THE UNIVERSITY OF CALGARY

S – Logic: A Higher-Order Logic For Deductive Databases

BY

Mengchi Liu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST, 1990

ISBN 0-315-61976-7

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled *"S-Logic: A Higher-Order Logic for Deductive Databases"* submitted by Mengchi Liu in partial fulfillment of the requirements for the degree of Master of Science.

John G Cleary
Supervisor, Dr. John Cleary
Department of Computer Science

[illegible]
Dr. Brian Gaines
Department of Computer Science

Ken Loose
Dr. Ken Loose
Department of Computer Science

Anthony Kusalik
Dr. Anthony Kusalik
Department of Computational Science
University of Saskatchewan

Date August 27, 1990

Abstract

In this thesis, I discuss the significant problems inherent in deductive databases which result from the integration of relational database and logic programming techniques. I examine two broad areas where problems are apparent: complex object modeling and higher-order features. By complex object modeling, I mean the ability to naturally represent object identity, data abstractions, and inheritance. By higher-order features, I mean the ability to uniformly represent schema and sets.

A summary is given of attempts, in both the database and logic programming fields, to solve these problems separately. Among them are semantic data models which use data abstractions and inheritance, and extended logic terms which can represent the existence and internal structure of complex objects.

Besides addressing these problems, this thesis also tries to solve them. It proposes a new higher-order logic language called S-logic which results from the integration of the semantic data model and extended logic term approaches. It shows that S-logic naturally models complex objects, and represents desired higher-order features and sets. It also shows that an S-logic program can be transformed into Prolog so that S-logic is implementable in practice.

The major original contributions of the research presented here are twofold. First, a language, S-logic, is defined which has expressive syntax. Second, a well-defined least fixpoint semantics is given for definite S-logic programs.

Acknowledgements

I would like to thank my supervisor, Dr. John Cleary, for his valuable advice and support throughout this research project. His insight and knowledge were fundamental to the success of this thesis. I have been most impressed by his quick understanding of tricky points and his tiny but forcible counter-examples.

Thanks are also due to my friend Vinit Kaushik for the many discussions and for his patience in correcting my English mistakes.

I am also grateful to the Department of Computer Science, The University of Calgary, for financial support.

Love and appreciation to my wife Hongbo Liang for her tremendous support and sacrifice enabling me to dedicate myself to the completion of this thesis.

Finally, I would like to thank my parents whose encouragement began long before my work on this thesis.

Contents

List of Figures

Chapter 1

Introduction

Databases and logic programming are two independently developed areas in computer science. Database technology has evolved in order to efficiently organize, manage and maintain large amounts of data. The relatively slow speed of secondary devices holding the data is one of the main limitations of database systems in the past. Hence, the internal organization of databases has been the primary focus of research in the past. Besides, the need to share information among a variety of users requires strict rules governing the manipulation of data to be imposed to preserve the integrity of the database and to guarantee privacy for each user. This led to the development of several basic models.

A data model is a collection of well-defined concepts that helps the database users to understand and express the static and dynamic properties of applications. It determines the types of data structures visible to the user and the operations allowed on these structures. It also provides the conceptual basis for thinking about the applications and provides a formal basis in developing and using the database systems. A typical data model consists of two parts: a set of generating rules for constructing structural properties and a set of operations for expressing behavioral properties of applications [6, 7, 43]. A database schema consists of the definition of structural properties of all application object types based on the concepts provided by the corresponding data model. Corresponding to the schema is a data repository called a database which is an instance of the database schema. The process of capturing the information requirements of applications and producing the corresponding database schema is called data modeling. Obviously, the complexity of the data modeling depends on the data model used.

Conventional data models, which include relational, hierarchical and network models, are machine-oriented. They arrange data in fixed linear sequences of field values and thus provide an efficient basis for storing and processing data. Each of the conventional models is based on some idealized data structure and has a set of operations associated with this structure. Of them, the relational data model is the most significant and widely used. The main attraction of the relational model is its mathematical clarity, which facilitates non-procedural, high-level queries and thus separates the user from the internal organization of data. In the last two decades, a great deal of thought and ingenuity has been invested in the efficient processing of queries and updates of relational databases in secondary memory.

Logic programming began in the early 1970's as a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. Logic programming is based on mathematical logic, which is formalized in terms of proof theory and model theory. Proof theory provides formal specifications for correct reasoning with premises, while model theory prescribes how general assertions may be interpreted with respect to a collection of specific facts. Logic programming is programming by description. It uses logic to represent knowledge and uses deduction to solve problems by deriving logical consequences. Most logic programming is based on the Horn-clause form, which is a variant of first-order logic. Horn-clause logic has well defined model-theoretic and procedural semantics [29]. The state of the art in logic programming is represented by Prolog in its various manifestations.

Relational database and logic programming techniques have been found to be strongly similar in their representation of data. In the last few years, a lot of effort has been made toward the application of logic to relational databases. They are also found to be complementary. The combination of logic programming and relational database techniques has led to the active research area of deductive databases. It combines the benefits of these two approaches, such as representational and operational uniformity, deductive power, efficient secondary storage access, etc.

Unfortunately, significant problems remain inherent in this synthesis. There are several broad areas where problems are apparent. The first problem area is that deductive databases based on relational databases and Prolog cannot naturally model complex objects, which include object identity, data abstractions and inheritance. The second problem area is that it cannot naturally deal with higher-order features which include schema and sets. Traditionally, a separate language is provided to specify and manipulate the schema information, and sets are not directly supported at all. These problems result from the underlying relational data models and pure Prolog which uses inexpressive flat structures.

To improve the expressiveness of the relational data model, semantic data models have been proposed which use data abstractions and inheritance. So do extended terms with internal structure in logic programming. But, none of them can independently solve the above problems of deductive databases.

This thesis proposes a higher-order logic language for deductive databases called S-logic which is based on the semantic data model and extended term approaches. S-logic can naturally support object identity, semantic data abstractions and inheritance, and allows the definition and manipulation of database schema and data in an integrated framework.

1.1 Organization of Thesis

In order to make this thesis as self-contained as possible, in Chapter 2, I first introduce the relational database and logic programming techniques. Then I introduce the deductive database technique based on these two approaches.

Chapter 3 analyzes the problems of the deductive databases. Since the problems result from the underlying relational data model and pure Prolog, I then discuss some solutions to these problems in these two areas. This results in the motivation of S-logic.

The core of the thesis lies in Chapter 4 and Chapter 5 where a higher-order logic language is introduced first informally and then formally.

Chapter 6 presents a transformation algorithm which converts S-logic to Prolog.

Finally, I summarize and discuss further research directions in Chapter 7.

Chapter 2

Background

2.1 Relational Databases

Here, a briefly introduction is given to the basic concepts and definitions which underlie the relational data model.

A *domain* is a usually finite set of values. The *Cartesian Product* of domains $D_1, D_2, ..., D_n$ (not necessarily distinct) is denoted by $D_1 \times ... \times D_n$ and is the set of all tuples $(x_1, ..., x_n)$ such that $x_i \in D_i, i = 1, ..., n$. A *relation* is a subset of the Cartesian product of one or more domains. The *arity* of a relation $R \subseteq D_1 \times ... \times D_n$ is n. The number of tuples in R is called its *cardinality*. A relation is *finite* if its cardinality is finite. A *database* is a finite set of finite relations.

It is customary (though not essential) when discussing relations to represent a relation as a table in which each row represents a tuple. Examples of this representation are shown in Figure 2.1, which illustrates a relation describing employees. In the tabular representation of a relation, the following properties, which derive from the definition of a relation, should be observed:

1) no two rows are identical;

2) row order is insignificant;

3) column order is significant; and

4) every entry is an atomic value.

In the tabular representation of a relation, it is customary to name the table and to name each column, as shown in Figure 2.1. The table is named by a *relation name.* Each column of the table is called an *attribute* and has an *attribute name.* It should be noted that different attributes can draw values from the same domain in a relation.

person

name	age	address
Bob	40	257 9th Av SW
Henry	50	128 2nd St NW
John	62	439 5th Av NE
Jenny	24	725 6th Av SW
Smith	30	283 4th St SE

speaks

name	language
Bob	English
Henry	Chinese
Henry	Spanish
John	English
Smith	Franch

Figure 2.1: Examples of Relations.

A column (or set of columns) whose values uniquely identify rows of a relation is called a *candidate key* of the relation. It is possible for a relation to have more than one candidate key. In this case, it is customary to designate one as the *primary key.*

Often a column or set of columns in one relation will correspond to a key of another relation so that different relations can be related. It is called a *foreign key.* A foreign key need not be (and often is not) a key of its own relation.

The structure of a relation is represented by its *relation schema* which consists of a relation name and all its attribute names, whereas the specific relation is said to be an *instance* of the relation schema.

Not all instances of a relation schema have meaningful interpretations; that is, they do not correspond to valid sets of data according to the intended semantics of the database. One therefore introduces a set of constraints, referred to as *integrity constraints*, associated with a relation schema to ensure that the database meets the intended semantics. There are two major kinds of integrity constraints: type constraints, which require the arguments of relations to be belong to specified domains, or dependency constraints, which express structural properties of relations.

To summarize, a database schema consists of a collection of relation schemes together with a set of integrity constraints. A database instance, also called a database state, is a collection of relation instances, one for each relation schema in the database schema. A database state is said to be valid if all relation instances that it contains obey the integrity constraints.

Over the relational data model, there are three major relational languages to manipulate data in a relational database: relational algebra, domain calculus, and tuple calculus. All of them are equivalent [45]. Among them, relational algebra has strong origin in mathematics. The relational algebra is a collection of operators that deal with whole relations, yielding new relations as a result. The major operators of relational algebra include the following:

- *Projection*: Given a relation R and A a set of attribute names of R, the projection operation represented by $\pi_A(R)$ returns only the specified columns of the given relation, and eliminates duplicates from the results.

- *selection*: Given a relation R and P a collection of conditions over the relation, the selection operation represented by $\sigma_P(R)$ selects only those tuples of a relation which satisfy the given conditions.

- *(Natural) Join*: Given two relations R and S, the natural join operation represented by $R*S$ is formed by computing the Cartesian product, $R \times S$, selecting out all tuples whose values on each attribute common to R and S coincide, and projecting one occurrence of each of the common attributes.

Of these three major operators, join is most frequently used to draw relationships between different relations via common attributes. However the join operator is also quite time consuming. A lot of efforts have been made to improve its performance.

2.2 Logic Programming

Mathematical logic is the study of the relationship between beliefs and conclusions. For example, if we believe that *Art* is the father of *Bob* and that fathers are parents, then we can conclude that *Art* is the parent of *Bob*. The first two sentences *logically imply* the conclusion. In logic programming the programmer encodes in a logic program a set of beliefs about the application area by using *clauses*, and the machine applies *rules of inference* to known beliefs and derives conclusions that are logically implied by those beliefs. Subsequent applications allow a program to derive further conclusions. And so forth.

Most logic programming is based on Horn-clause form, which is a variant of first-order logic. A Horn-clause logic program [29] is a set of clauses of the form $p_0 \leftarrow p_1, ..., p_n$. Each $p_i, 1 \leq i \leq n$ is called either a *positive literal* (*atom*) if it has the form $p(t_1, ..., t_m)$ or *negative literal* if it has the form $\neg p(t_1, ..., t_m)$, where p is an m-place *predicate* symbol and $t_1, ..., t_m$ are *terms*. A term can be either a *constant*, a *variable*, or a *function* which takes terms as its arguments. Normally, constants, functions and predicates are represented by a lower-case letter, while variables by upper-case letters or the underscore symbol. The positive literal p_0 is called the *head* or *conclusion*, and p_1 through p_n form the *body* or *conditions* of the clause. A term which contains no variables is called a *ground term*. A clause (a literal) in which no variables appear is called a *ground clause* (*ground literal*). In terms of operational semantics, the meaning of a clause may be paraphrased as follows: in order to prove that p_0 is true, it is sufficient to prove that p_1 through p_n are true. A clause with an empty set of conditions is always true; it is called a *fact*. A clause with non-empty head and conditions is called a *rule*. A clause with an empty head, on the other hand, is called a *goal* which the system tries to prove.

Two complementary aspects of clauses are of interest. One deals with *semantics* or *model theory*, the specification of truth values to clauses, whereas the other deals with *syntax* or *proof theory*, the derivation of a clause from a given set of clauses.

2.2.1 Semantics: Interpretation and Model

The declarative semantics of a logic program is given by the usual model-theoretic semantics of formulas in first-order logic.

In semantics we are concerned with interpretation, where an *interpretation* of a set of clauses consists of the specification of a nonempty set (or domain) D, over which the variables range. Every constant is assigned to an element of D. Each n-ary function symbol is assigned a mapping from D^n to D. And every n-place predicate is assigned an n-ary relation on D^n. An interpretation thus specifies a meaning for each symbol in the formula. A *variable assignment* assigns each variable an element in the domain. Given an interpretation I with domain D, and a variable assignment V, the *truth value*, true or false, of a clause can be obtained as follows. If R is the relation assigned to an n-place predicate symbol p, then the positive literal $p(t_1, ..., t_m)$ evaluates to true if $< t_1, ..., t_m > \in R$; otherwise it evaluates to false. A negative literal $\neg p$ evaluates to true if p is false; otherwise it evaluates to false. p_1, p_2

evaluates to true if both p_1 and p_2 are true; otherwise it evaluates to false. $p_1 \leftarrow p_2$ evaluates to true if either p_1 is false or p_2 is true; otherwise it evaluates to false.

A *model* of a program is an interpretation in which all clauses are true. A clause G is said to be a *logical consequence* of a program P iff G is true in all models of P. This is denoted by $P \models G$. However, it is impossible to prove that G is true in all models of P. The question can be changed to another one and we get a useful result which states that G is a logical consequence of P iff $P \cup \{\neg G\}$ is *unsatisfiable*, that is, if no interpretation is a model.

It turns out that there is a small and convenient class of interpretations, which can show us unsatisfiability. These are the *Herbrand interpretations.* Given a set of clauses P, the domain U of a Herbrand interpretation is the *Herbrand universe*, which is the set of all ground terms in P. A Herbrand interpretation is any interpretation based on the Herbrand universe, in which each constant is assigned itself in U, while every n-ary function symbol is assigned a mapping U^n to U denoted by itself. A *Herbrand model* of P is a Herbrand interpretation which is a model for P. A very useful theoretical result is that in order to prove unsatisfiability of a set of clauses, it suffices to consider only Herbrand interpretations.

Of the Herbrand models, we are most interested in the exact one called the *least Herbrand model* (or *minimal model*) which is the intersection of all Herbrand models. It has a very important property: every atom in the least Herbrand model is a logical consequence of the set of clauses.

Let P be a program, H is the set of all Herbrand interpretations of P which forms a complete lattice under the partial order of set inclusion $\subseteq$, we define a mapping $T : H \rightarrow H$ as follows. Let I be a Herbrand interpretation. Then $T(I) = \{p \in H :$

$p \leftarrow p_1, ..., p_n$ is a ground instance of a clause in P and $\{p_1, ..., p_n\} \subseteq I\}$. If I is a model of P, then we have $T(I) \subseteq I$. Since T is defined over a complete lattice and it is monotonic, it has a least fixpoint $lfp(T)$. a is a least fixpoint of T if a is a fixpoint (that is, $T(a) = a$) and for all fixpoints b of T, we have $a \subseteq b$. An interesting result is that the least Herbrand model is equal to $lfp(T)$.

2.2.2 Proof Theory

The first-order predicate calculus is a formal system that can create, or deduce new clauses, which are logical consequence of a given set of clauses by using the rules of inference. Of the rules that have been invented, resolution is the most extensively studied and used in logic programming.

Two literals are said to be *unifiable* if they can be made identical by some *substitution* to the variables. For example, literals $p(X, bob)$ and $p(art, Y)$ are unifiable with the substitution $\{X/art, Y/bob\}$ which is to be read: substitute art for X, bob for Y. Literals $p(X, bob)$ and $p(Y, art)$, however, are not unifiable.

$$\begin{array}{l}(1) parent(X, bob) \leftarrow father(X, bob).\\ (2) grandparent(art, Z) \leftarrow parent(art, Y), parent(Y, Z).\\ (3) grandparent(art, Z) \leftarrow father(art, bob), parent(bob, Z).\end{array}$$

Figure 2.2: Example of Resolution

Given two clauses with unifiable literals on different sides of two clauses, the resolution rule can be used to create, or deduce a new clause in which the left- and right-side are the unions of the left- and right-hand sides of the two original clauses, with the unified expressions deleted and the unifying substitution applied to the remaining expressions. An example of resolution is given in Figure 2.2, where a new clause (3) is deduced from the clauses (1) and (2) using resolution which unifies $parent(X, bob)$ and $parent(art, Y)$.

Resolution is used mostly to carry out refutation proofs: Given a program P and a goal G, in order to prove G is deducible from P, written as $P \vdash G$, we can try to show that P and $\neg G$ are not simultaneously satisfiable. If we can derive the empty clause, that is, a clause with no conditions and no conclusions, then P and G cannot simultaneously be satisfiable, thus we have proved the goal $P \vdash G$. When a goal contains variables and the empty clause can be derived, then we have proved the goal, and furthermore found desired answers from the substitution.

The most important two results we can get here is that if P and $\neg G$ have a refutation, i.e. $P \vdash G$, then $G \in lfp(T)$, where $lfp(T)$ is the fixpoint of the mapping T described earlier, which implies that G is a logical consequence of P, that is, $P \models G$. This means that the resolution refutation is *sound* in that any conclusion it draws is guaranteed to be correct so long as its premises are correct. On the other hand, if $P \models G$, then $G \in lfp(T)$, which implies that $P \cup \{\neg G\}$ has a refutation and hence $P \vdash G$. This means that resolution refutation is also *complete* in that it can derive any logical implication from a given set of premises. We note here that the mapping T provides a link between the model theory and proof theory of a set of clauses. The semantics of Horn-clause logic can thus be described declaratively as well as operationally.

Logic programming is programming by description [17]. In traditional programming, one builds a program by specifying the operations to be performed in solving a problem, that is, by saying how the problem is to be solved. In logic programming, however, a program is constructed by describing its application area, that is, by saying what is true in terms of clauses which has the requisite declarative semantics. The system will use the rules of inference to choose specific operations to draw conclusions about the application area and to answer questions even though these answers are not explicitly recorded in the description.

The most often used logic programming language is top-down, left-right backtracking Prolog.

2.3 Deductive Databases

Deductive databases result from the integration of relational database and logic programming techniques. To show this clearly, let us first analyze the difference and connection between relational databases and Prolog, and the advantages and disadvantages of each.

Pure Prolog is based on Horn-clause logic and a sequential execution-control model. Rules are searched and goals are examined in the order in which they are specified (SLD resolution). Thus, the responsibility for the efficient execution and termination of programs rests with the programmer: an improper ordering of the predicates or rules may result in poor performance or even in a non-terminating program. In addition, a number of extra-logical constructs (such as the *cut*) have been grafted onto the language, turning it into an imperative, rather than a purely declarative language.

Relational systems are superior to standard implementations of Prolog with respect to ease of use, data independence, suitability for parallel processing and secondary storage access [44]. The control over the execution of query languages is the responsibility of the system which, via query optimization and compilation techniques, ensures efficient performance over a wide range of storage structures and database demographies. The working assumption is that the volume of data to be manipulated is too large to be contained in the memory of a computer and hence, that special techniques for secondary memory data access and update must be employed.

However, the expressive power and functionality offered by a relational database query language is limited compared with the logic programming languages. Besides, relational query languages are often powerless to express complete applications, and are thus embedded in traditional programming languages. This method causes an *impedance mismatch* [31, 48] between programming and relational query languages.

Prolog, on the other hand, can be used as a general-purpose programming language. It is in fact being used so with great success in varied applications such as symbolic manipulation, rule-based expert systems and natural language parsing.

Now let us see the inherent connection between the relational model and Prolog. A logic program can be considered as a natural and powerful generalization of the relational model [18, 46, 38] because any relational tuple can be expressed as an atom, i.e., a predicate of the form $p(t_1, ...t_m)$. Relational databases can be considered from the viewpoint of logic in two different ways: either the *model-theoretical view* or the *proof-theoretical view.* When considered from the model-theoretical view, queries and integrity constraints are clauses that are to be evaluated on the interpretation using the semantic definition of truth. From the proof-theoretical view, queries and integrity constraints are considered to be clauses that are to be proved. In order to determine answers to queries, using the latter view, one can derive data from the given clauses, therefore achieving the desired deductive power.

The use of mathematical logic in describing relational database models has helped to solve a number of important problems, including the definition of formal query languages, the treatment of incomplete information (null values) in databases, and the definition and enforcement of integrity constraints. The primary attraction of logic here is the elegant formalism capable of expressing facts, deductive information, integrity constraints, and queries in a uniform way. Besides, by using first-order logic as a database language, it is possible to explore well-developed techniques of theorem proving for providing powerful deductive tools. Lastly, logic provides a firm theoretical basis upon which one can pursue the conventional data model theory in general.

Based on the above comparison, it seems possible and productive to combine these two approaches and get the benefits of both. This combination did result in a new topic in computer science called *deductive databases.*

The advantages of deductive databases can be summarized as follows:

(1). Representational and operational uniformity. Horn-clause form can be used to express facts, integrity constraints, deductive information, and queries in a uniform way.

(2). By using first-order logic as a database language, it is possible to use well-developed techniques of theorem proving to provide powerful deductive tools.

(3) Logic provides a firm theoretical basis upon which one can pursue problems of relational data model theory in general such as the treatment of incomplete information (null values) in databases, and the definition and enforcement of integrity constraints.

(4). It has great potential to be efficiently implemented based on the existing relational database and Prolog technology.

As the theoretical basis has been formed, the next thing is to efficiently implement deductive databases. There are two ways. One can be termed as loosely-coupled Prolog and relational databases. The other as tightly-coupled Prolog and relational databases.

In loosely-coupled systems, the connection between Prolog and relational databases is obtained by building an interface. The large collection of Prolog facts is managed in secondary storage by using the existing relational database technology, for example, NU-Prolog does this [42]. This approach suffers from a mismatch between the computational models of these coupled subsystem: Prolog is oriented towards a fact (or tuple) at a time model, while relational model is oriented towards a set at a time.

Tightly-coupled systems, on the other hand, use a logic-based language like Prolog, but is free of the sequential execution model and other spurious constructs of Prolog. It is based on bottom-up, fixpoint computation by extending database compilation and optimization techniques to handle the richer functionality of the language. LDL [44] is an example of this kind.

Chapter 3

Motivation

Even though deductive databases have the advantages given in the previous chapter, they have significant inherent problems. Here I will discuss two broad areas where problems are apparent: one is complex object modeling; the other is higher-order features. These problems lie in the expressiveness of deductive database languages. Since their expressiveness depends on the underlying relational database and first-order logic languages, I will focus my discussion on relational databases and first-order languages. Interweaved with the discussion of the problems, I will also survey some of the attempts that have been made to solve these problems. This gives the motivation for S-logic.

3.1 Complex Object Modeling

In many novel applications, such as CAD/CAM, office information systems, decision support systems, knowledge systems, and database management systems, it has been realized that there are a host of powerful data modeling concepts which need to be introduced in both programming languages and database models. One of these concepts is the need to model arbitrarily complex objects. The ability to model complex objects is a main feature of modern object-oriented programming languages. The goal of complex object modeling is to naturally represent object identity, data abstractions and inheritance [20, 26, 31]. Relational data model using flat relation structures and primary keys is not suitable for complex object modeling. In recent years, it has been recognized that Prolog is also not rich enough to naturally represent object identity, data abstractions and inheritance. The following subsections will show the reasons.

3.1.1 Object Identity

With Prolog, two facts cannot share subparts. Let us first see an example. Suppose we have two facts in Prolog:

book("Prolog", author(name("Bob", "Su"), address("257 9th Av SW")).
book("Databases", author(name("Bob", "Su"), address("257 9th Av SW")).

both talking about the same individual, and if he moves, both facts should be updated. Further more, the change is made by retracting those facts and asserting new facts:

book("Prolog", author(name("Bob", "Su"), address("128 2nd St NW")).
book("Databases", author(name("Bob", "Su"), address("128 2nd St NW")).

In interpreting these new facts, nothing about them requires that the domain element representing *name("Bob", "Su")* in the first case is the same as in the second. There is no way to say that everything stayed the same except the address. The problem here is that two facts cannot share subparts in the Horn clause logic programming language Prolog. The subpart is an object. We need something to identify it and we should be able to refer to its identity. A solution to this problem requires explicitly using a meaningless identifier to represent each object. For example, we can use:

book("Prolog", m1).
book("Databases", m1).
author(m1, name("Bob", "Su"), address("257 9th Av SW")).

This solution requires systematically introducing "meaningless" terms such as *m1* to identify the corresponding object by the user. There ought to be some way of doing this without exposing the inner organization of the database. This capability is called object identity [16].

Object identity is the property of an object that distinguishes it from all others regardless of their content, location, or addressability [30, 16]. Two criteria are used to measure the degree of content and location independence which the object identity provides: data independence which means that identity is preserved through changes in either data values or structure; location independence which means that identity is preserved through movement of objects among physical locations or address spaces.

The solution given above which uses $m1$ to identify the object is not data independent. Some programming languages using variables to represent identity are not location independent.

In relational databases, the notion of user-defined primary (identifier) keys is used to represent the identity of an object. This representation of identity is supported in many existing database systems. However, it is not data independent either. Any change to the identifier keys will cause a discontinuity in identity.

