots, SV_{j,p_j}$ are id-terms.

  Intuitively, the symbols $SV_{j,1}, \ldots, SV_{j,p_j}$ are meta variables representing either

  * data terms as the output of $SMeth_j$; or
  * the id-terms of instance objects as the output of $SMeth_j$.

The order of each component in a data term is immaterial, and the same method name may occur more than once. Furthermore, a data term does not need to have all of the different components described above, but at least one of them is required to indicate that it is a data term. When a method name does not have any proper argument, we may omit the empty round bracket () following it. For example, instead of writing $D([FMeth_i() \rightarrow V_i])$ we may write it as $D([FMeth_i \rightarrow V_i])$. ∎

Note that if $FMeth_i$ (or respectively $SMeth_j$) is a ground id-term then it represents only one method, but if it is a non-ground id-term then it represents a family of methods. A non-ground id-term representing a method name could be useful when we want to express a method that behaves differently according to the instantiation of the variable(s) in its id-term.

A data term is used to define the instance-of assertion and the methods that can be applied to an instance object. In other words, a data term is used to define the state and the behaviour of an instance object.

Definition A *class term* has the following structure:

$$
\begin{aligned}
&C([isa \rightarrow SuperClassOfC],\\
&\quad [FMeth_1(A_{1,1}, \ldots, A_{1,n_1}) \Rightarrow \{R_{1,1}, \ldots, R_{1,t_1}\}], \ldots,\\
&\quad [FMeth_k(A_{k,1}, \ldots, A_{k,n_k}) \Rightarrow \{R_{k,1}, \ldots, R_{k,t_k}\}],\\
&\quad [SMeth_1(S_{1,1}, \ldots, S_{1,o_1}) \Rrightarrow \{SR_{1,1}, \ldots SR_{1,u_1}\}], \ldots,\\
&\quad [SMeth_l(S_{l,1}, \ldots, S_{l,o_l}) \Rrightarrow \{SR_{l,1}, \ldots, SR_{l,u_l}\}])
\end{aligned}
$$

where:

- C is an id-term.

  Intuitively, the symbol C represents an id-term denoting a class object whose instance objects have methods with types given by the method signatures specified inside the round brackets.

- *The subclass-assertion component* $[isa \rightarrow SuperClassOfC]$, where *isa* is an auxiliary symbol, and *SuperClassOfC* is an id-term.

  Intuitively, this component is specially used for defining the relationship between the class C and its superclass by the special attribute "*isa*." The symbol *SuperClassOfC* represents the id-term of the class object to which the class object denoted by C belongs; and *SuperClassOfC* and C could be the same id-term.

- *The single-valued-method signature* $[FMeth_i(A_{i,1}, \ldots, A_{i,n_i}) \Rightarrow \{R_{i,1}, \ldots, R_{i,t_i}\}]$, where:

- $1 \leq i \leq k$; $n_i \geq 0$; $t_i \geq 0$.

- $FMeth_i$ is an id-term.

  Intuitively, the symbol $FMeth_i$ represents an id-term denoting a single-valued method. A single-valued method signature is indicated by "$\Rightarrow$" following its arguments.

- $A_{i,1}, \ldots, A_{i,n_i}$ are id-terms.

  Intuitively, the symbols $A_{i,1}, \ldots, A_{i,n_i}$ represent the id-terms of class objects as the proper argument of $FMeth_i$.

- $R_{i,1}, \ldots, R_{i,t_i}$ are id-terms.

  Intuitively, the symbols $R_{i,1}, \ldots, R_{i,t_i}$ represent the id-terms of class objects as the output of method $FMeth_i$.

• *The set-valued-method signature* $[SMeth_j(S_{j,1}, \ldots, S_{j,o_j}) \Rightarrow\!\!\!\Rightarrow \{SR_{j,1}, \ldots, SR_{j,u_j}\}]$, where

  - $1 \leq j \leq l$; $o_j \geq 0$; $u_j \geq 0$.

  - $SMeth_j$ is an id-term.

    Intuitively, the symbol $SMeth_j$ represents an id-term denoting a set-valued method. A set-valued method signature is indicated by "$\Rightarrow\!\!\!\Rightarrow$ " following its arguments.

  - $S_{j,1}, \ldots, S_{j,o_j}$ are id-terms.

    Intuitively, the symbols $S_{j,1}, \ldots, S_{j,o_j}$ represent the id-terms of class objects as the proper arguments of $SMeth_j$.

  - $SR_{j,1}, \ldots, SR_{j,u_j}$ are id-terms.

    Intuitively, the symbols $SR_{j,1}, \ldots, SR_{j,u_j}$ represent the id-terms of class objects as the output of $SMeth_j$.

As in the case for a data term, the order of each component in a class term is immaterial, and there is no restriction that the same method should not appear more than once. Furthermore, a class term does not need to have all of the different components described above. At least one component is required to indicate that it is a class term. When a method does not have any proper argument, we may omit the empty round brackets () following the method name. ■

Informally, the id-terms of class objects $R_{i,1}, \ldots, R_{i,t_i}$ represent the type of the output returned by the *FMeth*$_i$ when invoked in the context of an instance object from class C on arguments which are instances of classes $A_{i,1}, \ldots, A_{i,n_i}$. The id-terms of classes $SR_{j,1}, \ldots, SR_{j,u_j}$ represent the type of the outputs returned by the *SMeth*$_j$ when invoked in the context of an instance object from class C on arguments which are instances of classes $S_{j,1}, \ldots, S_{j,o_j}$.

The special attribute *isa* is used to define class hierarchies or the more general directed acyclic graphs of id-terms denoting class objects. The set of all id-terms denoting classes is partially ordered by subclass relationships.

Restriction The special attribute *isa* in class terms (or *instOf* in data terms) should be treated like a keyword in Pascal, in the sense that its name cannot be used for anything else. It is also required in OOL that an id-term that has been used to denote an instance object cannot be used to denote a class object as well, and vice versa.

Notice that in the previous definition of a data term, the output of methods can also be data terms. This allows us to embed data terms in other data terms to an arbitrary depth (*nested* data terms). On the other hand, the output of method signatures in class terms are already restricted to id-terms. The reason for this restriction is that we do not see many advantages of allowing the output to be class terms; without this restriction, we may face unnecessary extra complexity in the analysis of the logic and its presentation.

Definition A ***flat data term*** is a data term where the outputs of all methods, defined in it, are restricted to id-terms. In the previous definition of a data term, this means:

- V_i is an id-term;
- $SV_{j,1}, \ldots, SV_{j,p_j}$ are id-terms. ■

Definition A *term* is defined as follows:

1. Every data term is a term.
2. Every class term is a term.
3. There are no other terms. ■

Definition A ***flat term*** is defined as follows:

1. Every flat data term is a flat term.
2. Every class term is a flat term.
3. There are no other flat terms. ■

Definition Every term is a *molecular formula*, simply called a *molecule*. ■

Definition Every flat term is a *flat molecular formula*, simply called a *flat molecule*.■

A molecular formula is defined similarly as for F-logic [KifLaWu90]. The reason to name it a molecular formula is because a data term or a class term of the form *objId*$([Comp_1], \ldots, [Comp_n])$ is not really atomic and can be decomposed into, for example, *objId*$([Comp_1]) \wedge \ldots \wedge$ *objId*$([Comp_n])$.

Informally, a molecular formula that cannot be decomposed into a simpler one is called an *atomic formula*[1] or simply an *atom*.

[1]This notion of decomposing a formula into its atomic formulae will be defined formally later, in Section 3.6.

The following are examples of molecular formulae that are not atomic formulae:

$objId([instOf \rightarrow clsId],[fmeth \rightarrow val\text{-}id])$

$objId([smeth \twoheadrightarrow \{val\text{-}id_1, val\text{-}id_2\}])$

The following are examples of atomic formulae:

$objId([instOf \rightarrow clsId])$	$clsId([isa \rightarrow superclsId])$
$objId([fmeth \rightarrow val\text{-}id])$	$clsId([fmeth \Rightarrow \{clsId_1\}])$
$objId([smeth \twoheadrightarrow \{val\text{-}id_1\}])$	$clsId([smeth \Rrightarrow \{clsId_1\}])$
$objId([smeth \twoheadrightarrow \{\}])$	$clsId([smeth \Rrightarrow \{\}])$

Definition We define *formulae* recursively as follows:

1. Every molecular formula is a formula.
2. If A and B are formulae then so are $\neg A$ and $A\,op\,B$, where $op \in \{\wedge, \vee, :\!-, \equiv\}$.
3. If A and B are formulae then so are $\forall X\ A$, $\exists Y\ B$, where X and Y are variables.
4. An expression is a formula only if it can be shown to be a formula by the above three conditions. ∎

Definition Given an OOL alphabet, the OOL language over that alphabet is the set of all formulae constructed from the symbols of the alphabet. ∎

3.4 The Semantics of an OOL Language

The interpretation of an OOL language is the result of a modification of the interpretation of an F-logic language [KifLaWu90] to suit our model. While in F-logic the same object is both a class and an instance, in OOL we clearly distinguish between the two kinds of objects. An instance object can only be an instance, and a class object can only be a class.

The following notation will be used in presenting the semantic structure of OOL. Let $\mathcal{N}$ be the set of all natural numbers, and $\{S_n\}_{n\in\mathcal{N}}$ be a collection of sets, indexed by the natural numbers. Then $\prod_{n=0}^{\infty} S_n$ denotes their Cartesian product. Let U be a set. Then $Partial(U^{n+1},U)$ denotes the set of all partial functions $U^{n+1} \longrightarrow U$, and $\mathcal{P}(U)$ denotes the power-set of U.

Definition For a language $\mathcal{L}$ of OO-logic, its *semantic structure* (also called *interpretation*) I is a tuple $\langle U,\ I_{id},\ \prec,\ I_{inst},\ I_{\rightarrow},\ I_{\twoheadrightarrow},\ I_{\Rightarrow},\ I_{\Rrightarrow}\rangle$, where:

1. U : a non-empty set, called the domain of interpretation.

   Similar to F-logic, U is considered as the set of all "actual" objects in the "possible world" I.

2. I_{id} : a mapping that interprets every n-ary function symbol $f \in F$ by a total function $U^n \longrightarrow U$, where $n \geq 0$.

   Intuitively, when $n = 0$, $I_{id}(f)$ is equal to an element of U. Thus, function mbols are interpreted in the same way as in Predicate Calculus. Ground id-terms are considered as object identities (or object denotations) of "actual" objects in U. These object identities are interpreted by the "actual" objects in U through I_{id}.

3. $\prec$: a strict (non-reflexive) partial-order relation on U,

   Intuitively, this $\prec$ ordering is the semantic counterpart of the proper subclass assertion (through the special attribute *isa*) in the language, such that if $c,\ c' \in U$, then a statement $c \prec c'$ is read intuitively as "c is a proper subclass of c'."

The ordering $\preceq$ is defined as usual for either $\prec$ or $=$. Among the class objects in U, we have the partial-order relation $\preceq$ (i.e., reflexive, antisymmetric, and transitive). Notice that, unlike F-logic, the reflexive property is only for class objects, and the relation $\prec$ is only among class objects in U. This relation is extended to tuples over U as follows: Let tuples $\vec{a}, \vec{b} \in U^n$, where $\vec{a} = \langle a_1, \ldots, a_n \rangle$ and $\vec{b} = \langle b_1, \ldots, b_n \rangle$. Then $\vec{a} \preceq \vec{b}$ iff $a_i \preceq b_i$ for each $i = 1, \ldots, n$.

4. I_{inst}: the interpretation of the special attribute *instOf* as a binary relation,

$$I_{inst}(instOf) \in \mathcal{P}(U^2)$$

such that: Given $a, b, c \in U$,

$$\text{if } b \preceq c \text{ and } \langle a, b \rangle \in I_{inst}(instOf), \text{ we have } \langle a, c \rangle \in I_{inst}(instOf).$$

Intuitively, for each ordered pair element $I_{inst}(instOf)$, the first component is an instance object and the second component is the class object where it belongs. This ordered pair is the semantic counterpart of an instance-of assertion (through the special label "*instOf*") in the language.

By interpreting the special attribute *instOf* as a binary relation (which is similar to C-logic [CheWa89] that interprets attributes (labels) as binary relations), an instance object may belong to several classes and a class object may have more than one instance object.

5. $I_{\rightarrow} : U \longrightarrow \prod_{n=0}^{\infty} Partial(U^{n+1}, U)$

This mapping associates each element of U with a sequence of partial functions $U^{n+1} \longrightarrow U$.

This mapping is intended for the interpretation of each single-valued method. The interpretation is similar to that in F-logic [KifLaWu90].

Let μ be any ground id-term denoting a single-valued method. Then, $I_{\rightarrow}$ associates a sequence of partial functions $U^{n+1} \longrightarrow U$ to $I_{id}(\mu) \in U$. Each single-valued method may have different arities. For each single-valued method of arity $n \geq 0$, there is only one partial function $U^{n+1} \longrightarrow U$ in the sequence.

Let a method λ be an element of U. Then $I_{\rightarrow}(\lambda)$ represents a sequence of mappings. This sequence is indexed by the arity $n \geq 0$. Let the notation $I^n_{\rightarrow}(\lambda)$ refer to the n-th component of such a sequence, i.e., a mapping of arity $n + 1$. The actual arity of an n-ary method is $(n + 1)$ because the instance object from where the method is invoked (also called the context object), is included as the first argument. The remaining n arguments correspond to the proper arguments of the method.

In object-oriented programming terminology, the notation

$$I^n_{\rightarrow}(\lambda)(contxt, arg_1, \ldots, arg_n)$$

can be thought of as a message to the object *contxt* to invoke the single-valued method λ on arguments $arg_1, \ldots, arg_n$.

This interpretation, which associates a partial function to each arity of a method, allows us to overload method names, i.e., the same id-term (of a method) may have different numbers of arguments, where each arity has a different interpretation. In the next item, we will see that we can also overload a method name in another way, i.e., to overload a method name for both single-valued and set-valued methods.

6. $I_{\twoheadrightarrow} : U \longrightarrow \prod_{n=0}^{\infty} Partial(U^{n+1}, \mathcal{P}(U))$

This mapping associates each element of U with a sequence of partial functions $U^{n+1} \longrightarrow \mathcal{P}(U)$.

This mapping is intended for the interpretation of every set-valued method. The interpretation is defined similarly to that in F-logic [KifLaWu90].

Given $\lambda \in U$, $n \geq 0$, the explanation for $I_{\twoheadrightarrow}(\lambda)$ (respectively, $I^n_{\twoheadrightarrow}(\lambda)$) is similar to that for $I_{\rightarrow}(\lambda)$ (respectively, $I^n_{\rightarrow}(\lambda)$), with the difference that each n-ary set-valued method is interpreted by a partial set-valued function.

Before we present the next mapping, i.e., $I_{\Rightarrow}$, we need the following two definitions of upward closed sets and antimonotonic partial functions.

A set of classes $S \subseteq U$ is *upward closed* if for any class $c \in S$ and $c' \in U$ such that $c \prec c'$ (i.e., c is a proper subclass of c'), we have $c' \in S$. The notation $\mathcal{P}_{\uparrow}(U)$ will be used to denote the set of all *upward-closed* subsets of U.

Let φ be a partial function $U^{n+1} \longrightarrow \mathcal{P}_{\uparrow}(U)$, where $n \geq 0$. The partial function φ is *antimonotonic*[2] if and only if for every pair of $(n+1)$-tuples of classes $\langle contxtcl, \vec{clargs}\rangle$, $\langle subcontxtcl, \vec{clargs}\rangle \in U^{n+1}$ such that $subcontxtcl \prec contxtcl$ and $\varphi(contxtcl, \vec{clargs})$ is defined, we have

- $\varphi(subcontxtcl, \vec{clargs})$ is also defined, and
- the relation $\varphi(subcontxtcl, \vec{clargs}) \supseteq \varphi(contxtcl, \vec{clargs})$ holds.

The notation $Antimonotonic(U^{n+1}, \mathcal{P}_{\uparrow}U)$ will be used to denote the set of all partial functions $U^{n+1} \longrightarrow \mathcal{P}_{\uparrow}(U)$ that are antimonotonic.

7. $I_{\Rightarrow} : U \longrightarrow \prod_{n=0}^{\infty} Antimonotonic(U^{n+1}, \mathcal{P}_{\uparrow}U)$

This mapping associates each element of U with a sequence of antimonotonic partial functions $U^{n+1} \longrightarrow \mathcal{P}_{\uparrow}(U)$.

[2]We define the antimonotonicity property differently from that in F-logic [KifLaWu90] to avoid later difficulties in defining the well-typing conditions and in designing the well-typing inference rule. In [KifLaWu90] this property is defined as follows: if every pair of vectors of classes $\vec{a}, \vec{b} \in U^{n+1}$ such that $\vec{b} \prec \vec{a}$ and $\varphi(\vec{a})$ is defined, then $\varphi(\vec{b})$ is also defined and the relation $\varphi(\vec{b}) \supseteq \varphi(\vec{a})$ holds.

This mapping is intended to interpret every single-valued-method signature, i.e., to interpret the types of every single-valued method's argument(s) and its output.

Given $\lambda \in U$, $n \geq 0$, the explanation for $I^n_{\Rightarrow}(\lambda)$ is similar to that for $I^n_{\rightarrow}(\lambda)$.

8. $I_{\Rightarrow\!\!\!\Rightarrow} : U \longrightarrow \prod_{n=0}^{\infty} Antimonotonic(U^{n+1}, \mathcal{P}_{\uparrow}(U))$

   This mapping associates each element of U, with a sequence of antimonotonic partial functions $U^{n+1} \longrightarrow \mathcal{P}_{\uparrow}(U)$.

   This mapping is intended to capture semantically the type of every set-valued method's argument(s) and the type of its output(s).

The Well-Typing Conditions

The following well-typing conditions provide the link between $I_{\rightarrow}$ and $I_{\Rightarrow}$, and the link between $I_{\longrightarrow\!\!\!\!\rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$.

For all $m, obj, a_1, \ldots, a_n, cobj, ca_1, \ldots, ca_n \in U$:

1. For single-valued methods:

   if $I^n_{\rightarrow}(m)(obj, a_1, \ldots, a_n)$ and $I^n_{\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, and $\langle obj, cobj \rangle \in I_{inst}(instOf)$, we have:

   $$\langle a_i, ca_i \rangle \in I_{inst}(instOf) \text{ for each } i = 1, \ldots, n$$

   For set-valued methods:

   if $I^n_{\longrightarrow\!\!\!\!\rightarrow}(m)(obj, a_1, \ldots, a_n)$ and $I^n_{\Rightarrow\!\!\!\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, and $\langle obj, cobj \rangle \in I_{inst}(instOf)$, we have:

   $$\langle a_i, ca_i \rangle \in I_{inst}(instOf) \text{ for each } i = 1, \ldots, n$$

2. For single-valued methods:

   if for some $p \in U$, $p = I^n_{\rightarrow}(m)(obj, a_1, \ldots, a_n)$ and $I^n_{\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, and $\langle obj, cobj\rangle \in I_{inst}(instOf)$, then

   $$for\ every\ q \in I^n_{\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$$

   we have:

   $$\langle p, q\rangle \in I_{inst}(instOf)$$

   For set-valued methods:

   if for some $p \in U$, $p \in I^n_{\twoheadrightarrow}\ (m)(obj, a_1, \ldots, a_n)$ and $I^n_{\Rightarrow\!\!\!\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, and $\langle obj, cobj\rangle \in I_{inst}(instOf)$, then

   $$for\ every\ q \in I^n_{\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)\}$$

   we have:

   $$\langle p, q\rangle \in I_{inst}(instOf)$$

■

A discussion for $I_{\Rightarrow}$:

Let $I^n_{\Rightarrow}(\lambda)$ denote the n-th component of sequence $I_{\Rightarrow}(\lambda)$, where $\lambda \in U$ is an interpretation of a method name. Informally, the intended meaning of $I^n_{\Rightarrow}(\lambda)$ is the type of the corresponding $(n+1)$-ary function $I^n_{\rightarrow}(\lambda)$. Every element of the domain of $I^n_{\Rightarrow}(\lambda)$ (i.e., an $(n+1)$-tuple of classes $\langle contxtcl, clarg_1, \ldots, clarg_n\rangle$) is viewed as the type of any $(n+1)$-tuple of instance objects $\langle contxt, arg_1, \ldots, arg_n\rangle$, on which $I^n_{\rightarrow}(\lambda)$ can be applied correctly. This means that *contxt* is an instance of class *contxtcl*, and arg_i is an instance of class $clarg_i$ for each $i = 1, \ldots, n$.

Notice that the first argument of $I^n_{\Rightarrow}(\lambda)$ is the context class where the function is invoked.

On the other hand, $I^n_{\Rightarrow}(\lambda)$ $(contxtcl, clarg_1, \ldots, clarg_n)$ is intended to be the type of $I^n_{\rightarrow}(\lambda)$ $(contxt, arg_1, \ldots, arg_n)$ where *contxt* is an instance of class *contxtcl*, and arg_i is an instance of class $clarg_i$ for each $i = 1, \ldots, n$. This means that

$$I^n_{\rightarrow}(\lambda)(contxt, arg_1, \ldots, arg_n)$$

must belong to every class in $I^n_{\Rightarrow}(\lambda)(contxtcl, clarg_1, \ldots, clarg_n)$.

An intuitive explanation of the *upward-closedness* requirement is:
Assume that we have $I^n_{\rightarrow}(\lambda)(contxt, arg_1, \ldots, arg_n)$, as an instance of a class $c \in I^n_{\Rightarrow}(\lambda)$ $(contxtcl, clarg_1, \ldots, clarg_n)$, then it is natural that $I^n_{\rightarrow}(\lambda)$ $(contxt, arg_1, \ldots,$ $arg_n)$ must belong to every super class of c. For example, if *joe* is an instance of class *student* and *student* is a subclass of class *person* then naturally *joe* is also an instance of class *person*. This is why $I^n_{\Rightarrow}(\lambda)(contxtcl, clarg_1, \ldots, clarg_n)$, should be upward closed.

An intuitive explanation of the *antimonotonicity requirement* is:
Assume that $I^n_{\Rightarrow}(\lambda)$ is applicable to a tuple of classes $\langle contxtcl, clarg_1, \ldots, clarg_n\rangle$, by antimonotonicity, it should be also applicable to any tuple $\langle subcontxtcl, clarg_1,$ $\ldots, clarg_n\rangle$, where *subcontxtcl* is a subclass of *contxtcl*.

Let $I^n_{\rightarrow}(\lambda)$ be applicable to any tuple $\langle subcontxt, arg_1, \ldots, arg_n\rangle$ such that *subcontxt* is an instance of class *subcontxtcl* and arg_i is an instance of class $clarg_i$ for each $i = 1, \ldots, n$.

The value of $I^n_{\rightarrow}(\lambda)$ $(subcontxt, arg_1, \ldots, arg_n)$ should be compatible with both values of $I^n_{\Rightarrow}(\lambda)$ $(subcontxtcl, clarg_1, \ldots, clarg_n)$ and $I^n_{\Rightarrow}(\lambda)$ $(contxtcl,\ clarg_1,\ \ldots,\ clarg_n)$. Then, we have the following relation:

$$I^n_{\Rightarrow}(\lambda)(subcontxtcl, clarg_1, \ldots, clarg_n) \supseteq I^n_{\Rightarrow}(\lambda)(contxtcl, clarg_1, \ldots, clarg_n)$$

The reason for the superset relation is because $I^n_{\rightarrow}(\lambda)(subcontxt, arg_1, \ldots, arg_n)$ should belong to every class to which $I^n_{\rightarrow}(\lambda)(contxt, arg_1, \ldots, arg_n)$ belongs, where $contxt$ is an instance of $contxtcl$ and arg_i is an instance of $clarg_i$ for each $i = 1, \ldots, n$.

Given $\lambda \in U$, the explanation for $I^n_{\twoheadrightarrow}(\lambda)$ is similar to that of $I^n_{\rightarrow}(\lambda)$, except for obvious changes for set-valued methods.

3.4.1 Variable Assignment

A variable assignment $\mathcal{V}$ is a mapping from the set of variables V to the domain U. This variable assignment is extended to id-terms and terms (molecular formulae).

Definition For an id-term or a term t, $\mathcal{V}(t)$ is defined as follows:

1. $\mathcal{V}(t) = \mathcal{V}(Y)$ if $t = Y$ and $Y \in V$.
2. $\mathcal{V}(t) = \mathcal{V}(d) = I_{id}(d)$ if t is an id-term d of arity 0, where $d \in F$.
3. $\mathcal{V}(t) = \mathcal{V}(f(\ldots, T, \ldots)) = I_{id}(f)(\ldots, \mathcal{V}(T), \ldots)$, if $t = f(\ldots, T, \ldots)$ is an id-term of arity ≥ 1, where $f \in F$.
4. $\mathcal{V}(t) = \mathcal{V}(O_{id}([\ldots], \ldots, [\ldots]) = \mathcal{V}(O_{id})$, if t is a data term or a class term with id-term O_{ia}. ■

3.4.2 The Meaning of a Formula Under a Semantic Structure

Let I be a semantic structure and $\mathcal{V}$ be a variable assignment. A molecular formula T is *true* under the semantic structure I with respect to a variable assignment $\mathcal{V}$ (denoted by $I \models_{\mathcal{V}} T$) if and only if I has an object $\mathcal{V}(T)$ with properties as specified in T. This interpretation is defined formally below.

3.4.2.1 Satisfaction of Data Terms

Definition (An instance-of assertion)

Let T be a *flat data term* of the form

$$D([\mathit{instOf} \rightarrow \mathit{ClassOfD}]),$$

where D, $\mathit{ClassOfD}$ are id terms. Then

$$I \models_{\mathcal{V}} T \text{ iff } \langle \mathcal{V}(D), \mathcal{V}(\mathit{ClassOfD}) \rangle \in I_{inst}(\mathit{instOf})$$

Intuitively, $I \models_{\mathcal{V}} T$ iff $\mathcal{V}(D)$ is an instance of $\mathcal{V}(\mathit{ClassOfD})$ under the semantic structure I. ∎

Definition (A flat-single-valued-method assertion)

Let T be a *flat data term* of the form

$$D([\mathit{FMeth}(A_1, \ldots, A_n) \rightarrow V]),$$

where D, FMeth, $A_1, \ldots, A_n$, V are id-terms, and $n \geq 0$.

$I \models_{\mathcal{V}} T$ iff

- $I^n_{\rightarrow}(\mathcal{V}(\mathit{FMeth}))(\mathcal{V}(D), \mathcal{V}(A_1), \ldots, \mathcal{V}(A_n))$ is defined; and
- $\mathcal{V}(V) = I^n_{\rightarrow}(\mathcal{V}(\mathit{FMeth}))(\mathcal{V}(D), \mathcal{V}(A_1), \ldots, \mathcal{V}(A_n))$. ∎

Definition (A nested-single-valued-method assertion)

Let T be a *nested data term* of the form

$$D([\mathit{FMeth}(A_1, \ldots, A_n) \rightarrow \mathit{Data}]),$$

where D, FMeth, $A_1, \ldots, A_n$, are id-terms, $n \geq 0$, and Data is a meta variable representing a data term.

$I \models_{\mathcal{V}} T$ iff

- $I^n_{\rightarrow}(\mathcal{V}(FMeth))(\mathcal{V}(D), \mathcal{V}(A_1), \ldots, \mathcal{V}(A_n))$ is defined;
- $\mathcal{V}(Data) = I^n_{\rightarrow}(\mathcal{V}(FMeth))(\mathcal{V}(D), \mathcal{V}(A_1), \ldots, \mathcal{V}(A_n))$, and
- $I \models_{\mathcal{V}} Data$. ■

Definition (A flat-set-valued-method assertion)
Let T be a *flat data term* of the form

$$D([SMeth(S_1, \ldots, S_l) \twoheadrightarrow \{SV_1, \ldots SV_m\}]),$$

where D, $SMeth$, $S_1, \ldots, S_l$, $SV_1, \ldots, SV_m$ are id terms, $l \geq 0$, and $m \geq 0$.

