THE UNIVERSITY OF CALGARY

NLO: A DEDUCTIVE OBJECT BASE LANGUAGE

by

Mengchi Liu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JULY, 1992

ISBN 0-315-79194-2

Name MENG CHI LIU

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE — SUBJECT TERM

0984 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics ... 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

_____ ☐☐☐ U·M·I

SUJET — CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS
Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION
Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE
Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE
Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES
Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-âtre social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES
Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE
Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT
Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hôpitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES
Sciences Pures

Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÇres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Électronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie

Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et ther modynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systâmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE
Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

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 *"NLO: A Deductive Object Base Language"* submitted by Mengchi Liu in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dr. Brian Gaines, Supervisor
Department of Computer Science

Dr. John G. Cleary
Department of Computer Science

Dr. Mildred L. G. Shaw
Department of Computer Science

Dr. Douglas H. Norrie
Department of Mechanical Engineering

Dr. Jiawei Han
Department of Computer Science
Simon Fraser University

Date July 24, 1992

Abstract

Deductive databases result from the integration of relational database and logic programming techniques. This thesis analyzes two significant problems inherent in this integration: namely complex object modeling and higher-order features. It also gives a critical analysis of related work in logic programming and deductive databases, such as extended logic terms which can represent the existence and complex intensional structure of objects, and extended deductive database languages which incorporate higher-order features.

A novel deductive database language called NLO (Natural Logic for Objects) is proposed. It is based on the semantic and object-oriented data models, extended logic term approaches and extended deductive database languages.

The major original contributions of the research presented here are threefold. First, the language NLO is defined with enough expressive power to solve the above two problems. Second, the language is given a firm logical foundation. This includes extended Herbrand models, the model intersection property of definite programs, fix-point, least and minimal models, stratification, perfect models, and precisely defined semantics of NLO programs. Third, a transformation algorithm is given which converts NLO programs and queries into semantically equivalent Prolog programs and queries. Together these imply that NLO is not only syntactically expressive and semantically sound, but also fully implementable in practice.

Acknowledgements

I am indebted to numerous people, without them this thesis would not have been possible.

First, I would like to express my sincere thanks to Dr. Brian Gaines, my supervisor for many helpful discussions which helps me to formalize my thoughts. His assistance, encouragement, valuable suggestions, and wide-ranging knowledge have made this thesis possible.

I would also like to extend special sincere thanks to Dr. John Cleary, for his valuable criticisms, guidance, suggestions, and financial support throughout this research project. His insight and knowledge were fundamental to the success of this thesis.

Thanks are also due to my candidacy examination committee members and many others for offering valuable comments, or encouragement regarding the work presented in this thesis: Dr. Mildred Shaw, Dr. K.L Chowdhury in the Department of Mechanical Engineering, Dr. John Heintz in the Department of Philosophy, Dr. Jiawei Han at Simon Fraser University, Dr. Paul Kwok, and Dr. Lisa Higham.

Thanks goes to Dr. Michael Kifer at SUNY in USA for helpful discussions during DOOD' 91 in Munich and for sending technical reports. His work on F-logic has significantly influenced this research. Thanks also to Dr. Serge Abiteboul, INRIA in France, Dr. Andreas Heuer, TU Clausthal in Germany, for sending me some technical reports which are related to this research. Thanks also to Dr. Ravi Krishnamurthy and Dr. C. Zaniolo, MCC in USA, for answering my questions about L2 and LDL by emails.

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. I simply would not have done it without her unconditional love and support through my periods of depressions. Thanks also to my little handsome son Robert for putting up with the time away from him.

Finally, I would like to thank my parents, Jingfu Liu and Qingshu Wang for putting up with far too much from me for four years and for their understanding and encouragement began long before my work on this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Databases and logic programming have been two independently developed fields in computer science in the last two decades.

Database technology has evolved in order to efficiently organize, manage and maintain a large amount of data on secondary storage. This led to the development of several basic data 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 provides the conceptual basis for thinking about applications and provides a formal basis for developing and using the database systems. The relational data model was developed as a simplification of more complex, machine-oriented hierarchical and network models, to enable set-oriented, non-procedural data manipulation.

Logic Programming began in the early seventies as a direct outgrowth of earlier work in automated theorem proving and artificial intelligence. Logic programming is based on first-order logic, formalized in terms of proof theory and model theory. Proof theory provides formal specifications for correct reasoning with premises, while model theory analyses how general assertions may be interpreted with respect to a collection of specific facts. First-order logic was not used in programming actual applications until the introduction of Prolog, a language for PROgramming in LOGic.

Prolog uses a restricted form of more general theorem proving techniques to provide efficiency and programmability.

Throughout the seventies and early eighties, the use of both Prolog and relational databases has become widespread, not only in academic or scientific environments, but also in the commercial world [CGT90].

In recent years, there has been a growing interest in the integration of logic programming and relational databases to generate a new type of systems, called deductive databases, which use logic programming to make deductions about the contents of a relational database. This integration combines the benefits of these two approaches, such as representational and operational uniformity, ease of use, deductive power, firm theoretical basis, and efficient secondary storage access.

Unfortunately, current deductive databases are quite limited in their expressive power. They cannot support complex object modeling in a direct and natural way, although this is a common requirement of advanced database applications [Mai86, KL89, LR89]. They also cannot support higher-order features such as schema and sets in a uniform way [KN88]. The problems result from the use of inexpressive flat structures in the underlying relational data model and logic programming languages.

As a reaction to the lack of expressiveness, in relational databases, logic programming, and correspondingly in deductive databases, a number of attempts have been made to increase the expressive power.

To improve the relational data model, a number of new approaches called semantic and object-oriented data models have emerged during the last two decades. These aim to provide increased expressiveness to the user and incorporate a richer set of

semantics into the database. Examples are RM/T [Cod79], FDM [Shi79], TAXIS [MBW80], SDM [HM81], SHM+ [Bro84], Galileo [ACO85], SAM* [Su86], Gemstone [MSOP86], IFO [AH87], Orion [KBC+87], FAD [DKV88], O_2 [LR89], Vbase [And91], etc. They provide a number of powerful data modeling manipulation concepts for complex object modeling including object identity, object properties, types and inheritance.

To improve the expressiveness of logic programming, some extensions have been made by using extended logical terms with internal structures, such as LOGIN [AKN86], O-Logic [Mai86], Revised O-Logic [KW89], and F-logic [KL89, KLW90]. To improve the expressiveness of deductive database languages, LDL [TZ86], COL [AG88], IQL [AK89], L2 [KN88], etc. have been proposed which incorporate certain higher-order features.

Semantic and object-oriented data models are quite expressive. However, the proposed extended logic term approaches and extended deductive database languages are not general enough to capture the most important ideas of these models.

This thesis proposes a deductive database language called NLO (Natural Logic for Objects), based on the extended logic terms approaches and extended deductive database languages. It is a natural generalization of semantic and object-oriented data models. It has expressive and deductive powers that can naturally represent and manipulate complex objects and desired higher-order features in a uniform way.

This thesis also investigates the syntactic and semantic properties of NLO programs. These include extended Herbrand models, the model intersection property of definite programs, fixpoint, least and minimal models, stratification, perfect models,

and precisely defined semantics of NLO programs.

1.1 Organization of Thesis

In order to make this thesis as self-contained as possible, Chapter 2 first introduces relational databases and logic programming. Then it describes deductive databases based on these two approaches.

Chapter 3 analyzes the problems of deductive database design. It discusses the requirements of many significant database applications and shows why the relational data model and logic programming languages cannot satisfy these requirements.

Chapter 4 gives a critical analysis of related work. This gives the motivation for the deductive database language NLO.

Chapter 5 introduces NLO by examples and shows its advantages over other approaches.

Chapter 6 presents the syntax and semantics of NLO.

Chapter 7 focuses on the extended Herbrand models and discusses the syntactic and semantic properties as well as the precisely defined semantics of NLO programs.

Chapter 8 presents a formal transformation algorithm which converts NLO programs and queries into semantically equivalent Prolog programs and queries.

Finally, Chapter 9 summarizes the research and discusses potential topics for future research. Table 9.2 compares NLO with other approaches.

Chapter 2

Background

This chapter provides an introduction to the major concepts of relational databases and logic programming which will be used in the following chapters.

2.1 Relational Databases

The kernel of the relational database technology is the relational data model. The relational data model, although not the first data model used in database management systems, has grown in importance since its exposition by Codd in 1970. The most important reason for the model's popularity is its simplicity and formality. It takes over the complex, machine-oriented hierarchical and network models to enable powerful, set-oriented, declarative, i.e. non-procedural, data query or manipulation.

2.1.1 The Relational Data Model

In the relational model, data are organized using relations which are defined as follows. A *domain* is just a 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 any subset of the Cartesian product of one or more domains. The members of a relation are called *tuples*. Note that a relation is a set, therefore tuples in a relation are distinct, and

the order of tuples is irrelevant. 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 when discussing relations to represent a relation as a table and name it by a relation name in which each row is a tuple and each column is often given a name called its *attribute.* In this view, a tuple is a list of values and a relation is a *set of lists.* Attribute values of a tuple are determined by their positions in the tuple. However, relations can also be viewed as a *set of mappings* from the attribute names to values in the domains of the attributes. In this view, attribute values of a relation are determined by their attribute names rather than positions.

Chapter 3 will show that current deductive databases can only take the first view, i.e., relations as sets of lists rather than sets of mappings, which means that attribute values of a tuple are determined only by their positions in the tuple. This is certainly inconvenient to the user.

A minimal subset of the attributes of a relation whose values uniquely identify the tuples of the relation is called a *key* of the relation. It is possible for a relation to have more than one key. In this case, it is customary to designate one as the *primary key.* The ordered set of attribute names for a relation is called the schema of this relation. If a relation is named by **REL**, its relation schema has attributes A_1, A_2,...,A_n, and A_1, ..., A_m is a primary key, then the relation schema is written as $\mathbf{REL}(\mathbf{A_1}, ..., \mathbf{A_m}, ..., A_n)$. The specific relation is said to be an *instance* of the relation schema.

Not all possible 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. Therefore a set of constraints, referred to as *integrity constraints*, is introduced to be associated with relation schemas to ensure that the database meets the intended semantics. There are two major kinds of integrity constraints: type constraints, which require the attribute values of relations to belong to specified domains, and dependency constraints, which express structural properties of relations.

Often a subset of the attributes of one relation will correspond to a key of another relation so that different relations can be *implicitly* related. It is called a *foreign key.* A foreign key need not be (and often is not) a key of its own relation. Relations are related normally through foreign keys.

NAME	PHONE	DEPT
Bob	6124	Mathematics
Henry	3620	Philosophy
John	5016	Physics
Jenny	6017	Mathematics
Smith	9015	Physics
Sally	3105	Mathematics

EMPLOYEE

DEPT	LOCATION
Philosophy	Building 3
Mathematics	Building 1
Physics	Building 2
Chemistry	Building 5

DEPARTMENT

Figure 2.1 Two Example Relations.

Example 2.1 Figure 2.1 shows two relations. The relation EMPLOYEE has attributes NAME, PHONE, and DEPT where NAME is a primary key and DEPT is a foreign key. The schema of EMPLOYEE is EMPLOYEE(**NAME**, PHONE, DEPT). The relation DEPARTMENT has attributes DEPT and LOCATION where

DEPT is a primary key. The schema of DEPARTMENT is DEPT(**NAME**, LOCATION). Here an integrity constraint requires that the values of attribute DEPT in the relation EMPLOYEE should be the values of attribute DEPT in the relation DEPARTMENT. □

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

2.1.2 Relational Query Languages

Associated with the relational data model, there are two kinds of relational query or manipulation languages which can be used to express queries about relations in a relational database:

Relational algebra is defined through several operators that apply to relations and produce other relations. Queries of relational algebra naturally suggests some order in which operations can be applied to the database. However, queries are often transformed equivalently into some optimized forms so that they can be processed in a more efficient way.

Relational Calculus expresses a query by means of a first-order logic formula on the tuples or domains of the database; the result of the query is also a relation that satisfies the formula. The relational calculus is a kind of pure declarative query language, because the query expression in the calculus does not suggest a method

for computing the answer.

These two kinds of languages have been proved to be equivalent in their expressive power [Ull88]. Relational algebra seems to be inherent in the relational model. This thesis considers relational algebra only.

Relational algebra consists of five basic operators: selection, projection, Cartesian product, union, and difference, each of which applies to relations, yielding a new relation as a result.

- *Selection*: Given a relation R and a collection of conditions P over the relation, the selection operation, denoted by $\sigma_P(R)$, produces a relation with the same schema as that of R, whose tuples are in R and satisfy the given conditions.

- *Projection*: Given a relation R and a subset of attribute names A of R, the projection operation, denoted by $\pi_A(R)$, produces a relation which consists of the specified columns of the given relation, and eliminates duplicates from the results. The schema of the result relation is A.

- *Cartesian Product*: Given two relations R and S of arity r and s respectively, the Cartesian product operation, denoted by $R \times S$, is a relation of arity $r + s$, whose tuples are formed by all the possible concatenations of tuples of R and tuples of S.

- *Union*: Given two relations R and S which have identical schema, the union operation, denoted by $R \cup S$, produces a relation with the same schema, whose tuples are in R or S or both.

- *Difference*: Given two relations R and S which have identical schema, the difference operation, denoted by $R - S$, produces a relation with the same schema, whose tuples are in R but not in S.

The most often used operation is called natural join or join which can be derived from the above basic operations:

- (*Natural*) *Join*: Given two relations R and S, the natural join operation, denoted by $R \bowtie S$, 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.

The join operation is used to draw *explicit* relationships between different relations via common attributes such as foreign keys. However the natural join operator is also quite time consuming. A lot of efforts have been invested to improve its performance [Ull88].

NAME	DEPT
Bob	Mathematics
Jenny	Mathematics
Sally	Mathematics

Answers to a)

LOCATION
Building 3

Answer to b)

Figure 2.2 Answers to Queries

Example 2.2 In order to get the names and phone numbers of employees in the Department of Mathematics and the location of Henry's department from the two

relations in Figure 2.1, we can use

a) $\pi_{\text{NAME,PHONE}}(\sigma_{\text{DEPT=Mathematics}}\text{EMPLOYEE})$

b) $\pi_{\text{LOCATION}}(\sigma_{\text{NAME=Henry}}(\text{EMPLOYEE} \bowtie \text{DEPT}))$

respectively. The answers to these queries are shown in Figure 2.2. □

2.2 Logic Programming

Logic programming is based on mathematical logic which is the study of the relationship between formally expressed premises and conclusions. For example, if we assume that *Art* is a parent of *Bob* and that a parent is an ancestor, then we can infer that *Art* is a ancestor of *Bob*. The first two sentences imply the conclusion. In logic programming the programmer encodes in a logic program a set of premises about the application and the machine applies *rules of inference* to known premises and derives conclusions that are logically implied by those premises. Subsequent applications allow a program to derive further conclusions.

Most logic programming is based on clause form, which is a restricted form of first-order logic. The most general kind of program clause usually considered is

$$A \leftarrow L_1, ..., L_n. \tag{1}$$

where A is an *atom* and $L_1, ..., L_n$ are *literals* [ABW88, Llo87, She88]. An atom is of the form $p(t_1, ..., t_m)$ where p is an m-ary *predicate* symbol and $t_1, ..., t_m$ are *terms*. A literal is either an atom or the *negation* of an atom, i.e. of the form $\neg p(t_1, ..., t_m)$. 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

atom A in the rule is called the *head* or *conclusion*, and L_1 through L_n form the *body* or *conditions* of the program clause.

A program clause $A \leftarrow L_1, ..., L_n$, is a universally quantified first-order formula $\forall X_1 ... \forall X_m (A \vee \neg(L_1 \wedge ... \wedge L_n))$ where $X_1, ..., X_m$ are all variables appearing in the program clause. But variables appearing only in the body may be equivalently regarded as quantified existentially within the body, while other variables are universally quantified over the entire clause, that is, $\forall X_1 ... \forall X_l (A \vee \neg \exists X_{l+1} ... \exists X_m (L_1 \wedge ... \wedge L_n))$ where $X_{l+1}, ..., X_m$ are variables appearing only in the body.

There exist three types of program clauses: *facts, rules*, and *queries* (or *goals*) [ABW88, Llo87, GM92]. A fact is a program clause with an empty set of conditions. A rule is a program clause with non-empty head and conditions. A query is a program clause with an empty head and is normally represented as $?- L_1, .., L_n$.

A *logic program* is a finite set of facts and rules. A logic program is *definite* if it does not have negation in its rules. Otherwise, it is *normal.* Queries may be associated with a logic program to be answered whether or not they can be satisfied by the program.

A term which contains no variables is called a *ground term.* A program clause (a literal) in which no variables appear is called a *ground clause* (*ground literal*).

A *substitution* θ is a finite set of the form $\{X_1/t_1, ..., X_n/t_n\}$, where each X_i is a distinct variable and each t_i is a term distinct from X_i. The substitution is *ground* if every term t_i is ground. Let $\theta = \{X_1/t_1, ..., X_n/t_n\}$ be a substitution and E is either a term, a literal or a program clause, then $E\theta$, the *instance* of E by θ is a term, a literal or a program clause obtained from E by simultaneously replacing each occurrence

of the variable X_i in E by the term t_i.

Example 2.3 An example of a definite logic program P_1 is as follows.

(1). $parent(art, bob) \leftarrow$.
(2). $parent(mary, art) \leftarrow$.
(3). $ancestor(X, Y) \leftarrow parent(X, Y)$.
(4). $ancestor(X, Y) \leftarrow parent(X, Z), ancestor(Z, Y)$.

This program has two ground facts and two rules. The first fact says "*Art* is a parent of *Bob*." The second fact says "*Mary* is a parent of *Art*." The first rule says "for all X and Y, if X is a parent of Y, then X is an ancestor of Y." The second rule says "for all X, Y, and Z, if X is a parent of Z and Z is an ancestor of Y, then X is an ancestor of Y," or "for all X and Y, X is an ancestor of Y if there exists a Z such that X is an ancestor of Z, and Z is a parent of Y." The query ?– $ancestor(X, bob)$, can be used to ask if there exists an X such that X is a parent of *Bob* based on the program. □

Example 2.4 An example of a normal logic program P_2 is as follows.

(1). $inWater(peter) \leftarrow$.
(2). $canSwim(phil) \leftarrow$.
(3). $sink(X) \leftarrow inWater(X), \neg canSwim(X)$.
(4). $happy(X) \leftarrow inWater(X), \neg sink(Y)$.

This program has two facts and two rules. The first fact says "*Peter* is in the water." The second fact says "*Phil* can swim." The first rule says "for all X, if X is in the water and X cannot swim, then X sinks." The second rule says "for all X, if X is in the water and X does not sink, then X is happy." The query ?– $happy(peter)$, can be used to ask if *Peter* is happy or not based on the program. □

What a logic program can compute or what the intended semantics of a logic

program is, and how the output of a logic program can be computed; and when given queries, what the answers to the queries should be and how these answers can be found are the contents of two complementary aspects of logic programming. One is called *model theory*, which deals with *semantics*, and the other is called *proof theory* which deals with *syntax* [ABW88, CGT90, Llo87, She88].

Model theory is, in general, considered as the more intuitive or declarative approach to the meaning of a program. Proof theory, on the other hand, often provides more efficient computational methods.

2.2.1 Model Theory

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

In model theory, we are concerned with interpretations and models. An *interpretation* of a set of program clauses consists of a nonempty set D, called the *domain* of the interpretation, over which the variables range, and an assignment to each constant of some fixed element in D, to each n-ary function symbol of a mapping from D^n into D, to each n-place predicate symbol of an n-place relation on D^n. An interpretation thus specifies a meaning for each symbol in a set of program clauses.

A *variable assignment* assigns each variable an element in the domain.

Given an interpretation I with a domain D, and a variable assignment V, the *truth value*, true or false, of a program clause, with respect to I and V, can be obtained as follows. If p' is the relation assigned to an n-place predicate symbol p, then the positive literal $p(t_1, ..., t_m)$ evaluates to true if $< t'_1, ..., t'_m > \in p'$, where $t'_1, ..., t'_m$ are the term assignments of $t_1, ..., t_m$ with respect to I and V; otherwise it

evaluates to false. A negative literal $\neg p$ evaluates to true if p is false; otherwise it evaluates to false. If both p_1 and p_2 are true, then p_1, p_2 evaluates to true; otherwise it evaluates to false. If either p_1 is true or p_2 is false, then $p_1 \leftarrow p_2$ evaluates to true; otherwise it evaluates to false.

Let ψ be a program clause, I an interpretation and V a variable assignment. If ψ evaluates to true with respect to I and V, then we say that ψ is *satisfied* by I and V, denoted by $I \models V(\psi)$.

For a program clause ψ, all variables in it are universally quantified. So given an interpretation I, ψ is true under I if and only if it is true with respect to every possible variable assignment V and I. In other words, ψ is *satisfied* by I, denoted by $I \models \psi$ if and only if for every possible variable assignment V, $I \models V(\psi)$.

A *model* of a logic program P is an interpretation which satisfies all facts and rules of P. A program clause C is said to be a *logical consequence* of a logic program P, denoted by $P \models C$, if and only if for every model M of P, $M \models C$.

In first-order logic, given a logic program P without queries, we are concerned with all its logical consequences, that is $\{C \mid P \models C\}$. Given a logic program P with a query ?- $L_1, ..., L_n$, we are concerned whether or not the query can be satisfied by the program, that is, whether or not there are ground substitution θ such that $P \models L_i\theta$, $1 \leq i \leq n$. Therefore, we consider all possible interpretations and models of the program with or without queries. This declarative view of a logic program may be the ultimate ideal of logic programming. But it is not the way current logic programming makes [She88]. There are two reasons. One is that we do not intend to consider all possible interpretations and models of a logic program

in logic programming. For the fact $parent(art, bob)$ in Example 2.3, the constants *art*, *bob* and predicate *parent* are intended to be interpreted as persons *Art*, *Bob* and the parent relation between persons respectively, rather than something else. Another reason is that in a logic program only the true facts are asserted and the rules can only infer true facts, because the volume of false facts is usually much greater than the volume of true facts. Negative facts are neither asserted nor can be inferred. In this sense the program is incomplete. But it is very convenient to the user and it results in efficient implementations [She88]. For example, we do not assert that $\neg parent(bob, art)$, $\neg parent(bob, mary)$, $\neg parent(art, mary)$, etc. in Example 2.3 and $\neg inWater(phil)$, $\neg canSwim(peter)$ etc. in Example 2.4. These practical reasons prohibit us to focus on logical consequences of a logic program because what we obtain from a program may not be logic consequences of the program.