As discussed in [16], the most powerful technique for supporting object identity is using surrogates. Surrogates are system-generated globally unique identifiers, completely independent of any physical location and data associated with objects.

3.1.2 Data Abstractions and Inheritance

The growing demand for systems of ever-increasing complexity and precision has stimulated the need for higher-level concepts, tools and techniques in every area of Computer Science. Many of these techniques are based on abstraction mechanisms that advocate the development of software in a stepwise fashion, each step involving only some of the details of the whole problem while others, hopefully the less relevant ones, are suppressed until some later step. An *abstraction* is a simplified description that emphasizes some of the system's details or properties while suppressing others [8]. Abstractions can be used to organize and structure pieces of information into some natural and conceptually meaningful units (usually hierarchies). All the details of representing or implementing such a structure can be ignored at this abstracted level by the user of the structure so that the user can then just concentrate on objects and relationships between them and obtain more meaningful units. Each such unit of information is easily accessible in the system.

Essential to modeling complex objects are the following four abstractions which are used to describe properties of different aspects of objects [2, 3, 8, 19, 21, 36, 40, 41].

Classification is a form of abstraction in which a collection of objects is considered as a higher-level object class. An *object class* is a precise characterization of all properties shared by each object in the collection. An *object* is an instance of an object class if it has the properties defined in the class. Classification represents an **instance-of** relationship between an object class in a schema and an object in a database. For example, an object class `employee` that has properties `name`, `age`, and `address` may have as an instance the object with property values *"Bob"*, 25, *"257 9th Av SW"*. For another example, a `Best_Selling_Book` object class consists of all `Book` objects with sales greater than $10,000. Classification provides a mechanism for the specification of the type of a specific object. In the reminder of the chapter, "object" is used to refer to object classes and the associated objects except when the two concepts must be distinguished.

Aggregation, generalization, and association are used to relate objects. Some properties of an object are determined through inheritance by the role it plays in one or more of these relationships. *Aggregation* is a form of abstraction in which a relationship between *component objects* is considered as a higher-level *aggregate object.* This is a **part-of** relationship. For example, class `person` could be an aggregate of its component class `name`, `age`, and `address`.

Generalization is a form of abstraction in which a relationship between *category objects* is considered as a higher-level *generic object.* It represents the **is-a** relationship. For instance, `employee`, is a generalization of the classes `secretary`, `manager` and `accountant`.

Association is a form of abstraction in which a relationship between *member objects* is considered as a higher-level *set object.* This is the **member-of** relationship. For example, the set object `trade-union` is an association of the member class `employee`, each object in `trade-union` is a set in which each element belongs to `employee`.

Inherent in these four abstractions is property inheritance. Property inheritance means that all properties of an object class are passed on to other objects or object classes. Generalization and classification support downward inheritance. For example, class `secretary`, `manager` and `accountant` inherit all properties of the class `employee`, while `john` inherits all the properties its object class `person` possesses such as `name`, `age`, and `address`. Aggregation and association support upward inheritance. For example, the properties of `name`, `age`, and `address` are inherited by the aggregate class `employee`, while the properties of `employee` such as `name`, `address` and `profession` are inherited by its set class `trade-union`.

Property inheritance enables us to reduce the redundancy of the specification while maintaining its completeness. Using some of the four abstractions for relating objects and constructing new objects, we can express very complex objects and take advantage of inheritance.

There are other abstraction mechanisms, but the four above have received the most attention in both the programming language and database areas and are especially suitable for databases applications.

The relational model has been found inadequate to support complex object modeling. Its structure is too simple to naturally support sets, general hierarchies or class (or type), subclass and instance taxonomies. Relationships between objects have to be kept in the user's mind and obtained by using join operations. From the implementation point of view, using relational databases for complex object modeling is very costly in both speed and storage space because of the join operations and redundant information in the relationships.

It is argued in [38] that Prolog could use logical implication to express data abstractions and inheritance. However Hassan and Nasr claim in [20] that using logical implication to represent data abstractions and inheritance does not naturally represent what we mean. For example, when it is asserted that "whales are mammals", we understand that whatever properties mammals possess should also hold for whales. In traditional logic, this meaning of inheritance can be well captured by the semantics of logical implication:

$$\forall x Whale(x) \Rightarrow Mammal(x)$$

This is indeed *semantically* satisfactory. However, it is argued that it is not *pragmatically* satisfactory. In a first-order logic deduction system using this implication, inheritance from "mammal" to "whale" is achieved by an *inference* step. But the special kind of information expressed in this formula somehow does not seem to be meant as a deduction step—thus lengthening proofs. Rather, its purpose seems to be to accelerate, or focus, a deduction process—thus shortening proofs. I do not intended to enter the debate here as to whether implication can be made efficient or not. Rather, I have, as a practical measure, assumed that some other more specific mechanism will be needed.

3.1.3 Solutions

As a result of the lack of expressiveness in both relational databases and first-order languages, some attempts have been made to provide the missing functionalities.

Semantic Data Models Based on conventional data models, a number of new data models called semantic data models have been developed to provide increased expressiveness to the modelers and incorporating a richer set of semantics into the stored data [6, 7, 8, 21, 36].

Many semantic data models support object identity by classifying objects into two kinds: abstract objects and printable objects. Abstract objects are referenced using internal identifiers while printable objects are referenced by themselves, i.e. printable objects are also their object identifiers. The primary reason for this is that abstract objects may not be uniquely identifiable using printable attributes that are directly associated with them.

Most semantic data models support the four data abstractions and inheritance in the way discussed earlier. The complex objects are first classified into object classes, then, generalization is used to represent the *isa* relationships between object classes, association is used to represent set classes and aggregation is used to describe the properties of each object class. The result of using these abstractions forms the schema of the semantic database.

Figure 3.1 shows part of a schema definition based on TAXIS [33], where each *class* is an aggregation which defines a class and its properties by using other classes. Generalization is represented by the *isa* relationship over the classes. If (A *isa* B) then every definitional property of B is also a definitional property of A. Moreover, A can have additional properties that B does not have at all, or it can redefine some of the properties of B. *Entity* is an metaclass with no property, *Person* is defined as a class with four properties: *name*, *sex*, *age*, and *address*. *Person* is also a generalization of *student* and *employee*, therefore *student* inherits the *name*, *sex*, *age*, and *address* properties of *person* but redefines the *age* property by restricting *age* value. Besides, *Student* has its own properties, such as *studying_in* a *Dept*, *taking* a number of *Courses*, and *borrowing* a number of *Books*, where *set of Course* and *set of Book* are associations. In TAXIS, inheritance can be multiple. *workingstudent* is an example in Figure 3.1, which inherits all the properties of *Student* and *Employee* and redefines the *salary* property. A class defined by $\{|0 :: 120|\}$ is called *finitely defined.* It has a finite collection of instances including all integers from 0 to 120.

In semantic data models, objects are directly related by these abstractions, so the relationship between objects can be obtained without using the *join* operation.

```
class Person isa Entity with
        name: String;
        sex: {|'Male', 'Female'|};
        age: {|1 :: 120|};
        address: String;
end Person.

class Student isa Person with
        age: {|15 :: 30|};
        studying_in: Dept;
        taking: set of Course;
        borrowing: set of Book;
end Student.

class Employee isa Person with
        age: {|20 :: 65|};
        working_in: Dept;
        salary: {|0 :: 50000|};
end Employee.

class Workingstudent isa Student, Employee with
        salary: {|0 :: 15000|};
end Workingstudent.

class Dept isa Entity with
        name: String;
        head: Employee;
end Dept.

class Course isa Entity with
        name: string;
        credit: {|0 :: 10|};
end Course.

class Book isa Entity with
        name: string;
        author: person;
        price: {|0 :: 150};
end Book.
```

Figure 3.1: Examples of Semantic Database Schema

ψ-Terms In logic programming, extended first-order terms called ψ-terms are proposed in [20] to replace traditional terms. A ψ-term consists of a type constructor, labels, and sub-ψ-terms. Examples of ψ-terms analogous to those in Figure 3.1 are given in Figure 3.2. It should be noted that ψ-terms do not support set classes so that taking a number of courses and borrowing a number of books can not be directly represented.

```
(1)person(name ⇒ string;
          sex ⇒ ['Male', 'Female'];
          age ⇒ [0...120];
          address ⇒ string).

(2)student = person(age ⇒ [15...30];
                    studying_in ⇒ dept).

(3)employee = person(age ⇒ [20...65];
                     working_in ⇒ dept;
                     salary ⇒ [0...50000]).

(4)workingstudent < student.

(5)workingstudent = employee(salary ⇒ [0...15000]).

(6)dept(name ⇒ string;
        head ⇒ employee).

(7)course(name ⇒ string;
          credit ⇒ [0...10]).

(8)book(name ⇒ string;
        author ⇒ string;
        price ⇒ {|0 :: 150|}).
```

Figure 3.2: Examples of ψ-Terms

In Figure 3.2, the first entry defines a ψ-term called *person* which has attributes *name*, *sex*, *age* and *address*, and sub-ψ-terms *string*, {*'Male', 'Female'*}, [1...120], and *string*. The second entry defines *student* as a subtype of *person* which inherits all the properties of *person* but redefines *age*, and has its own property *studying_in* a *dept*. The third entry defines *employee* as a subtype of *person*. The fourth entry defines *workingstudent* as a subtype of *student* without any redefinition. The fifth entry defines *workingstudent* as a subtype of *employee* with redefinition of salary. And so on.

A ψ-term differs from a traditional term. It is not a fixed-arity term. Its arguments are identified by attribute labels, not by position. It allows information to be organized into a meaningful hierarchy not just over flat structures. A ψ-term in fact is an aggregation hierarchy. The aggregates are called type constructors. Association is not supported. Generalization is represented either by the partial order $<$ on the set of type constructors like (4), or by a ψ-term definition like (2), (3) and (5) in Figure 3.2. Both are used to define the subtype relationships. The subtype relation in a ψ-term reflects a *set inclusion* interpretation, i.e., if the set of students is contained in the set of persons, then the type *student* is a subtype of the type *person*.

The motivation of the ψ-term approach is to extend the unification algorithm to compute the ψ-term that is the greatest lower bound of two given ψ-terms. The major contributions of the paper [20] is that a ψ-term is more meaningful and expressive than the traditional term. Besides, the extended unification algorithm can replace costly resolution to draw inheritance information which might result in more efficient Prolog systems. The ψ-term approach can support classification, aggregation, and generalization with set inclusion semantics. But, it does not support object identity and association. Besides, the paper only considers unification and is not general enough for the object-oriented aspects of deductive databases.

O-Logic family An extended first-order logic called O-Logic (logic for objects) is described in [31]. The kernel of O-logic is the O-term, which is used to represent complex objects. An O-term is similar to a ψ-term, with a class name, a variable or a data value, labels and sub-O-terms. Following is an example of O-terms where *employee*, *person*, *dept*, *string*, and *number* are class name; E, D, and P are variables; *"Bob"*, *"Male"*, 40, *"CPSC"*, and *"John"* are data values; and *name*, *sex*, *age*, *working_in* and *head* are labels.

$$
\begin{aligned}
&employee{:}E(name \longrightarrow string{:}\text{“Bob”};\\
&\qquad sex \longrightarrow string{:}\text{“Male”};\\
&\qquad age \longrightarrow num{:}40;\\
&\qquad working\_in \longrightarrow dept{:}D(name \longrightarrow string{:}\text{“CPSC”};\\
&\qquad\qquad head \longrightarrow person{:}P(name \longrightarrow string{:}\text{“John”}))).
\end{aligned}
$$

Clearly, O-logic supports aggregation and classfication. The class information can be generalized into a schema for the database. But it does not support association. Also generalization and the corresponding inheritance are not well-defined. Besides, object identity is represented by a variable in O-logic which has the problems discussed in Section 3.1.1.

C-logic [12] extends O-logic by introducing function symbols to represent object identity so that the quantification problem can be expressed explicitly. For example, to represent the quantification $\forall X \forall Y \exists C$ in C-logic, one can use $id(X, Y)$ directly to stand for the object C. This is consonant with the surrogate representation of object identity. Besides, C-logic can be transformed to first-order logic so that the semantics is well-defined. However, C-logic cannot represent inheritance naturally because it still uses logical implication for that purpose.

Based on O-logic, an extended O-logic was first presented in [25], followed by a more general F-logic (frame logic) in [26] as a solution to the problems of O-logic. An F-term consists of a class to which the complex objects belong, an object constructor, labels and sub-F-terms which can be single-valued or set-valued. An example of the F-term is given below where *student*, *string*, *int*, *dept*, *course*, and *book* are class objects, *bob*, *cpsc*, *cs*433, *cs*521, *prolog*, *databases*, and *math* are object identities, *name*, *sex*, *age* are single-valued labels and *taking* and *borrowing* are set-valued labels. It says that object *bob* identifies a student called Bob, male, aged 25, studying in the computer science department, who takes a number of courses identified by *cs*433 and, *cs*521 and borrows a number of books identified by *prolog*, *databases*, and *math*.

$$\begin{aligned}
student : bob[&name \rightarrow string : \text{“}Bob\text{”};\\
&sex \rightarrow string : \text{“}Male\text{”};\\
&age \rightarrow int : 25;\\
&studying\_in \rightarrow dept : cpsc\};\\
&taking \rightarrow \{course : cs433, course : cs521\};\\
&borrowing \rightarrow \{book : prolog, book : databases, book : math\}].
\end{aligned}$$

F-terms can support both aggregation and association. A generalization hierarchy is represented by specifying F-terms. For example, to represent *person* is a generalization of *student* and *employee* in F-logic, we can use *person:student*[...] and *person:employee*[...]. In this way, all classes and objects form a lattice, and thus inheritance can be reasoned about. In order to avoid higher-order semantics, F-logic does not support classification. There is no distinction between an individual object and an object class. An object can play dual roles: an instance of its class and a class of its instances. For example, *student* has dual roles in *student:bob*[...]

and *person:student*[...]. In this way F-logic achieves its so-called higher-order syntax with first-order semantics.

F-logic is a general logic language compared to the ψ-term approach, and has a well-defined semantics compared to O-logic. However, it has several problems which prevent it from being a feasible deductive database language. It is impossible to define an overall schema for the database in F-logic because object classes and individual objects are interwoven together. But the database schemas are essential to many database applications. We can only have individual objects with arbitrary properties in F-logic databases. Besides, the dual roles of objects make the semantics of F-logic quite complicated and unintuitive. Also, the lattice of F-logic is over all objects not just over object classes, to specify a complex object we should also put it into the lattice by specifying its class and its instances. Besides, the lattice of F-logic is in reverse order from normal. However, specifying a lattice is quite different from specifying an object. This makes F-logic non-uniform, difficult to use, and hard to update to maintain integrity.

3.2 Higher-Order Features

In this section, I discuss two important higher-order features needed in deductive databases. One is how to define and manipulate database schema and data in an integrated framework. The other is how to deal with set expressions.

3.2.1 Uniformity of Schema and Data

In traditional deductive databases, the definition and manipulation of data is done in a uniform way. However, the definition and manipulation of schema (or meta) information and data is not supported in such an integrated framework. To reason about schema information we need the capabilities of higher-order logic. For example, we might need to express a query which contains a higher-order variable X to list all the predicates in the database. The substitution for X should range over all the predicates. Similarly, we might need to express queries containing higher-order variables to list all attributes in the database or attributes in a predicate, etc. But higher-order logic have been met with skepticism since the unification problem is undecidable. So normally, a separate language is provided to specify and manipulate the schema information.

In fact, deductive database applications require a rather restricted form of higher-order logic. Can we provide a direct semantics for them?

A higher-order language for deductive databases is proposed in [27] which encompasses meta-data and data by allowing higher-order predicates to be defined in the language. The solution to higher-order unification is based on a bottom up semantics where unification is replaced by matching, i.e. only one of the two terms contains variables. The higher-order variables are limited to range over database attributes and predicates. A rule with higher-order variables can be rewritten by replacing variables with attributes or predicates. The rewritten rules are in first-order logic and their meaning is well-defined. Since the number of database attributes and predicates have to be finite, this makes the language decidable. This semantics is called *replacement semantics* in that paper. Even though the higher-order language proposed in that paper is not rich enough to represent object identity, data abstractions and inheritance, etc., the replacement semantics for higher-order variables in that paper is of great value. It provides a natural way toward the integration of the definition and manipulation of schema and data.

F-Logic [26] has an appearance of a higher-order logic, but it is tractable and has first-order semantics. It is capable of modeling certain higher-order features such as sets, class/subclass hierarchy and schemas. The first feature is modeled by means of set-valued functions described in Section 3.1.3. F-logic reifies classes and model membership by means of a lattice ordering instead of the true set-theoretic membership. It does not distinguish between individual objects and classes. All the objects are taken from the same domain and are organized into a lattice. The same object can be viewed as an instance of its superclass which is below it in the lattice, and at the same time, as a class of all objects located above it in the lattice. So, any element p may appear in an F-term in the instance position, q:p[...] and in the class position, p:r[...]. This gives F-Logic a feel of a higher-order language, but it is essentially first-order.

3.2.2 Set Expressions

The addition of set expressions to Prolog is very important because many problems cannot be expressed in pure Prolog without such an extension [47]. The extension takes the form of a built-in predicate:

$$setof(X, P, S)$$

which is read as "The set of instances of X such that P is provable is S". The term

P represents a goal or goals. The term X is a variable occurring in P. The set S is represented as a list whose elements are sorted into a standard order without any duplicates.

Unfortunately, there is no published formal semantics for for the *setof* predicate in Prolog. Besides, the usage of lists as sets is not expressive enough. The simple membership predicate has to specify details about implementation, such as how to iterate over the sets. When a predicate involves more than one set, the program can become quite complicated and unintuitive, which is contrary to the general philosophy of logic programming [28].

LDL [4, 44] proposed a different way of expressing sets based on bottom-up fixpoint computation. Set terms can be generated in LDL by using two constructors: set-enumeration and set-grouping. Set enumeration is the process of constructing a set by listing its elements. For example, if we want to derive a relation on sets of book titles from the *book* base relation such that their total price does not exceed \$100. We can use the following rule in LDL:

$$\begin{aligned} book\_deal(\{X, Y, Z\}) \leftarrow\ & book(X, P_x), book(Y, P_y), book(Z, P_z), \\ & X \neq Y, X \neq Z, Y \neq Z, \\ & P_x + P_y + P_z < 100. \end{aligned}$$

The same thing can not be directly represented in Prolog.

Unlike set-enumeration, in set-grouping, the set is constructed by defining its elements by a property (i.e., a conjunction of predicates) that they satisfy. The following example shows a set-grouping in which all of the parts supplied by a supplier are grouped with the supplier number, where $< P\# >$ represents the constructed set.

$$part\_sets(S\#, < P\# >) \leftarrow supplier(S\#, P\#).$$

LDL is based on traditional first-order logic, so the problems of complex object modeling discussed in the previous section still exist. Besides, its use of sets and grouping has other severe semantic problems which result from LDL's syntactic limitations.

Let's first look at a program which consists of a single fact in Prolog:

$$q(2).$$

The possible models for the program are $\{q(2)\}, \{q(1), q(2)\}, \{q(1), q(2), q(3)\}$. The intersection of these models is the minimal model {q(2)}. However this property does not hold for LDL because of the sets. Consider the following example:

$$q(2).$$
$$p(< X >) : -q(X).$$

This program may have the following models:

$$\{q(1), q(2), p(\{1, 2\})\},$$
$$\{q(2), q(3), p(\{2, 3\})\},$$
$$\{q(1), q(2), q(3), p(\{1, 2, 3\})\},$$

The intersection of the above models is not a model as it does not contain $p(\{2\})$. The reason is that the predicate p freezes its arguments such that $p(\{1, 2\})$ and $p(\{1, 2, 3\})$ are not comparable. To solve this problem, LDL introduces the unnatural and complex concept on top of the notion of minimality.

Let us look at another example in LDL which is equivalent to the famous set-theoretic paradox:

$$p(X).$$
$$p(< X >) : -p(X).$$

There is no model for this program because p has to contain the set of all sets as an element. The main reason for this problem is that the same field can have both a set value and a single value. To cope with these kinds of problems, LDL introduces a restriction called stratification on its programs which would forbid the program above.

C-logic [12], Extended O-logic [25], and F-logic [26] support sets by using set-valued (or multi-valued) labels. They combine set-enumeration and set-grouping into a single form. For the above set grouping example of LDL, the equivalent in F-logic is:

$$\begin{array}{l} supplier\_set : id(S\#)[sno \rightarrow S\#, supply\_set \rightarrow \{P\#\}] \Leftarrow \\ \qquad supplier : X[sno \rightarrow S\#, supply \rightarrow P\#]. \end{array}$$

where $id(S\#)$ is an object constructor which stands for exactly one object identity for each $S\#$, *supply_set* is a set-valued label, *sno*, *supply* are single-valued labels and $\{P\#\}$ is a set grouping notation. The rule groups all of the parts supplied by a supplier into a set represented by $\{P\#\}$.

As discussed in [25], the sets here are flat, i.e., a set may contain only object identities as its elements, not other sets. But an object identifiers may represent a set, so that sets of arbitrary depth can be modeled. The use of single-valued labels and set-valued labels can syntactically avoid the semantic problems of LDL. More discussions on these aspects can be found in [25].

3.3 Motivation of S-logic

Approaches to deductive databases are torn by two opposing forces. On one side there are the stringent real-world requirements of actual databases. The requirements include efficient processing as well as the ability to express complex and subtle real-world relationships. On the other side are the simple and clear semantics of logic programming and its deductive power. The need for expressiveness has forced the deductive databases away from their simple roots in logic programming and relational techniques.

In this chapter, I have discussed two problems underlying the deductive databases, and shown a number of solutions, such as semantic data models, ψ-terms, O-logic, C-logic, F-logic, and LDL. But none of them can solve all three problems. Is it possible to obtain an uniform language which is expressive enough to solve all these two problems? The answer is yes. The rest of this thesis proposes a language for deductive databases called S-logic, which can meet these needs. To model complex objects, the proposed language should be able to model object identity, data abstractions and inheritance, i.e. represent semantic data models in general. Besides, it is also a logic language. It should be able to logically represent the existence and internal structure of complex objects. I term this language S-logic. To represent schema and sets, S-logic should also be a higher-order language.

Chapter 4

Informal Presentation and Examples

An S-logic program consists of four parts: type system, database, rules, and queries. The type system is the schema of the database and rules, which consists of all type definitions, and a lattice via the subtype (i.e. subset) relationship over all types. The database consists of all objects that satisfy the type system. Rules are used to represent deductive information over the database. Queries are defined over the database, rules, and type system in a uniform way to obtain information from the program.

4.1 Type Systems

A type in S-logic is a named class object which has two aspects. Under the dynamic aspect, a class object denotes the set of objects (object identities) in the class, and such membership may be changed by updates. This aspect is called the *extension* of the class object (or type). The static aspect represents common structural properties of all objects in the class, i.e., the properties an object needs in order to belong to a certain class. This may be changed if the type system is modified. This aspect is called the *intension* of the class object (or type). A type here naturally corresponds to a classification as discussed in Chapter 2. That an object belongs to a type means its object identity belongs to the set denoted by the type. In S-logic, objects are divided into two non-overlapping kinds: abstract objects and printable objects.

Abstract objects are identified by their object identities, while printable objects are identified by themselves.

A type based on printable objects is normally defined by specifying all the objects belonging to this type, i.e., its extension. A type based on abstract objects is defined by specifying the properties which should be satisfied by all of its objects, i.e., its intension, and leaving the extension with the database.

There are four kinds of types in S-logic: basic types, record types, set types, and built-in types.

The basic types include integer, string, and their subsets. Subsets are defined either by enumerating or by specifying the ranges and have type names associated with them. For example, $agetype = integer(\{1..120\})$ specifies a subset of integer, each element of which is between 1 and 120 inclusive, while $gender = string(\{\text{'}Male\text{'}, \text{'}Female\text{'}\})$ specifies a subset of string called *gender* which has only two elements *Male* and *Female*. All objects of basic types are also called printable objects.

A record type consists of a root type and a collection of properties which have to be satisfied by all the objects belonging to this root type. A record type is used to define the intension of a root type. A property is described by an attribute label associated with a type called a component type. An attribute is a function from the root type to the component type.

An example of a record type is

$$\begin{aligned} person(&name \rightarrow string,\\ &sex \rightarrow gender,\\ &age \rightarrow agetype). \end{aligned}$$

Here *person* is a record type which has the attributes *name*, *age*, and *sex*. *name* is a mapping from *person* to *string*, read as every object in *person* has a name which is an object in *string*. *age* is a mapping from *person* to *agetype* which is a subtype of *integer*, read as every object in *person* has an age which is a integer between 1 to 120. *sex* is a mapping from *person* to *gender*, read as that the sex of every object in *person* is either male or female.