$I \models_{\mathcal{V}} T$ iff

- $I^n_{\twoheadrightarrow}(\mathcal{V}(SMeth))(\mathcal{V}(D), \mathcal{V}(S_1), \ldots, \mathcal{V}(S_l))$ is defined; and
- $\{\mathcal{V}(SV_1), \ldots, \mathcal{V}(SV_m)\} \subseteq I^l_{\twoheadrightarrow}(\mathcal{V}(SMeth))(\mathcal{V}(D), \mathcal{V}(S_1), \ldots, \mathcal{V}(S_l))$. ■

Definition (A nested-set-valued-method assertion)
Let T be a *nested data term* of the form

$$D([SMeth(S_1, \ldots, S_l) \twoheadrightarrow \{Data_1, \ldots, Data_m\}]),$$

where D, $SMeth$, $S_1, \ldots, S_l$ are id-terms, $Data_1, \ldots, Data_m$ are meta variables representing data terms, $l \geq 0$, and $m \geq 0$.

$I \models_{\mathcal{V}} T$ iff

- $I^n_{\twoheadrightarrow}(\mathcal{V}(SMeth))(\mathcal{V}(D), \mathcal{V}(S_1), \ldots, \mathcal{V}(S_l))$ is defined;
- $\{\mathcal{V}(Data_1), \ldots, \mathcal{V}(Data_m)\} \subseteq I^l_{\twoheadrightarrow}(\mathcal{V}(SMeth))(\mathcal{V}(D), \mathcal{V}(S_1), \ldots, \mathcal{V}(S_l))$ and
- $I \models_{\mathcal{V}} Data_i$, for $i = 1, \ldots, m$. ■

Definition (A data term in general)

Let T be a ***data term*** of the form

$$D([Comp_1], \ldots, [Comp_n]),$$

where D is an id term, and each component $Comp_i$ (for $i = 1, \ldots, n$) is one of the following (where the symbols are defined as in the previous definitions of data terms):

- $instOf \rightarrow ClassOfD$, i.e., an instance-of-assertion component;
- $FMeth(A_1, \ldots, A_n) \rightarrow V$, i.e., a flat-single-valued-method component;
- $FMeth(A_1, \ldots, A_n) \rightarrow Data$, i.e., a nested-single-valued-method component;
- $SMeth(S_1, \ldots, S_l) \twoheadrightarrow \{SV_1, \ldots, SV_m\}$, i.e., a flat-set-valued-method component;
- $SMeth(S_1, \ldots, S_l) \twoheadrightarrow \{Data_1, \ldots, Data_m\}$, i.e., a nested-set-valued-method component.

Then $I \models_V T$ iff $I \models_V D([Comp_i])$ for each $i = 1, \ldots, n$. ∎

It follows immediately from the last definition that every data term of the form

$$D([Comp_1], \ldots, [Comp_n])$$

is logically equivalent to formula $D([Comp_1]) \wedge \ldots \wedge D([Comp_n])$, where the meaning of "$\wedge$" is as usual, i.e.,

$$I \models_V D([Comp_1], \ldots, [Comp_n]) \text{ iff } I \models_V D([Comp_1]) \wedge \ldots \wedge D([Comp_n]).$$

It also follows from the definition of the satisfaction of a nested data term that a nested data term such as:

$$D([FMeth_1(A_1, \ldots, A_n) \rightarrow V([FMeth_2(B_1, \ldots, B_k) \rightarrow C]))$$

is logically equivalent to

$$D([FMeth_1(A_1, \ldots, A_n) \rightarrow V]) \wedge V([FMeth_2(B_1, \ldots, B_k) \rightarrow C])$$

A similar equivalent relation is used for nested-set-valued methods.

3.4.2.2 Satisfaction of Class Terms

Definition (An isa assertion)

Let T be a *class term* of the form

$$C([isa \rightarrow SuperClassOfC]),$$

where C, $SuperClassOfC$ are id-terms.

$$I \models_{\mathcal{V}} T \text{ iff } \mathcal{V}(C) \preceq \mathcal{V}(SuperClassOfC)$$

Intuitively, $I \models_{\mathcal{V}} T$ iff $\mathcal{V}(C)$ is a subclass of $\mathcal{V}(SuperClassOfC)$ under the semantic structure I. ■

Definition (A single-valued method signature)

Let T be a *class term* of the form

$$C([FMeth(A_1, \ldots, A_n) \Rightarrow \{R_1, \ldots, R_l\}]),$$

where C, $FMeth$, $A_1, \ldots, A_n$, $R_1, \ldots, R_l$ are id-terms, $n \geq 0$, and $l \geq 0$.

$I \models_{\mathcal{V}} T$ iff

- $I^n_{\Rightarrow}(\mathcal{V}(FMeth))(\mathcal{V}(C), \mathcal{V}(A_1), \ldots, \mathcal{V}(A_n))$ is defined; and
- $\{\mathcal{V}(R_1), \ldots, \mathcal{V}(R_l)\} \subseteq I^n_{\Rightarrow}(\mathcal{V}(FMeth))(\mathcal{V}(C), \mathcal{V}(A_1), \ldots, \mathcal{V}(A_n))$. ■

Definition (A set-valued method signature)

Let T be a *class term* of the form

$$C([SMeth(S_1, \ldots, S_t) \Rrightarrow \{SR_1, \ldots, SR_u\}]),$$

where C $SMeth$, $S_1, \ldots, S_t$, $SR_1, \ldots, SR_u$ are id-terms, $t \geq 0$ and $u \geq 0$.

$I \models_{\mathcal{V}} T$ iff

- $I^t_{\Rrightarrow}(\mathcal{V}(SMeth))(\mathcal{V}(C), \mathcal{V}(S_1), \ldots, \mathcal{V}(S_t))$ is defined; and
- $\{\mathcal{V}(SR_1), \ldots, \mathcal{V}(SR_u)\} \subseteq I^t_{\Rrightarrow}(\mathcal{V}(SMeth))(\mathcal{V}(C), \mathcal{V}(S_1), \ldots, \mathcal{V}(S_t))$. ∎

Definition (A class term in general)

Let T be a *class term* of the form

$$C([Comp_1], \ldots, [Comp_n]),$$

where C is an id term, and each component $Comp_i$ (for $i = 1, \ldots, n$) is one of the following (where the symbols are defined as in the previous definitions of class terms):

- $isa \rightarrow SuperClassOfC$;
- $FMeth(A_1, \ldots, A_n) \Rightarrow \{R_1, \ldots, R_l\}$;
- $SMeth(S_1, \ldots, S_t) \Rrightarrow \{SR_1, \ldots, SR_u\}$.

Then $I \models_{\mathcal{V}} T$ iff $I \models_{\mathcal{V}} C([Comp_i])$ for each $i = 1, \ldots, n$. ∎

Following the definition above, each class term of the form $C([Comp_1], \ldots, [Comp_n])$ is logically equivalent to formula $C([Comp_1]) \wedge \ldots \wedge C([Comp_n])$, i.e.,

$$I \models_{\mathcal{V}} C([Comp_1], \ldots, [Comp_n]) \text{ iff } I \models_{\mathcal{V}} C([Comp_1]) \wedge \ldots \wedge C([Comp_n])$$

The meaning of formulae $\alpha \vee \beta$, $\alpha \wedge \beta$ and $\neg\alpha$ are all defined in the usual way. For quantifiers, the meaning is also ordinary as follows:

- $I \models_{\mathcal{V}} (\forall X\alpha)$ if for *every* $\mathcal{V}'$ that agrees with $\mathcal{V}$ except possibly on X, $I \models_{\mathcal{V}'} \alpha$;
- $I \models_{\mathcal{V}} (\exists X\alpha)$ if for *some* $\mathcal{V}'$ that agrees with $\mathcal{V}$ except possibly on X, $I \models_{\mathcal{V}'} \alpha$.

From the above definition of quantified formulae, we can conclude that if a formula α is closed (not containing any free variable) then its meaning is independent of variable assignments, and the notation that the structure I satisfies α can be denoted as $I \models \alpha$.

A structure I with respect to a variable assignment $\mathcal{V}$ is called a *model* of a formula α if $I \models_{\mathcal{V}} \alpha$. Let P be a set of formulae. Then, the structure I, with respect to a variable assignment $\mathcal{V}$, is called a *model* of P if for every $\alpha \in P$, $I \models_{\mathcal{V}} \alpha$. The set of formulae P logically implies (or semantically entails) a formula α (denoted by $P \models \alpha$) if and only if every model of P is also a model of α.

3.5 Incorporating Predicates

In certain cases where an application is more appropriately represented in a value-based setting, or when we need to mix predicates and objects, we may need to have first-order predicates as well as objects ([KifWu89, KifLaWu90, CheWa89]). For example, we may need to verify that certain objects stand in a particular symmetric relationship to each other, such as adjacency and equality. These kinds of relationships can be encoded as objects, but the resulting representations may not be natural. For example, such a case arises in representing equalities or inequalities among numbers. In the next subsection, we will show how predicates are encoded as terms and how to incorporate predicate-atomic formulae into OOL syntactically and semantically. This is the result of adapting the corresponding techniques used in [KifWu89, KifLaWu90].

3.5.1 Encoding Predicates as OOL Terms

To encode an n-ary predicate symbol p, we introduce a new class $\hat{p}$. Let $func_p$ be a new n-ary function symbol. Then, we assert

$$(\forall A_1 \ldots \forall A_n)\langle func_p\langle A_1, \ldots, A_n\rangle([instOf \rightarrow \hat{p}])\rangle$$

and represent an atomic formula of Predicate Calculus of the form

$$p(T_1, \ldots, T_n)$$

by means of an OOL term

$$func_p\langle T_1, \ldots, T_n\rangle([arg_1 \rightarrow T_1], \ldots, [arg_n \rightarrow T_n])$$

For example, suppose that we have the following atomic formula of Predicate Calculus:

$$(a \doteq b)$$

where $\doteq$ is an infix predicate symbol, and a, b are constants. Then, we represent this atomic formula by the following data terms:

$$f_{eq}\langle a, b\rangle([arg_1 \rightarrow a], [arg_2 \rightarrow b])$$

$$f_{eq}\langle a, b\rangle([instOf \rightarrow \widehat{eq}])$$

where f_{eq} is a binary function symbol, and a, b are the associated *id-terms.* We also introduce class $\widehat{eq}$ to simulate the predicate $\doteq$, and assert the *instOf* relationship.

Similar to F-logic [KifLaWu90], to avoid the need for converting every predicate into OOL terms such as given in the example above, we incorporate predicates into the semantics of OOL.

3.5.2 Syntax of The Logic Extended with Predicates

To include predicates in the object-oriented logic, we extend the alphabet of an OOL language with a set $\wp$ of predicate symbols.

Definition If $p \in \wp$ is an *n-ary predicate symbol* and $Id_1, \ldots, Id_n$ are *id-terms*, then $p(Id_1, \ldots, Id_n)$ is a *predicate-atomic formula* or simply called a *P-atom.* ■

Definition A *molecular formula* is either a term (in the sense of OOL) or a predicate-atomic formula. ■

Definition A *flat molecular formula* (simply called a *flat molecule*) is either a flat term or a predicate-atomic formula. ■

Definition Formulae are now defined to be constructed out of terms and predicate-atomic formulae plus the usual logical connectives and quantifiers. ■

Definition A *literal* is either a molecular formula (also called a positive literal) or a negated molecular formula (also called a negative literal). ■

3.5.3 Semantics of The Logic Extended with Predicates

The tuple of the original semantic structure (interpretation) I (which is defined in Section 3.4) is augmented with a new mapping for predicate symbols. Each n-ary predicate symbol is interpreted as an n-ary relation on the domain of interpretation U by using the function $I_\wp$, as follows:

$$I_\wp(p) \subseteq U^n, \text{ for each } n\text{-ary predicate symbol } p \in \wp$$

The Meaning of a Predicate-Atomic Formula Under a Semantic Structure Given a semantic structure $I = \langle U, I_{id}, \prec, I_{inst}, I_\wp, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow} \rangle$, let $\mathcal{V}$ be a variable assignment, p be a predicate symbol, and $Id_1, \ldots, Id_n$ be id-terms. Then

$$I \models_{\mathcal{V}} p(Id_1, \ldots, Id_n) \text{ iff } \langle \mathcal{V}(Id_1), \ldots, \mathcal{V}(Id_n)\rangle \in I_p(p)$$

The interpretation for the *equality predicate* ($\doteq$) is defined as follows:

$$I_p(\doteq) : \{\langle \alpha, \alpha\rangle | \alpha \in U\}$$

It follows immediately from the definition above that: if t_1 and t_2 are id-terms, we have $I \models_{\mathcal{V}} (t_1 \doteq t_2)$ iff $\mathcal{V}(t_1) = \mathcal{V}(t_2)$.

The satisfaction of a formula consisting of different kinds of molecular formulae under a semantic structure and the notion of a model are defined in the usual way.

3.6 Properties of Semantic Entailment

In this subsection, we shall describe properties of the semantic entailment ($\models$) of OOL. For presentation simplicity, the properties are presented through assertions about ground molecular formulae. All of the assertions follow from the definition of the semantic structure in Section 3.4 and 3.5. These lemmas shall be used later as the basis for the semantic-entailment closure of Herbrand interpretations, and for designing the inference rules of an OOL proof theory.

For all of the following lemmas, let G be a set of ground-molecular formulae, F^* a set of ground id-terms, I a semantic structure $\langle U, I_{id}, \prec, I_{inst}, I_p, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rrightarrow}\rangle$, and $\mathcal{V}$ any variable assignment.

The Equality Predicate ("$\doteq$")

Lemma 3.6.1 *(Reflexivity of "$\doteq$")*
For each $o \in F^*$, $I \models (o \doteq o)$.

Proof: From the definition $I_{\wp}(\doteq) : \{\langle\alpha, \alpha\rangle | \alpha \in U\}$ and the definition of $\mathcal{V}$, the proof follows immediately. ■

Lemma 3.6.2 *(Symmetry of "$\doteq$")*
Let $a, b \in F^*$. If $I \models (a \doteq b)$ then $I \models (b \doteq a)$.

Proof: The proof follows immediately from the definition of $I_{\wp}(\doteq)$ and the definition of $\mathcal{V}$. ■

Lemma 3.6.3 *(Transitivity of "$\doteq$")*
Let $a, b, c \in F^*$. If $I \models (a \doteq b)$ and $I \models (b \doteq c)$ then $I \models (a \doteq c)$.

Proof: The proof follows immediately from the definition of $I_{\wp}(\doteq)$ and the definition of $\mathcal{V}$. ■

Lemma 3.6.4 *(Substitution of "$\doteq$")*
Let $a, b \in F^*$ and L be a literal. If $I \models (a \doteq b) \wedge L$ and L' is the result of replacing an occurrence of a in L with b, then $I \models L'$.

Proof: The proof follows immediately from the definitions of the satisfaction of a literal, $I_{\wp}(\doteq)$, and $\mathcal{V}$. ■

The Isa Relation

Lemma 3.6.5 *(Reflexivity of The Isa Relation)*
Let $a, b_1, \ldots, b_n, c, fm, sm, p, q \in F^*$.

1. *If $I \models a([isa \rightarrow p])$ then $I \models a([isa \rightarrow a])$ and $I \models p([isa \rightarrow p])$.*

2. *If $I \models q([\text{instOf} \rightarrow a])$ then $I \models a([isa \rightarrow a])$.*

3. If $I \models a([fm(b_1, \ldots, b_n) \Rightarrow \{c\}]$, *then*
$I \models a([isa \rightarrow a])$,
$I \models b_1([isa \rightarrow b_1]), \ldots, I \models b_n([isa \rightarrow b_n])$, *and*
$I \models c([isa \rightarrow c])$.

4. If $I \models a([sm(b_1, \ldots, b_n) \Rightarrow\!\!\!\Rightarrow \{c\}]$, *then*
$I \models a([isa \rightarrow a])$,
$I \models b_1([isa \rightarrow b_1]), \ldots, I \models b_n([isa \rightarrow b_n])$, *and*
$I \models c([isa \rightarrow c])$.

Proof:

1. By the definition of the satisfaction of an *isa* assertion, $I \models a([isa \rightarrow p])$ iff $\mathcal{V}(a) \preceq \mathcal{V}(p)$. By the definition of $\preceq$, $\mathcal{V}(a)$ and $\mathcal{V}(p)$ are class objects. From the reflexivity of $\preceq$ on any class object, $\mathcal{V}(a) \preceq \mathcal{V}(a)$ and $\mathcal{V}(p) \preceq \mathcal{V}(p)$. By the definition of the satisfaction of an *isa* assertion, $I \models a([isa \rightarrow a])$ and $I \models p([isa \rightarrow p])$.

2. By the definition of I_{inst}, we have an ordered pair $\langle \mathcal{V}(q), \mathcal{V}(a) \rangle \in I_{inst}(instOf$, where $\mathcal{V}(a)$ is a class object. Since $\mathcal{V}(a)$ is a class object, by the definition of $\preceq$, we have $\mathcal{V}(a) \preceq \mathcal{V}(a)$. Thus, $I \models a([isa \rightarrow a])$.

3. By the definition of the satisfaction of a single-valued-method signature, we have

   - $I^n_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(a), \mathcal{V}(b_1), \ldots, \mathcal{V}(b_n))$ is defined, and
   - $\mathcal{V}(c) \in I^n_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(a), \mathcal{V}(b_1), \ldots, \mathcal{V}(b_n))$,

   where $\mathcal{V}(a)$ $\mathcal{V}(b_1), \ldots, \mathcal{V}(b_n)$, and $\mathcal{V}(c)$ are class objects. From the reflexivity of $\preceq$ on any class object, $\mathcal{V}(a) \preceq \mathcal{V}(a)$, $\mathcal{V}(b_1) \preceq \mathcal{V}(b_1)$, $\ldots, \mathcal{V}(b_n) \preceq \mathcal{V}(b_n)$, and $\mathcal{V}(c) \preceq \mathcal{V}(c)$. Thus $I \models a([isa \rightarrow a])$, $I \models b_1([isa \rightarrow b_1])$, $\ldots$, $I \models b_n([isa \rightarrow b_n])$, and $I \models c([isa \rightarrow c])$.

4. The proof is similar to proof (3), by using $I_{\Rightarrow\!\!\!\!\not\,\Rightarrow}$'s definition on $\mathcal{V}(sm)$. ■

Lemma 3.6.6 *(Transitivity of The Isa Relation)*
Let $a, b, c \in F^*$. If $I \models a([isa \rightarrow b])$ and $I \models b([isa \rightarrow c])$, then $I \models a([isa \rightarrow c])$.

Proof: Since $I \models a([isa \rightarrow b])$ and $I \models b([isa \rightarrow c])$, then $\mathcal{V}(a) \preceq \mathcal{V}(b)$ and $\mathcal{V}(b) \preceq \mathcal{V}(c)$. By the transitivity of $\preceq$, $\mathcal{V}(a) \preceq \mathcal{V}(c)$. By the definition of the satisfaction of an *isa* assertion, $I \models a([isa \rightarrow c])$. ■

Lemma 3.6.7 *(Antisymmetry of the Isa Relation)*
Let $a, b \in F^*$. If $I \models a([isa \rightarrow b])$ and $I \models b([isa \rightarrow a])$, then $I \models (a \doteq b)$.

Proof: Since $I \models a([isa \rightarrow b])$ and $I \models b([isa \rightarrow a])$, then $\mathcal{V}(a) \preceq \mathcal{V}(b)$ and $\mathcal{V}(b) \preceq \mathcal{V}(a)$. By the antisymmetry of $\preceq$, $\mathcal{V}(a) = \mathcal{V}(b)$. By the definition of $\doteq$, $I \models (a \doteq b)$. ■

The Instance-of Relation

Lemma 3.6.8 *(InstOf-Isa-Transitivity of The Instance-of Relation)*
Let $a, b, c \in F^*$. If $I \models a([\text{instOf} \rightarrow b])$ and $I \models b([isa \rightarrow c])$, then $I \models a([\text{instOf} \rightarrow c])$.

Proof: From $I \models a([instOf \rightarrow b])$, by the definition of I_{inst}, we have an ordered pair $\langle \mathcal{V}(a), \mathcal{V}(b) \rangle \in I_{inst}(instOf)$ (object $\mathcal{V}(a)$ is an instance of class $\mathcal{V}(b)$). From $I \models b([isa \rightarrow c])$, we have $\mathcal{V}(b) \preceq \mathcal{V}(c)$, ($\mathcal{V}(b)$ is a subclass of $\mathcal{V}(c)$). By the instOf-isa transitivity of the instance-of assertion, from instance-of pair $\langle \mathcal{V}(a), \mathcal{V}(b) \rangle$ and subclass pair $\mathcal{V}(b) \preceq \mathcal{V}(c)$, we have $\langle \mathcal{V}(a), \mathcal{V}(c) \rangle \in I_{inst}(instOf)$. By the definition of the satisfaction of an instance-of assertion, we have $I \models a([instOf \rightarrow c])$. ■

The Type Inheritance Properties

Lemma 3.6.9 *(Single-Valued-Method Type Inheritance)*
Let $a_1, \ldots, a_n, fm, p, p', q \in F^*$, *and* $t = p([fm(a_1, \ldots, a_n) \Rightarrow \{q\}])$.

$$\textit{If } I \models t \textit{ and } I \models p'([isa \rightarrow p]),\ \textit{then } I \models p'([fm(a_1, \ldots, a_n) \Rightarrow \{q\}]).$$

Intuitively, this lemma says that if p' is a subclass of p then the signature of a single-valued method defined on p is inherited by p'.

Proof: Let u denote $p'([fm(a_1, \ldots, a_n) \Rightarrow \{q\}])$. Given a variable assignment $\mathcal{V}$, it follows from the satisfaction of a single-valued method signature defined before that we have for term t:

$$\mathcal{V}(q) \in I_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Since $I \models p'([isa \rightarrow p])$, by the definition of the satisfaction of an *isa* assertion, we have $\mathcal{V}(p') \preceq \mathcal{V}(p)$. By the anti-monotonicity of $I_{\Rightarrow}(\mathcal{V}(fm))$, we have:

$$\mathcal{V}(q) \in I_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(p'), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Now, $I \models u$ follows from the definition of the satisfaction of a single-valued method signature. ∎

Lemma 3.6.10 *(Set-Valued-Method Type Inheritance)*
Let $a_1, \ldots, a_n, sm, p, p', q \in F^*$, *and* $t = p([sm(a_1, \ldots, a_n) \Rightarrow\!\!\!\Rightarrow \{q\}])$.

$$\textit{If } I \models t \textit{ and } I \models p'([isa \rightarrow p]),\ \textit{then } I \models p'([sm(a_1, \ldots, a_n) \Rightarrow\!\!\!\Rightarrow \{q\}])$$

Intuitively, this lemma says that if p' is a subclass of p then the signature of a set-valued method defined on p is inherited by p'.

Proof: Let u denote $p'([sm(a_1, \ldots, a_n) \Rrightarrow \{q\}])$. For any variable assignment $\mathcal{V}$, it follows from the satisfaction of a set-valued method signature defined before that we have for term t:

$$\mathcal{V}(q) \in I_{\Rrightarrow}(\mathcal{V}(sm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Since $I \models p'([isa \rightarrow p])$, by the definition of the satisfaction of an *isa* assertion, we have $\mathcal{V}(p') \preceq \mathcal{V}(p)$. By the anti-monotonicity of $I_{\Rrightarrow}(\mathcal{V}(sm))$, we have:

$$\mathcal{V}(q) \in I_{\Rrightarrow}(\mathcal{V}(sm))(\mathcal{V}(p'), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Now, $I \models u$ follows from the definition of the satisfaction of a set-valued method signature. ■

The Range-Supertyping Properties

Lemma 3.6.11 *(Single-Valued-Method Range Supertyping)*
Let $a_1, \ldots, a_n, fm, p, qr \in F^*$, and let $t = p([fm(a_1, \ldots, a_n) \Rightarrow \{q\}])$.

If $I \models t$ and $I \models q([isa \rightarrow r])$, then $I \models p([fm(a_1, \ldots, a_n) \Rightarrow \{r\}])$

Proof: For any variable assignment $\mathcal{V}$, it follows from the satisfaction of a single-valued-method signature defined before that we have for term t:

$$\mathcal{V}(q) \in I_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Since $I \models q([isa \rightarrow r])$, by the definition of the satisfaction of an *isa* assertion, we have $\mathcal{V}(q) \preceq \mathcal{V}(r)$. By the upward-closedness of $I_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$ we have:

$$\mathcal{V}(r) \in I_{\Rightarrow}(\mathcal{V}(fm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Now, $I \models p([fm(a_1, \ldots, a_n) \Rightarrow \{r\}])$ follows from the definition of the satisfaction of a single-valued method signature. ■

Lemma 3.6.12 *(Set-Valued-Method Range Supertyping)*
Let $a_1, \ldots, a_n, sm, p, qr \in F^*$, *and let* $t = p([sm(a_1, \ldots, a_n) \Rightarrow\!\!\!\!\Rightarrow \{q\}])$.

$$\textit{If } I \models t \textit{ and } I \models q([isa \rightarrow r]), \textit{ then } I \models p([sm(a_1, \ldots, a_n) \Rightarrow\!\!\!\!\Rightarrow \{r\}])$$

Proof: For any variable assignment $\mathcal{V}$, it follows from the satisfaction of a set-valued-method signature defined before that we have for term t:

$$\mathcal{V}(q) \in I_{\Rightarrow\!\!\!\!\Rightarrow}(\mathcal{V}(sm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Since $I \models q([isa \rightarrow r])$, by the definition of the satisfaction of an *isa* assertion, we have $\mathcal{V}(q) \preceq \mathcal{V}(r)$. By the upward-closedness of $I_{\Rightarrow\!\!\!\!\Rightarrow}(\mathcal{V}(sm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$ we have:

$$\mathcal{V}(r) \in I_{\Rightarrow\!\!\!\!\Rightarrow}(\mathcal{V}(sm))(\mathcal{V}(p), \mathcal{V}(a_1), \ldots, \mathcal{V}(a_n))$$

Now, $I \models p([sm(a_1, \ldots, a_n) \Rightarrow\!\!\!\!\Rightarrow \{r\}])$ follows from the definition of the satisfaction of a set-valued method signature. ■

The Well-Typing Conditions

Lemma 3.6.13 *(Well-Typing Properties)*

Let $a, ca, x_1, \ldots, x_n, x_1', \ldots, x_n', y, z, fm, sm \in F^*$.

- *For single-valued methods:*
  if $I \models a([fm(x_1, \ldots, x_n) \rightarrow y])$, $I \models ca([fm(x_1', \ldots, x_n') \Rightarrow \{z\}]$, *and* $I \models a([\text{instOf} \rightarrow ca])$, *then*
  - $I \models y([\text{instOf} \rightarrow z])$;
  - $I \models x_i([\text{instOf} \rightarrow x_i'])$ *(for* $1 \leq i \leq n$*);*

- *For set-valued methods:*
  if $I \models a([sm(x_1, \ldots, x_n) \twoheadrightarrow \{y\}])$, $I \models ca([sm(x_1', \ldots, x_n') \Rrightarrow \{z\}]$, *and* $I \models a([\text{instOf} \rightarrow ca])$, *then*
  - $I \models y([\text{instOf} \rightarrow z])$;
  - $I \models x_i([\text{instOf} \rightarrow x_i'])$ *(for* $1 \leq i \leq n$*)*.

Proof: The proof follows immediately from the well-typing conditions. ∎

Single-Valued Methods

Lemma 3.6.14 *(Functionality)*

Let $a, x_1, \ldots, x_n, v_1, v_2, fm \in F^*$:
If $I \models a([fm(x_1, \ldots, x_n) \rightarrow v_1])$ *and* $I \models a([fm(x_1, \ldots, x_n) \rightarrow v_2])$ *then*

$$I \models (v_1 \doteq v_2)$$

Proof: The proof follows immediately from the interpretation of single-valued methods by $I_{\rightarrow}$. ∎

Before we present the next lemma, we need the following definition.

The following definition of atomic formulae constituting a molecular formula is similar to that in F-logic [KifLaWu94].