Therefore, in logic programming, we are concerned with the intended interpretations and models of a logic program, that is, so-called Herbrand interpretations and Herbrand models. In some simple case, that is, for a definite logic program, the intended meaning of a program clause in a logic program is also the logical consequence of the program.

Herbrand Interpretations and Models

In logic programming, only a special kind of interpretation is of interest, rather than general interpretations. This special kind of interpretation gives the intended semantics of logic programs. These are the so-called *Herbrand interpretations.*

Given a program P, the domain U of a Herbrand interpretation includes all constant symbols in P. Each constant symbol is assigned to itself. Every n-ary

function symbol is assigned to a mapping U^n to U denoted by itself too. For example, if f is a unary function and $a, b, c, ...$ are constants, then f is interpreted by $\{f(a), f(b), f(c), ...\}$. Therefore, all Herbrand interpretations have the same domain and assign the same meaning to the constant symbols and function symbols in the program. Each n-place predicate is interpreted as mapping from U^n to the set $\{true, false\}$. Thus, Herbrand interpretations differ from one another only in the interpretations of predicate symbols. This means that Herbrand interpretations differ from one another only in the truth value of ground facts.

Note that in Herbrand interpretations, the so-called freeness axioms are assumed, that is, the axioms

$$f(X_1, ..., X_n) \neq g(Y_1, ..., Y_m) \tag{2}$$

for each pair f, g of distinct functions

$$f(X_1, ..., X_n) = f(Y_1, ..., Y_m) \rightarrow X_1 = Y_1 \wedge ..., \wedge X_n = Y_n \tag{3}$$

for each function f, and

$$t(X) \neq X \tag{4}$$

for each term $t(X)$ different from X in which X occurs [She88]. For this reason, function symbols are usually called uninterpreted or freely interpreted because they have no a priori meaning. As a result, $husband(mary) = john$ cannot be directly expressed.

Normally, a Herbrand interpretation is represented as a set which includes all the ground facts that are interpreted true. The set of all possible ground facts is called the *Herbrand Base*. So, each different Herbrand interpretation corresponds to a different subset of the Herbrand Base.

Let H be a Herbrand interpretation of the program P. Then for a ground fact F, $H \models F$ if and only if $F \in H$; $H \models \neg F$ if and only if not $H \models F$, that is, $F \notin H$. For a non-ground fact F, $H \models F$ if and only if for each ground substitution θ, $F\theta \in H$; $H \models \neg F$ if and only if not $H \models F$, that is, there exists a ground substitution θ' such that $F\theta \notin H$. For a rule r of the form $A \leftarrow L_1, ..., L_n$, $H \models r$ if and only if for each ground substitution θ, whenever $H \models L_1\theta$, ..., $H \models L_n\theta$, then $H \models L\theta$.

A *Herbrand model* of a logic program P is a Herbrand interpretation which is a model for P. Every logic program has at least one Herbrand model which is the Herbrand Base. However, the Herbrand Base is usually not used as the intended semantics as it contains all possible ground facts. The usual approach is to look for small Herbrand models in order to make the least number of assumptions concerning what is true in a logic program [GM92].

Let P be a logic program, ?- $L_1, .., L_n$ a query, and M a Herbrand model of P. An answer to the query ?- $L_1, ..., L_n$ under M is either *no* if there does not exist a ground substitution θ such that $M \models L_i\theta, 1 \leq i \leq n$, or *yes* if there exists a ground substitution θ such that $M \models L_i\theta, 1 \leq i \leq n$. In the later case, an answer is normally represented by a ground substitution.

Example 2.5 For the definite logic program P_1 in Example 2.3, following are some possible Herbrand models:

$M_1 = \{parent(art, bob), parent(mary, art),$
$ancestor(art, bob), ancestor(mary, art), ancestor(mary, bob)\}$
$M_2 = \{parent(art, bob), parent(mary, art), parent(art, mary),$
$ancestor(art, bob), ancestor(mary, art), ancestor(art, mary)$
$ancestor(mary, bob), ancestor(art, art), ancestor(mary, mary)\}$
$M_3 = \{parent(art, bob), parent(mary, art), parent(bob, mary),$
$ancestor(art, bob), ancestor(mary, art), ancestor(bob, mary)$

$$ancestor(mary, bob), ancestor(bob, art), ancestor(art, mary)\}$$
$$M_4 = \{parent(art, bob), parent(mary, art), parent(bob, art),$$
$$ancestor(art, bob), ancestor(mary, art), ancestor(bob, art)$$
$$ancestor(mary, bob), ancestor(bob, bob), ancestor(art, art)\}$$

For the query ?- $ancestor(X, bob)$, different models give different answers. Under M_1 and M_2, the answers to the query are $\{X/art\}$, and $\{X/mary\}$. Under M_3 and M_4, the answers to the query are $\{X/art\}$, $\{X/mary\}$, and $\{X/bob\}$. □

Example 2.6 For the normal logic program P_2 in Example 2.4, following are some possible Herbrand models:

$$M_1 = \{inWater(peter), canSwim(phil), sink(peter)\}$$
$$M_2 = \{inWater(peter), canSwim(phil), canSwim(peter), happy(peter)\}$$
$$M_3 = \{inWater(peter), canSwim(phil), canSwim(peter), sink(peter)\}$$
$$M_4 = \{inWater(peter), inWater(phil), canSwim(peter), canSwim(phil),$$
$$sink(peter), sink(phil), happy(peter), happy(phil), \}$$

For the query ?- $happy(peter)$, under M_2 and M_4, the answer is yes; while under M_1 and M_3, the answer is no. The models M_3 and M_4 contains information contradictory to what is intended, such as $canSwim(peter)$, $sink(peter)$, and $sink(peter$, $happy(peter)$. □

Since there are many Herbrand models for a given logic program, how can we define the semantics? Two kinds of Herbrand models are of special interests: least models and minimal models. If a model M of a program P is a subset of every model of P, then M is called a *least* model of P. If M is a model of P such that no model of P is its proper subset, then it is called a *minimal* model of P [ABW88]. Thus, a least model is a minimal model, but not necessarily conversely. The semantics of a definite program is given by its least model. For a normal program, there may be

more than one minimal model, its semantics is given by one of its minimal models called perfect model which is preferable to other minimal models.

Least Model Semantics for Definite Program

A definite program P, that is, program without negation, has the following properties [Llo87, ABW88]:

1. The intersection of its Herbrand models is itself a Herbrand model.
2. It has a unique least Herbrand model M_P.
3. Every ground fact in M_P is a logical consequence of P.

Property 1 is usually called the *model intersection property* of definite programs. For the property 2, the least Herbrand model M_P is just the intersection of all possible Herbrand models. Property 3 says that every ground fact are deducible from P. Therefore, the declarative semantics of a definite program P is given by its least Herbrand model M_P [SS86, CGT90]. This M_P tells us exactly what the program can compute, answer or prove. The answers to a query ?- $L_1, ..., L_n$ associated with a definite program P, are all ground substitutions θ such that $M_P \models L_i\theta$, $1 \leq i \leq n$.

Example 2.7 Consider the Herbrand models in Example 2.5, $M_1 \cap M_2$, $M_2 \cap M_3$, $M_1 \cap M_2 \cap M_3 \cap M_4$, etc. are models and M_1 is the least Herbrand model which gives the semantics of the definite program P_1 in Example 2.3. Note non-least Herbrand models, such as M_2, include facts which are not deducible from the program, such as $parent(bob, art)$. The answers to the query ?- $ancestor(X, bob)$, are $\{X/art\}$ and $\{X/mary\}$. □

Perfect Model Semantics for Normal Program

For a program P with negation, the intersection of Herbrand models does not need to be a model of P and P may have no least model but several minimal models.

Example 2.8 Consider the normal program P_3:

$$even(0) \leftarrow \neg odd(0)$$

It has three models $\{even(0)\}$, $\{odd(0)\}$, $\{even(0), odd(0)\}$ but their intersection is the empty set ϕ which is not a model. The models $\{even(0)\}$ and $\{odd(0)\}$ are minimal, but there exists no least model. The non-minimal model $\{even(0), odd(0)\}$ contains information which is contradictory to what is intended and thus cannot be taken as the intended semantics of the program. □

Example 2.9 Consider the Herbrand models in Example 2.6 for the normal program P_2 in Example 2.4, $M_1 \cap M_2$, $M_2 \cap M_3$ are not models of P_2. Both M_1 and M_2 are minimal models of P_2. The contradictory information only appears into non-minimal models. □

As the above examples show, non-minimal models, such as $\{even(0), odd(0)\}$ in Example 2.8 and M_3 and M_4 in Example 2.6 may include facts which are contradictory to what is intended. This suggests the semantics of a normal program should be given by one of it minimal models. Since several minimal Herbrand models exist for a normal program, as the above examples show, which one should be considered as its intended semantics? Of course one would like to select the most "natural" and "intuitive" one from the different minimal Herbrand models, which is supported by the program. For the program P_3, the fact $odd(0)$ cannot be inferred and should be

taken as false. This is intended for the negated facts. Then the fact $even(0)$ can be inferred and thus the minimal model $\{even(0)\}$ is supported. This model can be considered more natural and intuitive than the other minimal model $\{odd(0)\}$. Similarly, for the program P_2, M_1 is more natural and intuitive than M_2 in Example 2.6.

Unfortunately, not every normal program has a minimal model which can be considered more natural and intuitive than any others.

Example 2.10 Consider another normal program P_4:

(1). $female(mary) \leftarrow \neg male(mary)$.
(2). $male(mary) \leftarrow \neg female(mary)$.

This program has two minimal models: $\{female(mary)\}$ and $\{male(mary)\}$, If $female(mary)$ is taken as false, then the model $\{male(mary)\}$ is obtained. If $male(mary)$ is taken as false, then the model $\{female(mary)\}$ is obtained. However, it is certainly not clear whether $female(mary)$ or $male(mary)$ should be taken as false, based on the program itself. □

The program P_4 shows that when recursion is combined with negation, the semantics of the program is problematic. Therefore, a syntactic constraint called *stratification* is added to normal programs which disallows the combination of recursion with negation that may obscure the semantics of the program.

If a predicate p is in the head and a negation of a predicate q is in the body, then p is said to depend-on q, denoted by $>$. This depends-on relation is transitive. A program is called *stratified* if there is no predicate p in the program such that p depends-on q, q also depends-on p. If a program P is stratified, then P can be

partitioned into a set of stratums $\{P_1, ..., P_n\}$, and $P = P_1 \cup ... \cup P_n$, such that if a predicate p is in the head of a rule in P_i, then every predicate q which p depends on can only be a head of a rule in $\cup_{j<i} P_j$.

Example 2.11 The programs P_1 in Example 2.3, P_2 in Example 2.4 and P_3 in Example 2.8 are stratified. But the program P_4 in Example 2.8 is not stratified, while The programs P_1 and P_3 just have one stratum. The programs P_2 can be partitioned as follows.

$$\begin{aligned} P_2 = &\{inWater(peter) \leftarrow ., canSwim(phil) \leftarrow .\} \cup \\ &\{sink(X) \leftarrow inWater(X), \neg canSwim(X).\} \cup \\ &\{happy(X) \leftarrow inWater(X), \neg sink(X).\} \end{aligned}$$

□

Suppose that M and N are two distinct models of a stratified normal program P. N is called *preferable* to M, if for every ground fact A in $N - M$ there exists a ground fact B in $M - N$, such that A depends on B. A model M is called *perfect* if there are no models preferable to M. Every perfect model is minimal [Prz88]. Therefore, this unique distinguished minimal Herbrand model can be selected in a very natural and intuitive way as the intended semantics of the program.

Intuitively and naturally, a fact B is true if it is asserted or is deducible from some rule, otherwise it is intended to be false and its negation is then true. But if such B is true in some model M, then the model N results from M by removing B (making it false) and adding A which depends on the negation of B does reflect the intended semantics.

Note that the least Herbrand model of a definite logic program is a special perfect model of the program.

The answers to the query ?- $L_1, ..., L_n$ associated with a normal program P, are all ground substitutions θ such that $M_P \models L_i\theta, 1 \leq i \leq n$.

Example 2.12 Consider Example 2.8 again. For the program P_3, since $\{even(0)\} - \{odd(0)\} = \{even(0)\}$, $\{odd(0)\} - \{even(0)\} = \{odd(0)\}$ and *even* depends on *odd*, $\{even(0)\}$ is preferable to $\{odd(0)\}$ and therefore $\{even(0)\}$ is the perfect model of P_3. For another example, consider Example 2.6 again. Since $M_1 - M_2 = \{sink(peter)\}$, $M_2 - M_1 = \{canSwim(peter), happy(peter)\}$ and *sink* depends on *canSwim*. So M_1 is preferable to M_2 and M_1 is also the perfect model of P_2. For the query ?- *happy*(*peter*), the answer is *no* based on the perfect model M_1. □

2.2.2 Proof Theory

By model theory, we know what a program computes or proves. When given a query, we know what the answers to the query should be, based on the intended model of the program. However, this does not lead to any constructive method for evaluating the program and computing the answers to the query. Such methods are the content of the proof theory.

In proof theory, we are concerned with what can be derived by the application of some given rules of inference from a given program.

There are two kinds of approaches for evaluating programs which are called *forward chaining* or *bottom-up* computation and *backward chaining* or *top-down* computation respectively. Bottom-up computation is better for computing all ground facts that a program can prove, or all answers to a given query; while top-down computation is better for computing one answer to a specific query at a time.

Top-Down Computation for Definite Programs

In Top-down computation, an inference rule called *resolution* is the most extensively studied and used to deduce new program clauses.

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

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.

Example 2.13 Given two program clauses as follows.

(1). $ancestor(X, art) \leftarrow parent(X, art)$.
(2). $ancestor(X, bob) \leftarrow ancestor(X, Y), parent(Y, bob)$.

Using the resolution rule to unify $ancestor(X, art)$ in (1) and $ancestor(X, Y)$ in (2), we get

(3). $ancestor(X, bob) \leftarrow parent(X, art), parent(art, bob)$. □

Resolution is used mostly to carry out refutation proofs: Given a definite program P and a query ?- $L_1, .., L_n$, in order to prove $L_1, .., L_n$, is deducible from P, written as $P \vdash L_1, .., L_n$, we can try to show that P and $\neg(L_1, .., L_n)$, are not simultaneously satisfiable. If we can derive the empty clause ϕ, that is, a clause with no conditions and no conclusions, then P and $\neg(L_1, .., L_n)$, cannot simultaneously be satisfiable,

thus we have proved the query $P \vdash L_1, .., L_n$. When a query contains variables and the empty clause can be derived, then we have proved the query, and furthermore found a desired answer from the substitution.

Example 2.14 Consider the definite program P_1 in Example 2.3 and the query ?- $ancestor(X, bob)$. Using resolution we can derive the empty clause and get one answer $\{X/art\}$ as follows.

?- $ancestor(X, bob)$	query
?- $parent(Y, bob)$.	Clause (3) in P_1.
?- ϕ with $\{X/art\}$	Clause (1) in P_1.

The following derivations show how to get the second answer $\{X/mary\}$ for the query.

?- $ancestor(X, bob)$	query
?- $parent(Z, bob), ancestor(X, Z)$.	Clause (4) in P_1.
?- $ancestor(X, art)$.	Clause (1) in P_1.
?- $parent(X, art)$.	Clause (3) in P_1.
?- ϕ with $\{X/mary\}$	Clause (2) in P_1.

□

The resolution refutation has following properties. If a definite program P and $\neg(L_1, ..., L_n)$ have a refutation, i.e. $P \vdash L_1, ..., L_n$, and M_P is the least Herbrand model of P, then $M_P \models L_1, ..., L_n$ (and $P \models L_1, ..., L_n$ also). This means that the resolution refutation is *sound* in that any conclusion it draws is guaranteed to be correct with respect to its intended semantics so long as its premises are correct. On the other hand, if $M_P \models L_1, ..., L_n$ (or $P \models L_1, ..., L_n$), then $P \cup \{\neg(L_1, ..., L_n)\}$ has a refutation and hence $P \vdash L_1, ..., L_n$. This means that resolution refutation is also *complete* in that it can derive any correct conclusion from a given program with respect to the intended semantics.

Top-Down Computation for Stratified Normal Programs

The above inference system is very specialized. It cannot deduce negative information. In a program with negations, another inference rule called the *negation as failure rule* is also used to infer negative information. It states that for a normal program P if *not* $P \vdash A$ then infer $\neg A$.

Negation as failure is easily and efficiently implemented by the above resolution proof. Suppose we have a query ?- $\neg A$. The system tries the query ?- A. If ?- A succeeds, then ?- $\neg A$ fails, while if it fails then ?- $\neg A$ succeeds.

Example 2.15 Consider the normal program P_2 in Example 2.4 and the query ?- *happy*(*peter*). Using resolution (R) and negation as failure (N) rules, we can get the answer *no* as follows.

?- *happy*(*peter*)	query
?- *inWater*(*peter*), ¬*sink*(*peter*).	By R on Clause (4) in P_2.
?- ¬*sink*(*peter*).	By R on Clause (1) in P_2.
?- *sink*(*peter*).	Subquery 1.
?- *inWater*(*peter*), ¬*canSwim*(*peter*).	By R on Clause (3) in P_2.
?- ¬*canSwim*(*peter*).	R on Clause (1) in P_2.
?- *canSwim*(*peter*).	Subquery 2.
?- *canSwim*(*peter*) *fails*	By R
?- ¬*canSwim*(*peter*) *succeeds*	By N.
?- *sink*(*peter*) *succeeds*	By R.
?- ¬*sink*(*peter*) *fails*	By N.
?- *happy*(*peter*) *fails*	Answer to the query by R □

Let P be a normal program and M_P its perfect model. Like resolution refutation, we have if $P \vdash L_1, ..., L_n$, then $M_P \models L_1, ..., L_n$ (but not $P \models L_1, .., L_n$). This means that our top-down computation is *sound* in that any conclusion it draws is guaranteed to be correct with respect to its intended semantics so long as its premises

are correct. Also, if $M_P \models L_1, ..., L_n$, then $P \vdash L_1, ..., L_n$. This means that our top-down computation is also *complete* in that it can derive any correct conclusion from a given program with respect to the intended semantics.

The most popular logic programming language Prolog uses this top-down, computation.

Bottom-Up Computation for Definite Programs

Consider a rule r of the form $A \leftarrow A_1, ..., A_n$ and a list of ground facts $g_1, ..., g_n$ If a substitution θ exists such that for each $1 \leq i \leq n$, $A_i\theta = g_i$, then from rule r and the facts $g_1, ..., g_n$, we can infer in one step the ground fact of $A\theta$. The inferred fact may be either a new fact or it may be already known. This inference rule is called *elementary production* [CGT90]. Like resolution, it is a meta-rule, since it is independent of any particular rules in a program.

Based on the elementary production, we can first infer all ground facts because their bodies are empty and are always satisfied. This is normally considered as one step even through it may actually take many steps to finish. Then we can infer new ground facts using rules and inferred ground facts for another step. This process keeps going until we reach a state in which no more new facts can be produced.

Example 2.16 Consider the program P_1 in Example 2.3 again. The facts that can be inferred in step i are shown below in the set S_i, and the process stops when i is 3.

$$
\begin{aligned}
S_1 &= \{parent(art, bob), parent(mary, art)\} \\
S_2 &= \{parent(art, bob), parent(mary, art), \\
&\quad ancestor(art, bob), ancestor(mary, art)\} \\
S_3 &= \{parent(art, bob), parent(mary, art), \\
&\quad ancestor(art, bob), ancestor(mary, art), ancestor(mary, bob)\}
\end{aligned}
$$

Note here that S_3 is exactly the same as the least Herbrand model M_1 in Example 2.7. Since we have inferred $ancestor(art, bob)$, and $ancestor(mary, bob)$, we can conclude that $\{X/art\}$ and $\{X/mary\}$ are the only two correct answers to the query $?- ancestor(X, bob)$. □

The same as the resolution refutation, the inference rule elementary production is also sound and complete with respect to the intended semantics of the program.

The above bottom-up computation for definite programs can be naturally described as a least fixpoint computation based on concepts from the mathematical theory of lattices.

A *lattice* is a set with a partial ordering ($\leq$) relation. For a lattice L and a set $X \subseteq L$, $a \in L$ is called an *upper bound* of X if $x \leq a$ for all $x \in X$. A *least upper bound* of X, $lub(X)$, is unique, if it exists. In a similar but opposite manner the notions of lower bound and greatest lower bound can be defined. A lattice L is *complete* if $lub(X)$ and $glb(X)$ exist for every subset X of L. A complete lattice L has a bottom element which is $glb(L)$ denoted by $\perp$.

Let L be a complete lattice for the following definitions. Let $X \subseteq L$, X is said *directed* if it contains an upper bound for its every finite subset. A mapping $T : L \rightarrow L$ is *monotonic* if $x \leq y$ implies $T(x) \leq T(y)$, and *continuous* if $T(lub(X)) = lub(T(X))$ for every directed subset X of L. An element $a \in L$ is called a *fixpoint* of T if $T(a) = a$. If there exists a fixpoint a of T such that for all fixpoints b of T, $a \leq b$, then a is called a *least fixpoint* of T and is written as $lfp(T)$.

Powers of a monotonic mapping T are defined as follows:

$$T \uparrow 0 = \bot$$
$$T \uparrow (i+1) = T(T \uparrow i)$$
$$T \uparrow \omega = lub\{T \uparrow i \mid i < \omega\}$$

where ω is the first infinite ordinal.

Following two results about lattices will be used to describe fixpoint computation.

(1) If T is monotonic then $lfp(T)$ exists.

(2) If T is continuous then $lfp(T) = T \uparrow \omega$.