A record type naturally corresponds to an aggregation as discussed earlier. Besides, a generalization can also be represented by a record type via the *isa* label in S-logic. The *isa* label is just an identical mapping from the root type to its supertype. It is useful here because it represents a subtype (or subset) relationships between the root type and its super type and allows automatic property inheritance. Besides, properties of the super type can be redefined and additional properties can be introduced. This representation is quite similar to that in semantic data models such as TAXIS [33]. Look at the following example in S-logic:

$$\begin{aligned}student(&isa \rightarrow person,\\ &age \rightarrow young,\\ &studying\_in \rightarrow dept,\\ &taking \rightarrow \{course\},\\ &borrowing \rightarrow \{book\}).\end{aligned}$$

The example defines that *student* is a subtype of *person*. Every definitional property of *person* is also a definitional property of *student* via the *isa* label, except *age* property of *person* is redefined. Moreover, *student* has additional properties that *person* does not have at all, like studying in a department, taking a number of courses, and borrowing a number of books. The properties of *student* from inheritance are called *inherited properties*, which are *name*, *sex*, and *age*. The redefined properties and own properties of *student* are called *direct properties* which are *age*, *studying_in*, *taking*, and *borrowing*. The intended meaning here is that every object belonging to the type *student* also belongs to the type *person*, i.e., the extension of *student* is set-included in the extension of *person*. If an object, say *John*, is a *student* then he has to be a *person* first via the *isa* label. As a person, he should have all the properties of *person* with his age between 15 and 30. As a student, he also has all the properties of *student*. So, the *isa* label here can allow all the properties of *person* to be inherited by *student* without any duplication in the type system.

Based on the above discussion, we know that when an object belongs to *person* it does not mean that it has only the definitional properties of *person*, it means that it has at least those properties of *person* because it may belong to *student* and has all the properties of *student*. That a type has few properties means its objects have few restrictions and therefore are more general than those having more properties. In fact, here I give a set-inclusive interpretation to the subtype relationships. The subtype relation is reflexive, asymmetric, and transitive. Therefore it is a *partial order*. Using an *isa* label in record types might seem confusing, but I think it is syntactically expressive and semantically sound.

A set type specifies a structure consisting of elements of an identical type called the set element type. Each object of a set type is a subset of its set element type. The extension of a set type is the power set of the extension of its set element type. To prevent infinite construction of set types, S-logic only allows a set type to be constructed from a set element type which is not a set type itself. For example, *book* is a type which denotes all the books (in a library), then $\{book\}$ is a set type whose set element type is *book*. Each object of $\{book\}$ is a subset of *book*. Using a set type $\{book\}$, we can easily relate a person to a set of books. A set type corresponds to an association as discussed in Chapter 2.

There are two built-in types in S-logic. One is called *all* which has no properties at all but includes all the possible objects. According to the above discussion, an object in a type that has few properties means it may have other properties. It may belong to another type with more properties. So objects in *all* can belong to other types and every type in the type system is a subtype of *all*. The other built-in type is called *none* which has no object but has all the possible properties. Therefore, it is a subtype of all the possible types in the type system.

Figure 4.1 shows an example of a sample type system which defines seventeen types for database and rules. The first defines a record type *person* which has attributes: *name*, *age*, *sex*, *spouse*, *address*, *father*, and *mother*. Note that *person* is recursively defined. The second defines a record type *student* who is a subtype of *person* with a redefinition of the age and who studies in a department and takes a number of courses and borrows a set of book. The third defines a record type *employee* who is also a subtype of the *person* with a redefinition of his age and who works in a department and heads a number of people, who has a property salary. The fourth defines a record type *workingstudent* who is a subtype of both *student* and *employee* with two redefinitions of his age and salary. The fifth, sixth, seventh and eighth define *agetype*, *young*, *midage* and *ymage* as subtypes of *integer*. The ninth *employeesalary* as a subtype of *integer*, while the tenth also defines *support* as a subtype of *integer*. The eleventh defines *gender* as a subtype of *string*. The twelfth defines a record type *dept* which has a department name and a head who is an employee, and the staff who are employees working in the dept. The thirteenth defines a record type *course* which has a course name, its credit, and a number of students who take the course. The fourteenth defines a record type *book* which consists of the four parts: the book's name, its author, its publisher, and its price. The fifteenth defines a record type *family* which has a father, a mother, and a number of children. The sixteenth defines a record type *house* which has a location in which a number of people live. The last defines a record type *sameage* which is intended to describe how many people have the same age.

Not all subtype definitions are useful. Suppose we have subtype relationships defined by following type definitions:

(1) $employee(isa \rightarrow tourist)$.
(2) $tourist(isa \rightarrow businessman)$.
(3) $businessman(isa \rightarrow employee)$.

According to the set-inclusive interpretation, these three types will contain the same set of objects. Therefore, these types are redundant.

Besides, not all redefinitions of the properties of a subtype are meaningful. Let $person(age \rightarrow agetype)$ be a record type, *student* be a subtype of *person*. Since the *age* property of *person* is *agetype*, the *age* property of *student* has to be a subtype of *agetype*. Multiple inheritance makes things a little tricky. Suppose *student* has an attribute called social insurance number (SIN) of which the range is from 100000 to 199999, *employee* also has a SIN of which the range is from 200000 to 399999, and *workingstudent* is a subtype of both *student* and *employee*, then *workingstudent* can not inherit the SIN property but redefine the SIN property. According to the set-inclusive interpretation, an object in *workingstudent* is also the same object in both *student* and *employee*. Its SIN should belong to the range from 100000 to 199999 and the range from 200000 to 399999. So, no objects of *workingstudent* can have this property. On the other hand, if the intersection of both ranges is not empty,

(1) $person(name \rightarrow string,$
$\quad sex \rightarrow gender,$
$\quad age \rightarrow agetype,$
$\quad spouse \rightarrow person,$
$\quad address \rightarrow string,$
$\quad father \rightarrow person,$
$\quad mother \rightarrow person).$

(2) $student(isa \rightarrow person,$
$\quad age \rightarrow young,$
$\quad studying\_in \rightarrow dept,$
$\quad taking \rightarrow \{course\},$
$\quad borrowing \rightarrow \{book\}).$

(3) $employee(isa \rightarrow person,$
$\quad age \rightarrow midage,$
$\quad working\_in \rightarrow dept,$
$\quad heading \rightarrow \{person\},$
$\quad salary \rightarrow employeesalary).$

(4) $workingstudent(isa \rightarrow student, isa \rightarrow employee,$
$\quad age \rightarrow ymage, salary \rightarrow support).$

(5) $agetype = integer(\{1..120\})$

(6) $young = integer(\{15..30\})$

(7) $midage = integer(\{25..60\})$

(8) $ymage = integer(\{25..30\})$

(9) $employeesalary = integer(\{0..50000\})$

(10) $support = integer(\{0..15000\})$

(11) $gender = string(\{\text{'Male'}, \text{'Female'}\}),$

(12) $dept(name \rightarrow string, head \rightarrow employee, staff \rightarrow \{employee\}).$

(13) $course(name \rightarrow string, credit \rightarrow integer, taken\_by \rightarrow \{student\}).$

(14) $book(name \rightarrow string, author \rightarrow person,$
$\quad published\_by \rightarrow string, price \rightarrow integer).$

(15) $family(father \rightarrow person, mother \rightarrow person, children \rightarrow \{person\}).$

(16) $house(location \rightarrow string, occupied\_by \rightarrow \{person\}).$

(17) $sameage(number \rightarrow agetype, shared\_by \rightarrow \{person\}).$

Figure 4.1: A Sample Type System

then *workingstudent* can only have an non-empty subset of the intersection as the range of its SIN. In Figure 4.1, the range of the attribute *age* of *workingstudent* is redefined as *ymage*, a subset of both *young* and *midage* which are ranges of the attribute *age* of both supertypes *student* and *employee* respectively. In general, if several types have a common attribute, their subtype can not inherit but redefine this common attribute of which the component type must be a subtype of the component types of the common attribute of its supertypes. Look at another example. Let $a(l \rightarrow student)$ and $b(l \rightarrow employee)$ be two record types and c is a subtype of both a and b. If *workingstudent* is a subtype of both *student* and *employee*, then $c(isa \rightarrow student, isa \rightarrow employee, l \rightarrow workingstudent)$ is a meaningful record type.

All types except set types in the S-logic's type system forms a (complete) lattice based on subtype relationships. A partially-ordered set $< S, \leq >$ is a lattice, in mathematical sense, if the least upper bound and the greatest upper bound exist for every subset of S. Types in the lattice are either built-in, user-defined, or inferred from what the user has defined. The top element of the lattice is *all* and the bottom element is *none*. Every type has its biggest subtypes under it in the lattice. Figure 4.2 shows a lattice based on the type system of Figure 4.1. In the lattice, *all* has as biggest subtypes *string*, *sameage*, *dept*, *family*, *integer*, *course*, *person*, *book* and *house*; *string* has *gender* as its biggest subtype; *integer* has *agetype* and *employeesalary* as its biggest subtypes; *person* has *student* and *employee* as its biggest subtypes; *gender*, *sameage*, *dept*, *family*, *ymage*, *support*, *course*, *book*, *workingstudent*, and *house* have *none* as their biggest subtypes; and so on.

In summary, a type itself corresponds to a classification and its elements inherit all the properties of the type if there are any. A set type corresponds to an association.

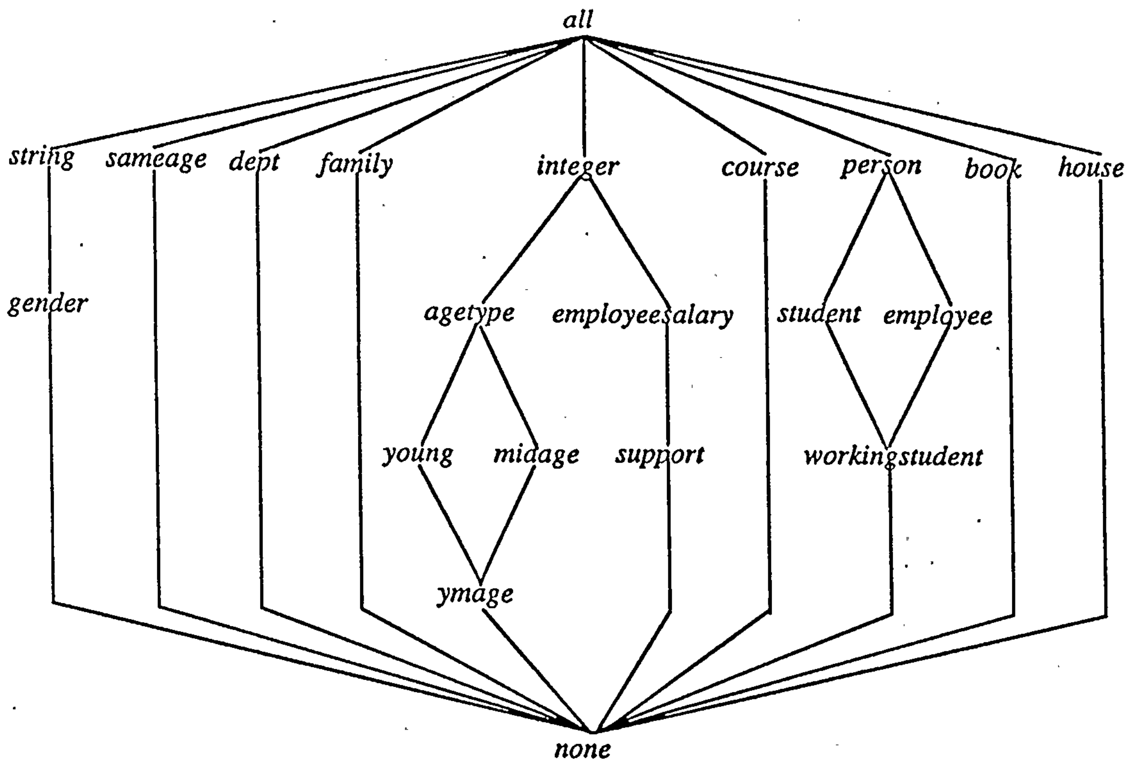

Figure 4.2: The Lattice Over the Sample Type System

A record type definition with labels other than *isa* corresponds to an aggregation. A record type with *isa* labels corresponds to a generalization and the properties of a super type are automatically inherited by this type. The type system consists of all type definitions and a lattice over all the types based on subtype relationships.

4.2 Database

The type system determines the extension of basic types, set types, and built-in types, and the intension of record types. The extension of record types is determined by the database and rules. The database in S-logic consists of all record objects, i.e., objects of the record types. Such membership can be changed by updates. So the database is not fixed, it can be changed from time to time. But here, I assume that there is a fixed database.

In the database, every record object belonging to a record type has all the properties of this type. According to the type system, a type inherits the properties of its supertypes via an *isa* label, individual objects in this type should have all the properties of this type, either of its own or from inheritance. Therefore, there can be no *isa* labels in the database. If a property of an object is not specified in the database, S-logic will assume an uninstantiated value, i.e, the attribute value exists but unknown. If a conflict happens, then *fail* will be inferred in the queries. There are two kinds of objects in S-logic, printable objects which are identified by themselves, and abstract objects which are identified by their identities (or surrogates) which are data independent and location independent. It is easy to tell whether an object is printable or abstract according to the type system. In S-logic, the database only tells what is known. The unknown information can be inferred directly from the type system. Besides, the order of properties of each object is not important at all. The component types are omitted in the database for convenience.

Figure 4.3 shows a database corresponding to the type system of Figure 4.1, where abstract objects are represented by meaningful identities for a better understanding. The first says that *sally* stands for an object in the type *person*, called *Sally*, aged 14, female, whose father is an object in *person* represented by *bob*, whose mother is also an object in *person* represented by *mary*, whose spouse and address are unknown. The second says that *john* stands for an object also in the type *person*, called *John*, aged 62, male, and living at *439 5th Av NE*. Other information about this person is unknown. The third says that *jenny* stands for an object in *student*, called *Jenny*, aged 24, female, whose spouse is the object represented by *smith* in *person*, who is studying in a department represented by *math* and taking three courses represented by *m*203, *m*321, and *cs*213. The fourth also talks about a student called *Phil*, aged 18, female, whose father is *bob*, whose mother is *mary*, who is studying in a department represented by *cpsc* and taking two courses represented by *cs*213 and *cs*450, who has borrowed two books represented by *pascal* and *prolog*. The fifth says that *mary* stands for an object in *employee*, called *Mary*, aged 39, female, whose spouse is the object represented by *bob*, who is working in a department, identified by *bookstore*, and whose salary is $35000 a year. The other information about this employee is unknown. The sixth says that *henry* stands for an object in *employee* called *Henry*, aged 50, male, living at *128 2nd St NW*, who is working in a department represented by *cpsc*, and whose salary is $50,000 a year. The seventh says that *bob* is also an object in *employee*, aged 40, male, living at *257 9th Av SW*, whose father is an object in *person* identified by *john* and who is working in a department represented by *math*, and whose salary is $40,000 a year. The eighth says that *smith* stands for an object in *workingstudent*, called *Smith*, aged 30, male, who lives at *3 7th Av SW* and whose father is an object in *person* represented by *john*, who is studying in a department represented by *cpsc*, and taking a course represented by *cs*450, who is working in a department represented by *cpsc*, and whose salary is $12,000 a year. The ninth says that *dennis* is also an object in *workingstudent*, male, living at *3 7th Av SW*, whose father is *henry*, who is studying in *math* department, who is working in *bookstore*, whose salary is $8,000 a year. And so on.

(1) $sally : person(name \rightarrow$ *'Sally'*, $age \rightarrow 14, sex \rightarrow$ *'Female'*,
$father \rightarrow bob, mother \rightarrow mary)$.

(2) $john : person(name \rightarrow$ *'John'*, $age \rightarrow 62, sex \rightarrow$ *'Male'*,
$address \rightarrow$ *'439 5th Av NE'*$)$.

(3) $jenny : student(name \rightarrow$ *'Jenny'*, $age \rightarrow 24, sex \rightarrow$ *'Female'*,
$spouse \rightarrow smith, father \rightarrow henry,$
$studying\_in \rightarrow math, taking \rightarrow \{m203, m321, cs213\})$.

(4) $phil : student(name \rightarrow$ *'Phil'*, $age \rightarrow 18, sex \rightarrow$ *'Male'*, $father \rightarrow bob,$
$mother \rightarrow mary, studying\_in \rightarrow cpsc,$
$taking \rightarrow \{cs213, cs450\}, borrowing \rightarrow \{pascal, prolog\})$.

(5) $mary : employee(name \rightarrow$ *'Mary'*, $age \rightarrow 39, sex \rightarrow$ *'Female'*,
$spouse \rightarrow bob, working\_in \rightarrow bookstore, salary \rightarrow 35000)$.

(6) $henry : employee(name \rightarrow$ *'Henry'*, $age \rightarrow 50, address \rightarrow$ *'128 2nd St NW'*,
$sex \rightarrow$ *'Male'*, $working\_in \rightarrow cpsc, salary \rightarrow 50000)$.

(7) $bob : employee(name \rightarrow$ *'Bob'*, $age \rightarrow 40, address \rightarrow$ *'257 9th Av SW'*,
$sex \rightarrow$ *'Male'*, $father \rightarrow john, working\_in \rightarrow math,$
$salary \rightarrow 40000)$.

(8) $smith : workingstudent(name \rightarrow$ *'Smith'*, $age \rightarrow 30, address \rightarrow$ *'3 7th Av SW'*,
$sex \rightarrow$ *'Male'*, $father \rightarrow john, studying\_in \rightarrow cpsc,$
$taking \rightarrow \{cs450\}, working\_in \rightarrow cpsc)$.

(9) $dennis : workingstudent(name \rightarrow$ *'Dennis'*, $age \rightarrow 30, sex \rightarrow$ *'Male'*,
$father \rightarrow henry, studying\_in \rightarrow math$
$working\_in \rightarrow bookstore, salary \rightarrow 8000)$.

(10) $cpsc : dept(name \rightarrow$ *'Computer Science'*, $head \rightarrow henry)$.

(11) $math : dept(name \rightarrow$ *'Mathematics'*, $head \rightarrow bob)$.

(12) $bookstore : dept(name \rightarrow$ *'Book Store'*, $head \rightarrow mary)$.

(13) $cs213 : course(name \rightarrow$ *'Programming Language'*, $credit \rightarrow 2)$.

(14) $cs450 : course(name \rightarrow$ *'Artificial Intelligence'*, $credit \rightarrow 3)$.

(15) $m203 : course(name \rightarrow$ *'Calculus'*, $credit \rightarrow 6)$.

(16) $m321 : course(name \rightarrow$ *'Algebra'*, $credit \rightarrow 4)$.

(17) $pascal : book(name \rightarrow$ *'Pascal'*, $author \rightarrow henry,$
$published\_by \rightarrow$ *'Prentice'*, $price \rightarrow 35)$.

(18) $prolog : book(name \rightarrow$ *'Prolog'*, $author \rightarrow john,$
$published\_by \rightarrow$ *'Springer'*, $price \rightarrow 50)$.

Figure 4.3: A Sample Database

4.3 Rules and Queries

Based on the database, deductive information can be defined by using rules in S-logic. A rule in S-logic is of the form $p \Leftarrow p_1, ..., p_n$. Every $p_i (1 \leq i \leq n)$ in the body is either a positive literal which is a record object with variables, a comparison expression over variables and objects, or a negative literal which consists of a negation sign ($\neg$) and a positive literal. The head p is a positive literal.

A rule can be used to deduce attribute values for existing objects, or to describe how to construct objects and obtain their attribute values. S-logic provides functions called object constructors to construct objects. For example, $id(mary, bob)$ is an object constructor which denotes an object of proper type in the context.

Figure 4.4 shows several rules defined on the database given in Figure 4.3. The first rule obtains an address by assuming that a person lives with their father if they are less than 20 years old. With this rule, we do not need to repeat address information for a person under 20 in the database. If we specify address information in the database and we also can deduce address information, then two address values should be exactly the same because the address attribute is a single-valued label. Otherwise a *fail* will be returned. X and Z are variables over objects of *person*, A is a variable over objects of *agetype*, and Y is a variable over objects of type *string*. The second rule obtains an address by assuming that if a person is married they are assumed to live with their spouse. With this rule, we do not need to repeat address information for each couple in the database. The third rule says that the spouse relation is reflexive so that we need only describe one party, for example, the wife, instead of both parties. The fourth rule says that all students who take a certain course can be derived from each individual student who takes this course. Note here $\{Y\}$ is a set grouping notation which represents a set over which the variable Y ranges. The fifth rule says that all employees that are managed by an employee X can be derived from each employee whose department head is X. The sixth query says that all the staff who are working in a dept can be derived from each employee object who is working in this dept. The above six rules are used to deduce attribute values for existing objects. The other three rules are used to describe how to construct objects and obtain their attribute values. The seventh rule says that objects in type *family* are constructed by an object constructor *id*(X, Y). For every pair of X, Y which are known, *id*(X, Y) obtains exactly one object identity (surrogate) which is also referred to by *id*(X, Y). It also says that the attribute values of each object of *family* can be obtained from the existing objects of *person*. The eighth rule says that objects in type *house* are constructed by object constructor *id*(X) and the attribute values of each object are obtained from the existing objects in *person*. Rule 9 is similar.

Queries are defined over the database, rules, and the type system in a uniform way in S-logic. A query starts with the question mark ? and is followed by a conjunction of literals. A positive literal in a query is either a type with variables, an object with variables, a comparison expression over variables and types, or a comparison expression over variables and objects. A negation literal in a query consists of a negation sign $\neg$ and a positive literal. Note literals in a query are more general than those in a rule, which contains types and type variables.

Queries are used to ask for information which is either in the database, derivable from rules, or in the type system. If an attribute value is unknown, then an unbound variable will be returned to a query. We can omit uninteresting variables in queries for convenience.

(1) $X : person(address \rightarrow Y) \Leftarrow A \leq 20$
$X : person(age \rightarrow A, father \rightarrow Z),$
$Z : person(address \rightarrow Y).$

(2) $X : person(address \rightarrow Y) \Leftarrow$
$X : person(spouse \rightarrow Z),$
$Z : person(address \rightarrow Y).$

(3) $X : person(spouse \rightarrow Y) \Leftarrow$
$Y : person(spouse \rightarrow X).$

(4) $X : course(taken\_by \rightarrow \{Y\}) \Leftarrow$
$Y : student(taking \rightarrow \{X\}).$

(5) $X : employee(heading \rightarrow \{Y\}) \Leftarrow$
$Y : employee(working\_in \rightarrow D), D : dept(head \rightarrow X).$

(6) $X : dept(staff \rightarrow \{Y\}) \Leftarrow$
$Y : emplyee(working\_in \rightarrow X).$

(7) $id(X,Y) : family(father \rightarrow X, mother \rightarrow Y, children \rightarrow \{Z\}) \Leftarrow$
$Z : person(father \rightarrow X, mother \rightarrow Y).$

(8) $id(X) : house(address \rightarrow X, occupied\_by \rightarrow \{Y\}) \Leftarrow$
$Y : person(address \rightarrow X).$

(9) $id(X) : sameage(age \rightarrow X, shared\_by \rightarrow \{Y\}) \Leftarrow$
$Y : person(age \rightarrow X).$

Figure 4.4: Sample Rules

Figure 4.5 shows 9 sample queries and the corresponding answers based on the sample database and sample rules. The first query asks for information about all persons who are over or equal to 50 years old. The printed answer to this query, based on the database, is given in answers 1.1 and 1.2. The second query asks for information about all workingstudents who study and work in the same department. The only answer is given in answer 2.1. The third query asks for information about

(*Query 1.*)	$?X : person(age \rightarrow Y, sex \rightarrow Z), Y \geq 50.$
(*Answer 1.1.*)	$X = john, Y = 62, Z =$ *'Male'*.
(*Answer 1.2.*)	$X = henry, Y = 50, Z =$ *'Male'*.
(*Query 2.*)	$?X : workingstudent(studying\_in \rightarrow Y, working\_in \rightarrow Y).$
(*Answer 2.1.*)	$X = smith, Y = cpsc.$
(*Query 3.*)	$? : student(name \rightarrow$ *'Phil'*, $borrowing \rightarrow \{X\}),$ $X : book(author \rightarrow Y, price \rightarrow Z).$
(*Answer 3.1.*)	$X = pascal, Y = henry, Z = 35.$
(*Answer 3.2.*)	$X = prolog, Y = john, Z = 50.$
(*Query 4.*)	$?cs213 : course(taken\_by \rightarrow \{X\}), X : student(studying\_in \rightarrow Y).$
(*Answer 4.1.*)	$X = jenny, Y = math.$
(*Answer 4.2.*)	$X = phil, Y = cpsc.$
(*Query 5.*)	$?X : student(taking \rightarrow \{Y\}), Y : course(name \rightarrow Z).$
(*Answer 5.1.*)	$X = jenny, Y = m203, Z =$ *'Calculus'*,
(*Answer 5.2.*)	$X = jenny, Y = m321, Z =$ *'Algebra'*,
(*Answer 5.3.*)	$X = jenny, Y = cs213, Z =$ *'Programming Language'*,
(*Answer 5.4.*)	$X = phil, Y = cs213, Z =$ *'Programming Language'*,
(*Answer 5.5.*)	$X = phil, Y = cs450, Z =$ *'Artificial Intelligence'*,
(*Answer 5.6.*)	$X = smith, Y = cs450, Z =$ *'Artificial Intelligence'*,
(*Query 6.*)	$?X : family(mother \rightarrow mary, children \rightarrow \{Y\}), Y : (age \rightarrow Z).$
(*Answer 6.1.*)	$X = id(mary, bob), Y = sally, Z = 14.$
(*Answer 6.2.*)	$X = id(mary, bob), Y = phil, Z = 18.$
(*Query 7.*)	$?bookstore : dept(staff \rightarrow \{X\}), X : employee(salary \rightarrow Y).$
(*Answer 7.1.*)	$X = mary, Y = 35000.$
(*Answer 7.1.*)	$X = dennis, Y = 8000.$
(*Query 8.*)	$? : house(address \rightarrow X, occupied\_by \rightarrow Y).$
(*Answer 8.1.*)	$X =($*'257 9th Av SW'*$), Y = \{mary, bob, sally, phil\}.$
(*Answer 8.2.*)	$X =($*'439 5th Av NE'*$), Y = \{john\}.$
(*Answer 8.3.*)	$X =($*'3 7th Av SW'*$), Y = \{jenny, smith\}.$
(*Answer 8.4.*)	$X =($*'128 2nd St NW '*$), Y = \{henny\}.$
(*Query 9.*)	$?X : sameage(age \rightarrow 30, shared\_by \rightarrow Y).$
(*Answer 9.1.*)	$X = id(30), Y = \{smith, dennis\}.$