Definition The *atomic formulae constituting a molecular formula* are defined formally as follows:

- Given a *predicate-atomic formula*, it itself is an atomic formulae.
- Given a *class term*:

$$C([\mathit{isa} \rightarrow \mathit{SuperClassOfC}], \ldots,$$
$$[FM_i(A_{i,1}, \ldots, A_{i,n_i}) \Rightarrow \{R_{i,1}, \ldots, R_{i,t_i}\}], \ldots,$$
$$[SM_j(S_{j,1}, \ldots, S_{j,l_j}) \Rrightarrow \{SR_{j,1}, \ldots SR_{j,u_j}\}], \ldots).$$

Its atomic formulae are the following class terms:

$C([\mathit{isa} \rightarrow \mathit{SuperClassOfC}])$,

$C([FM_i(A_{i,1}, \ldots, A_{i,n_i}) \Rightarrow \{\}])$,

$C([FM_i(A_{i,1}, \ldots, A_{i,n_i}) \Rightarrow \{R_{i,1}\}]), \ldots, C([FM_i(A_{i,1}, \ldots, A_{i,n_i}) \Rightarrow \{R_{i,t_i}\}])$,

$C([SM_j(S_{j,1}, \ldots, S_{j,l_j}) \Rrightarrow \{\}])$,

$C([SM_j(S_{j,1}, \ldots, S_{j,l_j}) \Rrightarrow \{SR_{j,1}\}]), \ldots, C([SM_j(S_{j,1}, \ldots, S_{j,l_j}) \Rrightarrow \{SR_{j,u_j}\}])$.

- Given a ***flat data term***:

$$D([\mathit{instOf} \rightarrow \mathit{ClassOfD}], \ldots,$$
$$[FM_i(A_{i,1}, \ldots, A_{i,n_i}) \rightarrow V_i]), \ldots,$$
$$[SM_j(S_{j,1}, \ldots, S_{j,l_j}) \twoheadrightarrow \{SV_{j,1}, \ldots, SV_{j,s_j}\}], \ldots)$$

its atomic formulae are the following data terms:

$D([\mathit{instOf} \rightarrow \mathit{ClassOfD}])$,

$D([FM_i(A_{i,1}, \ldots, A_{i,n_i}) \rightarrow V_i])$,

$D([SM_j(S_{j,1}, \ldots, S_{j,l_j}) \twoheadrightarrow \{\}])$,

$D([SM_j(S_{j,1}, \ldots, S_{j,l_j}) \twoheadrightarrow \{SV_{j,1}\}]), \ldots, D([SM_j(S_{j,1}, \ldots, S_{j,l_j}) \twoheadrightarrow \{SV_{j,s_j}\}])$.

- Given a non-flat data term, we first need to decompose it into its equivalent of a conjunction of flat data terms. Then we decompose each flat data term as shown above. ∎

Definition *An atomic formula* (or simply called an *atom*) is a molecular formula such that either it has only one constituting atomic formula, or it has only two constituting atomic formulae: itself, and an atomic formula whose output is an empty set $\{\}$. ■

Satisfaction of a Molecular Formula

Lemma 3.6.15 *(Molecular Formulae)*
Given any ground molecular formula, t,

$$I \models t \text{ iff } I \models \alpha, \text{ for each atomic formula } \alpha \text{ constituting } t.$$

Proof: Follows immediately from the definition of the satisfaction of a molecular formula. ■

3.7 Herbrand Interpretations

The proof theory that we are going to develop is resolution based, where we prove the truth of a formula from a given set of closed formulae through the unsatisfiability property. Similar to the resolution-based proof theory in classical first-order logic, we will use the following definition and proposition about unsatisfiability.

Definition Let S be a set of closed formulae of an OOL language L, and let I be a semantic structure of L. The semantic structure I is a *model* for S if I is a model for each formula of S. ■

As usual, for a finite set of closed formulae $S = \{\phi_1, \ldots, \phi_n\}$ of an OOL language L, a semantic structure I is a model of S ($I \models S$) iff I is a model for the conjunction of its elements ($I \models \phi_1 \wedge \ldots \wedge \phi_n$ iff $I \models \phi_i$ for each $i = 1, \ldots, n$).

Proposition 3.7.1 *Let S be a set of closed formulae and R be a closed formula of an OOL language L. Then R is a logical implication of S iff $S \cup \{\neg R\}$ is unsatisfiable.*

Proof: For the ***only if*** direction, assume that R is a logical entailment of S. Let I be a semantic structure of L and let I be also a model for S. Then, by the assumption, I is also a model for R. Thus I is ***not*** a model for $S \cup \{\neg R\}$, i.e., $S \cup \{\neg R\}$ is unsatisfiable.

For the ***if*** direction, assume that $S \cup \{\neg R\}$ is unsatisfiable. Let I be any semantic structure of L. Suppose that I is a model for S Because $S \cup \{\neg R\}$ is unsatisfiable, I is not a model for $\neg R$. Hence, I is a model for R (i.e., $I \models R$), this means that R is a logical entailment of S. ∎

As in classical first order logic, the problem we have is to determine $S \cup \{\neg R\}$ is unsatisfiable or not, where S is a set of closed formulae and R is a closed formula of an OOL language. To prove that $S \cup \{\neg R\}$ is unsatisfiable, we have to show that there is ***no interpretation*** satisfying $S \cup \{\neg R\}$. In classical logic, there exists a class of interpretations, i.e., Herbrand interpretations, which are the only class of interpretation for which we need to show unsatisfiability. However, it is required that the set S must be a set of clauses (a clause is defined as usual, i.e., a disjunction of literals where all variables are implicitly universally quantified at the outer most level). We need to find a way of removing existentially quantified variables from a set of closed formulae so that satisfiability is preserved. Skolemization is well known in classical first order logic to deal with existentially quantified variables.

Id-terms and quantification in OOL have been defined in a similar way to id-terms and quantification in F-logic [KifLaWu90], which in turn have been defined in a similar way to terms and quantification in Predicate Calculus. For this reason, we can use the Skolemization process and the associated Skolem theorem available in Predicate Calculus to deal with existentially quantified variables. We begin this section with the Skolem function and Skolem's theorem, because we want to assume that all formulae are Skolemized.

Theorem 3.7.1 (Skolem Function) *Given a language L of OOL, with the set of function symbols F, let $I = \langle U, I_{id}, \prec, I_{inst}, I_{\wp}, I_{\rightarrow}, I_{\Rightarrow}, I_{\twoheadrightarrow}, I_{\Rightarrow\!\!\!\Rightarrow} \rangle$, be the semantic structure. Let $\forall X_1 \ldots \forall X_n \exists Y \varphi$ be a closed formula with distinct variable symbols $X_1, \ldots, X_n, Y$. Assume that φ does not contain quantifiers $Q_1 X_1, \ldots, Q_n X_n$, where Q_i is either $\exists$ or $\forall$, for each $i = 1, \ldots, n$. We extend F with a new n-ary function symbol g. Then, whenever a semantic structure I' (i.e., the expanded I after F is extended with g) is a model of $\forall X_1 \ldots \forall X_n \varphi\{Y/g(X_1, \ldots, X_n)\}$, I' is also a model of $\forall X_1 \ldots \forall X_n \exists Y \varphi$.*

Conversely, for every semantic structure I that is a model of $\forall X_1 \ldots \forall X_n \exists Y \varphi$, F can be extended with a new function g (also called a "Skolem function") such that I' (i.e., the expanded I) is a model of $\forall X_1 \ldots \forall X_n \varphi\{Y/g(X_1, \ldots, X_n)\}$.

As a consequence, $\forall X_1 \ldots \forall X_n \exists Y \varphi$ is satisfiable if and only if

$$\forall X_1 \ldots \forall X_n \varphi\{Y/g(X_1, \ldots, X_n)\} \textit{is satisfiable.}$$

Proof: For the *if* part: The proof follows immediately from the fact that every model of $\forall X_1 \ldots \forall X_n \varphi\{Y/g(X_1, \ldots, X_n)\}$ is also a model of $\forall X_1 \ldots \forall X_n \exists Y \varphi$ is also satisfiable.

To prove the *only if* part, assume that the semantic structure I is a model of $\forall X_1 \ldots \forall X_n \exists Y \varphi$. This means that for each $\langle a_1, \ldots, a_n \rangle \in U^n$, there is an element $b \in U$ such that $I \models_{\mathcal{V}(Y/b)} \varphi$, where $\mathcal{V}(Y/b)$ is a variable assignment with $\mathcal{V}(Y/b)(X_1) = a_1, \ldots, \mathcal{V}(Y/b)(X_n) = a_n$ and $\mathcal{V}(Y/b)(Y) = b$. We define a selection function G, that supplies such an element $b = G(a_1, \ldots, a_n)$ for each tuple $\langle a_1, \ldots, a_n \rangle \in U^n$. Then $I \models_{\mathcal{V}(Y/G(a_1, \ldots, a_n))} \varphi$, for all $\langle a_1, \ldots, a_n \rangle \in U^n$ and for any variable assignment $\mathcal{V}$ such that $\mathcal{V}(X_1) = a_1, \ldots, \mathcal{V}(X_n) = a_n$. Now we expand I to I' by defining $I'_{id}(g)$ to be the chosen selection function G. Therefore, $I' \models \forall X_1 \ldots \forall X_n \varphi(Y/g(a_1, \ldots, a_n))$, or equivalently, I' is a model of $\forall X_1 \ldots \forall X_n$

$\varphi\{Y/g(X_1,\ldots,X_n)\}$. ■

Theorem 3.7.2 (cf. Skolem Theorem) *Given a closed formula, α, and its Skolemization, α', then α is unsatisfiable if and only if α' is unsatisfiable.*

Proof: The proof for this Skolem theorem is almost identical to the proof of Skolem theorem in Predicate Calculus. As in Predicate Calculus, we first convert the formula α into its prenex form. Then we replace each existentially quantified variable by a Skolem function with n arguments, where $n \geq 0$. It has been shown in Theorem 3.7.1 that Skolemization preserves satisfiability. ■

In OOL, we can embed data terms in other data terms to an arbitrary depth (*called non-flat data terms* or *nested-data terms*). It has been shown in Section 3.4.2 that we can transform any nested-data term into an equivalent conjunction of flat data terms. For example,

$$a([fmeth_1 \rightarrow id([b \rightarrow c])]) \text{ is equivalent to } a([fmeth_1 \rightarrow id]) \wedge id([b \rightarrow c]).$$

We will use this property to simplify the proof theory and notation by *focusing on clauses consisting of flat terms or P-atoms only.* By this assumption, we can avoid unnecessary complexity of our definitions, notation and analysis, without reducing the generality of our results.

Assumption

From this point, until the end of this chapter, we assume that, unless otherwise specified, all formulae are Skolemized and consist of only flat molecular formulae (also called flat molecules).

Definition Given a language $\mathcal{L}$ with a set F of function symbols, its *Herbrand universe* is the set of all ground id-terms F^*. The *Herbrand Base* (denoted by $\mathcal{B}$) is defined as the set of all ground molecular formulae. ■

Definition A subset $\mathcal{H}$ of $\mathcal{B}$ is a *Herbrand interpretation* of the language $\mathcal{L}$ if and only if *it is closed under semantic entailment* "$\models$" introduced in Section 3.4 and 3.5.■

Similar to F-logic [KifLaWu90], closedness is required here because a set of ground flat terms may imply other ground flat terms in a non-trivial way. For examples:
$\{gradStd([isa \rightarrow student]), student([isa \rightarrow person])\} \models gradStd([isa \rightarrow person])$
$\{din([children \twoheadrightarrow \{inas\}]), din([children \twoheadrightarrow \{id2\}])\} \models din([children \twoheadrightarrow \{inas, id2\}])$

Definition *Truth* in the Herbrand interpretation $\mathcal{H}$ is defined as follows:

- a ground molecular formula (a ground positive literal) γ is true in $\mathcal{H}$ iff $\gamma \in \mathcal{H}$;
- a negative ground literal $\neg l$ is true in $\mathcal{H}$ iff $l \notin \mathcal{H}$;
- a ground clause $l_1 \vee \ldots \vee l_n$, where l_i's (for $1 \leq i \leq n$) are literals, is true in $\mathcal{H}$ iff at least one l_i $(1 \leq i \leq n)$ is true in $\mathcal{H}$;
- a non-ground clause C is true in $\mathcal{H}$ iff all ground instances of C are true in $\mathcal{H}$.

■

Definition A Herbrand interpretation $\mathcal{H}$ is a *Herbrand model* of a set of clauses $\boldsymbol{S}$ if every clause in $\boldsymbol{S}$ is true in $\mathcal{H}$ (denoted by $\mathcal{H} \models \boldsymbol{S}$). ■

Closure Properties of a Herbrand Interpretation

Because of the closure requirement under "$\models$," a Herbrand interpretation has the following closure properties (which can be verified straight forwardly from the semantic-entailment properties presented in Section 3.6).

- Equality predicate-atomic formulae form a congruence relation, i.e., *reflexivity, symmetry, transitivity,* and *substitution. Reflexivity, symmetry and transitivity* properties are defined as usual. The *substitution* property is as follows: if $x \doteq$

y, $l \in \mathcal{H}$ and l' is the result of replacing an occurrence of x in l by y, then $l' \in \mathcal{H}$.

- The *isa* relationships (subclass relationships) among class objects in $\mathcal{H}$ form a *partial order* relation (*reflexive, transitive,* and *antisymmetric*).
  - *reflexivity property:*
    - If $a([isa \rightarrow p]) \in \mathcal{H}$ then $a([isa \rightarrow a])\mathcal{H}$ and $p([isa \rightarrow p])\mathcal{H}$.
    - If $q([instOf \rightarrow a]) \in \mathcal{H}$ then $a([isa \rightarrow a]) \in \mathcal{H}$.
    - If $a([fm(b_1, \ldots, b_n) \Rightarrow \{c\}] \in \mathcal{H}$, then
      $a([isa \rightarrow a]) \in \mathcal{H}$,
      $b_1([isa \rightarrow b_1]) \in \mathcal{H}, \ldots, b_n([isa \rightarrow b_n]) \in \mathcal{H}$, and
      $c([isa \rightarrow c]) \in \mathcal{H}$.
    - If $a([sm(b_1, \ldots, b_n) \Rrightarrow \{c\}] \in \mathcal{H}$, then
      $a([isa \rightarrow a]) \in \mathcal{H}$,
      $b_1([isa \rightarrow b_1]) \in \mathcal{H}, \ldots, b_n([isa \rightarrow b_n]) \in \mathcal{H}$, and
      $c([isa \rightarrow c]) \in \mathcal{H}$.
  - *transitivity property:*
    If $a([isa \rightarrow b]), b([isa \rightarrow c]) \in \mathcal{H}$, then $a([isa \rightarrow c]) \in \mathcal{H}$.
  - *antisymmetry property:*
    If $I \models a([isa \rightarrow b]), b([isa \rightarrow a]) \in \mathcal{H}$, then $(a \doteq b) \in \mathcal{H}$.
- The *instOf* and *isa* attributes have the following property:
  if $a([instOf \rightarrow b]) \in \mathcal{H}$ and $b([isa \rightarrow c]) \in \mathcal{H}$ then $a([instOf \rightarrow c]) \in \mathcal{H}$.
- For *class* terms, there are properties as follows:
  - *type inheritance* properties:

    * if $a([fm(x_1, \ldots, x_n) \Rightarrow \{y\}]),\ b([isa \rightarrow a]) \in \mathcal{H}$ then $b([fm(x_1, \ldots, x_n) \Rightarrow \{y\}]) \in \mathcal{H}$;
    * if $a([sm(x_1, \ldots, x_n) \Rrightarrow \{y\}]),\ b([isa \rightarrow a]) \in \mathcal{H}$ then $b([sm(x_1, \ldots, x_n) \Rrightarrow \{y\}]) \in \mathcal{H}$.
  - *range supertyping* properties:
    * if $a([fm(x_1, \ldots, x_n) \Rightarrow \{y\}]),\ y([isa \rightarrow z]) \in \mathcal{H}$ then $a([fm(x_1, \ldots, x_n) \Rightarrow \{z\}]) \in \mathcal{H}$;
    * if $a([sm(x_1, \ldots, x_n) \Rrightarrow \{y\}]),\ y([isa \rightarrow z]) \in \mathcal{H}$ then $a([sm(x_1, \ldots, x_n) \Rrightarrow \{z\}]) \in \mathcal{H}$.

- *The well-typing* properties:
  - if $a([fm(x_1, \ldots, x_n) \rightarrow y]),\ ca([fm(x'_1, \ldots, x'_n) \Rightarrow \{z\}]) \in \mathcal{H}$, and $a([instOf \rightarrow ca]) \in \mathcal{H}$, then
    * $y([instOf \rightarrow z]) \in \mathcal{H}$;
    * $x_i([instOf \rightarrow x'_i]) \in \mathcal{H}$ (for $1 \leq i \leq n$);
  - if $a([sm(x_1, \ldots, x_n) \twoheadrightarrow \{y\}]), ca([sm(x'_1, \ldots, x'_n) \Rrightarrow \{z\}]) \in \mathcal{H}$, and $a([instOf \rightarrow ca]) \in \mathcal{H}$, then
    * $y([instOf \rightarrow z]) \in \mathcal{H}$;
    * $x_i([instOf \rightarrow x'_i]) \in \mathcal{H}$ (for $1 \leq i \leq n$).

- Data terms have the *functionality* property. This means that if, as an example, we have:
  $a([fm(x_1, \ldots, x_n) \rightarrow v_1]),\ a([fm(x_1, \ldots, x_n) \rightarrow v_2]) \in \mathcal{H}$, then $v_1 \doteq v_2 \in \mathcal{H}$.

- For each molecular formula t: $t \in \mathcal{H}$ iff $\alpha \in \mathcal{H}$ for every atomic formula α constituting t.

3.8 Herbrand Interpretations and Semantic Structures

This section describes the relation between the Herbrand interpretations and the semantic structures of OOL described in Section 3.4. The relation between the Herbrand Interpretations and the semantic structures is defined similarly to that of F-logic [KifLaWu90].

The Herbrand Interpretation of a Semantic Structure

Given an OOL language L, let I be a semantic structure of a set of clauses $\boldsymbol{S}$. The Herbrand interpretation of I is the set of ground flat molecules that are true in I.

The Semantic Structure of a Herbrand Interpretation

For a Herbrand interpretation $\mathcal{H}$, the corresponding semantic structure:

$$I_{\mathcal{H}} = \langle U, I_{id}, \prec, I_{inst}, I_{\wp}, I_{\rightarrow}, I_{\rightarrow\!\!\!\rightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow} \rangle$$

is defined as follows:

1. The domain of interpretation U is the quotient of the set of all ground id-terms F^* by the equality predicate-atomic formulae in $\mathcal{H}$, i.e., $F^*/\doteq$. The equivalence class of an id-term t is denoted by $[t]$.

2.
   - $I_{id}(c) = [c]$, for each 0-ary function (constant) symbol $c \in F$.
   - $I_{id}(f)([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$, for every n-ary ($n \geq 1$) function symbol $f \in F$.

3. The ordering of class objects in U is defined by the *isa* assertions in $\mathcal{H}$: for all class objects $[a], [b] \in U$, $[a] \preceq [b]$ iff $a([isa \rightarrow b]) \in \mathcal{H}$.

4. The instance-of relation in U is defined by the *instOf* assertions in $\mathcal{H}$: for all objects $[c], [d] \in U$, $\langle [d], [c] \rangle \in I_{inst}(instOf)$ iff $d([instOf \rightarrow c]) \in \mathcal{H}$.

5. $I^n_{\rightarrow}([funM])([obj],[t_1],\ldots,[t_n]) =$
$$\begin{cases} [v] \text{ if } obj([funM(t_1,\ldots,t_n) \rightarrow v]) \in \mathcal{H} \\ \text{otherwise: undefined.} \end{cases}$$

6. $I^n_{\rightarrow\!\!\!\rightarrow}\ ([setM])([obj],[t_1],\ldots,[t_n]) =$
$$\begin{cases} \{\ [v] \mid obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{v\}]) \in \mathcal{H}\ \} \text{ if} \\ \quad obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{\}]) \in \mathcal{H} \\ \text{otherwise: undefined.} \end{cases}$$

7. $I^n_{\Rightarrow}\ ([funM])([obj],[t_1],\ldots,[t_n]) =$
$$\begin{cases} \{\ [v] \mid obj([funM(t_1,\ldots,t_n) \Rightarrow \{v\}]) \in \mathcal{H}\ \} \text{ if} \\ \quad obj([funM(t_1,\ldots,t_n) \Rightarrow \{\}]) \in \mathcal{H} \\ \text{otherwise: undefined.} \end{cases}$$

8. $I^n_{\Rightarrow\!\!\!\Rightarrow}\ ([setM])([obj],[t_1],\ldots,[t_n]) =$
$$\begin{cases} \{\ [v] \mid obj([setM(t_1,\ldots,t_n) \Rightarrow\!\!\!\Rightarrow \{v\}]) \in \mathcal{H}\ \} \text{ if} \\ \quad obj([setM(t_1,\ldots,t_n) \Rightarrow\!\!\!\Rightarrow \{\}]) \in \mathcal{H} \\ \text{otherwise: undefined.} \end{cases}$$

9. $I_{\wp}(p) = \{\ \langle [t_1], \ldots [t_n] \rangle \mid p(t_1, \ldots t_n) \in \mathcal{H}\ \}.$

The constructed $I_{\mathcal{H}} = \langle U, I_{id}, \prec, I_{inst}, I_{\wp}, I_{\rightarrow}, I_{\rightarrow\!\!\!\rightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow} \rangle$ is indeed a semantic structure because of the way its components are defined and the closure of the Herbrand interpretation $\mathcal{H}$ under the semantic entailment "$\models$."

It follows from the definition of $I^n_{\rightarrow\!\!\!\rightarrow}\ ([setM])([obj,[t_1],\ldots,[t_n])$ in (6) that being "undefined" and being "empty" are different. It is undefined in (6) if $\mathcal{H}$ does not contain any atom of the form $obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \ldots])$, nor an atom of the form $obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{\}])$. On the other hand $I^n_{\rightarrow\!\!\!\rightarrow}\ ([setM])([obj,[t_1],\ldots,[t_n])$ is empty if $\mathcal{H}$ contains $obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{\}])$, but it does not contain any atom of the form $obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{v\}])$ for some $v \in F^*$. Similarly for $I^n_{\Rightarrow}\ ([funM])([obj],[t_1],\ldots,t_n])$ and $I^n_{\Rightarrow\!\!\!\Rightarrow}\ ([setM])([obj],[t_1],\ldots,[t_n])$.

Proposition 3.8.1 *Given a set S of clauses, $I_{\mathcal{H}} \models S$ if $\mathcal{H} \models S$.*

Proof: We need to show that for each clause $c \in S$, if $\mathcal{H} \models c$ then $I_{\mathcal{H}} \models c$. We show this by considering the following cases:

1. c is a ground molecule:

   By the definition of $I_{\mathcal{H}}$ it is straight forward to see that $I_{\mathcal{H}} \models c$ if $\mathcal{H} \models c$.

2. c is a non-ground molecule:

   $I_{\mathcal{H}} \models c$ iff $I_{\mathcal{H}}$ satisfies every ground instance of c. Since $\mathcal{H} \models c$, every ground instance of c is an element of $\mathcal{H}$. Thus, by the first case, $I_{\mathcal{H}} \models$ every ground instance of c. It follows that $I_{\mathcal{H}} \models c$

3. c is a negative-ground literal $\neg g$:

   $\mathcal{H} \models \neg g$ iff $g \notin \mathcal{H}$. By the definition of $I_{\mathcal{H}}$, if $g \notin \mathcal{H}$ then $I_{\mathcal{H}} \not\models g$. Thus $I_{\mathcal{H}} \models \neg g$.

4. c is a negative-non-ground literal $\neg l$:

   $I_{\mathcal{H}} \models \neg l$ iff $I_{\mathcal{H}} \not\models$ every ground instance of l. Since $\mathcal{H} \models \neg l$, every ground instance of l is *not* an element of $\mathcal{H}$. Hence, by the first case, $I_{\mathcal{H}} \not\models$ every ground instance of l. It follows that $I_{\mathcal{H}} \models \neg l$.

5. c is a disjunction of literals $l_1 \vee \ldots \vee l_n$:

   $I_{\mathcal{H}} \models l_1 \vee \ldots \vee l_n$ iff $I_{\mathcal{H}} \models l_i$ for some i (by the usual definition of satisfying a disjunction of literals). From the results of the previous cases, $I_{\mathcal{H}} \models l_i$ if $\mathcal{H} \models l_i$

The above cases show that if $\mathcal{H} \models c$ then $I_{\mathcal{H}} \models c$. ∎

Proposition 3.8.2 *Given a set S of clauses, then S is unsatisfiable if and only if S has no Herbrand model.*

Proof: The proof for the *only if* part is shown by a contradiction. Assume that S is unsatisfiable, but S has a Herbrand model. It follows that because S has a Herbrand model, by proposition 3.8.1, S is satisfiable. A contradiction!

The *if* part is proved by a contradiction, too. Assume that there is no Herbrand model but S is satisfiable. We will show that if S is satisfiable then S has a Herbrand model. From a semantic structure I that satisfies S, a Herbrand interpretation $\mathcal{H}$ is defined as the set of all ground molecules that are true in I. Because I satisfies S, the way the Herbrand interpretation is defined (i.e., the set of all ground molecules that are true in I), and the notion of truth in a Herbrand interpretation (as defined earlier), then $\mathcal{H}$ will also satisfy S. Thus S has a Herbrand model. A contradiction! ∎

3.9 Herbrand's Theorem

The Herbrand theorem in classical logic says that a set S of clauses is unsatisfiable if and only if some finite subset of ground instances of clauses in S is unsatisfiable. As in classical logic, this theorem plays a similar-important role in OOL. This theorem is proved by considering maximal finitely satisfiable sets, similarly to that of F-logic [KifLaWu90], which in turn is similar to the proof of compactness theorem in [Ende72].

Definition A set G of ground clauses is *finitely satisfiable*, if every finite subset of G is satisfiable. ∎

Definition A finitely-satisfiable set G of ground clauses is *maximal* if no other set of ground clauses containing G is finitely satisfiable. ∎

Lemma 3.9.1 *Let S be a finitely satisfiable set of ground clauses. Then there exists a maximal finitely satisfiable set T such that $S \subseteq T$.*

Proof: Let $\boldsymbol{A}$ be a set of *all finitely satisfiable sets of ground clauses that contain* $\boldsymbol{S}$. $\boldsymbol{A}$ is partially ordered by subset relation $\subseteq$. It is also a non-empty set because $\boldsymbol{S} \in \boldsymbol{A}$. For every *chain*[3] $\Delta \subseteq \boldsymbol{A}$, the least upper bound of the chain, Υ, is also in $\boldsymbol{A}$. This is because Υ contains $\boldsymbol{S}$ and is finitely satisfiable, thus by definition of $\boldsymbol{A}$ we conclude that $\Upsilon \in \boldsymbol{A}$. Based on Zorn's lemma[4] there will be a maximal element $\boldsymbol{T} \in \boldsymbol{A}$. By the definition of $\boldsymbol{A}$, the maximal element $\boldsymbol{T}$ contains $\boldsymbol{S}$ (i.e., $\boldsymbol{S} \subseteq \boldsymbol{T}$) and $\boldsymbol{T}$ is finitely satisfiable. ∎

Lemma 3.9.2 *Let $\boldsymbol{T}$ be a maximal finitely satisfiable set of ground clauses. Then:*

1. *For every ground molecule g either $g \in \boldsymbol{T}$ or $\neg g \in \boldsymbol{T}$.*

2. *For every ground clause, $l_1 \vee \ldots \vee l_n$, $\in \boldsymbol{T}$ if and only if some $l_i \in \boldsymbol{T}$, where $1 \leq i \leq n$.*

Proof: (1) Since $\boldsymbol{T}$ is finitely satisfiable, then we have either $\boldsymbol{T}\cup\{g\}$ or $\boldsymbol{T}\cup\{\neg g\}$ is finitely satisfiable. Because $\boldsymbol{T}$ is maximal, then it contains either g or $\neg g$. (2) Similarly to (1), since $\boldsymbol{T}$ is finitely satisfiable, then we have either $\boldsymbol{T}\cup\{l_i\}$ or $\boldsymbol{T}\cup\{\neg l_i\}$ is finitely satisfiable, where $1 \leq i \leq n$. Since $\boldsymbol{T}$ is maximal, and for satisfying a disjunction literals, at least one of the literals must be satisfiable, then $\boldsymbol{T}$ contains l_i for some $i = 1, \ldots, n$. ∎

Lemma 3.9.3 *Let $\boldsymbol{T}$ be a maximal finitely satisfiable set of ground clauses and $\mathcal{H}$ be the set of all ground molecules (positive literals) in $\boldsymbol{T}$. Then $\mathcal{H}$ is a Herbrand interpretation.*

[3]Given a set A that is partially ordered by subset relation ($\subseteq$), any subset $\Delta \subseteq A$ is a *chain (fully ordered set)* if and only if for all $x, y \in \Delta$, either $x \subseteq y$ or $y \subseteq x$.