Let P be a program and HB be the Herbrand Base. Then 2^{HB} is the set of all possible Herbrand interpretations of P which forms a complete lattice under the partial order of set inclusion $\subseteq$, a mapping $T_P : 2^{HB} \rightarrow 2^{HB}$ is defined as follows. Let I be a Herbrand interpretation. Then $T_P(I) = \{A \in HB \mid A \leftarrow A_1, ..., A_n$ is a ground instance of a clause in P and $\{A_1, ..., A_n\} \subseteq I\}$. So $T_P(I)$ contains all immediate consequences of the rules of P applied to I. If I is a model of P, then we have $T_P(I) \subseteq I$. Since T_P is defined over a complete lattice and it is monotonic, it has a least fixpoint $lfp(T_P)$. Also it is continuous, so $lfp(T_P) = T_P \uparrow \omega$. An interesting result for the least Herbrand model M_P is $M_P = T_P \uparrow \omega$. This means M_P can be bottom-up computed from the empty set which is $T_P \uparrow 0$. $T_P \uparrow 1$ corresponds to the step one of the computation, $T_P \uparrow 2$ corresponds to the step two, ..., $T_P \uparrow \omega$ corresponds to the state in which no more new facts can be produced, see Example 2.16 for detail.

Bottom-Up Computation for Stratified Normal Programs

For a stratified normal program $P = P_1 \cup ... \cup P_n$, the bottom-up computation first computes the least Herbrand model M_1 for the stratum P_1, then the least Herbrand model M_2 for the stratum P_2 based on M_1, ..., finally the least Herbrand model M_n

for the stratum P_n based on M_{n-1}. The negation as failure rule is also used here for negative facts. That is, if a rule in stratum P_i uses a negative literal $\neg A$, then $\neg A$ is true if there is a ground substitution θ such as only if $A\theta \notin M_j, j \leq i$.

Example 2.17 Consider the program P_2 in Example 2.4 again. The facts that can be inferred in stratum i are shown below in the set M_i, and the process stops when i is 2.

$$M_1 = \{inWater(peter), canSwim(phil)\}$$
$$M_2 = \{inWater(peter), canSwim(phil), sink(peter)\}$$

Note here M_2 is exactly the same as the perfect model M_1 in Example 2.12. Since we do not have $happy(peter)$ in M_2 we can conclude that *no* is the correct answer to the query ?– $happy(peter)$. □

Such bottom-up computation is also sound and complete with respect to the intended semantics. It can also be naturally described by a fixpoint computation. However, the T_P operator need not be monotonic for a stratified program. For instance, in P_3 of Example 2.8 with the predicates *even* and *odd*, we have $T_P(\phi) = \{even(o)\}$ and $T_P(\{odd(0)\}) = \phi$. To avoid this problem, cumulative powers of an operator T are used for stratified program as follows:

$$T \uparrow 0(I) = I$$
$$T \uparrow (i+1)(I) = T(T \uparrow i(I)) \cup T \uparrow i(I)$$
$$T \uparrow \omega(I) = \cup\{T \uparrow i(I) \mid i < \omega\}$$

Let P be a stratified program such that $P = P_1 \cup ... \cup P_n$. Then the bottom-up computation can be described as a sequence of least fixpoint computations through the levels $1, ..., n$, via the strata $P_1, ..., P_n$ as follows:

$$M_1 = T_{P_1} \uparrow \omega(\phi)$$
$$M_2 = T_{P_2} \uparrow \omega(M_1)$$
.....
$$M_n = T_{P_n} \uparrow \omega(M_{n-1}) = M_P$$

In other words, we first compute the least fixpoint M_1 corresponding to the first stratum of the program from the empty set. Then we compute the least fixpoint M_2 corresponding to the second stratum of the program based on M_1. Finally the computation terminates with the result $M_n = M_P$, the perfect model of the program.

2.2.3 Summary of Logic Programming

Logic programming is programming by description [GG84]. 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, that is, by saying what is true in terms of clauses which have the requisite declarative semantics. The system will use the rules of inference to choose specific operations to draw conclusions about the application and to answer queries even though these answers are not explicitly recorded in the description. The semantics of program clause logic can be described declaratively as well as operationally.

2.3 Deductive Databases

In recent years, there has been a growing interest in the integration of relational databases and logic programming. The most important reason for such interest is that the integration is not only possible but also beneficial. The relational database

and logic programming techniques have been found to be strongly similar in their representation of data and complementary in their implementations.

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 [TZ86]. They provide the technology for managing large, shared, persistent, and reliable data collections. The control over the execution of query languages is the responsibility of the system which, through query optimization and compilation techniques, ensures efficient performance over a wide range of storage structures. Physical structure changes do not affect the users view of data in the database. 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 are limited compared with those of logic programming languages. Relational query languages are often inadequate to express complete applications, and are thus embedded in traditional programming languages. However, these two kinds of languages are almost always mismatched in their type systems and their programming style, which cause the so-called *impedance mismatch* [Mai87, ZAKB+85] problem between programming and relational query languages.

On the other hand, logic programming offers a general programming language which is a natural and powerful generalization of the relational data model. It can express data, constraints, deductive information and queries in a uniform way. It has no mismatch problem. Query and constraint representation are possible in a

homogeneous formalism and their evaluation requires the same inferencing mechanisms, hence enabling more powerful reasoning about the database contents. The logic programming language Prolog is in fact being used so with great success in varied applications such as symbolic manipulation, rule-based expert systems and natural language parsing [Bra86]. However, Pure Prolog is based on the program clause logic and a sequential execution-control model. Rules are searched and queries 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.

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 [GMN84, Ull88, Rei84]. Any tuple $\langle t_1, ..., t_m \rangle$ of a relation p can be expressed as 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.* The model-theoretical view has contributed to the understanding of the semantics of a databases and answers to queries. The proof-theoretical view enables us to derive information which is not explicitly expressed, 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 clear 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 data model theory in general.

Based on the above comparison, it seems possible and productive to integrate these two approaches and obtain the benefits of both. This integration has resulted in an active field in computer science called *deductive databases* [GMN84].

The advantages of deductive databases can be summarized as follows:

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

(2). Ease of use. The user only needs to specify what should be done declaratively, how to do it is the responsibility of the system. Besides, physical structure changes will not affect users view of data in the database.

(3). Deductive power. By using program clause logic as a database language, it is possible to use well-developed techniques of inference to provide powerful deductive tools.

(4). 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.

(5). Efficient secondary storage access. 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 relational databases and Prolog is obtained by building an interface. The large collection of Prolog facts is managed in secondary storage by using the existing relational database technology. Systems of this kind include EDUCE [Boc86], CGW [CGW86], and NU-Prolog [TL86]. 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 the relational model is oriented towards a set at a time computation.

Tightly-coupled systems, on the other hand, use a logic-based language called Datalog which is a similar to Prolog, but has no function symbols and 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 [TZ86, BNST91] is an example of this kind which extends Datalog by adding set grouping and set enumeration constructors.

Chapter 3

Problems with Basic Deductive Databases

Deductive databases represent the convergence of logic programming and relational databases and combine the benefits of both, such as representational and operational uniformity, ease of use, deductive power, data independence, efficient secondary storage access, etc. However, such basic deductive databases are quite limited in their expressive power. They cannot support complex object modeling in a direct and natural way, although this is a common requirement of advanced database applications [Mai86, KL89, LR89]. They also cannot support higher-order features such as schema and sets in a uniform way [KN88]. These problems result from the use of inexpressive flat structures in the underlying relational data model and logic programming languages. This chapter investigates why the relational data model and logic programming languages cannot support complex object modeling and higher-order features and discusses what should be incorporated to extend deductive databases, based on the work of object-oriented programming languages and semantic and object-oriented data models.

3.1 Complex Object Modeling

Many significant applications require effective representation, storage, and manipulation of structured objects of high complexity. These include computer-aided software engineering (CASE), mechanical and electrical computer-aided design (CAD), computer-aided manufacturing (CAM), scientific and medical applications, graphics representation, office automation, knowledge representation for artificial intelligence, and business modeling applications. It has been realized that there are many powerful data modeling and manipulation concepts which need to be introduced in both programming languages and data models [Bee89, HK87, SS77, PM88]. One of these concepts is the need to model arbitrarily complex objects. In fact, the ability to model complex objects is characterized as one of the most important features of modern object-oriented programming languages, semantic and object-oriented data models [GH91].

Complex object modeling requires the adequate representation and manipulation of object identity, object properties, types, classes and inheritance [HM81, KL89, Mai87, Shi79]. The standard logic programming language and relational model using unexpressive flat structures are not rich enough to support them directly and naturally. Corresponding deductive database languages also inherit these problems.

3.1.1 Object Identity

Object identity is the mechanism for identifying and referring to objects. The purpose of introducing it is mainly for object sharing and updates. In traditional logic, object names are assumed to be different and not to ever change. Thus they can

be used to identify and represent objects. The same is true for logic programming systems. However, actual applications often violate this assumption. Two distinct objects may have the same name. For instance, it is not uncommon that two individuals have exactly the same name. In addition, object names are subject to change by the users. A database having no conflicts on object names initially may be changed to have such conflicts.

Even if we require that all objects always have distinct names, current logic programming systems cannot handle changes of object names properly. If two facts share an object, the system does not keep track of them. If the name of the shared object in one fact is changed, the system does not know whether or not the name of the object in the other fact should be changed. The user has to keep in mind which facts share an object and must change all of them explicitly if he wants to change the name of the object.

Example 3.1 Consider following two facts which describe the same individual called *Bob*, who is the author of the books *Prolog* and *Databases*:

$$author(prolog, bob).$$
$$author(databases, bob).$$

If he changes his name to *Henry*, both facts should be updated. The change is made by retracting those facts and asserting new facts:

$$author(prolog, henry).$$
$$author(databases, henry).$$

In interpreting these new facts, nothing about them requires that the domain element representing *Bob* in the first case to be the same as in the second. There is no way to say that everything stayed the same except the name. □

One solution to this problem requires explicitly using a unique but meaningless identifier which is also a term, or more specifically, a constant, to represent each object.

Example 3.2 We can use following facts to replace the facts in Example 3.1:

$author(prolog, o1)$.
$author(databases, o1)$.
$object(o1, bob)$.

To change the individual's name from *Bob* to *Henry*, we just need to retract the fact $object(o1, bob)$ and assert a new fact $object(o1, henry)$. The solution here requires systematically introducing "meaningless" terms such as $o1$ to identify corresponding objects explicitly by the user. □

Here, the user himself must remember the positions of such terms in the corresponding predicates and their meanings, and make sure that they will not be changed by chance. The system cannot treat these special terms as object identifiers and prevent any changes on them. Also, it is unreasonable to restrict such name changes.

In the relational data model, user-defined and user-controlled primary keys or foreign keys are intended for object identity. A major problem with this method is that object identity is subject to change and a single such change may require many other changes in different relations in order to keep the database consistent and meaningful. For example, suppose we use social insurance numbers (SIN), social security numbers (SSN), or other kind of serial numbers as object identity, for various reasons such as status changing, we may need to change such numbers. Since these numbers may participate in many relations as primary keys and foreign keys, all of them should be changed. It is unreasonable to disallow the change of primary

keys as well as foreign keys because they are user-defined and user-controlled values. The example below shows problems which apply to both logic programming and relational databases.

Example 3.3 Suppose Mary has a car which is a white 1992 Toyota Corolla. Here we mean a specific car. How could we identify it? —"Mary's car"? —we cannot because Mary may sell the car to someone; "white 1992 Toyota Corolla"? —there are lots of them; "car serial number?" —Another factory might use the same number for their car; "both serial number and Toyota Corolla?" —if Toyota changes its name to Tayoto someday, the identity discontinues. □

The object discussed in Example 3.3 is a physical entity in the real world, but current logic programming and relational databases have problems identifying such an object, given the above situations. The approach to object identity which is being used is not general enough to deal with these possible situations.

In object-oriented programming languages and semantic and object-oriented data models, two mechanisms are used to represent object identity: addresses and surrogates.

Address In a program or a database, associated with every object is its unique record number, i.e., address. Some language and data models take advantage of these unique addresses and use them to identify and refer to objects, such as Smalltalk, Gemstone [MSOP86], O_2 [LR89], etc.

However, using addresses for object identity is still problematic. A major problem is that it is not physical independent in the sense that moving objects in storage

devices changes object identity.

Surrogates Objects are better identified by something independent of their addresses and related values. Several data models use system-generated surrogates to represent, identify and refer to objects in databases. These include RM/T, SDM, Orion, etc. Here, the user just needs to tell the system to generate a unique surrogate and then use it for some object. The generated surrogates can only be deleted, but are not subject to change.

However, using system-generated surrogates is not always the best way. As long as they are unique and unchangeable, user-defined surrogates are adequate also. For example, the user first inserts a surrogate, if it already exists in the database, then this insertion fails and user has to choose another one. This approach relies on the system to do lookups. The advantage of this approach is that surrogates can be meaningful to the user. It is used by TAXIS, Vbase, etc. But the user has to interact with the system every time, which may not be convenient for some applications. Ideally, system-generated and user-defined surrogates should both be supported.

As discussed above, introducing addresses and surrogates is mainly for object sharing and updates. There is another kind of object which have no problem with sharing and updates, exemplified by numbers and strings. This kind of object can be identified by its representation without the above problems. In this case, the representation of an object and its identity can coincide.

Set Objects In semantic and object-oriented data models, we often need to model set objects [Bro84, AH87, HK87]. A set object is a collection of objects. There are two different ways to deal with its identity. One is using a surrogate or an address as the identity of a set object. Any changes to sets do not affect their identity. The other way is to treat a set object itself as its own identity. An insertion or deletion does not change a set, but rather it produces a new set from the given set. These two approaches are used in semantic and object-oriented data models. Choosing one or the other depends on the other factors such as object properties which immediately follows.

3.1.2 Objects and Object Properties

In many semantic and object-oriented data models and object-oriented programming languages, everything is modeled as an individual object and set objects are then constructed based on these individual objects. Constructing set objects from individual objects is normally called *set formation* [Ull88]. Individual objects are normally divided into two kinds, basic (or atomic) objects and composite (or constructed) objects [KBC+87, Car84]. A basic object is a nondecomposable value such as a string and a number. A composite object is made up of a collection of basic objects, composite and set objects.

Composite objects are normally used to describe complicated physical entities and conceptual entities in the real world such as persons, cars, departments, and universities. In relational databases and deductive databases, objects making up a composite objects may spread among different relations or predicates because the unknown values cannot be allowed in relations. For example, we cannot combine

relations EMPLOYEE and DEPARTMENT in Example 2.1 into one because no employees in the database are known in the Department of Chemistry. But in many semantic and object-oriented data models and object-oriented programming languages, a composite object must have all component objects exist in it and normally factual attributes (field names, or labels) are used to name the component objects. A component object named by a factual attribute is then called the *factual attribute value.* Each factual attribute and corresponding object is then called a *factual property* of the composite object. A factual attribute is called *single-valued* if the factual attribute value is an individual objects. Otherwise it is called *set-valued.* Thus composite objects are described via properties and thus acquire connotations. This approach is normally called tuple formation [LR89] or record formation [Car84, Ull88] by which objects are viewed as tuples or records. Note a composite object finally consists of basic objects via various tuple formation and set formation.

Example 3.4 Consider a student called Smith, aged 29, male, who is studying in the Department of Computer Science, takes courses CS 413 and CS 521, borrows books Prolog and Databases. By tuple formation, this object is represented as follows.

tuple(name: "Smith",
age: 29,
gender: "Male",
studiesIn:**tuple**(name: "Computer Science",...),
takes: {**tuple**(courseNo:"CS 413",...),
tuple(courseNo: "CS 521",...)},
borrows: {**tuple**(bookName: "Prolog",...),
tuple(bookName:"Databases",...)}).

This object is a tuple composed of several basic objects, a tuple and two sets of tuples which are in turn composed of basic objects. □

In programming languages and data models based on tuple formation, addresses or surrogates are just used as an implementation mechanism for object sharing and updates. Set objects normally have separate identity from their contents (states) here. Basic objects also have their representation as their identity. If a component object is a tuple or a set, it can have an identifier of some kind, i.e., address or surrogate. Its inclusion in other objects is then implemented by using this identifier. Therefore, it can be shared by more than one object and updating its contents does not affect such sharing. But these identifiers have no meaning in the database or program. This approach is used by IFO [AH87], FAD [DKV88], Vbase [And91], Galileo [ACO85].

Tuple formation is not necessarily the best way to represent complex objects. Circular reference such as person's spouse's spouse is still this person cannot be directly represented by such formation. Since physical and conceptual entities in the real world should have some kind of surrogates as their identity in the database, as discussed in the last section, these surrogates can be viewed as objects called *representational objects* because they represent real world entities. In [HK87, Bee89], they are called abstract objects. But the term "abstract objects" have been extensively used in logic and AI with completely different meanings [Zal88].

Factual properties of representational objects are described by a number of partial functions called factual attributes which relate the described representational object to other objects, which can be atomic object, representational object, or set of representational objects. Since the related representational object may be described by other factual attributes, complex objects are then constructed. This approach can be

called *function* (or *attribute*) *formation* in which objects are viewed as surrogates and object properties are represented by functions on these surrogates. Here the objects related through factual attributes are also called factual attribute values which are different from the factual attribute values in tuple formation. Since factual attribute values can be either individual objects or sets, the corresponding factual attributes are called single-valued or set valued. Set objects normally have no separate identity.

In this approach, a surrogate is no longer an identification or implementation mechanism, but a representation mechanism. It has full meaning in the database. Circular reference is not a problem any more. This approach is used by SDM [HM81], TAXIS [MBW80], Orion [KBC+87].

Example 3.5 Consider the object in Example 3.4 again. Let *smith* be a surrogate for the above object and *compSci, cs413, cs521, prolog* and *databases* be surrogates for the Department of Computer Science, courses CS 412 and CS 521, books Prolog and Databases respectively. By function formation, this object is described as follows.

smith (name: "Smith",
age: 29,
gender: "Male",
studiesIn: *compSci*,
takes: {*cs413, cs521*},
borrows: {*prolog, databases*}). □

In the relational data model, there is no concept of properties in this sense. Attributes of the relational model are simply the column names of relations. The relational attribute values can be identified either by their names if we use the set of mappings definition for relations or by their positions if we use the set of lists

definition for relations. The properties of an object can be represented by having both the object identity which is the primary key and related objects or object identifiers which are foreign keys in the same tuple of some relation. The related objects may also have properties and are represented in the same way so that deeply nested structured objects can exist. The set-valued properties are normally simulated by a set of tuples.

NAME	AGE	STREET-NO	STREET
Mary	18	182	Rocky
Bob	52	3452	Golden
Jenny	37	1834	Silver

(a) PERSON

CAR-NO	MODEL	YEAR
632087	Toyota Corolla	1989
724512	Ford Mustang	1990
393762	Mercedes Benz	1985
789413	Honda Accord	1992

(b) CAR

CAR-NO	NAME
632087	Mary
724512	Bob
393762	Bob

(c) OWNED-BY

Figure 3.1 Example Representation of Object Properties by Relations.

Example 3.6 The relations in Figure 3.1 represent objects, persons and cars. Each

individual in the relation PERSON has properties *name, age, street number*, and *street name*. Each vehicle in relation CAR has properties *car serial number, model*, and *year*. Each person also has a set-valued property *own* which relates the person to a set of cars. But it is indirectly represented by the intermediate relation OWNED-BY. For example, *Bob* has two cars: a 1990 Ford Mustang and a 1985 Mercedes Benz. □

The example above shows that the relationships between persons and cars are represented through common values in different relations indirectly. To obtain such a relationship, for example, what cars are owned by Mary, we must perform join operations over these three relations. Users must keep in mind the relationships between relations. One might argue that a single relation can be used to represent all objects for the above example. The problem with this is that some persons without cars like Jenny and some cars without owner like the 1992 Honda Accord cannot exist in the relation. For this reason, the relational model is sometimes called a syntactic model, i.e., there is no semantics in the relations. The semantics are in the user's mind.

In logic programming, uninterpreted function symbols can be used to describe objects. The user can take advantage of this interpretation for organizing objects, representing object properties and making them behave as tuple formation.

Example 3.7 Figure 3.2 shows a fact which represents information about cars. The term *vehicle*(...) describes a certain car and the term *owner*(...) describes its owner. The subterm *no*(6320847) represents the serial number of the car, *model*(*toyota-corolla*) represents the model of the car and *year*(1989) represents the year when the

car was made for. The rest are self explanatory. □

$$
\begin{array}{l}
car(vehicle(no(6320847), \\
\qquad model(toyota - corolla), \\
\qquad year(1989)), \\
\quad owner(name(mary), \\
\qquad age(18), \\
\qquad address(number(182), \\
\qquad\qquad street(golden))).
\end{array}
$$

Figure 3.2 Example Representation in Logic Programming.

But this approach has several problems. First, it does not give direct semantics to these object properties because these function symbols are not interpreted. Second, the interpretation of argument positions within a predicate is not transparent to the user. Indeed, in using the term *owner*(...) in the above fact, one must always be aware that the first argument is a name, the second is an age, etc. Third, set-valued properties cannot be supported directly. The problems with this will be discussed further in the next section. Finally, circular reference cannot be directly represented.

First-order logic was developed to give precise meaning to statements in mathematics. It does not support the description and manipulation of the existence and intensional structure of complex objects naturally and directly [Mai86]. Logic programming based on first-order logic inherits this problem.

In deductive databases, we have two choices to represent object properties: using uninterpreted function symbols as in logic programming, or taking the view of relations as sets of lists and using the position names as the relational attribute names. We cannot take the other view of relations as sets of mappings because a

tuple corresponds to a fact, the positions of a fact in which the components (relational attribute values) appear are important. The problems with the first choice have been discussed. In relational databases, we can query attribute values simply by using attribute names. However, in deductive databases, the position name is in the contents of schema and is not usable in the database level. The next section will show why. So the user must keep the schemas in his mind when referring to the relational attribute values. This means the user must know precisely all structures of different fact collections (each of them represents a relation), instead of the information contents of the facts which is typical to the relational model. So the management of large applications becomes more complex from the view point of the user. Data independence is lost.

In summary, in deductive databases, the representation of object properties is not direct and natural, and querying on object properties is problematic.