Figure 4.5: Sample Queries and Answers (I)

(*Query 1.*)	$?sally : X(L \rightarrow Y).$
(*Answer 1.1.*)	$X = person, L = name, Y =$ '*Sally*'.
(*Answer 1.2.*)	$X = person, L = sex, Y =$ '*Female*'.
(*Answer 1.3.*)	$X = person, L = age, Y = 14.$
(*Answer 1.4.*)	$X = person, L = address, Y =$ '*257 9th Av SW*'
(*Answer 1.5.*)	$X = person, L = spouse, Y = Y.$
(*Answer 1.6.*)	$X = person, L = father, Y = bob.$
(*Answer 1.7.*)	$X = person, L = mother, Y = mary.$
(*Query 2.*)	$?student(L \rightarrow Y).$
(*Answer 2.1.*)	$L = name, Y = string.$
(*Answer 2.2.*)	$L = sex, Y = gender.$
(*Answer 2.3.*)	$L = age, Y = young.$
(*Answer 2.4.*)	$L = spouse, Y = person.$
(*Answer 2.5.*)	$L = address, Y = string.$
(*Answer 2.6.*)	$L = father, Y = person.$
(*Answer 2.7.*)	$L = mother, Y = person.$
(*Answer 2.8.*)	$L = studying\_in, Y = dept.$
(*Answer 2.9.*)	$L = taking, Y = \{course\}.$
(*Answer 2.10.*)	$L = borrowing, Y = \{book\}.$
(*Query 3.*)	$?X(isa \rightarrow Y).$
(*Answer 3.1.*)	$X = student, Y = person.$
(*Answer 3.2.*)	$X = employee, Y = person.$
(*Answer 3.3.*)	$X = workingstudent, Y = student.$
(*Answer 3.4.*)	$X = workingstudent, Y = employee.$
(*Query 4.*)	$?employee := X.$
(*Answer 4.1.*)	$X = \{bob, dennis, henry, mary, smith\}.$
(*Query 5.*)	$?person := X!, employee := Y!, Z = X - Y.$
(*Answer 5.1.*)	$Z = \{sally, john, jenny, phil\}.$
(*Query 6.*)	$?student := X!, employee := Y!, X \leq Y.$
(*Answer 6.1.*)	$fail.$
(*Query 7.*)	$?ymage := X.$
(*Answer 7.1.*)	$X = \{25, 26, 27, 28, 29, 30\}.$
(*Query 8.*)	$?gender := X.$
(*Answer 8.1.*)	$X =\{$'*Male*', '*Female*'$\}.$

Figure 4.6: Sample Queries and Answers (II)

books which have been borrowed by a particular student called *Phil*. The expected information is who the book's author is and what the price is. Here $\{X\}$ is a set grouping variable and X is bound to an object. The fourth query asks for who are those students taking the *cs*213 course and in which dept they are studying. The answer to this query is obtained via rule 4 in Figure 4.4. The fifth query asks for the information about courses which each student is taking. The sixth query asks for information about *mary*'s family, especially her children and their ages. The answer is obtained via rule 7. The seventh query asks for information about the staff of the *bookstore* dept and their salary. The answer is obtained via rule 6. The eighth query asks for information about each house. The expected information about this house is what is the location of the house and who lives there. Here Y is a set-valued variable which has to be bound to a set. The answer is obtained via rule 1, 2 and 8. Rule 1 finds that *sally* lives with his father *bob* because he is under 20; rule 2 finds that *mary* lives with her spouse *bob*; rule 8 puts the persons living at same address into a set. The ninth asks for information about people who have the same age, 30. The answer is obtained via rule 9. These examples also show the difference between set grouping variables and set-valued variables.

Figure 4.6 shows 8 queries over the meta-information and the corresponding answers based on the sample type system, database, and rules. The first query asks for information about *sally*'s type, attribute labels, and corresponding values. If an attribute value is unknown, an uninstantiated variable is returned. The second query asks for property information about *student* in the type system. Note that *student* is a subtype of *person*, and therefore all properties of *person* are given as answers to the query. The third query asks for the subtype relations in the type system. The fourth query asks for the extensions of *employee*. The fifth query asks for the difference of the extensions of *person* and *employee*, i.e., those objects in *person* not in *employee*. Note $X!$ and $Y!$ are just S-logic notation for not displaying their values which is called projection in relational databases. The sixth query asks for whether or not the extension of *student* is included in the extension of *employee*. The seventh query asks for the extension of *ymage*. The eighth query asks for the extension of *gender*.

Chapter 5

Formal Presentation

S-logic is a logic language for deductive databases. This chapter defines the formal syntax and semantics of S-logic. The syntax is concerned with valid programs admitted by the grammar of S-logic. The semantics is concerned with the meanings attached to the valid programs and the symbols they contain.

5.1 Syntax of S-logic

This section introduces the syntax of S-logic, i.e., its alphabet, types, database, terms, rules and queries.

Definition 5.1 The *alphabet* of S-logic consists of eleven classes of symbols:

(1). the set $\mathcal{A} = \{all, none\}$;
(2). the set $\mathcal{B} = \{integer, string\}$;
(3). a countably infinite set $\mathcal{Z}$ of integers;
(4). a countably infinite set $\mathcal{S}$ of strings;
(5). a countably infinite set $\mathcal{O}$ of symbols called *object identifiers*;
(6). a countably infinite set $\mathcal{C}$ of symbols called *type constructors*;
(7). a countable infinite set $\mathcal{L}$ of symbols called *attributes labels* containing *isa*;
(8). a countable infinite set $\mathcal{F}$ of function symbols, called *object constructors*;
(9). a countably infinite set $\mathcal{V}$ of symbols called *variables*;
(10). logical connective $\Leftarrow$; logical comparatives $=, \leq, \geq, <, >, \neq$; and $\neg$.
(11). comma, dot, {, }, (,), ‘, ’, $\rightarrow$, :, ?, !, -, &, |.

Here the sets $\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, $\mathcal{S}$, $\mathcal{O}$, $\mathcal{C}$, $\mathcal{L}$, $\mathcal{F}$, $\mathcal{V}$ are assumed to be pairwise disjoint.

Definition 5.2 *all* and *none* are types of S-logic called *built-in* types, which represent the biggest type and the smallest type respectively.

Definition 5.3 The *basic types* of S-logic are defined as follows:

- Every element of $\mathcal{B}$ is a basic type, i.e. *integer* and *string* are basic types.
- If t is a type constructor in $\mathcal{C}$, then $t = string(\{a_1, ..., a_n\})$, $(n \geq 1)$ is a basic type and $a_1, ..., a_n \in \mathcal{S}$ are called *objects* of the type t.
- If t is a type constructor in $\mathcal{C}$, then $t = integer(\{lb..rb\})$ is a basic type and $lb, rb \in \mathcal{Z}$ are called the *left bound* and *right bound* of the range of the type t.

Definition 5.4 If $p, p_1, ..., p_n, n \geq 1$ belong to $\mathcal{C}$ or $\mathcal{B}$, $isa, l_m, ..., l_n$ $0 \leq m \leq n$ are labels of $\mathcal{L}$, and we have $p(isa \rightarrow p_1, ..., isa \rightarrow p_m, l_{m+1} \rightarrow p_{m+1}, ..., l_n \rightarrow p_n)$, then p is a type of S-logic called a *record type.* Each $l_i \rightarrow p_i$, $m < i \leq n$ is called a *direct property* of p, each p_i is called a *component type* of p. For each direct property $l_i \rightarrow p_i$ of p, the label l_i is called an *attribute* and is either a set-valued label if p_i is a set type, or a single-valued lable if p_i is not a set type. If $l_{i_1} \rightarrow p_{i_1}, ..., l_{i_{n_i}} \rightarrow p_{i_{n_i}}$, $1 \leq i \leq m$ are properties of p_i, and $l_{i_k} \neq l_j$, $1 \leq k \leq n_i$, $m+1 \leq j \leq n$, then they are also properties of p called *inherited properties*, each $p_i, 1 \leq i \leq m$ is called a *supertype* of p.

According to above definition, a record type inherits all the properties of its supertypes (if any) but may redefine them and may have its own properties.

Definition 5.5 If s is a type of S-logic other than a set type, then $\{s\}$ is a type of S-logic called *set type* and s itself is called a *set element type.*

Here I exclude the possibility of infinite useless set types, such as $\{...\{person\}...\}$. Intuitively, $\{all\}$ is the biggest set type and $\{none\}$ is the smallest set type.

Definition 5.6 The *type system* S of S-logic consists of a finite set of types defined according to the definition 5.2 to 5.5.

Definition 5.7 The *objects* of S-logic are defined as follows:

- Every element of $\mathcal{Z}$, $\mathcal{S}$ and $\mathcal{O}$ is an object, called a *basic object.* Every element of $\mathcal{Z}$ and $\mathcal{S}$ is called a *printable object* and every element of $\mathcal{O}$ is called an *abstract object.*
- If f is a n-ary function symbol from $\mathcal{F}$, $o_1, ..., o_n$ are basic objects, then $f(o_1, ..., o_n)$ is also called a *basic object.*
- If $o_1, ..., o_p$ are basic objects, then $\{o_i, ..., o_p\}$ is called a *set object.* $\{\}$ denotes the empty set object.
- Let $p \in \mathcal{C}$, $l_1, ..., l_n \in \mathcal{L}, n \geq 1$, o be a basic object and $o_1, ..., o_n$ be basic objects or set objects. Then $o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow o_n)$ is called a *record object* of p. Each $o_i, 1 \leq i \leq m$ is called an *attribute value* of l_i of the object o.

Definition 5.8 The *database* of S-logic consists of a finite set of record objects.

Next I will define rules and queries which are based on S-terms.

Definition 5.9 The *variables* of S-logic are defined as follows:

- Every element of $\mathcal{V}$ is called a *variable.* $\mathcal{V}$ is divided into four disjoint kinds: *basic variables* (or *single-valued variables*), *set-valued variables, type variables* and *label variables.*

- If X is a basic variable, then $\{X\}$ is a *set grouping variable.*

Definition 5.10 An *object constructor* is defined inductively as follows:

> If f is an n-ary function symbol from $\mathcal{F}$, $Y_1, ..., Y_n, (n \geq 1)$ are basic variables, basic objects, or object constructors, then $f(Y_1, ..., Y_n)$ is also an object constructor.

Definition 5.11 The *typed S-terms* are defined as follows:

- If P is a type or type variable and S is either a set object or a set-valued variable, then $P := S$ is a typed S-term.
- If $P, P_1, ..., P_n, n \geq 0$ are types or type variables, then $P(isa \rightarrow P_1, ..., isa \rightarrow P_n)$ is a typed S-term.
- If P is a record type or a type variable, $P_1, ..., P_n$ are types or type variables, and $L_1, ..., L_n (n \geq 1)$ are labels (not isa) or label variables, then $P(L_1 \rightarrow P_1, ..., L_n \rightarrow P_n)$ is a typed S-term.
- If P is a type or type variable, $L_1, ..., L_n, (n \geq 0)$ are label variables or labels other than isa and at least one of $P, L_1, ...L_n$ is a variable, and X is either a basic variable, a basic object or an object constructor, $X_1, ..., X_n$ are either basic variables, set-valued variables, set grouping variables, basic objects, set objects, or object constructors, then $X : P(L_1 \rightarrow X_1, ..., L_n \rightarrow X_n)$ is a typed S-term.

The typed S-terms are used only in queries to ask information about the type system. Note that a type in the type system is a closed typed S-term, i.e., without variables.

Following are examples of typed S-terms where X, Y, L are variables.

$person := X.$
$X(isa \rightarrow person).$
$X(L \rightarrow Y).$
$X : employee(L \rightarrow Y).$

Definition 5.12 The *basic S-terms* are defined as follows:

- If p is a type, X is either a basic variable, a set-valued variable, a set grouping variable, a basic object, or an object constructor, then $X : p$ is a basic S-term.
- If p is a type in $\mathcal{C}$, X a basic variable, a basic object, or an object constructor, $X_i, 1 \leq i \leq n$ is either a basic variable, a basic object, an object constructor a set-valued variable, a set grouping variable or a set object, then $X : p(l_1 \rightarrow X_1, ..., l_n \rightarrow X_n)$ is a basic S-term.

For example, $children(X,Y){:}family(l \rightarrow Z)$ and $X{:}student(taking \rightarrow Y)$ are two examples of basic S-terms where X and Y are examples. Note that there are no *isa* labels in basic S-terms.

The basic S-terms are used in both rules and queries. Note that a record object in the database is a closed basic S-term.

Definition 5.13 The *S-terms* consists of all typed S-terms and basic S-terms.

Let $X : p$ be a basic S-term where X is a variable and p is type. If X is not of interest, then $X : p$ can be replaced by $: P$ for convenience. If p is not of interest, then $X : p$ can be replaced by $X :$ for convenience.

Definition 5.14 A *basic literal* is defined as follows:

(1). A basic S-term is a positive basic literal.

(2). $\psi_1 = \psi_2$, $\psi_1 \leq \psi_2$, $\psi_1 \geq \psi_2$, $\psi_1 < \psi_2$, $\psi_1 > \psi_2$, and $\psi_1 \neq \psi_2$ are basic literals, where ψ_1, ψ_2 are basic variables or basic objects. They are also called comparison expressions.

(3). If p and q are basic literals, then the disjunction of p and q denoted as $p; q$ is a disjunctive basic literal.

(4). If p is a positive basic literal, then $\neg p$ is a negative basic literal.

Definition 5.15 A *rule* is an expression of the form $p \Leftarrow p_1, ..., p_n$ where the body $p_1, .., p_n$ is a conjunction of basic literals, the head p is a positive basic literal and all variables in the head must occur in the body.

According to the definition, a rule has no type variables or label variables. Rules are used to derive information about objects and object attributes. Following is an example rule:

$$f(X,Y) : family(children \rightarrow \{Z\}) \Leftarrow$$
$$Z : person(father \rightarrow X, mother \rightarrow Y).$$

Definition 5.16 A *literal* is defined as follows:

(1). An S-term is a positive literal.

(2). $\psi_1 = \psi_2$, $\psi_1 \leq \psi_2$, $\psi_1 \geq \psi_2$, $\psi_1 < \psi_2$, $\psi_1 > \psi_2$ $\psi_1 \neq \psi_2$, are literals, where ψ_1, ψ_2 are either basic variables, basic objects, set variables or set objects.

(3). If p and q are literals, then $p; q$ is a disjunctive literal.

(4). If p is a positive literal, $\neg p$ is a negative literal.

The basic literals are subset of literals, the later allows type variables, label variables, set variables or set objects occur.

Definition 5.17 A *query* is a conjunction of literals starting with the question mark $?p_1, ..., p_n$.

A query is used to ask information about objects, object attributes and the type system.

Definition 5.18 A *program* P is a triple $P = \langle S, DB, R \rangle$.

(1). S is a type system,

(2). DB is a database,

(3). R is a finite collection of rules.

5.2 Semantics

Definition 5.19 Let L be a set, a *binary relation* R on L is a subset of the cartesian product $L \times L$.

Unless specified otherwise, all relations will implicitly considered to be binary from now on. The notation xRy stands for $\langle x, y \rangle \in R$.

Definition 5.20 A relation R on a set L is

- *reflexive* iff xRx for all $x \in L$.
- *symmetric* iff xRy implies yRx, for all $x, y \in L$.
- *antisymmetric* iff xRy and yRx imply $x = y$, for all $x, y \in L$.

- *transitive* iff xRy and yRz imply xRz, for all $x, y, z \in L$.

Definition 5.21 A relation on a set L is a *partial order* if it is reflexive, antisymmetric and transitive.

Definition 5.22 Let S be a set with a partial order $\leq$. Then $a \in S$ is an *upper bound* of a subset X of S if $x \leq a$, for all $x \in X$. Similarly, $b \in S$ is a *lower bound* of X if $b \leq x$, for all $x \in X$.

Definition 5.23 Let S be a set with a partial order $\leq$. Then $a \in S$ is the *join* or *least upper bound* (abbreviated *lub*) of s subset X of S if a is an upper bound of X and, for all upper bounds a' of X, we have $a \leq a'$. Similarly, $b \in S$ is the *meet* or *greatest lower bound* (abbreviated *glb*) of s subset X of S if b is a lower bound of X and, for all lower bounds b' of X, we have $b' \leq b$.

Definition 5.24 A partial ordered set L is a *meet-semilattice* iff for every subset X of L, there is a *glb*.

Definition 5.25 A partial ordered set L is a *join-semilattice* iff for every subset X of L, there is a *lub*.

Definition 5.26 A partial ordered set L is a *lattice* iff it is both a meet-semi-lattice and a join-semi-lattice.

Definition 5.27 Given a program P of S-logic, its *interpretation* I is a tuple $\langle U, \Sigma, \Gamma, \pi, \sigma . g_C, g_L, g_O, g_F \rangle$. [1]

[1]This definition and some of the following definitions are based on [26]

(1). U is a universe of all objects.

(2). Σ is a finite set of semantic types of which each denotes a class of objects of U under the mapping π.

(3). Γ is a finite set of semantic labels, which is divided into three kinds: Γ_+, Γ_*, Γ_{isa}.

a. Γ_+ is the set of single-valued labels.

b. Γ_* is the set of set-valued labels.

c. Γ_{isa} is a singleton set $\{\delta_{isa}\}$.

(4). π interprets each semantic type as a subset of U, i.e., $\pi : \Sigma \rightarrow 2^U$.

(5). σ interprets each semantic label as a partial mapping as follows:

a. for each $f \in \Gamma_+$, $\sigma(f)$ is a single-valued mapping $U \rightarrow U$,

b. for each $f \in \Gamma_*$, $\sigma(f)$ is a set-valued mapping $U \rightarrow 2^U$, and

c. $\sigma(\delta_{isa})$ is an identity mapping $U \rightarrow U$, i.e., for each $u \in U$, $\sigma(\delta_{isa})(u) = u$.

(6). g_C is a homomorphic function which interprets each syntactic type in $\mathcal{C}$ as a semantic type in Σ, i.e., $g_C : \mathcal{C} \rightarrow \Sigma$.

(7). g_L is a homomorphic function which interprets each syntactic label in $\mathcal{L}$ as a semantic label in Γ, i.e., $g_L : \mathcal{L} \rightarrow \Gamma$, especially, $g_L : isa \rightarrow \delta_{isa}$.

(8). g_O is a homomorphic function $g_O : \mathcal{O} \cup \mathcal{S} \cup \mathcal{Z} \rightarrow U$.

(9). g_F is a function which interprets each k-ary object constructor as a mapping $U^k \rightarrow U$.

An interpretation I gives a denotational semantics to a program. It maps every syntactic object to a semantic object by the mapping g_O; every syntactic type to a semantic type by the mapping g_C; every syntactic label to a semantic label by the mapping g_L; every syntactic function to a semantic function by the mapping g_F. It interprets every semantic type as a class of objects by the mapping π; every semantic label as a function by the mapping σ.

Definition 5.28 Given an interpretation I, the intended semantics of types is given by $\pi \circ g_C$ as follows:

(1). For the basic types, $\pi(g_C(integer)) = \mathbf{Z} \subset U$; $\pi(g_C(string)) = \mathbf{S} \subset U$;
if $s = string(\{a_1, ..., a_n\})$, then $\pi(g_C(s)) = \{g_O(a_1), ..., g_O(a_n)\} \subset \mathbf{S}$;
if $s = integer(\{lb..rb\})$, then $\pi(g_C(s)) = \{x : g_O(lb) \leq x \leq g_O(rb)\} \subset \mathbf{Z}$.

(2). For a set type {s}, $\pi(g_C(\{s\})) = 2^{\pi(g_C(s))} \subseteq 2^U$.

(3). For a record type with definition $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$,
$\pi(g_C(p)) = \{x : g_L(l_i) \in \Gamma, \sigma(g_L(l_i))(x) \in \pi(g_C(p_i)), 1 \leq i \leq n, \}$.

(4). For built-in types, $\pi(g_C(all)) = U$; $\pi(g_C(none)) = \{\}$.

Note that $\pi \circ g_C$ determines the extensions of types in S-logic.

Definition 5.29 Two types p and q have a *subtype relation* based on some interpretation I denoted by $p \leq q$ if $\pi(g_C(p)) \subseteq \pi(g_C(q))$.

So two types have subtype relation if their extensions have subset relation. Immediately, we have following theorems.

Theorem 5.1 Let S be a type system of a program P. Then the subtype relation is a partial order on S.□

Theorem 5.2 Let S be a type system of a program P.

(1). If $s = integer(\{lb..rb\}) \in S$, then $s \leq integer$.

(2). If $t = string(\{a_1, ..., a_n\}) \in S$, then $s \leq string$.

(3). If $p \leq q$, $p, q \in S$, then $\{p\} \leq \{q\}$.

(4). $none \leq p$ for all $p \in S$.

(5). $p \leq all$ for all $p \in S$.

Proof: (1) (2) (4) (5) are trivial. For (3), since $p \leq q$, we have $\pi(g_C(p)) \subseteq \pi(g_C(q))$. For every $x \subseteq \pi(g_C(p))$, we have $x \subseteq \pi(g_C(q))$. So we have $2^{\pi(g_C(q))} \subseteq 2^{\pi(g_C(q))}$, i.e., $\{p\} \leq \{q\}$. □

Record types have the following properties.

Theorem 5.3 Let p be a record type with $p(isa \rightarrow p_1, ..., isa \rightarrow p_m, l_{m+1} \rightarrow p_{m+1}, ..., l_n \rightarrow p_n), 1 \leq m \leq n$, Then $p \leq p_i, 1 \leq i \leq m$, i.e., p is a lower bound of $p_1, ..., p_m$ under the relation $\leq$, and $\pi(g_C(p)) \subseteq \cap_{j=1}^{j=m} \pi(g_C(p_j))$.

Proof: we have that for each $x \in \pi(g_C(p))$, $\sigma(g_L(isa))(x) \in \pi(g_C(p_i)), 1 \leq i \leq m$ by the definition. Since $\sigma(g_L(isa))(x) = \sigma(\delta_{isa})(x) = x \in \pi(g_C(p_i))$, so $\pi(g_C(p)) \subseteq \pi(g_C(p_i))$. Therefore we have $p \leq p_i, 1 \leq i \leq m$ and $\pi(g_C(p)) \subseteq \cap_{j=1}^{j=m} \pi(g_C(p_j))$. □

This theorem says that the extension of p is a subset of the intersection of the extensions of $p_1, ..., p_m$.

Theorem 5.4 If $p_1(..., isa \rightarrow p, ...), ..., p_m(..., isa \rightarrow p, ...)$ are m record types in the type system S. Then $p_i \leq p, 1 \leq i \leq m$, i.e., p is a upper bound of $p_1, ..., p_m$ under the relation $\leq$, and $\cap_{j=1}^{j=m} \pi(g_C(p_j)) \subseteq \pi(g_C(p))$.

Proof: Since $\pi(g_C(p_i)) \subseteq \pi(g_C(p))$, so $p_i \leq p, 1 \leq i \leq m$. Therefore $\pi(g_C(p)) \supseteq \cap_{j=1}^{j=m} \pi(g_C(p_i)), 1 \leq i \leq m$. □

This theorem says that the extension of p is a superset of the union of the extensions of $p_1, ..., p_m$, or p is an upper bound of $p_1, ..., p_m$. For example, *person* is an upper bound of *student* and *employee* in Figure 4.1.

Theorem 5.5 Let p be a record type with $p(isa \rightarrow p_1, ..., isa \rightarrow p_m, l_{m+1} \rightarrow p_{m+1}, ..., l_n \rightarrow p_n), 1 \leq m \leq n$, $l_s \rightarrow p_s$ be a redefined property of p, $m < s \leq n$. and $p_{s_1}, ..., p_{s_k}, 1 \leq s_1 \leq ... \leq s_k \leq m$ have properties $l_s \rightarrow p_{t_1}, ... l_s \rightarrow p_{t_k}$ respectively. Then $\sigma(g_L(l_s))(\pi(g_C(p))) \subseteq \cap_{j=t_1}^{j=t_k} \pi(g_C(p_j))$.

Proof: We have that

$\pi(g_C(p)) \subseteq \pi(g_C(p_{s_i})), 1 \leq i \leq k$ and

$\sigma(g_L(l_s))(\pi(g_C(p_{s_i}))) \subseteq \pi(g_C(p_{t_i})), 1 \leq i \leq k$.

$\sigma(g_L(l_s))(\pi(g_C(p))) \subseteq \pi(g_C(p_{t_i})), 1 \leq i \leq k$.

Therefore $\sigma(g_L(l_s))(\pi(g_C(p))) \subseteq \cap_{j=t_1}^{j=t_k} \pi(g_C(p_j))$.□

Not all redefinitions are meaningful. The following theorem shows us the reason.

