INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
On integration of object-oriented features with deductive data language

Lou, Yanjun, Ph.D.
Case Western Reserve University, 1992
ON INTEGRATION OF OBJECT-ORIENTED FEATURES
WITH DEDUCTIVE DATA LANGUAGE

by
YANJUN LOU

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: Dr. Z. Meral Ozsoyoglu

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY

May 1992
CASE WESTERN RESERVE UNIVERSITY
GRADUATE STUDIES

We hereby approve the thesis of

YANJUN LOU

candidate for the PHD
degree.*

Signed: [Signature]
(Chairman)

[Signature]

[Signature]

Date Nov 22, 1991

*We also certify that written approval has been obtained for any proprietary material contained therein.
I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

[Signature]
ON INTEGRATION OF OBJECT-ORIENTED FEATURES WITH DEDUCTIVE DATA LANGUAGE

Abstract

by

YANJUN LOU

Integrating value-oriented and object-oriented data models is one of the active research directions in database field. In this thesis, we first discuss the properties of an object-oriented data model that combines the simplicity of relational data model and features of object-oriented data models. Meta variables are introduced for information hiding, data abstraction, and type inheritance. Partial order is defined on classes as well as on types to describe inheritance relationship. The concept of named values bridges the concepts of values in logical models and objects in object-oriented models. With union values, values can be formed from its subset values. Semantics and key features of the model are also discussed.

Next, we consider Horn clause programs from an object-oriented perspective. Current Horn clause languages (Prolog/Datalog) do not have the concept of methods and there is no polymorphism. However, their declarative paradigm and formal theory have won enormous popularity. Our observation is that Horn clause languages can be extended and object-oriented features can be integrated with the support of our data model. Based on this observation, we present a rule-based deductive data language, LLO, which has object-oriented features such as object identities, polymorphism and encapsulation. Methods are defined by rules and method inheritance is achieved through typing and unification mechanisms. Datalog and nested relational models are shown to be subsets of LLO. Semantics of the language is analyzed and LLO as a full-fledged programming language are addressed. A formalization of the subset semantics is given and some results of fixpoint semantics of LLO programs are presented.
Acknowledgments

First and foremost, I would like to express my sincerest appreciation to my advisor Dr. Z. Meral Ozsoyoglu for her help, guidance, support and encouragement in all facets of this research, and especially, her influence to my academic and professional growth.

Next, I would like to thank Dr. George W. Ernst, Dr. Miles H. Kennedy and Dr. Leon S. Sterling for being on my committee, and for their time, enthusiasm and valuable suggestions.

I would also like to use this opportunity to express my appreciation to all the members of the database group, especially Dr. Tekin Ozsoyoglu, Kaizheng Du, Naci Ishakbeyoglu, Atakan Kurt and Jian Wang. Their friendship makes my study at CWRU interesting and enjoyable.

I am deeply grateful to my mother, my wife and my daughter for their love, encouragement, and support. Thanks to my parents-in-law, who take care of my little daughter for the whole period of my study.

My study at Case Western Reserve University is made possible by a scholarship from the State Education Commission of the People’s Republic of China, a graduate assistantship from the Computer Engineering and Science Department of Case Western Reserve University, and a research assistantship through the National Science Foundation under Grants IRI88-11057, IRI90-08632, and FAW Award IRI90-24152.
Contents

1 Introduction .............................................................................. 1
   1.1 The Impedance Mismatch .................................................. 1
   1.2 Basic Concepts of Object-Oriented Databases .................... 3
   1.3 Deductive Databases .......................................................... 6
   1.4 Problem Solved ................................................................. 8
   1.5 Related Work .................................................................. 12
   1.6 Overview ....................................................................... 16

2 Object-Orientation in Horn Clause Framework ......................... 19
   2.1 Complex Object Modeling ............................................... 19
   2.2 Methods on Individual Objects ......................................... 23
   2.3 Methods on Classes .......................................................... 23
   2.4 Method Inheritance .......................................................... 25
      2.4.1 The Data Model Revisited ......................................... 26
      2.4.2 Method Inheritance .................................................... 27

3 The Data Model .................................................................... 28
   3.1 Meta Variables in Data Model .......................................... 28
      3.1.1 Types ...................................................................... 29
      3.1.2 Values .................................................................... 33
      3.1.3 Named Values, Objects and Classes .............................. 34
3.1.4 Databases ......................................................... 36
3.2 Semantics ......................................................... 38
  3.2.1 The Implicit Class Associated with a Type ............... 39
3.3 Inheritance by Partial Order .................................... 40
  3.3.1 Semantics of Inheritance .................................. 40
  3.3.2 Syntactical Descriptions of Inheritance .................. 40
3.4 Features of the Data Model ..................................... 41
  3.4.1 Named Values: Bridge between the Value-Oriented Model and Object-Oriented Model .................................. 41
  3.4.2 Meta variables: Information Hiding and Data Abstraction 42
  3.4.3 New Features of Inheritance ............................... 43
3.5 Summary .......................................................... 44

4 The Underlying Logic .............................................. 46
  4.1 The Syntax ...................................................... 46
    4.1.1 Id-terms .................................................. 47
    4.1.2 Terms .................................................... 47
    4.1.3 Some Considerations .................................... 48
    4.1.4 LLO Types and Declarative Formulae ...................... 49
  4.2 Semantics ...................................................... 52
  4.3 Relational Programming in LLO ................................. 56
    4.3.1 Relational Model ......................................... 56
    4.3.2 Nested Relational Model ................................ 57
    4.3.3 Function Id-terms as Relation Names ..................... 58

5 Type Checking in LLO .............................................. 60
  5.1 Basic Concepts ................................................ 60
  5.2 Well Typing in LLO ........................................... 61
  5.3 Type Inference ................................................ 65
6 LLO As A Programming Language 67
6.1 Horn Clauses in LLO ............................. 67
6.2 Herbrand Interpretations .......................... 73
6.3 Unification and Query Evaluation .................. 76

7 Methods and Method Inheritance 83
7.1 Rules as Methods .................................. 83
7.2 Generic Methods ................................... 85
7.3 Identity Denotation and Population ................ 88
7.4 Method Inheritance ................................ 93
7.5 Multiple Inheritance: A Discussion ................. 97
7.5.1 More Than One Methods Match with a Message .. 98
7.5.2 Multiple Inheritance ............................. 99
7.6 Summary ........................................... 101

8 More on Programming in LLO 102
8.1 Methods and States ................................. 102
8.1.1 Defining Methods on Objects Directly .......... 103
8.1.2 States of Objects and Features ................ 104
8.1.3 Common Components for Objects in One Class .... 104
8.2 Externals ......................................... 105
8.3 Query about Type Information ...................... 106
8.4 Integration of Functional Programming and Logic Programming 108

9 Fixpoint Semantics 110
9.1 Introduction ...................................... 110
9.1.1 Intersection of Models ........................ 111
9.1.2 An Example .................................... 113
9.1.3 Subset Semantics ............................... 114
9.1.4 Our Approach .................................................. 116

9.2 Semi-Lattices on Terms, Atomic Literals and Interpretations ... 120
   9.2.1 Partial Order on Terms .................................. 120
   9.2.2 Partial Order on Atomic Literals ....................... 122
   9.2.3 Partial Order on Interpretations ....................... 123

9.3 Subset Semantics of Basic \( LLO^+ \) Programs ................. 125
   9.3.1 Equal Set Semantics .................................. 126
   9.3.2 Subset Semantics Based on Semi-Lattices .............. 126

9.4 Subset Semantics of \( LLO^+ \) .................................. 129

9.5 Program Subset Semantics in the Equal Set Semantics Framework 132
   9.5.1 Problems with Subset Semantics ....................... 132
   9.5.2 Subset Join: Join by Greatest-Lower-Bounds .......... 133
   9.5.3 Inference by Subset .................................. 135

9.6 Semantics of Stratified LLO Programs ........................ 138
   9.6.1 Stratification ....................................... 138
   9.6.2 Procedural Semantics of Stratified Programs .......... 142

10 Conclusions .................................................. 145
Chapter 1

Introduction

In this chapter, we first discuss the so called impedance mismatch between programming languages and data manipulation languages. There are many efforts in overcoming this mismatch. Then we briefly review basic concepts of object-oriented databases and logical data languages. The contribution of this thesis is summarized in Section 1.4. In Section 1.5, we give a brief survey on works related to our research. Finally, we give an overview of the organization of this thesis.

1.1 The Impedance Mismatch

In databases, the "impedance mismatch" between data manipulation language (DML) of the database, and the general purpose programming language in which the application is developed has inspired intensive research activities. As has been pointed out in [ZdMa 90], there are two aspects of this mismatch. One is the difference in programming paradigms; for example, between a declarative DML such as SQL and the imperative programming language such as C. Another aspect is the mismatch of type systems. Some information loss will occur during the structural (type) transformation, if the programming languages do not have a matching type (i.e. relation) in the relational model. The cause for this mismatch is the computational incompleteness of the DML, which makes it necessary to
embed a database application in a more expressive language.

The research efforts in solving this notorious problem are focused on two approaches. 1). To enhance the expressive power of DML and make it more expressive. 2). To enrich the data type of general purpose programming languages and add more database features. A new field, Database Programming Language, has been popular among many database researchers.

Object-oriented database (OODB) is one of the active research areas in solving this impedance mismatch. OODB adopts the effective techniques developed in software engineering, programming language research and artificial intelligence and provides complete computational DML. Examples include abstract data types (ADT's), abstraction and encapsulation, specialization and classification, and delegation of behavior, to name a few. Many experts have predicated that OODB will, most probably, become the dominate data model of the next generation databases the same way as happened to relational data model in the 70s and 80s. Unfortunately although there are some consensus on the most important features of the OODB systems, much of the terminology used is overloaded [AbKa 89b] and experts still do not have a commonly accepted definition on those features. A formal model, such as the one proposed for relational data model by Codd [Codd 70], is still missing and such a model is crucial for its success. As pointed out in the [ABDDMZ 89], three points characterize this field at this stage:

1. The lack of a common data model.
2. The lack of a formal foundation.
3. Strong experimental activities, such as $O_2$ [Ban+ 88], Orion [BKK 88], and GemStone [MaSt 87].

On the other hand, the deductive approach has gained enormous popularity both in database field and programming languages, represented by Prolog and its derivatives. Logic languages have the support of theoretical foundations and formalization (Relational algebra is such an example). The computational completeness of logic languages shows that they are capable of overcoming the
mismatch problem if they can support complex object data models. But the current deductive data languages, represented by Datalog, lacks complex structures, which are crucial in modeling applications in CAD/CAM, CASE and office management. Traditionally, logical models are considered value-oriented [Ullm 88] and lack facilities such as classification, encapsulation and abstraction, and other good features important to an object-oriented system.

From the above analysis, it can be seen that integrating object-oriented features with deductive languages can be a solution to the impedance mismatch problem, with the support of the logical semantics and better programming practice in object-oriented system. Many results have been reported on combining those two approaches in the literature. In this thesis, we focus our attention on object-oriented data modeling and operational semantics of complex objects, especially sets. A new approach of object and class modeling will be proposed and a deductive DML, called LLO, will be presented.

1.2 Basic Concepts of Object-Oriented Databases

This section explains the most important concepts in object-oriented systems. The discussion follows the pervaded consensus in the literature, especially [ABDDMZ 89, Kim 90, ZdMa 90, Wegner 90].

An object-oriented database system (OODBS) is a database system which supports an object-oriented data model, including a number of concepts found in object-oriented programming languages [Kim 89, Kim 90].

An object is an abstract machine that defines a protocol through which users of the object may interact [ZdMa 90]. The state of the object is encapsulated and only the implemented protocol has access to it. Information hiding through defined interface of the object is a crucial feature of an object-oriented language.

The protocol of the object consists of a set of methods with typed signatures. One method may have several different implementations which are hidden from the users and have special privileges to access the state of the object. A message is a method call which is sent to an object. A method is selected to respond to
the message according to the signatures of the methods. This method selecting process is called *message dispatching*. The responding method carries out some computation and returns the message with derived information from the state of the object. The state of the object can be changed through methods.

Each object has an *object identity* (oid), that is, a handle of it [KLW 90]. The conceptual counterpart is *object denotation*. The identity of an object allows us to express the notion of shared structures and overcome the update anomalies. By using oid, we can encode cyclicity [KuVa 84]. An oid is immutable. There is no operation that can ever change the correspondence between an object and its oid [ZdMa 90]. Object identity is also called surrogate in [Godd 79], l-value in [KuVa 84] or object identifiers in [KhCo 86]. Object identity has been used in programming languages. Recent research in database programming reaffirmed the importance of object identity database systems. The basic idea is that all access to the object must be made by referring to its oid and its interface. In OODB, two objects in one class can have the same complex state because of the distinguishable object identities. This is impossible in relational model, where two tuples in one relation can not be exactly the same.

In this thesis, we refer to the state of an object as its value (A value here is really the *complex object* in [BaKh 86]). Complex value modeling and manipulation have long been an active research direction as the new applications of databases such as CAD/CAM, CASE etc need more modeling power. The frequently discussed complex value constructors include *set*, *tuple*, *list* and *array*. In object-oriented databases, the complex value constructors must be *orthogonal*: any constructor should be applicable to any value [ABDDMZ 89]. The constructors in relational model are not orthogonal, because the set constructor can only be applied to tuple values and tuple constructor can only be applied to atomic values. Even though the Nested Relational Model [OOM 87, OzYu 87, ScSc 86] allows tuple constructor being applied to non-atomic values, the top level constructor must always be a set (relation). It should be pointed out that *object identities* (oid's) are values and can be parts of other complex values. *Type* is
used to specify the structures of complex values. Complex values can be type-
checked statically at compile time.

There are many similar objects in a database, that is, they share a similar
complex value structure and a common set of methods. The concept of class is
introduced for grouping those objects together and defining a commonly shared
set of methods. Instead of defining methods on individual objects, methods are
defined on classes and shared by each object in the class. In our context, each
class has a type specifying the structure of the complex value of each object. Each
method has an interface part and an implementation part. When a message is
sent to an object, a method is selected to respond to the message. Classes serve
as templates from which objects can be created. We may think of a class as
specifying a behavior common to all objects of the class. The public interface
determines the behavior of the object while the private complex value is used
as its storage [Wegner 90, WeZd 88]. The mechanism of hiding the state of an
object as well as its method implementations is called encapsulation.

Encapsulation is a powerful system structuring technique in which a system
is made up of a collection of modules each accessible through a well defined
interface. In OODB, encapsulation is based on classes. The state of an ob-
ject is shielded off to unexpected intruders. The modularity makes database
maintenance and extension a lot easier.

Another important concept of object-oriented system is inheritance, which
allows reuse of the behavior defined on a class when defining new classes. Sub-
classes of a class inherit methods defined for their ancestor classes. In addition,
they may have their own new methods and new state components. We dis-
tinguish type inheritance (structural or complex value inheritance) and class
inheritance (method inheritance and object inclusion). By inheritance, type hi-
erarchies and class hierarchies can be established. Inheritance classifies classes
(types) in much the same way classes (types) classify objects (complex values).
The ability to classify classes (types) provides greater classification power and
conceptual modeling abilities. Classification of classes (types) may be referred
to as second order sharing, management and manipulation of behavior that complements the first-order management of objects (values) by classes (types) [Wegner 90]. In an inheritance hierarchy, objects (values) of a subclass (subtype) are completely substitutable in contexts that expects objects (values) of the superclass (supertype). As has been pointed out in [ABDDMZ 89], inheritance has two advantages: it is a powerful modeling tool, because it gives a concise and precise description of the world; it helps in factoring out shared specification and implementations in applications.

In [ZdMa 90], they present a threshold model that can be used as a yardstick to determine whether or not a given database system should be considered to be an object-oriented database system.

1. It must provide database functionality, such as persistence, data modeling and query facilities.

2. It must support object identity.

3. It must provide encapsulation. This encapsulation should be the basis on which all abstract objects are defined.

4. It must support objects with complex state. The state of an object may refer to other objects, which in turn, may have incoming references from elsewhere.

In addition, we believe inheritance is an important feature of object oriented system and should be included here. Typechecking is another issue that is missing on the list. For a detailed discussion, see [ZdMa 90].

1.3 Deductive Databases

In this section we give a brief review of the basic deductive data language, namely the Datalog. Our presentation is based on Ullman’s text book [Ullm 88].

Datalog programs are built from atomic formulae, which are predicate symbols with a list of arguments, e.g., \( p(X_1, \ldots, X_n) \). An argument in Datalog can
be either a constant or a variable. Names beginning with lower case letters are used for constants and predicate symbols, while names beginning with upper case letters are used for variables. Each predicate symbol has an arity associated with it. Predicate symbols in Datalog denote relations, where components appear in a fixed order. A reference to a column is only by its position among arguments of a given predicate symbol. A predicate whose relation is stored in the database is called an extensional database (EDB) relation, and one defined by logical rules is called an intensional database (IDB) relation. We also construct atomic formulae with the arithmetic comparison predicates, $=$, $<$, $>$, $\leq$, $\geq$, etc, called build-in predicates, such as $X < Y$. Note that in Datalog, function symbols are not allowed.

**Clauses and Horn Clauses**

A literal is either an atomic formula or a negated atomic formula. Negated atomic formula is denoted by $\neg p(t_1, \ldots, t_n)$, called negated literal. Atomic formula not negated is called positive literal. A clause is a sum (logical OR) or a set of literals. A Horn clause is a clause with at most one positive literal. So a Horn clause takes one of the forms:

1. A single positive literal, i.e., $p(X, Y)$, which we regard as a fact.

2. One or more negative literals, with no positive literal, called query or integrity constraint.

3. A positive literal and one or more negative literals, which is a rule.

A Datalog program is a set of Horn clauses.

The underlying mathematical model of data for Datalog is essentially that of relational model. The meaning of Datalog programs follows the model-theoretical point of view. The expressive power of Datalog is analyzed against that of relational algebra and relational calculus in [Ullm 88]. Different program evaluation strategies are also discussed there.
1.4 Problem Solved

While several deductive query models have been developed in the object-oriented paradigm [AbKa 89b, KiLa 89], more elaboration is needed on how methods are defined on classes and how method inheritance works. Cardelli and Wegner [CaWe 85, Card 88] have developed a formal semantics for multiple inheritance in a functional paradigm. According to Cardelli [Card 88], even though an object in a subclass is an object in its superclass, the extra information contained in objects of subclasses makes structures of objects in the superclass diversified. Several questions need to be answered in order to have an object-oriented database in a deductive framework. For instance,

1. There are structural mismatches of a class and its subclasses. Deductive languages, such as Horn clauses, lack the ability to deal with inheritance.

2. There is no corresponding concept of method as in procedural languages.

In this dissertation, we will discuss a data model proposed in [LoOz 90a] that will help solving the problem of structural mismatch between a class and its subclasses. Then we extend the Horn clauses and introduce a method concept that is comparable to the one in procedural languages. Our work is based on the following observations on the resolution procedure in Horn clauses [Lloy 84]

- A program clause is a method\(^1\). The head of the rule is its interface, and the body is its implementation (here we adopt the method concept given in [AKW 90]).

- Each goal clause (query) is a message.

- Unification algorithm dispatches the message to the proper method.

- The resolution process connects all those methods together and finishes the query evaluation.

\(^1\)According to Kowalski [Kowalski 74]. Each rule in Prolog is a procedure.
Example 1.4.1 The following program in Datalog computes the transitive closure from relation \( r \) and stores the result in \( p \).

\[
p(X, Y) : -r(X, Y)
\]

\[
p(X, Y) : -r(X, Z), p(Z, Y)
\]

Query \( p(X, Y) \)? is a message that will be sent to the database. The method that responds to this message (\( p(X, Y) \)) will be selected by the unification algorithm, and the transitive closure of relation \( r \) will be the answer.

This method (if we call it a method) is defined solely for relation \( r \). If we have another relation \( q \) and want to get its transitive closure, we need to write another method. There is no sharing in Datalog at all. \( \Box \)

The above paraphrasing views logic programs from an object-oriented perspective. It is obvious that a predicate name may be seen as an object identifier (intension). The extension of each predicate is the value of that object. A relational data model can be seen as an oversimplified object-oriented system in which relations correspond to objects. That is, an object can only take set of tuples as its value. In this case, either there is no class or each class has only one object. Or, a relation can be seen as a class, the schema of the relation is the type of the class and each tuple is an object. In this case, there is no explicit object identifier. Postgres [StRo 86] takes this approach. Compared with the common features of object-oriented database models [Banc 88, ZdMa 90, Kim 90], we have the following problems when relational model is viewed from an object-oriented perspective:

- No data inheritance;
- No polymorphism, both inheritance and parametric [Card 88]; Methods are defined on constant predicates (or objects) and cannot be shared;
- No encapsulation or abstraction;
- Object identifiers can not be used as components of another object: thus its modeling power is limited.
To overcome these difficulties, much work needs to be done.

- New data models to represent objects. The data model should provide support for a deductive query language with object-oriented features.
- Method definition and method polymorphism. This is crucial for an object-oriented query model.
- Higher order syntax. Object identifiers and sets are allowed to be components of objects, and methods may take objects as arguments.

As has been pointed out, method inheritance polymorphism is the major obstacle in turning logic based models into object-oriented query models. The structural differences between a class and its subclasses makes it difficult for any method on one class to be applied to its subclasses in Horn clause framework. Motivated by this problem, we introduce the concept of meta variables in our data model. Meta variables are used for data abstraction, information hiding and inheritance. Type inheritance hierarchies are built up by meta variable instantiation. Part of the structure of a subtype is hidden by the meta variable in its supertype, which makes all the instances in a type having the uniform structure. All information will be kept when a value or an object is used as a value or an object of its supertype. In other words, type and its subtype have the same structure at a higher level. This feature makes it possible for a rule based language to have method inheritance since the structural barrier between instances of a type and that of its subtypes is removed. Meta variable virtually plays the role of parameter (or meta parameter) in a type.

We also introduce the concept of named values which can be shared through their identifiers and redundancies can be avoided. Named values, when assigned to classes, become objects. Values are allowed in $O_2$ [Ban+ 88, LeRi 89] but do not have identifiers. The role played by values is really limited. They can only appear in objects and are not addressible directly. In IQL [AbKa 89b], in addition to the concept of values, relations are also included as a part of the data model. A relation is actually a set value with an identifier, which makes it accessible, and relational operations are easily applicable. Following
this direction, we introduce the concept of *named values*. A named value is a value with an identifier. A set value with an identifier is a relation. Thus the concept of named values is more general than relations. Not only can we define operations directly on relations, we can also define operations on other types of values.

We introduce the concept of *union value* which allows a value to be formed from its subvalues. The value-subvalue relationship makes it possible to establish a *value hierarchy*. That is, a value inherits data from its subvalues. For example, all the students in a university is a value, and is the union of students from each college. We can operate, access the students in the university by an identifier, and we also can operate and access the students in each college. There is a *data inheritance* relationship between the students in a university and the students in one of its colleges. Union values facilitate binding several values together and organizing smaller values into bigger ones.

Another important result in our model is that the value constructors can be applied *orthogonally*. A value whose top constructor is set is called a set value and the one whose top constructor is tuple is called a tuple value. The states of objects are modeled by values.

In integrating object-oriented features with deductive language, method definition and method inheritance are the key issues. The introduction of meta variables makes objects in a class having the uniform structures. This provides the most important support for a query model with methods and method inheritance.

In this dissertation, we propose a deductive language named LLO for object-oriented databases (Abb. Logic Language for Objects) based on the data model with meta variables. One of the novel features of LLO is that it facilitates defining generic methods as well as concrete ones. That is, in LLO, rules whose left hand side predicate terms start with function id-terms are generic methods, and those that start with constant object identifiers are concrete computation. Utilizing meta variables, subclass objects have the same structure as superclass
objects. Consequently, a method defined on a superclass can be shared by its subclasses. So method inheritance can be achieved uniformly by checking the types associated with methods. What we need to do is to modify the unification algorithm accordingly to incorporate type/class information. Furthermore, encapsulation and abstract data types are realized by function id-terms which are the driving force in defining methods. LLO is an extension of HiLog [CKW 89] with types and other object-oriented features. COL [AbGr 87a] and LDL [Bee+ 87] are subsets of LLO, since data function and grouping operation can be represented in it. By introducing named values and methods, LLO is an extension of IQL [AbKa 89b]. Unlike F-Logic, where concepts are overloaded, LLO strictly separates instances and types, objects and classes.

1.5 Related Work

The commonly praised advantages of relational programming (including relational algebra and Datalog) are set-orientation, view definition facility, declarative style and optimization techniques. Nested relation models [ScSc 86, OOM 87] and complex object models are extensions which try to catch up the requirements of new applications while keeping those advantages. CCO [BaKh 86], LPS [Kuper 87], LDL [Bee+ 87] and COL [AbGr 87a] are efforts in extending Datalog with complex objects. Among all the complex object constructors, set is the major concern. LDL proposed a grouping mechanism to construct set from flat data. In COL, data functions are introduced to construct set values in complex objects. LPS [Kuper 87] is a logic language extended with set terms and universal quantifiers (∀) in the bodies of Horn clauses. It has been proven that LPS and LDL have the same expressive power [Kuper 88, Bee+ 87]. However, as we pointed out before, the complex object constructors in those models are not orthogonal, that is, the top level constructor in any complex object must be a set.

It is well known that the semantic expressiveness of the relational models is limited; they do not provide sufficient mechanism to allow a database schema
to describe the meaning of a database [HaMcL 81]. Such models employ overly simple data structures to model an application environment, causing information loss. Even though the nested relational models and the extensions to Datalog have made considerable progress in semantic data modeling, there is still room for improvement. CCO [BaKh 86] is a data model developed in the logic framework. It is a value oriented model. The whole database can be represented as a single value. Complex object (value) constructors can be applied orthogonally, which is an improvement over the nested relational model. A partial order is defined on complex objects, and a database is organized in a lattice structure. Join operations can be applied on values by subobject relationship, i.e., two values are joinable if they share a common subobject. The problem with CCO is that there is no name for values. Addressing the whole database for each query is simply not practical. The semantic data models starting from E-R model try to develop database design guidelines. The focus of most of the semantic models [HaMcL 81, AbHu 87] is primarily concerned with structures and semantics. Language aspects have not been studied in depth. Most of the object-oriented concepts in OODB can be traced to earlier semantic data models. For example, in SDM [HaMcL 81], class is introduced as a set of entities. Each class has a set of member attributes (methods). An entity in a class can take another entity or a set of entities from another class as its attribute values (complex object). Classes are organized into inheritance hierarchies. A subclass inherits attributes from its superclass. Names (identifiers) are introduced to denote a class or an entity. The IFO model [AbHu 87] incorporates features of most semantic data models: it is an object based model, with aggregation (tuple constructor), classification (set constructor), function (attribute or method), specialization and generalization (inheritance). Furthermore, the IFO model has been formally defined, which simplifies the investigation. In [AbGr 87a], COL is used as a language for IFO. However, some semantics expressible in IFO can not be properly handled in COL, such as specialization and generalization.

Order sorted logic programming is another research direction. Unlike first order logic, where there is only one universe of discourse, order sorted logic divides
the universe into sorts, called structural universes. One sort can be a subsort of another, that is, there is a sort-subsort relationship among sorts. A set of sort symbols with the subsort order imposed on it is called a sort hierarchy. For example, sort boat is a subsort of sort vehicle. During inference about vehicle, boat can be used in place of vehicle. Although order sorted logic has the same expressive power as the first order logic [Walther 84, Walther 89, Schauss 89], its inference efficiency is greatly improved. The unification and resolution problem of order sorted logic are discussed in detail in [Walther 84, Schauss 89]. PROTOS-L [Beierle 89] is a logic programming language based on order sorted logic. Modules are defined in PROTOS-L as an abstraction tool. Order sorted logic achieves inheritance between sorts and subsorts. Uninterpreted function symbols can be seen as a kind of data structures built on the elements of sorts.

Feature logic is based on the results of order sorted logic. The most important characteristics of feature logic is that data about an object is modeled by features, which can be addressed one by one. Some times features are also called methods. Inheritance (orders) and methods (features) are two of the most important characteristics in object-oriented systems. Login [AiNa 86a], O-Logic and F-Logic [Maier 86, KiWu 89, KiLa 89] proposed that an object could address its attributes one by one, or access several attributes together (limiting the schema "from below" by specifying what is generally true about the class). Object is emphasized and attributes of an object are methods defined on it. But as pointed out by Cardelli [Card 88], there are some anomalies with this approach and parameterized types cannot be properly handled. Cardelli and Wegner [CaWe 83] suggested a solution by introducing quantifiers $\exists$ and $\forall$ in types. Even though IQL adopted the semantics of Cardelli [Card 88], it seems only data inheritance is achieved. It is not clear from [AbKa 89b] what is the method and how methods are being inherited among classes and their subclasses. In this paper, we take a different approach to attack this problem. We introduce Meta variables, which, we believe, will overcome some of those difficulties. A meta variable is a type variable, i.e., it takes types as its values. A type with a meta variable as its attribute actually represents a set of types. Utilizing meta variables, type
hierarchies can be built, and inheritance relationship can be established.

Object-oriented algebras [Osbo 89, StOz 90, ScSc 90] and other query languages have been developed using object identity. But the concept of method is ignored. At the same time, F-Logic [KiLa 89] uses nonground id-terms as labels to define methods. In F-Logic, a subclass can also be an instance of the superclass, and classes can be used as objects. Due to this overloading of concepts, it is confusing whether a method defined for a class is defined on its subclass as a whole or on individual objects in the class. There are also restrictions defining methods inside objects and classes. Such operations similar to the join in the relational model do not have intuitive definitions.

LDM [KuVa 84] introduces l-values into database and tries to unify the network, hierarchical and relational data models. An object, like variables in procedural languages, has an l-value (identifier or address) and a r-value (value stored at that address or the value the identifier denotes). With l-values, data can be shared and redundancy can be avoided. More importantly, cyclic data can be constructed easily. But queries can not build l-values themselves, a formula in LDM just says when a database satisfies it, at the same time, gives no way to construct such an object. Invention of object identifiers are addressed in IQL [AbKa 89b] and ILOG [HuY 90] in a deductive framework. However, as we have mentioned earlier, only data inheritance is discussed and there is no clear concept of method.