3.1.3 Types and Classes

Objects often share common factual structural and behavioral properties. To be able to describe them uniformly and make them more meaningful, the concept of types and classes has been introduced [AFOP88, Bor88, Bee89].

Objects sharing common factual structural and behavioral properties are normally grouped into classes. Corresponding to each class, a type is used to give a precise specification of common properties shared by all objects in the class. The properties defined for a type are called *definitional properties* of the type, in contrast to the factual properties of objects, and are normally represented different from factual properties. Therefore, type and class are two closely related aspects. They

represent the same group of objects with different functionality. The type represents the intensional or definitional aspect of this group. The class represents the extensional aspect of this group.

In some language such as Galileo, both class name and type name must be given explicitly. But this is inconvenient to the user [Bor88]. So normally, class names and their type names are the same. For example, the type INTEGER may only allow arithmetic operations on it and the class INTEGER denotes the set of all possible integers; while the type STRING may only allow equality and inequality operations on it and class STRING denotes the set of all possible strings.

Types can also be viewed as constraints on structural and behavior properties. In this view, typing helps to enforce correctness and detect errors [CW85, Bor88].

In most object-oriented programming languages and semantic and object-oriented data models, types for basic objects, such as strings and integers, called basic types, are built-in as STRING and INTEGER and so are their objects. That is, these types do not need to be defined and objects in the corresponding classes do not need to be specified. But types for composite objects or representational objects, depending on how objects are formed, must be defined explicitly. The extensions of these types, that is, the classes are application dependent and may vary from time to time.

For every type A, its set type represented by **set**(A) (or **set of** A) is automatically defined which normally allows usual set operations on the objects possessing this type and the corresponding class represented also by **set**(A) which is power set of the class A, that is, **set**(A) $= 2^{A}$. Therefore, objects possessing a set type **set**(A) has to be a homogeneous set in the sense that all objects in the set must possess the same type

A. Other kinds of set objects are normally disallowed.

Types for composite objects or representational objects are defined differently, depending on how objects are formed. For tuple formation, structural properties of a type are represented by a list of attributes and corresponding attribute types which determine possible factual attribute values. This kind of types are called constructed type or record types [Car84, KBC+87, Ull88]. Each object possessing such a type is a tuple, and classes are sets of tuples.

Example 3.8 The type for the tuple object in Example 3.4 can be defined as follows.

```
type STUDENT
        name: STRING;
        gender: STRING;
        age: INTEGER;
        studiesIn: DEPARTMENT(name: STRING; ...);
        takes: set(COURSE(name: STRING; ...));
        borrows: set(BOOK(name: STRING; ...));
end STUDENT.
```
□

For function formation, objects are just surrogates and classes are sets of surrogates. Structural properties of a class, that is, its type is represented by a list of definitional attributes which are mappings from the class to other classes. This kind of types are called *representational types* in this thesis. Each object possessing such a type has a list of factual attributes which link this object to other objects (surrogates).

Example 3.9 The type for the representational object in Example 3.5 can be defined as follows.

```
type STUDENT
        name: STRING;
        gender: STRING;
        age: INTEGER;
        studiesIn: DEPARTMENT;
        takes: set(COURSE);
        borrows: set(BOOK);
end STUDENT.
```

□

In terms of expressive power, representational types are more powerful than constructed types, because circular reference is allowed. For example, we can have PERSON(father:PERSON, mother:PERSON, children:**set**(PERSON)) only as a representation type rather than a constructed type. This is because the structural properties in constructed types have to be built bottom-up.

The interaction between types and objects is modeled normally in two different ways in programming languages and data models. One is called *conforms-to* which states that if an object possesses the structure that a type expects its elements to have, then the object conforms to the type and is an element of the corresponding class. The condition for conformity only bounds the object structure from below. It is prescriptive: an object can have more structure than the type specifies and still conforms to the type. In this way, types can be inferred from objects. This approach applies to tuple formation only.

The other way is called *asserted-of* which states that an object possesses a type if and only if it is explicitly asserted to be a member of the corresponding classes. That it is asserted of a type. The reason for this is that two different types may have exactly the same properties and we cannot tell which class an object is in by its properties. For example, type NEWSPAPER and JOURNAL may have the same

properties: *name, publisher.* The asserted-of approach is intended to give this higher-level control to the user: it is up to the user to decide on the intended conceptual constraints. In this case, even though some attribute values of an object are unknown, it still possesses the type. Unlike the conforms-to approach, here type inference is disallowed. This approach applies to function formation and tuple formation. It is especially useful for database applications. All the extensions to logic programming and deductive databases which will be examined use this approach. Therefore, the rest of this chapter focus on asserted-of approach.

3.1.4 Property Inheritance

The factual properties of objects determined by certain types, as discussed above, are intended to be incomplete. That is, further factual properties can be added to these objects via the introduction of subtypes. A subtypes is a type which inherits all definitional properties from its supertypes and can have extra definitional properties local to itself.

The relationship between types is normally modeled by *is-a* (subtype of) which states that if A is-a B then all definitional properties of type B are also definitional properties of type A, and every object in class A either by confirms-to or by asserted-of is also in class B. The is-a relationship is a partial order. That is, it is reflexive, antisymmetric and transitive.

Subtypes of basic types are supported only in some models and languages. When supported, they normally do not need to be defined, they are just directly used in other type definitions. Using subtypes of basic types can make other type definitions more precise and meaningful. For example, {15..35} is a subtype of INTEGER, which

contains all integers between 15 and 35 inclusive. Similarly, {"Male", "Female"} is a subtype of STRING, which contains only two strings "Male" and "Female". For the type definitions in Examples 3.8 and 3.9, it makes more sense if pairs gender:STRING and age:INTEGER are changed to gender:{"Male","Female"} and age:{15..35}.

Normally, the type OBJECT (or ENTITY) for all possible representational objects or composite objects is built-in. It has no structural properties but it may have some kind of behavioral properties such as equality or inequality operations applicable to its objects. Its extension is application dependent and has to be inserted into or deleted from the corresponding class explicitly by the user. Every composite type or representational type is a subtype of OBJECT so that the behavioral properties can be inherited.

Using inheritance, types can be and are often organized into a meaningful type subsumption hierarchy (taxonomy). The type subsumption hierarchy affects both intensional as well as extensional aspects of types. The former is manifested in the form of inheritance: subtypes inherit all definitional properties from their supertypes. The latter takes the form of subset inclusion, the class corresponding to a type is included in the class corresponding to its supertypes.

Normally, a type definition contains its (immediate) supertypes and definitional properties local to itself and may further refine some definitional properties of its supertypes.

Example 3.10 Figures 3.3 and 3.4 show how types can be defined in two typical databases. Both of them define two types: PERSON and STUDENT. Type PERSON is-a ENTITY and have definitional properties: *name, gender, age* and *address.*

```
define type PERSON
        supertypes = {ENTITY}
        properties = {
                name: STRING;
                gender: STRING;
                age: INTEGER;
                address: STRING;}
end PERSON;

define type STUDENT
        supertypes = {PERSON}
        properties = {
                studiesIn: DEPARTMENT;
                takes: set[COURSE];
                borrows: set[BOOK];
end STUDENT.
```

Figure 3.3 Type Definitions in Vbase.

Type STUDENT is-a PERSON with additional definitional properties: *studiesIn* a department, *takes* a set of courses, *borrows* a set of books. In Figure 3.3, subtypes of INTEGER and STRING cannot be used, therefore definitional property refinement on age is disallowed. But in Figure 3.4 subtypes of INTEGER and STRING, such as {15..35}, {0..125} and {"Male", "Female"} can be used, which makes the type definition more meaningful and precise. By saying STUDENT is-a PERSON in type definitions, all definitional properties of PERSON are automatically inherited by STUDENT and all objects in the class STUDENT are also in the class PERSON. If we need to delete or add some definitional properties such as *gender* from or to PERSON, we do not need to change the description for STUDENT at all. If an object is deleted from or inserted to the class STUDENT, it is automatically deleted from or inserted to the class PERSON. □

```
class PERSON isa ENTITY with
        name: STRING;
        gender: {"Male", "Female"};
        age: {0..125};
        address: STRING;
end PERSON.

class STUDENT isa PERSON with
        age: {15..35};
        studiesIn: DEPARTMENT;
        takes: set of COURSE;
        borrows: set of BOOK;
 end STUDENT.
```

Figure 3.4 Type Definitions in TAXIS.

Property inheritance has following advantages. It enhances semantics expressiveness and reduces conceptual complexity of a system specification by providing a natural structure for defining and sharing definitional properties. It increases system maintainability by allowing new definitional properties to be added by augmentation, rather than mutation of existing code. It allows for sharing of code and implementation and reduces the redundancy of the specification while maintaining its completeness.

All the type definitions constitute the schema for the database and all entered objects satisfying the schema form the database. Like the relational model, most semantic and object-oriented data models offer a language for schema definition and query and another language for object manipulation and query. The reason for this separation is that schema information is used not only for querying but also for strong type checking, enforcing integrity and database organization. Therefore, its

usage is quite different from object information in databases. For example, in Vbase, schema language is called TDL and object language is called COP.

In the relational model, a relational schema can be viewed as a type definition for the relation. Each attribute in a relation corresponds to a class named by itself. It can also be viewed, quite artificially, as a mapping from the class of the objects represented by the primary key to the class denoted by the attribute. Set-valued definitional properties are not directly supported. Instead, they must be represented as many-one or many-many relationships. These relationships among relations are represented by the common data values and are not supported by the system, rather they are kept in the user's mind and obtained through the use of join operations.

Example 3.11 Consider the relations in Figure 3.1, the relational schema PERSON(**NAME**, AGE, STREET-NO, STREET) can be viewed as a type definition for type PERSON. The attribute AGE can be viewed as a mapping from the object class denoted by NAME to the class denoted by AGE; while the set-valued property *own* is represented by a many-one relationship from CAR to PERSON via three relations PERSON, OWNED-BY, and CAR. □

The subsumption relationship between two types A and B can be represented by two relations A and B, which only share the same key values in relational systems, see example below. This representation suffers the same problem, that is, the join operation has to be used to obtain the inherited properties via the common key values.

Example 3.12 Consider the Example 3.6 again. We can use relational schemas PERSON(**NAME**, AGE, ADDRESS) and EMPLOYEE(**NAME**, SALARY, MANAGER) to represent type PERSON and EMPLOYEE in the relational model. To obtain the properties which EMPLOYEE inherited from PERSON, we must use the join operation over the common key NAME. We cannot ask what properties an EMPLOYEE, say *Mary* has without knowing that EMPLOYEE is related to PERSON and without joining them. □

In summary, the structure of the relational data model is too simple to directly support general hierarchies of types with complex nested structures.

It is argued by Reiter in [Rei84] that logic programming systems could use logical implication to express inheritance. However Hassan and Nasr claim in [AKN86] 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.
>
> Naturally, this meaning of inheritance can be well captured in logic by the semantics of logical implication. Indeed,
>
> $$\forall x Whale(x) \Rightarrow Mammal(x)$$
>
> is *semantically* satisfactory.
>
> However, 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."

What the argument suggests is that current logic programming systems are not smart enough to track inheritance information represented by logical implication and take advantage of it to produce more efficient systems. Using a special structure other than logical implication to represent inheritance explicitly by the user could be beneficial.

3.2 Higher-Order Features

In mathematical logic, the primitive symbols include constants, function symbols, predicate symbols and various variables. In first-order logic, only individual variables can be used, i.e., variables can only be used in the places where constants can be used. In second-order logic, not only individual variables, but also function variables and predicate variables can be used. In other words, variables can not only be used in place of constants, but also in place of predicates and functions. Since predicates are interpreted as sets and relations, variables can represent not only elements, but also sets, subsets, relations and functions constructed out of the domain of discourse in second-order logic. In higher-order logic, variables can be used over functions defined on functions, etcetera.

In deductive databases, to manipulate schema and sets naturally lead us into higher-order logic. This section focuses on the problems associated with these two

higher-order features.

3.2.1 Schema

A deductive database is a model of some portion of the real world in which one is interested. This model is partitioned into two parts: (1) the schema, which captures generic, time-invariant structural or definitional information, and (2) the database, which captures specific, more volatile individual or factual information which satisfies the schema. Usually the database is much larger than the schema. The schema includes all predicate and function definitions, as well as integrity constraints on legal values. It tells how the database is structured, such as how many relations there are and how many attributes within each relation are used in the database. This information allows the system to organize the database effectively and enables static type checking which avoids a large number of common errors. It also gives some kind of meaning to the data in the database. Relational names are described by a set of attribute names and attribute names tell what the components in a relation tuple are used for and thus make the relational names meaningful.

Example 3.13 Suppose the database consists of the three relations in Example 3.6: PERSON, CAR, OWNED-BY. The schema for each relation tells how many attributes are used and what they are used for. For the relation PERSON, it has exactly four attributes: *person name*, *age*, *street number* and *street name* and the corresponding relational schema is PERSON(**NAME**, AGE, STREET-NO, STREET). Given a tuple of PERSON or a fact such as $person(mary, 18, 182, rocky)$. We know how to interpret each component based on the schema. We cannot insert $person(henry, 25)$,

because it does not obey the relational schema. □

It is a natural requirement of deductive databases to have function and predicate variables to query the schema. Unfortunately, this cannot be done in the same way as we query the database. There are several difficulties.

First, schema information is not compatible with either relation tuples or facts. In other words, schema information cannot be directly and naturally represented in the same way as individual information is dealt with. In deductive database, the definition and query of data is done in a uniform way. However, the definition and query of schema (or meta) information and data cannot be supported in such an integrated framework directly and naturally. We have to separate the schema and the database and also use a mechanism different from logic programming to represent schema and queries on them. This is a serious impediment to the development of integrated systems.

Example 3.14 Consider Example 3.6 again. The schema for the relation PERSON is PERSON(**NAME**, AGE, STREET-NO, STREET). However, it cannot be represented as it is now. For example, how could a fact $person(mary, 18, 182, rocky)$ be distinguished from the schema. Even if some mechanism can be used to tell them apart in the way they are now, there may still be difficulties to query them in the logic programming style. We may wish to use a variable X to list all predicates or relation names in the schema. The substitution for X should range over all the predicate or relation names. However, the present form of deductive databases cannot represent this query in such a simple way. Since relations or predicates can have different numbers of arguments, simply using a predicate variable does not allow us to express this

query in the usual way which logic programming uses. So we have to use a number of queries $X(\_)$ for unary relations name, $X(\_,\_)$ for binary relations name, etc. and there should not be an upper bound for this sequence in theory. Even though we know the relation name PERSON and we want to know the attribute names, if we have no idea about the number of attributes, we still need to use a number of queries, such as $person(A_1)$, $person(A_1, A_2)$, ... to find out how many attributes the relation has and what they are. Only the query $person(A_1, A_2, A_3, A_4)$ can succeed in this case and report the first attribute name is *name*, the second is *age*, etc. □

Secondly, even the above way to query the schema cannot work. In relational databases, the user can use attribute names to query the database. But this cannot be done in deductive databases based on traditional logic programming. It leads us beyond first-order logic to higher-order logic to deal with the schema. However, the use of higher-order facilities introduces serious technical problems which will be discussed later in this section. So normally, a separate language is provided to specify and query the schema information.

Based on the above discussion, the language for a schema is unrelated to the logic programming language. As a consequence, the user cannot use the schema information such as attribute names to query the database because of the separation of the schema language and the logic programming language, even though the relational schemas give some kind of interpretation to the data in the database. Instead, he must find out schema information using the schema language and remember it, such as relational names and attribute names, while he wants to query the database and interpret the results. For example, Let $person(john, mary, bob)$ be a fact obtained

by the user. This fact may be interpreted in many different ways, such as *john*'s mother is *mary*, and his father is *bob*, or *mary* and *bob* are *john*'s children, etc. By looking up the schema using the schema language or remembering the schema information for *person*, the user can then give the right interpretation. This is certainly inconvenient to the user.

3.2.2 Sets

As discussed earlier, the representation and manipulation of sets is an important aspect of complex object modeling. In mathematical logic, unary predicates can be used to represent sets and their arguments represent their elements. Using variables to range over various sets actually requires these variables to range over predicate symbols based on standard semantics. This is therefore a higher-order feature.

As will be discussed later in this section, the use of higher-order facilities introduces serious technical problems. Besides, representing sets by predicates has other problems. Here there is no notion of sets syntactically, only constants, function symbols, predicate symbols and various variables. Therefore, a set cannot be represented directly in the usual mathematical sense. It has to be represented via a predicate indirectly or semantically. As example 3.5 shows, complex object modeling requires syntactic sets. Using only semantic sets to simulate is not intuitive and convenient. Besides, since there is no sets syntactically, there is no syntactic variables over such sets. As a result, the relationships between sets such as subset, disjoint etc. cannot be represented directly and intuitively.

Example 3.15 Suppose Mary speaks English, French, Chinese, and Bob speaks

English and French. In mathematical logic, predicate symbols *marySpeaks* and *bobSpeaks* can be used to represent sets of languages which Mary and Bob speak:

$marySpeaks(english).$
$marySpeaks(french).$
$marySpeaks(chinese).$
$bobSpeaks(english).$
$bobSpeaks(french).$

So a set is represented by a set of unary relation which is a special case of the representation of the relational model. A similar example in the relational model has been discussed in Example 3.6. To find out whether the languages spoken by Bob are also spoken by Mary, set-valued variables cannot be directly used here to do the comparison. Instead, an individual variables have to be used to ask the query: $\leftarrow bobSpeaks(X), \neg(marySpeak(X))$, which says if there is a language X which is spoken by Bob but not spoken by Mary. This representation seems to be procedural. That is, it tells how it will be done. This is contrary to the general philosophy of logic programming. If syntactic sets like {*english, french, chinese*} {*english, french*} can be used to represent the languages spoken by Mary and Bob in some way as Example 3.5 shows and variables X and Y can be used to range over them respectively, then query $\leftarrow Y \subseteq X$ or $\leftarrow subset(Y, X)$ seems to be more direct and intuitive. □

Since sets are so useful, an important extension has been proposed and used extensively in the logic programming language Prolog [War82]. The extension is based on the mathematical definition of sets: $S = \{X|P\}$, where S stands for the defined set, X stands for the variable ranging over the set and P is a predicate in which X occurs. It is read as "The set of all instances of X such that P is true is S." Here S is a syntactic set. In Prolog, the extension takes the form of a built-in

predicate:

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

Unfortunately, there is no published formal semantics for the *setof* predicate in current logic programming languages. The difficulties for the semantics is due to the general higher-order logic problems which will be presented shortly, because predicate variables can be used in it.

In addition, in Prolog, the set S is represented as a list whose elements are sorted into a standard order without any duplicates. A list is defined as an uninterpreted function consisting of the special functor "." applied on the head, and the tail which is also a list, either empty [] or having its own head and tail.

$$.(Head, Tail)$$

Example 3.16 The set $\{english, french, chinese\}$ can be represented in Prolog by a list $[english, french, chinese]$ which is an alternative notation for $.(english, .(french, .(chinese, [])))$. Using the *setof* predicate, the query in Example 3.15 can be represented as follows.

$$\begin{array}{l} \leftarrow setof(X, marySpeaks(X), S_1), \\ \quad setof(X, bobSpeaks(X), S_2) \\ \quad subset(S_1, S_2). \end{array}$$

where the predicate *subset* will be defined in Example 3.18. □

Example 3.17 Using lists in Prolog, the membership predicate is usually be defined as follows.

$$\begin{array}{l} member(X, [X|L]) \leftarrow. \\ member(X, [X|L]) \leftarrow member(X, L). \end{array}$$

□

Representing sets as lists also causes problems. The semantics for lists is quite different from that of sets. When a predicate involves more than one set, the rules can become quite complicated and unintuitive. The user has to specify details about implementation, such as how to iterate over the sets (see examples below). This is also contrary to the general philosophy of logic programming as well as deductive databases [Kup87]. Whenever possible, the user should not have to deal with the control structures in the program.

Example 3.18 In Prolog, the predicates that say one set is a subset of another and two sets are disjoint could be defined as follows.

$$subset([], L) \leftarrow .$$
$$subset([X|L_1], L) \leftarrow member(X, L), subset(L_1, L).$$

$$disjoint(L, []) \leftarrow.$$
$$disjoint([], L) \leftarrow.$$
$$disjoint([X|L_1], [Y|L_2]) \leftarrow disjoint([X|L_1], L_2), X \neq Y,$$
$$disjoint(L_1, [X|L_2]).$$ □

In summary, even with the higher-order problems as will be discussed shortly, sets cannot be directly and naturally represented in mathematical logic. To naturally account for the complex object modeling, a new logical semantics should be developed which includes syntactical sets as well as variables for these syntactical sets.

3.2.3 Problems with Higher-Order Logic

In mathematical logic, various formal systems called theories are studied [Hat82]. A theory has two aspects: syntax and semantics. The syntactic aspect is concerned with well-formed formulas admitted by the grammar of a formal language, as well as

deeper proof-theoretic issues. The semantics is concerned with the meanings attached to the well-formed formulas and the symbols they contain. A theory consists of an alphabet, an object language, a set of axioms, and a set of inference rules. The language consists of the well-formed formulas of the theory built out of the alphabet. The axioms are a designated subset of well-formed formulas. Based on the set of axioms, whatever can be inferred from the language using inference rules are called theorems.

An interpretation simply consists of some domain of discourse and a semantic function which attaches some meaning to each of the symbols in the language and decides the truth or falsity of every well-formed formula. The relationship between valid well-formed formulas of a language and theorems is of most interest in mathematical logic. A desired property is that valid well-formed formulas are exactly the theorems. This is so called completeness and soundness property of the inference rules.

First-order logic theories allow only individual variables. Higher-order logic theories allows not only individual variables, but also function variables and predicate variables.

For a theory (first-order and higher-order), it is often required that the alphabet, the language, and the set of axioms be effective in the sense that there is some procedure which can decide whether a given object is or is not a sign of the system, or whether a given expression is or is not a well-formed formula, or whether a given well-formed formula is or is not an axiom. A theory is called has an effective notion of proof if all the theorems are a decidable set [Hat82, BJ89].