Theorem 5.6 Let p be a record type with $p(isa \rightarrow p_1, ..., isa \rightarrow p_m, l_{m+1} \rightarrow p_{m+1}, ..., l_n \rightarrow p_n), 1 \leq m \leq n$, $l_s \rightarrow p_s$ be a redefined property of p, $m + 1 \leq s \leq n$. and $p_{s_1}, ..., p_{s_k}, 1 \leq s_1 \leq ... \leq s_k \leq m$ have properties $l_s \rightarrow p_{t_1}, ... l_s \rightarrow p_{t_k}$ respectively. Then $\pi(g_C(p_s)) \subseteq \cap_{j=t_1}^{j=t_k} \pi(g_C(p_j))$.

Proof: Suppose not $\pi(g_C(p_s)) \cap \bigcap_{j=t_1}^{t_k} \pi(g_C(p_j)) \subseteq \bigcap_{j=t_1}^{t_k} \pi(g_C(p_j))$, then there may exist an x, $x \in \pi(g_C(p))$ such that $\sigma(g_L(l_s))(x) = y \in \pi(g_C(p_s)) \setminus \bigcap_{j=t_1}^{t_k} \pi(g_C(p_j))$. So we have $x \in \pi(g_L(p))$ but not $x \in \pi(g_L(p_i))$, $1 \leq i \leq m$, which contradicts our intended semantics (by theorem 5.3). Therefore, we have $\pi(g_C(p_s)) \cap \bigcap_{j=t_1}^{t_k} \pi(g_C(p_j)) \subseteq \bigcap_{j=t_1}^{t_k} \pi(g_C(p_j))$, which implies $\pi(g_C(p_s)) \subseteq \bigcap_{j=t_1}^{j=t_k} \pi(g_C(p_j))$.□

For example, look at the record type *workingstudent* in Figure 4.1. It is a subtype of *student* and *employee* of which both have property *age* $\rightarrow$ *young* and *age* $\rightarrow$ *midage* respectively. *workingstudent* redefines the property to be *age* $\rightarrow$ *ymage* where $\pi(g_C(ymage)) = \pi(g_C(young) \cap \pi(g_C(midage))$. Besides, *employee* has a property *salary* $\rightarrow$ *employeesalary*, *workingstudent* redefines this property to be *salary* $\rightarrow$ *support* and $\pi(g_C(support) \subset \pi(g_C(employeesalary))$. Therefore this record type is meaningful.

Definition 5.30 A record type is *acceptable* if it satisfies the theorem 5.6. A type system is *acceptable* if each record type in it is acceptable.

Theorem 5.7 If that all types except set types in a type system S form a lattice under the subtype relation and for each record type p with $p(isa \rightarrow p_1, ..., isa \rightarrow p_m, l_{m+1} \rightarrow p_{m+1}, ..., l_n \rightarrow p_n), 1 \leq m \leq n$, where $l_s \rightarrow p_s$ is a redefined property of p, $m < s \leq n$ and $p_{s_1}, ..., p_{s_k}, 1 \leq s_1 \leq ... \leq s_k \leq m$ have properties $l_s \rightarrow p_{t_1}, ... l_s \rightarrow p_{t_k}$ respectively, and p_s is a greatest lower bound of $p_{t_1}, ..., p_{t_k}$ in the lattice, then the type system S is acceptable. □

This theorem shows how to syntactically check whether a type system is meaningful or not. Clearly, the sample type system in Chapter 4 is acceptable. In the following discussions, all type systems of programs are assumed to be acceptable.

In the above discussion, we exclude set types. But directly from theorem 5.2(3), we have

Theorem 5.8 If all types except set types in a type system S form a lattice under the subtype relation $\leq$ with *all* as the biggest element and *none* as the smallest element, then all set types also form a lattice with $\{all\}$ as the biggest element and $\{none\}$ as least element. $\Box$

Definition 5.31 Given an interpretation I, the intended semantics of objects is as follows:

(1). For each basic object $o \in \mathcal{S} \cup \mathcal{Z} \cup \mathcal{O}$, its intended semantics is given by an unique element $u \in U$ such that $g_O(o) = u$.

(2). For each basic object $f(o_1, ..., o_n)$, its intended semantics is given by a unique element $u \in U$ such that $g_O(f(o_1, ..., o_n) = u$.

(3). For each set object $\{o_1, ..., o_p\}$, its intended semantics is given by a subset of U which is $\{g_O(o_1), ..., g_O(o_p)\} \subset U$. We use $g_O(\{o_1, ..., o_p\})$ to stand for $\{g_O(o_1), ..., g_O(o_p)\}$.

(4). For each record object in the database,

$o : p(l_1 \rightarrow o_1, ..., l_{m-1} \rightarrow o_{m-1}, l_m \rightarrow \{o_{m,1}, ..., o_{m,k_m}\}, ..., l_n \rightarrow \{o_{n,1}, ..., o_{n,k_n}\})$,

the intended semantics is

(a.) $g_O(o) \in \pi(g_C(p))$;

(b.) for each single-valued lable $l_i, \sigma(g_L(l_i)) \in \Gamma_+$, $\sigma(g_L(l_i))(g_O(o)) = g_O(o_i)$, $1 \leq i < m$;

(c.) for each set-valued label l_j,

$$g_L(l_j) \in \Gamma_*,\ g_O(\{o_{j,1},...,o_{j,k_j}\}) \subseteq \sigma(g_L(l_j))(g_O(o)),\ m \leq j \leq n;$$

Definition 5.32 Let $P = \langle S, DB, R\rangle$ be a program and I be an interpretation. A record object in the database DB, $o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow o_n)$ is *well-typed* if there exist a record type p with $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$ in the type system such that $g_O(o_i) \in \pi(g_C(p_i))$, $1 \leq i \leq n$. The database DB is *well-typed* if every record object of it is well-typed.

Unless specified otherwise, all DBs are assumed to be well-typed.

Definition 5.33 A *variable assignment*, ν, is a ground substitution which assigns an element in U to a basic variable, a subset of U to a set-valued variable, a type in Σ to a type variable, and a label in Γ other than Γ_{isa} to a label variable. Besides, it is extended to non-variable elements as follows:

(1). if $o \in \mathcal{O} \cup \mathcal{S} \cup \mathcal{Z}$, then $\nu(o) = g_O(d)$;

(2). if $l \in \mathcal{L}$, then $\nu(l) = g_L(l)$;

(3). if $c \in \mathcal{C}$, then $\nu(c) = g_C(c)$;

(4). if $f \in F$, then $\nu(f) = g_F(f)$;

(5). $\nu(f(..., X, ...)) = g_F(f)(..., \nu(X), ...)$;

(6). $\nu(\{o_1, ..., o_n\} = \{g_O(o_1), ..., g_O(o_n)\}$;

(7). if t is an S-term, then $\nu(t)$ is still an S-term resulting from t by applying ν to every object, label, type, variable, and object constructor of t.

Now I define the notion of well-typed basic S-terms, satisfaction of S-terms, rules and programs.

Definition 5.34 Given an interpretation I and a variable assignment ν, *well-typed* basic S-terms are defined as follows:

(1). For a basic S-term $X : p$, it is well-typed iff $\nu(X) \in \pi(g_C(p))$.

(2). For a basic S-term $t = X : p(l_1 \rightarrow X_1, ..., l_n \rightarrow X_n)$, it is well-typed iff there is a record type p with $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$ in the type system such that $\nu(X) \in \pi(g_C(p)), \nu(X_i) \in \pi(\nu(p_i)), (1 \leq i \leq n)$.

Definition 5.35 Given an interpretation I and a variable assignment ν, the satisfaction of an S-term ψ by I and ν, denoted by $\models_I \nu(\psi)$, is defined as follows:

(1). For a basic S-term $X : p$, $\models_I \nu(X : p)$ iff $X : p$ is well-typed.

(2). For a basic S-term $t = X : p(l_1 \rightarrow X_1, ..., l_n \rightarrow X_n)$, $\models_I \nu(t)$ iff

(a.) t is well-typed;

(b.) $\sigma(g_L(l_i))(\nu(X)) = \nu(X_i), 1 \leq i \leq n$, if $g_L(l_i) \in \Gamma_+$;
$\sigma(g_L(l_i))(\nu(X)) \supseteq \nu(X_i), 1 \leq i \leq n$, if $g_L(l_i) \in \Gamma_*$;

(c.) if X_i is a set grouping variable and $X_i = \{Y\}$, then $\nu(Y) \in g_L(l_i)\nu(X)$, $(1 \leq i \leq n)$.

(3). For a typed S-term $P := S$, $\models_I \nu(P := S)$ iff $\nu(P) \in \Sigma$, $\pi(\nu(P)) = \nu(S)$.

(4). For a typed S-term $t = P(isa \rightarrow P_1, ..., isa \rightarrow P_n)$, $\models_I \nu(t)$ iff $\nu(P), \nu(P_1), ..., \nu(P_n) \in \Sigma$, and $\pi(\nu(P)) \subseteq \pi(\nu(P_i)), 1 \leq i \leq n$.

(5). For a typed S-term $t = P(L_1 \to P_1, ..., L_n \to P_n)$, $\models_I \nu(t)$ iff $\nu(P), \nu(P_1), ..., \nu(P_n) \in \Sigma$, $\nu(L_i) \in \Gamma - \{\delta_{isa}\}$, and $\sigma(\nu(L_i))(\pi(\nu(P))) \subseteq \pi(\nu(P_i))$, $1 \leq i \leq n$.

(6). For a typed S-term $t = X : P(L_1 \to X_1, ..., L_n \to X_n)$, $\models_I \nu(t)$ iff $\models_I X : \nu(P)(\nu(L_1) \to X_1, ..., \nu(L_n) \to X_n))$.

Definition 5.36 Given an interpretation I and a variable assignment ν, the satisfaction of literals other than S-terms are defined as follows:

(1). $\models_I \nu(\psi_1 = \psi_2)$ iff $\nu(\psi_1) = \nu(\psi_2)$.

(2). $\models_I \nu(\psi_1 \neq \psi_2)$ iff $\nu(\psi_1) \neq \nu(\psi_2)$.

(3). $\models_I \nu(\psi_1 \leq \psi_2)$ iff $\nu(\psi_1) \leq \nu(\psi_2)$, if $\nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$, or $\nu(\psi_1) \subseteq \nu(\psi_2)$ if $\nu(\psi_1), \nu(\psi_2) \subset U$.

(4). $\models_I \nu(\psi_1 \geq \psi_2)$ iff $\nu(\psi_1) \geq \nu(\psi_2)$, if $\nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$, or $\nu(\psi_1) \supseteq \nu(\psi_2)$ if $\nu(\psi_1), \nu(\psi_2) \subset U$.

(5). $\models_I \nu(\psi_1 < \psi_2)$ iff $\nu(\psi_1) < \nu(\psi_2), \nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$, or $\nu(\psi_1) \subset \nu(\psi_2)$ if $\nu(\psi_1), \nu(\psi_2) \subset U$.

(6). $\models_I \nu(\psi_1 > \psi_2)$ iff $\nu(\psi_1) > \nu(\psi_2), \nu(\psi_1), \nu(\psi_2) \in \mathbf{Z}$, or $\nu(\psi_1) \supset \nu(\psi_2)$ if $\nu(\psi_1), \nu(\psi_2) \subset U$.

(7). $\models_I \nu(p; q)$ iff $\models_I \nu(p)$ or $\models_I \nu(q)$

(8). If ψ is an S-term, $\models_I \nu(\neg\psi)$, iff not $\models_I \nu(\psi)$.

Clearly, for a ground S-term ψ, i.e, an S-term without variable, its satisfaction is independent of a variable assignment, and it can be simply written as $\models_I \psi$.

Theorem 5.9 Let $P = \langle S, DB, R \rangle$ be a program and DB be well-typed. Then for each type p in the type system and each record object r in the database, we have $\models_I p$ and $\models_I r$. $\Box$

Definition 5.37 Let I be an interpretation $r = p \Leftarrow p_1, ..., p_n$ a rule, $\models_I r$ iff for each variable assignment ν, if $\models_I \nu(p_i)$ for each $p_i, 1 \leq i \leq n$, then $\models_I \nu(p)$, or for some variable assignment ν, not $\models_I \nu(p_i)$ for some $p_i, 1 \leq i \leq n$.

Note here that the treatment of attribute values of the set-valued labels of a record object or a basic S-term is different to those of the single-valued labels. Look at following example.

$$phil : student(name \rightarrow \text{'}Phil\text{'}, taking \rightarrow \{cs213, cs311\}).$$
$$jenny : student(taking \rightarrow \{X\}) \Leftarrow phil : student(taking \rightarrow \{X\}).$$

Here, the first record object says that *phil*'s name is exactly *'Phil'* and can not be anything else. But the courses which *phil* takes include *cs*213 and *cs*311 but is not restricted to them. There may be other courses. The rule says that all courses which *phil* takes are also taken by *jenny*, i.e., all courses which *phil* takes are included in the courses which *jenny* takes, but not the only courses which *jenny* takes. This representation of the attribute values of set-valued labels is natural.

If we want to represent that all courses which *phil* takes are only $\{cs213, cs311\}$ and all courses which *jenny* takes are only those which *phil* takes, i.e., the set in a record object or a basic S-term is exactly the attribute value of some set-valued label, then we have to introduce stratification on sets as LDL [4] does. In LDL, a program is stratified if we are able to label the predicate symbols of the program with non-negative integers such that for every rule $p(...) \leftarrow L_1, ..., L_n$, the label of a predicate symbol appearing in the body within a negative literal or a literal containing set terms is less than the label of p, and the labels of the other predicate symbols in the body are less than or equal to the label of p.

Also note that there is a major difference between S-logic and Horn-clause logic in their satisfaction of a rule. In Horn-clause logic, a rule is always satisfied by all interpretations, while in S-logic, a rule may not be satisfied by any interpretation. Two reasons for a rule in S-logic not to be satisfied based on the above definition. One is that the head of the rule which is a basic S-term is not well-typed. The other is that the head of the rule contains variables which do not occur in the body of the rule, or occur in the comparison expressions in the body. For example, suppose we have a rule $f(X) : p(l_1 \rightarrow X, l_2 \rightarrow Y) \Leftarrow X : q$, where l_1 and l_2 are single-valued labels. This rule has a variable Y in the head but not in the body. For each variable assignment, we can assign Y a different value but assign X to the same one. So the head of the rule is not satisfiable but the body is. Therefore this rule is not satisfiable by any interpretation. Similarly, rules like

$$f(X) : p(l_1 \rightarrow X, l_2 \rightarrow Y) \Leftarrow X : q, Y = Z,$$
$$f(X) : p(l_1 \rightarrow X, l_2 \rightarrow Y) \Leftarrow X : q, Y > 5,$$
$$f(X) : p(l_1 \rightarrow X, l_2 \rightarrow Y) \Leftarrow X : q, Y \neq 5, \text{ etc.}$$

are also not satisfiable.

Clearly, we can have a syntactic restriction on rules to guarantee their satisfiability.

Theorem 5.10 A rule is satisfiable if all of its variables in the head also occur in the body rather than comparison expressions.

Since all variables in the head of a rule occur in the body, the type information is easy to obtain. Therefore it is straightforward to check whether the head is well-typed or not.

From here on, we assume that all rules are satisfiable.

Definition 5.38 Let I be an interpretation and T be a set of S-terms or rules, $\models_I T$ iff $\models_I T_i, T_i \in T$.

Theorem 5.11 Let $P = \langle S, DB, R \rangle$ be any program and DB well-typed. Then we have $\models_I S$ and $\models_I DB$. □

In this thesis, I will only consider definite programs [29], i.e., the body of a rule has no negative basic literals. Programs with negative basic literals in the body of rules are quite complicated and will be explored later. Unless specified otherwise, all programs will be implicitly considered as definite programs from now on.

Definition 5.39 Let I be an interpretation and $P = \langle S, DB, R \rangle$ a program, $\models_I P$ iff $\models_I S \cup DB \cup R$.

Based on theorem 5.11, we have $\models_I P$ iff $\models_I R$.

Definition 5.40 A *model* M of a program P is an interpretation such that $\models_M P$.

Definition 5.41 Let P be a program and F be a ground S-term. We say F is a *logical consequence* of P written as $P \models F$, if for every interpretation I of P, $\models_I P$ implies that $\models_I F$.

It is impossible to prove $P \models F$ by proving that for every interpretation $I \models_I P$ implies $\models_I F$. The question can be changed to another one which is possible.

Definition 5.42 Let P be a program, we say P is *unsatisfiable* if no interpretation of P is a model.

Theorem 5.12 Let P be a program and F be a ground S-term. Then F is a logical consequence of P iff $P \cup \{\neg F\}$ is unsatisfiable. [2]

Proof: Suppose that F is a logical consequence of S. Let I be an interpretation of P and suppose I is a model for P. Then I is a model for F. Hence I is not a model for $P \cup \{\neg F\}$. Thus $P \cup \{\neg F\}$ is unsatisfiable.

Conversely, suppose $P \cup \{\neg F\}$ is unsatisfiable. Let I be any interpretation of L. Suppose I is a model for P. Since $P \cup \{\neg F\}$ is unsatisfiable, I can not be a model for $\neg F$. Thus I is a model for F and so F is a logical consequence of P. □

Definition 5.43 Given a program $P = \langle S, DB, R\rangle$ and a query Q, an *answer* to the query Q is a variable assignment ν for all variables of Q such that $P \models \nu(Q)$.

Applying these definitions to programs, we see that when we give a goal Q to the system, with program P loaded, we are asking the system to show that $P \cup \{\neg Q\}$ is unsatisfiable. Theorem 5.7 states that showing $P \cup \{\neg Q\}$ is unsatisfiable is exactly the same as showing that there exists ν such that $\nu(Q)$ is a logical consequence of P.

To prove that $P \models Q$ where P is a program and Q is a query, the basic problem is that of determining the unsatisfiability, or otherwise, of $P \cup \{\neg Q\}$. According to the definition, this implies showing *every* interpretation of $P \cup \{\neg Q\}$ is not a model. Needless to say, this still seems to be a formidable problem. However, like

[2]The whole theory from here on is based on [29]

first-order logic, it turns out that there is a much smaller and more convenient class of interpretations, which are all that need to be investigated to show unsatisfiability. These are the so-called Herbrand interpretations.

5.3 Herbrand Interpretations

Definition 5.44 An interpretation $H = \langle U, \Sigma, \Gamma, \pi, \sigma, g_C, g_L, g_O, g_F \rangle$ is a *Herbrand interpretation* iff the following conditions hold:

(1). $U = U_* \cup 2^{U_*}$.

$U_* = \cup_{i=1}^{\infty} U_i$.

$U_i = U_{i-1} \cup \{f(o_1, ..., o_k) : f$ is a functor of arity k, and $o_j \in U_{i-1}, 1 \leq j \leq k\}$

$U_0 = \mathcal{S} \cup \mathcal{Z} \cup \mathcal{O}$,

(2). $\Sigma = \mathcal{C}$.

(3). $\Gamma = \mathcal{L}$.

(4). $g_C(p) = p$, for every $p \in \mathcal{C}$.

(5). $g_L(l) = l$, for every $l \in \mathcal{L}$.

(6). $g_O(o) = o$, for every $o \in \mathcal{Z} \cup \mathcal{S} \cup \mathcal{O}$.

(7). $g_F(f) = f$, for every $f \in \mathcal{F}$.

The domains of different Herbrand interpretations are the same, which are $U \cup \Sigma \cup \Gamma$. Besides, types, labels objects and and object constructors are interpreted as themselves in Herbrand interpretations. Only the extensions of types and the mappings of labels may be interpreted differently. So we can just represent an interpretation by listing all the extensions of types and all mappings of labels.

Example 1: Suppose we have following program.

(a). Type System

$p(f \rightarrow integer).$
$q(s \rightarrow \{integer\}).$

(b). Database

$o_1 : p(f \rightarrow 1).$
$o_2 : p(f \rightarrow 2).$

(c). Rules

$o : q(s \rightarrow \{X\}) \Leftarrow P : p(f \rightarrow X).$

Clearly the database and rules are well-typed. An interpretation for this program is

$$I = \{\pi(p) = \{o_1, o_2, o_3\}, \sigma(f)(o_1) = 1, \sigma(f)(o_2) = 2, \sigma(f)(o_3) = 3,$$
$$\pi(q) = \{o\},\ \sigma(s)(o) \supseteq \{1,2,3\}\}$$

It is more intuitive to represent the interpretation in the following way:

$$I = \{o_1 : p(f \rightarrow 1), o_2 : p(f \rightarrow 2), o_3 : p(f \rightarrow 3), o : q(s \rightarrow \{1,2,3\})\}.$$

Later on, interpretations will be represented by listing all the extensions of types and all mappings of labels in the record object form.

Definition 5.45 Given a program P, a *Herbrand model* is a Herbrand interpretation which is a model for P.

For example 1, the interpretation I is obviously a model for the given program.

Theorem 5.13 Let P be a program and suppose p has a model. Then P has a Herbrand model.

Proof: Let I be an interpretation of P. We define a Herbrand interpretation I' as follows:

$I' = I'_1 \cup I'_2$.

$I'_1 = \{s : s = p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n) \text{ and } \models_I s\}$

$I'_2 = \{t : t = o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow \{o_{n,1}, ..., o_{n,k_n}\}) \text{ and } \models_I t\}$

It is straightforward to show that if I is a model, then I' is also a model. □

Theorem 5.14 (Herbrand Theorem) Let P be a program. Then P is unsatisfiable if P has no Herbrand models.

Proof: If P is satisfiable, then the above theorem shows that it has a Herbrand model. □

To prove that $P \models Q$ where P is a program and Q is a query, the basic problem has now changed to prove that $P \cup \{\neg Q\}$ has no Herbrand models. We will see that we only need to consider a special Herbrand model which is the least Herbrand model. This model is precisely the set of record objects plus the type system. We will also obtain an important fixpoint characterisation of the least Herbrand model.

Unless specified otherwise, all interpretations from here on will be implicitly considered as Herbrand interpretations and all models as Herbrand models.

Example 2: Look at the following program of which the type system is omited.

$f(P, H) : q(s \rightarrow X) \Leftarrow P : p(s \rightarrow X), H : h(s \rightarrow X).$
$p_1 : p(s \rightarrow \{X\}) \Leftarrow R : r(f \rightarrow X).$
$r_1 : r(f \rightarrow 1).$
$h_1 : h(s \rightarrow \{1\}).$

Possible models for the program are:

$M_1 = \{r_1 : r(f \rightarrow 1), h_1 : h(s \rightarrow \{1\}), p_1 : p(s \rightarrow \{1\}), f(p_1, h_1) : q(s \rightarrow \{1\})\}.$
$M_2 = \{r_1 : r(f \rightarrow 1), r_2 : r(f \rightarrow 2), h_1 : h(s \rightarrow \{1\}), p_1 : p(s \rightarrow \{1, 2\}),$
$f(p_1, h_1) : q(s \rightarrow \{1\})\}.$
$M_3 = \{r_1 : r(f \rightarrow 1), r_2 : r(f \rightarrow 3), h_1 : h(s \rightarrow \{1\}), p_1 : p(s \rightarrow \{1, 3\}),$
$f(p_1, h_1) : q(s \rightarrow \{1\})\}.$

Definition 5.46 Let P be a program, $I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_C, g_L, g_O, g_F \rangle$ and $I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_C, g_L, g_O, g_F \rangle$ be two models of P. Then I_1 is a *sub-interpretation* of I_2, denoted by $I_1 \subseteq I_2$ iff the following conditions hold:

(1). $\pi_1(p) \subseteq \pi_2(p)$, for every $p \in \Sigma$.

(2). if $\sigma_1(l)$ is defined on $o \in U$ then $\sigma_1(l)(o) = \sigma_2(l)(o)$, for every single-valued label $l \in \Gamma_+$.

(3). if $\sigma_1(l)$ is defined on $o \in U$ then $\sigma_1(l)(o) \subseteq \sigma_2(l)(o)$, for every set-valued label $l \in \Gamma_*$.

Clearly, $M_1 \subseteq M_2$, $M_1 \subseteq M_3$ but not $M_2 \subseteq M_3$ or $M_3 \subseteq M_2$ for Example 2. Immediately, we have the following theorem.

Theorem 5.15 The sub-interpretation relation over all possible interpretations of a given program is a partial order. □

Definition 5.47 Let P be a program and $I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_C, g_L, g_O, g_F \rangle$ and $I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_C, g_L, g_O, g_F \rangle$ be two interpretations of P. The *intersection* $I = \langle U, \Sigma, \Gamma, \pi, \sigma, g_C, g_L, g_O, g_F \rangle$, of I_1 and I_2, denoted by $I = I_1 \cap I_2$, is defined as follows:

(1). $\pi(p) \subseteq \pi_1(p) \cap \pi_2(p)$, for every $p \in \Sigma$.

(2). $\sigma(l)$ is defined on $o \in U$ if both $\sigma_1(l)$ and $\sigma_2(l)$ is defined on o, and $\sigma_1(l)(o) = \sigma_2(l)(o), o \in \pi_1(p)$, $o \in \pi_2(p)$, for some $p \in \Sigma$, then $\sigma(l)(o) = \sigma_1(l)(o)$ and $o \in \pi(p)$ for every single-valued label $l \in \Gamma_+$.

(3). $\sigma(l)$ is defined on $o \in U$ if both $\sigma_1(l)$ and $\sigma_2(l)$ is defined on o, and $o \in \pi_1(\{p\})$ and $o \in \pi_2(\{p\})$ for some $p \in \Sigma$, then $\sigma(l)(o) = \sigma_1(l)(o) \cap \sigma_2(l)(o)$ and $o \in \pi(\{p\})$, for every set-valued label $l \in \Gamma_*$.