[4]Zorn's Lemma: *if X is a non-empty partially ordered set such that every chain $\subseteq X$ has an upper bound, then X contains a maximal element.*

Proof: The set $\boldsymbol{T}$ is clearly closed under $\models$ because otherwise it is not maximal (by definition). Since the interpretation $\mathcal{H}$ is defined as the set of all ground molecules in $\boldsymbol{T}$, $\mathcal{H}$ is also closed under $\models$. Thus, by the definition of a Herbrand interpretation, $\mathcal{H}$ is a Herbrand interpretation. ■

Theorem 3.9.1 (cf. Herbrand's Theorem) *A set of clauses $\boldsymbol{S}$ is unsatisfiable if and only if some finite subset of ground instances of clauses in $\boldsymbol{S}$ is unsatisfiable.*

Proof: The proof of the *if* part: If some finite subset of ground clauses of $\boldsymbol{S}$ is unsatisfiable, then $\boldsymbol{S}$ is also unsatisfiable, by the definition of satisfiability. The *only if* part is proved by contradiction. Assume that $\boldsymbol{S}$ is unsatisfiable, but the set $\boldsymbol{G}$ of ground instances of clauses in $\boldsymbol{S}$ is finitely satisfiable. Then we will show that $\boldsymbol{S}$ is satisfiable.

$\boldsymbol{G}$ is assumed to be finitely satisfiable. Based on Lemma 3.9.1, $\boldsymbol{G}$ can be extended to some maximal finitely satisfiable set of ground clauses, $\boldsymbol{T}$. $\mathcal{H}$ is defined as the set of all ground molecules (positive literals) in $\boldsymbol{T}$, then based on Lemma 3.9.3 we conclude that $\mathcal{H}$ is a Herbrand interpretation. Now, we are going to show that for every ground clause c, $\mathcal{H} \models c$ if and only if $c \in \boldsymbol{T}$. We show this by considering the following cases:

1. c is a ground molecule: by the definition of $\mathcal{H}$ in Lemma 3.9.3, $\mathcal{H} \models c$ iff $c \in \boldsymbol{T}$;

2. c is a negative-ground literal $\neg g$, then $\mathcal{H} \models \neg g$ iff $g \notin \mathcal{H}$. By definition, $\mathcal{H}$ contains all the molecules in $\boldsymbol{T}$, thus $g \notin \mathcal{H}$ iff $g \notin \boldsymbol{T}$. By the first part of Lemma 3.9.2, $\neg g \in \boldsymbol{T}$

3. c is a disjunction of ground literals $l_1 \vee \ldots \vee l_n$:
   $\mathcal{H} \models l_1 \vee \ldots \vee l_n$ iff $\mathcal{H} \models l_i$ for some i (by the usual definition of satisfying a disjunction of literals) iff $l_i \in \boldsymbol{T}$ (based on results 1 and 2)
   iff $l_1 \vee \ldots \vee l_n \in \boldsymbol{T}$ (based on the second part of Lemma 3.9.2).

The above cases show that $\mathcal{H}$ satisfies every clause in $\boldsymbol{T}$, thus $\mathcal{H}$ is a Herbrand model of $\boldsymbol{T}$. Since $\boldsymbol{G} \subseteq \boldsymbol{T}$, $\mathcal{H}$ is also a Herbrand model of $\boldsymbol{G}$. Furthermore, because $\boldsymbol{G}$ is the set of all ground instances of clauses in $\boldsymbol{S}$, $\mathcal{H}$ is also a Herbrand model of $\boldsymbol{S}$. Finally, based on Proposition 3.8.2, $\boldsymbol{S}$ is satisfable. ∎

3.10 Proof Theory

We shall present a sound and complete proof theory for the semantic-entailment ($\models$) relation defined in Section 3.4. There are twelve inference rules in the proof theory to make the theory complete. The core rules of the proof theory are resolution, factoring and paramodulation.

3.10.1 Substitution

The notion of ***substitution*** is adapted from classical logic. Given a language $\mathcal{L}$ and a set of variables V, a ***substitution*** σ is a mapping:

$$V \longrightarrow \{\text{id-terms of } \mathcal{L}\}$$

where the domain of the substitution is a finite set $domain(\sigma) \subseteq V$, such that for any $Y \in domain(\sigma)$, $\sigma(Y) \neq Y$. This notion of substitution is extended to id-terms (respectively, predicate-atomic formulae) in a similar way to substitution on "terms" (respectively, "atomic formulae") in classical first-order logic, as follows:

$$\sigma(f\langle t_1, \ldots, t_n\rangle) = f\langle \sigma(t_1), \ldots, \sigma(t_n)\rangle$$

$$\sigma(P(t_1, \ldots, t_n)) = P(\sigma(t_1), \ldots, \sigma(t_n))$$

Where f is a function symbol, P is a predicate symbol, and $t_1, \ldots, t_n$ are id-terms.

Substitution σ is further extended to data terms and class terms as follows:

$$\sigma(D([instOf \rightarrow E], [FM(A_1, \ldots, A_n) \rightarrow T], \ldots,$$
$$[SM(B_1, \ldots, B_n) \twoheadrightarrow \{\ldots, S, \ldots\}], \ldots)) =$$
$$\sigma(D)([instOf \rightarrow \sigma(E)], [\sigma(FM)(\sigma(A_1), \ldots, \sigma(A_n)) \rightarrow \sigma(T)], \ldots,$$
$$[\sigma(SM)(\sigma(B_1), \ldots, \sigma(B_n)) \twoheadrightarrow \{\ldots, \sigma(S), \ldots\}], \ldots)$$

$$\sigma(C([isa \rightarrow D], [FM(A_1, \ldots, A_n) \Rightarrow \{\ldots, T, \ldots\}], \ldots,$$
$$[SM(B_1, \ldots, B_n) \Rrightarrow \{\ldots, S, \ldots\}], \ldots)) =$$
$$\sigma(C)([isa \rightarrow \sigma(D)], [\sigma(FM)(\sigma(A_1), \ldots, \sigma(A_n)) \Rightarrow \{\ldots, \sigma(T), \ldots\}], \ldots,$$
$$[\sigma(SM)(\sigma(B_1), \ldots, \sigma(B_n)) \Rrightarrow \{\ldots, \sigma(S), \ldots\}], \ldots)$$

Finally, substitution σ is extended to OOL formulae by letting the substitution commute with logical connectives and quantifiers.

Substitution σ is called a *ground substitution* if $\sigma(Y) \in F^*$ (the set of all ground id-terms) for each $Y \in domain(\sigma)$. Let σ be a substitution and α be a formula. Then $\sigma(\alpha)$ is called an *instance* of α. This instance $\sigma(\alpha)$ is called a *ground instance* if it does not contain any variable.

3.10.2 Unification

Unification of id-terms and predicate-atomic formulae (abbreviated, P-atoms) are defined as in Predicate Calculus.

Definition Let T_1 and T_2 be a pair of id-terms. A substitution σ is a unifier of T_1 and T_2 if and only if $\sigma(T_1) = \sigma(T_2)$. This unifier σ is the *most general unifier* (mgu(T_1, T_2)), if for any other unifier μ of T_1 and T_2, there exists a substitution γ, such that $\mu = \gamma \circ \sigma$. ■

The notion of mgu is extended to include ordered tuples of id-terms.

Definition Let $\langle A_1, \ldots, A_n \rangle$ and $\langle B_1, \ldots, B_n \rangle$ be ordered tuples of id-terms. These two tuples are ***unifiable*** if and only if there is a substitution σ such that $\sigma(A_i) = \sigma(B_i)$, for each $i = 1, \ldots, n$. The unifier σ is *most general* if and only if for any other unifier μ of the two tuples, there exists a substitution γ, such that $\mu = \gamma \circ \sigma$. ∎

We can observe that the mgu of $\langle A_1, \ldots, A_n \rangle$ and $\langle B_1, \ldots, B_n \rangle$ will coincide with the mgu of $f(A_1, \ldots, A_n)$ and $f(B_1, \ldots, B_n)$, where f is an n-ary function symbol. This will allows us to use any standard algorithm for unifying Predicate Calculus' terms of this type.

The notions of unifier and the most general unifier are further extended to (OOL) terms. For an OOL term (i.e., either a data or a class term), a unifier is only required to have a term be a ***subterm*** of the other. First we define what subterm means.

Definition Let $T_1 = Id(\ldots, [\ldots], \ldots)$ and $T_2 = Id(\ldots, [\ldots], \ldots)$ be a pair of OOL terms (i.e., either data terms or class terms) with the same object identity Id. T_1 *is a subterm of* T_2, denoted by $T_1 \sqsubseteq T_2$, if and only if every atomic formula of T_1 is also an atomic formula of T_2. ∎

Definition *A substitution σ is a unifier of T_1 into T_2 if and only if $\sigma[T_1] \sqsubseteq \sigma[T_2]$.* ∎

As in F-logic [KifLaWu90], we need to modify the definition of mgu for this kind of unification.

Definition Let T_1 and T_2 be a pair of class or data terms and let α and β be a pair of unifiers of T_1 into T_2. We say that α is *more general* than β (denoted by $\alpha \trianglelefteq \beta$) if and only if there is a substitution γ such that $\beta = \gamma \circ \alpha$. *A unifier α of T_1 into T_2 is most general if for every other unifier β, $\beta \trianglelefteq \alpha$ implies $\alpha \trianglelefteq \beta$.* ∎

In order to distinguish this kind of most general unifier from the usual mgu, notation $mgu_{\sqsubseteq}(T_1 into T_2)$ is used to denote a most general unifier of T_1 *into* T_2.

Examples:

1. Let $t_1 = P([fm(A,B) \rightarrow C])$ and let $t_2 = x([fm(e,f) \rightarrow g],[sm \twoheadrightarrow \{g,h\}])$. The term t_1 is unifiable into t_2 with the unifier $\{P/x,\ A/e,\ B/f,\ C/g\}$. However t_2 is not unifiable into t_1 because it is impossible to have every atomic formula of t_2 also be an atomic formula of t_1.

2. Let $t_1 = a([sm \twoheadrightarrow B])$ and let $t_2 = a([sm \twoheadrightarrow \{c,d,e\}])$. We can find three different unifiers of t_1 into t_2, i.e., $\{B/c\}$, $\{B/d\}$, and $\{B/e\}$. However we cannot say that any one of them is more general than another, thus any one of them can be called most general. This situation has motivated the following definition of a *complete* set of $mgu_{\sqsubseteq}$ [KifLaWu90].

Definition A set Σ of most general unifiers of T_1 into T_2 is *complete* if for every other unifier θ of T_1 into T_2, there is an $\alpha \in \Sigma$ such that $\alpha \trianglelefteq \theta$. ■

This complete set is a unique-up-to-the-equivalence mgu$_{\sqsubseteq}$ case, in which a set of unifiers Ω_1 is equivalent to Ω_2 if for every $\sigma_1 \in \Omega_1$, there is $\sigma_2 \in \Omega_2$ such that $\sigma_1 \trianglelefteq \sigma_2 \trianglelefteq \sigma_1$, and vice versa.

3.10.2.1 A Term Unification Algorithm

We will describe an algorithm to find a complete set of $mgu_{\sqsubseteq}$'s for a pair of OOL terms (i.e., either data or class terms). This algorithm and its correctness lemma are adapted from [KifLaWu90], with minor changes. A complete set of $mgu_{\sqsubseteq}$'s is not unique. However, since any two complete sets of $mgu_{\sqsubseteq}$'s are equivalent to each other, we need to find only one of them.

Since an OOL term is equivalent to a conjunction of its constituting atomic formulae, and an $mgu_{\sqsubseteq}$ of T_1 into T_2 is only required to cause each atom of T_1 to be also an atom of T_2, different correspondences between atoms of T_1 and T_2 may lead to different $mgu_{\sqsubseteq}$'s. Because of this, an algorithm for finding a complete set of $mgu_{\sqsubseteq}$'s from a pair of terms should consider all such correspondences.

Let α be an atom of one of the following forms:

1. An atom with a non-empty value, in either of the following forms:
   - $D([M(A_1,\ldots,A_n) \rightarrow B])$; or $D([M(A_1,\ldots,A_n) \Rightarrow \{B\}])$; or
   - $D([M(A_1,\ldots,A_n) \twoheadrightarrow \{B\}])$; or $D([M(A_1,\ldots,A_n) \Rightarrow\!\!\!\!\Rightarrow \{B\}])$; or
   - $D([instOf \rightarrow B])$; or $D([isa \rightarrow B])$.
2. An atom with an empty value, in either of the following forms:
   - $D([M(A_1,\ldots,A_n) \Rightarrow \{\}])$; or
   - $D([M(A_1,\ldots,A_n) \twoheadrightarrow \{\}])$; or $D([M(A_1,\ldots,A_n) \Rightarrow\!\!\!\!\Rightarrow \{\}])$.

In order to simplify the presentation of the unification algorithm, we will use the following notation:

- $oid(T) = D$, i.e., *oid* returns the context id-term of an OOL term T;
- $method(\alpha) = M$, i.e., returning the id-term of the method in atom α, or for the instance-of and subclass assertions: $method(\alpha) = instOf$ and $method(\alpha) = isa$ respectively;
- $arg_i(\alpha) = A_i \; (i = 1, \ldots, n)$;
- $val(\alpha) = B$ if α is of the form 1, or $val(\alpha) = \emptyset$ if α is of the form 2;
- if T is an OOL term, $atoms(T)$ denotes the set of atoms in T.

- if T_1 and T_2 are terms, then *maps*(T_1, T_2) denotes the set of all possible mappings $\{\Psi : \textit{atoms}(T_1) \longrightarrow \textit{atoms}(T_2)\}$ such that for each $\Psi \in \textit{maps}(T_1, T_2)$ and each $\alpha \in \textit{atoms}(T_1)$ the arities, and the arrow symbols of the methods as well as instance-of/subclass assertions in α , and $\Psi(\alpha)$ are the same.

- We have shown earlier that the mgu of a pair of tuples of id-terms $\langle A_1, \ldots, A_n\rangle$ and $\langle B_1, \ldots, B_n\rangle$ coincides with the mgu of the usual Predicate Calculus terms $f(A_1, \ldots, A_n)$ and $f(B_1, \ldots, B_n)$. Hence, we can use any standard unification algorithm to unify a pair of id-term tuples. We use the following notation: *unify*$(\vec{V_1}, \vec{V_2})$, to denote the unification procedure of a pair of id-term tuples $\vec{V_1}$ and $\vec{V_2}$.

The algorithm is given in Figure 3.1.

Lemma 3.10.1 (Term Unification) *The given algorithm indeed finds a complete set of* $mgu_{\sqsubseteq}$*'s of* T_1 *into* T_2

Proof: It is stated explicitly in the algorithm that all elements of Σ (i.e., σ_Ψ's) are $mgu_{\sqsubseteq}$'s of T_1 into T_2. Thus we just need to show that Σ is indeed complete.

Let γ be a unifier of T_1 into T_2:

- By the definition of Ψ, there is a mapping $\Psi \in \textit{maps}(T_1, T_2)$ such that the mapping of each each $\alpha \in \textit{atoms}(T_1)$ into the corresponding atom $\Psi(\alpha) \in T_2$ coincides with a mapping associated with the unifier γ.

- It follows directly from the the algorithm, that substitution σ_Ψ constructed in the inner loop of step (2) is a most general unifier that unifies each $\alpha \in \textit{atoms}(T_1)$ into the corresponding atom $\Psi(\alpha) \in T_2$. Based on the definition of $mgu_{\sqsubseteq}$, we conclude that the unifier $\sigma_\Psi \trianglelefteq \gamma$ (γ is an instance of σ_Ψ).

Input: a pair of OOL terms T_1 and T_2.

Output: a complete set Σ of $mgu_{\sqsubseteq}$'s of T_1 **into** T_2.

```
1. if $oid(T_1)$ and $oid(T_2)$ are unifiable then $\theta \leftarrow unify(oid(T_1), oid(T_2))$
     else STOP, $T_1$ and $T_2$ are NOT unifiable.
2. $\Sigma \leftarrow \{\}$        /*$\Sigma$ will be used to denote the complete sets of $mgu_{\sqsubseteq}$'s*/
   for each mapping $\Psi \in maps(T_1, T_2)$ do
     $\sigma_\Psi \leftarrow \theta$    /*$\sigma_\Psi$ is used to store an $mgu_{\sqsubseteq}$*/
     for each atom $\alpha \in atoms(T_1)$ do
       $\psi \leftarrow \Psi(\alpha)$
       if $val(\alpha) = \emptyset$
       then $unify(\sigma_\Psi(\vec{V_1}), \sigma_\Psi(\vec{V_2}))$, where $\vec{V_1} = \langle method(\alpha), arg_1(\alpha), \ldots, arg_n(\alpha)\rangle$
                                                  $\vec{V_2} = \langle method(\psi), arg_1(\psi), \ldots, arg_n(\psi)\rangle$
            if $\sigma_\Psi(\vec{V_1})$ and $\sigma_\Psi(\vec{V_2})$ are unifiable
              then $\sigma_\Psi \leftarrow unify(\sigma_\Psi(\vec{V_1}), \sigma_\Psi(\vec{V_2})) \circ \sigma_\Psi$
              else DISCARD $\sigma_\Psi$ and GOTO (*), to select another $\Psi$
            endif
       else unify$(\sigma_\Psi(\vec{V_1}), \sigma_\Psi(\vec{V_2}))$,
            where $\vec{V_1} = \langle method(\alpha), arg_1(\alpha), \ldots, arg_n(\alpha), val(\alpha)\rangle$
                  $\vec{V_2} = \langle method(\psi), arg_1(\psi), \ldots, arg_n(\psi), val(\psi)\rangle$
            if $\sigma_\Psi(\vec{V_1})$ and $\sigma_\Psi(\vec{V_2})$ are unifiable
              then $\sigma_\Psi \leftarrow unify(\sigma_\Psi(\vec{V_1}), \sigma_\Psi(\vec{V_2})) \circ \sigma_\Psi$
              else DISCARD $\sigma_\Psi$ and GOTO (*), to select another $\Psi$
            endif
       endif
     endfor
     $\Sigma \leftarrow \Sigma \cup \{\sigma_\Psi\}$
   (*)
   endfor
3. Return $\Sigma$ which is a complete set of $mgu_{\sqsubseteq}$'s of $T_1$ into $T_2$.
Note that the set $\Sigma$ will be empty if $T_1$ is not unifiable into $T_2$.
```

Figure 3.1: An Algorithm for Finding a Complete Set of $mgu_{\sqsubseteq}$'s

Based on the two consideration above we conclude that Σ, which is the union of most general unifiers σ_{Ψ}'s, is a complete set of $mgu_{\sqsubseteq}$'s of T_1 into T_2. ■

3.10.3 Inference Rules

As in classical predicate logic, we have to standardize apart the clauses involved in the unification operation where variables are renamed so that these clauses will not have common variable names. Note that in the inference rules to be presented, occasionally we show only atomic formulae (or their negation) as literals instead of flat molecules. This is done only to simplify the presentation of the inference rules.

Seven of the inference rules (from a total of twelve rules) are similar to that of F-logic [KifLaWu90]. The inference rules of this OOL proof theory differ from that of F-logic in the following rules:

- Isa-Reflexivity Rule.
  Since OOL distinguishes between instance objects and class objects, the isa-reflexivity property is applicable to class objects only.

- Argument-Subtyping Rule.
  This rule is not available in the proof theory of OOL, because the antimonotonic property of a method signature in OOL is applicable to the first argument only, i.e., for the context class.

- Well-Typing Rule.
  In F-logic [KifLaWu90], method signatures may be defined on both the class(es) of an instance object and the instance object itself. On the other hand, in OOL, method signatures are defined only on class objects.

- InstOf-Isa Transitivity Rule.

The proof theory of F-logic [KifLaWu90] does not have this kind of rule, since F-logic does not distinguish between instance-of relation and isa (subclass) relation.

- Elimination Rule.

  F-logic considers an empty data term or class term of the form *Id*(), where *Id* is an id-term, as a tautology. On the contrary, OOL does not consider such an empty term as either a data term or a class term (i.e., an empty term is not an OOL term). Consequently, the proof theory of OOL does not need a rule to deal with empty terms.

The inference rules of the proof theory for OOL are as follows:

1. Resolution Rule

Let $C_1 = \neg T_1 \vee C'$ and $C_2 = T_2 \vee C''$ be a pair of clauses with no variables in common (standardized apart), where T_1 and T_2 represent flat molecular formulae , and C', C'' represent clauses. Suppose that T_1 is unifiable into T_2 with an $mgu_{\sqsubseteq}$ θ. The resolution rule is as follows:

$$\text{from } C_1 \text{ and } C_2, \text{ infer } \theta(C' \vee C'')$$

It is worth noting that when the literals involved are non atomic formulae, the resolution process may be asymmetric because the $mgu_{\sqsubseteq}$ $\theta = mgu_{\sqsubseteq}(T_1 \text{ into } T_2)$ could be different from $mgu_{\sqsubseteq}(T_2 \text{ into } T_1)$. Moreover, the latter $mgu_{\sqsubseteq}$ might not exist.

2. Factoring Rule

Unlike the factoring rule in classical logic, we need different rules for positive and negative literals, because of the way $mgu_{\sqsubseteq}$ is defined. For the case of positive literals: consider a clause of the form $C = T_1 \vee T_2 \vee C'$, where T_1 and T_2 denote *positive* literals, and C' is a clause. Suppose that T_1 is unifiable into T_2 with an $mgu_{\sqsubseteq}$ θ. The factoring rule is as follows:

$$\text{from } C, \quad \text{infer } \theta(T_1 \vee C')$$

For the case of negative literals: let $C = \neg T_1 \vee \neg T_2 \vee C'$, and let T_1 be unifiable into T_2 with an $mgu_{\sqsubseteq}$ θ, the factoring rule is as follows:

$$\text{from } C, \quad \text{infer } \theta(\neg T_2 \vee C')$$

The clauses inferred by one of the factoring rules are called *factors* of C.

3. Paramodulation Rule

As in F-logic [KifLaWu90], this rule is used to capture the equality relation that may arise among id-terms.

Notation. Let E represent a clause or a literal, and let $E[Id_1]$ represent an expression that contains an id-term Id_1. If one occurrence of the id-term Id_1 is replaced by an id-term Id_2, the result is denoted by $E[Id_2]$.

Let $C_1 = L[Id_1] \vee C''$ and $C_2 = (Id_2 \doteq Id_3) \vee C'''$ be a pair of clauses, with no variables in common, where Id_1, Id_2, and Id_3 represent id-terms. C'' and C''' represent clauses, and $L[Id_1]$ represents a literal containing Id_1. If the id-terms Id_1 and Id_2 are unified with an mgu θ, the paramodulation rule says:

$$\text{from } C_1 \text{ and } C_2, \text{ infer } \theta(L[Id_3] \vee C'' \vee C''')$$

4. Isa-Reflexivity Rule

This rule is intended to capture the semantic-entailment property of isa reflexivity on class objects.

- Given a clause $C = A([isa \rightarrow B])$, where A and B represent id-terms, for each $X = A,\ B$ infer

$$X([isa \rightarrow X])$$

- Given a clause $C = Z([instOf \rightarrow A])$, where Z and A represent id-terms, we infer

$$A([isa \rightarrow A])$$

- Given a clause $C = A([FunM(B_1, \ldots, B_n) \Rightarrow \{Y\}])$, where $FunM,\ A,\ B_1, \ldots, B_n,\ Y$ represent id-terms, for each $X = A,\ B_1, \ldots, B_n,\ Y$ infer

$$X([isa \rightarrow X])$$

- Given a clause $C = A([SetM(B_1, \ldots, B_n) \Rightarrow\!\!\!\Rightarrow \{Y\}])$, where $SetM,\ A,\ B_1, \ldots, B_n,\ Y$ represent id-terms, for each $X = A,\ B_1, \ldots, B_n,\ Y$ infer

$$X([isa \rightarrow X])$$

5. Isa-Antisymmetric Rule

This rule is intended to capture the semantic-entailment property of isa-antisymmetry.

Let $C_1 = A([isa \rightarrow B]) \lor C'$ and $C_2 = B'([isa \rightarrow A']) \lor C''$, be clauses with no variables n common, where C' and C'' represent clauses, and A, B, A', B' represent id-terms. Suppose that θ is an mgu of tuples of id-terms $\langle A, B\rangle$ and $\langle A', B'\rangle$. Then, the anti-symmetric rule says:

$$\text{from } C_1 \text{ and } C_2 \text{ infer } \theta((A \doteq B) \lor C' \lor C'')$$

6. Isa-Transitivity Rule

This rule is intended to capture the semantic entailment property of isa-transitivity.

Let $C_1 = A([isa \rightarrow B]) \;\vee\; C'$ and $C_2 = B'([isa \rightarrow D] \;\vee\; C''$ be clauses with no variables in common, where C', C'' represent clauses, and A, B, B', D represent id-terms. Suppose that θ is an mgu of B and B'. The isa-transitivity rule says that:

$$\text{from } C_1 \text{ and } C_2, \text{ infer } \theta(A([isa \rightarrow D]) \vee C' \;\vee C'')$$

7. Well-Typing Rule

This rule is intended to capture the well-typing conditions.

Let $FunM, A, A_1, \ldots, A_n, P, FunM', CA, CA_1, \ldots, CA_n, Q, A'$, and CA' be symbols denoting id-terms, and let C, C' and C'' be symbols denoting clauses.

Let

$$\begin{aligned} &C_1 = A([FunM(A_1, \ldots, A_n) \rightarrow P]) \vee C, \\ &C_2 = CA([FunM'(CA_1, \ldots, CA_n) \Rightarrow \{Q\}]) \vee C' \text{ and} \\ &C_3 = A'([instOf \rightarrow CA']) \vee C'' \end{aligned}$$

be clauses with no variables in common, such that the two tuples $\langle FunM, A, CA\rangle$ and $\langle FunM', A', CA'\rangle$ are unifiable with mgu θ, then from C_1, C_2, and C_3 infer:

$$\begin{aligned} &\theta(\; P([instOf \rightarrow Q]) \;\vee C \vee C' \;\vee C'') \text{ and} \\ &\theta(\; A_i([instOf \rightarrow CA_i]) \;\vee C \vee C' \;\vee C''), \text{ for each } i = 1, \ldots, n \end{aligned}$$

Informally the rule says that, whenever the signature of a method is defined, then the arguments and the value of a method should be from appropriate types.

For set valued methods, the rule is defined similarly, except several obvious changes such as C_1 and C_2 are replaced by

$$\begin{aligned} &C_1 = A([SetM(A_1, \ldots, A_n) \twoheadrightarrow \{P\}]) \vee C \text{ and} \\ &C_2 = CA([SetM'(CA_1, \ldots, CA_n) \Rrightarrow \{Q\}]) \vee C''. \end{aligned}$$

8. Type-Inheritance Rule

This rule is intended to capture the semantic-entailment property of type inheritance.