There are two kinds of semantics for higher-order logic, standard model, and general model [Hat82, vBD83, BJ89]. In standard model, the logical consequences are the well-formed formulas which are true in all possible models. In general model, the logical consequences are the well-formed formulas which are true in all general models which are models satisfying some constraints.

Higher-order logic is certainly more expressive than first-order logic. However, it suffers various problems. Important properties held for first-order logic fail for higher-order logic [BJ89]. First-order logic has an effective notion of proof which is complete and sound with respect to the intended interpretation. This is the content of Godel's completeness theorem, which says in any predicate calculus, the theorems are precisely the logically valid well-formed formulas. As a result, the set of universally valid first-order formulas is recursively enumerable based on Godel's numbers [Hat82, vBD83, BJ89]. But with second-order logic, the set of second-order validities is not arithmetically definable in standard semantics, let alone recursively enumerable, and hence an effective and complete axiomatization of second-order validity is impossible [vBD83]. In general model, second-order validities can be defined, But there is no effective notion of proof for validity.

Even the simplest questions about the model theory of second-order logic turn out to raise problems of set theory, rather than logic. If two models are first-order equivalent and one of them is finite, they must be isomorphic. If we use second-order equivalence and relax finiteness to, say, countability, it has been proved that this question is undecidable.

The most significant problem with higher-order logic is that higher-order uni-

fication is undecidable [Gol81, Hue73]. In automatic theorem proving and logic programming, unification is the kernel for deduction. The undecidability of higher-order logic makes such deduction impossible. Given a query of higher-order logic, we may end up with infinite waiting. This means that we cannot use it to query the schema and the database.

In practice, many mathematical theories are first-order even though they involve sets, subsets, relations. These includes number theory, ZF set theory [Hat82]. The reason for this possibility is that these theories take advantages of the difference between higher-order syntax and higher-order semantics. A theory has a higher-order syntax if it appears to be higher-order, i.e., have variables for predicates, functions, sets, or, subsets, etc. A theory has a higher-order semantics if its variables range over not only individuals, but also domains of individuals, relations and functions constructed out of the domain of individuals in its semantics.

Example 3.19 ZF set theory has sets, supports subset relations and allows variables for them. But it is first-order because there is no variables for function and predicate symbols. Based on this syntactical classification, it, can be said, has first-order syntax. However semantically, individuals are interpreted as sets which can contain other sets. Therefore, the variables for individuals become the variables for sets. Every term is referred to as a set. The domain of interpretation no longer consists of individuals, but rather of sets constructed out of individuals. Its semantics is no longer that of traditional first-order logic. Based on this semantic classification, it, can be said, has higher-order semantics. But if higher-order syntax is used, major axioms of the ZF set theory would be represented more naturally. With higher-order

logic, one really enters the realm of set theory. However, the first-order version of these axioms have turned out to be sufficient for many mathematical purposes. □

3.3 Summary

To model complex object, we need proper notions to represent object identity, single-valued properties and set-valued properties, syntactical sets, types, classes and inheritance. Since classes and objects are two fundamentally different concepts and used differently, there ought to be some way to distinct them clearly. But it is also desirable to treat classes as meta-objects and to view the associated type definitions for their instances as part of their own properties and make the schema of a database accessible to its users [Bor88]. In this sense, object properties and class properties should be somewhat uniformly represented.

To represent and manipulate schema and sets, we need variables for not only individuals, but also predicates and functions. This leads to higher-order logic. However, higher-order logic has serious problems with its semantics and unification. It seems to be a dilemma.

As will be discussed later, deductive databases only require very limited higher-order features and it seems possible to avoid these problems.

We also need variables for syntactical sets. In such case, we no longer have well formed logical foundation based on mathematical logic. Substantial theoretical work is needed to account for this.

In the next chapter, various extensions to mathematical logic, logic programming and deductive databases are examined against the above criteria.

Chapter 4

Critical Analysis of Related Work

In the last chapter, the need for complex object modeling and higher-order features has been discussed and the corresponding concepts have been introduced. This chapter examines typical extensions to mathematical logic, logic programming and deductive databases in the literature and shows why they cannot naturally and directly support complex object modeling and higher-order features. The examined extensions to mathematical logic and logic programming can be called extended logic term approaches.

4.1 Extended Logic Term Approaches

As a result of the lack of expressiveness in mathematical logic and logic programming, some attempts have been made to provide those missing functionalities, such as object identity, object properties, sets, types, inheritance, and well-formed semantics. These include LOGIN [AKN86], O-Logic [Mai86], Revised O-Logic [KW89], and F-logic [KL89, KLW90]. They try to provide various notions to represent complex objects and higher-order features. This section examines them individually from a historical point of view based on the concepts developed in the last chapter. Following the usual conventions in logic programming, variables are represented by upper-case letters and words, non-variables are represented by lower-case words.

4.1.1 LOGIN

In LOGIN of Act-Kaci and Nasr [AKN86], a program consists of four parts: a type signature, ψ-terms, predicates and rules. Predicates and rules are the same as traditional logic programming. But terms are extended to more expressive ψ-terms and unification is extended to deal with the type signature.

The type signature Σ is a partially ordered set of type symbols containing two special elements: a greatest element ($\top$) and a least element ($\bot$). All types in the signature are representational types. The symbol $\top$ corresponds to the type OBJECT which denotes the set of all possible representational objects and the symbol $\bot$ denotes the empty set. The type signature is just a type hierarchy as discussed earlier. The type signature determines the subtype relationship ($<$) between all representational types. Even though subtypes of basic types are used in LOGIN and can form a type hierarchy, they are not included in the signature. The reason might be that subtypes of basic types in LOGIN usually have no names.

The ψ-terms are used to define types and describe objects. For every type a, if there is no type b except $\top$ in the signature such that $a < b$, then its type definition can be represented directly by a ψ-term. A ψ-term consists of a type symbol, labels, sub-ψ-terms and arrows ($\Rightarrow$) from labels to sub-ψ-terms. Labels are used to represent attributes defined on the type symbol. Each label denotes a function from the root class to the class denoted by its associated sub-ψ-term. That is, LOGIN uses function formation to describe complex objects.

Example 4.1 Let the type *person* have definitional attributes *name* which is a string; *gender* which is either "Male" or "Female"; *age*, an integer between 0 and 120;

lives, an addresse consisting of attributes *number*, an integer, *street, city* which are strings. Suppose that there is no type B such that *person* $< B$ is in the signature. Then its definition in LOGIN can be expressed by a ψ-term as follows.

$$\begin{aligned} &person(name \Rightarrow string;\\ &\quad gender \Rightarrow [\text{“Male”, “Female”}];\\ &\quad age \Rightarrow [0...120];\\ &\quad lives \Rightarrow address(number \Rightarrow integer;\\ &\qquad\qquad street \Rightarrow string;\\ &\qquad\qquad city \Rightarrow string)). \end{aligned}$$ □

If $a < b$ is in the signature, then all definitional properties of b defined by a ψ-term are inherited by a. But if a has its own properties or further refines some properties of b, then this must be expressed in LOGIN by $a = \Psi$, where Ψ is a ψ-term which has b as its root symbol and includes the properties local to a and refined type properties of b. Each label in Ψ is no longer a function from the class b but from the class a.

Example 4.2 Let *student* $<$ *person* be in the signature. Besides the inherited definitional properties from *person* in Example 4.1, suppose *student* has its own definitional attribute *studiesIn* a department and further refines *person*'s definitional attributes *age* to 15 till 35. This definition can be represented in LOGIN as follows.

$$\begin{aligned} &student = person(studiesIn \Rightarrow department;\\ &\qquad\qquad age \Rightarrow [15...35]). \end{aligned}$$ □

In LOGIN, objects are represented by user-defined identifiers. That is, objects are viewed as surrogates in the sense discussed earlier. LOGIN uses the asserted-of approach for types of objects. That is, objects of a type must be asserted in LOGIN. To assert an object o to be of type t, we can use $o = \Psi$, where Ψ is a ψ-term which

has t as its root symbol and includes all the known factual properties of o defined on type t.

Example 4.3 In LOGIN, to assert *mary* to be of type *person*, where *mary* is the object identifier for a person called Mary, female, aged 18, living in 3452, Golden street, Calgary, and to assert *smith* of type *student*, where *smith* is the object identifier for a student called Smith, Male, aged 29, who studies in the Department of Computer Science, we could use

$$
\begin{aligned}
mary = person(&name \Rightarrow \text{“Mary”};\\
&gender \Rightarrow \text{“Female”};\\
&age \Rightarrow 18;\\
&lives \Rightarrow address(number \Rightarrow 3452;\\
&\qquad\qquad street \Rightarrow \text{“Golden”};\\
&\qquad\qquad city \Rightarrow \text{“Calgary”})).\\
smith = student(&name \Rightarrow \text{“Smith”};\\
&gender \Rightarrow \text{“Male”};\\
&age \Rightarrow 29;\\
&studiesIn \Rightarrow cpsc)).
\end{aligned}
$$

□

Even though types and objects are both represented by ψ-terms, the results of such representation have fundamental difference. A complete definition of a type can be obtained by combining both a ψ-term and the signature. But a description of an object by a ψ-terms itself is complete.

A ψ-term differs from a traditional term. It is not a fixed-arity term. Its arguments are identified by their attribute labels, not by their positions.

In LOGIN, set types are not supported and therefore set-valued properties cannot be defined for types directly in the way discussed in the last chapter. For example, within the type definition for *student* in Example 3.8, the fact that every student

taking a set of courses cannot be defined. But information about set valued attributes of individual objects can be represented by using traditional predicates and lists.

Example 4.4 The fact that student identified by *smith* takes two courses: cs413 and cs 521 and borrows two books: Prolog and Databases, can be represented in LOGIN by

$$takes(smith, [cs413,\ cs521]).$$
$$borrows(smith, [prolog, databases]).$$ □

The intention of LOGIN is to extend the unification algorithm of Prolog. The major contributions of LOGIN are that ψ-terms are more meaningful and expressive than traditional terms. Object identity, single-valued factual properties, type and inheritance can be naturally supported. Inheritance information can result in more efficient Prolog systems.

However, the approach is restricted to the unification algorithm, that is, given two ψ-terms, how to unify them with respect to the signature. The denotational semantics is given only to ψ-terms, not to programs. It is not clear in which logic this unification algorithm would yield a sound proof procedure [KLW90]. Besides, given a program of LOGIN, what can be computed or what is the intended semantics and what properties the program may have are not explored. Objects cannot be generated here by rules. Besides, ψ-terms still function as normal terms. That is, they are used in traditional predicates which are not ψ-terms, as the above example shows. Within a predicate, positions for terms are still important and each predicate still has fixed-arity. Like Prolog, set objects are represented by lists and variables can range over these lists. If LOGIN is used as the deductive database language, the

schema is very complicated, we have ψ-terms as type definitions for objects, we also have types for predicates which should be defined differently and separately for the reasons discussed in the last chapter.

4.1.2 O-Logic

An extended first-order logic called O-Logic (logic for objects) was described by Maier [Mai86]. In O-logic, a program consists of only two parts: O-terms and rules. O-terms are used to describe objects and rules are used to derive new objects. An O-term is similar to a ψ-term in syntax, with a variable, labels, sub-O-terms and arrows ($\rightarrow$) from labels to sub-O-terms. Type names can be added in front of the variable in an O-term.

Example 4.5 Objects similar to *bob* and *mary* in Example 4.5 can be represented by O-terms as follows.

$$
\begin{aligned}
&\mathit{person} : P(\mathit{name} \rightarrow \mathit{string} : \text{“Mary”};\\
&\qquad \mathit{gender} \rightarrow \mathit{string} : \text{“female”};\\
&\qquad \mathit{age} \rightarrow \mathit{integer} : 18;\\
&\qquad \mathit{lives} \rightarrow \mathit{address} : A(\mathit{number} \rightarrow 3452,\\
&\qquad\qquad \mathit{street} \rightarrow \mathit{string} : \text{“Golden”},\\
&\qquad\qquad \mathit{city} \rightarrow \mathit{string} : \text{“Calgary”})).\\
&\mathit{student} : S(\mathit{name} \rightarrow \mathit{string} : \text{“Smith”},\\
&\qquad \mathit{gender} \rightarrow \mathit{string} : \text{“Male”},\\
&\qquad \mathit{age} \rightarrow \mathit{integer} : 29,\\
&\qquad \mathit{studiesIn} \rightarrow \mathit{dept} : D(\mathit{name} \rightarrow \mathit{String} : \text{“Computer Science”})).
\end{aligned}
$$

where P, A, S, and D are variables. □

The variables in O-terms are somewhat misleading. They are not logical variables intended to range over all objects of some types. A variable is intended to bind one

representational object semantically. It helps to view such a variables as telling the system to generate a surrogate. Each label is a single-valued function which relates the described object to another object. Note even though type names can be used, type definitions are not proposed.

Unlike LOGIN, there is no predicate in O-logic. O-terms can function not only as terms but also as atoms. This uniformity is certainly an advantage. Rules can be defined by O-terms alone. However, variables in a rule are different from variables just in O-terms. The variables in the body of a rule or in both the body and head are intended to range over all objects of some types. The variables only in the head are like the variables just in O-terms. If the body is satisfied, then it tells the system to generate a surrogate for the object described in the head.

Example 4.6 Consider an object-creating rule that forms an *interestingPair* object for each employee whose manager has the same name. This is defined by the following rule.

$$interestingPair{:}P(emp \rightarrow employee{:}E, mgr \rightarrow employee{:}M) \Leftarrow$$
$$employee{:}E(name \rightarrow string{:}N,$$
$$worksIn \rightarrow dept{:}D(manager \rightarrow employee{:}M(name \rightarrow string{:}N)).$$

In this rule, if the body is satisfied with some objects bound to E, N, D, and M, a unique object is intended to be created and bound to P. □

However, the object creating rules in O-logic are problematic. These rules themselves do not determine what objects should be created. For the above example, the problem is how P should be generated, since P does not occur in the body. The author recognized that P should be existentially quantified, with respect to other

variables in the rules. But the scope of the existential quantifier remains unspecified in the rule, there are several possible cases, $\forall E \forall M \forall N \forall D \exists P$ and $\forall E \forall M \exists P \forall N \forall D$, etc. each of which is reasonable and each of which has different semantics.

In summary, object identity and single-valued factual properties are supported in O-logic. But set-valued properties are not expressible. Types, inheritance, schema, set objects are not supported, let alone higher-order features. The semantics is not well-defined.

4.1.3 Revised O-Logic

Based on O-logic, Revised O-logic was presented by Kifer and Wu [KW89], as a solution to the problems of O-logic. A program in Revised O-logic consists of revised O-terms (here called E-terms for convenience) and rules. Like O-terms, E-terms can only be used to represent objects rather than types. O-terms are extended here to include sets (set objects). An E-term consists of a class, an object identifier, labels, sub-E-terms which can be single-valued or set-valued, and arrows.

Example 4.7 Consider the object *smith* in Example 3.5. It cannot be represented in O-logic at all. It can be indirectly represented in LOGIN using a ψ-term in Example 4.3 and two predicates in Example 4.4. But it can be directly represented by an E-term as follows.

$$
\begin{aligned}
student : smith[&name \rightarrow string : \text{“Smith”},\\
&age \rightarrow integer : 29;\\
&gender \rightarrow string : \text{“Male”},\\
&studiesIn \rightarrow dept : compSci,\\
&takes \rightarrow \{course : cs413,\ course{:}cs521\}\\
&borrows \rightarrow \{book : prolog, book : databases\}].
\end{aligned}
$$

where *smith, compSci, prolog* and *databases* are some kind of object identifiers. Since *compSci, prolog* and *databases* can be further described in the same way, complex objects are thus obtained. □

Unlike the semantic and object-oriented models, E-terms do not require set objects to be homogeneous. That is, a set object can consist of objects of different types, for example, $\{student : john, course : prolog\}$ is allowed.

The problems with object-creating rules in O-logic are solved here by introducing explicit skolem functions of existential variables called object constructors so that the quantification problem can be explicitly expressed by the user according to the intended semantics. For example, to represent the quantification $\forall X \forall Y \exists Z$, one can use $f(X, Y)$ directly to stand for the object Z. This is consonant with the system-generated surrogate representation of object identity. For the same X and Y, different object identifiers can be generated by using different object constructors.

Example 4.8 The object-creating rule in Example 4.6 can be expressed in Revised O-logic as follows.

$$interestingPair{:}f(E,M)[emp \rightarrow employee{:}E, manager \rightarrow employee{:}M] \Leftarrow employee{:}E[name \rightarrow string{:}N, worksIn \rightarrow dept{:}D[manager \rightarrow employee{:}M[name \rightarrow string{:}N]]].$$

where $f(E, M)$ is an object constructor. □

In the Revised O-logic, object identity, object generation, single-valued and set-valued factual properties are supported. Class and object are strictly distinct. But type definitions and inheritance are not supported and set-valued variables are still not allowed. It has been given a well-defined semantics, but it is quite complicated and unintuitive because inconsistent information is allowed by way of terms $p : \top$.

Example 4.9 In Revised O-logic, the following two E-terms are allowed:

$$student : smith[age \rightarrow integer : 29].$$
$$student : smith[age \rightarrow integer : 32].$$

The semantics is so designed that the following E-term is logically entailed by the above:

$$student : smith[age \rightarrow integer : \top].$$

where $\top$ is the meaningless object. The claimed advantage for this is that inconsistent knowledge about age only prevents a meaningful answer regarding *smith*'s age, but it does not affect the consistent parts of the database. □

4.1.4 F-Logic

In order to incorporate type definitions and inheritance in the revised O-logic, F-logic was proposed by Kifer, Lausen and Wu [KL89, KLW90]. In F-logic, there are three kinds of terms: *signature F-terms* which are used to define types with single-valued and set-valued definitional properties; *data F-terms* which are used to represent complex objects with single-valued and set-valued factual properties; *is-a F-terms* which are used to represent subtype-of relationship between types and asserted-of relationship between objects and types, and to enforce property inheritance. Like the revised O-logic, objects are still viewed as surrogates and the object-generating rules in Revised O-logic are extended so that classes can be generated using rules. But unlike the revised O-logic, here the set objects are required to be homogeneous. Also the ability to deal with incomplete information is abandoned.

Example 4.10 The type definitions in Example 3.9 and the object description in Example 3.5 are represented in F-logic by signature F-terms in Figure 4.1, data

$person[name \Rightarrow string;$
$\quad gender \Rightarrow sex;$
$\quad age \Rightarrow age;$
$\quad lives \Rightarrow string]$

$student[age \Rightarrow young;$
$\quad studiesIn \Rightarrow department;$
$\quad takes \Rightarrow\!\!\!\!\Rightarrow course;$
$\quad borrows \Rightarrow\!\!\!\!\Rightarrow book]$

Figure 4.1 Signature F-Terms.

F-terms in Figure 4.2, and is-a F-terms in Figure 4.3.

In the signature F-terms, single-valued and set-valued definitional properties are represented by $\Rightarrow$ and $\Rightarrow\!\!\!\!\Rightarrow$ respectively.

In the data F-terms, single-valued and set-valued factual properties are represented by $\rightarrow$ and $\rightarrow\!\!\!\!\rightarrow$ respectively.

All is-a F-terms determine a partial order ':' over all classes and objects. Classes and objects are not distinguished and are interweaved together by is-a F-terms. In *student* : *person*, *student* is a class; while in *smith* : *student*, *smith* is an object. Also not only subclasses but also objects inherit type properties. Based on is-a F-terms in Figure 4.3 and signature F-terms in Figure 4.1 *student* inherits all type

$smith[name \rightarrow$ "Smith";
$\quad age \rightarrow 29;$
$\quad gender \rightarrow$ "Male";
$\quad studiesIn \rightarrow compSci;$
$\quad takes \rightarrow\!\!\!\!\rightarrow \{cs413, cs521\};$
$\quad borrows \rightarrow\!\!\!\!\rightarrow \{prolog, databases\}]$

Figure 4.2 Data F-Terms.

student : *person*
smith : *student*
compSci : *department*
cs413: *course*
cs521: *course*
prolog : *book*
databases : *book*

age : *integer*
young : *age*
29 : *young*
sex : *string*
"Male" : *sex*
"Female" : *sex*

Figure 4.3 IS-A F-Terms.

properties of *person* and has its own type properties and further refines the *age* property of *person*. The object *smith* inherits all type properties of *student*. Therefore, $smith[name \Rightarrow string, takes \Rightarrow\!\!\!> course]$ is inherited by *smith*. Factual properties of objects are represented by data F-terms using $\rightarrow$ and $\rightarrow\!\!\!>$ and must satisfy their type constraints. □

F-logic is the most elaborate approach so far. It supports object identity, object generation, single-valued and set-valued definitional and factual properties, syntactic sets, types and inheritance. The object-generating rules in Revised O-logic is extended so that classes can be generated using rules.

However, F-logic does not support higher-order features in a natural way. It mixes the is-an-instance-of between objects and classes and is-a-subtype-of relationships between types. In F-logic, classes and objects are interwoven together. Objects can play dual roles: an instance of its class and a class of its instances. This special treatment of classes and objects makes F-logic have higher-order syntax but first-order semantics. Therefore, it is impossible to define an overall schema for the database in F-logic in the sense discussed in the last chapter. As a result, the queries in F-logic like ?– X : *person* is problematic. It may ask for objects of *person*. It

may also ask for subclasses of *person*. Given an answer a, the user has no idea if a is an object or a class. There is no way for the system to implicitly find out this kind of information. The proposed solution is to use symbols ! in front of objects and # in front of classes in all F-terms explicitly by the user which is inconvenient to the user. Objects and classes are different level concepts and have different functionality, there should be some simple way to distinguish them.

A second problem with F-logic is that set-valued variables are not allowed even though set objects can be used. To query a set object, the user has to use individual variables to get one answer at a time. For example, to query all the courses taken by Smith in F-logic, we have to use ?- $smith[takes \twoheadrightarrow \{X\}]$ where $\{X\}$ represents a set over which the variable X ranges. For this reason, nested sets cannot be supported. These certainly limit its expressive power. The reason for excluding them is also to avoid higher-order semantics.