On the other hand, there have been discussion about utilizing part of higher order logic to handle complex object [Enderton 72]. HiLog [CKW 89], HILLOG-R [ChenGard 88, ChenQ 89, ChKa 91], F-Logic [KiLa 89], O-Logic [KiWu 89], among others, are works along this direction. HiLog [CKW 89] proposed the higher order syntax and first order semantics. Even though it is basically a value-oriented language and did not discuss object-oriented features directly, we find that the mechanisms of defining methods are already developed there and parametric polymorphism is easily implemented. But it did not address how method inheritance can be achieved.
It is commonly expected that object-oriented query models should be compatible with the current database query models in order to utilize the accumulated techniques in this field. COL [AbGr 87a] and LDL [Bee+ 87] are examples of upgrading Datalog with complex objects. IQL [AbKa 89b] and ILOG [HuY 90] are aiming to build a deductive query model with object-oriented features. Login [AiNa 86a] is another example for integrating data inheritance with Prolog. We follow the same route here and try to build a query model that will include current logical models (declarative and conventionally value-oriented) as its subsets.

The relationship among research activities in complex object modeling, multi-paradigm integration and evolution of deductive database systems is shown in Figure 1.1.

1.6 Overview

The rest of the thesis is organized as follows:

In Chapter 2, we briefly review our approach through an example. We explain the motivation for meta variables and present several important concepts, such as object identifiers, method definition and inheritance.

In Chapter 3, we formalize our data model. Concepts such as types, classes, objects and named values are defined. Meta variables and their roles in type inheritance are addressed. A partial order is defined on types as well as on classes. The most important features of the data model are deliberated.

The underlying logic of LLO is presented in Chapter 4. It is shown there that relational model and nested relational model are subset of the logic.

Chapter 5 addresses issues of type checking and type inference. Though strong typing is very important in reducing errors and increasing productivity, type declaration is a burden in many situations. In object-oriented data model, the inheritance and sharing make type inference easy to do.

In Chapter 6, Horn clauses (LLO) are used as programming language. The
Figure 1.1: OODB in a Deductive Framework: A Review
unification algorithms of first order language and order sorted language are extended to higher order typed cases. The main idea is that a term used to substitute a variable must have a type which is a subtype of that of the variable. Unlike first order logic, predicate terms (symbols) must participate the unification and determine the success of message dispatch.

Method definition and method inheritance are addressed in Chapter 7. An LLO rule is a method. If the predicate term in the head of the rule is a constant oid, it defines a concrete method. If the predicate term is a function id-term with variables as its arguments, it defines a generic method. The predicate term in the head of the rule is the interface of the method and its type is the signature of the method. Methods defined on supertypes can be applied to subtypes. Message dispatch is based on signatures of the method and the message.

Chapter 8 addresses other programming issues in LLO, such as its relationship with functional programming and its interface with other programming paradigms. This chapter also discusses how to retrieve type information and method definitions.

The semantic issues of LLO programs are presented in Chapter 9. We point out that there are two kinds of semantics when sets are involved in a language: one is subset semantics and another is equal set semantics. Even though most deductive data languages use the equal set semantics, there are works done to explore the subset semantics. We present a formalization of subset semantics and explain how to program subset semantics in a equal set semantics framework. In this chapter we also extend the stratification concept of Datalog and show that a stratified LLO program has a fixpoint as its minimal model.

Finally, in Chapter 10, we summarize our work and point out some future research directions.
Chapter 2

Object-Orientation in Horn Clause Framework

In this chapter, we describe the proposed extension to Horn clauses (Datalog) and the supporting data organization through an example. The structural part of the data model is presented in Chapter 3 and the operational part, the LLO language, is discussed in Chapter 4.

2.1 Complex Object Modeling

Our example is set in a university environment. Suppose we want to build a database to manage the student records in a deductive framework. Figure 2.1 shows the types in the database that interest us:

- PersonType: [name]
- StudentType: [name, {Course}]
- FacultyType: [name, title]
- GraduateType: [name, {Course}, Faculty]
- CourseType: [coursename, credithour, Faculty, {[Student, grade, {Assignment, grade}]}}]

Figure 2.1: Types Describe Complex Structures
In Figure 2.1, name, title, coursername, credithour and grade are basic types. Course, Faculty, Student and Assignment are class names. Types specify structures of values, which are states of objects. Person type has a name component. Student type has a name, and a set of Courses. Graduate type, in addition to the name and a set of Courses taken, has an advisor. Faculty type, on the other hand, has a name and a title. Course type describes a course offered by a Faculty member, and has a coursername, credithour, a set of students taking this course, each student has a grade and set of assignments.

According to the inheritance discussed in Cardelli [Card 88], a subtype has more information. We have the type inheritance hierarchy shown in Figure 2.2.

Each value in a subtype is a also a value of the supertype. In other words, if a value type-checks in its own type, it should also type-check for any of its supertypes.

An object is composed of an object identifier, a state and a set of methods. While types are used to describe state structures of objects, classes are used to group objects in order to share structures and methods. Each class has a type which specify the structure of objects in this class. For example, Person is a class of type Person type. Student is a class of type Student type, Faculty is a class of type Faculty type. Graduate is a class of type Graduate type and Course is a class of type Course type. The class hierarchy for our example is shown in Figure 2.3.

Note that in our model, the concepts of class and type are defined recursively.
A class is also a type. As shown in Figure 2.1, Course, Faculty, Student there are classes, which means that object identifiers from those classes are components of values in those types. The example also indicates the basic difference between types and classes. By separating the concepts of types and classes, we can model the situation in which more than one classes have the same type but may have different interface (methods).

Every object in a lower class in the class hierarchy is also an object of its parent class. In addition, the type of a subclass must be a subtype of the type of its superclass.

A class has a set of objects. New objects can be created and assigned to a class. At any time, a class can be used as a set value (set of oid's). In IQL [AbKa 89b], classes and relations are the basic operation units. That is not practical for some applications in which we want to send messages to one specific object in a class instead to each and every object in the class. Under this circumstance, it is very helpful to access individual objects through the methods defined on them. Otherwise, we need to create a class for the object accessed, paying extra costs.

Figure 2.4 describes a fragment of the database we are modeling. Here the major concern is focused on structural part, i.e., objects and their values, class-subclass relationship and the grouping of objects into classes.
Objects in class Student: john, mark, carol, harry
Objects in class Graduate: carol, harry
Objects in class Faculty: bogus
Objects in class Person: John, mark, carol, harry, bogus
Objects in class Course: cs341, cs375, cs395, cs411, cs571, cs475, cs380

The following lists some objects and their values:

john: ["john smith", {cs341, cs375, cs411}]
carol: ["carol hengo", {cs571, cs475, cs380, cs411}, bogus]
mark: ["mark wendy", {cs571, cs411, cs395, cs560}]
harry: ["harry strong", {cs341, cs375, cs411}, bogus]
bogus: ["bogus rich", "associate professor"]
cs341: ["introduction to DB"]. 3. bogus. {{john. A. [{ass1, A}, {ass2, B}]}, [mark, W, {}]}
cs411: ["programming in logic"]. 3. bogus. {{john. B+, [{ass4111, A}, {ass4112, C}]},
[carol. A. [{ass4111, B}, {proj411, A}]]}

The class inheritance hierarchy can be specified by the following partial order:

Student ≤ Person
Faculty ≤ Person
Graduate ≤ Student.

It is easy to check that the value of each object conforms to the type of its class.

Figure 2.4: Data Representation: Part of a Database
2.2 Methods on Individual Objects

For the database given above, we ask the following query:

\[ \text{Print the transcript for john.} \]

This query is actually a message, which requires a method defined on object john to respond. The following is such a method:

\[
\text{john.transcript[Name, \{(CourseName, CreditHour, Grade}\}]} : -
\text{john[Name, SetOfCourses], SetOfCourses(X),}
\text{X[CourseName, CreditHour, Instructor, Y], Y(john, Grade, Z).}
\]

Note that for a set term \( X \), \( X(Y) \) means \( Y \) is an element in \( X \). For example, in the above rule, \( \text{SetOfCourses}(X) \) means \( X \) is a course in \( \text{SetOfCourses} \), and \( Y(john, Grade, Z) \) means \( [john, Grade, Z] \) is an element in \( Y \). Remember that \( Y \) here denotes value of set type, i.e., the set of students taking this course, their grades and assignment information. The above rule, if activated, assigns a value to id \( \text{john.transcript} \).

The query now can be formulated as a message:

\[ \downarrow \text{john.transcript}, \]

where \( \text{john.transcript} \) is an oid and will be populated by the method defined on object \( \text{john} \). \( \text{john.transcript} \). The \( \downarrow \) operator will expose the value assigned to \( \text{john.transcript} \), functioning like a printing process.

2.3 Methods on Classes

The method \( \text{john.transcript} \) defined on object \( \text{john} \) can only be used to print john's transcript. If we want to get another student's, let's say, mark's transcript, we need to define another method on object mark. Since the states of both objects have exactly the same structure, two methods will be the same for the same functionality. Even though Datalog has the advantage of computing by set, i.e., applying the same computation to each tuple of the relation at the same time, there is no method definition, method sharing or genericity. Different
programs must be written for the same computation on different relations for the same functionality even though they share the same scheme. (Note that in our model a relation is an object with set value.)

In this thesis, we introduce function id-terms into our query language. One important result in logic programming is that logic language with functions is complete [Ullm 88]. But the introduction of function terms also makes the semantics of logic programs hard to analyze (not terminating if function terms are interpreted). There are function terms in Prolog, called uninterpreted function terms, which are actually names of structures. In this study, instead of introducing function terms in term level, we use function terms as object identifiers, that is, instantiated function id-terms. An id-term has an intensional part and an extensional part [CKW 89]. The intensional part is a denotation of the object, while the extensional part is the state of the object. Thus function id-terms can be used as terms or predicate terms, depending on which part we are interested in. For example, in predicate term \( f(a, b)(X, Y) \), \([X, Y]\) is part of the state of the object denoted by \( f(a, b) \). Here we concerned about the extension of the object. On the other hand, \( f(a, b) \) appearing in \( p(f(a, b), e) \) is an oid, where the intensional part of the object denoted by \( f(a, b) \) is a component of the extensional part of object denoted by \( p \). The later case is like an uninterpreted function term in Prolog. So our language has the higher order syntax and first order semantics (if there is no exposure operation).

By using function id-terms, we can define a method on class \texttt{Student} to get the transcript of a student.

\[
\text{transcript}(S)[\text{Name} . \{ [\text{Course}.\text{Name}, \text{CreditHour}, \text{Grade}] \}] : = \\
S[\text{Name}, \text{SetOfCourses}]. \text{SetOfCourses}(X), \\
X[\text{Course}.\text{Name}, \text{CreditHour}, \text{Instructor}, Y], Y(X, \text{Grade}, Z).
\]

with signature

\[
\text{transcript} : \text{Student} \rightarrow \text{Transcript}(	ext{Student}).
\]

In this method, the variable \( S \) in the head plays the role of passing information from the head to the body. This method is generic and is shared by every
object in class *Student*. Variable *S* is of type *Student* and can be instantiated by any object in *Student*. Now, if we want to get the transcript of mark, we just send the message

\[ \text{transcript(mark)}? \]

to the database. Of course this method will respond only to messages whose arguments are objects in class *Student*.

### 2.4 Method Inheritance

From the class inheritance (see Figure 2.3), it is clear that objects in class *Graduate* are also objects in class *Student*. Since *carol* is an object in class *Graduate*, should the method defined on class *Student* in last section respond to the following message

\[ \text{transcript(carol)}? \]

Sure it should, since *carol* is also a *student*. But *carol* has extra information. She has an advisor. This extra information will make the above message fail, because Horn clauses do not have the facility to handle this situation. In deductive framework, only exact match in structure succeeds in method selection (by unification). So the method defined can only be shared among students who are not graduate students. Horn clauses is a strict language and does not permit any structural flexibility.

There are other problems due to structural differences between a class and its subclasses [Card 88, CaWe 85]. For example, when a subclass object is assigned to a superclass, the extra information in the subclass will be lost.

One solution is feature logic, such as Login [AiNa 86a] and F-Logic [KiLa 89]. The idea is to break the state of an object into “pieces”, or into attributes, and address them one by one. This approach obviously solved the structural obstruction problem. However, it is not appropriate for large data due to efficiency reasons. Further, it is not clear how to properly model the situation when an object takes a set as its state other than a tuple. In addition, it seems that it is
Persontype: \([\text{name, } m_1]\)
Studenttype: \([\text{name, } \{\{\text{Course}\}, \ m_2\}]\)
Facultytype: \([\text{name, title}]\)
Graduatetype: \([\text{name, } \{\{\text{Course}\}, \ Faculty\}]\)

where \(m_1\) and \(m_2\) are meta variables. Their domains are specified as follows:

\[
m_1: \{ \{\text{Course}, \ m_2\}, \ \text{title} \}
\]
\[
m_2: \{ \ \text{advisor} \}
\]

Through meta variables, we have the same inheritance hierarchy in Figure 2.2.

Figure 2.5: Meta Variables and Types

It is not natural to keep the value-oriented feature, that is, relational programming, in the new model. Some kind of coding is need in order to combine those two paradigms [AiNa 86a, KiLa 89].

2.4.1 The Data Model Revisited

Since the problem occurs at the structural level, which is specified by types, it is natural to seek a solution there and extend the type inheritance. Following this line, we come back to the data modeling and introduce meta variables into types and use them as facilities for information hiding and data abstraction.

A meta variable is a type variable and can be used in constructing new types. Each meta variable has a domain, specifying the structures hidden by the meta variable. For the types in Figure 2.1, the extended types with meta variables are shown in Figure 2.5. With meta variables, the extra information in the subtype will be accommodated by the meta variable in the supertype. But the value component corresponding to the meta variable will not be available, i.e., hidden from the supertype. In other words, the value of a subtype and the value of its supertype will have same structure if looked from the supertype point of view.
2.4.2 Method Inheritance

Now objects in a class have a uniform state structure, due to the meta variable in its type. The extra information of objects from subclasses is hidden, or absorbed by the meta variable. However, the method definition still needs to be changed accordingly to comply with the role played by meta variable. More specifically, the access to the extension part (state) of an object should have a corresponding part to the meta variable in its type.

The extended version of the method transcript is given below:

\[
\text{transcript}(S)[\text{Name}, \{\{\text{CourseName, CreditHour, Grade}\}\}] : = \\
S[\text{Name, SetOfCourses, W}], \text{SetOfCourses}(X), \\
X[\text{CourseName, CreditHour, Instructor, Y}], Y(X, Grade, Z).
\]

where \( W \) is a variable of type \( m_2 \), a meta variable. Only values of types in the domain of the meta variable \( m_2 \) can be instantiated with \( W \), and those values are hidden from the current method.

With the support of the extended data model and newly defined method, method inheritance is achieved. Now, we can get carol’s transcript without any structural obstruction by using method transcript defined on class Student. The extra information in carol’s state will be hidden by \( W \).

Actually, we can use the originally defined method as a short hand for the method defined in this subsection. The reason is that the variable \( W \) only play the role as an information absorber. Whatever instantiated with it can not be accessed. Otherwise, the encapsulation rule, or the privacy of the state of an object in a subclass will be broken. So as far as we put a variable of the meta variable type there and this variable does not appear anywhere else in the method, it will be fine.
Chapter 3

The Data Model

In this chapter, the concept of meta variable is discussed — a meta variable is a type variable. Concepts of types, classes, objects and named values are defined. Using meta variables, a partial order is defined on types by which type inheritance hierarchies can be built. One most important aspect of meta variables is that it makes instances of a type and those of its subtypes uniform.

3.1 Meta Variables in Data Model

In this section, we introduce meta variables into our data model. The presentation follows the notation in $O_2$ [Ban+ 88, LeRi 89] and IQL [AbKa 89b]. The reader may find it useful to refer to those papers when necessary.

We assume the existence of the following countably infinite and pairwise disjoint sets of atomic symbols:

1) basic type symbols $B \{B_1, B_2, \ldots \}$, specifically $\bot$ is a basic type symbol;
2) meta variables $\mathcal{M} \{m_1, m_2, \ldots \}$;
3) types $\mathcal{T} \{\tau_1, \tau_2, \ldots \}$;
4) type IDs $\mathcal{TID} \{T_1, T_2, \ldots \}$;
5) class names $\mathcal{C} \{C_1, C_2, \ldots \}$;
6) identifiers $\mathcal{X}_{id} \{p_1, p_2, \ldots \}$.
3.1.1 Types

Types are defined the usual way as in [Ban+ 88, LeRi 89, AbKa 89b]. Basic symbol \( B \in \mathcal{B} \) is a type. A class name \( \mathcal{C} \in \mathcal{C} \) is a type. A type name is a type. And we have tuple types and set types. Especially, a meta variable \( m \in \mathcal{M} \) is a type.

In our presentation, type ID’s and class names start with capital letters and basic types are denoted by bold face symbols. The following are examples of types:

- \([\text{Name}, \text{age}, m_1]\) is a tuple type, where \(\text{Name}\) is a type, \(\text{age}\) is a basic type, and \(m_1\) is a meta variable.

- \([\text{deptname}, \{\text{Personnel}\}, \{\text{Faculty}\}, \{\text{Student}\}, \{\text{TA}\}]\) is a tuple type, where \(\text{deptname}\) is a basic type, \(\text{Personnel, Faculty, Students}\) and \(\text{TAs}\) are class names, \(\{\text{Personnel}\}\) etc. are set types.

If there exist meta variables in a type, we call this type \textit{meta type}. The set of types defined is denoted by \(\mathcal{T}\).

Meta type is a type in which meta variables hide part of the structural information from the current type. So if we specialize the part hidden by the meta variable, we will get another type, that is, a subtype. Thus, the instantiation of the meta variables is the key point in type inheritance.

In this paper, we restrict the number of meta variables in a type to at most one. It is easy to show that multiple meta variables in a type can be represented by types that have only one meta variable. In particular, a conventional type can be seen as a meta type without meta variable.

**Definition 3.1.1** Meta variable instantiation \(\eta\) is a function from \(\mathcal{M}\) to \(2^\mathcal{T}\) such that every meta variable is mapped to a set of types. For a meta variable \(m\), we call its image under \(\eta\), \(\eta(m)\), the domain of \(m\).

Each type in \(\eta(m)\) represents the structure that is hidden by \(m\). There is also a case that a meta variable does not hide anything in the subtype (i.e., the
subtype does not have any attributes corresponding to the meta variable). So, there is at least one element in η(m) to represent this situation. The basic type symbol ⊥ is used for this purpose and is omitted if there is no confusion.

Definition 3.1.1 can be extended to types in a natural way:\footnote{The extension of η is made similar to the extension of σ, the mapping from class names to types in [LeRi 89].}

- **Basic types**: For every basic type τ, η(τ) = \{τ\}.

- **Type IDs**: For every type ID, T, η(T) = η(τ), where τ is the type T denotes.

- **Tuple types**: For every tuple type τ = [τ₁, …, τₙ],

  \[\eta(τ) = \{[τ₁', …, τₙ']|τ₁' \in η(τ₁) \text{ and } … \text{ and } τₙ' \in η(τₙ)\}\].

- **Set types**: For each set type τ = \{τ₁\},

  \[\eta(τ) = \{τ₁'|τ₁' \in η(τ₁)\}\].

The transitive domain of m is denoted by η*:

\[η¹(m) = η(m)\].

\[η^k(m) = \bigcup_{τ \in η^{*−1}(m)} η(τ) \cup η^{k−1}(m)\].

\[η^*(m) = \bigcup_{k=1}^{∞} η^k(m)\].

Let τ₁ and τ₂ be two types from \(\mathcal{T}\). m be a meta variable in τ₁. If τ₂ can be obtained by substituting m in τ₁ by an element from η(m), i.e., τ₂ ∈ η(τ₁), then we call τ₂ subtype of τ₁ and τ₁ is called super-type of τ₂. τ₂ is also called instance type of meta type τ₁. This concept is similar to the partial order defined in [Card 88]. The importance of meta variable will be further addressed in Section 3.4.

The function η is very important in discussing type inheritance. For a finite set of types \(\mathbf{T}\), the interrelationship between types can be visualized through a
Directed Acyclic Graph (DAG). A DAG for a set of types \( T \) is a graph \( < T, E > \),
where each type \( \tau \in T \) is a node, and there is an edge \( e \in E \) from \( \tau_1 \) to \( \tau_2 \) if \( \tau_2 \in \eta(\tau_1) \). We call the DAG's introduced by \( \eta \), type inheritance hierarchies.

The following is an example showing the role played by \( \eta \). Multiple inheritance [Card 88] is now represented by meta variables. The problem addressed in [AbKa 89b], i.e., legal instances with attributes that do not appear in the schema, is virtually eliminated.

Example 3.1.2 (Type hierarchy) Suppose we have the following types:
- \([\text{Name}, \text{age}, m]\),
- \([\text{Name}, \text{age}, [\text{GPA}, m_1]]\),
- \([\text{Name}, \text{age}, m_2, \text{sal}]\),
- \([\text{Name}, \text{age}, [\text{GPA}, \text{sal}]\),

where \( m, m_1 \) and \( m_2 \) are meta variables.

\[
\eta(m) = \{[\text{GPA}, m_1], [m_2, \text{sal}]\},
\]

\[
\eta(m_1) = \{\text{sal}\}, \text{ and }
\]

\[
\eta(m_2) = \{\text{GPA}\}.
\]

Function \( \eta \) can be extended to types, as shown below:

\[
\eta([\text{Name}, \text{age}, m]) = \{[\text{Name}, \text{age}, [\text{GPA}, m_1]], [\text{Name}, \text{age}, [m_2, \text{sal}]\}
\]

\[
\eta([\text{Name}, \text{age}, [\text{GPA}, m_1]]) = \{[\text{Name}, \text{age}, [\text{GPA}, \text{sal}]\}
\]

\[
\eta([\text{Name}, \text{age}, [m_2, \text{sal}]]) = \{[\text{Name}, \text{age}, [\text{GPA}, \text{sal}]\}
\]

We use \( \text{Persontype} \) as a shorthand for \([\text{Name}, \text{age}, m]\), \( \text{Studenttype} \) for \([\text{Name}, \text{age}, [\text{GPA}, m_1]]\), \( \text{Stafftype} \) for \([\text{Name}, \text{age}, [m_2, \text{sal}]\) and \( \text{Tattype} \) for \([\text{Name}, \text{age}, [\text{GPA}, \text{sal}]\). The type inheritance hierarchy is shown in Figure 3.1.

\[\text{Persontype} \quad \rightarrow \quad \text{Studenttype} \quad \rightarrow \quad \text{Stafftype} \quad \rightarrow \quad \text{Tattype}\]

Figure 3.1: Type Hierarchy
It should be pointed out that general type hierarchies can be modeled by meta variables. New elements can be put into or deleted from the domain of a meta variable. New subtypes can be introduced or removed and type hierarchy will change accordingly.

The concept of type in our model is used to model the structural part of data. Every class name is associated with a type. This type is called the schema of the class. Every identifier also has a type. This type restricts the values that the identifier takes.

Let \( C \) be a finite set of class names and \( N \) be a finite set of identifiers. A schema \( \sigma \) is a function from \( C \cup N \) to \( T \).

As in [LeRi 89], this definition can be extended to types in a natural way.

- **Basic types:** for every basic type \( \tau \), \( \sigma(\tau) = \tau \).

- **Type IDs:** For every type ID \( T \), \( \sigma(T) = \sigma(\tau) \), where \( \tau \) is the type \( T \) denotes.

- **Tuple types:** \( \sigma([\tau_1, \ldots, \tau_n]) = [\sigma(\tau_1), \ldots, \sigma(\tau_n)] \).

- **Set types:** \( \sigma(\{\tau\}) = \{\sigma(\tau)\} \).

Schema \( \sigma \) functions as structural abstraction. Applying \( \sigma \) to a type actually brings out the parts in a type hidden by class names and type ID's. Those structures associated with class names represent the structures of objects which the classes can take.

- For class name \( \text{Dept} \), \( \sigma(\text{Dept}) = \{\text{Name}, \text{Deptname}\}, \{\text{Personnel}\}, \{\text{Faculty}\}, \{\text{Student}\}, \{\text{TA}\} \).

- For class name \( \text{Student} \), \( \sigma(\text{Student}) = \{\text{Name}, \text{age}, [\text{GPA}, m_1]\} \).

- Recursive or cyclic structures can also be specified. For a class name \( P \), \( \sigma(P) = \{\text{a}, \{P\}\} \) (see [AlbKa 89b]).

- Let \( o_1 \) be an identifier, then its type is given by: \( \sigma(o_1) = \{\text{Name}, \text{age}, [\text{GPA}, m_1]\} \).
3.1.2 Values

In addition to the values defined in [LeRi 89], we introduce the union values into our model. There are many situations in which one value is the union of several values. Sometimes we need to address the union of the values as a whole (union value) and sometimes we need to use one of them. This concept together with the concept of named values discussed in the next subsection will make it possible to collect a set value from several subset values.

Suppose every basic type $B_i$ has a set of symbols $D_i$ associated with it, and $null$ is the only symbol associated with type $\bot$. $D$ is defined as $D = \cup_i D_i$.

**Definition 3.1.3 (values)**

1. Every element $d \in D$ is a value, called basic value; specifically, $null$ is a basic value.

2. Every object identifiers $p \in \mathcal{N}_{id}$ is a value.

3. If $v_1, v_2, \ldots, v_k$ are values, then $\{ v_1, v_2, \ldots, v_k \}$ is a set value. $v_i$ is called element value of $\{ v_1, v_2, \ldots, v_k \}$. $\{ \}$ denotes the empty set value.

4. If $v_1$ is a value, $\ldots, v_n$ is a value, then $[v_1, \ldots, v_n]$ is a tuple value. $v_i$ is called attribute value of $[v_1, \ldots, v_n]$.

5. If $v_1, \ldots, v_n$ are values, then $v_1 | \ldots | v_n$ is a union value. $v_i$ is called subvalue.

We denote the set of values thus defined as $\mathcal{V}$.

Here the union values are introduced to model the value hierarchy for set values, i.e., a value is composed of its subvalues. If an identifier denotes, directly or indirectly, a set value, it can also be used as a subvalue.

**Example 3.1.4 (cf. Example 3.1.2).**

- $\{ [ ['john', 'smith'], 23, [3.65, 15000]], [ ['bill', 'rich'], 25, [2.90, 12000]] \}$ is a set value of type $\{ \text{Ttype} \}$.

- Let $p_i$ be a set value of type $\{ \text{Ttype} \}$.

  $\{ [ ['don', 'gorden'], 19, [3.82]], [ ['mary', 'king'], 20, [3.25]] \}$
be a set value of type \{Studenttype\}, then
\[ p_1 \mid \{\{\text{'don', 'gorden'} \}, 19, [3.82]\},\{\text{'mary', 'king'} \}, 20, [3.25]\}\]
is a union value of \{Studenttype\}.

- Similarly, let \( p_1 \) be a set value of type \{Tatype\}. Then
\[ p_1 \mid \{\{\text{'harry', 'bull'} \}, 32, [23,000]\}\]
is a union value assigned to \( p_3 \) and finally,

- Let \( p_2 \) be a set value of type \{Studenttype\}, and \( p_3 \) be a set value of type \{Stafftype\}. Then
\[ p_2 \mid p_3 \]
is a union value assigned to \( p_4 \).

\[ \square \]

### 3.1.3 Named Values, Objects and Classes

A **named value** is a value with an identifier. When a named value is put into a **class**, it is called an **object**, and its identifier is the object identifier. **A Named value specifies the structural part of the object.**

A relation in relational model is a named value which is composed of a relation name (identifier) and a set of tuples (tuple values). Instead of modeling relation directly, as has been done in IQL [AbKa 89b], we model named values, which may take tuples, sets or even atomic values. Named values are different from objects in that they can be manipulated directly by operations. This approach combines the advantages of value oriented systems and that of object oriented systems.

Let \( N \) be a finite set of identifiers. A **mapping** \( \delta \) for \( N \) is a partial function from \( N \) to \( V \) such that every identifier \( p \in N \) is assigned a value \( v \in V \).

Each application of \( \delta \) opens one level of encapsulation of values or named values.

- **Basic values:** For every basic value \( v \) which is not an identifier, \( \delta(v) = v \).
  (Remember that oid's are basic values).

- **Set values:** \( \delta(\{v_1, \ldots, v_n\}) = \{\delta(v_1), \ldots, \delta(v_n)\} \) and \( \delta(\{\}) = \{\} \).
• Tuple values: \( \delta([v_1, \ldots, v_n]) = [\delta(v_1), \ldots, \delta(v_n)] \).

• Union values: \( \delta(v_1 | \ldots | v_n) = \delta(v_1) | \ldots | \delta(v_n) \) if there exists \( v_i \), \( \delta(v_i) \) is not a set value; otherwise, \( \delta(v_1 | \ldots | v_n) = \delta(v_1) \cup \ldots \cup \delta(v_n) \).

The mapping of union values deserves more discussion here. Union value is defined for set values. On the other hand, one or more of the subvalues may be identifiers which are mapped by \( \delta \), directly or indirectly, to set values.

Example 3.1.5 (Named value) We continue with Example 3.1.4, where only values are given. Here \( \delta \) associates a value with each identifier. It can be easily seen that \( p_1 \) denotes a set of Tattype values, \( p_2 \) denotes a set of Studenttype values, \( p_3 \) denotes a set of Stafftype values and \( p_4 \) denotes a set of Personotype values. If \( \delta \) is applied more than one time, the named values will be opened gradually.

- \( \delta(p_1) = \{ \{ ['john', 'smith'] \}, 23. [3.65, 15000], [ ['bill', 'rich'], 25. [2.90, 12000] ] \}
- \( \delta(p_2) = p_1 \{ [ [ 'don', 'gorden' ]], 19. [3.82], [ ['mary', 'king'], 20. [3.25] ] \}
- \( \delta(p_3) = p_1 \{ [ [ 'harry', 'bull' ], 32. [23000 ] ] \) and finally,
- \( \delta(p_4) = p_2 \mid p_3 \).

Through \( \delta \) and the union value, we can build named value hierarchies (Figure 3.2).

![Figure 3.2: Named Value Hierarchy](image)

Note: There is no clear agreement on whether a set is an object [Kim 90, ZdMa 90]. In our model, set of objects is not an object². But a set is a value.