Let $C_1 = A([FunM(A_1, \ldots, A_n) \Rightarrow \{T\}]) \vee C'$ and $C_2 = B([isa \rightarrow A']) \vee C''$ be clauses with no variables in common, where $FunM, A, A_1, \ldots, A_n, T, B, A'$ represent id-terms and C', C'' represent clauses. This rule also applies when the output $\{T\}$ is an empty set $\{\}$. Suppose that A and A' are unified with an mgu θ. The rule says:

$$\text{from } C_1 \text{ and } C_2, \text{ infer } \theta(B([FunM(A_1, \ldots, A_n) \Rightarrow \{T\}]) \vee C' \vee C'')$$

Similarly, for set valued methods, aside from the obvious change of replacing the symbol $\Rightarrow$ with $\Rightarrow\!\!\!\Rightarrow$.

9. Range-Supertyping Rule

This rule is intended to capture the semantic-entailment property of range-supertyping of method signatures.

Let $C_1 = A([FunM(A_1, \ldots, A_n) \Rightarrow \{P\}]) \vee C'$ and $C_2 = P'([isa \rightarrow Q]) \vee C''$ be clauses with no variables in common, where $FunM, A; A_1, \ldots, A_n, P, P'Q$ represent id-terms and C', C'' represent clauses. Suppose that P and P' are unified with an mgu θ. The rule says:

$$\text{from } C_1 \text{ and } C_2, \text{ infer } \theta(A([FunM(A_1, \ldots, A_n) \Rightarrow \{Q\}]) \vee C' \vee C'')$$

10. InstOf-Isa Transitivity rule

This rule is intended to capture the semantic-entailment property of instance-of-isa transitivity.

Let $C_1 = A([instOf \rightarrow P]) \vee C'$ and $C_2 = P'([isa \rightarrow Q]) \vee C''$ be clauses with no variables in common, where A, P, P', Q represent id-terms and C', C'' represent clauses. Suppose that θ is an mgu of P and P'. The instOf-isa transitivity rule says:

from C_1 and C_2, infer $\theta(A([instOf \rightarrow Q]) \vee C' \vee C'')$

11. Functionality Rule

This rule is intended to capture the semantic-entailment property of the functionality of a single-valued method.

Let

$$C_1 = A([FunM(A_1, \ldots, A_n) \rightarrow B]) \vee C' \text{ and}$$
$$C_2 = A'([FunM'(A'_1, \ldots, A'_n) \rightarrow B']) \vee C''$$

be clauses with no variables in common, where the symbols $FunM, A, A_1, \ldots, A_n$, $B, FunM', A', A'_1, \ldots, A'_1, B$ represent id-terms and C', C'' represent clauses.

Suppose that the tuple of id-terms $\langle A, FunM, A_1, \ldots, A_n \rangle$ is unified to the tuple of id-terms $\langle A', FunM', A'_1, \ldots, A'_n \rangle$ with an mgu θ. The rule says:

from C_1 and C_2, infer $\theta(B \doteq B' \vee C' \vee C'')$

12. Merging Rule

This rule is intended to combine information contained in different data or class terms, based on the property that a molecular formula can be decomposed into its constituting atomic formulae.

Definition Let T_1 and T_2 be a pair of OOL terms with the same object identity. A term T is called *a merge of T_1 and T_2* if the set of atoms in T is equal to the union of the sets of atoms in T_1 and T_2. ∎

For example, consider the following pair of data terms that has a common set-valued method $SetM$:

$D([FunM_1 \rightarrow q], [SetM(x) \twoheadrightarrow \{r\}])$, and

$D([FunM_2 \rightarrow s], [SetM(x) \twoheadrightarrow \{t\}])$

This pair can be merged in more than one way. For example, the two terms can be merged into one of the following:

$D([FunM_1 \rightarrow q], [FunM_2 \rightarrow s], [SetM(x) \twoheadrightarrow \{r\}], [SetM(x) \twoheadrightarrow \{t\}])$, or

$D([FunM_1 \rightarrow q], [FunM_2 \rightarrow s], [SetM(x) \twoheadrightarrow \{r, t\}])$

Hence we have more than one possible merge term. Following the corresponding F-logic's definition [KifLaWu90], we can distinguish a certain kind of merge (called *canonical*) that has a uniqueness property.

Definition A *canonical merge* of T_1 and T_2 (denoted by $merge(T_1, T_2)$) is a merge that does not contain repeated identical invocations of set-valued methods, and the signatures of set-valued and single-valued methods (recall that the output of a single-valued method's signature is also a type set). ■

This canonical merge is unique up to a permutation of the elements in the ranges of set-valued methods, or the ranges of method signatures. Notice that in the above example the second merge is canonical.

Consider a pair of clauses $C_1 = T \vee C'$ and $C_2 = T' \vee C''$ with no common variables, where both T and T' denote data or class terms, and C', C'' denote clauses. Suppose that there is mgu θ unifying the object-identity parts of T and T', and R denotes the canonical merge of $\theta(T)$ and $\theta(T')$. The merging rule says:

$$\text{from } C_1 \text{ and } C_2, \text{ infer } R \vee \ \theta(C' \vee \ C'')$$

3.10.4 Soundness

Let S be a set of clauses, and R be a clause. A *deduction* of R from S (denoted by $S \vdash R$) is a finite sequence of clauses $C_1, \ldots C_n$ such that $C_n = R$, and for each $i = 1, \ldots, n$, C_i is one of the following:

1. an element of S;
2. an inferred clause from C_k, or from C_k and C_l, or from C_k, C_l, and C_m, where k, l, $m < i$, by using one of the inference rules. Note that the only inference rule that needs three previous clauses is the well-typing inference rule.

If C_n is an empty clause, denoted by $\Box$, the deduction is called a *refutation.*

Theorem 3.10.4.1 (Soundness) *If a set of clauses S deduces a clause R ($S \vdash R$), then S logically entails R ($S \models R$)*

Proof:

A proof for the soundness of the resolution rule:
Given a language $\mathcal{L}$ of OOL, let I be any interpretation. We need to show that if:

- $I \models \neg T_1 \vee C'$,
- $I \models T_2 \vee C''$, and
- θ is an $mgu_{\sqsubseteq}$ of T_1 into T_2,

then $I \models \theta(C' \vee C'')$.

For any given I, we must have either $I \models \theta(T_1)$ or $I \not\models \theta(T_1)$, thus we have either one of the following:

1. $I \models \theta(T_1)$.

   Since $I \models \neg\theta(T_1) \vee \theta(C')$, and $I \models \theta(T_1)$, it follows that $I \models \theta(C')$.

2. $I \not\models \theta(T_1)$.

   By the definition of the satisfaction of a molecular formula, from $I \not\models \theta(T_1)$ and $\theta(T_1) \sqsubseteq \theta(T_2)$, we conclude $I \not\models \theta(T_2)$. Sinc $I \models \theta(T_2) \vee \theta(C'')$ and $I \not\models \theta(T_2)$, it follows that $I \models \theta(C'')$.

Collecting the two results above, for any interpretation I, and given preconditions as above, we have either $I \models \theta(C')$ or $I \models \theta(C'')$. By the usual definition of logical connective $\vee$, it follows that $I \models \theta(C \vee C')$.

A proof for the soundness of the factoring rule:

1. Given a language $\mathcal{L}$ of OOL, let I be any interpretation. We need to show that if $I \models T_1 \vee T_2 \vee C'$ and θ is an $mgu_{\sqsubseteq}$ of T_1 into T_2, then $I \models \theta(T_1 \vee C')$.

   By the usual semantics of $\vee$, the proof is equivalent to showing that whenever $I \models T_1 \vee T_2$ and θ is an $mgu_{\sqsubseteq}$ of T_1 into T_2, then $I \models \theta(T_1)$.

   After the substitution θ, $I \models \theta(T_1 \vee T_2)$ means that $I \models \theta(T_1)$ or $I \models \theta(T_2)$. Since $\theta(T_1) \sqsubseteq \theta(T_2)$, whenever $I \models \theta(T_2)$, we also have $I \models \theta(T_1)$. Thus whenever $I \models \theta(T_1 \vee T_2)$, we have $I \models \theta(T_1)$.

2. Given a language $\mathcal{L}$ of OOL, let I be any interpretation. We need to show that if $I \models \neg T_1 \vee \neg T_2 \vee C'$ and θ is an $mgu_{\sqsubseteq}$ of T_1 into T_2, then $I \models \theta(\neg T_2 \vee C')$.

   By the usual semantics of $\vee$, the proof is equivalent to showing that whenever $I \models \neg T_1 \vee \neg T_2$ and θ is an $mgu_{\sqsubseteq}$ of T_1 into T_2, then $I \models \theta(\neg T_2)$.

   After the substitution θ, $I \models \theta(\neg T_1 \vee \neg T_2)$ means that $I \models \neg\theta(T_1)$ or $I \models \neg\theta(T_2)$. Since $\theta(T_1) \sqsubseteq \theta(T_2)$, whenever $I \models \neg\theta(T_1)$, we also have $I \models \neg\theta(T_2)$. Consequently, if $I \models \theta(\neg T_1 \vee \neg T_2)$, then $I \models \neg\theta(T_2)$.

The proofs for the rest of the inference rules follow straight-forwardly from the forms of the inference rules and the semantic-entailment properties presented in Section 3.6. ∎

3.10.5 Completeness

The proof for completeness is carried out in two steps. First, we prove the completeness of the proof theory on ground clauses by using Herbrand's theorem. Then, we use a lifting lemma, similar to the one in classical logic, to show that the completeness on ground clauses is also valid for the non-ground case. The proof is the result of adapting the completeness proof of F-logic [KifLaWu90], with some changes and additions to suit OOL.

As introduced earlier, the notation $\Box$ will be used to denote an empty clause.

Lemma 3.10.5.1 (Unsatisfiability of a Set of Ground Literals) *Let $\boldsymbol{G}$ be a set of ground literals. If $\boldsymbol{G}$ is unsatisfiable then there are ground molecules A and B such that $A \sqsubseteq B$, $\neg A \in \boldsymbol{G}$ and there is a deduction of B from $\boldsymbol{G}$ ($\boldsymbol{G} \vdash B$). Note that if A and B are ground-predicate-atomic formulae, $A \sqsubseteq B$ means that A is identical to B.*

Proof: Recall that $A \sqsubseteq B$ means that A is a subterm or equal to B (i.e., every atomic formula constituting A is also an atomic formula constituting B). We will prove this lemma by a contradiction. Assume that there are no molecules A and B, such that $A \sqsubseteq B$, $\neg A \in \boldsymbol{G}$, and $\boldsymbol{G} \vdash B$. We will show that as a consequence of this assumption $\boldsymbol{G}$ is satisfiable. Consider the set of molecules (positive literals) that is defined as follows:

$$D = \{\alpha \,|\, \alpha \text{ is a subterm of some molecule } \beta \text{ deducible from } \boldsymbol{G} \text{ (i.e., } \boldsymbol{G} \vdash \beta)\}$$

We have shown before (in Section 3.8) how to construct a semantic structure $I_{\mathcal{H}}$ from any Herbrand interpretation $\mathcal{H}$, such that $I_{\mathcal{H}} \models S$ if $\mathcal{H} \models S$, where S is a set of clauses. By applying the same construction to D, we obtain a semantic structure $I_D = \langle U, I_{id}, \prec, I_{inst}, I_p, I_{\rightarrow}, I_{\rightarrow\!\!\!\rightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow} \rangle$. The constructed I_D is indeed a semantic structure because of the following:

- D is closed under deduction "$\vdash$," by the definition of D.
- Each component of the semantic structure I_D has the property prescribed by the semantic requirement in Section 3.4 that can be shown as follows:

  1. The domain of interpretation U is the quotient of the set of all ground id-terms of D (denoted by F^*, where F is the set of all functions symbols used in D) by the equality predicate-atomic formulae in D, i.e., $F^*/ \doteq$. The equivalence class of an id-term t is a ted by $[t]$.
  2. – $I_{id}(c)$ · · $[c]$, for each 0-ary function (constant) symbol $c \in F$.

     – $I_{id}(f)([t_1], \ldots, [t_n]) = [\, f(t_1, \ldots, t_n)\,]$, for every n-ary ($n \geq 1$) function symbol $f \in F$.
  3. The ordering of class objects in U is defined by the *isa* assertions in D: for all class objects $[a], [b] \in U$, $[a] \preceq [b]$ iff $a([isa \rightarrow b]) \in D$. The partial-order relation, $\preceq$, among classes is captured by the closure of D under the following rules: isa-reflexivity, isa-antisymmetry, and isa-transitivity.
  4. The instance-of relation in U is defined by the *instOf* assertions in D: for all objects $[c], [d] \in U$, $\langle [d], [c] \rangle \in I_{inst}(instOf)$ iff $d([instOf \rightarrow c]) \in D$. The instOf-isa transitivity property is captured by the closure of D by the instOf-isa transitivity rule.
  5. $I^n_{\rightarrow}([funM])([obj], [t_1], \ldots, [t_n]) =$

$$\begin{cases} [v] \text{ if } obj([funM(t_1,\ldots,t_n) \rightarrow v]) \in D \\ \text{otherwise: undefined.} \end{cases}$$

The functionality requirement of $I_{\rightarrow}$ is captured by the closure of D under the functionality rule.

6. $I^{n}_{\rightarrow\!\!\!\rightarrow} ([setM])([obj],[t_1],\ldots,[t_n]) =$

$$\begin{cases} \{\ [v] \mid obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{v\}]) \in D\ \} \text{ if} \\ \quad obj([setM(t_1,\ldots,t_n) \rightarrow\!\!\!\rightarrow \{\}]) \in D \\ \text{otherwise: undefined.} \end{cases}$$

7. $I^{n}_{\Rightarrow} ([funM])([obj],[t_1],\ldots,[t_n]) =$

$$\begin{cases} \{\ [v] \mid obj([funM(t_1,\ldots,t_n) \Rightarrow \{v\}]) \in D\ \} \text{ if} \\ \quad obj([funM(t_1,\ldots,t_n) \Rightarrow \{\}]) \in D \\ \text{otherwise: undefined.} \end{cases}$$

The antimonotonic and upward-closedness properties are captured by the closure of D under the type-inheritance and range-supertyping rules.

8. $I^{n}_{\Rightarrow\!\!\!\Rightarrow} ([setM])([obj,[t_1],\ldots,[t_n]) =$

$$\begin{cases} \{\ [v] \mid obj([setM(t_1,\ldots,t_n) \Rightarrow\!\!\!\Rightarrow \{v\}]) \in D\ \} \text{ if} \\ \quad obj([setM(t_1,\ldots,t_n) \Rightarrow\!\!\!\Rightarrow \{\}]) \in D \\ \text{otherwise: undefined.} \end{cases}$$

Similar to the case of $I_{\Rightarrow}$, the antimonotonic and upward-closedness properties are captured by the closure of D under the type-inheritance and range-supertyping rules.

9. $I_{\mathcal{P}}(p) = \{\ \langle [t_1],\ldots,[t_n] \rangle \mid p(t_1,\ldots t_n) \in D\ \}$.

- The well-typing conditions are captured by the well-typing rules.

Thus I_D is indeed a semantic structure since it fulfills the properties prescribed in Section 3.4.

Now we need to show that for each molecule A (i.e., a positive literal):

$$I_D \models A \text{ iff } A \in D \tag{3.1}$$

The proof for the *if* part follows directly from the soundness of the inference rules. To show the *only if* part, we need to consider the following three cases of A:

1. A is a predicate-atomic formula: from the construction of I_p, it follows directly that if $I_D \models A$ then $A \in D$.

2. A is a data term consisting of a[illegible]s $\alpha_1, \ldots, \alpha_n$: $I_D \models A$ iff $I_D \models \alpha_i$, for each $i = 1, \ldots, n$. From the construction of I_{inst}, $I_{\rightarrow}$ and $I_{\rightarrow\!\!\!\rightarrow}$ we conclude that $I_D \models \alpha_i$ iff $\alpha_i \in D$. Hence, $I_D \models A$ iff $I_D \models \alpha_i$ iff $\alpha_i \in D$ $(i = 1, \ldots, n)$. By the definition of D, there are terms $\beta_1, \ldots, \beta_n$ deducible from $\boldsymbol{G}$ such that α_i is a subterm of β_i for each $i = 1, \ldots, n$.

   By using the merging rule, D contains β which is a canonical merge of $\beta_1, \ldots, \beta_n$. It follows that A is a subterm of β, because each atomic formula constituting A is also an atomic formula constituting β. By the definition of D, we conclude that $A \in D$.

3. A is a class term consisting of atoms $\alpha_1, \ldots, \alpha_n$: $I_D \models A$ iff $I_D \models \alpha_i$ $(i = 1, \ldots, n)$. From the construction of $\preceq$, $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$ we conclude that $I_D \models \alpha_i$ iff $\alpha_i \in D$. Hence, $I_D \models A$ iff $I_D \models \alpha_i$ iff $\alpha_i \in D$ $(i = 1, \ldots, n)$. Then we show that $A \in D$ in a similar way to the case of a data term above.

From the above three cases of A we conclude that if $I_D \models A$ then $A \in D$. Hence (3.1) is proved.

Let's consider the two possible cases of each literal element G:

1. For each positive literal $A \in G$, $A \in D$ (by the definition of D). Hence, $I_D \models A$ by (3.1).

2. Based on our assumption in the beginning of this proof that no $\neg A \in G$ such that $A \sqsubseteq B$ and $G \vdash B$, each negative literal $\neg A \in G$, A is not a subterm of any molecule deducible from G. Hence, $A \notin D$. From $A \notin D$ and (3.1), it follows that $I_D \not\models A$, this means that $I_D \models \neg A$.

From the two consideration above, I_D satisfies every literal in G, i.e., $I_D \models G$ (G is satisfiable). A contradiction! ∎

Theorem 3.10.5.1 (Completeness of Ground Deduction) *If a set of ground clauses G is unsatisfiable then there exists a refutation from G.*

Proof: Based on the Herbrand theorem (Theorem 3.9.1), we assume that G is finite and unsatisfiable. We will show that there is a refutation from G. The following proof is adapted from [KifLaWu90]. The proof is carried out by using a technique used in [Ande70]. The technique is executed by utilizing (strong) induction on the parameter *excess*(G), the number of excess literals in G. The parameter *excess*(G) is defined as follows:

$$\text{excess}(G) = (\textit{the number of occurrences of literals in } G) - (\textit{the number of clauses in } G)$$

Basis: excess(G) $= 0$. In this case, the number of literals in G is equal to the number of clauses in G. As a result, there are two possible situations: First, the empty clause $\square$ is in G, this means there is a refutation already; second, each clause in G is a literal. In the second situation, based on the previous unsatisfiability property on a set of ground literals (Lemma 3.10.5.1), we will have $G \vdash \neg A$ and $G \vdash B$ for some molecules A and B such that $A \sqsubseteq B$. If we apply the resolution rule to $\neg A$ and B, we will have the empty clause $\Box$.

Induction hypothesis: There exists a refutation for excess(G) $= m$, for all values $m < n$, where $m \geq 0$.

Induction Step: excess(G) $= n$, where $n > 0$. Under this condition, there exists a clause in G that contains more than one literal. Assume that C is such a clause. Separating this clause from the others, we have $G = \{C\} \cup G'$. Based on the assumption on C, $C = L \vee C'$, where L is a literal and C' is a clause. Because C is assumed to contain more than one literal, the clause C' is not empty. Based on the usual distributive law, $\{L \vee C'\} \cup G'$ is unsatisfiable if and only if so are $G_1 = \{L\} \cup G'$ and $G_2 = \{C'\} \cup G'$. Based on the induction hypothesis, there are separate refutations of G_1 and G_2, because $excess(G_1) \leq (n-1)$ and $excess(G_2) = (n-1)$. As a consequence, $G_1 \vdash \Box$ and $G_2 \vdash \Box$. If we apply to G the deduction sequence that deducts $\Box$ from G_2 ($G_2 = \{C'\} \cup G'$), we will deduct either L or $\Box$. If $\Box$ is deducted, then there is a refutation from G already. Otherwise (i.e., when $G \vdash L$), since $G' \subset G$, if we apply to $G \cup \{L\}$ the same deduction sequence that derive $\Box$ from G_1 ($G_1 = \{L\} \cup G'$), the empty clause $\Box$ will be derived. So, by combining the two deduction sequences, a refutation from G is obtained. ■

Proposition 3.10.1 *Given a pair of OOL terms (molecular formulae) M_1 and M_2, there exists a unification algorithm that compute a complete set of $mgu_{\sqsubseteq}$'s of M_1 into M_2.*

Proof: The algorithm and the proof of its correctness has been given in Section 3.10.2.1. ■

Lemma 3.10.5.2 (Lifting) *Suppose that C_1, C_2 (or C_1, C_2, C_3 if we use the well-typing inference rule) are clauses and C_1', C_2' (respectively, C_1', C_2', C_3') are their ground instances respectively. If R' is inferred from C_1' and C_2' (respectively from C_1', C_2', C_3') using one of the inference rules, then there exists a clause R such that R can be inferred from the factors of C_1 and C_2 (respectively, C_1, C_2, C_3) using the same inference rule, and R' is an instance of R.*

Proof: The proof is carried out by considering every inference rule separately. The proof for each type of inference rule is carried out in a similar way to that in Predicate Calculus, because the notion of substitution is similar. There are similarities in the way the proof is carried out for each of the inference rules. The following is an example of the proof for the resolution rule.

A proof of the Lifting lemma for the resolution rule:
Let $C_1 = \neg T_1 \vee Z$, $C_2 = T_2 \vee Z'$, and let their ground instances be $C_1' = \alpha(C_1) = \neg\alpha(T_1) \vee \alpha(Z)$, $C_2' = \beta(C_2) = \beta(T_2) \vee \beta(Z')$, respectively. Suppose that $R' = \alpha(Z) \vee \beta(Z')$ is the deduced clause (also called the resolvent) of C_1' and C_2'. This means that $\alpha(T_1)$ is unified into $\beta(T_2)$.

Let $\alpha = \{u_1/s_1, \ldots, u_k/s_k\}$ and $\beta = \{v_1/w_1, \ldots, v_m/w_m\}$, where $u_1, \ldots, u_k$ are variables in C_1 and $v_1, \ldots, v_m$ are variables in C_2.

Let $\varphi = \{v_1/u_{k+1}, \ldots, v_m/u_{k+m}\}$ (to be applied on C_2 only) be the changes of variables that standardized C_1 and C_2 apart. Then, let

$$\sigma = \{u_1/s_1, \ldots, u_k/s_k, u_{k+1}/w_1, \ldots, u_{k+m}/w_m\}.$$

Thus, $\sigma(C_1) = \alpha(C_1)$ and $\sigma \circ \varphi(C_2) = \beta(C_2)$. This means that $R' = \sigma(Z) \vee \sigma \circ \varphi(Z') = \alpha(Z) \vee \beta(Z')$, $\sigma(T_1) = \alpha(T_1)$, and $\sigma \circ \varphi(T_2) = \beta(T_2)$.

Since σ unifies T_1 into $\varphi(T_2)$, there is an $\text{mgu}_{\sqsubseteq}$ θ of T_1 into $\varphi(T_2)$ (chosen by, such as, the previously given unification algorithm), where, if $\theta \neq \sigma$, there is a substitution

γ such that $\sigma = \gamma \circ \theta$. Assume that R is deduced as the result of applying the resolution rule on C_1 and $\varphi(C_2)$ with the substitution θ as the mgu$_{\sqsubseteq}$. Since there is a substitution γ such that $\sigma = \gamma \circ \theta$, $R' = \sigma(Z \vee \varphi(Z')) = \gamma \circ \theta(Z \vee \varphi(Z')) = \gamma(R)$, (i.e., R' is an instance of R). ■

Theorem 3.10.5.2 (Completeness of OOL Deduction) *If a set* $\boldsymbol{P}$ *of clauses is* unsatisfiable, *then there is a refutation from* $\boldsymbol{P}$.

Proof: Suppose that $\boldsymbol{G}$ is the set of ground instances of $\boldsymbol{P}$. Based on the completeness theorem on the ground case (Theorem 3.10.5.1), there is a refutation of $\boldsymbol{G}$. By using the lifting lemma (Lemma 3.10.5.2), the ground refutation can be lifted to a refutation from $\boldsymbol{P}$. ■

CHAPTER 4

OOL as a Programming Language

In this chapter we outline an object-oriented-declarative language based on OOL. We start with the definitions of a database, a query, and an untyped-semantic structure. After that, we describe the components of an OOL program, the monotonic inheritance of signatures, an extension of OOL to include *class methods*, an extension of OOL to include *shared methods* with its non-monotonic inheritance, an example of an OOL program, and finally a discussion. We consider that F-logic [KifLaWu90] would have difficulties with the two extensions just mentioned.

4.1 Databases and Queries

Definition A *database* or a program is a finite set of clauses. ■

Definition A query is a statement of the form: :-Q, where Q is a molecular formula.■

Given a program P and a query :-Q, the set of answers with respect to the program P is the smallest set of molecular formulae such that

1. it contains all instances q of Q that are logically entailed by P (*i.e.*, $P \models q$);
2. the set of answers is closed under logical entailment $\models$.

The second condition is adapted from [KifLaWu90]. This condition is required to overcome problems due to the fact that a variable cannot be substituted by a set of id-terms (a variable cannot denote a set). Consider the following example:

Suppose that the database contains

$$joe([children \twoheadrightarrow \{jack, jim, john\}])$$

and the query is

$$:\text{-}\ joe([children \twoheadrightarrow \{X\}])$$

Based on the first condition, there are three instances of this query:

1. $joe([children \twoheadrightarrow \{jack\}])$,
2. $joe([children \twoheadrightarrow \{jim\}])$, and
3. $joe([children \twoheadrightarrow \{john\}])$.

From these three answers, we can conclude that the molecular formula (i.e., a data term):

$$joe([children \twoheadrightarrow \{jack, jim, john\}])$$

is their logical consequence. However, if we have only the first condition, this molecular formula cannot be considered as an answer to the query because a variable cannot be bound to a set. The second condition of being closed under logical entailment overcomes this problem.

4.2 Untyped-Semantic Structure

F-logic [KifLaWu90] can distinguish between well-typed and ill-typed logic programs by introducing untyped-semantic structures and untyped models. We will use this concept in OOL. By introducing the concept of untyped-semantic structures and untyped models, together with the well-typing conditions of OOL, we will ***be able to distinguish inconsistency due to type errors and inconsistency arising from other causes***, this feature would be helpful to OOL users. Based on F-logic's untyped-semantic structure concept [KifLaWu90], we define untyped-semantic structures for OOL below.

Definition Given an OOL language $\mathcal{L}$, its ***untyped-semantic structure:***

$$I = \langle U,\ I_{id},\ \prec,\ I_{inst},\ I_{\rightarrow},\ I_{\Rightarrow}, I_{\twoheadrightarrow},\ I_{\Rrightarrow} \rangle$$

is defined similarly to the kernel (typed) semantic structure of an OOL language defined in Section 3.4, except that it is not required to satisfy the well-typing conditions.■

Definition A satisfaction of a formula α by an untyped-semantic structure is defined in the same way as the one defined in Section 3.4. We also use the same notation $I \models_{\nu} \alpha$ (or $I \models \alpha$ if α is a closed formula). ■

Definition An *untyped model* of a formula α is an untyped-semantic structure I that satisfies α (denoted $I \models_{\nu} \alpha$). ■

The following lemma is derived from the above definitions.

Lemma 4.2.1 *If I is a typed-semantic structure (or respectively, a typed model) then I is an untyped-semantic structure (respectively, an untyped model). On the other hand, if I' is an untyped model that is also a typed-semantic structure then I' is a typed model.*

Proof: The proof follows from the definition of untyped-semantic structures. Because a typed-semantic structure (or respectively, a typed model) is an untyped-semantic structure (respectively, an untyped model) that also satisfies the well-typing conditions, therefore every typed-semantic structure (or typed model) is also an untyped-semantic structure (respectively, an untyped model). On the other hand, an untyped model that is also a typed-semantic structure means that the untyped model also satisfies the well-typing conditions. Thus it is a typed model by definition. ■

It should be noted that, in this thesis, the phrase semantic structure (respectively, model) without any qualifier ("typed" or "untyped") refers to typed-semantic structure (respectively, typed model). This is because the kernel semantic structure of OOL defined in Section 3.4 includes the well-typing condition.