A third problem is that its use of $\Rightarrow$, $\Rightarrow\!\!\!\!\!\!\Rightarrow$, $\rightarrow$ and $\twoheadrightarrow$ to represent properties makes it complicated. Even the factual properties of classes can be inherited by its objects. For example, if every student speaks at least English, then there can be a data F-term for the class *student* such that $student[speaks \twoheadrightarrow \{english\}]$ so that every instance of *student* can have this property. Such representation mixes definitional properties and factual properties.

Another minor problem is that subtypes of basic types cannot be defined in the way discussed earlier. For example, a subtype of integer cannot be represented as an interval without giving a name, such as [0..125]. Instead, the user has to use a lot of is-a F-terms, like $0 : age, 1 : age, ..., 125 : age$ explicitly. The semantics of F-logic is

quite complicated [Zan89].

Last, is-a F-terms and data and signature F-terms are separated which violates its goal of grouping data around objects. As in automatic theorem proving of first-order logic, negated information must be explicitly represented in the program. F-logic uses an existential quantification for object identifiers in clauses which makes the semantics different and more complex than the standard one of logic programming [Zan89]. Also, the intended semantics of its programs and their syntactic and semantic properties are not explored as in logic programming.

Criteria	LOGIN	O-Logic	R-Logic	F-Logic
Object is viewed as	surrogate	surrogate	surrogate	surrogate
Object Identity	surrogate	surrogate	surrogate	surrogate
Object Generation	No	Yes	Yes	Yes
Single-Valued Factual Property	function	function	function	function
Set-Valued Factual Property	prolog	No	function	function
Syntactic Sets	list	No	hetero	homo
Nested Sets	No	No	No	No
Set Variables	prolog	No	No	No
Separation of Classes and Objects	Yes	Yes	Yes	No
Uniformity of Terms and Atoms	No	Yes	Yes	Yes
Type Definitions	some	No	No	Yes
Subtypes of Basic types	Yes	No	No	some
Inheritance	Yes	No	No	Yes
Well-defined Semantics	No	No	complex	complex
Semantic Properties of Program	No	No	No	No

Table 4.1 Comparison of Extended Logic Term Approaches

4.1.5 Summary

This section has examined four typical extensions to traditional logic and logic programming, LOGIN, O-logic, Revised O-logic and F-logic, based on the concepts discussed in the last chapter. Table 4.1 summarizes the features of these extensions. Some abbreviations are used: function means function formation; prolog means the prolog approach; homo means sets are homogeneous; hetero means sets are heterogeneous; some means some of the desired functionalities are supported.

4.2 Extensions to Deductive Databases

Extended logic term approaches focus more on the complex object modeling than higher-order features. But higher-order features are natural requirement of deductive databases. In recent years, a few attempts have been made to deal with them, such as LDL, COL, IQL, LPS, L^2, etc. They are direct extensions to datalog by adding syntactic sets, schema operators and higher-order predicates. Complex object modeling is normally ignored in these approaches. This section analyses LDL, L^2, COL because they originate several important notions for higher-order features.

4.2.1 LDL

Based on traditional logic programming, LDL [BNST91, TZ86] proposes a way of expressing sets. Set terms can be generated in LDL by using two set constructors: set-enumeration and set-grouping. Set enumeration is the process of constructing a set by listing its elements. But in set-grouping, the set is constructed by defining its elements using a property (i.e., a conjunction of predicates) that they satisfy.

Example 4.11 Let the relation *book* include tuples whose first component is the title of a book, and whose second component is the price of the book. To derive a relation *bookDeal* on sets of three book titles from the *book* relation such that their total price is less than \$100, the following rule of LDL can be used:

$$bookDeal(\{X, Y, Z\}) \leftarrow book(X, P_x), book(Y, P_y), book(Z, P_z), \\ X \neq Y, X \neq Z, Y \neq Z, \\ P_x + P_y + P_z < 100.$$

Note that the same thing can not be directly represented in the standard logic programming language. □

Example 4.12 Given a parts-and-suppliers relation *supplier*, the following program of LDL uses the set-grouping constructor to group all parts each supplier supplies:

$$partSets(Supplier, \langle Part \rangle) \leftarrow supplier(Supplier, Part).$$

where $\langle ... \rangle$ in the head is the grouping operator. The intended meaning is to find, for each *Supplier*, all substitutions that satisfy the body, to collect the *Part*-values in them into a set, and to construct a tuple in the result from each *Supplier* and the corresponding set of *Part*'s. □

Introducing set constructors improves the expressive power of LDL to a great extent. Set variables and set constructors can be used to construct complicated nested sets.

Example 4.13 Consider the program P_1 in LDL:

$p(\langle X \rangle) \leftarrow q(X).$
$q(\langle Y \rangle) \leftarrow r(Y).$
$q(\langle Z \rangle) \leftarrow s(Z).$
$r(1).$
$s(2).$

This program involves sets {1}, {2}, {{1},{2}}. Note that none of the extended term approaches discussed have such expressive power.

Some restriction has to be imposed on programs of LDL in order to have well-defined semantics. In traditional logic programming with negation, every program has to be stratified so that there exists a minimal model that is, in a well-defined sense, preferable to all other models of the program, which can be used as the intended semantics of the program. Since set variables and set constructors are allowed in LDL, negation can be defined in terms of them. For example negation $\neg p(X)$, can be defined in terms of the set constructor as $q(\langle X \rangle) \Leftarrow p(X)$ and $q(Y), Y = \{\}$, where Y is a set variable. This means the stratification restriction on LDL programs should be extended for predicates involving set constructions. Otherwise, problems with negation are also problems here.

If a predicate p involves set constructions in the head and a predicate q is in the body, or p is in the head and a negation of a predicate q is in the body, then p depends-on q, denoted by $>$. This depends-on relation is transitive. A program is said to be *stratified* if there is no predicate p in the program such that p depends-on q, and q also depends-on p. The program in Example 4.12 is stratified.

Introducing set constructors also incurs problems. Extending stratification is not enough for LDL program. The well-known model-intersection property of traditional logic program fails in the context of LDL because of sets and grouping, with stratified program without using negation.

Example 4.14 Consider the following stratified program P_2 in LDL:

$q(2)$.
$p(\langle X \rangle) \leftarrow q(X)$.

This program may have the following models:

$M_1 = \{q(2), p(\{2\})\}$,
$M_2 = \{q(1), q(2), p(\{1,2\})\}$,
$M_3 = \{q(2), q(3), p(\{2,3\})\}$,
$M_4 = \{q(1), q(2), q(3), p(\{1,2,3\})\}$,

The intersection of the above models is $\{q(2)\}$ which 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 concept on top of the notion of minimality called *model dominate* which is somewhat unnatural [KW89]. For the above example, M_2 dominates M_4, M_3 dominates M_4, while M_1 dominates M_2 and M_3 and M_1 is the minimal model based on domination. □

LDL has a well-defined semantics. Important semantic properties of LDL programs are explored extensively. However, a number of unnatural concepts such as model domination are used which makes the semantics quite complicated [KW89]. As will be shown later, NLO includes all the expressive power of LDL but gets rid of its problems because of using attribute symbols.

LDL is based on traditional first-order logic, so the problems of complex object modeling discussed in the previous section still exist. Objects here are viewed as tuples and object identity is represented by object related values such as keys. Object properties, types are represented as in traditional deductive databases. Inheritance is not supported in the sense discussed earlier.

4.2.2 L^2

LDL extends traditional logic programming to include set constructors. But it cannot support schema and higher-order predicates in an integrated framework but relies on a separate language to specify the schema information and on evaluable predicates for expressing higher-order information. Furthermore, this schema information could not be used to compose a condition on the database. This is true for many other logic-based languages. To solve this problem, L^2 was proposed by Krishnamurthy and Naqvi [KN88].

Objects in L^2 are classified into four categories: atomic objects, functor objects, set objects and tuple objects. Atomic objects are basic objects discussed in the last chapter, such as integers, strings, etc. A functor object is recursively defined as an object of the form $f(object_1, ..., object_n)$, where f is an n-ary functor symbol. A set object is a named (not necessarily homogeneous) collection of objects. A tuple object is a sequence of named objects, whose names are called attributes, of the form $(attr_1 : object_1, ..., attr_k : object_k)$ in which each $attr_i : object_i$ pair refers to the $object_i$ that is the $attr_i$ attribute of the tuple and $object_i$ can be another tuple, a set object, a functor object, or an atomic object. Therefore, objects are viewed as tuples in L^2. Since attribute names can be used, the relations as sets of mappings approach can be used rather than relations as sets of lists.

A database is defined as a tuple object $database = (r_1 : s_1, r_2 : s_2, ..., r_n : s_n)$, whose attributes are relations, i.e., a set of tuple objects. A rule is of the form $p(...) \leftarrow X_1(...), ..., X_n(...)$ in which p is an attribute of *database* for a derived relation and each X_i is either an attribute of *database* or a variable ranging over all attributes

of *database*. Queries are then expressed over these relations.

Example 4.15 Suppose the database contains the following three relations: faculty, staff and student. Every relation has the same attributes: name, age, phone.

$$
\begin{aligned}
faculty : \{&(name : henry, age : 28, phone : 6320),\\
&(name : jenny, age : 41, phone : 5015),\\
&(name : peter, age : 25, phone : 1675)\},\\
staff : \{&(name : bev, age : 46, phone : 2518),\\
&(name : jone, age : 28, phone : 2518)\},\\
student : \{&(name : paul, age26, phone : 3108),\\
&(name : mary, age : 28, phone : 3538)\\
&(name : nancy, age : 24, phone : 4917)\}
\end{aligned}
$$

Then based on them, the following higher-order rules can be represented in L^2:

$$
\begin{aligned}
&(1).\ p_1(name : X) \leftarrow Y(name : X)\\
&(2).\ p_2(person : X, relation : Y) \leftarrow Y(name : X)\\
&(3).\ p_3(name : Z, relation : X) \leftarrow X(name : Z, phone : T),\\
&\qquad\qquad\qquad\qquad\qquad\quad Y(name : Z_1, phone : T), Z \neq Z_1\\
&(4).\ p_4(name : Z, relation : X) \leftarrow X(name : Z, phone : T),\\
&\qquad\qquad\qquad\qquad\qquad\quad Y(name : Z_1, phone : T), X \neq Y
\end{aligned}
$$

In the first rule, the relation p_1 is defined to include all the names of persons in faculty, staff, and student relations. In the second rule, the relation p_2 is defined to include all person names and their relation names. A query $?p_2(person : mary, relation : Y)$ computes mary's relation name, which is *student* as an answer. In the third rule, the relation p_3 includes the names of all persons and their relation names who share their phones. In the fourth rule, the relation p_4 includes the names of persons and the relation name who belong to different groups but share their phone. □

The schema information can be represented as a special tuple expression $p[metaexp]$ where p is a database attribute (i.e., relation name or predicate), *metaexp* is a

meta expression for schema information. The examples below show how to represent schema and use schema information.

Example 4.16 The following expression defines a schema for relation *person* to be a set of tuples, each of which contains two attributes, *names* and *age*, which are defined to be of *string* and *integer* type respectively.

$$person : [type = set]\{(name : [type = string], age : [type = integer])\}$$ □

Example 4.17 To compute all attribute names in the predicates of the database that are defined to be sets, irrespective of any value associated with these attributes can be achieved by the rule:

$$p_5(Y) \leftarrow X : (Y : [type = set])$$ □

Example 4.17 shows that meta information can be used in rules as well as queries in conjunction with normal data. This uniformity in the treatment of the schema information is achieved through the use of the higher-order predicates.

The solution to higher-order unification problem 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. The replacement semantics for higher-order variables in fact is of great value. It provides a technique for the integration of the

definition and manipulation of schema and data. In almost all deductive databases, only very limited higher-order features are used. Higher-order problems can actually be avoided based on the the replacement semantics. It is used by NLO as well.

The semantics of L^2 [KN88] is a simple extension of that of LDL except the addition of domains for function variables and predicate variables constructed out of the domain of individuals.

Certainly, L^2 is quite expressive in terms of higher-order features with a decidable unification algorithm. It gets rid of those problems of higher-order logic by restricting the language to be a decidable but expressive subset of higher-order logic for deductive databases. The semantics of a program is given by its minimal model. However, L^2 is not expressive in terms of complex object modeling. Object identity and inheritance issues are not addressed.

4.2.3 COL

The Complex Object Language (COL) [AG88] is an extension of Datalog by allowing function symbols in a way different from logic programming, which permits the manipulation of complex objects obtained using tuple and (heterogeneous) set constructors.

Terms are made not only of functions but also of set and tuple constructors. A term made of an n-ary function f and terms $t_1, .., t_n$ is, as usual, represented by $f(t_1, ..., t_n)$. A term made of a tuple constructor on terms $t_1, ..., t_m$ is represented by $[t_1, ..., t_m]$. A term made of a set constructor on terms $t_1, ..., t_k$ is represented by $\{t_1, ..., t_k\}$. Here a set can be heterogeneous. For example, [1, {2, mary}, f(2)], friend(mary), spouse(mary) are terms. Note objects in COL are viewed as tuples

like LDL and component objects are addressed by their positions.

Terms still function as terms, rather than as atoms like O-logic. Two particular predicates $=$ and $\in$ are built-in in the language and have a predefined interpretation, i.e., classical equality and membership. Formulas and rules are constructed in the usual way. But they are more complex because of the complex terms and the two built-in predicates.

Example 4.18 Following are examples of facts and rules:

(1). $parents(mary, \{bob, pam\})$;
(2). $address(mary, [3452, Golden, Calgary])$;
(3). $silly \in friends(mary)$;
(4). $phil = bestFriend(mary)$;
(5). $friend(X, Y) \leftarrow X \in friend(Y)$;
(6). $bestFriend(X) \in friends(X)$;
(7). $X \in \cup(Y, Z) \leftarrow X \in Y, X \in Z$;

The first fact says Mary's parents are Bob and Pam. The second fact tells mary's address. The third says silly is one of Mary's friends. The fourth says mary's best friend is phil. The fifth is a rule defining the predicate friend.. The sixth states that the best friend of somebody is also a friend of that person. The last rule defines set-valued function $\cup$, the usual set union. □

COL has a fundamental difference from traditional logic programming as well as LDL and L^2. It doesn't assume the freeness axiom,

$$f(X_1, ..., X_n) \neq g(Y_1, ..., Y_m)$$

for each pair f, g of distinct functions. So, $john = spouse(mary)$ can exist in a COL program. Certainly, COL gives a natural semantics to function symbols.

Like LDL, COL allows set variables, disallows function variables and predicate

variables, requires schema or types to be defined in a separate language, and represents object identity by object related names or keys. Inheritance is not supported.

4.2.4 Summary

This section has examined three typical extensions to deductive databases LDL, L^2 and COL, based on the concepts discussed in the last chapter. Table 4.2 summarizes the features of these extensions based on the same criteria in Table 4.1. Some abbreviations are used: relational means the relational approach, tuple form means the tuple formation, homo means sets are homogeneous, and hetero means sets are heterogeneous.

Criteria	LDL	L^2	COL
Object is viewed as	tuple	tuple	tuple
Object Identity	relational	relational	relational
Object Generation	relational	relational	relational
Single-Valued factual Property	tuple form	tuple form	tuple form
Set-Valued factual Property	tuple form	tuple form	tuple form
Syntactic Sets	hetero	hetero	homo
Nested Sets	Yes	Yes	Yes
Set Variables	Yes	Yes	Yes
Separation of Classes and Objects	Yes	Yes	Yes
Uniformity of Terms and Atoms	No	No	No
Type Definitions	relational	relational	relational
Subtypes of Basic types	No	No	No
Inheritance	No	No	No
Well-defined Semantics	Yes	Yes	Yes
Semantic Properties Explored	Yes	Yes	Yes

Table 4.2 Comparison of Extensions to Deductive Databases

4.3 Summary

Chapter 2 has shown that program clause logic is a natural generalization of the relational data model. This strong inherent connection results in the deductive databases which combine the benefits of both approaches, such as representational and operational uniformity, ease of use, deductive power, data independence, efficient secondary storage access, etc. These new kinds of systems extend the frontiers of computer science in an important direction and fulfill the needs of new applications. However, as Chapter 3 has shown, these kinds of deductive databases are quite limited in their expressive power. They cannot support in a direct and natural way complex object modeling which is a common requirement of advanced database applications. Also they cannot support objects and higher-order features such as schema and sets in a uniform way. Semantic and object-oriented data models are more expressive than the relational model. This chapter has shown that the typical extended logic term approaches and extended deductive database languages are not general enough to capture the most important ideas of these models. However, they tackle some of the problems successfully. Based on them, the rest of this thesis is dedicated to a novel deductive database language called NLO (Natural Logic for Objects). As will be seen, NLO is a natural generalization of semantic and object-oriented data models. It stands in the same relationship to the semantic and object-oriented data models as the program clause logic to the relational data model. It has expressive and deductive power and can naturally represent and manipulate complex objects and desired higher-order features in a uniform way.

Chapter 5

NLO – Informal Presentation

The present form of NLO does not deal with update issues formally, that is, how to insert an object, how to delete an object, and how to change attribute values of an object. Like LOGIN, F-logic, COL, LDL, it deals with what makes up an NLO program, how queries are represented, what properties an NLO program may have, what the precisely defined semantics of a program is, and what the correct answers to queries are.

It is assumed that all representational objects in NLO are represented by their identifiers. The generation of such identifiers has been discussed in chapter 3 but is not part of the formal semantics of NLO.

This chapter informally introduces NLO through a number of examples. It describes what makes up an NLO program, how queries are represented and what seem to be correct answers to queries.

5.1 Objects, Programs and Queries

In NLO, there are two kinds of objects: individual objects and set objects. Individual objects are classified according to their factual structural and behavioral properties. Classes are used to denote the sets of objects which share common factual properties and types are used to formalize definitional properties that constrain the factual

properties of all objects in the corresponding classes. Types and classes always have the same names. For individual objects, NLO takes the *asserted-of* approach. That is, an object o is of a type t only if it is asserted by $o : t$ explicitly or implicitly via inheritance or rules. An object possessing a type should have all the factual properties defined by the definitional properties of its type, but the factual attribute values may be unknown.

Set objects are formed out of individual objects and represented in the standard set formation. Correspondingly, a set type and set class are used for set objects. That is, if p is a type, then $set(p)$ is a set type. For set objects, NLO takes the *conforms-to* approach. That is, if $o_1, ..., o_n$ are individual objects of type p, then $\{o_1, ..., o_n\}$ represents a set object of type $set(p)$.

Types or classes themselves are meta individual objects which may have their own factual properties, besides their definitional properties. They belong to the meta type called *type* whose definitional properties constrain the factual properties of types or classes. Similarly, the meta type *type* is also an object and has its own factual properties. The type of *type* is defined to be itself.

Therefore, three disjoint kinds of objects function at three levels in any NLO program: ordinary objects at level one, types or classes at level two, and the unique meta type called *type* at level three.

Every object has a type. Every object may have definitional properties and factual properties. The definitional properties of an object o of type t are embraced in angle brackets, i.e., $o : t\langle ... \rangle$. The factual properties of an object o of type t are embraced in round brackets, i.e., $o : t(...)$.

In any NLO program, all information about objects at levels two and three is grouped into a type system. All information about objects at level one is divided into the object base and rules. Thus, an NLO program consists of three parts: a type system, an object base, and a set of rules. The object base contains all known information about objects. That is, all asserted representational objects and their factual properties. Rules are used to deduce the information which is not explicitly represented in the object base. Objects in the object base and inferred by rules must satisfy the type definitions in the type system in order to have meanings.

Example 5.1 Figure 5.1 shows a simple NLO program. The type system only contains one type definition for *person*. The type *person* has one factual property called *isa* which says that *person* is a subtype of the built-in type *object*. It has several definitional properties for its objects, *name*, *age*, *address*, etc. Note that we can use a subtype of the integers such as {0..125} in NLO.

The object base consists of seven asserted representational objects and their factual properties. Notice that these factual properties are all single-valued.

The rules of the program deduce factual properties of the objects in the object base. The first rule says that for each person X, his father or mother is one of his parents. Here $\{Y\}$ and $\{Z\}$ are called set grouping variables which represents sets of which Y and Z are elements respectively. The second rule says that for each person X, his parent Y is one of his ancestors. The third rule says that for each person X, an ancestor Y of his parent Z is his ancestor as well. The fourth rule say that for each person X, if he is not greater than 20 years old, then he lives with his father or mother Y. The last rule says that for each person X, he lives with his spouse Y. □

$$
\begin{aligned}
person : type(isa \rightarrow \{object\})\\
\langle name \rightarrow string,\\
age \rightarrow \{0..125\},\\
address \rightarrow string,\\
spouse \rightarrow person,\\
mother \rightarrow person,\\
father \rightarrow person,\\
parents \rightarrow set(person),\\
ancestors \rightarrow set(person)\rangle.
\end{aligned}
$$

(a). The Type System

$$
\begin{aligned}
&pam : person(spouse \rightarrow tom).\\
&tom : person(address \rightarrow \text{“257 9 Av NW”}).\\
&bob : person(father \rightarrow tom, mother \rightarrow pam,\\
&\qquad address \rightarrow \text{“128 2 St SW”}).\\
&liz : person(father \rightarrow tom).\\
&ann : person(age \rightarrow 20, father \rightarrow bob).\\
&pat : person(father \rightarrow bob, address \rightarrow \text{“439 5 Av NE”}).\\
&jim : person(age \rightarrow 16, mother \rightarrow pat).
\end{aligned}
$$

(b). The Object Base.

$$
\begin{aligned}
&X : person(parents \rightarrow \{Y\}) \Leftarrow X : person(father \rightarrow Y);\\
&\qquad\qquad X : person(mother \rightarrow Y).\\
&X : person(ancestors \rightarrow \{Y\}) \Leftarrow X : person(parent \rightarrow \{Y\}).\\
&X : person(ancestors \rightarrow \{Y\}) \Leftarrow X : person(parent \rightarrow \{Z\}),\\
&\qquad\qquad Z : person(ancestors \rightarrow \{Y\}).\\
&X : person(address \rightarrow Z) \Leftarrow (X : person(age \rightarrow A, father \rightarrow Y);\\
&\qquad\qquad X : person(age \rightarrow A, mother \rightarrow Y)),\\
&\qquad\qquad Y : person(address \rightarrow Z),\ A \leq 20.\\
&X : person(address \rightarrow Z) \Leftarrow X : person(spouse \rightarrow Y),\\
&\qquad\qquad Y : person(address \rightarrow Z).
\end{aligned}
$$

(c). Rules

Figure 5.1 A Sample NLO Program.