²This is consistent with Kim [Kim 90].
which can be assigned to an object. Note that an object must have an oid and a value (state). First of all, \( \{m\} \) is a type (parameterized type), where \( m \) is a meta variable. The domain of \( m \) includes all the relevant types. All set properties such as membership, subset relationship are defined on \( \{m\} \). By the type-subtype relationship defined above, every relevant set type is a subtype of \( \{m\} \) and share its methods. Second, set at a time operation is a well known advantage of Datalog over Prolog. By allowing set as value of an object, the set value will be encapsulated. A computation can be applied to each and every element in the set. Also a class is an object with a set value which is the set of objects assigned to it. We can apply a method of the class to all the objects in the class at the same time. In general, the type of a set is related to the type of the objects that it is allowed to contain [ZdMa 90]. If \( T_2 \) is a subtype of type \( T_1 \), then every instance of \( \{T_2\} \) (a set of value of type \( T_2 \)) is also an instance of type \( \{T_1\} \). For example, a set of graduate students is also a set of students. Both set of students and set of graduate students inherit methods from \( \{m\} \) if they are in the domain of \( m \). In addition, set of graduate students can inherit methods defined on set of students. Instead of defining type-subtype relationship by attribute set [Card 88], in our model the type-subtype relationship between set types is determined by meta variables. This is an alternative solution to the controversy on this topic.

### 3.1.4 Databases

A database schema \( S \) is a tuple \(<N, C, \sigma, \prec>\), where \( N \) is a set of identifiers, \( C \) is a set of class names, \( \sigma \) is a map from \( N \cup C \) to types, and \( \prec \) is a partial order on class names.

**Note:** If an identifier is assigned to a class, it takes the type of the class as its type. In this case, it is not necessary to assign a type to the oid. But a named value, if not assigned to a class, needs to be typed in order to type-check its value. In the definition given above, if an identifier belongs to a class, the type of the identifier should be consistent with the type of the class.
Example 3.1.6 Let Personnel be a class name of type Persontype, Student be a class name of type Studenttype, Staff be a class name of type Stafftype, and TA be a class name of Tatype. Further, we have Student \prec Personnel, Staff \prec Personnel, TA \prec Student, and TA \prec Staff. Then we have a class inheritance hierarchy as shown in Fig. 3.3.

\[
\begin{align*}
\text{Personnel} \quad &\quad \text{Student} \quad \text{TA} \quad \text{Staff} \\
\end{align*}
\]

Figure 3.3: Class Inheritance Hierarchy

Let C be a finite set of class names. An identifier assignment \( \pi \) for C is a function which assigns a finite set of identifiers to each class name in C such that if \( C_1 \prec C_2 \), then \( \pi(C_1) \subseteq \pi(C_2) \).

As discussed before, each class is also an object. For each class name \( C \), there is an oid \( o_C \), \( \delta(o_C) = \pi(C) \), such that the corresponding object is an instance of a class of type \{C\}. If there is no confusion, we also use C as the object identifier for the object corresponding to the class C instead of \( o_C \).

A named value whose identifier is assigned to a class is called the object of that class. For example, a relation in relational model is a named value (set of tuples). A relation name is an identifier which can be assigned to a class. In this case, the relation becomes an object.

A database instance \( I_S \) is a tuple \( < \pi, \delta > \) of schema \( S = < N, C, \sigma, \prec > \), where \( \pi \) is an oid assignment for C, and \( \delta \) is a partial function from N to values.

Due to inheritance, an object may belong to more than one class and a value may have more than one type. That is, the type of an object and a value is a set. If each and every object has a minimum class and each and every value has a minimum type, the data model is called a regular data model. A more formal discussion will be given in Chapter 5.
3.2 Semantics

The interpretation of types associates a set of values to each type. If there is no identifier involved in a value, then we call it pure value.

The closure of $\delta$, $\delta \times \delta \times \ldots$, i.e., repetitive application of $\delta$, is denoted as $\delta^\ast$. For any value or identifier $o$, $\delta^\ast(o)$ is a pure value.

As pointed out in [AbKa 89b], there are some problems using Cardelli’s multiple inheritance semantics. The advantage of meta variables, especially in a deductive framework, is that a type and its subtype have the same structure. Part of the subtype structure is hidden, or more accurately, is abstracted through meta variable.

The interpretation of types is similar to that in [LeRi 89], and is focused on types with meta variables.

Let $\sigma$ be a schema and a $\delta$ be a mapping. An interpretation under $\sigma$ and $\delta$ is a function that associates a set of values $I(\tau)$ to each type, satisfying the following properties:

1. For each basic type $B_i$, $I(B_i) = D_i$

2. For each type ID $T$, $I(T) = I(\tau)$, where $\tau$ is the type that $T$ denotes.

3. For each class name $C$, $I(C) = \pi(C)$.

4. For each tuple type $\tau = [\tau_1, \tau_2, \ldots, \tau_n]$, where $\tau_1, \tau_2, \ldots, \tau_n$ are not meta variables,

$$I(\tau) = \{[v_1, v_2, \ldots, v_n] | \forall i \in \{1, 2, \ldots, n\}, v_i \in I(\tau_i)\};$$

5. For each set type $\tau = \{\tau'\}$, where $\tau'$ is not a meta variable,

$$I(\tau) = \{\{v_1, v_2, \ldots, v_n\} | n \geq 0, 0 \leq i \leq n, v_i \in I(\tau')\},$$

in particular, $I(\{\}) = \{\}$.

6. For each meta type $\tau$, $I(\tau) = \bigcup_{\tau' \in \tau} I(\tau')$ (1).

Equation 1 specifies type inheritance. The union applies to all instance types. This is possible because at the level of $\tau$, there is no structural difference among all instance types since meta variables hide the difference.
There are semantical requirement for the values assigned to an object in a class. That is, for a class \( C \), the value of every object in \( C \) must be an instance of type \( \sigma(C) \). More formally, we have the following equation:

\[
\delta^*(\pi(C)) \subseteq I(\sigma(C)).
\]

### 3.2.1 The Implicit Class Associated with a Type

In [Card 88], type inheritance is implicit, i.e., if the set of attributes in type \( \tau_1 \) includes that of type \( \tau_2 \), then \( \tau_1 \) is a subtype of \( \tau_2 \). In [AbKa 89b], in addition to type inheritance, the hierarchy on classes is introduced to establish inheritance relationship among classes. In this subsection, we want to make a connection between class inheritance and type inheritance, and if possible, set up a uniform characterization. Intuitively, all the named values with the same type should share a common set of methods. For example, if \( \text{john} \) is a named value of type \( \text{Studenttype} \) (here \( \text{Studenttype} \) is a type ID, or a shorthand for a structure), then whatever class it is assigned, it is a student and there are common properties that hold for all students. In the following, a class is defined for each type so that all classes with this type are subclasses of this class. Methods can be defined on this class and shared (inherited) by all classes of this type.

**Definition 3.2.1 (Implicit Class of a Type)** For every type \( \tau \in T \), there is a class name \( C_{\tau} \), such that \( \sigma(C_{\tau}) = \tau \), and

\[
I(C_{\tau}) = \{ o \mid o \in X_{\text{id}}, \delta^*(o) \in I(\tau) \}.
\]

Now, when we are talking about methods defined on types \( \tau \), we mean the methods defined on the class \( C_{\tau} \). Thus in the schema \( < \text{N, C, } \sigma, < > \), the partial order \( < \) is extended by the following rules:

1. For each class name \( C \) with \( \sigma(C) = \tau \): \( C < C_{\tau} \).

2. Any identifier \( p \) with \( \sigma(p) = \tau \) will automatically be an object identifier in class \( C_{\tau} \).
3.3 Inheritance by Partial Order

3.3.1 Semantics of Inheritance

Corresponding to the two interpretations given in the last section, we define the partial order on types. The term refinement used in [LeRi 89] is used here to name the partial order.

Definition 3.3.1 (Partial order on types under I) Type $\tau_1$ is a refinement of type $\tau_2$ under interpretation $I$, denoted as $\tau_1 \preceq_I \tau_2$, if and only if $I(\tau_1) \subseteq I(\tau_2)$. □

Definition 3.3.2 (Partial order on types) Type $\tau_1$ is a refinement of type $\tau_2$ ($\tau_1 \preceq \tau_2$) if and only if for every interpretation $I$, $\tau_1 \preceq_I \tau_2$. □

The word refinement reminds us its meaning in software engineering. Here it carries the same connotation as information hiding. Since the type hierarchies defined are built up through the domains of meta variables, we achieved data abstraction and information hiding.

Example 3.3.3 In Example 3.1.2, Ttype is a refinement of Studenttype. Similarly, Class name TA is a refinement of class name Student in Example 3.1.6. □

3.3.2 Syntactical Descriptions of Inheritance

In Section 3.2, we defined one level type inheritance. The partial order $\prec$ on classes is given by the users. Following the discussion of inference rules and type checking algorithms in [Card 88, CaWe 85], we have a partial order ($\preceq$) on types in general, as shown in the following theorem:

Theorem 1 (syntactical inference rules for types)

1. For any type $\tau$, $\tau \preceq \tau$;
2. For any class names $C_1$ and $C_2$, $C_1 \preceq C_2$ if $C_1 < C_2$ or $C_1 = C_2$;

3. For meta variable $m$ and any type $\tau \in \eta(m)$, $\tau \preceq m$;

4. If $\forall i \in \{1, \ldots, n\}, \tau_i \preceq s_i$, $[\tau_1, \tau_2, \ldots, \tau_k] \preceq [s_1, s_2, \ldots, s_k]$;

5. If $\tau_1 \preceq \tau_2$, $\{\tau_1\} \preceq \{\tau_2\}$.

If we let the domain of $m$ be the set of all types for every meta variable $m$, i.e., $\eta(m) = T$, and every tuple type have one meta variable as its attribute, then we get the type inheritance given in [Card 88].

**Theorem 2** The inheritance semantics given by Cardelli in [Card 88] is a special case of the inheritance in our model.

### 3.4 Features of the Data Model

#### 3.4.1 Named Values: Bridge between the Value-Oriented Model and Object-Oriented Model

In the relational model, relation schemas and relations are the principle concepts, and the model is closed under relational algebra. The results of an operation is also a relation, which can be used as an operand immediately. But in Object-Oriented Database Systems [Banc 88], where methods are applied to objects in classes, it is not clear which class the resulting object belongs to, and how encapsulation works.

In $O_2$, values as well as objects can be used to construct more complex objects. IQL [AbKa 89b] introduces relations into its data model, so that relational data model is properly included. Extending this idea, we introduce named values to further address the importance of values in object-oriented data model. Both values and named values are used in constructing other values and named values. Furthermore, named values are directly accessible.
• Relations are special cases of named values, i.e., a named set value whose elements are tuple values. Thus relations in IQL are properly included in our model.

• Named values are more general since an identifier can be mapped to a basic value, a tuple value, a set value or a union value.

• Identifiers are allowed in named values. Recursive named values can be represented. Data abstraction and information hiding are achieved. Duplicate data can be represented through named values, that is, the same value could be assigned to different identifiers.

• Named values are objects if their identifiers are assigned to classes.

• Named values can be operated directly. For example, named set values can be handled as nested relations and relational algebra is applicable.

3.4.2 Meta variables: Information Hiding and Data Abstraction

In [Card 88], inheritance is implicit, i.e., if type \( \tau_1 \) has more attributes than type \( \tau_2 \), then \( \tau_1 \) is a subtype of type \( \tau_2 \). There are advantages and disadvantages of this approach. Some inheritance implied by the system may not be desirable and the user may not be aware of its existence. On the other hand, some inheritance can not be modeled, such as parameterized types and parameter polymorphism. Another example is that the inheritance between two types with exactly the same structure can not be handled properly, as there is no corresponding mechanism in [CaWe 85, Card 88] to handle this situation. On the other hand, as has been pointed out in [AbK a 89b], this approach will lead to legal instances with attributes that do not appear in the schema.

In F-Logic [KiLa 89] a lattice structure is used to cope with inconsistency solving. This is a nice feature, but inheritance is not only a way of inconsistency solving. Furthermore, some inconsistency should be reported instead of being hidden.
Another problem that raises a lot of interest recently is data abstraction. Cardelli and Wegner [CaWe 85] used universal and existential quantifiers to represent abstract data structures and hide details from users. Inspired by this approach, the meta variables are introduced in our model to achieve the goal of information hiding, data abstraction, and inheritance. A meta variable in a type hides part of structural detail in the subtype. Through the instantiation of meta variable, a type hierarchy is built up, with its root being the most abstract data type, and all the leaf types being detailed structures. The types between the root node and leaf nodes expose the hidden structure to a certain degree.

Through meta variables, polymorphism of types is achieved explicitly. Instances in a type and instances in its subtypes have the same structure if looked from the supertype point of view. We believe this uniform structure shared by a type and all its subtypes will play an important role for an object-oriented database system in a logical framework, as the barrier applying a method defined on a type to its subtype is removed.

3.4.3 New Features of Inheritance

Inheritance is one of the key features in an object-oriented system and is the most important one [Card 88]. Inheritance in database systems can be traced back to the E-R model and other semantic models, where ISA relationship and specialization and generalization relationship are being used to specify inheritance between types [AbHu 87]. In Smalltalk, the term class is used to represent a set of objects and methods associated with it. The term class and type are often used interchangeably. The concept of type is used for structure specification of objects, while the concept of class groups objects with similar structure together. Strictly speaking, these two concepts are different [AbKa 89b].

In Prolog [StSh 86], predicates with more arguments are transformed into predicates with fewer arguments by rules (projection in relational algebra). In Login [AiNa 86a], a special rule is introduced to define partial order. The resolution steps to achieve inheritance in Prolog are replaced by partial order checking.
Actually the partial order forms a semilattice on data. If two terms are not directly unifiable, their \( glb \) is calculated. The unification algorithm is modified accordingly.

Sciore [Scio 89] addressed the problem of polymorphism of classes and objects. As pointed out there, specialization hierarchies are typically treated as type-level constructs and are used to define various inheritance mechanisms. Specialization at object level creates a more flexible and powerful notion of inheritance by allowing objects to define their own inheritance path. Object inheritance is also a mean of data abstraction, as subobjects and components of an object are, in some sense, encapsulated.

In our model, features of inheritance described in IQL [AbKa 89b] and \( O_2 \) [Ban+ 88, LeRi 89] are maintained, such as the \( isa \) hierarchy on classes and type inheritance. In addition, we have extended inheritance mechanisms, and added new features.

- Since meta variables hide structural difference of objects of a class and its subclasses, methods defined on a superclass can be applied to its subclasses.

- Meta variables play the role of parameters in a type. The instance types of a meta type are derived by exposing the structures hidden by the meta variable. This is exactly the meaning of \emph{parametric inheritance} as discussed in [CaWe 85]. The multiple inheritance semantics of our model is more general than that given in [Card 88].

Method inheritance in a deductive framework will be discussed in Chapter 7.

3.5 Summary

We have presented a data model that takes advantage of the simplicity of the relational model and has most of the features of the object-oriented data models. We developed the named value concept which is more general than the concept
of nested relation in IQL [AbKa 89b] and value $O_2[LeRi 89]$. Meta variables are introduced to implement information hiding, data abstraction and inheritance. The concept of isa hierarchies in [AbKa 89b] is kept. The type inheritance is based on the class inheritance and meta variables. Different from the multiple inheritance semantics discussed in [Card 88], type-subtype relationship is determined through meta variable domains. Meta variables, functioning as parameters in types, also model parametric inheritance. For each type $\tau$ we defined a special class $C_\tau$ that takes all named values with type $\tau$ as its objects. All classes with type $\tau$ now is subclasses of $C_\tau$. Through this simple treatment, the concepts of class inheritance and type inheritance are unified naturally. We also presented a nice feature of values. The concept of union value, together with other constructs of values, make it possible to define a value by its subvalues. Value hierarchy or value inheritance is an extension and complements to other inheritance mechanisms.

One of our goal is to have a data model that can be used to support a rule based language with object-oriented flavors. Meta variables hide the structural difference between type and its subtypes, thus clear the way for the unification algorithm, which, at current form, can not handle predicates with different numbers of arguments. Through meta variables, a type and its subtypes have the same structure at a higher level, which makes inheritance straightforward. A rule based language based on this model is described in the next chapter.
Chapter 4

The Underlying Logic

Based on the data model we developed in Chapter 3, we design a logic that underlies our data language, LLO. Similar to first order logic, we present the semantic structure in Section 4.2. We show that relational programming, including first normal form and non first normal form, is part of the logic.

4.1 The Syntax

In this section, we introduce LLO: a Logic Language for Objects. We follow the generally accepted notations, such as those used in [Lloy 84]. In addition to the parentheses and logical connectives ($\lor, \land, \neg, \rightarrow$) and quantifiers ($\exists, \forall$), the alphabet of an LLO language includes the following disjoint sets of symbols:

1) finite set of basic type symbols $\mathcal{B}$, including meta variables;

2) a set of type names $\mathcal{TN}$;

3) a set of type constructors $\mathcal{TC}$;

4) a set of oid constructors, i.e., function symbols; also called method constructors $\mathcal{F}$; Function symbols of arity 0 are constant oid's.

5) a infinite set of variables $\mathcal{Var}$.

As usual, variables start with uppercase letters, constants and function symbols start with lowercase letters. Class names will start with uppercase letters
and in Slanted mood. In most situation we will indicate a class name clearly in order to avoid confusion. We first discuss id-terms and terms. Then we present the declarative part of the language.

### 4.1.1 Id-terms

1) A constant oid is an id-term.

2) If \( t_1, t_2, \ldots, t_n \) are terms (defined below), \( f \) is a function symbol of arity \( n \), then \( f(t_1, t_2, \ldots, t_n) \) is an id-term, called function id-term.

Id-terms are first defined in O-Logic [Maier 86, KiWu 89], and then extended in F-Logic [KiLa 89], where they are used for object identity denotation. In our model, id-terms are also called predicate terms. We make extensions and use id-terms as

- object identifiers inside literals;
- method calls to assign values to the oid’s they denote and
- method definitions when they appear at the predicate symbol position in the head of a rule.

The set of ground oid’s formed by the rules given above is denoted by \( O \), which includes all addressable entities in the database (note that class names are oid’s, i.e., \( C \subseteq O \)).

### 4.1.2 Terms

The value assigned to an id-term can be accessed by a special operation \( \downarrow \), which corresponds to the \( \delta \) function in the data model discussed in Chapter 3.

- All id-terms, variables and constants are terms.
- If \( t \) is a term, then \( \downarrow t \) is also a term. We call this term exposed term. \( \downarrow \) is called exposure operator.
- If \( t_1, t_2, \ldots, t_n \) are terms, then \( [t_1, t_2, \ldots, t_n] \) is also a term called tuple term.
• If \( t_1, t_2, \ldots, t_n \) are terms, then \( \{ t_1, t_2, \ldots, t_n \} \) is a term, called enumeration term or set term.

We call set \( \{ \} \), tuple \([\ ]\) and \( \downarrow \) value constructors. Ground terms are values defined in Section 3.3.2, denoted by \( \mathcal{V} \).

### 4.1.3 Some Considerations

The introduction of the \( \downarrow \) operation brings the object-oriented model and value-oriented model together (cf. \( \delta \) mapping in Section 3.3.2). But it violates the encapsulation. Since all id-terms denote objects, the result of a method call (through id-term) is an object. By applying \( \downarrow \), we get the value assigned to the object denoted by the id-term (sounds like a printing procedure). This feature may be useful for some applications. For example, the nesting operation in nested relational data model can be easily expressed using a method (grouping related elements together) and a \( \downarrow \) operation (get the value returned from the method). In order to achieve encapsulation, however, the application of this operation should be limited. It can only be applied to those “opened up” objects. There are some objects whose states are not accessible directly. In this case, application of \( \downarrow \) is not allowed. The states of these encapsulated objects can only be accessed through methods defined on them. We will discuss this problem further in Chapter 7 and Chapter 8.

Unlike other approaches that treat classes and individual objects as inherently differently entities, both objects and classes have identities and values. Our approach is also different from the one used in F-Logic, where classes and objects are indistinguishable, or, relatively defined. In LLO language, all oid’s denote objects, but a subset of oid’s is designated to be a class. For example, \textit{Student} is a class. But the set of oid’s in class \textit{Student} is the state (value) of the object denoted by oid \textit{Student}. We believe that there is a need to distinguish objects and classes, since methods are defined on classes, not on objects. On the other hand, we need to operate on the class as a whole. In this case, the class is operated as an object. This feature gives us the flexibility of handling classes
without the loss of encapsulation and abstraction.

Each function symbol has an arity, a nonnegative integer that determines how many arguments each symbol can take. Function symbols of arity \( \geq 1 \) are used to construct new objects out of existing ones. We can see in the next section that a function id-term of arity \( > 0 \) is actually defining a method which can be activated to populate the oid denoted by the function id-term. A ground id-term denotes a named value or an object in a class depending on whether it belongs to a class or not.

LLO is a higher order logic language. First, set values are allowed to be components of values. Second, id-terms are used to denote objects and can be parts of other objects.

4.1.4 LLO Types and Declarative Formulae

The language of LLO consists of a set of formulae constructed out of the alphabet symbols. Formulae are built from the so called LLO declarative terms by means of the usual connectives, \( \lor, \land, \rightarrow, \neg \), and quantifiers \( \forall, \exists \).

First we present LLO type terms:

1. Each basic type \( B \) is a type term;

2. Each meta variable \( m \) is a type term;

3. Each type name \( T.N \) and class symbol \( C' \) is a type term;

4. \( \{\tau\} \) is a type term if \( \tau \) is a type term; \([\tau_1, \ldots, \tau_n]\) is type term if \( \tau_1, \ldots, \tau_n \) are type terms.

LLO declarative terms are defined as follows:

1. Type name declaration term, \( T.N \circ \tau \), where \( T.N \) is a type name, \( \tau \) is a type term defined above; type term declarative terms are used to define a structure denoted by the type name.
2. Meta domain declaration term, \( m \triangleleft \tau \), indicating that \( m \) is a meta type variable and \( \tau \) is a type that is in the domain of \( m \).

3. Class type declaration term, \( c :: \tau \), where \( c \) is a class name, \( \tau \) is a type term; class type declaration terms are used to specify the value (state) structure, \( \tau \) here, of objects that belong to class \( c \).

4. Subclass declaration term, \( c_1 \prec c_2 \), where \( c_1 \) and \( c_2 \) are class names, meaning class \( c_1 \) is a subclass of class \( c_2 \).

5. Type declaration term, \( t : \tau \). where \( t \) is a term as defined above, \( \tau \) is a type term; stating that \( t \) is a term of type \( \tau \). In case \( t \) is an oid and \( \tau \) is a class name, this term declares that the object denoted by this oid belongs to the class denoted by the class name.

6. Object state declaration term, \( p(t_1, t_2, \ldots, t_n) \), or \( p[t_1, t_2, \ldots, t_n] \), where \( p \) is an id-term, called predicate term, \( t_1, t_2, \ldots, t_n \) are terms or declarative terms. also called argument terms. If predicate term is a function id-term, its arguments are called predicate arguments. If \( p \) is of set type, \( p(t_1, t_2, \ldots, t_n) \) tells us that \( [t_1, t_2, \ldots, t_n] \) is an element of the value assigned to \( p \). If \( p \) is of tuple type, \( p[t_1, t_2, \ldots, t_n] \) tells us that \( [t_1, t_2, \ldots, t_n] \) is the value assigned to object \( p \). Furthermore, if \( p \) is a class name and \( n = 1 \), then \( p(o) \) indicates that \( p \) is used as an id-term and \( o \) is a member of the set of objects assigned to \( p \). Note that for class \( c \), \( o : c \) indicates \( o \) is an object assigned to class \( c \). while \( c(o) \) means that \( c \) is an oid whose value is a set and \( o \) is an element of that set.

Every LLO declaration term is an atomic formula. LLO-formulae are constructed from other (simpler) LLO formulae by means of logical connectives and quantifiers. That is,

- Atomic formulae are LLO-formulae:
- \( \varphi \lor \psi, \varphi \land \psi, \neg \varphi \) are LLO-formulae if so are \( \varphi \) and \( \psi \);
• \( \forall X \varphi \) and \( \exists X \varphi \) are formulae if so is \( \varphi \) and \( X \) is a variable.

As usual, the implication \( \varphi \leftarrow \psi \) is defined as \( \varphi \lor \neg \psi \).

Type declaration term (5) above specifies type of the term \( t \), or, in generic program jargon, the signature of the term \( t \). For example,

1). if \( t \) is a constant id-term, then \( \tau \) is the structure of the value that could possibly be assigned to \( t \);

2). if \( t \) is a function id-term in the form of \( f(X_1, X_2, \ldots, X_n) \), then \( \tau \) is the type of oid denoted by \( f(X_1, X_2, \ldots, X_n) \) when \( X_1, X_2, \ldots, X_n \) are instantiated with ground terms. We can use this term to specify a method signature with function symbol \( f \):

\[
f(X_1 : \tau_1, X_2 : \tau_2, \ldots, X_n : t_n) : \tau
\]

which is an abbreviation of the following LLO-formula:

\[
f(X_1, X_2, \ldots, X_n) : \tau \land X_1 : \tau_1 \land \ldots \land X_n : \tau_n.
\]

It will be clear in later discussion that the above formula describe method interface with signature:

\[
f : \tau_1 \times \tau_2 \times \ldots \times \tau_n \rightarrow \tau.
\]

Similar to F-Logic, there are other advantages of querying schema, methods and inheritance information by declaring not only data terms, but also type and class information. For example, it is sometimes convenient to combine different kinds of LLO declaration terms and write simpler formulae, such as

\[X : c(a, Y, Z)\]

indicating \( X \) is an objects in class \( c \) whose state is a set value. The first component of the value assigned to \( X \) is a constant \( a \). This formula is a syntactic sugar for

\[X : c \land X(a, Y, Z)\].
Further, we may assume there is an oid \textit{class}, all classes in the database are its members. That is, we implicitly assume for each class symbol $c$, there is a declaration term $\textit{class}(c)$. We have a handle to all classes as a whole. In addition we may define a meta class named \textit{metaclass} and specify all classes as its subclasses. That is, if $c$ is a class symbol, we have $c \prec \textit{metaclass}$. Methods could be defined on \textit{metaclass} which will be shared by all classes. Methods such as creating new object are common to all classes. Obviously the objects in class \textit{metaclass} include all objects that belong to at least one class. Since \textit{metaclass} is a class, we will have declaration $\textit{class}(\textit{metaclass})$. Again, both \textit{class} and \textit{metaclass} are oid's and can be addressed directly.

\section{4.2 Semantics}

Type names are shorthands for types. If we like, all type names can be replaced by the types they stand for.

An interpretation $I$ has the following components:

\textbf{Domains of Meta Variables: $\eta$}

For each meta variable $m$, a set of types is assigned as the domain of $m$. In \textit{Chapter 3} the mapping $\eta$ is defined for this purpose.

\textbf{Object Assignment to Class: $\pi$}

For each class $c$, a set of oid's are assigned to $c$:

$$\pi : C \rightarrow 2^O.$$ 

\textbf{Interpretation Domain: Type interpretation $I_T$}

The interpretation domain $I_T$ of the language is composed of several subdomains.

1. A set $D_{B_i}$ that provide the interpretation of basic type $B_i$.

2. A oid universe $O$ which is set of identities used for object denotations.

3. A set $C$ of class symbols, which is a subset of $O$. 
4. A set $O_c$ as interpretation of class symbol $c \in \mathcal{C}$, where $O_c \subseteq O$. In Chapter 3, assignment $\pi$ is a mapping from class names to sets of oid's.

5. A set $\{S \mid S \subseteq D_T\}$ as interpretation of type $\{T\}$, where $D_T$ is the interpretation of type $\tau$.

6. A set $\{[a_1, a_2, \ldots, a_n] \mid a_1 \in D_{\tau_1}, a_2 \in D_{\tau_2}, \ldots, a_n \in D_{\tau_n}\}$ as interpretation of type $[\tau_1, \tau_2, \ldots, \tau_n]$, where $D_{\tau_1}$ is the interpretation of type $\tau_1$, $D_{\tau_2}$ is the interpretation of type $\tau_2$, ..., and $D_{\tau_n}$ is the interpretation of $\tau_n$.

7. A set $\bigcup_{\tau \in \eta(m)} D_{\tau}$ as interpretation of meta variable $m$, whose domain is a set of types represented by the meta variable mapping $\eta$.

Note that in our data model classes are treated as types, but generally types are not classes\(^1\).

**Type Assignment of id-terms: $\sigma$**

1. For each class name $c \in \mathcal{C}$, there is a type $\tau$ specifying the structure of objects in this class.

2. For each oid $o \in O$ which is not a class, there is a type $\tau$ specifying its structure. If $o$ is assigned to a class, it will take the type of its class.

**Value Assignment to id-term: $\delta$**

The value assignment $\delta$ is a mapping from $O$ to $\Psi$:

$$\delta : O \rightarrow \Psi.$$  

and there are two cases:

1. For oid $c$ which is a class name and is of type $\tau$, its value interpretation is defined the same as $\pi$ above (i.e., when classes are handled as types which have domains), with the restriction that all oids assigned to $c$ must be of type $\tau$.

---

\(^1\) In Chapter 3, we propose to group all identifiers whose values come from the same type into a class. In this case, when we say methods are defined on type, we actually mean methods defined on its corresponding type.
2. For each oid \( o \) of type \( \tau \) and \( o \) is not a class name, its value interpretation is an element of \( D_\tau \).

**Id-terms as Mapping: Object Constructors \( I_\mathcal{F} \)**

Each id-term can be seen as a member of mappings\(^2\)

\[
\bigcup_{k=0}^{\infty} U^k \rightarrow O.
\]

1. For each constant oid, its interpretation is a function: \( o :\rightarrow O \). A constant oid only denotes one object because it lacks the ability to deliver information (arguments).

2. For each function id-term of type \( \tau \) with arity \( n > 0 \), and whose argument types are \( \tau_1, \tau_2, \ldots, \tau_n \) respectively, then its interpretation is a function from \( D_{\tau_1} \times D_{\tau_2} \times \ldots \times D_{\tau_n} \) to \( D_\tau \). The importance of arguments is that they convey information. When an id-term with arguments is used as a predicate id-term in the head of a rule, its arguments will be used in the body of the rule. Different arguments will generate different oid's.

A *variable assignment*, \( \nu \), is a mapping from the set of variables \( \text{Var} \), to the domain \( U \), observing the type restrictions. That is, if a variable \( X \) is of type \( \tau \), then \( X \) is mapped to \( D_\tau \). Variable assignments are extended to id-terms in the usual way: \( \nu(o) = I_\mathcal{F}(o) \) if \( o \) is a 0-arity id-term, that is, a constant id-term. Recursively, \( \nu(f(...,t,...)) = I_\mathcal{F}(f)(...,\nu(t),...) \). The assignments can be further extended to the declaration terms. For example, \( \nu(t : \tau) = \nu(t) : \tau \).