Minimal Models Given a program P consisting of definite Horn clauses (where each clause has exactly one positive literal), we can group models of the program P into typed models and untyped models.

The following definition is adapted from [KifLaWu90].

Definition Formally, We write $M \ll^{hc} N$, where M and N are models if and only if for every molecular formula T, whenever $M \models T$ then $N \models T$. ■

Definition An untyped model M of a program P is *minimal* if for any other untyped model N of P, $N \ll^{hc} M$ implies $M \ll^{hc} N$. ■

Definition A *minimal-typed model* is a minimal-untyped model that also satisfies the well-typing conditions described in Section 3.4.

It follows from the above definition that being a minimal-typed model requires more conditions than simply being a minimal model among the class of typed models. This is because the model must be also minimal among the class of untyped models.

4.3 Program Components

In this section we discuss the conceptual division of an OOL program, based on the functionalities of the program components.

As defined before, a program or a database is a finite set of clauses. Similar to classical logic programming, a clause (also called a deductive rule) is written in the form: $H_1, \ldots H_m$:- $B_1, \ldots B_n$, where $H_1, \ldots, H_m$ are the head literals, and $B_1, \ldots, B_n$ are the body literals. A deductive rule that has head literals but an empty body is called a *simple fact*. By definition, each clausal rule is implicitly universally quantified. In this chapter, we restrict the programs to contain only definite Horn clauses, i.e., no negated literals in each rule's body and there is exactly one positive literal in each rule's head. The reason for this restriction is discussed in the next section.

A program specifies what each method does, each method signature (i.e., the type of each method's argument and the type of its output values), class hierarchies (through subclass assertions), and instance-of assertions.

Because of these different specifications, conceptually, we can divide a program into four components as follows:

1. *Definitions of data:* all rules with data terms containing method definitions, or P-atoms in the heads. This component specifies what every method does and relations among objects (through P-atoms). The output value of a method may be defined explicitly (the value is given) or defined implicitly by a deductive rule.

2. *Subclass-relationship definitions:* all rules with class terms that include the special attribute "*isa*" in the heads. This component arranges classes into class hierarchies (the tree-like hierarchies or the more general directed acyclic graphs)

3. *Instance-of relationship definitions:* all rules with data terms that includes the special attribute "*instOf*" in the heads. This component specifies instances of each class.

4. *Signature (type) definitions:* all rules with class terms indicating signatures of methods in the heads. This component specifies the type of each method's argument and the type of its output values.

Generally, the division of deductive rules into the above components is not disjoint. We may have rules with class terms in the head that can be grouped to the second and the fourth groups, because they contain *isa* attributes and method signatures. We can make the two components disjoint if we separate every class term such that every class term containing class-hierarchy information (through the *isa* attribute) will not contain signature information and on the other hand every class term containing signature information will not contain subclass-relationship information. Furthermore, we can make the separation better if we restrict the literals in every clause to be an atomic formula or its negation. However, this will reduce the expressiveness of a literal. In general, we cannot consider one component independent of another. This is because a rule body might contain literals that are defined in other components.

4.4 Subclass-Relationship Definitions

A progr [illegible] pecifies class hierarchies based on class terms containing the special attribute *isa*. To avoid many difficult problems, we restrict the form of clauses to definite-Horn clauses. Under this restriction, the logic rules cannot specify negative information such as $\neg a([isa \rightarrow b])$. Suppose that we do not restrict the program P to definite Horn clauses, then we can specify the negative information explicitly, such as $\neg a([isa \rightarrow b])$. Unfortunately, we may need very many such statements for a relatively small database, while we might not need all of the possible combinatorial relations. To solve this kind of problem, researchers have used various non-monotonic reasoning techniques. For definite-Horn clauses, a standard technique called Reiter's CWA (stands for Closed World Assumption [Reit78]) is widely used. For example, this technique would say that if program $P \not\models a([isa \rightarrow b])$ then non-monotonically derive $\neg a([isa \rightarrow b])$. As usual, for definite Horn clauses, CWA is equivalent to confining the logical entailment to minimal models (there would be a unique minimal model for the case of Herbrand interpretations).

If we allow negative literals in the body, we will need to use an extension of Reiter's CWA. Several extensions have been proposed such as the *perfect model* semantics [Przym88], the *stable-model* semantics [GelLif88], and the *well-founded* semantics [GelRosSch88, Gel89, Przym89]. We might be able to adapt one of these semantics to OOL. It is pointed out in [KifLaWu90] that there is a common feature of the approaches mentioned above, i.e., each of them distinguishes a subset of models called "canonic," and then defines the program semantics based on those models only. For the case of a set of Horn clauses, minimal Herbrand models coincide with the canonical models of the approaches mentioned above.

4.5 Instance-Of Relationship Definitions

These definitions include all rules that contain the special attribute *instOf* in their heads. It could be a simple fact such as:

$$joe([instOf \rightarrow person])$$

or a more complex one defined by a rule. In this subsection, we will discuss some examples of defining instance-of relationships by using rules, as the result of modifying some ideas from [KifLaWu90] to suit our logic.

In the following modified example from [KifLaWu90], we define objects that are instances of unions of two simpler classes, and objects that are instances of intersections of two simpler classes.

$$I([instOf \rightarrow or\langle T_1, T_2\rangle]) \text{ :- } I([instOf \rightarrow T_1])$$
$$I([instOf \rightarrow or\langle T_1, T_2\rangle]) \text{ :- } I([instOf \rightarrow T_2])$$
$$I([instOf \rightarrow and\langle T_1, T_2\rangle]) \text{ :- } I([instOf \rightarrow T_1]) \wedge I([instOf \rightarrow T_2])$$

Similar to F-logic [KifLaWu90], OOL allows definitions of *parametric-polymorphic* classes. These definitions are possible because class objects are represented by id-terms that may contain variables. For example, the following two clauses define a parametric-polymorphic class *list⟨Type⟩*, whose instances depend on the instantiation of its argument variable, *"Type"*.

$$nil([instOf \rightarrow list\langle Type\rangle])$$
$$cons\langle X, Y\rangle([instOf \rightarrow list\langle Type\rangle]) \text{ :- } X([instOf \rightarrow Type) \wedge Y([instOf \rightarrow list\langle Type\rangle])$$

As an example, *list⟨integer⟩* denotes the class of lists of integers. This class may contain elements such as:

$cons\langle 0, nil\rangle$,

$cons\langle 0, cons\langle 1, nil\rangle\rangle$,

$cons\langle 0, cons\langle 1, cons\langle 2, nil\rangle\rangle\rangle$,

$cons\langle 0, cons\langle 1, cons\langle 2, cons\langle 3, nil\rangle\rangle\rangle\rangle$, etc.

OOL also allows definitions of classes whose instances are *populated dynamically.* For example, the following rule defines a set of classes *csStd⟨Year⟩* parameterized by the variable *"Year"*:

$$S([instOf \rightarrow csStd\langle Year\rangle]) \text{:-}$$
$$S([instOf \rightarrow westernStd]) \wedge S([major \rightarrow computerSc]) \wedge S([enrollYear \rightarrow Year])$$

For example, the class $csStd\langle 1993\rangle$ is inhabited by objects representing Western students in 1993 and having a major in Computer Science.

4.6 Signature (Type) Definitions

Signature (or type, also called schema in database terminology) definitions will help users in defining correct usage of objects, debugging and maintaining data integrity. It will let the system detect wrong type data and queries. The purpose of type definitions is to impose type constraints on the arguments of a method as well as the outputs returned by the method. As an example, consider the following molecular formulae:

$student([isa \rightarrow person], [gpa(year) \Rightarrow integer])$

$joe([instOf \rightarrow student], [gpa(1993) \rightarrow 85])$

In this example, *gpa* is a single-valued method that when applied in the context of an instance of class *student* to an instance of class *year* returns a value of type *integer*. This type constraint forces the method *gpa*'s argument to be from type *year* and its returned value in any given *year* to be an *integer*.

4.6.1 The Well-Typing Conditions

The well-typing conditions of OOL apply only to methods whose signatures are defined. Thus, if a method is defined on an instance object and no signature is defined on that method, the well-typing conditions do not apply. Furthermore, that method definition will be regarded as correct with respect to the well-typing conditions. In other words, OOL does not force every method to have a signature. If we want to enforce typing on every method definition we will need an additional well-typing condition such as:

For all $m, obj, a_1, \ldots, a_n \in U$:

- For single-valued methods:
  if $I^n_{\rightarrow}(m)(obj, a_1, \ldots, a_n)$ is defined then $I^n_{\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$ is also defined, where $\langle obj, cobj\rangle \in I_{inst}(instOf)$ and $\langle a_i, ca_i\rangle \in I_{inst}(instOf)$ for each $i = 1, \ldots, n$.

  For set-valued methods:
  if $I^n_{\rightarrow\!\!\!\rightarrow}(m)(obj, a_1, \ldots, a_n)$ is defined then $I^n_{\Rightarrow\!\!\!\Rightarrow}(m)(cobj, ca_1, \ldots, ca_n)$ is also defined, where $\langle obj, cobj\rangle \in I_{inst}(instOf)$ and $\langle a_i, ca_i\rangle \in I_{inst}(instOf)$ for each $i = 1, \ldots, n$.

Unfortunately, if we have an additional condition such as given above, we will have difficulties in designing the inference rules for accommodating that condition. This is because such an additional condition requires a rule to infer the context class where the method signature must be defined. In addition, the rule also must infer the tuple of classes of the method signature's proper arguments. The difficulties increase because we allow an instance object to belong to more than one *immediate class* (this phrase will be defined after this paragraph). If we restrict an instance to belong to only one immediate class (such as in [Kim90]), the task of finding the context class where the method signature must be defined will be easier. However, we consider this restriction too stringent, while the task of finding the right class is still difficult. Thus, having such an additional condition will cause practical problems in inferring the context class where the signature of a method must be defined, and the tuple of classes of the method signature's proper arguments.

Definition A class ι, to which an instance ω belongs, is called an *immediate class* of ω if there is no proper subclass of ι such that ω is its instance. ∎

As an example of the difficulties just described, let a program P contain:

$j([instOf \rightarrow student])$ $gradstd([isa \rightarrow student])$
$j([instOf \rightarrow gradstd])$ $employee([wage \Rightarrow integer])$
$j([instOf \rightarrow employee])$ $student([name \Rightarrow string])$
$j([wage \rightarrow 1000])$

In this example, instance object j belongs to several classes. Based on the data term $j([wage \rightarrow 1000])$ alone, it is difficult to design an efficient inference rule for determining the class of j where the signature for the method *wage* must be defined. Here, instance j belong to classes ***student***, ***gradstd***, and ***employee***; its immediate classes are ***gradstd*** and ***employee***. The class of j where the method *wage* could be defined is either ***student***, or ***gradstd***, or ***employee***. Hence, solely based on the data term $j([wage \rightarrow 1000])$, we will have difficulty in determining on which class the signature of method *wage* must be defined. Obviously, we will not have this difficulty, if we put a restriction such that an instance object is allowed to belong to only one class (as [Kim90] does).

To avoid the kind of difficulty just described, we did not include the additional well-typing condition as described above (F-logic [KifLaWu90] can have that kind of additional well-typing condition because the signature of every method is defined on the same object where the method is defined). Despite the obvious disadvantage of being unable to enforce each method to have a signature, we have some advantageous properties of the well-typing conditions chosen for OOL:

- We are able to define an instance object that belongs to more than one immediate class.
- The logic has built-in type checking for those methods *whose signatures are defined.*
- As a programming language, the logic provides flexibility in the sense that users

are allowed to define some methods without forcing them to define the methods' signatures as well.

- If we want to enforce type checking for every method, it is still possible at the meta level. For example, because, in practice, the number of classes to which an instance belongs is likely to be small, the application system can provide a procedure that will check each method defined on an instance, whether or not its signature is defined in one of the classes where that instance belongs.
- The chosen well-typing conditions allow us to have a quite simple rule to accommodate the conditions, and to have a complete proof procedure.
- We are still able to distinguish errors due to ill-typing and errors arising from other causes, by making use of the untyped-semantic structures defined in Section 4.2.

4.6.2 Enforcing The Well-Typing Conditions

The properties of minimal-typed models of a set of definite Horn clauses can be used to enforce the well-typing conditions.

Definition We call a program P *well-typed* if and only if at least one of its minimal-untyped models is also a minimal-typed model. ■

The well-typing conditions can be viewed as a restriction such that the domain and range of every method defined in a program P satisfies all type constraint defined in P. Then, similar to F-logic [KifLaWu90], the semantics of OOL is defined by characterizing models through the minimal condition. This approach allows us to define type-error conditions by way of model-theoretic semantics.

As defined before, a minimal-typed model of a program P is a minimal-untyped model that also fulfills the well-typing conditions. Hence, the semantics of a program P can be defined with respect to minimal-typed models as follows:

$$P \models \phi \text{ iff for each minimal-typed model } M \text{ of } P,\ M \models \phi$$

The well-typing conditions also provide OOL with the ability to distinguish between inconsistencies due to type errors and inconsistencies arising from other causes, by using the following definition.

Definition A program P has a *type error* if it has a minimal-untyped model but no minimal-typed model. ∎

This definition allows us to develop static type checking.

Let P be a set of definite Horn clauses. The type constraint defined in P would be the set of all ground signatures that are true in a minimal-untyped model of P. Here, *a well-typed program P* means that the domain of definition and the range of every method defined in P satisfy the type constraint defined in P. For example, let program P contain:

$person([name \Rightarrow \{string\}])$
$joe([instOf \rightarrow person])$
$joe([name \rightarrow \text{“}John\text{”}])$

This program has a minimal-untyped model that also satisfies the well-typing conditions. Hence program P is well-typed (provided that "*John*" is an instance of class *string*).

On the other hand, if the program P contains:

$person([name \Rightarrow \{string\}])$
$joe([instOf \rightarrow person])$
$joe([name \rightarrow 5])$

the program P is not well-typed (assuming that output of the method *name* on object *joe*, i.e., 5, is not from class *string*). We can determine that the program P has a typing problem because it has a minimal-untyped model, but it does not have any minimal-typed model. It does not have a minimal-typed model because the output of the method *name* on *joe* is not from class *string* which is a violation of the well-typing conditions.

4.7 Monotonic Inheritance of Signatures

Signatures are inherited monotonically. A class inherits the signatures of its superclass (or superclasses). If a class has more than one immediate superclass, the class is said to have multiple inheritance. Whenever we have a class that has multiple inheritance, this class will have all of its superclasses' signatures *monotonically*. Every class accumulates all of the signatures inherited from its superclasses. Consequently, we need to assume that conflicting multiple-inherited signatures are not allowed, and we are also not allowed to redefine a method signature whose argument classes conflict with the inherited ones. We consider this restriction can be enforced at a practical level. If we allow such conflicts to happen, we will potentially face difficult problems from the point of view of logic programming such as non-monotonic inheritance and inconsistencies.

As an example of multiple inheritance, let a program P contain:

$$\begin{array}{ll}
person([name \Rightarrow \{string\}]) & employee([isa \rightarrow person]) \\
student([courses \Rightarrow \{string\}]) & student([isa \rightarrow person]) \\
employee([wage \Rightarrow \{integer\}]) & assistant([isa \rightarrow student]) \\
 & assistant([isa \rightarrow employee])
\end{array}$$

The class ***assistant*** will monotonically inherit signatures of methods ***name***, ***courses***, and ***wage*** from classes ***person***, ***student***, and ***employee*** respectively; thus, class ***assistant*** has the following signature:

$$assistant([name \Rightarrow \{string\}], [courses \Rightarrow \{string\}], [wage \Rightarrow \{integer\}])$$

The following is an example of redefining a method signature:

$$classA([meth1(cl1, cl2) \Rightarrow \{cl3\}])$$
$$classB([isa \rightarrow classA])$$
$$classB([meth1(cl1', cl2') \Rightarrow \{cl3\}])$$

The redefining of method *meth1* signature on *classB* is not allowed if either $cl1'$ is not a subclass of $cl1$, or $cl2'$ is not a subclass of $cl2$.

4.8 Class Methods

Class methods (also called *class attributes* for 0-ary methods) are methods defined on a class. This kind of method (which is inspired by [Kim90]) is not defined in the kernel of OOL (Section 3.3 and 3.4). Class methods are usually used to capture an aggregate property of the instances of a class. Examples of class methods are as follows:

- method *average-mark* for a class *cs28student*;
- method *maximum-age* for a class *student*;
- method *total-wage* for a class *employee*.

We keep the class methods apart from the kernel of OOL, to give it a greater flexibility for further extensions, and to allow us to regard the simpler properties of the kernel of OOL (Section 3.3 and 3.4) as the core concept of OODB systems.

A class method is only attached to the class where it is defined. *It is not inherited by its subclasses nor inherited by its instances.* Such methods are useful for aggregation operations over groups; one might compute average, total, maximum, or minimum.

We can add class methods to the kernel of OOL. Let's use "$\xrightarrow{c}$" (or "$\overset{c}{\twoheadrightarrow}$" if set-valued) for class methods instead of the usual symbols "$\rightarrow$" (respectively, "$\twoheadrightarrow$") used for the ordinary methods. We also need additional symbols "$\overset{c}{\Rightarrow}$" (respectively, "$\overset{c}{\Rightarrow\!\!\!\!\Rightarrow}$ ") for defining the signatures of single-valued class methods (respectively, set-valued class methods). Note that these signatures are defined on the same classes where the corresponding class methods are defined.

To extend the kernel semantics, we need additional interpretation functions $I_{\xrightarrow{c}}$, $I_{\overset{c}{\twoheadrightarrow}}$, $I_{\overset{c}{\Rightarrow}}$, $I_{\overset{c}{\Rightarrow\!\!\!\!\Rightarrow}}$ that are defined similarly to $I_{\rightarrow}$, $I_{\twoheadrightarrow}$, $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\!\Rightarrow}$ respectively, except that

- the methods are only defined on classes,
- the signatures are defined on the same classes where the corresponding methods are defined, and
- the signatures are *not* inherited by subclasses.

The well-typing conditions of the kernel of OOL also need to be extended to capture the well typing of class methods, as follows:

For all $m, cobj, a_1, \ldots, a_n \in U$:

1. For single-valued class methods:

if $I^n_{\overset{c}{\to}}(m)(cobj, a_1, \ldots, a_n)$ and $I^n_{\overset{c}{\Rightarrow}}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, then we have:

$$\langle a_i, ca_i \rangle \in I_{inst}(instOf) \text{ for each } i = 1, \ldots, n$$

For set-valued class methods:

if $I^n_{\overset{c}{\twoheadrightarrow}}(m)(cobj, a_1, \ldots, a_n)$ and $I^n_{\overset{c}{\Rightarrow\!\!\Rightarrow}}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, then we have:

$$\langle a_i, ca_i \rangle \in I_{inst}(instOf) \text{ for each } i = 1, \ldots, n$$

2. For single-valued class methods:

   if for some $p \in U$, $p = I^n_{\overset{c}{\to}}(m)(cobj, a_1, \ldots, a_n)$ and $I^n_{\overset{c}{\Rightarrow}}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, then, *for every* $q \in I^n_{\overset{c}{\Rightarrow}}(m)(cobj, ca_1, \ldots, ca_n)$ we have:

$$\langle p, q \rangle \in I_{inst}(instOf)$$

   For set-valued class methods:

   if for some $p \in U$, $p \in I^n_{\overset{c}{\twoheadrightarrow}}(m)(cobj, a_1, \ldots, a_n)$ and $I^n_{\overset{c}{\Rightarrow\!\!\Rightarrow}}(m)(cobj, ca_1, \ldots, ca_n)$ are defined, then *for every* $q \in I^n_{\overset{c}{\Rightarrow\!\!\Rightarrow}}(m)(cobj, ca_1, \ldots, ca_n)$ we have:

$$\langle p, q \rangle \in I_{inst}(instOf)$$

As examples, the following are class terms with class methods:

$student([isa \to student], [average\text{-}mark \overset{c}{\to} 75], [average\text{-}mark \overset{c}{\Rightarrow} integer])$

$family\text{-}car([isa \to car], [averageweight \overset{c}{\to} 2000], [averageweight \overset{c}{\Rightarrow} integer])$

Note that it would be difficult to extend F-logic [KifLaWu90] with class methods having such properties. Consider the class method *average-mark* on the class *student*. Assume that the class student has instances of individual students and subclasses such as *gradstd*, *undergradstd* and *visitingstd*. Recall that F-logic does not distinguish between class objects and individual objects. If we use F-logic, we will have difficulty in calculating the ***average-mark***, because the instances of the class ***student*** include both those intended as individual students and those intended as subclasses. This example shows ***the need for distinguishing between class objects and instance objects.*** It shows clear superiority of OOL over F-logic in this respect.

4.9 Shared Methods

In general, a ***shared method*** is a method that is specified on a class and the method applies to each instance of that class.

We can extend the kernel of OOL with shared methods. Let's use "$\overset{s}{\rightarrow}$" (respectively, "$\overset{s}{\twoheadrightarrow}$") for single-valued (respectively, set-valued) shared methods instead of the usual symbol "$\rightarrow$" (respectively, "$\twoheadrightarrow$") used for the ordinary methods. We also need an additional symbol "$\overset{s}{\Rightarrow}$" (or respectively, "$\overset{s}{\Rightarrow\!\!\!\Rightarrow}$ ") for defining the signature of a single-valued shared method (respectively, a set-valued shared method). Note that this signature is defined on the same class where the corresponding shared method is defined.

The general idea of shared methods is inspired by [Kim90]. We modify the subtle properties of shared methods to suit OOL. We want each shared method to have the following properties:

- Each shared method is specified only on a class.
- A shared method specified on a class is inherited (non-monotonically) by each of its subclasses, provided that method is not redefined on a subclass. This inheritance is non-monotonic, because a subclass is allowed to redefine the value of the inherited shared method. In other words, the same shared method specified on a subclass overrides the one it inherited from one of its superclasses.

- Every shared method is shared (monotonically inherited) by each *immediate instance*[1] of the class. When an instance belongs to several classes, and the same shared method is defined on more than one of those classes, that instance shares the method defined on the most specific one (i.e., its immediate class).

  No instance can have a different shared-method value(s) from what is defined in its immediate class.

- For each instance that belongs to more than one class, we do not allow conflicting shared methods from some of its classes such that a class is not more specific than the others. For example, we do not allow the following situation:

  $$\begin{array}{ll} \mathit{gradstd}([\mathit{isa} \rightarrow \mathit{student}]) & \mathit{undergrad}([\mathit{isa} \rightarrow \mathit{student}]) \\ \mathit{student}([\mathit{passing\text{-}grade} \overset{s}{\rightarrow} 65]) & \mathit{gradstd}([\mathit{passing\text{-}grade} \overset{s}{\rightarrow} 80]) \\ \mathit{doe}([\mathit{instOf} \rightarrow \mathit{gradstd}]) & \mathit{doe}([\mathit{instOf} \rightarrow \mathit{undergrad}]) \end{array}$$

  In this example *doe* belongs to both classes: *gradstd* and *underg·ad.* Class *gradstd* and class *undergrad* have the same shared method *passing-grade* with conflicting values (note that class *undergrad* inherits the method from *student*). Since we cannot say that one of them is more specific than the other; this kind of situation is not allowed.

- The signature of a shared method is defined on the same class where that shared method is defined. As usual, the signature is inherited monotonically by subclasses of the class.

This concept of shared methods is not defined in the kernel of OOL. The benefit of this is that OOL is not tied to any particular style of shared method inheritance.

[1]an instance is called an *immediate instance* of a class if the class is an immediate class of the instance

As mentioned above, a shared method specified on a subclass overrides the same method inherited from its superclass. This is known as Touretzky's principle [Tour86] that states: An inherited property from a more specific superclass should overwrite the inherited property due to a less specific superclass. The inheritance that we want is more general than that of [Tour86], because inherited properties are not limited to *zero*-ary methods (also called attributes).

F-logic [KifLaWu90] solved the problem of non-monotonic method inheritance by introducing preorder relations over models. We adapt this idea to OOL. However, we need some modification because in F-logic there is no distinction between class objects and instance objects.

The non-monotonic inheritance in OOL is defined through model-theoretic characterization. Following F-logic [KifLaWu90], the semantics of the non-monotonic inheritance is described using Shoham's minimization principle [Shoh88]. The main idea of Shoham's minimization principle is defining a preorder $\sqsubseteq^{inh}$ relation over models. In this ordering, "smaller" means that the *value* returned by some method changes "less" when descending down the class hierarchies. We shall explain this by the following example, which is a modification of an example in [Tour86 and KifLaWu90].

Suppose that we have the following clauses:

1. $elephant([colour \overset{s}{\rightarrow} \text{“}grey\text{”}])$
2. $royal\text{-}elephant([isa \rightarrow elephant], [colour \overset{s}{\rightarrow} \text{“}white\text{”}])$
3. $strange\text{-}elephant([isa \rightarrow royal\text{-}elephant])$

Then, assume that we have a model M_1 such that:

- $M_1 \models royal\text{-}elephant([isa \rightarrow elephant])$,
- $M_1 \models strange\text{-}elephant([isa \rightarrow royal\text{-}elephant])$,

- $M_1 \models \mathit{elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{grey}\text{”}])$,
- $M_1 \models \mathit{royal\text{-}elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{white}\text{”}])$, and
- $M_1 \models \mathit{strange\text{-}elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{white}\text{”}])$.

and a model M_2 such that:

- $M_2 \models \mathit{royal\text{-}elephant}([\mathit{isa} \rightarrow \mathit{elephant}])$,
- $M_2 \models \mathit{strange\text{-}elephant}([\mathit{isa} \rightarrow \mathit{royal\text{-}elephant}])$,
- $M_2 \models \mathit{elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{grey}\text{”}])$,
- $M_2 \models \mathit{royal\text{-}elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{white}\text{”}])$, and
- $M_2 \models \mathit{strange\text{-}elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{brown}\text{”}])$.

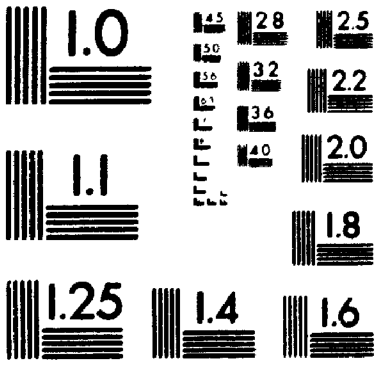
We say that $M_1 \sqsubseteq^{vh} M_2$ because in model M_1 the value of the method *colour* does not change when we descend from *royal-elephant* to *strange-elephant*, i.e., the colour for both classes is white (in model M_2, the colour changes from white to brown). This makes the value of the method *colour* change less in model M_1 than the one in model M_2.

Furthermore, consider a semantic structure M_3 such that:

- $M_3 \models \mathit{royal\text{-}elephant}([\mathit{isa} \rightarrow \mathit{elephant}])$,
- $M_3 \models \mathit{strange\text{-}elephant}([\mathit{isa} \rightarrow \mathit{royal\text{-}elephant}])$,
- $M_3 \models \mathit{elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{grey}\text{”}])$,
- $M_3 \models \mathit{royal\text{-}elephant}([\mathit{colour} \overset{s}{\rightarrow} \text{“}\mathit{grey}\text{”}])$, and

3 of/de 3

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT



- $M_3 \models$ *strange-elephant*([*colour*$\xrightarrow{s}$"*grey*"]).

The structure M_3 is smaller than M_2. It is even smaller than M_1 (i.e., $M_3 \sqsubseteq^{inh} M_1$) because the value of method ***colour*** does not even change when we descend from ***elephant*** to ***royal-elephant***. However, M_3 is ***not*** a model because it does not satisfy the second clause. In this example, model M_1 is $\sqsubseteq^{inh}$ minimal.

Following F-logic [KifLaWu90], in the case of conflicting multiple inheritance, we use the Credulous approach [TouHoTh87] which allows a subclass to inherit *either* of the possibilities. Another possible solution is to have the decision up to the user (an ad-hoc approach).