The intersection of interpretations has the following properties.

Theorem 5.16 The relation $\cap$ over interpretations of a given program is commutative and idempotent i.e., $I_1 \cap I_2 = I_2 \cap I_1$ and $I_1 \cap I_1 = I_1$ for any two interpretations I_1 and I_2. □

Theorem 5.17 The relation $\cap$ over interpretations of a given program is associative, i.e., $I_1 \cap (I_2 \cap I_3) = (I_1 \cap I_2) \cap I_3$ for any three interpretations I_1, I_2 and I_3.

Proof: Let

$I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_C, g_L, g_O, g_F \rangle$,

$I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_C, g_L, g_O, g_F \rangle$,

$I_3 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_C, g_L, g_O, g_F \rangle$,

$I_{23} = I_2 \cap I_3 = \langle U, \Sigma, \Gamma, \pi_{23}, \sigma_{23}, g_C, g_L, g_O, g_F \rangle$,

$I_{12} = I_1 \cap I_2 = \langle U, \Sigma, \Gamma, \pi_{12}, \sigma_{12}, g_C, g_L, g_O, g_F \rangle$,

$I_{123} = I_1 \cap I_{23} = \langle U, \Sigma, \Gamma, \pi_{123}, \sigma_{123}, g_C, g_L, g_O, g_F \rangle$,

$I'_{123} = I_{12} \cap I_3 = \langle U, \Sigma, \Gamma, \pi'_{123}, \sigma'_{123}, g_C, g_L, g_O, g_F \rangle$,

Now we prove that $I_{123} = I'_{123}$.

For I_{23} we have

$\pi_{23}(p) \subseteq \pi_2(p) \cap \pi_3(p)$,

$\sigma_{23}(l)(o) = \sigma_2(l)(o) = \sigma_3(l)(o)$ if $\sigma_2(l)(o) = \sigma_3(l)(o)$ for every $l \in \Gamma_+$,

$\sigma_{23}(l)(o) = \sigma_2(l)(o) \cap \sigma_3(l)(o)$ if $\sigma_2(l)$ and $\sigma_3(l)$ are defined on o for every $l \in \Gamma_*$;

So for I_{123} we have

$\pi_{123}(p) \subseteq \pi_1(p) \cap \pi_{23}(p) \subseteq \pi_1(p) \cap \pi_2(p) \cap \pi_3(p)$,

$\sigma_{123}(l)(o) = \sigma_1(l)(o) = \sigma_23(l)(o)$ if $\sigma_1(l)(o) = \sigma_{23}(l)(o)$ for every $l \in \Gamma_+$, i.e.,

$\sigma_{123}(l)(o) = \sigma_1(l)(o)$ if $\sigma_1(l)(o) = \sigma_2(l)(o) = \sigma_3(l)(o)$ for every $l \in \Gamma_+$,

$\sigma_{123}(l)(o) = \sigma_1(l)(o) \cap \sigma_{23}(l)(o) = \sigma_1(l)(o) \cap \sigma_2(l)(o) \cap \sigma_3(l)(o)$ if $\sigma_1(l)$, $\sigma_2(l)$ and $\sigma_3(l)$ are defined on o for every $l \in \Gamma_*$;

Similarly for I'_{123} we have

$\pi'_{123}(p) \subseteq \pi_1(p) \cap \pi_2(p) \cap \pi_3(p)$,

$\sigma'_{123}(l)(o) = \sigma_1(l)(o)$ if $\sigma_1(l)(o) = \sigma_2(l)(o) = \sigma_3(l)(o)$ for every $l \in \Gamma_+$,

$\sigma'_{123}(l)(o) = \sigma_1(l)(o) \cap \sigma_2(l)(o) \cap \sigma_3(l)(o)$ if $\sigma_1(l)$, $\sigma_2(l)$ and $\sigma_3(l)$ are defined on o for every $l \in \Gamma_*$;

Therefore we have $\sigma_{123} = \sigma'_{123}$. Now we prove that $\pi_{123} = \pi'_{123}$. In fact, we only need to prove that for every record type $p \in \Sigma$ $\pi_{123}(p) = \pi'_{123}(p)$. For every object $o \in \pi_{123}(p)$, it has to satisfy all the properties of p under the mapping σ_{123}. Since $\sigma_{123} = \sigma'_{123}$, o also satisfy all the properties of p under the mapping σ'_{123}, so $o \in \pi'_{123}(p)$. For the same reason, for every object $o \in \pi'_{123}(p)$ we have $o \in \pi_{123}(p)$. Therefore $\pi_{123} = \pi'_{123}$. □

Theorem 5.18 (Model Intersection Property) Let P be a program and $\{M_i\}_{i \in I}$ be a non-empty set of models for P. Then the intersection $\cap_{i \in I} M_i$ is also a model for P.

Proof: Let $M = \cap_{i \in I} M_i$. Clearly, M is an interpretation for P. If M is not model for P, then either some record types, record objects or rules are not satisfied by M.

It is trivial to show that M must satisfy the type system of the program P.

Suppose that a record object $t = o : p(f \rightarrow o_1, s \rightarrow \{o_{s_1}, ..., o_{s_n}\})$ can not be satisfied by M. Since $M_i, i \in I$ are models of P, then $\models_{M_i} t$. That is, $o \in \pi_i(p)$, $\sigma_i(f)(o) = o_1$, $\{o_{s_1}, ..., o_{s_n}\} \subseteq \sigma_i(s)(o)$, for all $i \in I$, i.e., $\{o_{s_1}, ..., o_{s_n}\} \subseteq \cap_{i \in I} \sigma_i(s)(o)$. Therefore $\models_M d$, which is a contradiction.

Suppose a rule $r = p \Leftarrow p_1, ..., p_n$ can be satisfied by M_i, but not M. If one of M_i can not satisfy the body of the rule, then M can not satisfy the body therefore satisfy the rule. So suppose all M_i can satisfy the body. Let $X_1, ..., X_m$ be all the basic variables and $Y_1, ..., Y_n$ be all the set-valued variables in the body of the rule. Let $\nu_i, i \in I$ be a variable assignment such that $\models_{M_i} \nu_i(r)$ If all variable assignments are the same, then $\models_M \nu_i(r)$, which is a contradiction. If some variable assignments are different only on some of $Y_1, ..., Y_n$, then we still have $\models_M \nu_i(r)$ based on the definition of the satisfaction of the basic S-terms for set-valued variables, which is another contradiction. If some variable assignments are different on some of $X_1, ..., X_m$, let $\nu = \cap_{i \in I} \nu_i$, then there must be some p_i such that not $\models_M \nu(p_i)$, therefore $\models_M r$, which is still a contradiction. □.

Definition 5.48 A model M of P is *minimal* iff for each model N of P, if $N \subseteq M$ then $N = M$.

Theorem 5.19 If a program P has a model, then it has a unique minimal model which is the intersection of all possible models for P denoted M_P. □

Note that the intersection of all possible models for P is just the greatest lower bound of all possible models.

For example 2, the minimal model is M_1 which is equal to the intersection of M_1, M_2, and M_3. If we define the union of two interpretations in a similar way, we will note that the union may even not be an interpretation. For example, the union of M_2 and M_3 will contain $r_2 : r(f \rightarrow 2)$ and $r_2 : r(f \rightarrow 3)$. Therefore all possible models of a program forms a meet-semilattice.

Theorem 5.20 Let P be a program. Then $M_P = \{F : F$ is a logical consequence of $P\}$.

Proof: We have that

F is a logical consequence of P.

iff $P \cup \{\neg F\}$ is unsatisfiable, by theorem 5.7.

iff $P \cup \{\neg F\}$ has no Herbrand models, by theorem 5.9.

iff for every Herbrand model M of P, not $\models_M \neg F$.

iff for every Herbrand model M of P, $\models_M F$.

iff $F \in M_P$. □

If a program has a unique minimal model, then it can be found by the operator defined as follows.

Definition 5.49 Given a program P and an interpretation I, then

$$T_P(I) = \{\nu(p) : p \Leftarrow p, ..., p_n \in R, \text{there exists } \nu \text{ such that } \models_I \nu(p_i, ..., p_n)\}.$$

Clearly, T_P is monotonic, i.e, if $I_1 \subseteq I_2$, then $T_P(I_1) \subseteq T_P(I_2)$.

Theorem 5.21 Let P be a program and I be an interpretation of P. Then I is a model for P iff $T_P(I) \subseteq I$.

Proof: I is a model for P iff for each rule $p \Leftarrow p_1, ..., p_n$ in P, we have $\models_I \nu(\{p_1, ..., p_n\})$ implies $\models_I \nu(p)$ iff $T_P(I) \subseteq I$. □

Theorem 5.22 Let L be a meet-semilattice and $T : L \rightarrow L$ be monotonic mapping and $T(x) \leq x$. Then T has a least fixpoint $lfp(T) = glb\{x : T(x) \leq x\}$. [3]

Proof: Put $G = \{x : T(x) \leq x\}$ and $g = glb(G)$. We show that $g \in G$. Now $g \leq x$ for all $x \in G$, so that by the monotonicity of T, we have $T(g) \leq T(x)$, for all $x \in G$. Thus $T(g) \leq x$, for all $x \in G$, and so $T(g) \leq g$, by the definition of glb. Hence $g \in G$.

Next we show that g is a fixpoint of T. It remains to show that $g \leq T(g)$. Now $T(g) \leq g$ implies $T(T(g)) \leq T(g)$ implies $T(g) \in G$. Hence $g \leq T(g)$, so that g is a fixpoint of T. $\Box$

Definition 5.50 The powers of the operator T_P is defined as follows:

$T_P \uparrow 0 = DB$

$T_P \uparrow n = T_P(T_P \uparrow n-1) \cup T_P \uparrow n-1, (n \geq 1)$

$T_P \uparrow \omega = lub\{T_P \uparrow n : \omega$ denotes the first ordinal number and $n \in \omega\}$

Theorem 5.23 The powers of the operator T_P has the following properties.

(a). For all α, $T_P \uparrow \alpha \subseteq lfp(T_P)$.

(b). For all $\alpha \in \omega$, $T_P \uparrow \alpha \subseteq T_P \uparrow (\alpha + 1)$

(c). For all $\alpha, \beta \in \omega$, if $\alpha \leq \beta$, then $T_P \uparrow \alpha \subseteq T_P \uparrow \beta$.

(d). For all $\alpha, \beta \in \omega$, if $\alpha \leq \beta$ and $T_P \uparrow \alpha = T_P \uparrow \beta$, then $T_P \uparrow \alpha = lfp(T_P)$. $\Box$

Theorem 5.24 Let $X = \{T_P \uparrow n : n \in \omega\}$. Then X is directed, i.e., every finite subset of X has an upper bound in X, and $\nu(\{p_1, ..., p_n\}) \subseteq lub(X)$ iff $\nu(\{p_1, ..., p_n\}) \subseteq I$, for some $I \in X$.

[3]In [29], this theorem holds for complete lattice, here I prove it also holds for meet-semilattice.

Proof: The first part of the theorem is straightforward. For the second part, it is trivial that $\nu(\{p_1, ..., p_n\}) \subseteq I$ implies $\nu(\{p_1, ..., p_n\} \subseteq lub(X)$.

Assume that $\nu(\{p_1, ..., p_n\}) \subseteq lub(X)$. Then for each $i, 1 \leq i \leq n$, we have $\nu(p_i) \in lub(X)$. If not $\nu(p_i) \in I$ for all $I \in X$, then not $\nu(p_i) \in lub(X)$, which is a contradiction to assumption. Therefore, for each $\nu(p_i)$, there is some $I_i \in X$ where $\nu(p_i) \in I_i$. Since there are only a finite number of I_i and every finite subset of X has an upper bound in X (part one of the theorem), we have some $I \in X$ such that $I = lub(\{I_1, ..., I_n\})$ and $\nu(\{p_1, ..., p_n\}) \subseteq I$. □

Theorem 5.25 Let $P = \langle S, DB, R \rangle$ and $X = \{T_P \uparrow n : n \in \omega\}$. Then T_P is continuous on X, i.e., $T_P(lub(X)) = lub(T_P(X))$, and $T_P \uparrow \omega = lfp(T_P)$.

Proof: Now we have that

$\nu(p) \in T_P(lub(X))$

iff $p \Leftarrow p_1, ..., p_n \in R$ and $\nu(\{p_1, ..., p_n\}) \subseteq lub(X)$

iff $p \Leftarrow p_1, ..., p_n \in R$ and $\nu(\{p_1, ..., p_n\}) \subseteq I$, for some $I \in X$

by theorem 5.24

iff $\nu(p) \in T_P(I)$ for some $I \in X$

iff $\nu(p) \in lub(T_P(X))$.

So we have $T_P(lub(X)) = lub(T_P(X))$.

For the second part of the theorem, we have that

$T_P(T_P \uparrow \omega) = T_P(lub(X)) = lub(T_P(X)) = lub\{T_P(T_P \uparrow n) : n \in \omega\} =$
$T_P \uparrow \omega$.

So $T_P \uparrow \omega = lfp(T_P)$. □

Theorem 5.26 Given a program $P = \langle S, DB, R\rangle$ which has a unique minimal model M_P, then $T_P \uparrow \omega$ exists and $T_P \uparrow \omega = M_P$.

Proof: $M_P = glb\{I : I \text{ is a model for } P\}$, by theorem 5.19

$= glb\{I : T_P(I) \subseteq I\}$, by theorem 5.21

$= lfp(T_P)$, by theorem 5.22

$= T_P \uparrow \omega$, by theorem 5.25. □

Theorem 5.27 Let P be a program and Q be a query. Suppose ν is variable assignment, then ν is an answer to the query Q iff $\models_{M_P} \nu(Q)$.

Proof: (Only if part:) We have that

$P \models \nu(Q)$ by definition 5.43

implies that for every model M of P, $\models_M \nu(Q)$

implies that for least (Herbrand) model M_P, $\models_{M_P} \nu(Q)$

(If part:)Now we have $\models_{M_P} \nu(Q)$

implies $\models_M \nu(Q)$ for every (Herbrand) model M

implies not $\models_M \neg\nu(Q)$

implies $P \cup \{\neg\nu(Q)\}$ has no (Herbrand) models

implies $P \cup \{\neg\nu(Q)\}$ has no models by theorem 5.14

implies $P \cup \{\neg\nu(Q)\}$ is unsatisfiable by definition 5.41

implies $P \models \nu(Q)$ by theorem 5.12. □

According to the above theorem, to prove that $P \models \nu(Q)$ where P is program and Q is a goal, we just need to consider the least (Herbrand) model M_P of P. If $\models_{M_P} \nu(Q)$ then ν is an answer to the query Q, otherwise it is not an answer. This concludes the whole theory for an S-logic program without negation.

Chapter 6

Transformation into First-Order Logic

Chapters 4 and 5 have shown that S-logic has an expressive syntax and sound semantics. This chapter will show that S-logic is also implementable in practice. It will show that satisfiable S-logic programs and queries can be transformed into a first-order Horn-clause logic program and queries and get correct answers. However, unsatisfiable programs can still work but generate undesired results.

6.1 Transformation of Type System

A type in S-logic is a name which has two aspects: extension and intension. The extension of a type is the set of all known objects belonging to this type, the intension of a type is the properties all objects belonging to this type have to have. So, each type of S-logic is transformed into four predicates: *class, attribute, class_object*, and *class_set*. *class* is used for denoting the existence of a type. If p is a type, then we will have $class(p)$ after the transformation. *attribute* is used for the intension of a type. If p has a property $b \rightarrow c$, then we will have $attribute(p, b, c)$. *class_object* is used for denoting that an object is known to belong to a class. If c is an object in type p, then we will have $class\_object(p, c)$. *class_set* is used for the extension of a type. If p is a finite type, and $\{a_1, ..., a_n\}$ are all elements of this type, then we have $class\_set(p, \{a_1, ..., a_n\})$. If p is a infinite type, then it is impossible to list all its extension. We can still use p to represent its extensions, so we have $class\_set(p, p)$. As well, another predicate *isa* is used to represent subtype relationships over the abstract types. If we know *student* is a subtype of *person*, then we have $isa(student, person)$.

6.1.1 Transformation of Basic Types

It is supposed that there are two built-in predicates: $integer(X)$ and $string(X)$ in the intended first-order logic. First, we have a general rule saying that every basic type is a type:

$$class(X) :- basic\_class(X).$$

Then for each basic type, we transform it into a predicate $basic\_class$. The following transformations of the two basic types, *integer* and *string* are always included in the transformed program of S-logic.

$$basic\_class(integer).$$
$$class\_object(integer, X) :- integer(X).$$
$$class\_set(integer, integer).$$

$$basic\_class(string).$$
$$class\_object(string, X) :- string(X).$$
$$class\_set(string, string).$$

The basic types of S-logic other than *integer* and *string* are transformed as follows:

- If we have $s = string(\{a_1, ..., a_n\})$ in the program P, then it is transformed into

$$basic\_class(s).$$
$$class\_object(s, X) :- class(string), class\_object(string, X),$$
$$(X = a_1; ...; X = a_n).$$
$$class\_set(s, X) :- class(s), setof(Y, class\_object(s, Y), X).$$

- If we have $s = integer(\{lb..rb\})$ in the program P, then it is transformed into

$$\begin{array}{l}
basic\_class(s). \\
class\_object(s,X) :- class(integer), class\_object(integer,X), \\
\qquad\qquad\qquad\qquad X \geq lb, X \leq rb. \\
class\_set(s,X) :- class(s), setof(Y, class\_object(s,Y), X).
\end{array}$$

In fact, the above rules for predicate *class_set* can be extended to the following general rule which applies to all types, so that we do not need to have one for each type.

$$\begin{array}{l}
class\_set(S,X) :- class(S), \\
\qquad\qquad\qquad\quad setof(Y, class\_object(s,Y), X), \\
\qquad\qquad\qquad\quad X \neq integer, X \neq string.
\end{array}$$

The transformations of *gender* = *string*({*'Male', 'Female'*}) and *agetype* = *integer*({1..120}) are as follows:

$$\begin{array}{l}
basic\_class(gender). \\
class\_object(gender,X) :- class(string), class\_object(string,X), \\
\qquad\qquad\qquad\qquad\qquad (X = \text{'Male'}; X = \text{'Female'}).
\end{array}$$

$$\begin{array}{l}
basic\_class(agetype). \\
class\_object(agetype,X) :- class(integer), class\_object(integer,X), \\
\qquad\qquad\qquad\qquad\qquad 1 \leq X \leq 120.
\end{array}$$

6.1.2 Transformation of Set Types

First, it is assumed that the transformed first-order logic has the *setof* predicate which treats a set as a list. It is trivial to define *subset* predicate over lists. According to the semantics of S-logic, if we have p which is either a basic type, a record type, or a built-in type, than we automatically have a set type $\{p\}$. But a set set type like $\{\{p\}\}$ is not allowed. So we have the following transformation. Besides, built-in types are used only for the semantics of S-logic programs so that we do not need to include them in the transformed program.

$class(set(S)) :- basic\_class(S);$
$\quad record\_class(S).$

$class\_object(set(S), X) :- class(set(S)),$
$\quad setof(Y, class\_object(S, Y), Z),$
$\quad subset(X, Z).$

6.1.3 Transformation of Record Types

The existence of record types is represented by the predicate *record_class*. The following rule is included in the transformed program which says that every record type is a type.

$class(X) :- record\_class(X).$

For each record type without *isa* label $p(l_1 \rightarrow p_1, ..., l_n \rightarrow p_n)$, its transformation is

$record\_class(p).$
$attribute(p, l_1, p_1) :- class(p_1).$
$...$
$attribute(p, l_n, p_n) :- class(p_n).$

For each record type with *isa* label

$$p(isa \rightarrow p_1, ..., isa \rightarrow p_m, l_{m+1} \rightarrow p_{m+1}, ..., l\_n \rightarrow p\_n),$$

its transformation is

$record\_class(p).$
$isa(p, p_1) :- class(p_1).$
$class\_object(p_1, X) :- class\_object(p, X).$
$attribute(p, X, Y) :- attribute(p_1, X, Y), X \neq l_{m+1}, ..., X \neq l_n.$
$...$
$isa(p, p_m) :- class(p_m).$
$class\_object(p_m, X) :- class\_object(p, X).$
$attribute(p, X, Y) :- attribute(p_m, X, Y), X \neq l_{m+1}, ..., X \neq l_n.$
$attribute(p, l_{m+1}, p_{m+1}) :- class(p), class(p_{m+1}).$
$...$

$attribute(p, l_n, p_n) : -class(p), class(p_n).$

For example, if there are two record types of S-logic :

$$\begin{aligned}
&person(name \rightarrow string,\\
&\quad sex \rightarrow gender,\\
&\quad age \rightarrow agetype,\\
&\quad address \rightarrow string).
\end{aligned}$$

$$\begin{aligned}
&student(isa \rightarrow person,\\
&\quad age \rightarrow young,\\
&\quad studying\_in \rightarrow dept,\\
&\quad taking \rightarrow \{course\},\\
&\quad borrowing \rightarrow \{book\}).
\end{aligned}$$

Their transformations are

$$\begin{aligned}
&record\_class(person).\\
&attribute(person, name, string) : -class(string).\\
&attribute(person, sex, gender) : -class(gender).\\
&attribute(person, age, agetype) : -class(agetype).\\
&attribute(person, address, string) : -class(string).
\end{aligned}$$

$$\begin{aligned}
&record\_class(student).\\
&isa(student, person) : -class(person).\\
&class\_object(person, X) : -class\_object(student, X).\\
&attribute(student, X, Y) : -attribute(person, X, Y), X \neq age.\\
&attribute(student, age, young) : -class(young).\\
&attribute(student, studying\_in, dept) : -class(dept).\\
&attribute(student, taking, set(course)) : -class(set(course)).\\
&attribute(student, borrowing, set(book)) : -class(set(book)).
\end{aligned}$$

6.1.4 Transformation of the Built-in Types

The existence of built-in types is represented by the predicates *built-in_class*(*all*), and *built-in_class*(*none*). The following rules are included in the transformed program which says that built-in types *all* and *none* are classes, all existing objects are also objects of *all* and all existing properties are also properties of *none*, and *all* is a supertype of all existing types and *none* is a subtype of all existing types.

$$
\begin{array}{l}
class(X) : -built\text{-}in\_class(X). \\
class\_object(P, X) : -class\_object(all, X). \\
attribute(none, X, Y) : -attribute(P, X, Y), class(P), P \neq none. \\
isa(X, all) : -class(X), X \neq all. \\
isa(none, X) : -class(X), X \neq none.
\end{array}
$$

A complete transformation of the sample type system of Chapter 4 is given in Appendix A.

6.2 Transformation of the Database

The database determines the extension of record types. It tells which type a record object in the database belongs to and what properties are known.

Let $o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow o_n)$ be a record object of the database. According to the definition, there is a record type p in the type system with properties $l_i \rightarrow p_i, 1 \leq i \leq n$ and $o \in p$, $o_i \in p_i, 1 \leq i \leq n$. Two predicates *class_object* and *attribute_value* are used for the transformation. The first is used for the extension of a type, the other is used for the intension of an object. Set objects are represented by a list. According to the semantics of a record object, we can have following transformation:

$$
\begin{array}{ll}
\multicolumn{2}{l}{class\_object(p, o)} \\
attribute\_value(o, l_1, o_1) : & -attribute(p, l_1, p_1), \\
 & class\_object(p_1, o_1). \\
\ldots & \\
attribute\_value(o, l_n, o_n) : & -attribute(p, l_1, p_n), \\
 & class\_object(p_n, o_n).
\end{array}
$$

However this transformation is not convenient because it needs to refer to the type system. It can be changed into a convenient transformation with a general rule as follows:

$class\_object(p, o)$
$attribute\_value0(o, l_1, o_1).$
...
$attribute\_value0(o, l_n, o_n).$

$attribute\_value(O, L, O_1) :- attribute\_value0(O, L, O_1),$
$\quad attribute(P, L, P_1), class\_object(P, O), class\_object(P_1, O_1).$

For example, suppose we have following record objects in the database:

$sally : person(name \rightarrow \text{'Sally'}, sex \rightarrow \text{'Female'}, age \rightarrow 14).$

$john : person(name \rightarrow \text{'John'}, sex \rightarrow \text{'Male'}, age \rightarrow 62,$
$\quad address \rightarrow \text{'439 5th Av NE'}).$

$jenny : student(name \rightarrow \text{'Jenny'}, sex \rightarrow \text{'Female'}, age \rightarrow 24,$
$\quad studying\_in \rightarrow math, taking \rightarrow \{m203, m321, cs213\}).$

Their transformations are

$class\_object(person, sally).$
$attribute\_value0(sally, name, \text{'Sally'}).$
$attribute\_value0(sally, sex, \text{'Female'}).$
$attribute\_value0(sally, age, 14).$

$class\_object(person, john).$
$attribute\_value0(john, name, \text{'John'}).$
$attribute\_value0(john, sex, \text{'Male'}).$
$attribute\_value0(john, age, 62).$
$attribute\_value0(john, address, \text{'439 5th Av NE'}).$

$class\_object(student, jenny).$
$attribute\_value0(jenny, name, \text{'Jenny'}).$
$attribute\_value0(jenny, sex\ \text{'Female'}).$
$attribute\_value0(jenny, age, 24).$
$attribute\_value0(jenny, studying\_in, math).$
$attribute\_value0(jenny, taking, [m203, m321, cs213]).$

$attribute\_value(O, L, O_1) :- attribute\_value0(O, L, O_1),$
$\quad attribute(P, L, P_1), class\_object(P, O), class\_object(P_1, O_1).$

A complete transformation of the sample database of Chapter 4 is given is Appendix B.