An LLO formula \( A \) is *true* under a semantic structure \( I \) with respect to a variable assignment \( \nu \), denoted \( I \models_\nu A \), if and only if the formula \( \nu(A) \) conforms to the semantic structure \( I \). More specifically:

- For meta domain declaration term \( m \triangleleft \tau \), \( I \models_\nu m \triangleleft \tau \) if and only if \( m \) is a meta variable and \( \tau \in \eta(m) \).

---

\(^2\)It is not necessary that every function term must be an oid. We make such restriction to simplify our presentation.
• For class type declaration term $c :: \tau$, $\nu \models c :: \tau$ if and only if for every $o \in \pi(c), \delta(o) \in D_{\tau}$.

• For subclass declaration term $c_1 \prec c_2$, $\nu \models c_1 \prec c_2$ if and only if $\pi(c_1) \subseteq \pi(c_2)$.

• For type declaration term $t : \tau$, there are two cases:
  1). if $\tau$ is a class name, then $\nu \models t : \tau$ if and only if $t \in \pi(\tau)$.
  2). if $\tau$ is a type but not a class, then $\nu \models t : \tau$ if and only if $\delta(t) \in D_{\tau}$.

• For object state declaration term $p(t_1, t_2, \ldots, t_n)$, $\nu \models p(t_1, t_2, \ldots, t_n)$ if and only if $\delta(p) = S$ and $[t_1, t_2, \ldots, t_n] \in S$.

For object state declaration term $p[t_1, t_2, \ldots, t_n]$, $\nu \models p[t_1, t_2, \ldots, t_n]$ if and only if $\delta(p) = [t_1, t_2, \ldots, t_n]$.

Note that the concept of classes is heavily overloaded. First, **classes are types** to achieve the effect of integrating value-oriented data and object-oriented data. When classes are types, we can use them in constructing more complex types, whose instances will have objects as their components. Second, **class is a group of objects**, which share a common set of methods and state structure. That is, classes are abstract data types. Encapsulation and information hiding are the major concern for the entities of classes. Third, a **class symbol is an identifier** whose state is a set of oid's. A class symbol can be used as a “handle” to the set of objects assigned to this class. Computation can be made upon the class as an object, which, if we like, can be assigned as instance to another class.

The meaning of formulae $\phi \lor \psi$, $\phi \land \psi$ and $\neg \phi$ is defined in the standard way:

- $\nu \models \phi \lor \psi$ if and only if $\nu \models \phi$ or $\nu \models \psi$.
- $\nu \models \phi \land \psi$ if and only if $\nu \models \phi$ and $\nu \models \psi$.
- $\nu \models \neg \phi$ if and only if $\nu \not\models \phi$.

For quantifiers in formulae, we have a similar definition:

- $\nu \models (\forall X)\phi$ if and only if for every $\mu$ that agrees with $\nu$ everywhere, except possibly on $X$, $\mu \models \phi$. 
\( I \models (\exists X)\phi \) if and only if there are some \( \mu \) that agrees with \( \nu \) everywhere, except possibly on \( X \), \( I \models_\mu \phi \).

### 4.3 Relational Programming in LLO

Unlike the F-Logic approach, where the relational programming is wired in by coding predicates into classes, our logic is a strict extension of first order logic. As indicated in [KiLa 89], some applications are more naturally described in a value-based setting, or a mixture of predicates and objects. Though there have been approaches that encode value-oriented terms into objects [BKK 88, KiLa 89], the advantages of the research results on relational programming are simply ignored or not well inherited. On the other hand, IQL [AbKa 89b] adopted an approach that puts objects, classes and relations in one system, and proved that languages like Datalog [Ullm 88], COL [AbKa 89b] and LDL [Bee+ 87] are all sublanguages of it. With this approach, all the techniques developed under the relational framework, such as query optimization, can be extended and fully utilized. Systems developed in the relational model can be extended to accommodate new features without being demolished and starting over from scratch. In our opinion, encapsulation is a good programming practice, but not every part of a database need to be encapsulated. In this section, we show that the relational data model is a submodel of LLO, and first order logic is a subset of the logic we just defined.

#### 4.3.1 Relational Model

The most important concept in relational model is relation, whose extension is a set of tuples of the same type. For a relation \( r \) of schema \( R : [\text{att}_1, \ldots, \text{att}_n] \), we can specify a corresponding type in LLO:

\[
R \circ [\text{att}_1, \ldots, \text{att}_n].
\]

and \( r \) is a named value declared as:

\[
r : \{R\}.
\]
i.e., \( r \) is an entity of type \( \{ R \} \), a set type whose element is of type \( R \). Note that in our model, several relations can share one schema (type).

In addition, the following restrictions apply:

1. \( att_1, \ldots, att_n \) in \( R \circ [att_1, \ldots, att_n] \) are basic types only.

2. There is no class concept.

3. There is no function symbol used in constructing oids (here we mean relation names).

4. All objects (relations) must be of set type, whose element type is a tuple.

Even though there are set and tuple constructors in relational model, tuple constructors can only be applied to basic types and set constructors can only be applied to tuples. (Recall the orthogonality requirement for data OODB modeling [ABDDMZ 89]).

### 4.3.2 Nested Relational Model

It has been proved that the relational data model is not expressive enough for database applications in CAD/CAM, CASE etc. One important extension to the relational data model is to allow tuple constructors being applied to set and tuple types [OOM 87, ScSc 86]. That is, nested relational model is the relational data model discussed in the last subsection without the restriction that the attributes are basic types only.

A nested relation \( r \) in general is of type \( \{ R \} \), where \( R \) is a name for a type, that is

\[
R \circ [att_1, att_2, \ldots, att_n]
\]

where \( att_i \), for \( i = 1, \ldots, n \) are types but not classes.

For example, if we represent the information of employees in an institution, we can define a type \( EMPS \), which is a set structure \( \{ EMP \} \). Every employee has a name, an address and a set of hobbies:

\[
EMP \circ [NAME, ADDRESS, HOBBIES]
\]
where NAME is a basic type, ADDRESS is a tuple type

[ST#, STREET, CITY, STATE, ZIP]

and HOBBIES is a set type whose element type is HOBBY, a basic type.

It is noted that LDL [Bee+ 87] and COL [AbGr 87a] are languages based on
the nested relational data model. Still the orthogonality is not achieved, because
the top level constructor of a relation is always a set.

4.3.3 Function Id-terms as Relation Names

If function id-terms are allowed to denote relation names, we could achieve the
effect of methods on schemas depending on the type of the arguments. Frequent
queries can be expressed by function terms. This gives us a flavor of functional
programming. A function term with variables can be seen as a generic relation
whose instantiation is a relation determined by assigning constants to variables
in the arguments.

For example, if we have a relation of schema

EMPS(NAME, SALARY),

we could use the generic relation \( f(X : EMPS, Y : SALARY) \) to denote the
results of selecting those employees from relation \( X \) of schema EMPS whose
salary is greater than \( Y \):

\[
f(X : EMPS, Y : SALARY)(NAME) : = X(X_1, X_2), X_2 \geq Y.
\]

When \( X \) and \( Y \) are replaced by appropriate constants, we expect \( f(X : EMPS, Y : SALARY) \) to be a relation. If we instantiate \( X \) with relation \( r \):

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>john</td>
<td>80000</td>
</tr>
<tr>
<td>harry</td>
<td>53000</td>
</tr>
<tr>
<td>louis</td>
<td>47000</td>
</tr>
<tr>
<td>malos</td>
<td>62000</td>
</tr>
</tbody>
</table>
then, \( f(r, 55000) \) denotes a relation:

\[
\text{NAME} \\
\text{john} \\
\text{malos}
\]

This extension makes relational programming higher order syntactically, but stays in first order semantically [CKW 89].
Chapter 5

Type Checking in LLO

Following the analysis of well sorted concepts of order sorted logic in [Schauss 89, Walther 84, Walther 89], we discuss type checking of LLO in this chapter. First we present some basic concepts. Then we give results on well typing. Finally we discuss type inference in object-oriented paradigm whose concepts such as sharing, polymorphism and inheritance provide great support for typing.

5.1 Basic Concepts

For an LLO term \( t \), operation \( V \) is defined to collect variables in terms:

- \( V(t) = \{t\} \) if \( t \) is a variable:
- \( V(t) = \cup V(t_i) \) for \( i = 1, 2, \ldots, n \) if \( t = f(t_1, \ldots, t_n) \).
- \( V(t) = \cup V(t_i) \) for \( i = 1, 2, \ldots, n \) if \( t = \{t_1, \ldots, t_n\} \) or \( t = [t_1, \ldots, t_n] \).

The operation \( V \) on other declarative terms can be defined similarly.

A term \( t \) with \( V(t) = \emptyset \) is a ground term. Actually the Herbrand universe is composed of all ground terms. We assume that for each type \( \tau \) in the language under discussion, there is at least one ground term of type \( \tau \). Otherwise, if \( D_\tau \)
is empty, then the set of formulae \( \{ p(X : \tau), \neg p(X : \tau) \} \) is not unsatisfiable, a situation that is against our intuition.

A subterm \( t_1 \) is replaced by another term \( s \) in term \( t \) is denoted by \( t[t_1 \leftarrow s] \).

A substitution \( \zeta \) is a mapping

\[
\zeta : \mathcal{V} \rightarrow \mathcal{U}
\]

such that the set \( \{ X \in \mathcal{V} | \zeta(X) \neq X \} \) is finite. The set of all substitutions is denoted as \( SUB \). \( U \) is the domain of discourse defined in the last chapter. The empty or identical substitution is denoted by \( \varepsilon \).

A substitution can be expressed as a finite set of variable-term pairs

\[
\{ X_1 \leftarrow t_1, \ldots, X_m \leftarrow t_m \}.
\]

The single pairs \( X_i \leftarrow t_i \) are called components or bindings.

Each substitution \( \zeta \) has a domain, denoted \( \text{DOM}(\zeta) \), defined as

\[
\text{DOM}(\zeta) = \{ X | \zeta(X) \neq X \}.
\]

And the codomain of the substitution is defined as

\[
\text{COD}(\zeta) = \zeta(\text{DOM}(\zeta)).
\]

We also define \( VC(\zeta) = V(COD(\zeta)) \), which gives the set of variables in the codomain in the substitution. A substitution is called ground iff \( VC(\zeta) = \emptyset \).

With \( \zeta|_W \) we denote the restriction of the substitution \( \zeta \) to the set of variables \( W \), that is, \( \zeta|_W(X) = \zeta(X) \) for \( X \in W \) and \( \zeta|_W(X) = X \) otherwise.

\( S \) is a mapping from the set of terms to types. We also use \( S(t) \) to denote the type of \( t \).

For a substitution \( \zeta \) and a term \( t \), \( t\zeta \) denotes applying substitution \( \zeta \) to term \( t \).

## 5.2 Well Typing in LLO

For an LLO language \( L \), the set of terms of type \( \tau \), \( T_{L,\tau} \), is constructed by the following rules:
1. $X \in T_{L, \tau}$, if $S(X)$ is a subtype of $\tau$.

2. $t \in T_{L, \tau}$, if $t : \tau' \in L$ and $\tau'$ is a subtype of $\tau$.

3. $t[X \rightarrow r] \in T_{L, \tau}$, if $t \in T_{L, \tau}$, $r \in T_{L, \tau'}$, $X$ is a variable of type $\tau''$ in $t$ and $\tau'$ is subtype of $\tau''$.

The last rules tells us that a new term $t'$ of sort $\tau$ is constructed from a term $t$ of type $\tau$ by replacing one of the variables in $t$ simultaneously by a term of type less than or equal to the type of this variable.

From Chapter 3 and [Schauss 89], we have the following lemmas.

**Proposition 5.2.1**

1. For all types $\tau$ and $\tau'$. $T_{L, \tau} \subseteq T_{L, \tau'}$ if $\tau$ is a subtype of $\tau'$.

2. For variables we have $X \in T_{L, \tau}$ iff $S(X)$ is a subtype of $\tau$.

*Proof.* By induction. ■

The set of terms of language $L$ can be represented as the union $\bigcup_{\tau \in L} T_{L, \tau}$, denoted by $T_L$. The set of ground terms of $L$ is denoted by $T_{L, g\tau}$.

For each term $t$, there is a set of types associated with it. For example, a TA is a staff, a student, a graduate student, and more generally a person. We have the following definitions: The types associated with a term $t$ is

$$S_L(t) = \{ \tau \in L \mid t \in T_{L, \tau} \}.$$ 

Obviously for each term $t \in T_L$, $S_L(t)$ is not empty. As shown in [Schauss 89], the set $S_L(t)$ is computable for each term $t$ for finite signatures.

**Well Typed Formulae**

A formula $p(t_1, \ldots, t_n)$ is well typed if $p$ is of type $[\tau_1, \ldots, \tau_n]$ and $t_i \in T_{L, \tau_i}$. Similarly, formula $p(t_1, \ldots, t_n)$ is well typed if $p$ is of type $[\tau_1, \ldots, \tau_n]$ and $t_i \in T_{L, \tau_i}$. 
We say a term type declaration \( t : \tau \) is *redundant* iff \( t \) is of type \( \tau \) with respect to the remaining term declarations. In the following discussion, we assume there is no redundancy.

A language \( L \) is *subterm closed* iff each subterm of all the terms in \( L \) is well typed.

A language is *regular* iff the subtype relationship is a partial order and for every term \( t \) in \( L \), the typing \( S_L(t) \) has a least type.

For regular language, the unique least type of a term \( t \) is denoted by \( L_S(t) \) and is called the type of the term.

**Proposition 5.2.2** For regular language \( L \),

\[
S_L(t) \subseteq S_L(s) \text{ iff } L_S(L_S(t)) \preceq L_S(t).
\]

**Definition 5.2.3** The set of well-typed substitutions \( SUB_L \) is defined as follows:

\[
SUB_L = \{ \zeta \in SUB | S_L(X) \subseteq S_L(X \zeta) \}.
\]

The condition in the definition means that the terms used to substitute variables in a term must have a more detailed type. This explains how value abstraction and information hiding are achieved. Note that in case a variable \( X \) is of type meta variable \( m \), all the values of type \( \tau \in \eta(m) \) could be used to instantiate variable \( X \). Well-typed substitutions map well-typed terms into well-typed terms.

**Proposition 5.2.4** For all well-typed terms \( t \in T_{L,\tau} \) and all well-typed substitutions \( \zeta \in SUB_L \), we have

\[
t\zeta \in T_{L,\tau}.
\]

*Proof.* By induction and definition.
Corollary 5.2.5 The composition ζζ of two well-typed substitutions ζ and ξ is again well-typed, i.e., $SUB_L$ is a monoid and $ɛ$ (empty substitution) is its identity element.

In the following we define an order on terms based on the properties of substitutions. This order is different from the type/class hierarchies we addressed in Chapter 3. The order on terms defined here simply indicates which term is more instantiated than another.

**Definition 5.2.6** Let $s, t \in T_L$, then

1. $s \succeq_L t$ iff there exists $ζ ∈ SUB_L$ such that $s = tζ$. In this case, we call $ζ$ an instantiating substitution of $t$ to $s$ and $s$ is an $L$-instance of $t$. We also call $ζ$ a matcher of $t$ to $s$ if $DOM(ζ) \cap V(s) = \emptyset$.

2. $s \equiv_L t$ iff $s \succeq_L t$ and $t \succeq_L s$

□

The instance relation of terms can be extended to well-typed substitutions:

**Definition 5.2.7** Let $W ⊆ Var$ and $ζ, ξ ∈ SUB_L$,

1. $ζ = ξ[W]$, iff $Xζ = Xξ$ for all $X ∈ W$.

2. $ζ \succeq_L ξ[W]$ iff there exists a $γ ∈ SUB_L$ with $ζ = γξ[W]$. In this case, we call $γ$ an instantiating substitution of $ξ$ to $ζ$ and $ζ$ an $L$-instance of $ξ$ modulo $W$.

3. $ζ \equiv_L ξ[W]$, iff $ζ \leq_L ξ[W]$ and $ζ \geq_L ξ[W]$.

□

The partial order on substitutions defined above compares two substitutions and points out which one is more general than another. This relationship plays a crucial role in determining the most general unifiers of two terms.

The following results are from [Schauss 89].
Proposition 5.2.8 For a finite language $L$, the sort $S_L(t)$ is effectively computable for all terms $t$.

The proof is by induction on the structure of terms.

Corollary 5.2.9 For two well-typed terms $s$ and $t$, $s \leq_L t$ is decidable. For two well-typed substitutions $\zeta$ and $\gamma$ it is decidable if $\zeta \leq_L \gamma[W]$ for a set of variable $W$.

Proposition 5.2.10 It is decidable whether a term declaration is redundant.

5.3 Type Inference

Strong typing is one of the most important features of modern languages. As a programming discipline, programming with strong typing can increase productivity and discover error early through static type checking. On the other hand, strong typing is a heavy burden to programmer, because one needs to specify types for all the procedures and data in the program. It is a good idea to have strong typing and at the same time liberate the programmer by inferencing types from the type declarations already made by programmers. In deductive programming, type inferencing is more important because detailed type declaration may obscure the program structure and ruin the benefit of declarative programming.

Fortunately the data model presented in Chapter 3 provides strong support to type inference for LLO programs. In object-oriented data models, objects are organized into classes, classes are built into inheritance hierarchies. Types and subtypes share common structures. The main idea in object-oriented programming is to share methods and structures, which means we do not need to specify the type for each object. All objects in one class share the same type.

We have the following inference rules:

1. Class Inheritance:

$$c_1 \leq c_2, c_2 \leq c_3 \vdash c_1 \leq c_3$$
and

\[ o : c_1, c_1 \leq c_2 \vdash o : c_2. \]

2. Type Inheritance:

\[ \tau_1 \leq \tau_2, \tau_2 \leq \tau_3 \vdash \tau_1 \leq \tau_3 \]

and

\[ t : \tau_1, \tau_1 \leq \tau_2 \vdash t : \tau_2. \]

3. Component Type of States of Objects:

\[ o : c_1, c_1 :: \{[\tau_1, \ldots, \tau_n]\}, o(t_1, \ldots, t_n) \vdash t_i : \tau_i \text{ for } i = 1, \ldots, n \]

and

\[ o : c_1, c_1 :: [\tau_1, \ldots, \tau_n], o[t_1, \ldots, t_n] \vdash t_i : \tau_i \text{ for } i = 1, \ldots, n. \]

4. Component Type of Named Values:

\[ o : \{[\tau_1, \ldots, \tau_n]\}, o(t_1, \ldots, t_n) \vdash t_i : \tau_i \text{ for } i = 1, \ldots, n \]

and

\[ o : [\tau_1, \ldots, \tau_n], o[t_1, \ldots, t_n] \vdash t_i : \tau_i \text{ for } i = 1, \ldots, n. \]

In the rest of this thesis, we assume all LLO programs are type checked. Unless necessary, we will omit type declarations. The interpretation \( I \) of a well typed program can be simply represented by

\[ I =< \pi, \delta >. \]

that is, a database instance discussed in Chapter 3 is an interpretation of a well typed LLO program.
Chapter 6

LLO As A Programming Language

In this chapter, we present LLO as a programming language. A subset of LLO, the set of Horn clauses, is first addressed. Then Herbrand interpretations and Herbrand models of LLO programs are discussed. Unification is an important issue in logic programming and is presented in a rule based form.

6.1 Horn Clauses in LLO

Definition 6.1.1 A literal is an atomic formula or the negation of an atomic formula. A positive literal is just an atomic formula. A negative literal is the negation of an atomic formula.

Definition 6.1.2 A clause is a formula of the form

\[ L_1 \lor L_2 \lor \ldots \lor L_n, \]

where each \( L_i \) is a literal.

Same as in F-Logic, skolemization in LLO is similar to its predicate calculus counterpart. Id-terms are used as predicate symbols and terms are identical to
terms in predicate calculus. Qualification is defined similarly in both cases. Note that the skolem functions introduced by the skolemization process are different from those in the LLO logic, the former are functions (dependency relationship) same as in the first order logic, while the later are object id-terms. It is easy to check that De Morgan's laws are valid for LLO language and hence every formula has an equivalent formula in conjunctive (resp. disjunctive) normal form. Based on the above analysis, we can transform every formula in LLO into a logically equivalent skolemized formula, which can be further transformed into a logically equivalent set of clauses using the De Morgan Law.

**Theorem 3** (cf. Skolem Theorem) Let \( \phi \) be an LLO-formula and \( \phi' \) be its normalized form after the skolemization. Then \( \phi \) is satisfiable if and only if so is \( \phi' \).

The proof of the above theorem is almost identical to the corresponding theorem in the first order logic. In the following we will focus on clausal forms of any LLO formula.

**Definition 6.1.3** A Horn Clause in LLO language is a clause that has at most one positive literal. A Horn clause that has one positive literal and zero negative literal is called a unit clause or a fact. A Horn clause that has no positive clause is called a goal clause, or a query clause.

A Horn clause \( A \lor \neg B_1 \lor \ldots \lor \neg B_n \) is also denoted as \( A : -B_1, \ldots, B_n \). A unit clause takes the form \( A: - \) or simply \( A \), and a goal clause takes the form \( : -B_1, \ldots, B_n \) or \( ?B_1, \ldots, B_n \).

**Definition 6.1.4** An LLO clause is a Horn clause of the form

\[
A : -B_1, B_2, \ldots, B_n
\]

in which \( A \) is a positive literal and \( B_i \) are literals.
An LLO program is a set of LLO clauses.

In the following examples, we assume the programs are type checked and if
there is no confusion, we omit the type declaration.

Example 6.1.5 In COL [AbGr 87a], the nesting operation is defined by the
following rules:

\[ s(X, f(X)) : -r(X, Y, Y'). \]
\[ f(X)(Y, Y') : -r(X, Y, Y') \]

where \( s \) and \( r \) are relations and \( f \) is a data function collecting data into a set.
This program can only do the nesting operation for relation \( r \).

In LLO, the nesting operation is implemented by the following program:

\[ s(X, f(r, X)) : -r(X, Y, Y'). \]
\[ f(R, X)(Y, Y') : -R(X, Y, Y'). \]

where \( R \) is a variable, \( r \) is a constant oid and \( f(R, X) \) is a function id-term. Even
though this program looks like the one in COL, there are several substantial
differences:

- For each constant \( a \), \( f(r, a) \) denotes an object identifier instead of a value.
  A value is assigned to the oid by calling method \( f(R, X) \), where \( R \) is instantiated
  with \( r \). In other words, \( f(r, a) \) is a call for the method \( f(R, X) \) which is defined
  by the second rule. If such a method does not exist, this oid does not have a
  value (a kind of null)

- \( R \) is an argument in \( f(R, X) \). The second rule itself is a method and
  assigns value to the oid denoted by \( f(R, X) \) (after \( R \) and \( X \) are instantiated).
  The second rule is independent of the first rule.

- If we replace the first rule by the following rule,
  \[ nest(R)(X, f(R, X)) : -R(X, Y, Y') \]
  then the program becomes a method that can be used to do the nesting for any
  relation \( r \) having a compatible type. The result will be an object denoted by
  \( nest(r) \).

Union and Intersection are frequently used data operations and are defined
as follows:
Union:

\[ \text{union}(S_1, S_2)(X) : -S_1(X) \]
\[ \text{union}(S_1, S_2)(X) : -S_2(X) \]

Intersection:

\[ \text{intersection}(S_1, S_2)(X) : -S_1(X), S_2(X) \]

The syntax of the union and intersection programs are the same as those in HiLog [CKW 89]. But HiLog is a value-oriented language and there is no inheritance. In LLO, objects and typing are introduced, and the roles played by function id-terms are extended. The result is an object-oriented deductive language with parametric and inheritance polymorphisms.

The following is another example illustrating object-oriented features of the language and the roles played by function id-terms.

Example 6.1.6 [AbKa 89b] Let \( r \) be a set of edges with type \([D, D]\) representing the graph \( G \), where \( D \) is the type of the node of the graph. The following program will transform each node in \( G \) in class \( C \). The set of objects corresponding to \( G \) is assigned to object \( p \). Class \( C \) is of type \([D, \{C\}]\), where \( \{C\} \) represents the set of objects for those nodes to which \( D \) has an edge. An instance of \([D, \{C\}]\) is actually an adjacency list for a node in \( G \).

First, for each node \( d \) appearing in \( r \), there is one object identifier denoted by the function id-term \( \text{adj}(r, d) \). and these oid's are populated by calling method \( \text{adj}(R, X) \).\(^1\)

\begin{align*}
(1) & \quad p(\text{adj}(r, X)) : -r(X, Y). \\
(2) & \quad p(\text{adj}(r, Y)) : -r(X, Y).
\end{align*}

The first attribute of the resulting object is the node \( X \), the second attribute is the set of objects whose first attribute share edges with \( X \). We use \( g(R, X) \) to collect all those objects. \( g(R, X) \) will get the value assigned to \( g(R, X) \).

\begin{align*}
(3) & \quad \text{adj}(R, X)[X, \downarrow g(R, X)] : -R(X, Y)
\end{align*}

\(^1\)Method signature which specifies the type of the output of a method will be discussed in Chapter 7.
Finally, we have the following rule to collect all those oid's related to node $X$ through their first attribute (node $Y$) into $g(R, X)$.

\begin{equation}
(4) \quad g(R, X)(\text{adj}(R, Y)) : -R(X, Y).
\end{equation}

Since $\text{adj}(R, Y)$ denotes oid, this rule simply collects all those oid's into a set and assigns it to $g(R, X)$.

For example, let $r$ be a graph with four nodes, $a, b, c$ and $d$. The set of edges is given below:

\begin{equation}
r: \quad [a, b], \ [b, c], \ [a, d], \ [c, d]
\end{equation}

First, for each node in $r$, there is one oid denoted by a ground id-term. Thus $p$ is assigned a set of objects \{adj($r, a$), adj($r, b$), adj($r, c$), adj($r, d$)\} (rule (1), (2)). $\text{adj}(r, a)$ is assigned the value $[a, \downarrow g(r, a)]$. By the last rule, $g(r, a)$ is assigned the value of \{adj($r, b$), adj($r, d$)\}. Since the function $\downarrow$ is to get to the value, the value assigned to $\text{adj}(r, a)$ is $[a, \{\text{adj}(r, b), \text{adj}(r, d)\}]$. Similarly, the value assigned to $\text{adj}(r, b)$ is $[b, \{\text{adj}(r, c)\}]$, the value assigned to $\text{adj}(r, c)$ is $[c, \{\text{adj}(r, d)\}]$, and the value assigned to $\text{adj}(r, d)$ is $[d, \{\}]$.

Note that in a similar example in F-Logic, every node in the graph is an object already and has a descendant attribute. So one rule similar to the nesting operation is enough to transform a graph represented by a set of edges into a set of nodes, each one of which has a descendant attribute. The program in LLO, as well as in IQL, actually does much more work, transforming value-oriented data into object-oriented ones. This is because in LLO and IQL, types and values are strictly separated, which corresponds to strong typing in programming languages. This, in turn [AbKa 89b], facilitates having a data model which generalizes the relational data model, most complex-object models and logical data models.

In the following we briefly review the interpretation for an LLO program.

The interpretation $I$ of a LLO language $L$ consists of:

1. The universe $U_L$, for an LLO language is the the universe $U'$ defined in
   \textit{Chapter 4} satisfying type constraints.
2. For each object id-term, a value is assigned to it.

3. For each class name, a set of oid's is assigned to it.

4. Structure declarations and types for classes and named values.

Since we assume that the database is well-typed, item 4 in a interpretation can be ignored.

Given an interpretation \( I \), a \textit{valuation} \( \theta \) is a function from variables to \( U_L \) satisfying type constraints. That is, \( \theta \) is a well-typed ground substitution in \( SUB_{L,g} \). \( \theta \) can be extended to terms and literals. For example, \( \theta(f(t_1, \ldots, t_n)) = f(\theta(t_1), \ldots, \theta(t_n)) \) for function id-terms. We have the notion of satisfaction similar to that in COL [AbGr 87a].

\textbf{Definition 6.1.7} The notion of satisfaction (\( I \models \)) for interpretation \( I \) is defined by:

1. For each ground atomic literal \( A \), there are several cases:
   - \( A \) takes the form of \( p(t) \), \( p \) is an oid, and \( t \) is a tuple, then \( I \models A \) if \( t \) is a member of the set value assigned to \( p \).
   - \( A \) takes the form of \( p[t] \), \( p \) is an oid, and \( t \) is a tuple, then \( I \models A \) if \( t \) is the value assigned to \( p \).
   - \( A \) is a type declaration term, then \( I \models A \) if \( A \in I \).

2. For ground terms \( b_1 \) and \( b_2 \), \( I \models b_1 = b_2 \) if only if \( b_1 = b_2 \) is a tautology.

3. For each ground negative literal \( \neg L \), \( I \models \neg L \) iff \( I \not\models L \).

4. Let \( r = A \models L_1, L_2, \ldots, L_m \). Then \( I \models r \) if and only if for each valuation \( \theta \) such that for each \( i \), \( I \models \theta L_i \), implies \( I \models \theta A \).
5. For each program $P$, $I \models P$ iff for each rule $r \in P$, $I \models r$.

A model $M$ of $P$ is an interpretation which satisfies $P$. \hfill \Box

### 6.2 Herbrand Interpretations

In our model, there are types. Meta variables and their domains specifies relationship among types and their subtypes. In particular, classes are special cases of types and each class has a type as a specifier of its object structure. The Herbrand base $U_P$ of an LLO program $P$ should be constructed according to the type declarations, function symbols and their signatures that appear in $P$.

**Definition 6.2.1** Let $P$ be an LLO program. The Herbrand universe $H_P$ for $P$ is the set of all ground terms formed out by applying the following rules recursively.

1. For each basic type symbol $\tau$ in $P$, all constants in $P$ of type $\tau$ is in $D_\tau$, which is a subset of $H_P$. (In the case that $P$ has no constants of type $\tau$, we add some constant, say, $a_\tau$, as an element in $D_\tau$).