Queries can be defined over the object base, rules, and type system to obtain information about objects, types.

(*Query 1.*) ?– $person : type\langle A \rightarrow T \rangle$.
(*Answer 1.1.*) $A = name, T = string$.
(*Answer 1.2.*) $A = age, T = \{0..125\}$
...

(*Query 2.*) ?– $person : type(isa \rightarrow S)$.
(*Answer 2.1.*) $S = \{object\}$.

(*Query 3.*) ?– $student : type$.
(*Answer 3.1.*) *no.*

(*Query 4.*) ?– $bob : person(A \rightarrow pam)$.
(*Answer 4.1.*) $A = mother$.

(*Query 5.*) ?– $X : person(ancestors \rightarrow S)$.
(*Answer 5.1.*) $X = bob, S = \{tom, pam\}$.
(*Answer 5.2.*) $X = liz, S = \{tom\}$.
(*Answer 5.3.*) $X = ann, S = \{tom, pam, bob\}$.
(*Answer 5.4.*) $X = pat, S = \{tom, pam, bob\}$.
(*Answer 5.5.*) $X = jim, S = \{tom, pam, bob, pat\}$.

(*Query 6.*) ?– $X : T(address \rightarrow A)$.
(*Answer 6.1.*) $X = pam, T = person, A =$ *"257 9 Av NW"*.
(*Answer 6.2.*) $X = tom, T = person, A =$ *"257 9 Av NW"*.
(*Answer 6.3.*) $X = bob, T = person, A =$ *"128 2 Av SW"*.
(*Answer 6.4.*) $X = ann, T = person, A =$ *"257 9 Av NW"*.
(*Answer 6.5.*) $X = pat, T = person, A =$ *"439 5 Av NE"*.
(*Answer 6.6.*) $X = jim, T = person, A =$ *"439 5 Av NE"*.

Figure 5.2 Sample Queries and Answers.

Example 5.2 Figure 5.2 shows several queries and the corresponding answers to them based on the program in Figure 5.1. The queries 1 to 3 ask about the type system. The first query asks the definitional properties of the type *person*. The

second query asks the factual property *isa* of the type *person*. The third query asks if there is a type *student* in the type system. The answers to them are directly in the type system. The rest of the queries ask for information in the object base and deduced from the rules. The fourth query asks what kind of relationship *bob* and *pam* have, that is, whether or not there is an attribute which links *bob* to *pam*. The fifth query asks for each person X, his ancestors. The answers to it are deduced from the rules in the program. The last query asks what type of object has the attribute *address* and what the attribute value is. The answers to it are either directly in the object base or derivable from the rules. □

The above simple program and queries give a flavor of programming in NLO. Following sections elaborate what can be represented in the type system, the object base, rules and queries.

5.2 Type System

There are four kinds of types in NLO: set types, built-in types, basic types, representational types.

For every type t, its set type $set(t)$ is automatically defined and the corresponding class is the power set of the class t, that is $set(t) = 2^t$. NLO extends the semantic and object-oriented data models by allowing nested set types to be automatically defined, for example, $set(set(set(t)))$.

The built-in types are *integer*, *string*, and *object*. These built-in types have no factual structural properties, but only factual behavioral properties in the sense discussed in Chapter 3. The classes *integer* and *string* are intended to include all

possible integers and strings respectively and do not need to be explicitly asserted. The class *object* is intended to include all representational objects and is therefore application dependent. It automatically includes all representational objects that are in its subclasses because of inheritance.

Basic types are used to define subtypes of *integer* or *string* either by enumerating or by specifying the ranges. Subtypes may or may not have names. For example, {0..125} specifies a subtype of integer without a name, whose class contains integers between 0 and 125; while *gender* = { *"Male", "Female"*} specifies a subtype of string called *gender*, whose class has only two elements *"Male"* and *"Female"*.

A representational type is a type for representational objects and is a named subtype of *object*. Like the semantic and object-oriented data models, NLO uses the symbol *isa* to represent subtype relationships between types as well as subclass or subset relationships between classes. It is used as a set valued factual attribute which relates one type to its immediate supertypes. Let t be a type and $s_1, ..., s_m$ are its immediate supertypes. It is asserted in NLO as

$$t : type(isa \rightarrow \{s_1, ..., s_m\})$$

The result of such a representation is profound. The subtype relationships between types represented by the *isa* attribute is a partial order. The type t inherits whatever properties each s_i may have of its own or inherited from its supertypes and the class t is included in each class s_i. Also the set class $set(t)$ is automatically included in each set class $set(s_i)$, so are $set(set(t))$, ..., $set(...(set(t))...)$.

Using the *isa* attribute turns out to be syntactically expressive and semantically sound.

A representational type may have definitional properties, besides the factual property called *isa*. The definitional properties of a type constrain the factual properties of its objects at one level lower. A definitional property of a type formally defines the attribute as a mapping by letting the class corresponding to the defined type be the range and some class as the domain of the attribute. If p is a type with a definitional attribute l and the domain of l is q, then this is represented in NLO by $p : t\langle l \rightarrow q\rangle$, and l is called a *definitional attribute* of p and q is called the *definitional attribute value* of the definitional attribute l of p.

There are two kinds of definitional attributes, single-valued and set-valued. A single-valued definitional attribute relates a non-set type to another non-set type, while a set-valued definitional attributes relates a non-set type to a set type as the program in Figure 5.1 shows.

All definitional properties applicable to a type are normally grouped together. That is, if a type p has definitional attributes $l_1, ..., l_n$ and definitional attribute values $p_1, ..., p_n$, then this is normally represented as $p : t\langle l_1 \rightarrow p_1, ..., l_n \rightarrow p_n\rangle$.

If a representational type has definitional properties defined by $p : t\langle l_1 \rightarrow p_1, ..., l_n \rightarrow p_n\rangle$. and factual properties defined by $p : t(isa \rightarrow \{q_1, ..., q_m\})$. then it is normally represented in the type system as

$$p : t(isa \rightarrow \{q_1, .., q_m\})\langle l_1 \rightarrow p_1, ..., l_n \rightarrow p_n\rangle. \quad (1)$$

The type *person* in Figure 5.1 is defined in this way.

Example 5.3 Figure 5.3 shows another example of a sample type system which defines 4 types. Note that the two type definitions PERSON and STUDENT in Example 3.9 are defined in NLO here. Based on the definitions, the type *student*

$$
\begin{aligned}
&person : type(isa \rightarrow \{object\}) \\
&\qquad \langle name \rightarrow string, \\
&\qquad\ sex \rightarrow gender, \\
&\qquad\ age \rightarrow \{0..125\}, \\
&\qquad\ address \rightarrow string\rangle.
\end{aligned}
$$

$$
\begin{aligned}
&student : type(isa \rightarrow \{person\}) \\
&\qquad \langle age \rightarrow \{15..35\}, \\
&\qquad\ studiesIn \rightarrow dept, \\
&\qquad\ takes \rightarrow set(course), \\
&\qquad\ borrows \rightarrow set(book)\rangle.
\end{aligned}
$$

$$
\begin{aligned}
&employee : type(isa \rightarrow \{person\}) \\
&\qquad \langle age \rightarrow \{20..65\}, \\
&\qquad\ worksIn \rightarrow dept, \\
&\qquad\ heads \rightarrow set(employee)\rangle.
\end{aligned}
$$

$$workStudent : type(isa \rightarrow \{student, student\})$$

Figure 5.3 A Sample Type System.

inherits all properties of *person* because of the factual attribute *isa* and refines the inherited property *age*. So is the type *employee*. The type *workStudent* multiply inherits all properties of *student* and *employee*. It is noticeable that the type representation of NLO is quite similar to TAXIS syntactically. □

The type system of NLO consists of all type information for the object base and rules. It is also a meta-object base which can be queried in the same way as the normal object base.

The factual properties of representational types are constrained by the meta type *type* which is predefined in the type system as follows:

$$type : type(isa \rightarrow \{type\})\langle isa \rightarrow set(type)\rangle.$$

5.3 Object Base

In NLO, individual objects are classified into three disjoint classes, *integer, string* and *object.* The classes of *integer* and *string* are explicitly defined. The class of *object* is implicitly defined. The classes of basic types which are subclasses of *integer* and *string* are explicitly defined. The classes of set types are implicitly defined. Only the classes of representational types, are application dependent and their contents determine some implicitly defined classes.

Every representational object o of type t has to be explicitly asserted by $o : t$.

A representational object may have and can only have factual properties. A factual property of an object represents a relationship this object has with another object. Such a relationship is called a *factual attribute* which is constrained by the definitional attribute of the corresponding type. The related object is called the *factual attribute value* of the corresponding factual attributes.

There are two kinds of relationships in NLO between objects, complete and partial. If o is an object of type t and is related *completely* to an object o_s via the factual attribute label l, then this is represented in NLO by $o : t(l \rightarrow o_s)$. It means that o_s is the complete factual attribute value of l on the object o, that is, $l(o) = o_s$. If o is an object of type t and is related partially to an object o_s via the factual attribute label l, then this is represented in NLO by $o : t(l \rightarrow o'_s)$. It means that o_s is only a partial factual attribute value of l on the object o, that is, $l(o) \supseteq o_s$.

Since a definitional attribute can be either single-valued or set-valued, the factual attribute can also be single-valued or set-valued correspondingly. A single-valued factual attribute relates an individual object to another individual object completely.

A set-valued factual attribute relates an individual object to a set object either partially or completely.

It is possible to have both $o : t(r \rightarrow \{o_{1,1}, ..., o_{1,m}\}')$ and $o : t(r \rightarrow \{o_{2,1}, ..., o_{2,n}\}')$ so that $r(o) \subseteq \{o_{1,1}, ..., o_{1,m}\} \cup \{o_{2,1}, ..., o_{2,n}\}$. Such interpretation for set-valued attributes can naturally represent many important applications, see next section.

Example 5.4 Following are several examples of objects and their factual properties.

$smith : person(name \rightarrow$ "$Smith$").
$smith : person(age \rightarrow 29)$.
$mary : student(takes \rightarrow \{cs413,\ cs521\})$.
$jenny : person(friends \rightarrow$ {bob, silly, phil}').

Here, *name* and *age* are single-valued factual attributes and *takes* and *friends* are set-valued factual attributes. The factual attribute value $\{cs413,\ cs521\}$ of *takes* is complete, while the factual attribute value $\{bob, silly, phil\}$ of *friends* is partial. □

An object o with a complete factual property represented by $o : t(l \rightarrow o_s)$ is called *well-typed* if t has a definitional property defined by $t : type\langle l \rightarrow s\rangle$ and o_s is an object of s.

An object o with a partial factual property represented by $o : t(l \rightarrow o'_s)$ is called *well-typed* if t has a definitional property defined by $t : type\langle l \rightarrow s\rangle$ and o_s is part (subset) of an object of s.

All factual properties applicable to an object are normally grouped together. That is, if an object o of type p has factual attributes $l_1, ..., l_n$ and factual attribute value $o_1, ..., o'_n$, then this is normally represented as

$$o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow o'_n). \tag{2}$$

Note the *isa* attribute is a meta-attribute, that is, it can only be used to relate types with special meanings. Therefore it cannot be used for normal objects.

Example 5.5 The object in Example 3.5 is represented in NLO as a representational object as follows.

$$\begin{aligned} smith : student(&name \rightarrow \text{“Smith”},\\ &age \rightarrow 29,\\ &sex \rightarrow \text{“Male”},\\ &studiesIn \rightarrow compSci,\\ &takes \rightarrow \{cs413,\ cs521\},\\ &borrows \rightarrow \{prolog, databases\}). \end{aligned}$$ □

The object base consists of all asserted representational objects of the form (2) which must satisfy the corresponding type definitions in order to have semantics.

Note that the object base only tells what is known about the values of factual attributes of objects. The unknown factual attribute values can be inferred directly from the type system.

Example 5.6 Consider the object *smith* in the example above. It satisfies the type definition of type *student* in Example 5.3. The factual attribute value of *address* can be inferred unknown. □

5.4 Rules

Based on the object base, deductive information can be defined by using rules in NLO.

Rules are defined in terms of NLO-terms and comparison expressions.

Similar to objects, variables in NLO can be either *individual variables* or *set variables* depending on whether the attributes are single-valued or set-valued (as defined by the type system). Set variables can be either *set-valued variables* or *set grouping variables* of the form $\{X\}$. A set grouping variable $\{X\}$ represents a set over which the variable X ranges. Here X itself is called a set element variable which can be either an individual variable and the set $\{X\}$ is then a set of individuals, or a set-valued variable and the set $\{X\}$ is then a set of sets, not both. A set grouping variable $\{X\}$ is similar to a set grouping variable $\langle X\rangle$ in LDL, see Example 4.11 and Example 4.12. A set grouping variable $\{X\}$ in NLO can appear in the head and the body of a rule, see examples below, but a set grouping variable $\langle X\rangle$ in LDL can only be in the head of a rule. However the set $\langle X\rangle$ in LDL can be a set of individuals and sets which makes the semantics more complicated.

Following the usual conventions in logic programming, variables are represented by upper-case letters and objects are represented by lower-case words in NLO.

To deal with object creation, NLO uses the method of the Revised O-Logic, that is, using explicit skolem functions of existential variables called object constructors. An object constructor is of the form $f(Y_1, ..., Y_n)$, where f is a skolem function symbol, each Y_i is either a variable, an object, or a simpler object constructor. For example, $i(a, f(Y, b), \{c, a\})$ and $g(X, h(c, \{Z\}))$ are object constructors, where i, f, g, h are skolem function symbols. In NLO, skolem functions are treated differently from attributes. Skolem functions are uninterpreted as the functions in logic programming, while attributes are interpreted.

An NLO-term is similar to a representational object of the form (2), but variables

and object constructors may be in place of the objects $o, o_1, ..., o_n$. An NLO-term is of the form

$$X : t(l_1 \rightarrow X_1, ..., l_m \rightarrow X_m) \tag{3}$$

where $X, X_1, ..., X_m$ are either variables, object constructors, or objects.

Example 5.7 Based on the type definitions in Example 5.1, following are several NLO-terms.

(1). $X : person$.
(2). $mary : person(name \rightarrow X, age \rightarrow Y)$.
(3). $f(X) : person(gender \rightarrow Y)$.
(4). $X : person(takes \rightarrow \{Y\})$.
(5). $X : person(borrows \rightarrow S)$.

Here X and Y are individual variables, S is a set-valued variable, $\{X\}$ is a set grouping variable, and $f(X)$ is an object constructor. □

A comparison expression in NLO is either a traditional arithmetic comparison between individual objects and individual variables using the operators $=$, $>$, $\geq$, $<$, $\leq$ $+$, $-$, $\times$ and $\div$, or a traditional set comparison expression between set objects and set variables using the operators $=$, $\supset$, $\supseteq$, $\subset$, $\subseteq$, $\cup$, $\cap$, and $/$. Note that whether or not an operator is applicable to some objects is fully determined by the types of the objects. For example, $+$, $-$, $>$, etc., are applicable only to integers.

Example 5.8 Following are example of comparison expressions in NLO.

(1). $X = Y \times 2$
(2). $X > 50$
(3). $X \leq 100$
(4). $Y =$ *"Mary"*
(5). $S = S_1 \cup S_2$

(6). $S_2 = S_3/S_4$
(7). $S_1 \subseteq S_2$

Here the first four expressions are arithmetic and the rest are set. □

A rule in NLO is of the form $A \Leftarrow L_1, ..., L_n$, where A is called the head of the rule which is an NLO-term and $L_1, ..., L_n$ is called the body of the rule, every L_i in the body of the rule is either an NLO-term, a negation of an NLO term or a comparison expression, and every variable occurs in the head also occurs in the body.

A rules can be used in two different ways. One is to deduce factual attribute values of existing representational objects. The other is to construct new representational objects and deduce their attribute values. In the later case, object constructors have to be used in the head of the rule.

The rules in Figure 5.1 are used to deduce factual attribute values for existing objects.

Example 5.9 Following are two object creation rules. The first one is the NLO version of the object creation rule in Example 4.7. The second one shows how sets of sets are formed by set grouping where S is a set-valued variable.

$$f(E, M) : interestingPair(worker \rightarrow E, manager \rightarrow M) \Leftarrow E : employee(name \rightarrow N,\ worksIn \rightarrow D), D : dept(manager \rightarrow M), M : employee(name \rightarrow N).$$

$$h(X, \{Y\}) : t(att_1 \rightarrow \{S\}) \Leftarrow X : s(att_2 \rightarrow \{Y\}), Y : s(att_3 \rightarrow S).$$ □

Note that in NLO, relationships between objects are represented by either single-valued attributes or set-valued attributes, while in traditional logic programming, they are normally represented by predicates which is always set-valued. This is one

of the reasons that NLO programs may not be satisfiable while traditional logic programs are always satisfiable. For example, the *gender* attribute of *person* in Example 5.2 is defined as single valued and if we infer someone whose gender is both female and male, then this program is not satisfiable.

Another reason for a rule not being satisfiable is that some factual attribute values which are not well-typed are inferred for some objects. For example, if the *age* attribute of *student* is defined from 15 to 35 and we infer someone whose age is 36.

Like traditional logic programming and LDL, rules in NLO have to be stratified in order to have a well-defined semantics. Stratification is based on the *depends-on* relation between types and attributes. If a type t involves set objects or set-valued variables in the head and another type s is in the body, or t is in the head and a negation of another type s is in the body, or s involves set objects or set-valued variables, then t is said to depend on s. If t depends on s, then t depends on all attributes of s, all attributes of t depends on s, and all attributes of t depends on the attributes of s. Within a type, if an attribute att_p involves set-valued variables in the head and another attribute att_q is in the body, or att_p is in the head and a negation of an attribute att_q is in the body, then att_p is said depends on att_q. This depends-on relation is transitive. A set of rules is said stratified if there is no attributes or type p, q such that p depends-on q, and q also depends-on p. Example 5.1 and 5.9 are stratified rules. Complex examples are given in Chapter 7.

5.5 Queries

Queries are defined over the object base, rules, and the type system in a uniform way in NLO.

Queries are defined in terms of NLO-terms, typed NLO-terms and comparison expressions.

A typed NLO-term is similar to an NLO-term of the form (3), but variables may be in places of the types and attributes. A typed NLO-term is one of the following form

$$
\begin{aligned}
&P : type\langle A_1 \rightarrow P_1, ..., A_m \rightarrow P_m \rangle \\
&P : type(isa \rightarrow Q) \\
&P : type(isa \rightarrow \{Q\}) \\
&P : type(isa \rightarrow \{q_1, ..., q_m\}') \\
&X : P(A_1 \rightarrow Y_1, ..., A_m \rightarrow Y_m)
\end{aligned}
$$

where $P, P_1, ..., P_m$ are variables or types, $A_1, ..., A_m$ are variables or attribute, Q is a variables or a set of types, and $X, Y_1, ..., Y_n$ are individual variables, set variables, object constructors, or objects.

A query is of the form ?- $L_1, ..., L_n$ where every L_i is either an NLO-term, a typed NLO-term, a negation of an NLO term or a typed NLO-term, or a comparison expression.

Queries are used to ask for information which is either in the object base, derivable from the rules based on the object base, or in the type system.

Example 5.10 Figure 5.4 shows some further examples of queries and the corresponding answers based on the previous examples. The first query asks the definitional properties of *workStudent* either directly defined or inherited. Note that the

(*Query 1.*) ?- $workStudent : type(L \rightarrow P)$.
(*Answer 1.1.*) $L = name, P = string$.
(*Answer 1.2.*) $L = sex, P = gender$.
(*Answer 1.3.*) $L = age, P = \{20..35\}$.
(*Answer 1.4.*) ...

(*Query 2.*) ?- $student : type(isa \rightarrow \{employee\}')$.
(*Answer 2.1.*) *no.*
(*Query 3.*) ?- $workStudent : type(isa \rightarrow \{X\})$.
(*Answer 3.1.*) $X = student$.
(*Answer 3.2.*) $X = employee$.

(*Query 4.*) ?- $X : P(borrows \rightarrow \{X\})$.
(*Answer 4.1.*) $X = smith, P = student, X = prolog$.
(*Answer 4.2.*) $X = smith, P = student, X = databases$.

(*Query 5.*) ?- $smith : T(L \rightarrow \{cs521\}')$.
(*Answer 5.1.*) $T = student, L = takes$.

(*Query 6.*) ?- $X : T(takes \rightarrow S, age \rightarrow Y), Y > 25$.
(*Answer 6.1.*) $X = smith, T = student, S = \{cs413, cs521\}, Y = 29$.

Figure 5.4 Sample Queries and Answers.

definitional attribute value of *age* is the intersection of definitional attribute values of *age* of *student* and *employee*. The second asks if *employee* is one of the supertypes of *student*. The third asks what types *workstudent* is a subtype of. The answers to these three queries are based on Figure 5.3. The fourth asks under what type of object X has the factual attribute *borrows* and what the factual attribute values are. The fifth asks under what type and what factual attribute the object *smith* has a factual attribute value *cs521*. The last asks under what type of object X has the factual attribute *takes* and *age*, and the attribute value of *age* is greater than 25 The answers to them are based on Example 5.5. □

5.6 Summary

Objects in NLO are viewed as surrogates and therefore object identity can be represented by the objects themselves. Objects can be generated in rules by using skolem functions in the same way as F-logic. Single-valued properties and set-valued properties are represented by attributes. Homogeneous syntactical sets can be used to represent set-valued properties of objects directly. Types and inheritance are supported in the natural way discussed in Chapter 3. To make type definitions more meaningful, subtypes of basic types can be used. Since types and objects are two fundamentally different concepts and used differently, they are distinct here in contrast to F-logic. However, types are also meta-objects so that objects and types are uniformly represented. Terms in NLO can function as either terms or atoms. The attributes are interpreted as mappings. To represent and manipulate schema and sets, variables for not only individuals, but also types, attributes and syntactic sets can be used. Since types and objects are uniformly represented, higher-order queries and normal queries can also be uniformly represented. The higher-order problems with such usage is avoided by taking the replacement semantics of L^2. Compared to other approaches, the semantics of NLO is quite simple, direct and natural.