In order to define preorder $\sqsubseteq^{inh}$, we will use the following additional notation which is adapted from [KifLaWu90]. Let I be a semantic structure, *gm* a ground shared method, *curcl* a ground id-term (intended to represent the current class), and $\vec{args} = \langle a_1, \ldots, a_k \rangle$ a vector of ground id-terms.

$$I(gm, \xrightarrow{s})(curcl, \vec{args}) = \begin{cases} \{v \mid I \models curcl([gm(\vec{args}) \xrightarrow{s} v])\} \\ \quad \text{if at least one such } v \text{ exists} \\ \\ \text{undefined, otherwise} \end{cases}$$

$$I(gm, \overset{s}{\twoheadrightarrow})(curcl, \vec{args}) = \begin{cases} \{v \mid I \models curcl([gm(\vec{args}) \overset{s}{\twoheadrightarrow} \{v\}])\} \\ \quad \text{if } I \models curcl([gm(\vec{args}) \overset{s}{\twoheadrightarrow} \{\}]) \\ \\ \text{undefined, otherwise} \end{cases}$$

Notice that $I(gm, \xrightarrow{s})(curcl, \vec{args})$ and $I(m, \overset{s}{\twoheadrightarrow})(curcl, \vec{args})$ denote sets of ground id-terms (distinguish this notation from $I_{\rightarrow}(\ldots)(\ldots)$ and $I_{\twoheadrightarrow}(\ldots)(\ldots)$, in Section 3.4, that returns the elements of the domain of I).

The preorder $\sqsubseteq^{inh}$ is defined formally as follows. Let I and J be a pair of semantic structures. Then, we have $I \sqsubseteq^{inh} J$ if one of the following two conditions is satisfied.

1. There are ground id-terms *curcl, gm,* and a vector of ground id-terms $\vec{args}$, where $I(gm, \overset{s}{\rightarrow})(curcl, \vec{args})$ is defined, such that

   $$I(gm, \overset{s}{\rightarrow})(curcl, \vec{args}) \neq J(gm, \overset{s}{\rightarrow})(curcl, \vec{args}), \text{ or}$$

   $$J(gm, \overset{s}{\rightarrow})(curcl, \vec{args}) \text{ is undefined.}$$

   Furthermore, there is a ground class term *eqCl* such that the following conditions are true:

   - $I \models curcl([isa \rightarrow eqCl])$, $J \models curcl([isa \rightarrow eqCl])$, and $I(gm, \overset{s}{\rightarrow})(eqCl, \vec{args}) = J(gm, \overset{s}{\rightarrow})(eqCl, \vec{args})$
   - for every class *mid* such that

     $$I \models curcl([isa \rightarrow mid]) \text{ and } I \models mid([isa \rightarrow eqCl]).$$

     It is the case that

     $$I(gm, \overset{s}{\rightarrow})(mid, \vec{args}) = I(gm, \overset{s}{\rightarrow})(eqCl, \vec{args}) = I(gm, \overset{s}{\rightarrow})(curcl, \vec{args})$$

2. Similarly for set-valued methods, i.e.,

   There are ground id-terms *curcl, gm,* and a vector of ground id-terms $\vec{args}$, where $I(gm, \overset{s}{\twoheadrightarrow})(curcl, \vec{args})$ is defined, such that

   $$I(gm, \overset{s}{\twoheadrightarrow})(curcl, \vec{args}) \neq J(gm, \overset{s}{\twoheadrightarrow})(curcl, \vec{args}), \text{ or}$$

   $$J(gm, \overset{s}{\twoheadrightarrow})(curcl, \vec{args}) \text{ is undefined.}$$

   Furthermore, there is a ground class term *eqCl* such that the following conditions are true:

- $I \models curcl([isa \rightarrow eqCl])$, $J \models curcl([isa \rightarrow eqCl])$, and $I(gm, \overset{s}{\rightarrowtail})(eqCl, \vec{args}) = J(gm, \overset{s}{\rightarrowtail})(eqCl, \vec{args})$
- for every class *mid* such that

$$I \models curcl([isa \rightarrow mid]) \text{ and } I \models mid([isa \rightarrow eqCl]).$$

It is the case that

$$I(gm, \overset{s}{\rightarrowtail})(mid, \vec{args}) = I(gm, \overset{s}{\rightarrowtail})(eqCl, \vec{args}) = I(gm, \overset{s}{\rightarrowtail})(curcl, \vec{args})$$

Intuitively, we say that $I \sqsubseteq^{inh} J$ if for some class *curcl*, I and J disagree on the interpretation of a shared method gm on some arguments $\vec{args}$, and we can show that the behaviour of gm in I is obtained by inheritance. However, we may also have $J \sqsubseteq^{inh} I$ based on another case of behaviour inheritance, this is why the relation $\sqsubseteq^{inh}$ is only preorder.

Now, we define models that are minimal for inheritance as follows:

Definition An untyped model M of a program P is *minimal for inheritance* (called $\sqsubseteq^{inh}$ minimal) if and only if for every other untyped model N of P, $N \sqsubseteq^{inh} M$ implies $M \sqsubseteq^{inh} N$. ∎

In order to combine inheritance and $\ll^{hc}$-minimization we need to modify the definition of minimal untyped model in Section 4.2.

Definition Let P be a Horn clause program. A *minimal untyped model* of P is an untyped semantic structure that is $\ll^{hc}$-minimal among the $\sqsubseteq^{inh}$ -minimal untyped models of P. ∎

A minimal typed model of program P is defined as before, i.e., a minimal untyped model that also fulfills the well-typing conditions.

We now define the non-monotonic logical entailment, $\models\!\!\approx$, as follows:

Definition Let P be a Horn clause program, then $P \approx\!\!\!\models \phi$ if and only if for every minimal typed model $M \models P$ then $M \models \phi$. ∎

Notice from the definition of minimal untyped model of program P that the $\sqsubseteq^{inh}$ -minimization must be done before $\ll^{hc}$-minimization. The process of $\sqsubseteq^{inh}$ -minimization followed by $\ll^{hc}$-minimization achieves the intended non-monotonic inheritance. For example, suppose that we have the following program:

1. *elephant*([*colour* $\xrightarrow{s}$ *"grey"*])
2. *royal-elephant*([*isa* $\rightarrow$ *elephant*], [*colour* $\xrightarrow{s}$ *"white"*])
3. *strange-elephant*([*isa* $\rightarrow$ *royal-elephant*])
4. *strange-elephant*([*isa* $\rightarrow$ *white-animals*]) :- *strange-elephant*([*colour* $\xrightarrow{s}$ *"white"*])

If we do $\ll^{hc}$-minimization before $\sqsubseteq^{inh}$ -minimization, then we will have a minimal untyped model where the terms *strange-elephant*([*isa* $\rightarrow$ *white-animals*]) and *strange-elephant*([*colour* $\xrightarrow{s}$ *"white"*]) are false because of the $\ll^{hc}$-minimization. However, if we do $\sqsubseteq^{inh}$ -minimization first, the two terms will be true because of the inheritance property, where *strange-elephant* inherits the colour *"white"*. This example illustrates the importance of the order of minimization, because the desired effect of non-monotonic inheritance is only achieved through the correct minimization order.

The following modified example from [KifLaWu90] illustrates the mechanism of inheritance overwriting. Let a program P be:

$p([isa \rightarrow q])$

$q([sm \xrightarrow{s} r])$

$p([sm \xrightarrow{s} t])$

Here, we want the clause $p([sm \xrightarrow{s} t])$ to overwrite the inheritance of $[sm \xrightarrow{s} r])$ from the superclass of p (i.e., q). In all models, the clause $p([sm \xrightarrow{s} t])$ must be true. In some model, we may have that $p([sm \xrightarrow{s} r])$ is true, and $r \doteq t$ is also true. Although such a model will survive $\sqsubseteq^{inh}$ -minimization, it will be eliminated during $\ll^{hc}$-minimization.

Similar to F-logic, inherited methods are overwritten point-wise, as shown by the following sample program:

$$p([isa \rightarrow q])$$
$$q([sm\langle a_1, a_2\rangle \overset{s}{\rightarrow} b], [sm\langle c_1, c_2\rangle \overset{s}{\rightarrow} d])$$
$$p([sm\langle a_1, a_2\rangle \overset{s}{\rightarrow} e])$$

In this example, the inheritance of $[sm\langle a_1, a_2\rangle \overset{s}{\rightarrow} b]$ is overwritten, but the inheritance of $[sm\langle c_1, c_2\rangle \overset{s}{\rightarrow} d])$ is not.

The inheritance overwriting for set-valued methods is rather peculiar as illustrated in the following program:

$$p([isa \rightarrow q])$$
$$q([sm_1 \overset{s}{\twoheadrightarrow} \{a, b\}], [sm_2 \overset{s}{\twoheadrightarrow} \{a, b, c\}])$$
$$p([sm_1 \overset{s}{\twoheadrightarrow} \{b\}], [sm_2 \overset{s}{\twoheadrightarrow} \{e\}])$$

In this example, the inheritance of $[sm_2 \overset{s}{\twoheadrightarrow} \{a, b, c\}]$ for p is overwritten, but the inheritance of $[sm_1 \overset{s}{\twoheadrightarrow} \{a, b\}]$ is still valid. We can see here that redefining the inherited value of method sm_1 from $\{a, b\}$ to its subset $\{b\}$ does not overwrite the inherited one. In general, based on the satisfaction of set-valued methods in OOL, redefining with a new value that is a subset of the inherited one will *not* overwrite the inherited one.

It should be noted that although the idea of non-monotonic inheritance of shared methods is the result of a modification of the technique of non-monotonic method inheritance used in an extension of F-logic [KifLaWu90], F-logic cannot have one of the properties we imposed: the ability to override an inherited shared method is limited to subclasses (every instance object cannot override its inherited shared methods). In F-logic, this ability would belong to any instance of the class since we cannot distinguish between instance objects and class objects. This shared-method extension of OOL shows another advantage of making a clear distinction between class objects and instance objects.

4.10 Example

In this section we present a sample OOL program about a university database that shows various features of OOL.

4.10.1 Subclass-Relationship Definitions

Figure 4.1 shows the subclass-relationship assertions that define part of the class hierarchies in the university database.

Sentence (1) asserts that class *student* is a subclass of class *person*. Similarly sentences (2) and (3) assert that *employee* and *child* are subclasses of class *person*. Sentence (4) asserts that *gradstd* is a subclass of both classes *student* and *employee*. Sentences (5) and (6) assert that *professor* is a subclass of *employee*, and *cs365std* is a subclass of *student*.

4.10.2 Instance-of and Data Definitions

Sentences (7) to (12) in Figure 4.2 define some parts of the instance-of relationships and data definitions in the university database.

Sentence (7) asserts that instance object *john* is an instance of class *professor*. It also assert the values of the attributes *name*, *address*, *children*, and *employedby* to "Johnny", "London,Ontario", $\{jean, tom, jane\}$, and *csd* respectively. The attribute *children* is set-valued, which is indicated by the double-headed arrow symbol "$\twoheadrightarrow$".

Similarly, sentences (8), (9), and (10) assert the instance-of relationships and other properties of objects *brian*, *sarah*, and *csd*. Notice in sentence (10) that it asserts the 1-ary method *numberOfUndergrad* and *numberOfGrad* whose values for the year 1993 are 270 and 30 respectively.

Sentence (11) asserts that *doe* is an instance of class *cs365std*.

Finally, sentence (12) asserts that *kelly* is an instance of class professor, has *jean* and *jane* as her children, and several other properties.

1. $student([isa \rightarrow person])$
2. $employee([isa \rightarrow person])$
3. $child([isa \rightarrow person])$
4. $gradstd([\ isa \rightarrow student],$
   $[\ isa \rightarrow employee])$
5. $professor([isa \rightarrow employee])$
6. $cs365std([\ isa \rightarrow student])$

Figure 4.1: Subclass-Relationship Definitions

4.10.3 Signature Definitions

In Figure 4.3, sentences (13) to (19) define some signatures, the class methods and the shared methods of the database.

Sentence (13) defines the typing for instances of class *person*. It asserts that the output type of methods *name* and *address* is *string*, and the output type of set-valued method *children* is from class *child*.

Sentence (14) asserts the typing of methods *employedby* and *boss* on class *employee*.

7. $john([instOf \rightarrow professor],$
   $[name \rightarrow \text{“}Johnny\text{”}],$
   $[address \rightarrow \text{“}London, Ontario\text{”}],$
   $[children \twoheadrightarrow \{jean, tom, jane\}],$
   $[employedby \rightarrow csd])$
8. $brian([instOf \rightarrow professor],$
   $[name \rightarrow \text{“}Brian\text{”}],$
   $[address \rightarrow \text{“}London, Ontario\text{”}],$
   $[salary \rightarrow 60000],$
   $[employedby \rightarrow csd])$
9. $sarah([instOf \rightarrow employee],$
   $[name \rightarrow \text{“}Sarah\text{”}],$
   $[address \rightarrow \text{“}London, Ontario\text{”}],$
   $[employedby \rightarrow csd])$
10. $csd([instOf \rightarrow depart],$
   $[deptName \rightarrow \text{“}ComputerScience\text{”}],$
   $[chair \rightarrow john],$
   $[address \rightarrow \text{“}London, Ontario\text{”}],$
   $[secretaries \twoheadrightarrow \{sarah, rob, jean\}],$
   $[numberOfUndergrad\langle 1993\rangle \rightarrow 270],$
   $[numberOfGrad\langle 1993\rangle \rightarrow 30])$
11. $doe([instOf \rightarrow cs365std])$
12. $kelly([instOf \rightarrow professor],$
   $[name \rightarrow \text{“}Kelly\text{”}],$
   $[address \rightarrow \text{“}London, Ontario\text{”}],$
   $[children \twoheadrightarrow \{jean, jane\}]\ )$

Figure 4.2: Instance-of and Data Definitions

13. $person([name \Rightarrow \{string\}],$ /*typing, single-valued attribute*/
$[address \Rightarrow string],$ /*we may remove curly brackets when*/
$[children \Rightarrow\!\!\!\Rightarrow child])$ /*only one class is involved*/

14. $employee([employedby \Rightarrow depart],$
$[boss \Rightarrow employee])$

15. $depart([departName \Rightarrow string],$
$[chair \Rightarrow professor],$
$[address \Rightarrow string],$
$[secretaries \Rightarrow\!\!\!\Rightarrow \{employee\}],$ /*typing, set-valued attribute,*/
$[numberOfUndergrad\langle year\rangle \Rightarrow integer],$
$[numberOfGrad\langle year\rangle \Rightarrow integer].$
$[numberOfStd\langle year\rangle \Rightarrow integer])$

16. $professor([boss \Rightarrow professor],$
$[degrees \Rightarrow\!\!\!\Rightarrow$ string $],$
$[publish \Rightarrow\!\!\!\Rightarrow article],$
$[salary \Rightarrow integer]),$
$[highestSalary \overset{c}{\Rightarrow} integer].$ /*typing. a single-valued class method*/
$[highestSalary \overset{c}{\rightarrow} 100000])$ /* asserting a class method value */

17. $student(\ [courses \Rightarrow\!\!\!\Rightarrow string])$

18. $gradstd([teaching \Rightarrow\!\!\!\Rightarrow string])$

19. $cs365std([passing\text{-}grade \overset{s}{\Rightarrow} integer],$ /*typing. a single-valued shared method*/
$[passing\text{-}grade \overset{s}{\rightarrow} 65])$ /* asserting a shared method value */

Figure 4.3: Signature Definitions

20. $E([boss \rightarrow B])$:- $E([instOf \rightarrow employee]) \wedge$
$E([employedby \rightarrow Dept]) \wedge$
$Dept([chair \rightarrow B])$

21. $D([numberOfStd\langle Year \rangle \rightarrow sum\langle U, G\rangle])$:- $D([instOf \rightarrow depart]) \wedge$
$D([numberOfUndergrad\langle Year\rangle \rightarrow U]) \wedge$
$D([numberOfGrad\langle Year\rangle \rightarrow G])$

Figure 4.4: Inference Rule Definitions

Sentence (15) asserts that the output of methods *departName, chair, address,* and *secretaries* are from classes *string, professor, string,* and *employee* respectively. It also asserts that the arguments of 1-ary method *numberOfUndergrad* and *numberOf-Grad* are from class *year* and their outputs are from class *integer.*

Sentence (16) defines the output types of methods *boss, degrees, publish* and *salary.* This sentence also asserts the value of *class* method *highestSalary* and its signature. The method is recognized as a class method from the special symbol "$\overset{c}{\rightarrow}$" and the signature is recognized as a class-method signature from the symbol "$\overset{c}{\Rightarrow}$". Recall that class methods are not inherited either by subclasses or by instances.

Sentence (17) and (18) assert the output types of methods *courses* and *teaching* respectively.

Sentence (19) asserts the value of *shared* method *passing-grade* and its signature. This shared method and its signature are indicated by the special symbols "$\overset{s}{\rightarrow}$" and "$\overset{s}{\Rightarrow}$" respectively. Recall that shared methods are inherited non-monotonically by subclasses and shared monotonically by their immediate instances.

Signature assertions are *not* enforced in OOL, but when the signature of a method is defined then the typing constraint is enforced on the argument(s) and output(s) of the method.

4.10.4 Inference Rule Definitions

Figure 4.4 shows two examples from the rules of the database.

Sentence (20) defines the conditions for the value of method *boss* on instances from class employee. Similarly, sentence (21) defines the conditions for the value of method *numberOfStd* in any given year for instances of class *depart.*

4.10.5 Sample Queries

Figure 4.5 shows some sample queries. The first query is asking for the number of students within *csd* in year 1993. To answer this query, rule (21) defining the 1-ary method *numberOfStd* is used. Based on rule (21) and sentence (10), the answer of this query is $N = 300$.

$:\text{-}\ csd([numberOfStd\ \langle 1993\rangle \rightarrow N])$

$:\text{-}\ sarah([boss \rightarrow B])$

$:\text{-}\ doe([passing\text{-}grade \rightarrow M])$

$:\text{-}\ john([children \twoheadrightarrow \{C\}])$

$:\text{-}\ john([children \twoheadrightarrow \{C\}]) \wedge kelly([children \twoheadrightarrow \{C\}])$

$:\text{-}\ E([instOf \rightarrow employee], [employedby \rightarrow csd])$

$:\text{-}\ S([instOf \rightarrow cs365std\ ])$

Figure 4.5: Sample Queries

The second query is asking for the name of *sarah*'s boss. In order to answer this query, rule (20) which defines the attribute *boss* is used. Based on rule (20), sentence

(9), and sentence (10), the answer is $B = john$.

The third query is asking for the value of method *passing-grade* on instance *doe*. Here, although, the method is not defined on the instance itself, the method value is obtained through the shared method (i.e., *passing-grade*) which is defined on the *doe*'s class, i.e., *cs365std*. So, the answer to this query is $M = 65$.

The fourth query is asking for the children of *john*. If we want the actual names of the children (instead of just the object identities), we can rewrite the query as

$$:\!-\ john([children \twoheadrightarrow \{C([name \rightarrow N])\}])$$

The fifth query is asking for the children that are shared by both *john* and *kelly*.

The sixth question is asking for the employees of the *csd* department (i.e., Computer Science Department's employees).

The seventh question is asking for the students who are instances of *cs365std* (i.e., students of the course CS365).

4.11 Discussion

We have seen that the monotonic inheritance of signatures is defined in the kernel of OOL, while the non-monotonic inheritance of shared methods is defined outside the kernel of OOL. Shared method inheritance is defined at a meta level through a minimization principle.

Extending OOL to include class methods requires only minor changes to the kernel of OOL semantics, because class methods and their signatures are not inherited by subclasses or instances. The concept of class methods also shows one possible extension, that can be added easily to OOL, but that would cause a problem (pointed out in 4.8) if this extension were done on F-logic. We do not include class methods in the kernel of OOL. This is done to simplify the kernel of OOL semantics and to include only the main characteristics of OO databases.

The monotonic inheritance of signatures is defined in the kernel semantics of OOL for the following reasons: First, this monotonic inheritance is considered unlikely to reduce the opportunity of having a complete proof procedure. Second, a common goal in logical modeling of numerous knowledge representation paradigms is to entrench the semantics as much as is practical [KifLaWu90]. Thus, since including the monotonic inheritance does not reduce the opportunity of finding a complete proof procedure, integrating it into the kernel semantics of OOL is the best choice.

The reason why the inheritance of shared methods is defined at the meta level is: Shared method inheritance involves a non-monotonic derivation rule i.e., inheritance of shared methods from a more specific superclass overrides inheritance from a less specific one. Thus if we integrate this principle into the kernel semantics of OOL, it may potentially reduce our opportunity of finding a complete proof procedure. However, we actually benefit from taking this decision. By putting the inheritance of shared methods outside the kernel semantics of OOL, we do not restrict OOL to any specific method of inheritance principle. Thus we can change the method-inheritance principle without changing the logic itself.

CHAPTER 5

Conclusions and Future Work

The goal of this thesis is to design a logic that can be used as a basis in designing object-oriented databases, similar to the role of Predicate Calculus which was used by Codd as the basis for designing relational databases [Codd 70]. Such an object-oriented logic can be used for reasoning about object-oriented database systems, and viewed as a formal basis for integrating the ideas of deductive databases and object-oriented databases.

We have designed an object-oriented logic (called OOL) that has the key properties required for object-oriented databases as described in [Kim90]. This logic can be used as a logical framework for natural representations and manipulations of complex objects. Similar to F-logic [KifLa89, KifLaWu90], this logic has overcome the objections to the use of logic-programming for object-oriented databases as stated by J.D. Ullman [Ullman92].

The main contribution of this thesis is designing a suitable logic for providing a firm-theoretical ground of OODBs, as Predicate Calculus serves as a firm-theoretical ground of relational databases. As stated in [NeuSton89], a theoretical ground is needed so that researchers in OODBs will be able to use a common set of terms and to define common goals.

The object-oriented properties that OOL has are object-identity, the ability to represent complex objects (including attributes, methods), typing, class, class hierarchy and inheritance. It also has a sound and complete resolution-based proof procedure, which makes it interesting computationally.

5.1 OOL and F-Logic [KifLa89, KifLaWu90]

When OOL was designed, we considered that F-logic [KifLa89, KifLaWu90] was the most comprehensive object-oriented logic known (in the sense of the object-oriented properties the logic has and in the sense of the discussions presented for the logic). Recall that F-logic [KifLa89, KifLaWu90] does not distinguish between instance-of relationships and subclass relationships. This property of F-logic causes us to "guess" from the context whether an object is intended to be an instance (individual) that cannot have any instance, or a class that can have instances. In other words, we cannot distinguish an instance from a class in F-logic.

One of the directions in OOL development was to change the situation where "an instance is also a class" of F-logic, in order to overcome problems such as mentioned at the beginning of Chapter 3. We have succeeded in making this change. OOL has clear syntactic and semantic distinctions between instance-of relationships and subclass relationships (see Chapter 3), as well as most properties of F-logic. In addition, we have shown (in Chapter 4) that OOL can be extended with class methods and shared methods, and pointed out that these extensions would be difficult to be done in F-logic [KifLaWu90].

We consider the syntactic distinction between the two kinds of relations in OOL is important, because basically a proof theory of a logic is a certain way of manipulating its symbols (syntactic objects). Thus, syntactic clarity plays an important role in reading and writing a program. Such clarity in syntax will help programmers and users in writing correct programs and in debugging.

5.2 OOL and the New F-Logic [KifLaWu94]

During its development, OOL was designed independently from the new F-logic [KifLaWu94] (we will call it "NF-logic" to distinguish it from previous versions of F-logic [KifLa89, KifLaWu90]).

The main difference between NF-logic [KifLaWu94] and F-logic [KifWu89, KifLaWu90] is that NF-logic has additional instance-of assertions. The reason for these additional assertions is not mentioned in [KifLaWu94]. Perhaps the addition is due to problems such as mentioned in the beginning of Chapter 3.

Despite the additional instance-of assertions, NF-logic [KifLaWu94] still has significant differences from OOL as follows:

- NF-logic:

  Similar to F-logic [KifLa89, KifLaWu90], an instance can also be a class and vice versa.

  OOL:

  One of the basic ideas of OOL is to make a clear distinction between classes and instances syntactically and semantically. Consequently, no instance can be a class, and vice versa.

- NF-logic:

  The well-typing conditions are not part of the kernel semantics of NF-logic, they are defined at the meta level of an extension of NF-logic. Thus, for any method $\mu \in U$ (where U is the domain of interpretation), there is no connection between the interpretation of the single-valued method $I_{\rightarrow}(\mu)$ (or respectively, $I_{\twoheadrightarrow}(\mu)$ for set-valued method) and the interpretation of the single-valued method signature $I_{\Rightarrow}(\mu)$ (respectively, $I_{\Rightarrow\!\!\!\Rightarrow}(\mu)$).

OOL:

The well-typing conditions are part of the kernel semantics of OOL, which provide the connection between $I_{\rightarrow}(\mu)$ and $I_{\Rightarrow}(\mu)$ and respectively, $I_{\twoheadrightarrow}(\mu)$ and $I_{\Rightarrow\!\!\!\Rightarrow}(\mu)$.

- NF-logic:

  Methods can be defined on both classes and instances.

  OOL:

  Methods can be defined on instances only. This is the logical consequence of a clear distinction between classes and instances: definitions of methods can only be for instances and definitions of method signatures can only be for classes. In Chapter 4, we discussed an extension of OOL that allows us to define *class methods* and *shared methods.* A class method is a different kind of methods intended to capture an aggregate property of the instances of the class. On the other hand, a shared method defined in a class is used to capture the shared property of the class' instances.

- NF-logic:

  Method signatures can be defined on both classes and instances.

  OOL:

  Method signatures can be defined on classes only.

- NF-logic:

  There is no restriction on the instance-of relation. Particularly, cyclic relationships are allowed, e.g., for any $a, b \in U$, both "the object a is an instance of object b" and "the object b is an instance of object a" are allowed, furthermore "the object a is an instance of object a" is also allowed.

OOL:

As a direct consequence of a clear distinction between classes and instances, a cyclic instance-of relationship is not allowed. It is shown in the next section that NF-logic allows a contradiction because it permits a class to be an instance of itself.

- The antimonotonicity property of method signatures is defined differently in OOL from that of F-logic or NF-logic, as described in Section 3.4. This different definition of antimonotonicity has helped in allowing the well-typing conditions to be included in the kernel semantics of OOL.

5.2.1 A Contradiction in the New F-Logic

NF-logic [KifLaWu94] has added a separate instance-of assertion symbol (corresponding to a membership relation in the semantics) to the existing subclass assertion symbol on F-logic [KifLaWu90]. Interestingly, for a flexibility reason, NF-logic allows a cyclic membership relation. In particular, a class is allowed to be an instance (member) of itself.

Inspired by how Bertrand Russel showed a paradox in the Frege logic system in 1902, we will show that a similar paradox can also happen in NF-logic.

We may formulate a set of classes that are not instances of themselves as follows:

Note that NF-logic uses the symbol ":" for the instance-of assertions, where, for example, $I : C$ is intended to mean that I is an instance of C.

$$\exists \mathrm{N}\ \forall \mathrm{C}\ (\mathrm{C} : \mathrm{N} \leftrightarrow \neg(\mathrm{C} : \mathrm{C}))$$

Let *nsac* be the class whose members are classes that cannot be members of themselves. Can *nsac* be a member of *nsac*? The proof below shows that from each answer its contradiction follows. Therefore we must conclude that *nsac* is not a class.

Proof:

(1) $\exists N\ \forall C\ (C : N \leftrightarrow \neg(C : C))$.

(2) $\forall C\ (C : nsac \leftrightarrow \neg(C : C))$. From (1).

(3) $(nsac : nsac \leftrightarrow \neg(nsac : nsac))$. From (2).

(4a) nsac : nsac
From (3)

(5a) $\neg$(nsac : nsac).
From (3) and (4a).
A contradiction of (4a)!

(4b) $\neg$(nsac : nsac).
From (3)

(5b) nsac : nsac.
From (3) and (4b)
A contradiction of (4b)! ■

The proof above shows a contradiction in NF-logic. This clearly shows the need for the restriction that a class should not be allowed to be an instance (or a member) of itself. This restriction is inherent in OOL where a class cannot play a role as an instance and an instance cannot play a role as a class.