2. For each class $c$, all the oid declared to be of type $c$ is in $D_c$, which is a subset of $H_P$.

3. If there is a type $[\tau_1, \ldots, \tau_k]$ in $P$, then $[a_1, \ldots, a_k]$ is in $D_{[\tau_1, \ldots, \tau_k]}$ if $a_1 \in D_{\tau_1}, \ldots, a_k \in D_{\tau_k}$. $D_{[\tau_1, \ldots, \tau_k]}$ is a subset of $H_P$.

4. If there is a type $\{\tau\}$ in $P$, then $S \in D_{\{\tau\}}$ if $S \subseteq D_\tau$. $D_{\{\tau\}}$ is a subset of $H_P$.

5. If $f$ is a function symbol with signature $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ in $P$, then $f(a_1, \ldots, a_n)$ is an element in $D_\tau$, which is a subset of $H_P$. \footnote{For simplicity we restrict $\tau$ to be a class. This restriction can be relaxed to any type with no problem.}

Constant $\text{oid's}$ are considered zero arity function id-terms.
6. If \( c_1 \) is a subclass of \( c_2 \) in \( P \), then \( D_{c_1} \subseteq D_{c_2} \).

7. If \( \tau_1 \) is a subtype of \( \tau_2 \) in \( P \), then \( D_{\tau_1} \subseteq D_{\tau_2} \).

Definition 6.2.2 Let \( P \) be an LLO program. The Herbrand base \( B_P \) for \( P \) is the set of all ground atoms which can be formed by using types, classes and ground terms from the Herbrand universe \( H_P \), observing the following rules:

1. For each element \( a \in D_\tau \), \( a : \tau \) is in \( B_P \).

2. For each class \( c \) and type \( \tau \), \( c : \tau \in B_P \).

3. For classes \( c_1 \) and \( c_2 \), \( c_1 :: c_2 \in B_P \).

4. For oid \( p \) of type \( [\tau_1, \ldots, \tau_k] \), \( p[a_1, \ldots, a_n] \in B_P \) if \( a_1 \in D_{\tau_1}, \ldots, a_k \in D_{\tau_k} \).

5. For oid \( p \) of type \( \{\tau\} \), \( p(a) \in B_P \) if \( a \in D_\tau \). Note that \( a \) is a tuple if \( \tau \) is a tuple type.

6. For each class name \( c \) and oid \( p \) that is of type \( c \), \( c(p) \) \( \in B_P \). This rule reflect our treatment that a class is an object whose set value is the set of objects assigned to this class. Note that when a class \( c \) is considered as an object, its type is \( \{c\} \).

A subset \( H \) of \( B_P \) is a Herbrand interpretation of \( P \) if and only if \( H \) is closed under the logical \( \vdash \) introduced in the last chapter, that is, \( H \vdash t \) if and only if \( t \in H \). (see type inference rules in Chapter 5).

Given a Herbrand interpretation \( H \), a valuation \( \theta_H \) is a function from variables to \( H_P \) satisfying type constraints. That is, \( \theta_H \) is a well-typed ground substitution in \( S U B_{LGr} \). \( \theta_H \) can be extended to terms and literals. For example, \( \theta_H(f(t_1, \ldots, t_n)) = f(\theta_H(t_1), \ldots, \theta_H(t_n)) \) for function id-terms. We have the notion of satisfaction similar to that in COL [AbGr 87a].
A data model discussed in Chapter 3 can be seen as an interpretation of LLO program, and from a Herbrand interpretation of a program P, a database instance can be constructed.

We call P, program under schema < O, C, σ, < >, where O is a finite subset of Bp, C is a finite set of class names, σ is a mapping from class names and type names to types and < is a partial order as given in Section 3.3.2. A data base instance \( I = \langle \pi, \delta \rangle \) is an interpretation of the program P, where \( \pi \) is an old assignment for class names and \( \delta \) is a mapping from identifiers to values (ref. Chapter 3). A database instance represents a set of ground facts\(^3\) similar to a Herbrand interpretation [AbKa 89b].

\[
ground-facts(I) = \{C(o)|C \in C, o \in \pi(C)\}
\cup \{o(t)|\delta(o) = S, S \text{ is a set value} , t \in S, o \in O\}
\cup \{o[t]|\delta(o) = t, t \text{ is not a set value} , o \in O\}
\]

If \( \text{ground-facts}(I) \) (together with type declarations) is a model of program P, we call the database instance I the model of the program also. The counterpart of \( \text{ground-facts}(I) \) in first order logic is its interpretation.

Similar to F-Logic, we could build a database instance \( I_H \) from a Herbrand model H of a program P. This correspondence gives us the following propositions:

**Proposition 6.2.3** Let P be a LLO program. Then P is unsatisfiable if and only if P has no Herbrand model.

*Proof.* It is easy to verify that for any Herbrand model H, \( H \models P \) if and only if \( I_H \models P \), where \( I_H \) is the database instance corresponding to H, as defined above.

In first order logic, a set S of clauses is unsatisfiable if and only if there is a finite set of ground instances of clauses in S that is unsatisfiable. This property is proved in Herbrand theorem. In LLO, Herbrand's theorem holds, and provides

\(^3\)Type declarations are skipped out since we assumed that a database instance is well-typed.
most important support for an effective query evaluation. Following the proof techniques developed in [Enderton 72] and [Lloy 84], we will briefly review the result and outline the proof of the Herbrand theorem. Detailed discussion on this issue also appeared in F-Logic.

The following lemma restates the compactness theorem of first order logic in LLO. A set \( S \) of ground LLO clauses is finitely satisfiable if every finite subset of \( S \) is satisfiable. A finitely satisfiable set \( S \) is maximal, if no other set of ground rules containing \( S \) is finitely satisfiable [KiLa 89].

Lemma 6.2.4 Given a finitely satisfiable set \( S \) of ground clauses, there exists a maximal finitely satisfiable set \( M \), such that \( S \subseteq M \).

Proof. Let \( \mathcal{A} \) be a collection of all finitely satisfiable sets of ground clauses that contain \( S \); \( \mathcal{A} \) is ordered by set inclusion. Since \( S \in \mathcal{A} \), \( \mathcal{A} \) is not empty. Furthermore, for every chain \( \Sigma \subseteq \mathcal{A} \), the least upper bound of the chain, \( \cup \Sigma \), is also in \( \mathcal{A} \) because 1) \( \cup \Sigma \) contains \( S \) and 2) \( \cup \Sigma \) is finitely satisfiable, for otherwise, neither is one of the elements of \( \Sigma \). By Zorn's Lemma, there is a maximal element \( M \) in \( \mathcal{A} \). Clearly, it is maximal, finitely satisfiable and has \( S \) as its subset.

\[ \]

Theorem 4 (Herbrand Theorem) A set \( S \) of clauses is unsatisfiable if and only if there is a finite unsatisfiable set \( S' \) of ground instances of clauses of \( S \).

6.3 Unification and Query Evaluation

The unification rules [Beierle 89] for the first order logic in the style of equation solving [MaMo 82] is presented in Figure 6.1, including the rules for elimination (E), decomposition (D), variable binding (B), and orientation (O). The idea is to start with a set of equations \( E = \{ t_1 \equiv t'_1, \ldots, t_n \equiv t'_n \} \) (also denoted as \( t_1 \equiv t'_1 \land \ldots \land t_n \equiv t'_n \)) representing the pairs of terms to be unified and to transform \( E \) into a set of equations \( E' \) that is in solved form by using the four
(E) \[ \frac{E \& X \equiv X}{E} \]

if \( X \) is a variable.

(D) \[ \frac{E \& f(t_1, \ldots, t_n) \equiv f(t_1', \ldots, t_n')}{E \& t_1 \equiv t_1' \& \ldots \& t_n \equiv t_n'} \]

(B) \[ \frac{E \& X \equiv t}{E \theta \& X \equiv t} \]

if \( X \) is a variable, \( t \) is a variable or a non-variable term, and \( X \) occurs in \( E \) but not in \( t \), and \( \theta = \{ X \leftarrow t \} \).

(O) \[ \frac{E \& t \equiv X}{E \& X \equiv t} \]

if \( X \) is a variable and \( t \) is not a variable.

Figure 6.1: Unification rules for the first order logic

given rules. \( E' \) is in solved form if it is of the form \( E' = \{ Z_i \equiv t_i | i \in \{1, \ldots, n\} \} \)
where \( Z_i \) are variables that do not occur elsewhere in \( E' \). In this case, the
substitution represented by \( \{ Z_1 \leftarrow t_1, \ldots, Z_n \leftarrow t_n \} \) is the most general unifier
of the equations given by \( E \).

Our language bears some similarity to the order sorted logic languages, such
as those discussed in [Walther 84, Walther 89, Schauss 89]. Major features of
the order sorted logic include:

1. The domain of discourse are divided into domains for each sort. The
domain of a subsort is a subdomain of its supersort.

2. In the unification algorithm, only terms of compatible types can be unified.
More specifically, a constant can be used to instantiate a variable if the sort of
the former is a subsort of the later. This is called inheritance.

The order sorted unification algorithm extends the first order unification
algorithm by enforcing sort restrictions whenever a substitution is made. The
equational form of the order sorted unification algorithm is given in Figure 6.2.

In LLO in addition to types (sorts), we introduced oid's and classes. turning
(E) \[
E & X \doteq X \\
E
\]
if \( X \) is a variable.

(D) 
\[
E & f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n) \\
E & t_1 \doteq t'_1 \& \ldots \& t_n \doteq t'_n
\]

(B1) 
\[
E & X \doteq t \\
E \theta & X \doteq t
\]
if \( X \) is a variable of sort \( s \), \( t \) is a variable or a non-variable term of sort \( s' \), \( s' \) is a subsort of \( s \), and \( X \) occurs in \( E \) but not in \( t \), and \( \theta = \{ X \leftarrow t \} \).

(B2) 
\[
E & X = Y \\
E \theta & X \doteq Z \& Y \doteq Z
\]
if \( X \) is a variable of sort \( s \), \( Y \) is a variable of sort \( s' \), \( X \neq Y, s' \not\subseteq s \), \( Z \) is a variable of sort \( s'' \) which is the greatest common subsort of \( s \) and \( s' \), and where \( \theta = \{ X \leftarrow Z, Y \leftarrow Z \} \).

(O) 
\[
E & t \doteq X \\
E & X \doteq t
\]
if \( X \) is a variable and \( t \) is not a variable.

Figure 6.2: The unification rules for order sorted logic
LLO into an order sorted higher order language. Thus, the unification algorithms developed in order sorted logic needs to be extended to accommodate new features of LLO.

Our discussion of LLO unification will be restricted to LLO programs without $\downarrow$ (exposure operation). Unifications of set terms are addressed in [Siekmann 84, Stickel 81, Smolka 89]. We will assume that unification of two set terms is treated properly.

**Definition 6.3.1** A well-typed substitution $\theta$ is an unifier for a set of LLO terms \{e₁, \ldots, eₙ\} if and only if $e₁\theta = \ldots = eₙ\theta$. The set \{e₁, \ldots, eₙ\} is unifiable if it has a unifier.

A well-typed unifier $\theta$ for a set \{e₁, \ldots, eₙ\} of expressions is most general if and only if for each well-typed unifier $\theta'$ of this set there is a well typed substitution $\gamma$ such that $\theta = \theta'\gamma$.

As has been proven in [Walther 84, Walther 89], in a sorted first order logic, if any two sorts have at most one maximal subsort, then the well-typed mgu exists if the unification succeeds. As we can see in the following, there is no unique mgu for terms that involve sets (or ACI functions, A stands for associative, C stands for commutative and I stands for idempotent).

One most important extension to the unification rules of order sorted rules in LLO is that id-terms, or more specifically predicate terms must participate in the unification, since successful unification of oid term in a message and the oid term in a method interface determines if this method should be used in responding to the message. So basically our unification is the integration of the unification rules in HiLog [CKW 89] and the those in order sorted logic [Walther 84, Walther 89, Schauss 89, Beierle 89]. In LLO, predicate id-terms are playing the role of predicate symbols in normal first order logic. Since function terms are id-terms and function terms have argument terms, their participation in the unification is crucial to achieve methods and method inheritance. On the other hand, function id-terms are data terms and can appear anywhere in a literal. In summary,
1). Predicate terms participate in the unification;

2). Type restrictions must be observed in building up unifiers.

In LLO, id-terms, when associated with their states, forms the predicate literals which participate in the unification as a whole. However, in the first order logic, order sorted or not, only arguments of predicate terms participate in the unification. In a sense, our approach is a strict extension to the first order case, where two literals are unifiable only if they have the same predicate symbol. So the participation of predicate symbols in the unification also works for the first order logic. The LLO unification rules are given in Figure 6.3.

It is important to note that the exposure operation \( \downarrow \) is not addressed in the unification algorithm. Whenever an exposure operation is involved, we need to know the state of the oid in order to make a successful unification. That complicates the resolution process.

Another issue is the type-subtype compatibility. Remember that we have introduced the meta variable to hide the structural difference between instances of a type and those of its subtypes. So our unification algorithm is simply the HiLog unification extended with types. All the extra information contained in a subtype is absorbed by the meta variable in the supertype.

In order sorted logic, if two terms participating in a unification have at most one maximal subsort, then among all unifiers of those terms, there exists a most general unifier. This is an important results for the efficiency of order sorted programs. But in LLO, the introduction of set terms upsets the chance of finding a most general unifier among terms because of the ACI properties of sets. For example, set \( \{X, Y\} \) and set \( \{a, b\} \) have unifiers \( \{X\!/a, Y\!/b\} \) and \( \{X\!/b, Y\!/a\} \), and no one is more general than another. But if there is no set term unifying with another set term, we still can find a mgu if those terms are unifiable. A set value can be unified with a variable by a single substitution.

There have been efforts in developing unification algorithms involving ACI functions, i.e., [Siekmann 84, Stickel 81, Smolka 89]. In this thesis, we will not go in to details. We assume there is a set of maximal unifiers among two terms.
(E) \[
\frac{E \& X \doteq X}{E}
\]
if \(X\) is a variable.

(D1) \[
\frac{E \& t(t_1, \ldots, t_n) \doteq t'(t'_1, \ldots, t'_n)}{E \& t \doteq t' \& t_1 \doteq t'_1 \& \ldots \& t_n \doteq t'_n}
\]
Note that here \(t\) and \(t'\) are id-terms, which may be constant id’s or function id-terms. This rule applies to id-terms \(t\) and \(t'\) of set type.

(D2) \[
\frac{E \& t[t_1, \ldots, t_n] \doteq t'[t'_1, \ldots, t'_n]}{E \& t \doteq t' \& t_1 \doteq t'_1 \& \ldots \& t_n \doteq t'_n}
\]
This rule is for id-terms \(t\) and \(t'\) of tuple type.

(D3) \[
\frac{E \& [t_1, \ldots, t_n] \doteq [t'_1, \ldots, t'_n]}{E \& t_1 \doteq t'_1 \& \ldots \& k \cdot t_n \doteq t'_n}
\]

(B1) \[
\frac{E \& X \doteq t}{E \theta \& X \doteq t}
\]
if \(X\) is a variable of type \(\tau\), \(t\) is a variable or a non-variable term of type \(\tau'\), \(\tau'\)
is a subtype of \(\tau\), and \(X\) occurs in \(E\) but not in \(t\), and \(\theta = \{X \leftarrow t\}\).

(B2) \[
\frac{E \& X \doteq Y}{E \theta \& X \doteq Z \& Y \doteq Z}
\]
if \(X\) is a variable of type \(\tau\), \(Y\) is a variable of type \(\tau'\), \(X \neq Y\), \(\tau' \not\subseteq \tau\), \(Z\) is a variable of type \(\tau''\) which is the greatest common subtype of \(\tau\) and \(\tau'\), and where \(\theta = \{X \leftarrow Z, Y \leftarrow Z\}\).

(O) \[
\frac{E \& t \doteq X}{E \& t \doteq X}
\]
if \(X\) is a variable and \(t\) is not a variable.

Figure 6.3: The unification rules for LLO
unless indicated otherwise.
Chapter 7

Methods and Method Inheritance

In Chapter 3, a data model is established for the representational aspects of complex objects in a database. In this Chapter, we focus on the behavioral aspect of objects. Methods are defined on one class or on several classes. Method inheritance is achieved along the class inheritance hierarchies. Based on the signature of a method, method overloading and overriding are easily achieved. Utilizing the logical inference mechanisms, we propose a solution to the multiple inheritance problem.

7.1 Rules as Methods

In F-Logic [KiLa 89], attributes are methods. The state of an object (set of attributes) is accessible by using the methods applicable to the object. In order to achieve encapsulation, a view concept similar to the one in relational data model is proposed to expose parts of the state to authorized users. But the concept of view, in our opinion, is not equivalent to the concept of encapsulation. Data independence is an important part of encapsulation and structural change of the state should be shielded.

In LLO, the structural part of an object is composed of an oid and a complex
state. There are two different ways to get access to the state depending on whether the object is encapsulated or not:

1. If the object is encapsulated, then a group of methods should be defined as interface.

2. If the object is not required to be encapsulated, the state of object can be accessed directly. For example, \( \downarrow \) operation can be applied and attribute values are available to users.

Methods can be defined on several classes/types or on a single class/type. Even though methods are being used as interface to the states of objects involved and to achieve encapsulation, in some situation, methods are used mainly to achieve abstraction of computation and sharing. Methods get attribute values from objects or derive new information from the states of those objects.

A rule in LLO is a method. By generic method, we mean a method defined on several classes/types and has the properties of polymorphisms, both parametric and inheritance [Card 88, CaWe 85]. As has been pointed out in Chapter 1, a Datalog rule is not a generic method. More specifically, in Datalog, predicate names are constants which prohibit any attempt of sharing. Hence rules in LLO whose predicate terms on the left hand side are constant oid’s can be considered as concrete methods. Example 1.4.1 in Chapter 1 defines a concrete method to get transitive closure for relation \( r \).

We argue that with function symbols used in id-terms, we can define generic methods on classes/types. Variables in the predicate id-term in the head of a rule play the role of passing information from the head to the body. The types of these variables specify input argument types of the method. Methods so defined are called computation schemas or templates.
7.2 Generic Methods

Each function symbol has an arity and a type associated with it. The type of a function symbol \( f \) takes the form:

\[
f : \tau_1 \times \tau_2 \times \ldots \times \tau_n \rightarrow \tau
\]

where \( n \) is its arity, \( \tau_1, \tau_2, \ldots, \tau_n \) are the types of its arguments and \( \tau \) is the type of the output object. In particular, each object identity (oid) \( o \) with type \( \tau \) can be seen as a constant function of type

\[
o : \rightarrow \tau.
\]

It is obvious that the type of a function takes the form of the signature of a method [AKW 90]. The arguments of the function id-term are the information that can be passed from the head of a rule to its body if the id-term is being used as a method interface. For constant oid's, no argument means no information passing. Consequently, constant oid's are interfaces of concrete methods.

When the head of a rule starts with a function id-term, this rule is defining a method. Id-terms with arguments define generic methods and id-terms without arguments define concrete methods. The type of the function is the signature of the method as in [AKW 90]. When a message (method call) is received by a method, a value is computed and assigned to the resulting object. The identifier of the resulting object is denoted by the function id-term (with all variables instantiated) at the head.

**Example 7.2.1** It is straightforward to define a method for the transitive closure program in Example 1.1.1¹.

\[
\text{trans-closure}(R)(X, Y) : = R(X, Y).
\]

\[
\]

with signature

\[
\text{trans-closure} : \tau \rightarrow \tau.
\]

¹The same observation is made in HiLog [CKW 89] from pure logic point of view.
where \( \tau \) is a class name of type \([D, D]\). Type \( D \) may be a meta-variable whose domain includes all relevant types. In this example, \( R \) is a variable of type \( \tau \). \( \text{trans-closure}(R) \) is the predicate id-term denoting the interface of the method. If we send a message \( \text{trans-closure}(r) \) to the database, method \( \text{trans-closure}(R) \) will respond to this message. The oid of the resulting object is \( \text{trans-closure}(r) \) and a value is assigned to this oid by the method (in this case, the transitive closure of \( r \)). The method can be called by instantiating variable \( R \) with an object which has type \( \tau \) or its subtype. However, in this example, parametric inheritance and method sharing are our major concerns.

\[
\square
\]

**Definition 7.2.2** A generic method by function id-term is a rule (set of rules) with the literal at left-hand-side starting with a function id-term \( f(X_1, X_2, \ldots, X_n) \) of type \( \tau_1 \times \tau_2 \times \ldots \times \tau_n \rightarrow \tau \). The body of the rule (rules) is the implementation of the method, the function id-term \( f(X_1, X_2, \ldots, X_n) \) is the interface of the method, and \( \tau_1 \times \tau_2 \ldots \times \tau_n \rightarrow \tau \) is the signature of the method.

\[
\square
\]

Since predicate names can be considered as constant function id-terms, all rules in LLO are methods.

With function id-terms, parametric polymorphism is achieved automatically. A functional programming flavor within deductive data model is nicely integrated into our system. As pointed out by Abiteboul and Grumbach [AbGr 87a], the absence of functions from the relational model together with their natural importance lead to the introduction of function dependencies. Function id-terms provide a way of expressing the functional dependency among related data objects. When function id-term is used inside a literal, it denotes an oid based on its arguments.

**Example 7.2.3** The program in Example 6.1.6 can be turned into a generic method, \( tr \), which calls other methods \( adj \) and \( g \), to finish the transformation.

The first two rules in Example 6.1.6 specify a concrete transformation, since \( p \) is an object identifier, not a function id-term. In order to have a generic method,
we replace $p$ at the beginning of both rules by $tr(R)$, and all other appearances of $r$ by $R$. $tr(R)$ together with its type is the interface of the method.

$$tr(R)(adj(R, X)) : -R(X, Y).$$
$$tr(R)(adj(R, Y)) : -R(X, Y).$$

When this method responds to a message $tr(r)$, $adj(R, X)$ will be instantiated, say, to $adj(r, a)$. The value of this oid will be assigned by method $adj(R, X)$. Methods $adj(R, X)$ and $g(R, X)$ are the same as defined in Example 6.1.6.

It is important to note that method $g(R, X)$ and method $adj(R, X)$ are not mutually recursive, since the ground id-term of $adj(R, X)$ is an oid, a first order element. Our interest is the intension part of the object. Method $g(R, X)$ simply groups together all those oid’s. This benefit comes from the separation between intension and extension part of an object. Interested reader may look at the example in IQL [AbKa 89b] and see the difference.

A method $f$ with signature

$$f : \tau \rightarrow \tau'$$

is a method defined on class $\tau$. A set of methods can be defined for each class this way. To achieve encapsulation, all access to the state of an object must use the methods defined on its class. Method interfaces should provide all the information the users need to know about objects in a class. Also the signature of a method plays the role of connecting the method with its classes/types. On the other hand, methods can also be defined to achieve computation abstraction or modularity of programs. We believe that not all objects in a database need to be encapsulated. For some classes, objects are grouped together because they have common properties. In these cases, methods are used for code sharing.

In many situations $n$ objects need to be retrieved. To accomplish this, a method can be defined on one class, using the rest of the classes as its arguments [KiLa 89]. In case those classes are symmetric, this approach is not natural [ZdMa 90]. $n$ methods need to be defined, one for each class. The generic method with arity $n$ discussed here may be a solution to this problem.
The advantages of using id-term in method definition are summarized in the following:

1. Methods are rules. Rules without function symbol in their heads define methods on individual objects (concrete methods) while rules with function symbols and variables in their predicate id-terms of the heads define methods on types/classes (generic methods). Methods and rules now have a uniform representation.

2. Identity denotation for the resulting object and method call are combined together.

3. Functional programming, object-oriented features and logic programming are integrated, because method interfaces are actually consistent with functional paradigm.

Another important feature of our approach is lazy evaluation. Since a ground function id-term denotes an object identifier, a method call through function id-term inside a literal need not be evaluated unless the value assigned to the oid is required.

**Example 7.2.4 [AbGr 87a]** Let p be an object with type [A, C], where C is a class name of type {{D, D}}. By using the trans-closure method in Example 7.2.1, we can transform the second attribute into its transitive closure and hide its detail by an oid.

\[ q(X, \text{trans-closure}(Z)) \Rightarrow -p(X, Z). \]

The program copies X over and get the transitive-closure of Z by a method call to trans-closure.  \(\Box\)

### 7.3 Identity Denotation and Population

As shown in the last section, when function id-term is used in defining method, it plays three different roles:
• Identity denotation. Each ground function id-term denotes an object identity.

• Oid population. The rule(s) defining a method will be applied to assign a value to the oid the function id-term denotes.

• Method specification. The type of a function id-term is the signature of the method, which specifies types of the import and export objects. The signature is used in type checking to ensure correct inheritance.

In the following we will examine the roles played by a function id-term when it is used as an argument inside a literal and explain how oid is populated. Oid denotation is concerned with object intension (object identity) and oid population is about object extension (value assigned to the object). As indicated in [CKW 89], the separation of intension and extension simplifies the semantics. In this section, we also compare our model with other models, specifically COL [AbGr 87a], LDL [Bee et al. 87], IQL [AbKa 89b] and ILOG [HuY 90].

Function id-term as object identity

When a function id-term appears inside a literal, it denotes an object identity. Here we are concerned about the intension of the object. The following is an example:

\[ R_2(X, f(X)) : \neg R_1(X, Y) \]

in which \( f(X) \) means for each value \( X \), there is an object identity. This oid is the second attribute of the tuple. The existence of the oid depends on the arguments of the function id-term. The type of the function symbol actually represents the functional dependency [Ullm 88, YuOz 86]. For the above example, \( f(X) \) specifies that the existence of this oid depends on \( X \), that is, \( f : X \rightarrow f(X) \). For each ground function id-term, if there exists a method with the function id-term as interface, then the oid it denotes get value from the method. Otherwise, it takes \{ \} as its value if it is a set typed object and \textit{null} for the rest of types.

Specifically, an oid used in a literal is the intension of the object (remember an oid is a constant function id-term).

Function id-term as oid population
Objects are populated by methods. When a function id-term \( f(X_1, \ldots, X_n) \) appears as predicate term in a literal, a method call \( f(X_1, \ldots, X_n) \) is activated to populate the invented oid the function id-term denotes, that is, to define the extension of the object. If there is such a method, it will specify how to form a value for the object. In this sense, function id-term in a literal adequately models the procedural data in Postgres [StRo 86].

As has been pointed out, a program is a set of methods, and methods assign values to objects.

**Example 7.3.1** The following is a method that will generate a power set \( \text{powerset}(R) \) for a given object \( R \) with set type. We do not specify the type of the import parameter here. However, the reader should bear in mind that all the objects and values are typed.

Empty set is always an element of the power set.

\[
\text{powerset}(R)(\{ \})
\]

If \( X \) is an element in \( R \), then \( \{X\} \) is an element of the power set.

\[
\text{powerset}(R)(\{X\}) : - R(X)
\]

Further, if \( X \) and \( Y \) are element of the power set, then the value of their union is an element of the power set.

\[
\text{powerset}(R)(\downarrow \text{union}(X, Y)) : - \text{powerset}(R)(X), \text{powerset}(R)(Y).
\]

Here, \( \text{powerset}(R) \) is a method interface, \( R \) is an import variable, \( \text{union}(X, Y) \) is function id-term and its method is defined in Example 6.1.5 in Chapter 6.

Both \( \text{powerset}(R) \) and \( \text{union}(X, Y) \) specify oid denotation and population. While \( \text{powerset}(R) \) creates one oid for each method call with ground term for \( R \), \( \text{union}(X, Y) \) denotes oid for each \( (X, Y) \) pair. Since we only interested in power set of \( R \) instead of a set of objects. \( \downarrow \) is used to discard the oid while retaining the value of \( \text{union}(X, Y) \).

Oid invention and population are first proposed in IQL [AbKa 89b]. In that model, variables in the head but not in the body of a rule indicate oid invention.
The invented oid's are populated separately by other rules. Our model provides a meaningful alternative. With function id-terms, we achieved oid denotation (function id-terms inside literals) and population (method definition by function id-terms). Function id-terms are more precise in determining how oid's will be created, as they contain information about functional dependency [AbGr 87a]. In IQL, since oid invention is realized by variables in the head but not in the body, the number of oid's that need to be invented is implicitly determined by the variables in the body. Oid invention and population must be done separately, which may cause some confusion.

Example 7.3.2 The following program segment from IQL [AbKa 89b] invents two oid's for each element appeared in relation $R$ (cf. Example 6.1.6):

$$R_0(x) : -R(x, y)$$
$$R_0(x) : -R(y, x)$$
$$R'(x, p, p') : -R_0(x)$$

We can achieve the same effect in our model by the following rules:

$$R'(X, oid_1(R, X), oids(R, X)) : -R(X, Y)$$
$$R'(X, oid_1(R, X), oids(R, X)) : -R(Y, X)$$

Where $oid_1$ and $oids$ are function symbols and $oid_1(R, X)$ and $oids(R, X)$ are function id-terms. Relation name $R$ in $oid_1(R, X)$ and $oids(R, X)$ are put there in case oid population needs more information, such as defining methods $oid_1(R, X)$ and $oids(R, X)$ by function id-terms. If we only want oid to be invented, or we can populate the invented oid without this information, we can remove $R$ from $oid_1(R, X)$ and $oids(R, X)$.

ILOG [HuY 90] also has an oid invention mechanism. The star symbol "*", only allowed as the first argument in the head, is a special notation for oid invention. The oid invention and population is done by one rule. The first argument indicates oid invention, and the rest arguments are components of the value assigned to the oid. For simplicity, ILOG is restricted to flat data only. More importantly, two objects may not have the same value\(^2\). An invented oid

\(^2\)In ILOG, two objects may have the same value by projection after oid invention. Note
can only take tuple value. ILOG can be easily represented in our model.

Example 7.3.3 The following rule in ILOG [HuY 90]

\[
\text{int-honaff}(*, d, g) \leftarrow \text{enrollment}(c, g), \text{has-grade}(c, g, 'A'), \text{offered-by}(c, d), \text{grad}(g).
\]

can be translated to LLO

\[
\text{int-honaff}(\text{dgpair}(D, G)) : \neg \text{enrollment}(C, G), \text{has-grade}(C, G, 'A'), \text{offered-by}(C, D), \text{grad}(G).
\]

\[
\text{dgpair}(D, G)(D, G).
\]

Actually all ILOG rules involving oid invention can be translated by the same pattern. □

IQL and ILOG introduced objects into the deductive data language and developed techniques for oid invention and population. While inheritance is discussed in IQL, its major concern is about data inheritance. It is not clear how method is defined and how method inheritance could be achieved. IQL adopts the data inheritance semantics by Cardelli [Card 88], in which subtypes have more attributes. We introduced meta variables into types and achieved the effect of data abstraction and information hiding. Furthermore, by using function id-terms, we introduce methods into deductive data language.