6.3 Transformation of S-terms

For each basic object, type, label, object constructor, basic variable, or set-valued variable, its transformation is still itself. For each set object, its transformation is a list. Empty list [] means an empty set. For example, the transformation of $\{a, b, c\}$ is [a, b, c]. Although a list structure is not a real set, the transformed program will treat it as a real set. The set-grouping variable will be transformed based on the context.

Transformation of Basic S-terms

- For a basic S-term $X : p$, its transformation is

  $class\_object(p, X)$.

- For a basic S-term $X : p(l_1 \rightarrow X_1, ..., l_m \rightarrow X_m, l_{m+1} \rightarrow \{X_{m+1}\}, l_n \rightarrow \{X_n\})$,

its transformation is

$$\begin{array}{l}
class(p),\\
class\_object(p, X),\\
attribute(p, l_1, P_1),\\
...,\\
attribute(p, l_n, P_n),\\
attribute\_value(X, l_1, X_1), class\_object(p_1, X_1),\\
...,\\
attribute\_value(X, l_m, X_m), class\_object(p_m, X_m),\\
attribute\_value(X, l_{m+1}, Y_{m+1}), member(X_{m+1}, Y_{m+1}),\\
\qquad class\_object(p_{m+1}, Y_{m+1}),\\
...,\\
attribute\_value(X, l_n, Y_n), member(X_n, Y_n),\\
\qquad class\_object(p_n, Y_n).
\end{array}$$

Transformation of Type S-Terms

- For a typed S-term $P := S$, its transformation is

  $class(P)$,
  $class\_set(P, S)$.

- For a typed S-term $P(isa \rightarrow P_1, ..., isa \rightarrow P_n)$, its transformation is

  $class(P)$,
  $class(P_1)$,
  ...,
  $class(P_n)$,
  $isa(P, P_1)$,
  ...,
  $isa(P, P_n)$.

- For a typed S-term $P(L_1 \rightarrow P_1, ..., L_n \rightarrow P_n)$, its transformation is

  $class(P)$,
  $class(P_1)$,
  ...,
  $class(P_n)$,
  $attribute(P, L_1, P_1)$,
  ...,
  $attribute(P, L_n, P_n)$.

- For a typed S-term $X{:}P(L_1 \rightarrow X_1, ..., L_m \rightarrow X_m, L_{m+1} \rightarrow \{X_{m+1}\}, L_n \rightarrow \{X_n\})$, where $P, L_1, ..., L_n$ might be variables, its transformation is

  $class(P)$,
  $class\_object(P, X)$,
  $attribute(P, L_1, P_1)$,
  ...,
  $attribute(P, L_m, P_m)$,
  $attribute(P, L_{m+1}, P_{m+1})$,
  ...,
  $attribute(P, L_n, P_n)$,
  $attribute\_value(X, L_1, X_1), class\_object(P_1, X_1)$,
  ...,
  $attribute\_value(X, L_m, X_m), class\_object(P_m, X_m)$,

$attribute\_value(X, L_{m+1}, Y_{m+1}), member(X_{m+1}, Y_{m+1}),$
$\quad class\_object(P_{m+1}, Y_{m+1}),$
$\ldots,$
$attribute\_value(X, L_n, Y_n), member(X_n, Y_n), class\_object(P_n, Y_n).$

6.4 Transformation of Rules

A rule consists of a head and a body of the form $p \Leftarrow body$. The body is a collection of basic literals which are either basic S-terms, negation of basic S-terms, disjunctive basic S-terms or comparison expressions. The previous section showd how to transform basic S-terms. Let ψ be a basic S-term and $trans(\psi)$ stand for the transformation of ψ. For a negation of basic S-terms $\neg\psi$, its transformation consists of the negative sign followed by the transformation of the basic S-terms without negation, i.e. $trans(\neg\psi) = \neg(trans(\psi))$. For a disjunctive basic S-term $\psi_1; \psi_2$, it transformation $trans(\psi_1; \psi_2) = trans(\psi_1); trans(\psi_2)$. For comparison expressions, their transformation are simply themselves, i.e., $trans(\psi_1 \theta \psi_2)$ $=trans(\psi_1)$ $\theta trans(\psi_2)$, where $\theta \in \{=\neq, \leq, \geq, >, <\}$ and ψ_1, ψ_2 are basic variables or basic objects. The transformation of the body of a rule is the conjunction of the transformation of the literals in the body. Later on $trans(body)$ will be used to stand for the transformation of the body. The transformation of the head of a rule is different from the transformation of an S-term, which depends on the usage of the rule.

Rules are used in two different ways. One is to deduce attribute values for existing objects. The other is to construct new objects and obtain their attribute values. These two usages lead to two slightly different transformations of the head of rules. The transformation of the body for both cases are the same.

Let $X : p(l_1 \rightarrow X_1, ..., l_m \rightarrow X_m, l_{m+1} \rightarrow \{X_{m+1}\}, L_n \rightarrow \{X_n\}) \Leftarrow body$ be a rule, where X is either a basic variable or a basic object, p is a type, $l_i, 1 \leq i \leq n$ are labels, $X_i, 1 \leq i \leq n$ are basic variables. This rule is used to deduce n attribute values for objects of type P. Its transformation is the following n sets of rules in first-order logic, each of which is used for one attribute value.

$attribute\_value(X, l_1, X_1) : -class(p), class\_object(p, X),$
$\quad attribute(p, l_1, P_1), class\_object(P_1, X_1), trans(body).$

$...,$

$attribute\_value(X, l_m, X_m) : -class(p), class\_object(p, X),$
$\quad attribute(p, l_m, P_m), class\_object(P_m, X_m), trans(body).$

$attribute\_value1(X, l_{m+1}, X_{m+1}) : -class(p), class\_object(p, X),$
$\quad trans(body).$

$attribute\_value(X, l_{m+1}, Y_{m+1}) : -attribute(p, l_{m+1}, P_{m+1}),$
$\quad setof(X_{m+1}, attribute\_value1(X, l_{m+1}, X_{m+1}), Y_{m+1}),$
$\quad class\_object(P_{m+1}, Y_{m+1}),$

$...,$

$attribute\_value1(X, l_n, X_n) : -class(p), class\_object(p, X),$
$\quad trans(body).$

$attribute\_value(X, l_n, Y_n) : -attribute(p, l_n, P_n),$
$\quad setof(X_n, attribute\_value1(X, l_n, X_n), Y_n),$
$\quad class\_object(P_n, Y_n),$

For example, given two S-logic rules of this kind:

(1) $X : person(address \rightarrow Y) \Leftarrow A \leq 20$
$\quad X : person(age \rightarrow A, father \rightarrow Z),$
$\quad Z : person(address \rightarrow Y).$

(2) $X : employee(heading \rightarrow \{Y\}) \Leftarrow$
$\quad Y : employee(working\_in \rightarrow D),$
$\quad D : dept(head \rightarrow X).$

Their transformations are:

(1) $attribute\_value(X, address, Y) : -class(person),$
$class\_object(person, X), attribute(person, address, P_1),$
$class\_object(P_1, Y), A \leq 20, class(person),$
$class\_object(person, X), attribute\_value(X, age, A),$
$attribute(person, age, P_2), class\_object(P_2, A),$
$attribute\_value(X, father, Z), attribute(person, father, P_3),$
$class\_object(P_3, Z), class(person), class\_object(person, Z),$
$attribute\_value(Z, address, Y),$
$attribute(person, address, P_4), class\_object(P_4, Y).$

(2) $attribute\_value1(X, heading, Y) : -class(employee),$
$class\_object(employee, X), class(employee),$
$class\_object(employee, Y), attribute\_value(Y, working\_in, D),$
$attribute(employee, working\_in, P_2), class\_object(P_2, D),$
$class(dept), class\_object(dept, D), attribute\_value(D, head, X),$
$attribute(dept, head, P_3), class\_object(P_3, X).$

$attribute\_value(heading, X, Z) : -attribute(employee, heading, P_1),$
$setof(Y, attribute\_value1(heading, X, Y), Z).$
$class\_object(P_1, Y),$

Let $f(X_1, .., X_m) : p(l_1 \rightarrow Y_1, ..., l_n \rightarrow \{Y_n\}) \Leftarrow body$ be a rule, where f is an m-ary function, $X_i, 1 \leq i \leq m$ are basic variables or basic objects, $Y_i, 1 \leq i \leq n$ are basic variables and $l_i, 1 \leq i \leq n$ are labels. This rule is used to construct objects and obtain their attribute values. Its transformation is slightly different from the transformation of the above rule which has one additional rule for the constructed object as follows, in addition to the n sets of rules for attribute values:

$$class\_object(p, f(X_1, ..., X_n)) : -class(p), trans(body).$$

For example, given an S-logic rule of this kind:

$id(X, Y) : family(father \rightarrow X, mother \rightarrow Y, children \rightarrow \{Z\}) \Leftarrow$
$Z : person(father \rightarrow X, mother \rightarrow Y).$

The transformation is:

$$\begin{aligned}&class\_object(family, id(X,Y)) :- class(family),\\&\quad class(person), class\_object(person, Z),\\&\quad attribute\_value(Z, father, X), attribute(person, father, P_4),\\&\quad class\_object(P_4, X), attribute\_value(Z, mother, Y),\\&\quad attribute(person, mother, P_5), class\_object(P_5, Y).\end{aligned}$$

$$\begin{aligned}&attribute\_value(id(X,Y), father, X) :- class(family),\\&\quad class\_object(family, id(X,Y)), attribute(family, father, P_1),\\&\quad class\_object(P_1, X), class(person), class\_object(person, Z),\\&\quad attribute\_value(Z, father, X), attribute(person, father, P_4),\\&\quad class\_object(P_4, X), attribute\_value(Z, mother, Y),\\&\quad attribute(person, mother, P_5), class\_object(P_5, Y).\end{aligned}$$

$$\begin{aligned}&attribute\_value(mother, id(X,Y), Y) :- class(family),\\&\quad class\_object(family, id(X,Y)), attribute(family, mother, P_2),\\&\quad class\_object(P_2, Y), class(person), class\_object(person, Z),\\&\quad attribute\_value(Z, father, X), attribute(person, father, P_4),\\&\quad class\_object(P_4, X), attribute\_value(Z, mother, Y),\\&\quad attribute(person, mother, P_5), class\_object(P_5, Y).\end{aligned}$$

$$\begin{aligned}&attribute\_value\mathrm{1}(children, id(X,Y), Z) :- class(family),\\&\quad class\_object(family, id(X,Y)),\\&\quad class(person), class\_object(person, Z),\\&\quad attribute\_value(Z, father, X), attribute(person, father, P_4),\\&\quad class\_object(P_4, X), attribute\_value(Z, mother, Y),\\&\quad attribute(person, mother, P_5), class\_object(P_5, Y).\end{aligned}$$

$$\begin{aligned}&attribute\_value(id(X,Y), children, A) :- attribute(family, children, P_3),\\&\quad setof(Z, attribute\_value\mathrm{1}(id(X,Y), children, Z), A).\\&\quad class\_object(P_3, A).\end{aligned}$$

It is trivial to eliminate the redundancy in the transformed program. A complete transformation of the sample rules of Chapter 4 is given in Appendix C.

6.5 Transformation of Queries

A query is a conjunction of literals starting with the question mark. Its transformation is just the conjunction of transformed literals. Section 6.3 already shows how to transform S-terms which are either basic S-terms or typed S-terms. Let S_1, S_2 be any two S-terms and their transformations are $trans(S_1)$ and $trans(S_2)$. For a negative literal $\neg S_1$, $trans(\neg S_1) = \neg trans(S_1)$. For a disjunctive literal $S_1; S_2$, $trans(S_1; S_2) = trans(S_1); trans(S_2)$. For comparison expressions $\psi_1 = \psi_2$, $\psi_1 \neq \psi_2$, their transformations are just themselves. For comparison expressions $\psi_1 \leq \psi_2$, $\psi_1 \geq \psi_2$, $\psi_1 < \psi_2$, $\psi_1 > \psi_2$, their transformation depending on whether or not ψ_1, ψ_2 are basic variables or basic objects. If ψ_1 and ψ_2 are basic variables or objects, then the transformations of above comparison expressions are themselves. If ψ_1 and ψ_2 are set-valued variables or set objects, then the transformation are as follows:

- For $\psi_1 \leq \psi_2$, it is transformed into $subset(\psi_1, \psi_2); \psi_1 = \psi_2$.
- For $\psi_1 \geq \psi_2$, it is transformed into $subset(\psi_2, \psi_1); \psi_1 = \psi_2$.
- For $\psi_1 > \psi_2$, it is transformed into $subset(\psi_2, \psi_1), \psi_1 \neq \psi_2$.
- For $\psi_1 < \psi_2$, it is transformed into $subset(\psi_1, \psi_2), \psi_1 \neq \psi_2$;

For example, given four queries of S-logic as follows:

$(1)?X : person(age \rightarrow Y, sex \rightarrow Z), Y \geq 50.$

$(2)?bookstore : dept(staff \rightarrow \{X\}), X : employee(salary \rightarrow Y).$

$(3)?smith : X(L \rightarrow Y).$

$(4)?P_1 := S_1, P_2 := S_2, S_1 \leq S_2.$

Their transformations are

$$\begin{aligned}(1)? - {} & class(person), class\_object(person, X),\\ & attribute\_value(X, age, Y), attribute(person, age, P_1),\\ & class\_object(P_1, Y), attribute\_value(X, sex, Z),\\ & attribute(person, sex, P_2), class\_object(P_2, Z),\ Y \geq 50.\end{aligned}$$

(2)$? - class(dept), attribute\_value(bookstore, staff, Z),$
$attribute(dept, staff, P_1), class\_object(P_1, Z), mem(X, Z),$
$class(employee), class\_object(employee, X),$
$attribute\_value(X, salary, Y), attribute(employee, salary, P_2),$
$class\_object(P_2, Y).$

(3)$? - class(X), class\_object(X, smith),$
$attribute\_value(smith, L, Y), attribute(X, L, P_1),$
$class\_object(P_1, Y).$

(4)$? - class(P_1), class\_set(P_1, S_1), class(P_2), class\_set(P_2, S_2),$
$subset(S_1, S_2).$

A complete transformation of the sample queries of Chapter 4 is given in Appendix D and Appendix E.

I have run the sample examples by using NU-Prolog, which gives me satisfcatory answers.

Chapter 7

Conclusion and Further Work

Approaches to deductive databases are torn by two opposing forces. On one side there are the stringent real-world requirements of actual databases. The requirements include efficient processing as well as the ability to express complex and subtle real-world relationships. On the other side are the simple and clear semantics of logic programming and its deductive power. The need for expressiveness has forced the deductive models away from their simple roots in logic programming.

In this thesis, I have shown two major problems underlying the first-order logic languages and examined several solutions to these problems. One is complex object modeling, the other is the ability to represent higher-order features. For complex object modeling, we need to represent object identity, data abstractions and inheritance. For higher-order features, we need to represent higher-order queries and sets. There is no direct way to represent these in first-order logic.

I propose a higher-order language called S-logic in an attempt to solve these two problems. S-logic supports object identity, data abstractions and inheritance, schemas, sets, and higher-order queries in a uniform way. Its definite programs have a well-defined least fixpoint semantics. Programs which in LDL do not have models have models in S-logic.

The treatment of multiple inheritance in S-logic is also noteworthy. In S-logic, objects are grouped into classes called types and types can be organized into a subtype hierarchy. Subtypes inherit all the properties of their supertypes and may have their own properties and may redefine (or restrict) their supertypes' properties. Inheritance can be either single or multiple. In the case of single inheritance, the subtype hierarchy has the form of a tree, i.e., every type has a unique supertype. In the case of multiple inheritance, a subtype can has more than one supertypes, the subtype relation forms a lattice. Multiple inheritance is more elegant than single inheritance, but more difficult to handle normally. In this thesis, I have given a clear set-inclusive semantics to multiple inheritance in S-logic.

I have also shown that S-logic can be transformed into Prolog so that S-logic is implementable in practice. Of course, S-logic is intended as a real deductive database language, and how to efficiently implement S-logic is a worthwhile topic for further research.

The theory developed here only applies to definite programs, i.e., the body of a rule in a program has no negation. Further work is needed to explore normal programs which include negation. It seems that a theory of negation can be developed similar to the stratification theory in [29].

7.1 Updates

There is another significant problem existing in deductive databases which I did not touch in this thesis. That is the update problem.

In Prolog, the basic update primitives are *assert* and *retract*. *Assert* is used to insert a single clause into the database. Assert always succeeds initially and fails when the computation backtracks. Clauses are deleted from the database in Prolog by calling *retract*. Initially, retract deletes the first clause in the database which unifies with the argument of retract. On backtracking, the next matching clause is removed. Retract fails when there are no remaining matching clauses.

The semantics of assert and retract are not well-defined. Even if we did take one particular implementation as the definition, the exact effect of calling code containing assert and retract is often difficult to predict. There are two factors to be considered: the set of answers returned and the resulting database update. These are interrelated and both rely on the procedural semantics of Prolog, rather than just the declarative semantics. The procedural semantics of Prolog affects what database updates are done. The order of execution of subgoals is as important as the logical content of the goal.

Many distributed Prolog systems have versions of retract with bugs or strange behavior (sometimes called "features"). In an external or distributed database system, the problems with assert and retract become much more severe. Even in the single user case, concurrent access is required to data structures on disk which may be quite complex. Multi-user access creates even more difficulties.

The notion of states is inherent in any notion of updates. The Dynamic Logic approach assigns state transition semantics to a logic program [35]. The closure operator associated with a logic program P computes a state of P in the sense that it assigns valuations to the variables of P. Updates can be viewed as transitions of a state through a state-space. In the absence of updates, a classical logic program has only one state and queries map this state to itself. So it reduces to the classical semantics of logic program. Two kinds of updates are distinguished in [35] which have different semantics. First, those in which update actions depend on the order of execution, that is, different orders of execution may yield different final states. This kind of update is represented by $(\alpha;\beta)$ where α and β stand for update predicates. The semantics for this kind does not require that α executed before and after β gives the same result. The other kind are those in which all different orders of execution yield the same final state. A syntactic test has been shown in that paper which can ensure this property.

The Dynamic Logic interpretation of updates [35] gives a clean semantics and is consonant with the operational meanings of the update predicates. But this semantics is not declarative and is too complicated to be useful.

To reduce the number of database states by grouping small changes into big ones and to address the issue of concurrency and atomicity of certain operations, the concept of transactions are introduced into deductive databases in [34]. The transaction concept has been widely used in relational database systems where transactions are normally transparent to the users. A transaction is a collection of updates which must be done atomically. This naturally specifies some form of concurrency control. In [35], a transaction is specified by two sets: the facts to be deleted (D) and the facts to be inserted (I). The new database state (New_db) after the transaction is defined in terms of the old database state (Old_db) before the transaction, D and I:

$$New\_db = (Old\_db - D) \cup I$$

This definition corresponds to performing deletions before insertions. Only if the transaction is committed, then the updates have been made by first doing all the deletions then all the insertions.

The main advantage of introducing transactions is that it gives a simple declarative semantics for updates. However explicitly specifying transactions seems to be a burden to the user.

Another approach which can solve the update problem is that of Starlog [13]. Starlog is a temporal logic programming language which handles time explicitly. Every predicate in Starlog has a temporal argument which is a real interval. So the database of Starlog is a history database and updates are represented as changes with "logical" time.

There are two ways in which time can be used in the Starlog database. One way is to use the time values to record actual history database information. Used in this way, it should be possible to query information about the past. A different way of using time in a database is just to express the semantics of updates and changes to the database. Used in this way, time would have no meaning within the database itself. In such a system the state of the database would be at its current time. A query could be made only at the current time and updates would be inserted and occur at the current time. The appropriate sequencing of updates would be ensured by giving independent sources of updates (for example different users in a multi-terminal system) their own unique time stamps.

A major feature of Starlog is that it permits a declarative semantics based on a bottom-up, least-fixpoint computation instead of the top-down, left-right backtracking of Prolog. Another is that it can be used in conjunction with an algorithm such as TimeWarp [24] to form a distributed database with a semantics identical to that of the sequential implementation.

It seems that it is possible to extend S-logic based on the ideas of Starlog to solve the update problem, i.e, incorporating an explicit temporal dimension into S-logic. I have started working on this subject.

It seems to me that the extended S-logic will be semantically sound. But a lot more work is needed.

Bibliography

[1] Abiteboul, S., Grumbach, S. "COL: A Logic-Based Language for Complex Objects," *Proc. Inter. Conf. on Extending Database Technology*, Venice, Italy, 1988, pp 271-293.

[2] Abiteboul, S., Hull, R. "IFO: A Formal Semantic Database Model," *ACM Trans. Database Systems*, Vol. 12, No. 4, (Dec. 1987), pp 525-565.

[3] Albano, A., Cardelli, L., and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Trans. Database Systems*, Vol. 10, No. 2 (June 1985), pp. 230-260.

[4] Beeri, C., Naqvi, S., Shmueli, O. and Tsur, S., "Sets and Negation in a Logic Database Language (LDL)," *MCC Tech. Report* DB-375-86, 1987.

[5] Bic, L. and Gibert, J. P.
"Learning from AI: New Trends in Database Technology" *IEEE Computer Society,* Vol. 19, No. 3 (March 1986), pp 44-54.

[6] Borkin, S. A., Data Models: A Semantic Approach For Database Systems, *MIT Press*, 1980.

[7] Brodie, M. L., "On the Development of Data Models," in *On Conceptual Modelling*, M. L. Brodie, J, Mylopoulos, and J. W. Schmidt Eds., Springer-Verlag, New York. 1984, pp. 19-48.

[8] Brodie, M. L., and Ridjanovic, D., "On the Design and Specification of Database Transactions," in *On Conceptual Modelling* M. L. Brodie, J, Mylopoulos, and J. W. Schmidt Eds. Springer-Verlag, New York. 1984, pp. 276-232.

[9] Cardelli, L., "A Semantics of Multiple Inheritance," *Proc. Inter. Sympo. on Semantics of Data Types*, LNCS 173, June, 1984, pp. 51-67.

[10] Ceri, S., Gottlob, G. and Wiederhold, G. "Efficient Database Access from Prolog" *IEEE Trans. Software Engineering*, Vol. 15, No. 2 (Feb. 1989), pp 153-164.

[11] Chen, P. P. S. "The Entity-Relationship Model — Toward a Unified View of Data" *ACM Trans. Database System,* Vol. 1, No. 1 (Mar. 1976), pp 9-36.

[12] Chen, W. and Warren, D. S., "C-Logic for Complex Objects," *ACM PODS*, 1989, pp. 369-378.

[13] Cleary, J. G., "Colliding Pucks Solved Using a Temporal Logic," *Proc. Conf. on Distributed Simulation*, Western Multiconference, S.C.S., San Diego, January 1990.

[14] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks" *Comm. ACM*, Vol. 13, No. 6 (June, 1970), pp 377-397.

[15] Codd, E. F. "Extending the Database Relational Model to Capture More Meaning" *ACM Trans. Database System*, Vol. 4, No. 4 (Dec. 1979), pp 297-434.

[16] Copeland, G. P. and Khoshafian, S. N., "Identity and Versions for Complex Objects" *MCC Tech. Reposrt*, DB-138-86, 1986.

[17] Genesereth, M.R., Ginsberg, M.L. "Logic Programming" *Communications of the ACM* Vol. 28, No. 9 (Sept. 1984), pp 933-941.

[18] Gallaire, H., Minker, J., and Nicolas, J. M., "Logic and Databases: A Deductive Approach," *ACM Computing Surveys*, Vol. 16, No. 2 (June 1984), pp. 153-186.

[19] Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Database Model.," *ACM Trans. Database Systems*, Vol. 6, No. 3 (Sept. 1981), pp. 351-386.

[20] Hassan, A. K., and Nasr, R., "LOGIN: A Logic Programming Language with built-in Inheritance," *Journal of Logic Programming*, Vol. 3, No. 3 (Oct. 1986), pp. 185-215.

[21] Hull, R. and King, R., "Semantic Database Modeling: Survey, Applications, and Research issues," *ACM Computing Surveys*, Vol. 19, No. 3 (Sept. 1987), pp. 201-260.

[22] Housel, B. C., Waddle, V. and Yao, S. B. "The Functional Dependency Model for Logicial Databases Design" in *Proc. 5th Intel. Conf. on Vary Large Databases*, October, 1979.

[23] Jacobs, B. E. "On Database Logic" *Journal of ACM* Vol. 29, No. 2 (April 1982), pp 310-332.

[24] Jefferson, D. R., "Virtual Time," *ACM Trans. on Programming Languages and Systems*, Vol. 7, No.3 (July, 1985), pp. 404-423.

[25] Kifer, M. and Wu, J., "A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited)," *Proc. ACM PODS*, 1989, pp. 379-393.

[26] Kifer, M. and Lausen, G., "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema," *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1989, pp. 134-146.

[27] Krishnamurthy, R,. and Naqvi, S., "Towards a Real Horn Clause Language," *Proc. 14th VLDB* Los Angeles, USA, 1988, pp. 252-263.

[28] Kuper, G. M., "Logic Programming With Sets", *Proc. ACM PODS*, 1987, pp. 11-20.