5.3 Specification of OODB Concepts Based on OOL

In Chapter 3, we present a logic that is suitable for object-oriented databases called OOL (object oriented logic). We suggest that OOL can play a similar role with regard to OODBs to that of Predicate Calculus being used as the basis for the specification of relational database systems.

The following is a summary of OO database concepts based on OOL:

- *Object and Object Identity.*

  Every real-world entity is uniformly modeled as an object. Then, every object is associated with a unique identifier, which is commonly called an object identity.

- *Attribute.*

  A zero-ary method (i.e., a method without any argument) is regarded as an attribute in the sense of [Kim90]. An object's *state* includes the values of applicable zero-ary methods, and its instance-of relation.

  There are two types of attributes: single-valued and set-valued attributes. The value of a single-valued attribute is also an object. The value of a set-valued attribute is a set of objects. The set of objects itself is not an object. However, it is possible to simulate a set of objects as an object with an attribute whose value is that set of objects.

- *Method.*

  A non-zero-ary method is regarded as a method in the sense of [Kim90]. An object's *behaviour* is the set of applicable non-zero-ary methods and their values. There are two types of methods: single-valued and set-valued methods.

- *Class.*

  A class is used for grouping all objects sharing the same set of methods, and for defining the type of a method's arguments and the type of its output. Unlike [Kim90], an object is allowed to belong to more than one class as those classes' instance. A class definition includes: the *subclass* relation specification, and the signatures (the typing) of its instances' methods. A new class may be defined using an existing class, this new class is called a subclass.

Based on the extension of OOL, we may define class methods (see Section 4.8) and shared methods (see Section 4.9).

A class method is a method defined on a class for capturing the aggregate properties of its instances. The signature of a class method is defined on the same class. The class methods and their signatures are not inherited either by subclasses or instances.

A shared method is a method defined on a class. This method applies to each immediate instance of the class (see Subsection 4.8.2 for its detailed properties).

- *Class Hierarchy and Inheritance.*

  A subclass inherits the subclass relation and methods' signatures from its superclass(es). Additional methods may be defined for a subclass. A class may have any number of subclasses.

  We may choose a single inheritance system or a multiple inheritance system. In a *single inheritance* system, we allow a class to have only one immediate superclass, where a class inherits the subclass relationship and the signatures of methods from only one immediate superclass (which also means inheriting all subclass relationships and signatures from the immediate superclass of the immediate superclass and so on until the root of the hierarchy). In this single-inheritance system, the classes form a tree like hierarchy (or tree like hierarchies) which is called a *class hierarchy* (or respectively, class hierarchies). In a *multiple inheritance* system, we allow a class to have any number of immediate superclasses, where a class inherits the subclass relationship and the signatures of methods from more than one immediate superclass. In this multiple-inheritance system, the classes form a rooted directed graph (or the more general, directed acyclic graphs).

  Unlike [Kim90], in OOL we do not assume the existence of a system defined class, CLASS, which is the only one root for all other classes in the system. We also do not assume that the directed acyclic graphs formed are connected.

5.4 Some Benefits of OOL

The following are some benefits of OOL:

- An object-oriented software technology is typically the result of a creative synthesis of previous ideas, tools and concepts [Banc93]. Its different origins make reaching agreement on its precise specification impractical or even damaging to the diversity of the field [Banc93]. Since the idea of object-oriented databases is much influenced by the development in object-oriented software technology, object-oriented database system developers have problems in determining a common specification. A strong theoretical framework is needed in this field.

  Since there is no commonly accepted formal foundation for object-oriented database systems, OOL can serve as their formal foundation; which in turn will help in forming a common data model for object-oriented database systems. As a formal foundation, we can use the results found in the logic for a better understanding of object-oriented database systems.

- In its application in a deductive object-oriented database system, OOL can reduce the impedance mismatch between the database query language and the host language.

- OOL as a formal foundation for object-oriented database systems will allow researchers to work on the same specification of an object-oriented database model for further investigations of its properties, features, limitations, modifiability, and possible extensions.

5.5 Future Work

Future work which could be done in the same direction is:

- The OOL programs we have discussed in Chapter 4 are restricted to Horn clauses. The restriction of a program to Horn clauses naturally reduces the complexity of finding a solution. However, this restriction also reduces significantly the expressive power of the language. There are previous results in classical logic programming for handling negative literals in the condition part of rules, such as [ApBlWal88, CavLlo89, Clark78, Reit78, PrzPrz88, Przym88, GelLif88], we might be able to adapt one of the existing techniques to OOL program. Research in finding a suitable way in handling non Horn clauses, especially negations on the rules' condition part would be a reasonable next possible step.

- In OO logic programming, we might find situations where some processes are better expressed in a procedural way. Developing an implementation technique that allows a user to program in a procedural way at certain points would be a useful feature.

- Solve problems associated with the heterogeneous structure of complex objects, such as how to store complex objects, how to cluster the components of complex objects together and how to store shared information.

REFERENCES

[Ande70] Anderson, F., and W. Bledsoe. A linear Format Resolution with Merging and a New Technique for Establishing Completeness. *Journal of the ACM*, 17(3), pp. 525–534, 1970.

[AbiKan89] Abiteboul, S., and P. Kannellakis. Object Identity as a Query Language Primitive. In *Proceedings of the ACM-SIGMOD 1989*, pp. 159–173, 1989.

[ApBlWal88] Apt, K. R., H. A. Bleir, and A. Walker. Towards a Theory of Declarative Knowledge, In *Foundation of Deductive Databases and Logic Programming*, J. Minker (ed), Los Altos, California, Morgan-Kaufmann, pp. 89–148, 1988.

[AtBa92] Atkinson, M., F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Building an Object-Oriented Database System, The Story of O_2*, F. Bancilhon, C. Delobel, and P. Kanellakis (eds), Morgan-Kaufmann, San Mateo, California, 1992.

[Banci86] Bancilhon, F. A Logic-Programming/Object-Oriented Cocktail. *SIGMOD RECORD*, Vol. 15, No. 3, September, pp. 11–21, 1986.

[Ban87] Banerjee, J., H. T. Chou, J. Garza, W. Kim, D. Woeld, N. Ballon, and H. J. Kim. Data Model Issues for Object-Oriented Applications. In *ACM Trans. Office Information Systems*, January, 1987.

[BunOh89] Buneman, P., and A. Ohori. Using Power Domains to Generalize Relational Databases. *Theoretical Computer Science*, 1989.

[CavLlo89] Cavedon, L., and J. W. Lloyd. A Completeness Theorem for SLDNF Resolution. In *Journal of Logic Programming*, 7, pp. 177–191, 1989.

[CheWa89] Chen, W., and D. S. Warren. C-logic for Complex Objects. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 369-378, March, 1989.

[ChKiWa89] Chen, W., M. Kifer, and D. S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings of the North American Conference on Logic Programming*, October, 1989.

[Clark78] Clark, K. L. Negation as Failure. In *Logic and Databases*, H. Gallaire and J. Minker (eds), New York, Plem Press. 1978.

[Codd70] Codd, E. F. A Relational Model For Large Shared Data Banks. *Communications of the ACM*, 13(6), pp. 377 387, 1970.

[Ende72] Enderton, H. B. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[FiKe85] Fikes, R., and T. Kehler, The Role of Frame-Based Representation in Reasoning, In *Communication of the ACM*, pp. 904–920, September, 1985.

[Gabr85] Gabriel, J., T. Lindholm, E. L. Lusk, and R. A. Overbeek. *A Tutorial on the Warren Abstract Machine for Computational Logic.* Technical Report ANL-84-84, Argonne National Laboratory, Argonne, Illinois, June, 1985.

[GelLif88] Gelfond, M., and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium.* p. 1070–1080, 1988.

[GelRosSch88] Van Gelder, A., K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 221-230, 1988.

[Gel89] Van Gelder, A. The alternating fixpoint of logic programs with negation. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 1–10, 1988.

[HulYos90] Hull, R., and M. Yoshikawa. ILOG: Declarative Creation & Manipulation of Object Identifiers. In *Proceedings of VLDB 1990*, pp. 455–468, 1990.

[Junus94] Junus, M. *A Logic for Object-Oriented Databases*, Technical Report # 425, Department of Computer Science, The University of Western Ontario, London, Ontario, May, 1994.

[KifLa89] Kifer, M., and G. Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, pp. 134–146, June, 1989.

[KifWu89] Kifer, M., 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*, pp. 379–393, March, 1989.

[KifLaWu90] Kifer, M., G. Lausen and J. Wu Logical Foundations of Object-Oriented and Frame-Based Language. *Technical Report 90/14*, Department of Computer Science, SUNY, NewYork, 1990.

[KifLaWu94] Kifer, M., G. Lausen and J. Wu Logical Foundations of Object-Oriented and Frame-Based Language. *Technical Report 93/06*, Department of Computer Science, SUNY, NewYork, (to appear in Journal of the ACM, 1995), 1994.

[Kim88] Kim, W., et al. Integrating an Object-Oriented Programming System with a Database System. In *Proceedings of Second International Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Diego, California, September, 1988.

[Kim89] Kim, W., et al. Features of The ORION Object-Oriented Database System. In *Object-Oriented Concepts, Applications, and Databases*. W. Kim and F. Lochovsky (eds), Reading, MA, Addison-Wesley, 1989.

[Kim90] Kim, W. Object-Oriented Databases: Definition and Research Directions. In *IEEE Transaction on Knowledge and Data Engineering*, Vol. 2, No. 3, September, 1990.

[LecRi88] Lecluse, C., and Richard, P. Modeling Inheritance and Genericity in Object-Oriented Data Model. In *Second International Conference on*

Database Theory (ICDT), LNCS # 326, pp. 223–238, Springer Verlag, Bruges, Belgium, 1988.

[LecRV88] Lecluse, C., P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. *Proc. of the ACM-SIGMOD International Conference on Management of Data-88*, pp. 424–433, 1988.

[Lloyd87] Lloyd, J. W. *Foundations of Logic Programming*, Second Edition, Springer Verlag, Berlin, 1987.

[LouOzs91] Lou, Y., and M. Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance. In *Proceedings of ACM-SIGMOD 1991*, pp. 198–207, 1991.

[Macle83] MacLennan, B. J. *A view of object oriented programming*, Naval Postgraduate School, NPS52-83-001, February, 1983.

[Maier86] Maier, D. A logic for Objects. *Workshop on Foundations of Deductive Databases and Logic Programming*, Washington DC, pp. 6–26, August, 1986.

[MaiWa80a] Maier, D., and D. S. Warren. *A Theory of Computed Relations*, Technical Report 80/12, Department of Computer Science, SUNY, Stony Brook, New York, November, 1980.

[MaiWa80b] Maier, D., and D. S. Warren. *Incorporation Computed Relations in Relational Databases*, Technical Report 80/17, Department of Computer Science, SUNY, Stony Brook, New York, December, 1980.

[Minsky81] Minsky, M. A Framework for Representing Knowledge. In *Mind Design*, J. Haugeland (ed), MIT Press, Cambridge, MA, 1981.

[NeuSton89] Neuhold, E., and M. Stonebraker Future Directions in DBMS research. in *SIGMOD Record*, 18(1), 1989.

[Page89] Page Jr. T. W. An Object-Oriented Logic Programming Environment for Modelling. *Ph.D. dissertation*, UCLA, Los Angeles, 1989.

[Przym88] Przymusinski, T. C. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker (ed), Morgan-Kaufmann, Los Altos, California, pp. 193–216, 1988.

[Przym89] Przymusinski, T. C. Every Logic Program Has a Natural Stratification and an Iterated Least Fixed Point Model. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1989.

[PrzPrz88] Przymusinski, H., and T. C. Przymusinski. Weakly Perfect Model Semantics for Logic Programs. In *Proceedings of The 5th International Information Systems and Logic Programming*, pp. 1106–1120, Seattle, 1988.

[Reit78] Reiter, R. On Closed World Databases. In *logic and Databases*, H. Gallaire and J. Minker (eds), New York, Plem Press, 1978.

[Tour86] Touretzky, D. S. *The Mathematics of Inheritance*, Morgan-Kaufmann, Los Altos, California, 1986.

[Ullm87] Ullman, J. D. Database Theory: Past and Future. *Proc. of the ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 1–10, 1987.

[Ullm88] Ullman, J. D. *Principles of Database and Knowledge-Base Systems.* Vol. I, Computer Science Press, Rockville, Maryland, 1988.

[Ullm91] Ullman, J. D. A Comparison Between Deductive and Object-Oriented Database Systems. In *LNCS # 566: Deductive and Object-Oriented Databases, Second International Conference on DOOD 1991 Proceedings*, D. Delobel, M.Kifer, Y. Masunaga (eds), Munich, Germany, December 1991, Springer-Verlag, Berlin, 1991.

Appendix A

The New F-Logic

A.1 Introduction

When we finished the design of our proposed logic (see Chapter 3), a new version of F-logic [KifLaWu94] appeared (also see [illegible]nus94]). From now on, we call this new version of F-logic as NF-logic to avoid confusion. We will summarize this NF-logic although it was not used as the basis of our proposed logic, as it is useful for a comparison study.

The main distinguishing property of NF-logic from its two previous versions [KifLa89, KifLaWu90] is the additional membership-relationship assertions between a class and its instances. This additional relationship is also the main distinguishing property between our proposed logic and the previous versions of F-logic [KifLa89, KifLaWu90]. However, there are still some important distinguishing properties between our proposed logic and NF-logic (see Section 5.2).

A.2 Syntax

The alphabet of an F-logic language, L, consists of:

- a set of object constructors F;
- an infinite set of variables V;
- the usual logical connectives and quantifiers, $\vee, \wedge, \neg, \leftarrow, \forall, \exists$;
- a set of auxiliary symbols: $(,), [,], \rightarrow, \twoheadrightarrow, \bullet\!\!\rightarrow, \bullet\!\!\twoheadrightarrow, \Rightarrow, \Rightarrow\!\!\!\!\Rightarrow$, etc.

Object constructors play the role of function symbols that have arities ≥ 0. An *id-term* is the usual term in Predicate Calculus composed of function symbols and variables. The set of all ground id-terms is denoted by $U(F)$, which is commonly known as the *Herbrand Universe*. Ground id-terms are considered to play the role of logical object identities, i.e., a logical abstraction of the implementation concept of physical object identities.

A.2.1 Molecular Formulae

A language of NF-logic consists of a set of NF-formulae that are formed from the alphabet symbols. NF-Formulae are constructed from simpler NF-formulae by using the usual connectives and quantifiers. The simplest kind of NF-formulae are called *molecular NF-formulae* (or simply *NF-molecules*).

Definition An NF-molecule in NF-logic is one of the following:

(1) an *is-a assertion* of the form $C :: D$ or of the form $O : C$, where C, D, and O are id-terms.

(2) An NF-molecule of the form

O[semicolon-separated list of *method expressions*]

A *method expression* can be either *a non-inheritable data expression, an inheritable data expression,* or *a signature expression*, as follows:

- *Non-inheritable data expressions* have one of the following two forms:
  - A non-inheritable scalar expression ($k \geq 0$):

$$ScalarMethod@Q_1, \ldots, Q_k \rightarrow T$$

  - A non-inheritable *set-valued* expression ($l, m \geq 0$):

$$SetMethod@Q_1, \ldots, Q_k \twoheadrightarrow \{S_1, \ldots, S_m\}$$

- *Inheritable* scalar and set-valued data expressions are similar to non-inheritable expressions except that "$\rightarrow$" is replaced with "$\bullet\!\!\rightarrow$" and "$\twoheadrightarrow$" is replaced with "$\bullet\!\!\twoheadrightarrow$".
- Signature expressions also take two forms:
  - A *scalar* signature expression ($n, r \geq 0$):

$$ScalarMethod@V_1, \ldots, V_n \Rightarrow (A_1, \ldots, A_r)$$

  - A set-valued signature expression ($s, t \geq 0$):

$$SetMethod@W_1, \ldots, W_n \Rrightarrow (B_1, \ldots, B_t)$$

In (1), the first isa assertion $C :: D$ states that C is a nonstrict subclass of D, i.e., including when C and D denote the same class. The second isa-assertion $O : C$ states that O is a member of class C. It should be noted that NF-logic allows a class to be a member of itself.

In (2), O is an id-term that denotes an object. ScalarMethod and SetMethod are also id-terms. The syntactic position of ScalarMethod points out that it is invoked on O as a scalar method. Similarly, the syntactic position of *SetMethod* points out a set-valued invocation. When a *ScalarMethod* or a *SetMethod* contains variables, it denotes a family of methods instead of a single method. The single-headed arrows,

$\rightarrow$, $\bullet\!\!\rightarrow$, and $\Rightarrow$, point out that the corresponding method is scalar. On the other hand, the double-headed arrows, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, and $\Rightarrow\!\!\!\!\!\Rightarrow$, point out that *SetMethod* denotes a set-valued function.

The terms T and S_i, in data expressions, are id-terms rer esenting the result of *ScalarMethod* and *SetMethod* respectively, when they are invoked on the host-object O with the arguments $Q_1, \ldots, Q_k$ and $R_1, \ldots, R_l$, respectively. These arguments are id-terms.

A_i and B_j, in signature expressions, are id-terms that represent the *types* of the results returned by the corresponding methods when invoked on an object of a class C with arguments of types $V_1, \ldots, V_n$ and $W_1, \ldots, W_s$, respectively. These arguments are id-terms. The notation $(\ldots)$ in signature expressions means that the output of the method must belong to *all* the classes listed on the right of " $\Rightarrow$ " or "$\Rightarrow\!\!\!\!\!\Rightarrow$".

The order of data and signature expressions in an NF-molecule is immaterial. NF-logic allows the same method and the same data/signature expressions to have multiple occurrences in the same NF-molecule. Similarly, the same id-terms may occur multiple times inside the braces in a data or signature expression. For convenience, the symbol "@" may be omitted when a method does not expect arguments. For example, we may write $P[Meth \rightarrow Val]$ instead of $P[Meth@ \rightarrow Val]$. Furthermore, if only one element appears inside { }, we may omit the brackets. For example, we may write $P[\ldots \twoheadrightarrow S]$ instead of $P[\ldots \twoheadrightarrow \{S\}]$.

A.2.2 Complex Formulae

NF-formulae are defined as follows.

- Molecular NF-formulae are NF-formulae;

- $\phi \vee \psi$, , $\phi \wedge \psi, \neg\phi$ are NF-formulae, if ϕ and ψ are NF-formulae;
- $\forall X\, \phi, \exists Y \psi$ are NF-formulae, if ϕ, ψ are NF-formulae and X, Y are variables.

Furthermore, a *literal* is defined as either a molecular NF-formula or a negation of a molecular NF-formula.

A.3 Semantics

The following notation shall be used in presenting the semantics. Given a pair of sets U and V, the notation $Total(U, V)$ is used for denoting the set of all total functions $U \rightarrow V$. The notation $Partial(U, V)$ denotes the set of all partial functions $U \rightarrow V$. The power-set of U is denoted by $\mathcal{P}(U)$. In addition, given a collection of sets $\{S_i\}_{i \in \mathcal{N}}$ parameterized by natural numbers, $\prod_{i=1}^{\infty} S_i$ will denote the Cartesian product of the S_i's.

A.3.1 NF-Structures

Semantic structures in NF-logic are called *NF-structures*. Given a language of NF-logic, L, an *NF-structure* is a tuple

$$I = \langle U, \prec_U, \in_U, I_F, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\!\rightarrow}, I_{\bullet\!\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rrightarrow} \rangle$$

.

Where U is the domain of I, $\prec_U$ is an irreflexive partial order on U, and $\in_U$ is a binary relation. The notation $a \preceq_U b$ means $a \prec_U b$ or $a = b$. The relation $\preceq_U$ and $\in_U$ is extended to tuples over U, Given $\vec{u}, \vec{v} \in U^n$ and $S \subseteq U^n$, we write $\vec{u} \preceq_U \vec{v}$ or $\vec{u} \in_U \vec{v}$ if the corresponding relationship holds between $\vec{u}$ and $\vec{v}$ component-wise.

The ordering $\prec_U$ on U is a semantic counterpart of the subclass relationship. That is, $a \prec_U b$ means that a is a subclass of b. The binary relation $\in_U$ is used to model class membership, i.e., $a \in_U b$ means that a is a member of class b. The relationship between $\prec_U$ and $\in_U$ is as follows: If $a \in_U b$ and $b \preceq_U c$ then $a \in_U c$.

NF-logic *does not put any other restriction on the class membership relation* ($\in_U$). Especially, $\in_U$ **does not have to be acyclic**. For example, $s \in_U s$ is allowed. The expression $u \in_U v$ means that v is an element of U that denotes a subset of U, and u is member of this subset.

U can be viewed as a set of all actual objects in a possible world I. Ground id-terms (i.e., the elements of $U(F)$) play the the role of logical object id's. These ground id-terms are interpreted by objects in U through the mapping $I_F : F \mapsto \bigcup_{i=0}^{\infty} Total(U^i, U)$. This mapping interprets every k-ary object constructor, $f \in F$, by a function $U^k \mapsto U$. For $k = 0$, $I_F(f)$ is identified with an element of U.

The remaining symbols in I denote mappings for interpreting the six types of method expressions in NF-logic.

A.3.2 Attachment of Functions to Methods

In NF-logic, id-terms are also used to denote methods. A method is a function that takes a host-object and a list of proper arguments and maps them into another object or a set of objects depending on whether the method is invoked as a scalar or a set-valued function.

As mentioned before, NF-logic can use any id-term for a method name. In order to allow variables to range over methods, NF-logic associates functions with each element of U instead of $U(F)$ (a set of ground id-terms). Moreover, because methods can have different arities, NF-logic needs to associate a function with an arity.

Formally, objects that play the role of methods are interpreted through an assignment of appropriate functions to each element of U, by using the mapping $I_{\to}, I_{\bullet\!\to}, I_{\twoheadrightarrow}$ and $I_{\bullet\!\twoheadrightarrow}$.

For each object, its role as a scalar method is obtained through the mapping:

- $I_{\to}, I_{\bullet\!\to} : U \mapsto \prod_{k=0}^{\infty} Partial(U^{k+1}, U)$

Each of these mappings associates a tuple of partial functions $U^{k+1} \mapsto U$ with every element of U; there is exactly one such function in the tuple, for each method of arity $k \geq 0$. This means that the same method can be invoked with different arities.

Furthermore, every method can be invoked as a scalar or as a set-valued function. This is achieved semantically by interpreting methods playing the role of set-valued methods through the mapping:

- $I_{\twoheadrightarrow}, I_{\bullet\!\twoheadrightarrow} : U \mapsto \prod_{k=0}^{\infty} Partial(U^{k+1}, \mathcal{P}(U))$

Each of these mappings associates a tuple of partial functions $U^{k+1} \mapsto \mathcal{P}(U)$ with every element of U.

The difference between $I_{\to}$ and $I_{\bullet\!\to}$ in the mapping above is that the " $\to$ " versions are used to interpret non-inheritable data properties, while "$\bullet\!\to$ versions are used to interpret the inheritable ones.

From the above definitions, for an $m \in U$, $I_{\to}(m)$ (or, $I_{\twoheadrightarrow}(m)$, $I_{\bullet\!\to}(m)$, $I_{\bullet\!\twoheadrightarrow}(m)$) is an infinite tuple of partial functions parameterized by the arity $k \geq 0$.

The following notation is used to refer the k-th component of such a tuple: $I^{(k)}_{\to}(m)$ (resp., $I^{(k)}_{\twoheadrightarrow}(m), I^{(k)}_{\bullet\!\to}(m)$, or $I^{(k)}_{\Rightarrow}(m)$). Therefore, a method m which occurs in a scalar non-inheritable data expression with k proper arguments is interpreted by $I^{(k)}_{\to}(m)$. Similarly for the other three types of occurrences. Notice that $I^{(k)}_{\to}(m)$ and the other three mappings are $(k+1)$-ary functions. The first argument is the host object, the other k arguments correspond to the *proper* arguments.

A.3.3 Attachment of Types to Methods

Since methods are interpreted as functions, NF-logic interprets a signature expression as a functional type, for specifying the type of a method. The functional type describes the types of the arguments to which the function can be applied and the types of the results returned by the function.

Functional types are described, model-theoretically, through the mappings $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$ as follows:

- $I_{\Rightarrow} : U \mapsto \prod_{k=0}^{\infty} PartialAntiMonotone_{\prec_U}(U^{k+1}, \mathcal{P}_{\uparrow}(U))$
- $I_{\Rightarrow\!\!\!\Rightarrow} : U \mapsto \prod_{k=0}^{\infty} PartialAntiMonotone_{\prec_U}(U^{k+1}, \mathcal{P}_{\uparrow}(U))$

Where $\mathcal{P}_{\uparrow}(U)$ is a set of all upward-closed subsets of U. A set $V \subseteq U$ is *upward closed* if $v \in V$ and $v \prec_U v'$ (where $v' \in U$) imply $v' \in V$. When V is viewed as a set of classes, upward closedness simply means that, for each class $v \in V$, the set V also contains all the superclasses of v. $PartialAntiMonotone_{\prec_U}(U^{k+1}, \mathcal{P}_{\uparrow}(U))$ denotes the set of partial *anti-monotonic* functions from U^{i+1} to $\mathcal{P}_{\uparrow}(U)$). For a partial function $\rho : U^{k+1} \mapsto \mathcal{P}_{\uparrow}(U)$), *antimonotonicity* means that if $\vec{u}, \vec{v} \in U^{k+1}$, $\vec{v} \preceq_U \vec{u}$, and $\rho(\vec{u})$ is defined, then $\rho(\vec{v})$ is also defined and $\rho(\vec{v}) \supseteq \rho(\vec{u})$.

A.4 Discussion

In subsection we discuss the properties of $I_{\Rightarrow}$ and $I_{\Rightarrow\!\!\!\Rightarrow}$. Similar to $I_{\rightarrow}$ we use $I_{\Rightarrow}^{(k)}(m)$ to refer to the k-th component of the tuple $I_{\Rightarrow}(m)$. Similarly for $I_{\Rightarrow\!\!\!\Rightarrow}^{(k)}(m)$ which is used for set-valued methods.

$I^{(k)}_{\Rightarrow}(m)$ is intended as the type of the $(k+1)$-ary function $I^{(k)}_{\rightarrow}(m)$. This means that the domain of definition of $I^{(k)}_{\Rightarrow}(m)$ is viewed as a set of $(k+1)$-tuples of classes, $\langle \boldsymbol{hostcls}, \boldsymbol{argtype}_1, \ldots, \boldsymbol{argtype}_k \rangle$, that type tuples of arguments, $\langle \boldsymbol{obj}, \boldsymbol{arg}_1, \ldots, \boldsymbol{arg}_k \rangle$, on which $I^{(k)}_{\rightarrow}(m)$ can be invoked correctly. For every tuple of classes, $\langle cls, \vec{\boldsymbol{types}} \rangle \in U^{k+1}$, if $I^{(k)}_{\Rightarrow}(m)(cls, \vec{\boldsymbol{types}})$ is defined, it represents the type of $I^{(k)}_{\rightarrow}(obj, \vec{\boldsymbol{args}})$ for any tuple of arguments such that $\langle obj, \vec{\boldsymbol{args}} \rangle \in_U \langle cls, \vec{\boldsymbol{types}} \rangle$. This means that if $v = I^{(k)}_{\rightarrow}(m)(obj, \vec{\boldsymbol{args}})$ is defined it must belong to every class in $I^{(k)}_{\Rightarrow}(m)(cls, \vec{\boldsymbol{types}})$. Similarly for the meaning of $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}$ as the type of the set-valued function $I^{(k)}_{\twoheadrightarrow}$.

It should be noted that the above relationship between $I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow}$ and $I_{\rightarrow}, I_{\twoheadrightarrow}$ is *not* part of the definition of NF-structures. Similarly the relationship between $I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow}$ and $I_{\bullet\!\rightarrow}, I_{\bullet\!\twoheadrightarrow}$ is not part of the definition of NF-structures. These relationships are defined at a meta level of NF-logic.

It should be noted again that this new F-logic[KifLaWu94] was not used as the basis of the development our object-oriented logic, since we found out about it when the design of our logic was done (also see [Junus94]). The kinds of F-logic that we used as the basis are [KifLa89] and [KifLaWu90].