The concept of function id-term in our model bears some resemblance to the concept of data function in COL [AbGr 87a] where data function is used to collect data together for further reference. Like data grouping in LDL [Bee+ 87], data function is introduced to cope with the problem of complex objects in logic data language. Data function is also a tool for procedural data, such as the one introduced in Postgress [StRo 86]. Function id-term in our model is an extension of data function. It plays very important role in turning logical data model into an object-oriented deductive data model. First, it denotes object identity; second, it is used to specify method and method interface. Even though COL has set constructs, it does not allow predicate names as arguments of data functions that in ILOG, the invented oid's are only addressable through a relation. It is not clear this is an advantage or a drawback.
or literals. Inspired by HiLog [CKW 89], in our language id-terms can be used as arguments of function id-terms as well as atoms. The language we defined has a higher order syntax and first order semantics (without the exposure operator), as has been shown in Chapter 4. By using function id-term, method can be defined and parametric and inheritance polymorphisms are easily achieved.

### 7.4 Method Inheritance

Let \( f(t_1, \ldots, t_n) \) be a method interface with signature

\[
\tau_1 \times \cdots \times \tau_n \rightarrow \tau,
\]

where \( \tau \) is a class name. Since constant oid is a special case of function id-term, the method interface given above is general.

Suppose \( f(t'_1, \ldots, t'_n) \) is a message (query) that is sent to the database, where \( t'_1, \ldots, t'_n \) are terms of types \( \tau'_1, \ldots, \tau'_n \) respectively and \( f(t'_1, \ldots, t'_n) \) is of type \( \tau' \). This message is calling for a method \( f(t'_1, \ldots, t'_n) \) with signature

\[
\tau'_1 \times \cdots \times \tau'_n \rightarrow \tau'.
\]

Should method \( f(t_1, \ldots, t_n) \) respond to this message?

To dispatch this message to method \( f(t_1, \ldots, t_n) \), the following typing requirements should be fulfilled:\(^3\)

1. \( \tau'_1 \preceq \tau_1, \ldots, \tau'_n \preceq \tau_n \);
2. \( \tau \preceq \tau' \).

Note that, the argument types of messages should be subtypes of the method, while the resulting type of the method should be a subtype of the message type.

The first requirement states that methods defined on supertypes are applicable to subtypes. The second requirement simply says that the resulting object can only be used as a supertype object. Both requirements are supported by meta variables in the data model described in Chapter 3. The unification algorithm in [Lloy 84] has been extended to check these requirements in order to

---

\(^3\)In the following presentation we use \( \preceq \) to denote \( < \) or \( = \).
dispatch a message to the proper method. Thus, in LLO, methods on one class
and on several classes/types have the uniform representation and also can be
handled in a uniform way.

Each meta variable has a domain, indicating the values (i.e. types) the meta
variable may take. Consequently, if a variable in a literal is of type $m$ which
is a meta variable, this variable can take values of type $\tau$ if $\tau \in \eta(m)$. So
from the view point of $m$, there is no structural differences among types in its
domain. Meta variable functions as a cushion between a type and its subtypes
and makes objects in a class have a uniform structure. On the other hand, Some
information in the resulting object could be hidden by a meta variable in the
supertype. For the example discussed in Chapter 3, if there exists a method with
output type $\text{Studenttype}$, then the resulting object of this method can be used as
an object of $\text{Persontype}$ by ignoring $\{\text{Course}, m_1\}$, but may not be used as an
object of $\text{TAType}$. Even though the object may be a TA, the meta variable $m_1$
in $\text{Studenttype}$ hides the details. and make it impossible to distinguish between
students in general and TA in particular.

Meta variables also provide a flexible way to model parameterized types
[Card 88]. The set type $\{m\}$ discussed in Section 3.2 is one example. All meth-
ods defined on class of type $\{m\}$ will be inherited by classes of type $\{T\}$ if
$T \in \eta(m)$. More importantly, the domain of a meta variable put a constraint
on the parameter. For example, instead of modeling one parameterized stack
for all types, we may set up several generic stacks $\text{stack}(m_1), \ldots, \text{stack}(m_k)$.
Each meta variable has a disjoint domain. Classes of stack objects may inherit
methods from a specific generic stack. This feature is useful for implementation
and efficiency purposes.

The following example illustrates how meta variables support method inher-
itance.

Example 7.4.1 Let $\text{Person}$ be a class name of type $\text{Persontype}$ with attributes
Name, Birth-year and a meta-variable $m$. Let $P$ be an object in $\text{Person}$. The
following is a method calculating the age of a person:
\[ age(P)(N) : -P(\text{Name, Birth-year}, X), N = \text{Curr-year} - \text{Birth-year}. \]

where \text{Curr-year} is a system variable. Or simply as
\[ age(P)(N) : -P(\text{Name, Birth-year}), N = \text{Curr-year} - \text{Birth-year}. \]

The difference of this two rule is that we dropped the variable \(X\) whose existence corresponding to the meta variable \( m \) in the type. Since \( X \) is simply used as a place holder, we can omit it and assume there is always such a variable there. This treatment enhances the encapsulation of the program, as the programmer does not need to know the position of the hidden part.

Now, we want to get the age of a TA john, whose type is a subtype of \text{Persontype}. Suppose the value of john is ["john", 1965, [c-john, 1000]] where \text{c-john} is a set value representing all the courses john has taken. Now we pose the following query:
\[ age(john)(N)? \]

Since the type of john is a subtype of \text{Person} in the method \( age(P) \), the unification succeeds, therefore, turns the method into a concrete computation:
\[ age(john)(N) : -john("john", 1965), N = \text{Curr-year} - 1965. \]

The variable \( X \) in literal \( P(\text{Name, Birth-year}, X) \) in method \( age(P) \) corresponds to the meta variable \( m \) in Persontype. Since \([\text{Courses}, \text{sal}]\) is in transitive domain of \( m \). \( X \) will absorb [c-john, 1000].

\[ \square \]

\textbf{Note:} Since there may be variables in the predicate id-terms in a message, there is an essential difference answering queries between Datalog and LLO. In Datalog, a predicate name (constant) and its arguments together form a query. The predicate name (also relation name) and its arity are used to choose rules. The variables in the arguments are confined to attributes of the relation, which is always finite. In LLO, as well as in HiLog, variables are allowed to be used as arguments in the predicate term of a message. This may cause some problems. Although the function symbol provides information about which method should respond to this message, variables in the predicate id-term may not have finite domain. That is, if the type of a variable is a class name, it
ranges over the objects of the class. However, if the type of a variable is a type (not a class name), its domain maybe infinite (for example, integer). In both cases, many objects may be generated for the message due to different bindings of variables. Ambiguity may arise as what is the response to the message, the whole set of objects or only one of them. The same analysis can be applied to the case where variables are used as predicate terms. HiLog has the same problem. Recently, Ross [Ross 91] analyzed the semantics of HiLog and gave some results on preservation under extensions and restrictedness. More research is needed to ensure termination and safety.

Example 7.4.2 (Continuation of Example 3.1.2). Let \textit{Student} be a class name of \textit{StudentType}. \textit{Students} be a class name of type \{\textit{Student}\}. We define a method \textit{select-by-GPA} on \textit{Students} and \textit{Criteria} with signature:

\[
\text{Students} \times \text{Criteria} \to \text{Students},
\]

where \textit{Criteria} is the type for \textit{GPA}. Given an object \(S\) of type \textit{Students}, the method will select those students whose \textit{GPA} is greater than or equal to a \textit{Criteria} value. This method will use another method \textit{gpa} with signature:

\[
\text{Student} \to \text{GPA}.
\]

The method \textit{select-by-GPA} is defined as follows:

\[
\text{select-by-GPA}(S, \text{Criteria})(S') : S(S'), \downarrow \text{gpa}(S') \geq \text{Criteria}.
\]

where variable \(S\) is of type \textit{Students}. \(S'\) is of type \textit{Student}. Method \textit{gpa} is defined by the following rule:

\[
\text{gpa}(S')(\downarrow \text{average}(\text{Courses})) : \downarrow S'(\text{Name, Birthyear, (Courses, X)}).
\]

Method average takes a set of courses and returns an average value of the grades (the detail of this method is omitted here).

Since \textit{Tatype} is a subtype of \textit{StudentType}, we can apply the method \textit{select-by-GPA} to object of type \{\textit{T,A}\}, where \textit{T} is a class name of \textit{Tatype}. If \textit{tas} is
a set of TA, the unification algorithm will select method \( select-by-GPA \) for the following query:

\[
select-by-GPA(tas, 3.2) \?
\]

All those TA's whose GPA \( \geq 3.2 \) will be selected, and the resulting object, with type \( \{TA\} \), will inherit methods from its type.

The extra information (such as salary) in TA will not interfere with the methods defined above, since the meta variable \( m_1 \) in \( StudentType \) will overlook the information. If we apply \( select-by-GPA \) to a set of students, which may include some TAs, the set will be handled as if all the objects have a uniform structure.

It is amazing that a small extension to the unification algorithm [Llo 84], in addition to the concept of generic methods and little bit of analogy, makes method inheritance easy and straightforward.

### 7.5 Multiple Inheritance: A Discussion

In LLO, function name overloading is allowed. That is, one function symbol can be used to define function with different signatures. Overloading has been widely used in programming languages. For example, in Ada, the symbol \( + \) is used for integer addition, real number addition, virtual number addition, even array addition. A kind of information hiding or abstraction is achieved since the users do not worry which function should be used for specific arguments. The system will respond with the right function depending on the actual arguments provided.

In LLO, function id-terms are used as method interfaces. One function symbol can be defined many times as long as all those definitions can be distinguish by their signatures. A message starting with a function symbol will chose one of the implementations depending on the signature of the incoming message.
7.5.1 More Than One Methods Match with a Message

In LLO, method names can be overloaded. Different signatures, when associated with a method name (function symbol), specify different methods. When a message is sent to the database, the signature (type) of the message will determine which method should be used to respond to it. Even though two methods have the same name, different argument types will result in different computation. For example, in a department store database, we may apply a size method to shoe objects and slack objects. Since shoe and slack are objects of different types, two different methods will respond to the calls, with the sizes returned in different measurements.

When overloading is allowed, it is possible that more than one methods match with a message. In this case, which method should be invoked to respond to the message? For example, let

\[ f(t_1, t_2, \ldots, t_n)(X) \]

be a message with signature requirement:

\[ f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \]

and we have the following methods that match with the message (here we only give signature, and use superscripts to distinguish them):

\[ f^1 : \tau_1^1 \times \ldots \times \tau_n^1 \rightarrow \tau^1 \]

\[ f^2 : \tau_1^2 \times \ldots \times \tau_n^2 \rightarrow \tau^2 \]

\[ \vdots \]

\[ f^k : \tau_1^k \times \ldots \times \tau_n^k \rightarrow \tau^k \]

We define an order on the signature of those methods.

\[ f^i \preceq f^j \text{ iff } (1) \tau_1^i \preceq \tau_1^j \text{ for } l = 1, \ldots, n \text{ and (2) } \tau^i \preceq \tau^j \]
The first condition means use the detail of the input as much as possible and the second condition says that the inside structure of the output should be hidden as much as possible. If a minimum method exists, then the minimum method will be activated to respond to the message. This treatment also reflects the idea of *overriding*, that is, a method defined on subclasses will override those defined on superclasses. For example, we may define a method \( \text{gpa} \) on class *Student*, a method \( \text{gpa} \) on class *TA*. If we have a message \( \text{gpa} \) with an object from class *TA*, obviously the method \( \text{gpa} \) defined on *TA* will be used to respond to the message.

If we take attributes as methods, and if two superclasses have the same attribute name, the subclass has a difficulty as from which class it should inherit an attribute. Let's say one attribute \( f \) is defined on class \( C_1 \), another attribute \( f \) is defined on class \( C_2 \), and we have

\[
f : C_1 \rightarrow \tau^1.
\]

\[
f : C_2 \rightarrow \tau^2.
\]

If a message \( f(X) \) is sent to an object in class \( C \), which is a subclass of both \( C_1 \) and \( C_2 \), this is a special case of the above conditions. A minimum method will be chosen to respond to the message. More discussion on attributes as methods will be given in *Chapter 8*.

Note that after a method is successfully chosen, its success in responding to a specific message depends on the unification of the message and the method. While this section talks about chose a method with the right signature, unification process will start the real inference (*Chapter 8*).

### 7.5.2 Multiple Inheritance

In case there is no minimal methods among all the overloaded method definitions, a *multiple inheritance* situation occurs. There have been no satisfactory answer to this question. The most frequently proposed solutions are:
(1). Doubtful solution: No method will be used to respond to the message. Under this approach, the message simply fails.

(2). Declare precisely which method should be inherited. Only the designated method will be used.

Some languages such as C++ only allow tree structure in the class hierarchies. The above problem simply does not appear. Other languages such as Eiffel [Meyer 86] and CLOS [Keene 89] use the second approach, that is, the programmer specify which method to inherit in case an ambiguity occurs.

Method overriding can be achieved in LLO. A method defined on supertype can be overridden by a method with same name in the subtype. When the unification dispatches a message, it will search the class inheritance hierarchy bottom up to find the method with the "best" matching type. In LLO, several rules could be used to implement one method. In this aspect, LLO is basically different from other procedural object-oriented languages, where one method implementation is done by one procedure or function. Utilizing this feature, all rules of the minimum methods together will be used as the method definition to respond to the message in case there are more than one minimum method. That is, if we have minimum methods $f^1$ and minimum method $f^2$ in respect to message $f$, and $f^1$ is defined by rules $r_1$ and $r_2$, $f^2$ is defined by rules $r_3$ and $r_4$, then $r_1, r_2, r_3, and r_4$ together form a method to respond to message $f$.

However, in case the method output is of tuple type, there are some problems. For example, if we define a method age on Student, and another method age on Staff, it is not clear how to handle the result when we want to get the age of a TA. Actually we have the same problem if a method is implemented by more than one rule and the output is of tuple type. We propose the following solutions to handle this situation:

1. In case all rules have the same tuple $t$ as their output, then $t$ is the result of the method.

2. If more than one tuple generated by those rules, and those tuples are not identical, then return failure.
Of course, the best way is to prevent this from happening by copying a method from a chosen superclass, as has been done in Eiffel and CLOS. Consequently there will be a single minimum method that can be called upon to answer a message.

Note that the above analysis applies to methods with output of tuple types. For output of set types, there will be no conflict, and our approach works fine.

### 7.6 Summary

From the above observations, it is clear that

1) Typing plays an important role in method inheritance. Method inheritance is achieved following the type/class hierarchies. The unification algorithm should do the type checking dynamically to guarantee the correct inheritance.

2) Unlike first order logic, where predicate symbols must be the same for a successful unification, predicate id-terms play an important role in the unification process. The predicate id-term in the message and the one in the method should be unified first according to the typing requirement 1 in Section 7.4. Then the arguments of the message and the method are unified satisfying requirement 2.

A discussion on static type checking of multi-methods by Agrawal, DeMichiel and Lindsay [ADL 91] appeared in OOPSLA’91. Different approaches are compared there.
Chapter 8

More on Programming in LLO

There are different views on what is a method in an object-oriented database system. In the literature, attributes, features and methods are often used interchangeably. In our model, we make it clear that an object has a state (value). Methods are used as interfaces of objects. Usually methods are defined on classes (generic methods), but can also be defined on objects directly (concrete methods). Our point is that our approach is more flexible. Features in F-Logic and Login can be represented in LLO. Similar to functions in COL, LLO methods can be defined outside LLO, i.e., in other paradigms as far as method interfaces (signatures) are clearly specified. If we like, type information and methods on objects can also be queried.

8.1 Methods and States

In feature based language, an object has features which are also called methods and can be accessed through the object identifier [AiNa 86a, KiLa 89, ChWa 89]. The problem with this approach is that it is not clear which features are accessible and which are not. In F-Logic [KiLa 89], a view concept is introduced to solve
this problem. The approach introduced there is similar to the view concept in relational model, that is, some features are exposed while others are hidden. Features can be accessed one by one or in a group by associating them with the oid.

In LLO, we emphasize the difference between state and methods of an object. The state of an object is hidden from any access. Encapsulation is achieved the same way as procedural object-oriented languages. Any access to the state of the object must go through the interface of the object—methods. Meta variables in types overcome the structural difference between types and subtypes, making method inheritance possible.

On the other hand, LLO provide the flexibility of defining methods directly on objects (without resort to the states) and expose part of the states through simple methods.

### 8.1.1 Defining Methods on Objects Directly

In many situations, we need to define methods for a class of objects individually. There is no generic rule (rules) to cover the general case. In LLO, the method definition mechanism allows us to handle this situation.

For example, if we have a class $C$ including objects $o_1, o_2, o_3$ and $o_4$. $o_1$ and $o_2$ belong to a subclass $C'$. A method $f$ can be define on class $C$. But for objects in class $C'$, we can define the method in a generic way. For object that is not in class $C'$, the effect of method $f$ must be specified individually.

The following rule

$$f(X : C')[X_1, X_2] : -X_1, \ldots]$$

specifies the generic part of the method, and the rest of the method is defined on individual object:

$$f(o_1 : C)[a_1, b_1]$$

$$f(o_4 : C)[a_4, b_4]$$

So if a method call $f(X : C')$ is placed, one of the rules will be used depending on the actual parameter instantiating with $X$. 
By the above discussion, an object can be seen as a class with only one element, i.e., the object itself. By this assumption, all methods are still defined on classes. This reflects the flexibility of our approach.

8.1.2 States of Objects and Features

In Login, F-Logic and C-Logic, feature and method are basically the same concept, which in turn, is comparable to the concept of attribute in relational models. It should be pointed out that there is no difficulty in representing the state of an object in LLO by features. For object \( o \) of type \([T_1, T_2, \ldots, T_n]\), we can define method \( f_1, \ldots, f_n \), each \( f_i \) corresponds to an attributes \( T_i \). If the state of \( o \) is \([a_1 : T_1, \ldots, a_n : T_n]\), then we can define method \( f_i \) on object \( o \) as follows:

\[
\begin{align*}
    f_1(o)[a_1] \\
    f_2(o)[a_2] \\
    \vdots \\
    f_n(o)[a_n].
\end{align*}
\]

If we like, we can see the state of an object as a short hand for a group of methods.

The concept of method in LLO is more general than those in F-Logic and C-Logic, since in LLO, a method can be defined on several classes. In F-Logic, all methods must be activated within one object environment. It is obvious that when several objects are involved in the computation, this treatment is not intuitive.

8.1.3 Common Components for Objects in One Class

There are situations in which all the objects in a class share the same components. For example, all TA in a university must maintain a gpa of 3.0, let's say. There are two approaches to handle this case:

1. Common components are saved in the class and are accessible to all objects:

2. Common components are duplicated in each object.
Obviously the second approach has some update anomalies. In LLO, we can define a method on the class:

\[ \text{minimumgpa}(X : TA)(3.0). \]

This information will be available for any objects in class \( TA \).

Some information, such as average grade of all TA (\( \text{avg} TA \)), is based on the class as a whole but available to each object. \( \text{setTAavg} \) is defined on class \( TAs \) of type \( \{ TA \} \), that is, a set of TAs. This method takes an object whose value is a set of TA and gives the average grade. Using this method, we can define \( \text{avg} TA \):

\[ \text{avg} TA(X : TA)(\downarrow \text{setTAavg}(TA : TAs)). \]

Where \( TA \) in the head is used as a type (class), and \( TA \) in the body is used as an oid whose type is \( TAs \). This example also explain the double role played by class: a class is a type and also a set valued object.

### 8.2 Externals

Function id-terms in LLO are used as method interfaces in method definitions. Methods are implemented by rules (or sets of rules). We can use externals the same way as in COL [AbGr 87a] to implement methods. An external is a function or a method that is evaluated "outside" LLO, i.e., using other paradigms.

Multiparadigm programming has many advantages. Some applications can be easily programmed in one programming paradigm while awkward in others. This phenomenon has stimulated research in integrating multiple programming paradigms together [GM 86, GM 87]. This thesis is one of the efforts trying to bring object-oriented features into logic data language. Externals provide an opportunity for LLO to solve problems that can be better handled in other paradigms.

Externals have been widely used in relational data models. Many aggregate functions such as count, avg, etc. are called built-ins whose semantics can not be
handled by the formal relational calculus. Only system provided externals exist in relational languages. Users are not allowed to define new ones.

The following example from [AbGr 87a] illustrate the role played by external functions. Since external functions are functions, function id-terms are interfaces. The only difference is that external functions are not defined by rules. They may be implemented in a procedure in C, C++ or even a function in Lisp. Moreover, in LLO we only concerned about their interfaces. Let \( P \) be a person. Consider a function \( \text{children} \) of type \( \text{person} \to \text{childrenset} \), where the type of \( \text{childrenset} \) is \( \{\text{person}\} \). The following rule selects persons who has more than three children and group them by number of children using a function \( f : \text{int} \to \text{persons} \) where \( \text{persons} \) is of type \( \{\text{person}\} \):

\[
f(N : \text{int})(P : \text{person}) : \neg \text{count} (\text{children}(P)) = N, N \geq 3.
\]

Where both \( \text{count} \) and \( \text{children} \) can be seen as external functions.

The introduction of externals may result in non-termination of programs if externals are non-terminating. It is required that all the arguments of externals are instantiated with ground terms when externals are called. Another restriction is that externals do not depend on other predicate terms in their definition. And further, each external function must be terminating when its arguments are instantiated with ground terms. With those restrictions, we can take externals as normal function id-terms.

### 8.3 Query about Type Information

As the type system in a database programming language is getting more and more complicated, there have been research trying to provide users with facilities to browse types. In F-Logic [KiLa 89], type information are formulated by rules and can be queried upon the same way as any other data. For example, we can find all the sellable items by collecting all objects which have a price attribute. We can also find all objects which, directly or indirectly, have some specific property. However, if users abuse this power, it will ruin all the efforts on
encapsulation and information hiding. Measures must be taken to protect the structures of objects from intruders.

In LLO, the type information is also part of the data and can be accessed in a restricted manner. That is, access to types must use methods defined on objects. For any object, we consider its type as part of its state. Methods could be defined to access the state of the object as well as the structure of the state. Furthermore, we can even define methods that browse the type (class) hierarchies and collect methods that are applicable to a specific object or class. For example, the following rule

\[
\text{method}_\text{on}(o : \text{object})(X) : -X(o)
\]

will give us all methods that is applicable to object \(o\). *Anytype* here means that the type there is not important. In a similar way, we can get the methods defined on a specific class \(C\):

\[
\text{method}_\text{on}(C : \text{class})(X) : -X(O : C).
\]

If we use \(P(..., t, ...\) to denote terms that has a component term \(t\) in it, then we can define a method to find methods defined on several classes. For example,

\[
\text{method}_\text{on}_\text{classes}(C_1 : \text{class}, C_2 : \text{class})(M) : -M(... X : C_1, ...),
\]

\[
M(... X : C_2, ...).
\]

It is also straightforward to define a method which gives all methods whose output belong to a specific class.

The following query list all the objects in type \(\tau\):

\[
O : \tau?
\]

If \(\tau\) is a class, objects in \(\tau\) will be listed. If \(\tau\) is not a class name, then all oid's of type \(\tau\) will be listed. It is also straightforward to write a rule that will give all the types of which an object \(o\) is an instance.

\[
type \text{of}(o)(T) : -o : T.
\]
Remember that if \( o \) is of type \( T_1 \) and \( T_2 \) is a supertype of \( T_1 \), then \( o \) is also of type \( T_2 \).

A method can be defined to print the states of all objects in a class:

\[
\text{print}(C : \text{class})(\downarrow O) : \neg \neg O : C.
\]

For a given class \( C \), a method is defined to get all its superclasses:

\[
\text{superclass}(C : \text{class})(X) : \neg C \prec X.
\]

A similar method can be defined to get subclasses.

Methods can be defined to access the type and state of an object. For instance, the following method

\[
P : \text{work-study} : \neg \neg P : \text{student}.\text{student} o [\text{Name}, \text{Birthyear}, \text{Salary}].
\]

means that if \( P \) is a student and \( P \) has a salary, then \( P \) is an object in work-study program.

### 8.4 Integration of Functional Programming and Logic Programming

Reddy [Reddy 86] surveyed the integration of functional programming and logic programming. In [DFP 86], the unification of functional and logic programming languages is further addressed. Every language paradigm has its strength and weakness. We claim that LLO has made its contribution toward integrating the goodies of both functional programming and logic programming. The bridging concept is played by functional id-terms.

As indicated in [DFP 86], while logic languages are mostly untyped, a majority of functional programming languages, such as ML, are strongly typed and allow polymorphism. Generic methods can be designed and shared through type hierarchies. The function id-terms, used as method interfaces as well as object id’s in association with the data model, just achieved that. We have addressed
this aspect in detail in Chapter 7. The ability to write higher order functions increases the expressive power of the language, enabling many related algorithms to be realised as specific instances of one generic function. In functional languages, functions are first class citizens and can be passed as parameters and returned as values. The function symbols in LLO are constants. Therefore, there is no problem to pass them around during the program evaluation.

Logic languages make no commitment as which variables are to be considered as inputs and which ones are considered as outputs. On the other hand, in functional programming, all parameters of the called function must be instantiated. The former uses unification to unifies a goal clause with the head of a rule, potentially producing bindings for the variables in both the clause and the goal [Reddy 86]. The later uses matching process to get variables in the function instantiated. It is clear that the unification is a process of two way traffic while the matching process is one way only. (This is called input-output directionality in [Reddy 86]). Essentially one logic program can represent many functional programs, depending on which variables in the program will be instantiated. Results from logic program may not be ground terms. There may be variables in the output of logic program and usually more than one result will be given. On the contrary, functions are deterministic, and only one ground answer will be given. The function id-terms in LLO are advantageous in that they are used to define methods, so polymorphism can be achieved. Unification is used in message dispatching, so two way binding is achieved (see Chapter 5).

In summary, function id-terms in LLO takes advantage of functions in functional programming and the benefit of logic programming framework. Those two paradigms are effectively integrated in one deductive language, LLO.
Chapter 9

Fixpoint Semantics

This chapter discusses the semantic issues of LLO programs. First, problems with equal set semantics are deliberated. Then subset semantics based on semi-lattice structures is presented. There are drawbacks in both semantics. Based on the analysis, we propose an approach to program subset semantics in the equal set semantics framework, taking advantages of both. Finally, we extend the stratification concept and prove that all stratified LLO programs have minimal models generated by the fixpoint operator.

9.1 Introduction

There have been a lot of work done to extend logic languages with complex objects. One of the most important constructs of complex objects is set. Set plays an important role in nested relational data models. LDL [Bee+ 87] introduced set constructs and a grouping mechanism to transform a flat structure into a set in a rule based framework. LPS [Kuper 87] introduced sets and logical quantifiers into Horn clauses and its power is compared to LDL. In COL [AbGr 87a, AbGr 87b], in addition to sets, data functions are introduced as data grouping tools. Another important research direction is to integrate object-oriented features with deductive data language. Object-identifiers (oid’s) is first
introduced into logic by Kuper and Vardi [KuVa 84]. Identifier invention is developed in IQL [AbKa 89b, AbKa 89a], in which the value-oriented data (relations) and object-oriented data (classes and oid’s) are processed by rules. Methods are incorporated into F-logic [KiLa 89]. The language LLO [LoOz 91a, LoOz 91b] achieves methods and inheritance based on a data model in which meta variables in types make objects in one class have uniform structure. As has been shown in the literature, while the introduction of oid is relatively easy to handle, the existence of set structures makes semantics hard to analyze. The question is, if we have set \( A \) and set \( B \) of the same type \( (A \cap B \neq \emptyset) \), what is the result if we try to 1). unify \( A \) and \( B \); 2). make a natural join on \( A \) and \( B \); and 3). verify \( B \) based on \( A \) (or vice versa).

### 9.1.1 Intersection of Models

Most deductive languages with complex objects extend the Prolog/Datalog semantics, which we call *equal set semantics*. That is, \( A \) and \( B \) are unifiable and joinable if \( A = B \). The intersection of two models in equal set semantics is model intersection. The semantics of those languages is further complicated by those set construction mechanisms such as data grouping in LDL [Bee+ 87] and data function in COL [AbGr 87a, AbGr 87b]. One of the problems is that some programs do not have a unique minimal model. And the intersection of models for a program may not be a model.

- **Two Sets Are Not Related Unless They Are Equal**

In the following example, even though set \( \{1,2\} \) in the literal \( p(\{1,2\}) \) and set \( \{2,3\} \) in the literal \( p(\{2,3\}) \) have a common subset, those two literals are unrelated since those two sets are not equal. Consequently, intersection of two models for a program may not be a model.

#### Example 9.1.1

The following program is from LDL [Bee+ 87]:

\[
p(<X>) : \neg q(X)
\]

Possible models for this program are:
\{q(1), q(2), p(\{1, 2\})\} and \\
\{q(2), q(3), p(\{2, 3\})\}.

However, the intersection \{q(2)\} of these two models is not a model since \\
p(\{2\}) is not in the intersection. \qed

HiLog [CKW 89] makes this point more explicit by coding sets into functions. 
Two sets are coded into the same function (thus comparable and joinable) if only 
if they are the same set.

- **Positive Programs are not Monotonic**

In COL [AbGr 87a] as well as in LDL [Bee+ 87], positive programs are not 
monotonic in general due to the semantics given to sets and the power of grouping 
and data function. For example, the following COL program \(P\) consists only 
one fact:

\[
q(F) : -
\]

where \(F\) is a data function. We have

\[
T_P(\{1 \in F\}) = \{q(\{1\}) \not\subset q(\{1, 2\})\} = T_P(\{1 \in F, 2 \in F\}),
\]

where \(T_P\) is the fixpoint operator [ADW 87]. From program \(P\), we can see clearly 
that \(T_P\) is not growing. The set in a predicate causes trouble. The set \(\{1\}\) in 
\(q(\{1\})\) and the set \(\{1, 2\}\) in \(q(\{1, 2\})\) seem not related and not comparable.