[29] Lloyd, J. W., "Foundations of Logic Programming," *Springer Verlag*, 2nd edition, 1987.

[30] MacLennan, B. J., A View of Object-Oriented Programming, *Naval Postgraduate School Tech. Report*, NPS52-83-001, 1983.

[31] Maier, D., "A Logic for Objects," *Proc. the Workshop on Deductive Databases and Logic Programming*, 1986

[32] Maier, D., "Why Database Languages are a Bad Idea," *Proc. the Workshop on Databases Programming Languages*, Roscoff, France,1987.

[33] Mylopoulos, J., Bernstein,P. A. and Wong, H. K. T. "A Language Facility for Designing Database-Intensive Applications" *ACM Trans. on Database Systems,* Vol. 5, No. 2 (June. 1980), pp 185-207.

[34] Naish, L., Thom, L. A., and Ramamohanarao, K., "Concurrent Database Updates in Prolog," *Proc. Fourth Inter. Conf. Logic Programming,* (July. 1987), pp. 178-195.

[35] Naqvi, S. and Krishnamurthy, R., "Database Updates in Logic Programming," *ACM Sym. on PODS*, 1988, pp. 261-262/

[36] Peckham, J. and Maryanski, F., "Semantic Database Models," *ACM Computing Surveys*, Vol. 20, No. 3 (Sept. 1988), pp. 153-189.

[37] Rybinski, H. "On First-Order Logic Databases" *ACM Trans. Database Systems,* Vol. 12, No. 3 (Sept. 1987), pp 325-349.

[38] Reiter, R., "Towards a Logical Reconstruction of Relational Database Theory," in *On Conceptual Modelling*, M. L. Brodie, J, Mylopoulos, and J. W. Schmidt Eds., Springer-Verlag, New York. 1984, pp. 19-48.

[39] Shipman, D. W. "The Functional Extending the Database Relational Model to Capture More Meaning" *ACM Trans. Database System,* Vol. 4, No. 4 (Dec. 1979), pp 297-434.

[40] Smith, J. M., and Smith, D. C. P., "Database Abstractions: Aggregation and Generalization," *ACM Trans. on Database Systems*, Vol. 2, No. 2 (June. 1977), pp. 105-133.

[41] Su, S. Y. W., "Modeling Integrated Manufacturing Data with SAM*," *IEEE Computer Society*, Vol. 19, No. 1 (Jan. 1986), pp. 34-49.

[42] Thom, J. A. and Zobel, J., NU–Prolog REference Manual, *Department of Computer Science, University of Melbourne, Tech. Report*, 86/10, 1986.

[43] Tsichritzis, D. C. and Lochovsky, F. H. Data Models *Prentice-Hall, Englewood Cliffs, N.J.* 1982

[44] Tsur, S. and Zaniolo, C., "LDL: A Logic-Based Data-Language," *Proc. 12th VLDB* Kyoto, Japan, 1986, pp. 33-41.

[45] Ullman, J. D. *Principles of Database Systesm.* 2nd. ed. Computer Science Press, Rockville, MD, 1982.

[46] Ullman, J. D. "Implementation of Logical Query Languages for Databases" *ACM Trans. Database Systems,* Vol. 1, No. 1 (Feb. 1983), pp 3-23.

[47] Warren, D. H. D., "Higher-Order extensions to PROLOG: are they needed," in *Machine Intelligence* 10, J.E.Hayes, Donald Michie, and Y-H. Pao, eds, Ellis Horwood with John Willey and Sons, 1982, pp. 441-454.

[48] Zaniolo, C., Act-Kaci, H., Beech, D., Cammarata, S., and Kerschberg, L., "Object Oriented Database Systems and Knowledge Systems," *MCC Tech. Report* DB-038-85, 1985.

Appendix A

Transformation of Sample Type System

(Type 1.)

$record\_class(person).$
$attribute(person, name, string) :- class(string).$
$attribute(person, sex, gender) :- class(gender).$
$attribute(person, age, agetype) :- class(agetype).$
$attribute(person, spouse, person) :- class(person).$
$attribute(person, address, string) :- class(string).$
$attribute(person, father, person) :- class(person).$
$attribute(person, mother, person) :- class(person).$

(Type 2.)

$record\_class(student).$
$isa(student, person).$
$class\_object(person, X) :- class\_object(student, X).$
$attribute(student, X, Y) :- attribute(person, X, Y), X \neq age.$
$attribute(student, age, young) :- class(young).$
$attribute(student, studying\_in, dept) :- class(dept).$
$attribute(student, taking, set(course)) :- class(set(course)).$
$attribute(student, borrowing, set(book)) :- class(set(book)).$

(Type 3.)

$record\_class(employee).$
$isa(employee, person).$
$class\_object(person, X) :- class\_object(employee, X).$
$attribute(employee, X, Y) :- attribute(person, X, Y), X \neq age.$
$attribute(employee, age, midage) :- class(midage).$
$attribute(employee, working\_in, dept) :- class(dept).$
$attribute(employee, heading, set(person)) :- class(set(person)).$
$attribute(employee, salary, employeesalary) :- class(employeesalary).$

(Type 4.)

$record\_class(workingstudent).$
$isa(workingstudent, student).$
$isa(workingstudent, employee).$
$class\_object(employee, X) :- class\_object(workingstudent, X).$
$class\_object(student, X) :- class\_object(workingstudent, X).$
$attribute(workingstudent, X, Y) :- attribute(student, X, Y),$
$\quad X \neq age, X \neq salary.$
$attribute(workingstudent, X, Y) :- attribute(employee, X, Y),$
$\quad X \neq age, X \neq salary.$
$attribute(workingstudent, age, ymage) :- class(ymage).$
$attribute(workingstudent, salary, support) :- class(support).$

(Type 5.)

$basic\_class(agetype).$
$class\_object(agetype, X) :- X \geq 1, X \leq 120, class\_object(integer, X).$

(Type 6.)

$basic\_class(young).$
$class\_object(young, X) :- X \geq 15, X \leq 30, class\_object(integer, X).$

(Type 7.)

$basic\_class(midage).$
$class\_object(midage, X) :- X \geq 25, X \leq 60, class\_object(integer, X).$

(Type 8.)

$basic\_class(ymage).$

$class\_object(ymage, X) :- X \geq 25, X \leq 30, class\_object(integer, X).$

(Type 9.)

$basic\_class(employeesalary).$
$class\_object(employeesalary, X) :- X \geq 0, X \leq 50,$
$\quad class\_object(integer, X).$

(Type 10.)

$basic\_class(support).$
$class\_object(support, X) :- X \geq 0, X \leq 15, class\_object(integer, X).$

(Type 11.)

$basic\_class(gender).$
$class\_object(gender, X) :- class(string), class\_object(string, X),$
$(X = \text{“}Male\text{”}; X = \text{“}Female\text{”}).$

(Type 12.)

$record\_class(dept).$
$attribute(dept, name, string) :- class(string).$
$attribute(dept, head, employee) :- class(employee).$
$attribute(dept, staff, set(employee)) :- class(set(employee)).$

(Type 13.)

$record\_class(course).$
$attribute(course, name, string) :- class(string).$
$attribute(course, credit, integer) :- class(integer).$
$attribute(course, taken\_by, set(student)) :- class(set(student)).$

(Type 14.)

$record\_class(book).$
$attribute(book, name, string) :- class(string).$
$attribute(book, no, string) :- class(string).$
$attribute(book, author, person) :- class(person).$
$attribute(book, published\_by, string) :- class(string).$
$attribute(book, price, integer) :- class(integer).$

(Type 15.)

$record\_class(family).$
$attribute(family, father, person) :- class(person).$
$attribute(family, mother, person) :- class(person).$
$attribute(family, children, set(person)) :- class(set(person)).$

(Type 16.)

```
record_class(house).
attribute(house, location, string) : -class(string).
attribute(house, occupied_by, set(person)) : -class(set(person)).
```

(Type 17.)

```
record_class(sameage).
attribute(sameage, number, agetype) : -class(agetype).
attribute(sameage, shared_by, set(person)) : -class(set(person)).
```

(General Rules for Type Systems)

```
class(X) : -basic_class(X); record_class(X); built-in_class(X).
class(set(X)) : -basic_class(X); record_class(X); built-in_class(X).
class_object(set(P), X) : -setof(Y, class_object(P, Y), Z), subset(X, Z).

basic_class(string).
class_object(string, X) : -string(X).
class_set(string, string).

basic_class(integer).
class_object(integer, X) : -integer(X).
class_set(integer, integer).

built-in_class(all).
built-in_class(none).
class_object(P, X) : -class_object(all, X).
attribute(none, X, Y) : -attribute(P, X, Y), class(P), P ≠ none.
isa(X, all) : -class(X), X ≠ all.
isa(none, X) : -class(X), X ≠ none.

subset([], X).
subset([X|Y], Z) : -mem(X, Z), del(X, Z, Z1), subset(Y, Z1).

del(X, [X|Y], Y).
del(X, [Z|Y], [Z|Y1]) : -del(X, Y, Y1).

mem(X, [X|_]).
mem(X, [Y|Z]) : -mem(X, Z).
```

$$\begin{aligned}&class\_set(X,Y) : -X \neq integer, X \neq string,\\&\qquad setof(A, class\_object(X,A), Y).\end{aligned}$$

Appendix B

Transformation of Sample Database

(Record Object 1.)

```
class_object(person, sally).
attribute_value0(sally, name, "Sally").
attribute_value0(sally, sex, "Female").
attribute_value0(sally, age, 14).
attribute_value0(sally, father, bob).
attribute_value0(sally, mother, mary).
```

(Record Object 2.)

```
class_object(person, john).
attribute_value0(john, name, "John").
attribute_value0(john, sex, "Male")
attribute_value0(john, age, 62).
attribute_value0(john, address, "439 5th Av NE").
```

(Record Object 3.)

```
class_object(student, jenny).
attribute_value0(jenny, name, "Jenny").
attribute_value0(jenny, sex, "Female").
attribute_value0(jenny, age, 24).
attribute_value0(jenny, spouse, smith).
attribute_value0(jenny, father, henry).
attribute_value0(jenny, studying_in, math).
attribute_value0(jenny, taking, [m203, m321, cs213]).
```

(Record Object 4.)

```
class_object(student, phil).
attribute_value0(phil, name, "Phil").
attribute_value0(phil, sex, "Male").
attribute_value0(phil, age, 18).
attribute_value0(phil, father, bob).
attribute_value0(phil, mother, mary).
```

```
attribute_value0(phil, studying_in, cpsc).
attribute_value0(phil, taking, [cs450, cs213]).
attribute_value0(phil, borrowing, [pascal, prolog]).
```

(Record Object 5.)

```
class_object(employee, mary).
attribute_value0(mary, name, "Mary").
attribute_value0(mary, sex, "Female").
attribute_value0(mary, age, 39).
attribute_value0(mary, spouse, bob).
attribute_value0(mary, working_in, bookstore).
attribute_value0(mary, salary, 35).
attribute_value0(mary, address, "128 2nd Av SW").
```

(Record Object 6.)

```
class_object(employee, henry).
attribute_value0(henry, name, "Henry").
attribute_value0(henry, sex, "Male").
attribute_value0(henry, age, 50).
attribute_value0(henry, father, bob).
attribute_value0(henry, working_in, cpsc).
attribute_value0(henry, address, "128 2nd Av NW").
attribute_value0(henry, salary, 50).
```

(Record Object 7.)

```
class_object(employee, bob).
attribute_value0(bob, name, "Bob").
attribute_value0(bob, sex, "Male").
attribute_value0(bob, age, 40).
attribute_value0(bob, father, john).
attribute_value0(bob, working_in, math).
attribute_value0(bob, address, "257 9th Av SW").
attribute_value0(bob, salary, 40).
```

(Record Object 8.)

```
class_object(workingstudent, smith).
attribute_value0(smith, name, "Smith").
attribute_value0(smith, sex, "Male").
attribute_value0(smith, age, 30).
```

```
attribute_value0(smith, father, john).
attribute_value0(smith, studying_in, cpsc).
attribute_value0(smith, working_in, cpsc).
attribute_value0(smith, address, "3 7th Av SW").
attribute_value0(smith, salary, 12).
attribute_value0(smith, taking, [cs450]).
```

(Record Object 9.)

```
class_object(workingstudent, dennis).
attribute_value0(dennis, name, "Dennis").
attribute_value0(dennis, sex, "Male").
attribute_value0(dennis, age, 30).
attribute_value0(dennis, father, henry).
attribute_value0(dennis, studying_in, math).
attribute_value0(dennis, working_in, bookstore).
attribute_value0(dennis, salary, 8).
```

(Record Object 10.)

```
class_object(dept, cpsc).
attribute_value0(cpsc, name, "Computer Science").
attribute_value0(cpsc, head, henry).
```

(Record Object 11.)

```
class_object(dept, math).
attribute_value0(math, name, "Mathematics").
attribute_value0(math, head, bob).
```

(Record Object 12.)

```
class_object(dept, bookstore).
attribute_value0(bookstore, name, "Book Store").
attribute_value0(bookstore, manager, mary).
```

(Reocrd Object 13.)

```
class_object(course, cs213).
attribute_value0(cs213, name, "Programming Language").
attribute_value0(cs213, credit, 2).
```

(Record Object 14.)

$class\_object(course, cs450).$
$attribute\_value0(cs450, name,$ *"Artificial Intelligence"*$).$
$attribute\_value0(cs450, credit, 4).$

(Record Object 15.)

$class\_object(course, m203).$
$attribute\_value0(m203, name,$ *"Calculus"*$).$
$attribute\_value0(m203, credit, 6).$

(Record Object 16.)

$class\_object(course, m321).$
$attribute\_value0(m321, name,$ *"Algebra"*$).$
$attribute\_value0(m321, credit, 4).$

(Record Object 17.)

$class\_object(book, pascal).$
$attribute\_value0(pascal, name,$ *"Pascal"*$).$
$attribute\_value0(pascal, author, henry).$
$attribute\_value0(pascal, published\_by,$ *"Practice"*$).$
$attribute\_value0(pascal, price, 35).$

(Record Object 18.)

$class\_object(book, prolog).$
$attribute\_value0(prolog, name,$ *"Prolog"*$).$
$attribute\_value0(prolog, author, john).$
$attribute\_value0(prolog, published\_by,$ *"Springer"*$).$
$attribute\_value0(prolog, price, 50).$

(General Rules for Database.)

$class\_object(set(S), X) : -class(set(S)),$
    $setof(Y, class\_object(S, Y), Z), subset(X, Z).$
$attribute\_value(O, L, O1) : -attribute\_value0(O, L, O1),$
    $attribute(P, L, P1), class\_object(P, O), class\_object(P1, O1).$

Appendix C

Transformation of Sample Rules

(Rule 1.)

```
attribute_value(X, address, Y) :-
        class(person), class_object(person, X),
        attribute(person, address, P1), class_object(P1, Y),
        A ≤ 20, class(person), class_object(person, X),
        attribute_value(X, age, A), attribute(person, age, P2),
        class_object(P2, A), attribute_value(X, father, Z),
        attribute(person, father, P3), class_object(P3, Z),
        class(person), class_object(person, Z),
        attribute_value(Z, address, Y), attribute(person, address, P4),
        class_object(P4, Y).
```

(Rule 2.)

```
attribute_value(X, address, Y) :-
        class(person), class_object(person, X),
        attribute(person, address, P1), class_object(P1, Y),
        class(person), class_object(person, X),
        attribute_value(X, spouse, Z), attribute(person, spouse, P2),
        class_object(P2, Z), class(person), class_object(person, Z),
        attribute_value(Z, address, Y), attribute(person, address, P3),
        class_object(P3, Y).
```

(Rule 3.)

```
attribute_value1(X, taken_by, Y) :-
        class(course), class_object(course, X),
        class(student), class_object(student, Y),
        attribute_value(Y, taking, XX), attribute(student, taking, P2),
        class_object(P2, XX), mem(X, XX).

attribute_value(X, taken_by, YY) :-
        attribute(course, taken_by, P1),
        setof(Y, attribute_value1(X, taken_by, Y), YY),
        class_object(P1, YY).
```

(Rule 4.)

$attribute\_value1(X, heading, Y) :-$
$\quad class(employee), class\_object(employee, X),$
$\quad class(employee), class\_object(employee, Y),$
$\quad attribute\_value(Y, working\_in, D),$
$\quad attribute(employee, working\_in, P_2),$
$\quad class\_object(P_2, D), class(dept), class\_object(dept, D),$
$\quad attribute\_value(D, head, X), attribute(dept, head, P_3),$
$\quad class\_object(P_3, X).$

$attribute\_value(X, heading, YY) :-$
$\quad attribute(employee, heading, P_1),$
$\quad setof(Y, attribute\_value1(X, heading, Y), YY),$
$\quad class\_object(P_1, YY).$

(Rule 5.)

$attribute\_value1(X, staff, Y) :-$
$\quad class(dept), class\_object(dept, X),$
$\quad class(employee), class\_object(employee, Y),$
$\quad attribute\_value(Y, working\_in, X),$
$\quad attribute(employee, working\_in, P_2),$
$\quad class\_object(P_2, X).$

$attribute\_value(X, staff, YY) :-$
$\quad attribute(dept, staff, P_1),$
$\quad setof(Y, attribute\_value1(X, staff, Y), YY),$
$\quad class\_object(P_1, YY).$

(Rule 6.)

$class\_object(family, id(X, Y)) :-$
$\quad class(family), class(person), class\_object(person, Z),$
$\quad attribute\_value(Z, father, X), attribute(person, father, P_4),$
$\quad class\_object(P_4, X), attribute\_value(Z, mother, Y),$
$\quad attribute(person, mother, P_5), class\_object(P_5, Y).$

$attribute\_value(id(X, Y), father, X) :- class(family),$
$\quad class\_object(family, id(X, Y)), attribute(family, father, P_1),$
$\quad class\_object(P_1, X), class(person), class\_object(person, Z),$
$\quad attribute\_value(Z, father, X), attribute(person, father, P_4),$
$\quad class\_object(P_4, X), attribute\_value(Z, mother, Y),$

$attribute(person, mother, P_5), class\_object(P_5, Y).$

$attribute\_value(id(X, Y), mother, Y) :- class(family),$
$class\_object(family, id(X, Y)), attribute(family, mother, P_2),$
$class\_object(P_2, Y), class(person), class\_object(person, Z),$
$attribute\_value(Z, father, X), attribute(person, father, P_4),$
$class\_object(P_4, X), attribute\_value(Z, mother, Y),$
$attribute(person, mother, P_5), class\_object(P_5, Y).$

$attribute\_value1(id(X, Y), children, Z) :- class(family),$
$class\_object(family, id(X, Y)),$
$class(person), class\_object(person, Z),$
$attribute\_value(Z, father, X), attribute(person, father, P_1),$
$class\_object(P_1, X), attribute\_value(Z, mother, Y),$
$attribute(person, mother, P_2), class\_object(P_2, Y).$

$attribute\_value(X, children, YY) :-$
$attribute(family, children, P_3),$
$setof(Y, attribute\_value1(X, children, Y), YY),$
$class\_object(P_3, YY).$

(Rule 7.)

$class\_object(house, id(X)) :- class(house),$
$attribute\_value(Y, address, X), attribute(person, address, P_3),$
$class\_object(P_3, X).$

$attribute\_value(id(X), location, X) :- class(house),$
$class\_object(house, id(X)), attribute(house, location, P_1),$
$class\_object(P_1, X),$
$class(person), class\_object(person, Y),$
$attribute\_value(Y, address, X), attribute(person, address, P_3),$
$class\_object(P_3, X).$

$attribute\_value1(id(X), occupied\_by, Y) :- class(house),$
$class\_object(house, id(X)),$
$class(person), class\_object(person, Y),$
$attribute\_value(Y, address, X), attribute(person, address, P_3),$
$class\_object(P_3, X).$

$attribute\_value(id(X), occupied\_by, YY) :-$
$attribute(house, occupied\_by, P_2),$
$setof(Y, attribute\_value1(id(X), occupied\_by, Y), YY),$
$class\_object(P_2, YY).$

(Rule 8.)

class_object(sameage, id(X)) :-$class(sameage),$
$class(person),$
$class\_object(person, Y),$
$attribute\_value(Y, age, X),$
$attribute(person, age, P_3),$
$class\_object(P_3, X).$

$attribute\_value(id(X), number, X) : -class(sameage),$
$class\_object(sameage, id(X)),$
$attribute(sameage, number, P_1), class\_object(P_1, X),$
$class(person), class\_object(person, Y),$
$attribute\_value(Y, age, X), attribute(person, age, P_3),$
$class\_object(P_3, X).$

$attribute\_value1(id(X), shared\_by, Y) : -class(sameage),$
$class\_object(sameage, id(X)),$
$class(person), class\_object(person, Y),$
$attribute\_value(Y, age, X), attribute(person, age, P_3),$
$class\_object(P_3, X).$

$attribute\_value(id(X), shared\_by, YY) : -$
$attribute(sameage, age, P_2),$
$setof(Y, attribute\_value1(id(X), shared\_by, Y), YY),$
$class\_object(P_2, YY).$

Appendix D

Transformation of Sample Queries (I)

(Query 1.)? $class(person)$,
$class\_object(person, X)$,
$attribute\_value(X, age, Y)$,
$attribute(person, age, P_1)$,
$class\_object(P_1, Y)$,
$attribute\_value(X, sex, Z)$,
$attribute(person, sex, P_2)$,
$class\_object(P_2, Z)$,
$Y \geq 50$.

(Query 2.)? $class(workingstudent)$,
$class\_object(workingstudent, X)$,
$attribute\_value(X, studying\_in, Y)$,
$attribute(workingstudent, studying\_in, P_1)$,
$class\_object(P_1, Y)$,
$attribute\_value(X, working\_in, Y)$,
$attribute(workingstudent, working\_in, P_2)$,
$class\_object(P_2, Y)$.

(Query 3.)? $class(student)$,
$class\_object(student, T)$,
$attribute\_value(T, name, \text{“Phil”})$,
$attribute(student, name, P_1)$,
$class\_object(P_1, \text{“Phil”})$,
$attribute\_value(T, borrowing, XX)$,
$attribute(student, borrowing, P_2)$,
$class\_object(P_2, XX)$,
$mem(X, XX)$,
$class(book)$,
$class\_object(book, X)$,
$attribute\_value(X, author, Y)$,
$attribute(book, author, P_3)$,
$class\_object(P_3, Y)$,

$attribute\_value(X, price, Z),$
$attribute(book, price, P_4),$
$class\_object(P_4, Z).$

(Query 4.)? $class(course),$
$class\_object(course, cs213),$
$attribute\_value(cs213, taken\_by, XX),$
$attribute(course, taken\_by, P_1),$
$class\_object(P_1, XX),$
$mem(X, XX),$
$class(T),$
$class\_object(T, X),$
$attribute\_value(X, studying\_in, Y),$
$attribute(T, studying\_in, P_2),$
$class\_object(P_2, Y).$

(Query 5.)? $class(student),$
$class\_object(student, X),$
$attribute\_value(X, taking, YY),$
$attribute(student, taking, P_1),$
$class\_object(P_1, YY),$
$mem(Y, YY),$
$class(T),$
$class\_object(T, Y),$
$attribute\_value(Y, name, Z),$
$attribute(T, name, P_2),$
$class\_object(P_2, Z).$

(Query 6.)? $class(family),$
$class\_object(family, X),$
$attribute\_value(X, mother, mary),$
$attribute(family, mother, P_1),$
$class\_object(P_1, mary),$
$attribute\_value(X, children, YY),$
$attribute(family, children, P_2),$
$class\_object(P_2, YY),$
$mem(Y, YY),$
$class(T),$
$class\_object(T, Y),$
$attribute\_value(Y, age, Z),$
$attribute(T, age, P_3),$

$class\_object(P_3, Z).$

(Query 7.)? $class(dept),$
$class\_object(dept, bookstore),$
$attribute\_value(bookstore, staff, XX),$
$attribute(dept, staff, P_1),$
$class\_object(P_1, XX),$
$mem(X, XX),$
$class(T),$
$class\_object(T, X),$
$attribute\_value(X, salary, Y),$
$attribute(T, salary, P_2),$
$class\_object(P_2, Y).$

(Query 8.)? $class(house),$
$class\_object(house, T),$
$attribute\_value(T, address, X),$
$attribute(house, address, P_1),$
$class\_object(P_1, X),$
$attribute\_value(T, occupied\_by, Y),$
$attribute(house, occupied\_by, P_2),$
$class\_object(P_2, Y).$

(Query 9.)? $class(sameage),$
$class\_object(sameage, X),$
$attribute\_value(X, number, 30),$
$attribute(sameage, number, P_1),$
$class\_object(P_1, 30),$
$attribute\_value(X, shared\_by, Y),$
$attribute(sameage, shared\_by, P_2),$
$class\_object(P_2, Y).$

Appendix E

Transformation of Sample Queires (II)

(Query 1.)? $class(X)$,
$class\_object(X, sally)$,
$attribute\_value(sally, L, Y)$,
$attribute(X, L, P_1)$,
$class\_object(P_1, Y)$.

(Query 2.)? $class(student)$,
$attribute(student, L, Y)$.

(Query 3.)? $class(X)$,
$class(Y)$,
$isa(X, Y)$.

(Query 4.)? $class\_set(employee, X)$.

(Query 5.)? $class\_set(student, X)$.

(Query 6.)? $class\_set(ymage, X)$.

(Query 7.)? $class\_set(gender, X)$.