More importantly, negation can be simulated in LDL and COL. The following 
COL program is from [AbGr 87a]:

\[
t \in F(t) : -p(t) \\
a(t, F(t)) : - \\
q(t) : -a(t, \{\})
\]

It is obvious that \(q(t)\) is equivalent to \(\neg p(t)\), since \(\{\}\) cannot be unified with 
\(F(t)\) if \(F(t) \neq \{\}\). \(^1\)

Similar argument applies to LDL as well. So negation is redundant in both 
languages. Positive programs lost the property of monotonicity. This makes the 
semantics hard to analyze.

---

\(^1\) If \(p(t)\) is true, then \(F(t)\) is not empty. \(a(t, \{\})\) is false, so is \(q(t)\). If \(p(t)\) false, \(F(t)\) will be 
empty and \(a(t, \{\})\) is true. \(q(t)\) will be true.
From Vadaparty [Vada 91], the grouping rule in LDL and data function in COL can express non-monotonic queries such as EVEN and EQUAL on set terms. It is interesting to note that these queries can not be expressed by other non-monotonic query languages such as stratified-Datalog, fixpoint queries, Bounded loop queries, etc.

9.1.2 An Example

Before we start talking our approach, let's first see an example.

Example 9.1.2 Let [College-name, {[Year, {Graduate}]}] be the type of the Univ class, meaning the College has a set of Graduates in that Year, and [Company-name, {[Year, {Scientist}]}], revenue] be the type of the Company class, meaning that the Company has recruited a set of Scientists in a specific year and has revenue. Graduate and Scientist are subclasses of class Person.

Further, we have the following objects in class Univ:

Univ : uutech

Univ : manc

The value of each university is declared as follows:

uutech : ['Univ. of Universal Tech', {[1985, {danny, leo}],
[1987, {john, mary, rich, larry, henry}]]]

manc : ['Manhattan College', {[1988, {cohn, missy, bryan, mike}]}]

and we have the following objects in class Company:

Company : gec

Company : glm

and each company has the following data:

gec : ['General Engine CO', {[1987, {john, mike, lee, harry, rich}]}, 30]

glm : ['Global Machine', {[1987, {henry, tom, mary}],
[1988, {larry, bryan, mary, vincent}]}, 45].

Suppose we are asking the following query:

*Find out those students who graduated from a college and entered a company the same year.*

To answer this query, consider the following program:

univcom(X1, X3, X4) :- Univ(Y), Y[X1, X3], Company(Z),

-
By the join-by-equal semantics, a university and a company is joinable if and only if the set of year and graduates tuples and the set of year and scientists tuples in the company are equal. This is a rare situation in the real world. For the above database, it is clear that the result will be empty. A solution in HILOG-R will be explained in the next subsection.

LDL and COL do not have a first-order semantics according to Chen, Kifer and Warren [CKW 89]. In HiLog [CKW 89], a higher order logic language, extensional and intensional parts of an object is separated, and set terms are encoded into functions. While first order semantics is achieved, HiLog is based on join-by-equal semantics.

9.1.3 Subset Semantics

Chen and Kambayashi [ChKa 91] proposed an approach to the semantical problem in a simple nested relational model with rule based language HILOG-R (also see [ChenQ 89, ChenGard 88]). Based on strong typing and unique name assumption (UNA), the idea is ungrouping or flattening the set, and then making inference on the ungrouped set of facts. The final results are composed of a set of flattened facts and need to be grouped together in order to recover the complex structure of the database (this is secured by the unique name assumption). In this approach, variables take only atomic values (leaf level in the complex objects structure). the join-by-equal is done after the ungrouping operation, i.e., at the element level. The effect is actually join-by-subset for set terms, and join by equal for other terms (whose ungrouped form do not change). The joining results of two set terms is their intersection. For example, the result of joining on two appearances of \( \{X\} \), one instantiated with set \( \{a, b, c\} \) and another with set \( \{a, d\} \) will be \( \{a\} \) instead of empty set. Partial order is defined on terms and partial inclusion similar to set membership is based on the partial order. They proved that each program has a least minimal model under this semantics. In order to get a subset answer to the query in Example 9.1.2, A HILOG-R program
looks like this (with attribute names):

\[
univcom(X_1, X_3, X_4) : \neg Univ(Y), Y[X_1, \{X_2, \{X_3\}\}], Company(Z), Z[X_4, \{X_2, \{X_3\}\}], X_5.
\]

Flattening operations are required before the join operation. Then natural join is done on the flattened facts. Finally grouping by UNA to recover the structures.

While in HILOG-R, each program has a unique least minimal model, there is room for improvement.

1. In LDL [Bee+ 87] and COL [AbGr 87a, AbGr 87b], program evaluation does not need ungrouping or flattening the facts. The ungrouping process in HILOG-R [ChKa 91] is time and space consuming.

2. HILOG-R is value oriented. Users need to know the structure in detail (leaf level) to write a program. There is no encapsulation or abstraction.

3. How to handle empty set and negation is not discussed. If there are empty sets in a relation, the flattening operation will lose information, which is a well known obstacle in nested relational data model when empty set is allowed without incorporating null values into the model. Since negation can be simulated by empty set in COL and LDL, it is necessary to consider those two problems together as one semantic issue.

Jaeschke and Schek [JaSc 82] proposed “intersection-join” in addition to the natural join for set attributes in their nested relational algebra early in 1982. Two sets are joinable if they share a common nonempty subset. This is the earliest subset semantics we know of. But the definition is not extended to multiple levels of sets. Bancilhon and Khoshafian [BaKh 86] proposed a calculus for complex objects (named CCO in the literature) in which a lattice structure is built on all complex objects (or values). Two values are joinable if and only if their joining components have a common value below them in the lattice. In CCO, if a value is a fact, then all the values downward in its lattice are facts also. So we can say CCO is a form of subset semantics. But CCO is not strongly
typed. It allows a subtuple (tuple with fewer attributes) to be inferred from a tuple. If an oid is associated with the tuple, each attribute can be addressed separately or several attributes be operated together. Unfortunately, CCO is value oriented and the whole database is organized as a single value. Our work can be considered as an extension of CCO in the object-oriented environment.

It should be pointed out that F-Logic [KiLa 89] also uses a subset semantics similar to that of HILOG-R, even though the authors did not say so explicitly. For a data F-term

\[ P[\ldots; SetM_j \cap S_{j,1}, \ldots, S_{j,l_j} \rightarrow \{T_{j,1}, \ldots, T_{j,n_j}\}; \ldots]. \]

its atoms include:

\[ P[SetM_j \cap S_{j,1}, \ldots, S_{j,l_j} \rightarrow \{\}] \]
\[ P[SetM_j \cap S_{j,1}, \ldots, S_{j,l_j} \rightarrow \{T_{j,1}\}] \]
\[ P[SetM_j \cap S_{j,1}, \ldots, S_{j,l_j} \rightarrow \{T_{j,2}\}] \]
\[ \vdots \]
\[ P[SetM_j \cap S_{j,1}, \ldots, S_{j,l_j} \rightarrow \{T_{j,n_j}\}] \]

Obviously, if an F-Logic formula containing a set term \( S \) is a fact, then after replacing this set with one of its subsets (even if an empty set), it is still a fact. This is exactly what we mean by subset semantics.

### 9.1.4 Our Approach

LLO is developed for object-oriented database queries. It is strongly typed and each object has a unique identifier. Higher order syntax makes complex object representation intuitive and straightforward. LLO separates an oid and its value to achieve encapsulation. Objects are organized into classes. An exposure operation \( \downarrow \) is proposed to get the value of an object if necessary and appropriate. In this Chapter, we try to formalize the subset semantics in an object-oriented framework. We show that the subset semantics can be programmed in the framework of equal set semantics. Our effort is to solve the problem raised in LDL, COL and HiLog. The concept of "intersection-join" is extended through
the partial order defined on terms. Programmers can achieve equal set semantics or subset semantics on their own choices.

The intuitive idea of subset semantics is that if a fact \( p \) has a set component \( s \), we could infer \( p' \) by replacing \( s \) in \( p \) by its subset. In Example 1.4, we can infer

\[
['Univ. of Universal Tech', {[1987, \{john, rich, larry\}]}] \text{ as well as}
\]

\[
['Univ. of Universal Tech', {[1987, \{john, henry\}]}],
\]

since \( \{john, rich, larry\} \) and \( \{john, henry\} \) are subsets of \( \{john, mary, rich, larry, henry\} \).

At the same time, when we want to check the following statement against the database:

\[
['Univ. of Universal Tech', {[1987, \{john, mike, lee, henry\}]}],
\]

instead of giving a true or false, the answer can be

\[
['Univ. of Universal Tech', {[1987, \{john, henry\}]}].
\]

In summary, the most prominent characteristics of subset semantics are:

1. Fact inference: if there exists a fact in the database with a set component, then it is still a fact if we replace the set by its subset (or more generally, replace a term by a term below it in its semi-lattice).

2. Subset Join: If two joining values are sets, then the join is successful if the sets have a nonempty intersection\(^2\).

3. Unification: Two terms are unifiable if the corresponding nonset subterms are equal and set terms have nonempty intersection\(^3\).

This Chapter will address the first two parts.

**Example 9.1.3** [Continuation of Example 9.1.2] In Example 9.1.2, nothing will be derived simply because the joining attribute values (sets) are not equal. We

\(^2\)More generally, two values are joinable if their \( glb \) is not null. Detailed discussion in Section 3.

\(^3\)Two values of compatible types are unifiable if their \( glb \) is not null. Partial order on values will be discussed in Section 3.
argue that in those cases subset semantics can be used in place of equal set semantics. The query in Example 9.1.2 based on the rule and the database there will be successful and provide some information under subset semantics:

\[
\text{univcom('Univ. of Universal Tech', \{[1987, \{john, rich\}]\}, 'General Engine, CO.')}
\]

\[
\text{univcom('Univ. of Universal Tech', \{[1987, \{henry, mary\}]\}, 'Global Machine')}
\]

\[
\text{univcom('Manhattan College', \{[1988, \{bryan\}]\}, 'Global Machine')}
\]

The following is another example.

**Example 9.1.4** The hobby example from HiLog [CKW 89] makes the difference between equal set semantics and subset semantics clear. For example, the following fact says john has hobbies \{tennis, jogging, basketball\}:

\[
\text{john : ['John Smith', \{tennis, jogging, basketball\}].}
\]

When we post the following query:

\[
\text{john : ['John Smith', \{tennis, basketball\}]?}
\]

In the equal set semantics this query will fail since \{tennis, basketball\} is not the set of john's hobbies. But \{tennis, basketball\} is a subset of his hobbies. Furthermore if we ask the following query:

\[
\text{john : ['John Smith', \{tennis, football\}]?}
\]

It will fail simply because \{tennis, football\} is not even a subset of john's hobbies. But \{tennis, football\} does contain some of john's hobbies (tennis here).

In the subset semantics, the first query will return the answer

\[
\text{john : ['John Smith', \{tennis, basketball\}]}
\]

while the second query will return answer

\[
\text{john : ['John Smith', \{tennis\}]}
\]
The examples given above illustrate that subset semantics may be meaningful in some situations, however, it is not quite enough. There are situations where clearly subset semantics can not catch the intended meaning of the program. For instance, the intended meaning of Example 1.2 may be to find out those universities whose graduates in a year all went to one company which did not recruit from elsewhere. In this case, an empty answer is not only OK, but also necessary. In Example 1.4, the answer to query

\[
\text{john : ['John Smith', \{tennis, football\}]？}
\]

may not be the one intended. In our opinion, both subset semantics and equal set semantics are needed for program interpretations in different situations.

In this chapter, we will analyze the semantics of LLO programs. In particular, we will show that the concept of ground-facts (first proposed in IQL \[AbKa 89b\] and extended in LLO \[LoOz 91b\]) is more general than the ungrouping discussed in HILOG-R \[ChKa 91, ChenGard 88\]. We will define the subset semantics on complex objects directly without resorting to the ungrouping and grouping operation. Our work can be considered as a formalization of the semantics presented in HILOG-R and an extension to CCO \[BaKh 86\] in an object-oriented deductive framework. The semi-lattice structures on terms, atomic literals and interpretations are introduced in Section 2. Based on those semi-lattices, two terms are joinable if only if they share a glb. This is more general than the subset join in \[JaSc 82\]. More importantly, we prove that under subset semantics, \(LLO^+\) programs are monotonic and the intersection of two models of a \(LLO^+\) program is still a model. In Section 3, we first look into the semantic issues of basic \(LLO^+\) programs, that is, \(LLO\) programs without exposure operation and negation. Then we will try to extend the results to \(LLO^+\) (\(LLO\) without negation) in Section 4. Next, in Section 5, we show how to program the subset semantics under the equal set semantics framework based on the partial order defined on terms. Finally, in section 6, we present a stratification strategy for \(LLO\) programs in equal set semantics and give some applicable results.
9.2 Semi-Lattices on Terms, Atomic Literals and Interpretations

In this section, we first discuss the partial order among ground terms and then we extend the partial order to atomic literals and interpretations.

9.2.1 Partial Order on Terms

A set element $t_1$ in a set term $s$ is *redundant* if there is another set element $t_2$ in set term $s$ such that $t_1 \subseteq t_2$. A term is *reduced* if there is no redundant component terms. We can get the reduced form of a term by removing the redundant set element from a set term.

**Example 9.2.1** The set element $\{a,b\}$ in term $[c, \{\{a,b,d\}, \{a,b\}, p\}]$ is redundant since $\{a,b\} \subseteq \{a,b,d\}$. The reduced form of term is $[c, \{\{a,b,d\}\}, p]$.

In the following, we will assume all terms are in reduced form.

First we define a special operator $\sqcap$ (called greatest lower bound (glb) operator)\(^4\) which will be used in place of equal in subset semantics. In the following discussion, $\epsilon$ is a constant of *polymorphic* type, that is, this constant is the least value in every type. $\epsilon$ denotes the nonexistent null. We can see from the following discussion that $\sqcap$ defines a semi-lattice on values of each type and $\epsilon$ is at the bottom of each semi-lattice.

**Definition 9.2.2** Let $t_1$ and $t_2$ be terms of the same type.

1. For atomic constants or constant oids $t_1$ and $t_2$.
   
   $t_1 \sqcap t_2 = t_1$ if $t_1 = t_2$.
   
   $t_1 \sqcap t_2 = \epsilon$ if $t_1 \neq t_2$.

2. For function id-terms $f(t_1,t_2,\ldots,t_n)$ and $f'(t_1',t_2',\ldots,t_n')$.

   $f(t_1,t_2,\ldots,t_n) \sqcap f'(t_1',t_2',\ldots,t_n') = f(t_1,t_2,\ldots,t_n)$ if $f = f'$ and

\[^4\text{This operator is applicable to terms of the same type.}\]
$t_i = t'_i$ for $i = 1, 2, \ldots, n$.

$f(t_1, t_2, \ldots, t_n) \cap f'(t'_1, t'_2, \ldots, t'_n) = \varepsilon$ otherwise.

For any two id-terms, their glb is $\varepsilon$ unless they are identical. A function symbol with different arguments denotes different objects.

3. For tuple terms $[t_1, t_2, \ldots, t_n]$ and $[t'_1, t'_2, \ldots, t'_n]$,

$[t_1, t_2, \ldots, t_n] \cap [t'_1, t'_2, \ldots, t'_n] = [t_1 \cap t'_1, t_2 \cap t'_2, \ldots, t_n \cap t'_n]$ if for all $i = 1, \ldots, n, t_i \cap t'_i \neq \varepsilon$.

$[t_1, t_2, \ldots, t_n] \cap [t'_1, t'_2, \ldots, t'_n] = \varepsilon$ otherwise.

4. For set terms $t_1$ and $t_2$,

$t_1 \cap t_2 = \{t \cap t' | t \subseteq t_1 \& t' \subseteq t_2, t \cap t' \neq \varepsilon\}$.

5. For any term $t$, $t \cap \varepsilon = \varepsilon, \varepsilon \cap t = \varepsilon$.

Example 9.2.3 The following are examples of $\cap$ operation on terms ($f$ is a function symbol.).

$f(a, b) \cap f(a, b) = f(a, b)$

$f(a, \{b, c\}) \cap f(a, \{b\}) = \varepsilon$

$a \cap a = a, a \cap b = \varepsilon$

$[a, \{b_1, b_2\}, c] \cap [a, \{b_2, b_5\}, c] = [a, \{b_2\}, c]$

$\{a_1, a_4\}, \{b_2, b_{10}, b_{11}\} \cap \{a_1, b_{10}\}, \{b_2, b_{11}, b_{15}\} = \{a_1\}, \{b_{10}\}, \{b_2, b_{11}\}$

The operation of $\cap$ is commutative and associative (AC). We also use $\cap \{t_1, t_2, \ldots, t_n\}$ to denote $t_1 \cap t_2 \cap \ldots \cap t_n$.

The $\cap$ operation defines a partial order ($\preceq$) on terms. That is, $t_1 \preceq t_2$ iff $t_1 \cap t_2 = t_1$. It can be shown that the greatest lower bound of $t_1$ and $t_2$ is the term $t_1 \cap t_2$. $t_1 \preceq t_2$ iff $t_1 \preceq t_2$ and $t_1 \neq t_2$. It is straightforward to prove $t_1 \cap t_2 \preceq t_1$ and $t_1 \cap t_2 \preceq t_2$. 


9.2.2 Partial Order on Atomic Literals

The \( \sqcap \) operation can be extended to atomic literals: For object \( p \) and \( p' \) whose value is of type \( \{[\tau_1, \ldots, \tau_n]\} \),

- \( p(t_1, t_2, \ldots, t_n) \sqcap p'(t'_1, t'_2, \ldots, t'_n) = (p \sqcap p')(t_1 \sqcap t'_1, t_2 \sqcap t'_2, \ldots, t_n \sqcap t'_n) \) if \( p \sqcap p' \neq \epsilon \) and \( t_i \sqcap t'_i \neq \epsilon \) for \( i = 1, \ldots, n \).

- \( p(t_1, t_2, \ldots, t_n) \sqcap p'(t'_1, t'_2, \ldots, t'_n) = \epsilon \) otherwise.

For object \( p \) and \( p' \) whose value is of type \( \{\tau_1, \ldots, \tau_n\} \), the operation \( p[t_1, \ldots, t_n] \sqcap p'[t'_1, \ldots, t'_n] \) is defined in a similar way.

Example 9.2.4 The following examples show the results of the operation \( \sqcap \) when applied to atomic literals (\( p \) and \( q \) are constant oid's).

\[
p(a, b, c) \sqcap p(a, b, c) = p(a, b, c)
\]

\[
q(a, \{b_1, b_2, b_3\}) \sqcap q(a, \{b_2, b_3\}) = q(a, \{b_2\})
\]

\[
p(a, b, c) \sqcap q(a, b, c) = \epsilon. \quad p(a, b, c) \sqcap p(a, a, b) = \epsilon
\]

\[\square\]

In the subset semantics, we say an atomic literal \( A_1 \) is implied or subsumed by another atomic literal \( A_2 \), denoted as \( A_1 \leq A_2 \) if only if

\[A_1 \sqcap A_2 = A_1.\]

Example 9.2.5 For atomic literals \( A_1 \) and \( A_2 \), \( A_1 \leq A_2 \) means \( A_1 \) is either equal to \( A_2 \) or there is at least a term in \( A_1 \) which is below the corresponding term in \( A_2 \) in the semi-lattice, as shown below:

\[
f(a)(b, \{c_1, c_8\}) \leq f(a)(b, \{c_1, c_4, c_8, c_10\})
\]

\[
p(a, b, c) \leq p(a, b, c)
\]

\[\square\]

It is clear that \( \leq \) defines a partial order on atomic literals. All atomic literals of the same type forms a semi-lattice. We have the following result.
Lemma 9.2.6

\[ A \cap A = A, A_1 \cap A_2 \leq A_1 \text{ and } A_1 \cap A_2 \leq A_2 \]

\(\square\)

Proof. By induction.

Similarly to terms, we have \(A_1 \leq A_2 \) iff \(A_1 \leq A_2\) and \(A_1 \neq A_2\).

9.2.3 Partial Order on Interpretations

In the following, we extend the \(\cap\) operation to sets of ground-facts: a set of ground-facts is called an interpretation. Remember that a database instance corresponds to an interpretation.

We say a ground atomic literal \(A\) in an interpretation \(I\) is redundant if there exists an atomic literal \(B \in I\) such that \(A \leq B\), i.e., \(A \cap B = A\). An interpretation \(I\) is called reduced if there is no redundant atomic literal \(A\) in \(I\). From now on, we will assume that all interpretations are in reduced form.

Let \(I_1\) and \(I_2\) be two interpretations. Then

\[ I_1 \cap I_2 = \{L_1 \cap L_2 | L_1 \in I_1, L_2 \in I_2 \text{ and } L_1 \cap L_2 \neq \epsilon\} \]

Example 9.2.7 let

\[ M : \{p(a, b, c), q(a, \{b_1, b_2, b_3, b_4\}, c)\} \]

and

\[ N : \{p(a, b, c), q(a, \{b_1, b_5\}, c), w(c, d)\} \]

be two interpretations. Then we have

\[ M \cap N = \{p(a, b, c), q(a, \{b_1, b_3\}, c)\} \]

\(\square\)
Example 9.2.8 For models $\{q(1), q(2), p(\{1, 2\})\}$ and $\{q(2), q(3), p(\{2, 3\})\}$ in Example 9.1.1, we have
$$\{q(1), q(2), p(\{1, 2\})\} \cap \{q(2), q(3), p(\{2, 3\})\} = \{q(2), p(\{2\})\}$$
which is still a model of the program. \qed

For two interpretations $M$ and $N$, we say $N$ includes $M$, denoted as $M \subseteq N$ iff
$$M \cap N = M.$$

Example 9.2.9 For models
$$M = \{F(1), p(\{1\}), q(1), q(2)\}$$
and
$$N = \{F(1), F(2), p(\{1, 2\}), q(1), q(2)\},$$
where $F$ is an oid with set value, we have
$$M \subseteq N.$$

\qed

Properties of Interpretations

Obviously the relationship $\subseteq$ defines a partial order on interpretations. For interpretations $M$ and $N$, $M \cap N$ is a greatest lower bound (glb) of them.

Lemma 9.2.10

$$(M \cap N) \subseteq M \text{ and } (M \cap N) \subseteq N$$

The set of interpretations forms a semi-lattice also and we have the following result.

Theorem 5 For interpretations $M$ and $N$, $M \cap N$ is their greatest lower bound (glb), that is, if there is another interpretation $K$ such that $K \subseteq M$ and $K \subseteq N$ then $K \subseteq (M \cap N)$.
Proof. For each $A \in K$, there is a $B_1 \in M$ and $B_2 \in N$ such that

$$A \preceq B_1 \text{ and } A \preceq B_2.$$ 

Obviously we have

$$A \preceq B_1 \cap B_2,$$

and $B_1 \cap B_2 \in M \cap N$. By definition, we have

$$K \subseteq M \cap N.$$

9.3 Subset Semantics of Basic LLO+$^+$ Programs

A basic LLO+$^+$ program is an LLO program without exposure operation and negation. An LLO+$^+$ program is an LLO program without negation. A basic LLO program is an LLO program without exposure operation. The relationship between LLO and its sublanguages is shown in Figure 9.1.

In this section, we first review the equal set semantics for basic LLO+$^+$ programs. Then we will present the subset semantics based on the semi-lattice structures defined on terms, atomic literals and interpretations. In this section and Section 5, we will present some nice results on subset semantics. In Section 6 we will point out its problems and propose an approach that will integrate the subset semantics in an equal set semantics framework.
9.3.1 Equal Set Semantics

First we define equal set semantics for basic LLO+ programs. Let $I$ be an interpretation\(^5\). The notion of satisfaction under equal set semantics, denoted as $\models_\approx$, is defined as:

1. For a ground atom $A$, if $A$ is a comparison formula then $I \models_\approx A$ iff it is a tautology, otherwise, $I \models_\approx A$ iff $A \in I$.

2. For rule $A : -B_1, B_2, \ldots, B_n$, $I \models_\approx A : -B_1, B_2, \ldots, B_n$ iff $I \models_\approx B_i$ $(i = 1, \ldots, n)$ implies $I \models_\approx A$.

3. For a program $P$, $I \models_\approx P$ iff for each rule $r$ in $P$, $I \models_\approx r$.

We call interpretations that satisfy program $P$ models of $P$.

Equal set semantics works fine for Datalog and HiLog languages, if there is no set terms involved (there is set in HiLog, but they used a coding method to transform set terms into atomic literals with extensions, making two sets incomparable unless they are equal). As has been discussed in the introduction, when sets are involved in a program, equal set semantics is not enough, or not convenient for some applications. Subset semantics is simulated in HILOG-R [ChKaN91] using equal set semantics. The major idea is to make sets comparable if they share a common nonempty intersection. Our observation is that we need both equal set semantics and subset semantics. While the equal set semantics checks whether two terms are equal, the subset semantics checks if two complex terms share a greatest lower bound in the semi-lattice.

9.3.2 Subset Semantics Based on Semi-Lattices

The semi-lattice structure on terms, atomic literals and on interpretations lays a solid basis for complex object and model comparison without digging into

\(^5\)Remember that an interpretation is a set of ground-facts from an object-oriented database instance (cf. Section 2).
detailed structures of objects. That is, instead of processing the data piece by piece, as has been done in HILOG-R, we can operate an object as a whole.

Let $I$ be an interpretation. The notion of satisfaction under subset semantics, denoted as $\models_\subseteq$, is defined as:

1. For a ground atom $A$, if $A$ is a comparison formula then $I \models_\subseteq A$ iff it is a tautology, otherwise, $I \models_\subseteq A$ iff there exists an atom $B \in I$ such that $A \leq B$.

2. For rule $A : -B_1, B_2, \ldots, B_n$, $I \models_\subseteq A : -B_1, B_2, \ldots, B_n$ iff $I \models_\subseteq B_i$ $(i = 1, \ldots, n)$ implies $I \models_\subseteq A$.

3. For a program $P$, $I \models_\subseteq P$ iff for each rule $r$ in $P$, $I \models_\subseteq r$.

We call interpretations that satisfies program $P$ under subset semantics subset models of $P$.

Example 9.3.1 The following program have many subset models

\[
p(a, \{b_1, b_2, b_3\}, c)
\]
\[
q(\{b_2, b_7\}, d)
\]
\[
w(X, Y, Z) : -p(X, Y, V). q(Y, Z)
\]

For example,

\[
M = \{p(a, \{b_1, b_2, b_3, b_{10}, b_{100}\}, c), q(\{b_1, b_2, b_7, b_{10}\}, w(a, \{b_1, b_2, b_{10}\}, d)\}
\]

is a model. (Remember that all the interpretations are in reduced form. Facts like $w(a, \{b_1, b_{10}\}, d)$ is redundant and is removed from the model.)

\[
N = \{p(a, \{b_1, b_2, b_3\}, c), q(\{b_2, b_7\}, d), w(a, \{b_2\}, d)\}
\]

is also a model. Obviously we have

\[N \subseteq M.\]
Minimal Model

One of the results of subset semantics is that basic $LLO^+$ programs are monotonic. This is one of the advantages of subset semantics over the equal set semantics given to LDL [Bee+ 87] and COL [AbGr 87a, AbGr 87b]. More formally we have:

**Theorem 6** Basic $LLO^+$ programs are monotonic under subset semantics in the sense that for a model $N$ of program $P$, if $M$ is an interpretation and $N \subseteq M$, then $M$ is also a model of program $P$.

**Proof.** Since $N \subseteq M$, for every ground atomic literal $B$ in $N$, there is an atomic literal $B' \in M$ such that $B \subseteq B'$. If $N \models A$ for any formula $A$, we also have $M \models A$ by the definition of satisfaction. □

If set terms are involved in the comparison predicates, even though the monotonicity still holds, model $M$ may not be supported [ABW 87] by the program.\(^6\)

**Example 9.3.2** The basic $LLO^+$ program

$$p(X) : -q(X). X \subseteq \{a, b, c\}.$$  

where $X$ is of set type.

$$M = \{q(\{a, c\}), p(\{a, c\})\}$$

is obviously a model for the program. For interpretation

$$N = \{q(\{a, c, e\}), p(\{a, c, e\})\}.$$  

we have $M \subseteq N$. But $N$ is not a supported model, because $p(\{a, c, e\})$ is not supported. □

Let $M$ and $N$ be two subset models of program $P$. We say model $M$ is smaller than model $N$ if only if $M \subseteq N$.

\(^6\)A literal is supported if it is a fact or it is the head of a ground rule and every literal in the body is supported.
Theorem 7 Let $P$ be a basic $LLO^+$ program and $M$, $N$ are subset models of $P$. Then $M \cap N$ is also a subset model of $P$ smaller than $M$ and $N$.

Proof. For an atomic literal $A$, $M \models \beta A$ means there is a ground term $B \in M$ such that $A \leq B$. On the other hand, $N \models \beta A$ means there is a ground term $B' \in N$ such that $A \leq B'$. Consequently, $A$ is a lower bound of $B$ and $B'$. It has been shown in the last section that $B \cap B'$ is the $glb$ of $B$ and $B'$. That is, we have $A \leq B \cap B'$. Since $B \cap B' \in M \cap N$, $M \cap N \models \beta A$. ■

Corollary 9.3.3 Let $\mu = \{M_1, M_2, \ldots\}$ be a set of models for basic $LLO^+$ program $P$, then $\sqcap \{M_1, M_2, \ldots\}$ is also a model for program $P$ and for any model $M \in \mu$,

$$\sqcap \{M_1, M_2, \ldots\} \subseteq M.$$

Definition 9.3.4 A model $M$ of program $P$ is minimal if for any model $N$ of $P$, $N \subseteq M$ implies $N \cap M = M$. ■

Theorem 8 Every basic $LLO^+$ program has a minimal model.

Proof. Let $\{M_1, M_2, \ldots\}$ be all the models of a basic $LLO^+$ program. By Theorem 7 and its corollary,

$$\sqcap \{M_1, M_2, \ldots\}$$

is also a model and is a minimal one. ■

9.4 Subset Semantics of $LLO^+$

We have shown in the last section that basic $LLO^+$ programs are monotonic and the intersection of two models is still a model. In this section we extend the results to more general $LLO$ programs. We allow the exposure operator in $LLO$ programs but not negation.
The exposure operator together with function id-terms plays the role of data functions in COL. Unlike COL, where negation is redundant (see Section 1), \( LLO^+ \) is not equivalent to LLO under subset semantics. For the COL program simulating negation using data function, its \( LLO^+ \) translation is as follows:

\[
\begin{align*}
    f(t)(t) & : -p(t) \\
    a(t, \downarrow f(t)) & : - \\
    q(t) & : -a(t, \{\})
\end{align*}
\]

Under the subset semantics, \( q(t) \) is always true. It does not matter \( t \) is an element in the value assigned to \( f(t) \) or not. Since \( \downarrow f(t) \) is a set type, it is trivial that \( \{\} \subseteq \downarrow f(t) \) (or \( \{\} \leq f(t) \)). The reason is that \( a(t, \downarrow f(t)) \) is a fact and is true all the time. Negation in LLO is not redundant under subset semantics. As a matter of fact, we have similar results for \( LLO^+ \) programs like those of basic \( LLO^+ \).

**Theorem 9** \( LLO^+ \) programs are monotonic under subset semantics.

**Proof.** Let \( P \) be a \( LLO^+ \) program, \( M \) and \( N \) be two subset models of program \( P \) and \( M \subseteq N \). We show by induction that

\[
T_P(M) \subseteq T_P(N).
\]

For \( A \in T_P(M) \), there are two cases:

1. If \( A \in M \), then there exists \( B \in N \) such that \( A \leq B \) since \( M \subseteq N \).

2. If \( A \) is inferenced from a rule

\[
A : -B_1, \ldots, B_n,
\]

then by induction, \( B_1, \ldots, B_n \) are implied by \( T_P(N) \), so \( A \) is also in \( T_P(N) \).

Under subset semantics, some redundant results may be derived. For example, if an \( \downarrow \) operator is involved in a recursion, an id may be exposed before it is completely populated. Under subset semantics, this is allowed and partial results can be used in the deduction due to the partial order defined on the atomic
literals. In some cases, the results will be redundant because when the oid is completely populated, those results will be produced again. However, some results that can be derived by partial population will not be available in the equal set semantics.

More importantly, we have the following results:

**Theorem 10** The intersection of two models of a LLO+ program is still a model under subset semantics.

**Theorem 11** Every LLO+ program has a minimal model.

The most important feature of subset semantics is that we can inference \( A \) from \( B \) if \( A \subseteq B \), or \( A \cap B = A \). This feature allows us to achieve two goals: the monotonicity of basic LLO+ programs and subset join between objects. We have discussed the monotonicity. For the subset join, we have the following example:

**Example 9.4.1** Let \( p \) be an object with set value

\[
\{[a_1, \{b_1, b_2, b_3\}], [a_2, \{b_2, b_3\}]\}
\]

and \( q \) be an object with set value

\[
\{[a_1, \{b_1, b_3\}], [a_5, \{b_4, b_5\}]\},
\]

Then execution of the following program

\[r(X,Y) : -p(X,Y), q(X,Y)\]

under the subset semantics will assign value

\[
\{[a_1, \{b_1, b_3\}]\}
\]

to \( r \). The reason is that \([a_1, \{b_1, b_3\}]\) can be derived under subset semantics from both objects \( p \) and \( q \). Note that

\[
\{[a_1, \{b_1\}]\} \quad \text{and} \quad \{[a_1, \{b_3\}]\}
\]

are also results, but they are redundant and are eliminated. \(\square\)
9.5 Program Subset Semantics in the Equal Set Semantics Framework

In this section, we first point out some problems with subset semantics. Then we develop an approach to program subset semantics for some objects and operations based on the semi-lattice structures we developed in Section 3. Our point is that subset semantics can be achieved by programming under the equal set semantics framework.

9.5.1 Problems with Subset Semantics

While subset semantics has very good properties, there are situations in which subset semantics will not express the intended meaning of programs. For instance, in Example 9.4.1, we interpret the join as subset join, but the intended meaning may be equal join. For terms that do not involve set components, the subset semantics has no effects. However, if set terms are involved, we lost the ability to do equal join. (Remember that in [JaSc 82], join by equal and join by subset are expressed by two different operators.) For set components in literals, the EVEN and ODD operations do not make sense under subset semantics since subset could legitimately take place of a set.

In subset semantics, redundant sets and atomic literals are being removed to make interpretations into reduced form, because those redundant elements do not carry more meaning in the sense of subset semantics. However, in some situations, this redundancy is not only useful, but also necessary. This case is similar to the relationship between bags and sets in Prolog [StSh 86].

Another serious question about the subset semantics is the complexity of program evaluation. Even though the reduced interpretation is used and facts are checked at the time of its usage, exponential complexity may still be the case for some of the applications.
Based on the above analysis, we propose that the subset semantics be programmed on need. For some objects and operations, if subset semantics is required, we can program the application in such a way that those objects and operations be interpreted to have subset semantics, while all other objects and operations will be interpreted as usual. After special treatment for objects and operations that ask for subset semantics, the LLO program will be interpreted in the equal set semantics framework.

### 9.5.2 Subset Join: Join by Greatest-Lower-Bounds

In Example 9.1.2 in Section 1, we have shown that subset semantics gives more information than the equal set semantics. In this subsection we show that the subset semantics can be achieved in the equal set semantics framework based on the semi-lattices on terms established in Section 3.

For rule $r$:

\begin{equation}
  A : -B_1, B_2, \ldots, B_n,
\end{equation}

if a variable $X$ appears more than once in the body, we call $X$ joining variable. The first appearance of $X$ in $r$ is denoted by $X^1$, the second appearance $X^2$, $\ldots$, and the $k$th appearance $X^k$. Note that $X^i$ has the same type as $X$ and does not appear in any other place in $r$.

There are two cases depending on the intention of the join:

**Case 1.** If the join on variable $X$ is equal join, then do nothing.

**Case 2.** If the join on variable $X$ is subset join, then the following steps transform rule $r$ into rule $r'$ using semi-lattices on terms.

1. Replace the $i$th appearance of $X$ with $X^i$ in $r$ for $i = 1, \ldots, k$. So different appearances of the same variable can take different values.

2. Put a literal $\bigcap\{X^1, X^2, \ldots, X^k\} \neq \epsilon$ in the body of the rule $r$. This condition will guarantee that those appearances of the same variable have a greatest lower bound (glb) which is not a $\epsilon$ (means joinable).
If $X$ is not of set type and involves no set components in its type, this condition simply check that all those appearances take the same value.

3. If $X$ also appears in $A$, the head of the rule, then a literal $X = \cap\{X^1, X^2, \ldots, X^k\}$ is added to the body of the rule $r$. This is the joined value in the result.

The above process is repeated for each variable which appears in the body at least twice and for which subset semantics is desired.

Let the rule after transformation be denoted as $r'$:

(2) $A' : -B'_1, B'_2, \ldots, B'_n, \text{join\_literals},$

where $\text{join\_literals}$ are the literals handling $\text{join\_by\_glb}$. Each rule in program $P$ is transformed using the same procedure. The transformed program of $P$ is denoted as $P'$.

Example 9.5.1 For rule $r$:

$$w(X, Y, Z) : -p(X, Y, Y), q(Y, Z),$$

variable $Y$ is a joining variable. There are three appearances of $Y$ in the body and we want the join to be a subset join. According to the procedure given above:

1. New variables $Y^1, Y^2$ and $Y^3$ are introduced which will replace the three appearances of $Y$ in the body of the rule:

2. A $\text{join\_literal } \cap\{Y^1, Y^2, Y^3\} \neq \epsilon$ is added to the body of the rule:

3. Finally, for the appearance of $Y$ in the head of the rule, we add another literal $Y = \cap\{Y^1, Y^2, Y^3\}$ to the body.

The following is the transformed rule $r'$:

$$w(X, Y, Z) : -p(X, Y^1, Y^2), q(Y^3, Z), Y = \cap\{Y^1, Y^2, Y^3\}, \cap\{Y^1, Y^2, Y^3\} \neq \epsilon.$$
Note that if the condition $\cap\{Y^1, Y^2, Y^3\} = \epsilon$ is used as failure, it can be removed from the body without affecting the meaning of the program.

**Example 9.5.2 (Continuation of Example 9.1.2)** In Example 9.1.2, the rule

$$\text{unicom}(X_1, X_3, X_4) : -\text{Univ}(Y), Y[X_1, X_3], \text{Company}(Z), Z[X_4, X_3, X_5]$$

can be modified as follows in order to get the subset join on variable $X_3$ under the equal set semantics:

$$\text{unicom}(X_1, X_2, X_3, X_4) : -\text{Univ}(Y), Y[X_1, X_3], \text{Company}(Z), Z[X_4, X_3^2, X_5]$$

$$X_3 = X_3^1 \cap X_3^2.$$

Then $X_3$ will take the greatest lower bound of $X_3^1$ and $X_3^2$ in the term semilattice. Since the glb of two set terms is their intersection, we will get the result as shown in Example 9.1.3.

\[\square\]

**9.5.3 Inference by Subset**

The approach given in last subsection solved the problem of join by subset on those desired variables. In the following, we discuss a more general approach, *inference by subset* for some components of specific objects. Join by subset is a natural results of inference by subset.

In some situations, it is desirable to infer results from a fact with complex object components by simply replacing those components with terms which are below the complex object terms in the term semi-lattice. That is, if $p(t_1, t_2, \ldots, t_n)$ is a fact, we want to infer $p(t'_1, t'_2, \ldots, t'_n)$, with

$$(3) \quad p(t_1, t_2, \ldots, t_n) \cap p(t'_1, t'_2, \ldots, t'_n) = p(t'_1, t'_2, \ldots, t'_n).$$

Or, more simply,

$$p(t'_1, t'_2, \ldots, t'_n) \leq p(t_1, t_2, \ldots, t_n).$$

This is different from the equal set semantics, where for interpretation $I$ and literal $A$, $I \models A$ iff $A \in I$. As shown in Example 9.1.4, if John has hobby $\{t\}$. 


jogging, basket-ball}, we want the query does john has the hobbies \{tennis, basket-ball\} to be true. This is a fact implied by the database if we allow inference by subset. Remember that in HiLog and other languages, sets are not comparable unless they are equal. Under this circumstance, the above query is asking if john has another set of hobbies, which is not true. In the following, we discuss the role played by the term semi-lattice in intergrating subset semantics with the equal set semantics framework. Subset semantics can be achieved for some complex components while other components are still interpreted by equal set semantics.

To infer \(p(t_1, \ldots, t'_i, \ldots, t_n)\) from \(p(t_1, \ldots, t_i, \ldots, t_n)\), where \(t_i\) is the complex term and \(t'_i \subseteq t_i\), we inclue the following rule:

\[
p(t_1, \ldots, X, \ldots, t_n) : \neg p(t_1, \ldots, t_i, \ldots, t_n), X \cap t_i = X
\]

With this rule, a set term in a complex object will carry more meaning. That is, an atomic literal derived from a fact in the database by replacing its terms with terms below them in the semi-lattice is still a fact.

Example 9.5.3 (Continuation of Example 9.1.4) In Example 9.1.4 in Section 1. we talked about john's hobby. To get the expected answer, i.e., \{tennis, basket-ball\} is a set (subset) of john's hobby, we need the following rule:

\[X[\text{Name}, Y] : \neg X[\text{Name}, \text{Hobby_set}], Y \cap \text{Hobby_set} = Y,\]

where variable \(X\) is of type Person. \(\text{Name}\) is of type Name. \(Y\) and \(\text{Hobby_set}\) are of type \{hobby\}. Then when we ask the query:

\[\text{john['John Smith', \{tennis, basket-ball\}]?}\]

the answer will be yes, since \{tennis, basket-ball\} \n \{tennis, jogging, basket-ball\} = \{tennis, basket-ball\}. The partial order on sets is actually subset order in this situation.

On the other hand, since \{tennis, foot-ball\} is not a subset of john's hobby-set, the query

\[\text{john['John Smith', \{tennis, foot-ball\}]?}\]
will fail. The following rule can be used to pick out hobbies a person has from a given set of hobbies:

\[
pick\_hobby(P, S)[Y] : -P[Name, Hobbies], Y = S \cap Hobbies.
\]
where variable \( P \) is of type Person, \( S \) and \( Y \) is of type \{hobby\}. To find john’s hobby from the set \{tennis, foot\_ball\}, we have the following query:

\[
pick\_hobby(john, \{tennis, foot\_ball\})[Y]?
\]
which will give us the answer \{tennis\}. \( \Box \)

For the subset inference given above, many answers may be given. In some situation users will prefer the greatest term in a query.

For a query \( p(t_1, t_2, \ldots, t_n) \), rule (3) gives us subset semantics. We can get the greatest term for a component \( t_i \): a greatest term \( X \) is a term (a) satisfying the query; (b) there is no other term \( Y \) satisfying \( Y \cap X = X \) \((Y \neq X)\) and \( Y \) is also an answer to the query.

\[
q(t_1, \ldots, X, \ldots, t_n) : -p(t_1, \ldots, X, \ldots, t_n), \neg check(X, Y).
\]

\[
check(X, Y) : -p(t_1, \ldots, X, \ldots, t_n), p(t_1, \ldots, Y, \ldots, t_n), Y \neq X, Y \cap X = X
\]

Up to now, we only discussed the subset semantics of one component in an object. Actually a more general rule can be given which, for a given atomic literal \( p(t) \), will generate all atomic literals below it in the atomic literal semi-lattice.

\[
(4) \quad X : -p(Y), p(Y) \cap X = X.
\]

Predicates with tuple values can be handled in the same way. For example, if \( p \) is an object whose value is:

\[
\{[a_1, \{b_1, b_2\}],[a_2, \{b_2, b_6\}]\}.
\]

Rule (4) will extend the state of \( p \) (subset closure of \( p \) under semi-lattice):

\[
\{[a_1, \{b_1, b_2\}],[a_1, \{b_1\}],[a_1, \{b_3\}],[a_2, \{b_2, b_6\}],[a_2, \{b_2\}],[a_2, \{b_6\}]\}.
\]

Then all operations on object \( p \) will be given a subset semantics. All join operations defined on \( p \) will be also based on subset join.
9.6 Semantics of Stratified LLO Programs

As shown in the last section, basic LLO$^+$ programs are monotonic under subset semantics. In this section, we will discuss the semantics of LLO programs with negation and exposure operation under equal set semantics. Here we assume that those objects whose intended meaning is subset semantics have been handled properly using the approach discussed in this chapter.

It turns out that in equal set semantics, set-valued data functions [AbGr 87a], grouping [Bee+ 87] and exposure operation [LoOz 91b] also potentially induce non-monotonicity as shown by the following program in LLO (cf. [AbGr 87a]):

\[
\text{bored}(X) : \downarrow \text{hobby}(X) = \{\}
\]

\[
\text{hobby}(X)(tv) : \neg \text{bored}(X)
\]

If john has no known hobby, then \text{hobby}(john) = \{\} by the closed world assumption. So john is bored. Therefore, tv watching will be a hobby of john's (\text{hobby}(john)(tv) will be deduced from the program). The problem is very similar to the problem of recursion through negation. For the program given above, we can not determine if a person is bored unless we know exactly the set of hobbies of this person. But \text{hobby} and \text{bored} are recursively defined and it is hard to make a judgement.

In this section, we give a brief analysis of the fixpoint semantics similar to that given in LDL and COL. While the \downarrow operator brings the object-oriented database system and value-oriented system together, it also makes the semantics of the language complicated. Thus our discussion here will be focused on the \downarrow operation.

9.6.1 Stratification

In LLO, ground id-terms are playing different roles as predicate terms and values. The \downarrow operator, when applied to an ground id-term, will get the value
assigned to the object the id-term denotes. However, the id-term must be populated first. For an id-term \( f(X, Y) \), \( \downarrow f(X, Y) \) is equivalent to a data function in COL. The grouping (or nesting) operation in LDL can also be simulated by \( \downarrow \) operation and a function id-term. Due to this connection, the stratification of LLO program bears some similarity to that of COL [AbGr 87a] and LDL [Bee+ 87]. On the other hand, LLO without the \( \downarrow \) operation is typed HiLog. So LLO without the \( \downarrow \) has higher order syntax and first order semantics [CKW 89]. Unlike data functions in COL or grouping terms in LDL, where they must be computed prior to the predicate terms in which they appear, ground id-terms in literals are themself values. The oid's denoted by ground id-terms are not required to be populated at the time those literals are computed.

Let \( P \) be an LLO program with the following restrictions:

- Variables in the literal in the head of each rule must appear in the body or in the predicate term in the head.

Variables as predicate terms cause troubles for safety and preservation under extension, as has been pointed out by Ross [Ross 91]. In COL, variable can be declared as global to avoid some of the problems, such as relationship of variables in different rules. The best way is to use local stratification and perfect semantics [Przy 88] to analyze the meaning of programs. Ground rules will remove variables in question. Unfortunately, local stratification is not decidable [ABW 87]. Another approach is instead of stratifying variables, we stratify the class of the variable. Of course, this will be too coarse since we are binding all objects in a class together in priority of evaluation. We will look into this problem further and find a solution in between. In the rest of this chapter, we first discuss how variables in predicate terms can be handled; then we give a stratification strategy; finally we follow the approach given to COL by Abiteboul and Grumbach [AbGr 87a] and present semantics for stratified LLO programs.

We call predicate id-terms and id-terms in exposed terms in program \( P \) stratifiers (Remember an exposed term is a term with a \( \downarrow \) preceding it). Each stratifier has a carrier, defined below:
1. If the stratifier is a constant oid (including function id-term with variables in its arguments, then its carrier is itself.

2. If the stratifier is a variable, then its carrier is its class name if it belongs to a class; otherwise, its carrier is its type (the variable stands for a named value).

3. If the stratifier is a function id-term with at least one variable in its arguments, then its carrier is the function symbol. (if there is no variable in its argument, this function id-term can be treated as a constant).

We replace predicate id-terms and exposed id-terms in program $P$ with their carriers and call the resulting program abstract of program $P$, denoted as $AP$. In $AP$, all predicate terms are constants.

The carrier corresponding to the predicate id-term in the head of a rule is also called defined carrier of the rule. Each rule has only one defined carrier.

We say a carrier is a total determinant if one of the following conditions hold true:

1. The carrier has an exposure operation applied to it.

2. The carrier corresponds to the predicate id-term of a negated literal.

A carrier is called partial determinant if it is not a total determinant.

Consider, for instance, the following LLO rule:

$$f(X)(X, \downarrow g(Y)) : \neg X(Y, Z), \neg s(Y)$$

where the class $f(X)$ is $c_f$, the class of $g(Y)$ is $c_g$, the class of $X$ is $c_X$, and the class of $s$ is $c_s$. Then its abstract rule is:

$$c_f(X, \downarrow c_g(Y)) : \neg c_X(Y, Z), \neg s(Y).$$

$c_f$ is the defined carrier of the rule, $c_g$ and $s$ are total determinants and $c_X$ is a partial determinant (note the roles played by $X$ in $P$ on the left hand side and on the right hand side are different).

A term with an $\downarrow$ operator preceding it should be computed before the literals in which it appears and the defined id-term of the rule.
Let \( a \) be the defined carrier of a rule \( r \). A total determinant \( b \) in \( r \) is also total determinant of \( a \), denoted as \( a < b \). That is, \( b \) must be populated before \( a \). Otherwise, \( b \) is a partial determinant of \( a \), denoted as \( a \leq b \), meaning \( b \) must be populated no later than \( a \). Note that when we say a carrier should be populated, there are four cases:

1. If the carrier is a constant oid, then we should populate the object it denotes.

2. If the carrier is a class, it means populate all objects in the class.

3. If the carrier is a type, it means populate all named values with this type.

4. If the carrier is a function symbol (of specific signature), it means populate all possible oids that can be formed using this function.

Program \( P \) is stratified if

1. There is no sequence of determinant terms in the rules of \( AP \) of the form:
   \[ t_1 \Delta_1 t_2 \cdots \Delta_{k-1} t_k \]
   such that \( t_1 \) is the same as \( t_k \), \( \Delta_i \in \{<, \leq\} \) for \( i = 1, 2, \ldots, k - 1 \) and at least one \( \Delta_i \) is <.

2. There is no relationship \( t_1 < t_2 \) or \( t_1 > t_2 \) such that \( t_1 \) is an oid in class \( t_2 \), or \( t_1 \) is a named value of type \( t_2 \).

We have the following proposition (cf. [ABW 87]).

**Proposition 9.6.1** Let \( P \) be a program, \( AP \) be its abstract program and \( O \) be the set of determinant terms appearing in \( AP \). Then \( P \) is stratified iff there is a partition \( O_1, O_2, \ldots, O_l \) of \( O \) such that

\[
\begin{align*}
    t_1 &\leq t_2, t_1 \in O_i \Rightarrow \exists j (i \leq j, t_2 \in O_j), \\
    t_1 &< t_2, t_1 \in O_i \Rightarrow \exists j (i < j, t_2 \in O_j)
\end{align*}
\]
For each partition \( O_1, O_2, \ldots, O_l \), program \( P \) is partitioned into \( P_1, \ldots, P_l \) [ABW 87]. Each \( P_i \) (\( 1 \leq i \leq l \)) is called a stratum or layer. If \( l = 1 \) then \( P \) is a monostratum program. Otherwise, it is a multistrata program.

The program in Example 6.1.6 is stratifiable, even though id-term \( g(R, X) \) and \( adj(R, X) \) are mutually recursive. The ground-term \( g(R, X) \) and \( adj(R, X) \) denote oid's when they appear inside literals. The interesting part is the intension of the object. Method \( g(R, X) \) simply groups together all those oid's. This benefit comes from the separation between intension and extension part of an object.

**Example 9.6.2** Consider rule from [AbGr 87a]:

\[
 f(Y)(X) : -s(Y, f(X)).
\]

This rule as a program in COL is not stratified, since \( f(X) \) is a data function and \( f < f \). But the same program in LLO is stratified. Unlike the data function in COL, the id-term \( f(X) \) in the body, if instantiated, denotes an object identifier.

And we have \( f \leq s \), which is the only relationship between determinants.

Let \( s \) be assigned the value

\[
 s = \{[1, f(2)], [2, f(3)], [1, f(3)]\}.
\]

The result of the program will be: \( f(1) : \{2, 3\}, \ f(2) : \{3\}. \)

\[\Box\]

### 9.6.2 Procedural Semantics of Stratified Programs

For a monostratum LLO program, its fixpoint semantics can be defined similar to those defined in COL and LDL. The fixpoint operator is applied to the extensional input iteratively until no more facts can be derived. For a multistrata program, the fixpoint operator is applied stratum by stratum, the result of a lower stratum is the extensional input to the next stratum.

Let \( P \) be a monostratum program, \( r \) be a rule

\[
 B : -B_1, \ldots, B_m.
\]

Let \( M \) be the set of ground-facts of interpretation \( I \). and \( \theta \) be a binding of variables to ground terms in \( P \) or \( M \). Then a ground literal \( B\theta \) is the result of
applying \( r \) to \( M \) with binding \( \theta \) if

\( B \theta \) and \( B_i \theta \) are ground-facts \((1 \leq i \leq m)\).

\( B_i \theta \in M \).

The operator \( T_P \) to \( M \) is defined by:

\[
T_P(M) = \{ A \mid A \text{ is the result of applying a rule } r \text{ in } P \text{ to } M \text{ with binding } \theta \}
\]

The fixpoint semantics of program \( P \) [ABW 87] is defined as:

\[
T_P \uparrow 0(M) = M
\]

\[
T_P \uparrow (i + 1)(M) = T_P(T_P \uparrow (i)(M)) \cup T_P \uparrow (i)(M)
\]

\[
T_P \uparrow \omega(M) = \cup_{i=0}^{\infty} T_P \uparrow (i)(M).
\]

It is well known that: an operator \( T \) is monotonic if \( I \subseteq J \) implies that \( T(I) \subseteq T(J) \); \( I \) is a fixpoint of \( T \) if \( T(I) = I \); and \( I \) is a pre-fixpoint of \( T \) if \( T(I) \subseteq I \). Further, a model \( M \) of \( P \) is justified or supported if for each \( A \in \text{ground-facts}(M) \), there exist a rule \( r \) in \( P \), and a binding of variables \( \theta_M \) such that \( A \) is the result of applying \( r \) with \( \theta_M \).

The following result is from [AbGr 87a].

**Proposition 9.6.3** Let \( P \) be a program, and \( M \) be the set of ground-facts of an interpretation \( I \). Then the next two statements are equivalent: (i) \( M \) is a (minimal) justified model of \( P \); (ii) \( M \) is a (minimal) fixpoint of \( T_P \).

A program without \( \downarrow \) operator is called simple program. As indicated above, simple LLO programs are HiLog programs with typing. From LDL [Bee+ 87] and HiLog [CKW 89], we have the following result:

**Lemma 9.6.4** Let \( P \) be a simple stratified LLO program and \( M \) be a set of ground-facts. \( T_P \uparrow \omega(M) \) is a minimum model of \( P \) w.r.t. \( M \).

\[\square\]

Since an id-term with a \( \downarrow \) operator is equivalent to a data function in COL, we have the following results from [AbGr 87a].

**Theorem 12** Let \( P \) be a monostratum program. Then for each set of ground-facts \( M \).
1. $T_P \uparrow \omega(M)$ is a minimal prefixpoint of $T_P$ containing $M$.

2. $T_P \uparrow \omega(\emptyset)$ is a minimal fixpoint of $T_P$.

For an interpretation $I$, we are interested in a special subset $O_I$ of the ground id-terms in $I$. The subset of $\text{ground-facts}(I)$ restricted to $O_I$ is denoted by:

$$\text{ground-facts}(I \mid_{O_I}) = \{ o(v) \mid o \in O_I, v \in \delta(o) \text{ or } v = \delta(o) \}.$$ 

To prove theorem 1, we need to prove that monostratum programs are growing, $O$-finitary and stable on $O$ for some set of object identifiers $O$.

**Definition 9.6.5** Let $P$ be a program. Then:

1. $T_P$ is growing [ABW 87] if for each interpretation $I$, $J$ and $M$,

$$\text{ground-facts}(I) \subseteq \text{ground-facts}(J) \subseteq \text{ground-facts}(M) \subseteq T_P \uparrow \omega(\text{ground-facts}(I)) \Rightarrow T_P(\text{ground-facts}(J)) \subseteq T_P(\text{ground-facts}(M)).$$

2. $T_P$ is $O$-finitary [AbGr 87a] if for each sequence $(I_n)$ of interpretations such that for each $n(n \geq 0)$, $I_n \subseteq I_{n+1}$, and $I_n\mid_O = I_0\mid_O$, then

$$T_P(\bigcup_{n=0}^\infty \text{ground-facts}(I_n)) \subseteq \bigcup_{n=0}^\infty T_P(\text{ground-facts}(I_n)).$$

3. $T_P$ is stable on $O$ if for each $I$, $(T_P(\text{ground-facts}(I)))\mid_O \subseteq \text{ground-facts}(I)\mid_O$. 

The following result is from COL.

By using the same techniques as in COL [AbGr 87a], for a multistrata program $P$ with partition $P_1, P_2, \ldots, P_l$, we have the following structure:

- $K_0 = \emptyset$, and

- $K_i = T_{P_i} \uparrow \omega(K_{i-1})$ for $i = 1, 2, \ldots, l$.

$K_m$ does not depend on stratification [ABW 87]. A multistrata program is executed stratum by stratum, starting from the lowest in the stratification structure. The result of lower stratum is the input to the next stratum.

**Theorem 13** Let $P$ be a stratified program. $K_i$ is a minimal fixpoint of $\bigcup_{i=1}^l T_{P_i}$. Thus $K_i$ is a minimal model of $P$. 

Chapter 10

Conclusions

In this thesis, we established a data model which we believe will provide proper support for deductive object-oriented data languages. Then we presented a typed language LLO, integrated with object-oriented features, such as object identity, methods and method inheritance. Most importantly, we focused on extending current Horn clause languages so that the accumulated techniques could be inherited.

Meta variables are introduced as a mean of data abstraction and information hiding. Instead of building inheritance hierarchies by attribute set of types, as in [Card 88], inheritance hierarchies are set up by instantiation of meta variables. This approach solves some problems raised in [Card 88], such as parameter polymorphism and extra attributes in subtypes. The concept of named values play the role of bridging value-oriented data models and object-oriented data models together. A relation can be a named set value and nothing is encapsulated, or assigned to a class for method sharing. In the later case, that relation is also encapsulated. One of our contribution is that we established an object-oriented data model which is an extension of relational data models, including first normal form and non first normal form. Our data model also reflects the ideas in semantic data models, such as IFO [AbHu 87] and SDM [HaMcL 81].

In LLC, rules are methods. Generic methods can be defined using function id-terms. Variables in function id-terms provide vehicles to pass information from the head of a rule to its body. With the support of the data model we developed.
method inheritance is made possible. The unification algorithm dispatches a message to a proper method using the signature information of the method and the message. LLO without \( \downarrow \) operation is typed HiLog and has first order semantics. With \( \downarrow \) operation, we can get the value or state denoted by an oid. This operation provides a channel between value-oriented and object-oriented data. The semantics of the language is no more complex than LDL or COL. However, in LLO, since the unification will also handle types with meta variables in addition to set types, one of the drawbacks is efficiency.

More research is needed on problems such as variables as predicate terms and variables as arguments of id-terms. This problem has been addressed by Ross [Ross 91]. He extended the concepts of range-restrictiveness, preservation under extensions and safety for HiLog programs. Some results restricting the use of variables are presented there. But more efforts are needed for LLO to be a tractable language. As we said, LLO is an extension of HiLog. There are problems in HiLog that are not well understood.

Another important topic is inference on meta information such as types of objects and class-subclass relationships. This is accomplished in F-Logic by overloading concepts. Even though we have given some examples in Chapter 8, more research needs to be done to evaluate the impact on object-oriented deduction, such as encapsulation and abstraction.

There are other issues such as program evaluation and optimizations. More work in this direction is necessary to make LLO a practical deductive language with object-oriented features.

Logic languages have formalized theory and semantics. Encouraged by the success of Prolog and relational data models, many applications are developed in deductive paradigm. On the other hand, object-oriented concepts are gaining popularity both in industry and university. Most importantly, many applications can not be properly handled in value-oriented paradigm. Object-oriented programming is the result of research in software engineering and data modeling. Many features, such as encapsulation, information hiding and method reuse,
have been proven to be good programming practice for years. It is our belief that integrating object-oriented features with deductive languages will result in programming languages that are declarative, user friendly and expressive.
Bibliography


[LoOz 90a] Lou, Y., and M. Ozsoyoglu. Types, Objects and Inheritance in an Object-Oriented Data Model, TR-90-10, Department of Computer Engineering and Science, Case Western Reserve University, March 1990. Accepted by European Conference on System Design and Automation (ESDA'91).