Chapter 6

Formal Presentation

This chapter defines the formal syntax and semantics of NLO, The syntax is concerned with valid programs and queries admitted by the grammar of NLO. The semantics is concerned with the meanings attached to the valid programs and the symbols they contain as well as answers to the queries.

6.1 Syntax of NLO

This section introduces the syntax of NLO, i.e., its alphabet, type systems, object bases, terms, rules and queries. Some of the definitions rely on the definitions below.

Definition 6.1 The *alphabet* of NLO consists of eleven classes of symbols:

(1). the set $\mathcal{A} = \{type\}$;
(2). the set $\mathcal{B} = \{object, integer, string\}$;
(3). a countably infinite set $\mathcal{T}$ of type symbols;
(4). a countably infinite set $\mathcal{Z}$ of integers;
(5). a countably infinite set $\mathcal{S}$ of strings;
(6). a countably infinite set $\mathcal{O}$ of object identifiers;
(7). a countable infinite set $\mathcal{L}$ of symbols for attributes labels containing *isa*;
(8). a countably infinite set $\mathcal{F}$ of function symbols containing *set*;
(9). a countably infinite set $\mathcal{V}$ of variable symbols;
(10). $\Leftarrow$; $\leq, \geq, <, >$; $\subseteq, \supseteq, \subset, \supset$; $=$, and $\neg$;
(11). $+, -, \times, \div$; $\cup, \cap, \setminus$; $(,)$, $\langle,\rangle$, $\{,\}$, $\rightarrow$, comma, dot, semicolon, ‘, ’, ′, :, ?–.

Here the sets $\mathcal{A}, \mathcal{B}, \mathcal{T}, \mathcal{Z}, \mathcal{S}, \mathcal{O}, \mathcal{L}, \mathcal{F}, \mathcal{V}$ and symbols in (10) and (11) are assumed to be pairwise disjoint. □

Definition 6.2 The *types* of NLO are defined as follows:

(1). The *type* in $\mathcal{A}$ is a type, called a *meta type.*

(2). Every element of $\mathcal{B}$ is a type, called a *built-in type.*

(3). A basic type is a type and a representational type is a type.

(4). If p is a type, then $set(t)$ is a type, called a *set type.* □

By the above two definitions, if p is a type, then $set(p)$, $set(...(set(p)))$, etc. are all types.

Definition 6.3 The *basic types* of NLO are defined as follows:

(1). $\{a_1, ..., a_n\}$ defines a basic type which is named by itself, where $a_1, ..., a_n \in \mathcal{S}$, or $a_1, ..., a_n \in \mathcal{Z}$, $(n \geq 1)$.

(2). Let t be a symbol in $\mathcal{T}$ and $a_1, ..., a_n \in \mathcal{S}$, or $a_1, ..., a_n \in \mathcal{Z}$, $(n \geq 1)$, then $t = \{a_1, ..., a_n\}$ defined a basic type t.

(3). $\{lb..rb\}$ defines a basic type named by itself, where $lb, rb \in \mathcal{Z}$, and lb and rb are called the *left bound* and the *right bound* of the range of the basic type $\{lb..rb\}$ respectively.

(4). Let t be a symbol in $\mathcal{T}$ and $lb, rb \in \mathcal{Z}$, then $t = \{lb..rb\}$ defines a basic type t, and lb and rb are called the *left bound* and the *right bound* of the range of the basic type t respectively. □

Definition 6.4 Let p belong to $\mathcal{T}$, l be a label of $\mathcal{L}$ and q be a type. Then p : $type\langle l \rightarrow q \rangle$ defines a *representational type* p and a *definitional property* of the type p. The label l is called a *definitional attribute* of p and q is called the *definitional attribute value* of l of the type p. □

The definitional property of a representational type is used to constrain factual properties of the representational objects.

Definition 6.5 Let p belong to $\mathcal{T}$, $q_1, ..., q_m, m \geq 1$ be types. Then $p : type(isa \rightarrow \{q_1, ..., q_m\})$ defines a *representational type* p and a *factual property* of p called *isa*. Each $q_i, 1 \leq i \leq m$ is called a *supertype* of p. □

It is intended that a representational type inherits all the definitional properties of its supertypes (if any) but may redefine them and may have its own properties. In other words, a representational type may have several definitional properties of the same name. For example, *student* can have both $\langle age \rightarrow \{0..120\}\rangle$ and $\langle age \rightarrow \{15..35\}\rangle$. This will be reflected in the semantics in the next section.

Definition 6.6 $p : type(isa \rightarrow \{q_1, ..., q_m\})\langle l_1 \rightarrow p_1, ..., l_n \rightarrow p_n\rangle$ stands for $p : type(isa \rightarrow \{q_1, ..., q_m\})$ and $p : type\langle l_1 \rightarrow p_1\rangle$, ... $p : type\langle l_m \rightarrow p_m\rangle$. □

Definition 6.7 The *type system* S of NLO consists of a finite set of type definitions and property definitions according to the definitions 6.3 to 6.6. □

Types in NLO are meta-objects and therefore have a type. The type for types is the meta-type *type*. Every representational type is intended to be a subtype of the built-in type *object*.

Definition 6.8 The meta type *type* and the built-in type *object* have following definitions:

(1). $type : type(isa \rightarrow \{type\})\langle isa \rightarrow set(type)\rangle$.
(2). $object : type(isa \rightarrow \{object\})$ □

The definitional property of the meta-type *type* is used to constrain factual properties of types. Note that the factual property *isa* of *object* is constrained by the definitional property *isa* of *type*.

Definition 6.9 The *objects* of NLO are defined as follows:

(1). Every element of $\mathcal{Z}$ and $\mathcal{S}$ is an object, called a *basic object.*

(2). Every element of $\mathcal{O}$ is an object, called called a *representational object.*

(3). If f is a n-ary function symbol from $\mathcal{F}$ other than *set* and $o_1, ..., o_m$ are objects, then $f(o_1, ..., o_m)$ is an object, called a *representational object.*

(4). If $o_1, ..., o_n$ are objects, then $\{o_i, ..., o_n\}$ is an object, called a *set object* $\{\}$ denotes the empty set object. □

By this definition, not only individual objects, both also set objects, set set objects, etc are allowed in NLO.

Definition 6.10 Let p be a representational type, $l \in \mathcal{L}$, $o \in \mathcal{O}$ and o_s be an object. Then $o : p(l \rightarrow o_s)$ defines a *full factual property* for the representational object o of the type p, which is $(l \rightarrow o_s)$, and l is called a *factual attribute* of o and o_s is called the *full factual attribute value* of l of the object o. □

Definition 6.11 Let p be a representational type, $l \in \mathcal{L}$, $o \in \mathcal{O}$ and o_s be a set object. Then $o : p(l \rightarrow o'_s)$ defines a *partial factual property* for the representational object o of type p, which is $(l \rightarrow o'_s)$, and l is called a *factual attribute* of o and o_s is called the *partial factual attribute value* of l of the object o. □

It is intended that an object can only have a unique full factual attribute value for a factual attribute, but may have several partial factual attribute values for a factual attribute only if it does not have a full factual attribute value for this attribute. This

will be reflected in the semantics in the next section. The partial factual attribute values are used for partly known attributes values.

Definition 6.12 $o : p(l_1 \rightarrow o_1, ..., l_m \rightarrow o_m, l_{m+1} \rightarrow o'_{m+1}, ..., l_n \rightarrow o'_n)$ stands for $o : p(l_1 \rightarrow o_1), ..., o : p(l_m \rightarrow o_m)$. $o : p(l_{m+1} \rightarrow o'_{m+1}), ..., o : p(l_n \rightarrow o'_n)$. □

Definition 6.13 An *object base* of NLO consists of a finite set of factual properties of representational objects, according to the definitions 6.10 to 6.12. □

The rest of this section defines rules and queries which are based on terms.

Definition 6.14 The *variables* of NLO are defined as follows:

(1). Every element of $\mathcal{V}$ is a variable.

(2). If X is a variable then $\{X\}$ is also a variable, called a *set grouping variable.* □

Definition 6.15 An *object constructor* is defined inductively as follows:

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

Definition 6.16 The *basic terms* are defined as follows:

(1). If p is a representational type, X is either a variable, an object, or an object constructor, then $X : p$ is a basic term.

(2). If p is a representational type, l is a label, X is either a variable, a representational object or an object constructor, and Y is either a variable, an object, or an object constructor, then $X : p(l \rightarrow Y)$ is a basic term.

(3). If p is a representational type, l is a label, X is either a variable, a representational object or an object constructor, and Y is either a variable, an object, or an object constructor, then $X : p(l \rightarrow Y')$ is a basic term.

(4). If $X : p(l_1 \rightarrow X_1)$, ... $X : p(l_n \rightarrow X_n)$ are basic terms, then $X : p(l_1 \rightarrow X_1, ..., l_n \rightarrow X_n)$ is a basic term. □

Definition 6.17 An arithmetic expression is recursively defined as follows.

(1). A variable or an object in $\mathcal{Z}$ is an arithmetic expression.

(2). If δ_1 and δ_2 are arithmetic expressions, then $\delta_1 + \delta_2$, $\delta_1 - \delta_2$, $\delta_1 \times \delta_2$, $\delta_1 \div \delta_2$ are arithmetic expressions.

(3). If δ is an arithmetic expression, then (δ) is an arithmetic expression. □

Definition 6.18 A set expression is recursively defined as follows.

(1). A variable or a set object is a set expression.

(2). If δ_1 and δ_2 are set expressions, then $\delta_1 \cup \delta_2$, $\delta_1 \cap \delta_2$, $\delta_1 \setminus \delta_2$ are set expressions.

(3). If δ is a set expression, then (δ) is a set expression. □

Definition 6.19 A *basic literal* is defined as follows:

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

(2). $\delta_1 = \delta_2$ is a positive basic literal, where δ_1, δ_2 are either variables or objects.

(3). If ψ is a positive basic literal, then $\neg\psi$ is a negative basic literal.

(4). If ψ_1, ψ_2 are two basic literals, then $\psi_1; \psi_2$ is a basic literal.

(5). $\delta_1 \leq \delta_2$, $\delta_1 \geq \delta_2$, $\delta_1 < \delta_2$, and $\delta_1 > \delta_2$ are basic literals, where δ_1, δ_2 are arithmetic expressions.

(6). $\delta_1 \subseteq \delta_2$, $\delta_1 \subset \delta_2$, $\delta_1 \supseteq \delta_2$, and $\delta_1 \supset \delta_2$, are basic literals, where δ_1, δ_2 are set expressions. □

Definition 6.20 A *rule* is an expression of the form $A \Leftarrow L_1, ..., L_n$ where the body $L_1, .., L_n, n \geq 1$ is a conjunction of basic literals, the head A is a positive basic literal and all variables in the head occur in the body. □

Rules are used in two different ways: generating representational objects using object constructors and obtaining object attribute values, as shown in Examples 5.5 and 5.6.

Definition 6.21 A *program* P is a triple $P = \langle S, OB, R \rangle$.

(1). S is a type system,

(2). OB is an object base,

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

A program is called *definite* if it does not have negations or sets in the body of a rule. Otherwise, it is called *normal.* □

Definition 6.22 The *typed terms* are defined as follows:

(1). If P is a type or a variable, then $P : type$ is a typed term.

(2). If P is a type or a variable, and $\{Q\}$ is a type grouping variable, and L is either isa or a variable then $P : type(L \rightarrow \{Q\})$ is a typed term.

(3). If P is a type or a variable, and $q_1, ..., q_n$ are types, and L is either isa or a variable, then $P : type(L \rightarrow \{q_1, ..., q_n\})$ is a typed term.

(4). If P is a type or a variable, and $q_1, ..., q_n$ are types, then $P : type(isa \rightarrow \{q_1, ..., q_n\}')$ is a typed term.

(5). If P is a representational type or a variable, Q is a type or a variable, and L is a label or a variable, then $P : type\langle L \rightarrow Q \rangle$ is a typed term.

(6). If $P : type\langle L_1 \rightarrow P_1 \rangle$, ... $P : type\langle L_n \rightarrow P_n \rangle$ are typed terms, then $P : type\langle L_1 \rightarrow P_1, ..., L_n \rightarrow P_n \rangle$ is a typed term.

(7). If P is a variable, X is either a variable, an object, or an object constructor, then $X : P$ is a typed term.

(8). If P is a representational type or a variable, L is a label or a variable, X is either a variable, a representational object or an object constructor, and Y is either a variable, an object, or an object constructor, then $X : P(L \rightarrow Y)$ is a typed term.

(9). If P is a representational type or a variable, L is a label or a variable X is either a variable, a representational object or an object constructor, and Y is either a variable, an object, or an object constructor, then $X : P(L \rightarrow Y')$ is a typed term.

(10). If $X : P(L_1 \rightarrow X_1)$, ... $X : P(L_n \rightarrow X_n)$ are typed terms, then $X : P(L_1 \rightarrow X_1, ..., L_n \rightarrow X_n)$ is a typed term. □

The typed terms are used only in queries to ask information about the type system.

Definition 6.23 A *typed literal* is defined as follows:

(1). A typed term is a positive typed literal.
(2). $\delta_1 = \delta_2$ is a positive typed literal, where δ_1, δ_2 are either variables or typed.
(3). If ψ is a positive typed literal, then $\neg\psi$ is a negative typed literal.
(4). If ψ_1, ψ_2 are two typed literals, then $\psi_1; \psi_2$ is a typed literal. □

Definition 6.24 A *query* is a conjunction of literals starting with ?–. □

Queries are used to ask information about objects, type, factual attributes of objects and types, definitional properties of types, which either is in the object base, the type system, or can be inferred from the rules of the program.

6.2 Mathematical Preliminaries

This section will introduce the mathematical concepts which will be used later in this thesis.

Definition 6.25 The *cartesian product* $S \times T$ of two sets S and T is the set of all pairs $\langle x, y \rangle$ where $x \in S$ and $y \in T$. □

Definition 6.26 A *binary relation* R from a set S to a set T is a subset of the cartesian product $S \times T$. That is, $R \subseteq S \times T$. □

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 6.27 A mapping from a set S to a set T, denoted by $f : S \to T$, is a relation from S to T such that every element of S is related to a unique element of T. That is, for all $x \in S$, and all $y_1, y_2 \in T$, xfy_1 and xfy_2 imply $y_1 = y_2$. For this reason, we note $f(x) = y$ rather than xfy and y is called the *image* of x under the mapping f. The set S is called the *domain* of the mapping, and the set T is called the *codomain.* □

Definition 6.28 A mapping $f : S \to T$ is said to be *one-to-one* if $f(x) = f(y)$ implies $x = y$, for all $x, y \in S$. □

Definition 6.29 A relation R on a set S is a relation from S to S. □

Definition 6.30 A relation R on a set S is

(1). *reflexive* iff xRx for all $x \in S$.
(2). *symmetric* iff xRy implies yRx, for all $x, y \in S$.
(3). *antisymmetric* iff xRy and yRx imply $x = y$, for all $x, y \in S$.
(4). *transitive* iff xRy and yRz imply xRz, for all $x, y, z \in S$. □

Definition 6.31 A relation on a set S is a *partial order* if it is reflexive, antisymmetric and transitive. □

Definition 6.32 A set S is a *partially ordered set*, denoted by $\langle S, \leq \rangle$, if the set S is endowed with a partial ordering relation $\leq$. □

Definition 6.33 A mapping f from a partially ordered set $\langle S, \leq_1 \rangle$ to a partially ordered set $\langle T, \leq_2 \rangle$ is said to be *monotonic* iff for any $x, y \in S$ $x \leq_1 y$ implies $f(x) \leq_2 f(y)$. □

Definition 6.34 Let $\langle S, \leq \rangle$ be a partially ordered set. 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 6.35 Let $\langle S, \leq \rangle$ be a partially ordered set. Then $a \in S$ is the *join* or *least upper bound* (abbreviated *lub*) of a 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 a 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 6.36 A partially ordered set $\langle L, \leq \rangle$ is a *meet-semilattice* iff for every subset X of L, there is a *glb*. □

Definition 6.37 A partial ordered set $\langle L, \leq \rangle$ is a *join-semilattice* iff for every subset X of L, there is a *lub*. □

Definition 6.38 A partial ordered set $\langle L, \leq \rangle$ is a *lattice* iff for each pair of elements $a, b \in L$ both $lub(\{a, b\})$ and $glb(\{a, b\})$ exist. □

Definition 6.39 A mapping f from a lattice $\langle L_1, \leq_1 \rangle$ to a lattice $\langle L_2, \leq_2 \rangle$ is a *lattice homomorphism* if it preserves the meet and join operations. That is, for every pair $x, y \in L_1$, $f(lub\{x, y\}) = lub(\{f(x), f(y)\})$, and $f(glb\{x, y\}) = glb(\{f(x), f(y)\})$. □

Thus, a lattice homomorphism is necessarily monotonic.

Definition 6.40 Let L be a meet-semilattice and $T : L \to L$ be a mapping. We say $a \in L$ is the *least fixpoint* of T if a is a fixpoint (that is, $T(a) = a$) and for all fixpoint $b \in L$, we have $a \leq b$. □

Theorem 6.1 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 $lfp(T) = glb\{x \mid T(x) \leq x\}$.

Proof: Put $G = \{x \mid T(x) \leq x\}$ 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. □

6.3 Semantics of NLO

Definition 6.41 An *interpretation* I of NLO is a tuple $\langle U, \Sigma, \Gamma, \pi, \sigma.g_{\mathcal{T}}, g_{\mathcal{L}}, g_{\mathcal{O}}, g_{\mathcal{F}} \rangle$,

(1). U is a countably infinite set called the *universe of objects*, which consists of three disjoint countably infinite sets **S**, **Z** and **O**. That is $U = \mathbf{Z} \cup \mathbf{S} \cup \mathbf{O}$.

(2). Σ is a non-empty set called the *universe of semantic types.*

(3). Γ is a countably infinite set called the *universe of semantic labels*, which consists of three disjoint sets: which contains $\{\gamma_{isa}\}$, Γ_f, Γ_s. every semantic label in Γ_f is called *single-valued* and every semantic label in $\Gamma_s \cup \{\gamma_{isa}\}$ is called *set-valued.*

(4). π maps each semantic type to its corresponding class. That is, $\pi : \Sigma \rightarrow 2^{U^+}$, where $2^{U^+} = 2^U \cup 2^{2^U} \cup 2^{2^{2^U}} ...$

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

(5.1). $\sigma(\gamma_{isa})$ is a mapping: $\Sigma \rightarrow 2^{\Sigma}$,

(5.2). for each $l \in \Gamma_f$, $\sigma(l)$ is a mapping $U \rightarrow U$,

(5.3). for each $l \in \Gamma_f$, $\sigma(l)$ is a mapping $U \rightarrow 2^{U^+}$.

(6). $g_{\mathcal{T}}$ is a lattice homomorphism which interprets each syntactic type in $\mathcal{T}$ as a semantic type in Σ, that is, $g_{\mathcal{T}} : \mathcal{T} \rightarrow \Sigma$.

(7). $g_{\mathcal{L}}$ is a one-to-one mapping which interprets each syntactic label in $\mathcal{L}$ as a semantic label in Γ, that is, $g_{\mathcal{L}} : \mathcal{L} \rightarrow \Gamma$, especially, $g_{\mathcal{L}} : isa \rightarrow \gamma_{isa}$.

(8). $g_{\mathcal{O}}$ is a one-to-one mapping $g_{\mathcal{O}} : \mathcal{S} \cup \mathcal{Z} \cup \mathcal{O} \rightarrow \mathbf{S} \cup \mathbf{Z} \cup \mathbf{O}$.

(9). $g_{\mathcal{F}}$ is a function which interprets the symbol *set* as a mapping $\Sigma \rightarrow \Sigma$ and interprets each k-ary object constructor as a mapping $U^k \rightarrow U$. □

An interpretation I gives a denotational semantics to the language. It maps every syntactic object to a semantic object by the mapping $g_{\mathcal{O}}$; every syntactic type to a semantic type by the mapping $g_{\mathcal{T}}$; every syntactic label to a semantic label by the mapping $g_{\mathcal{L}}$; every syntactic function to a semantic function by the mapping $g_{\mathcal{F}}$. It associates every semantic type with a class of objects by the mapping π. It interprets every semantic label as a function by the mapping σ.

It is assumed that $\leq, \geq, <, >, \subseteq, \supseteq, \subset \supset$, and $=$ have their standard meanings.

6.3.1 Satisfaction of Types

Definition 6.42 Given an interpretation $I = \langle U, \Sigma, \Gamma, \pi, \sigma.g_{\mathcal{T}}, g_{\mathcal{L}}, g_{\mathcal{O}}, g_{\mathcal{F}} \rangle$, the satisfaction of a type τ by I, denoted by $I \models \tau$, is defined by $g_{\mathcal{T}}$ and π as follows:

(1). For the built-in types:

(1.1). $I \models object$ iff $g_{\mathcal{T}}(object) \in \Sigma$ and $\pi(g_{\mathcal{T}}(object)) \subseteq \mathbf{O}$;

(1.2). $I \models integer$ iff $g_{\mathcal{T}}(integer) \in \Sigma$ and $\pi(g_{\mathcal{T}}(integer)) = \mathbf{Z}$;

(1.3). $I \models string$ iff $g_{\mathcal{T}}(string) \in \Sigma$ and $\pi(g_{\mathcal{T}}(string)) = \mathbf{S}$.

(2). For a set type $set(s)$, $I \models set(s)$ iff
$I \models s$, $g_{\mathcal{T}}(set(s)) = g_{\mathcal{F}}(set)(g_{\mathcal{T}}(s)) \in \Sigma$ and $\pi(g_{\mathcal{T}}(set(s))) \subseteq 2^{\pi(g_{\mathcal{T}}(s))} \subseteq 2^{U^+}$.

(3). For the basic types:

(3.1). if $\{a_1, ..., a_n\}$ is a basic type, then $I \models \{a_1, ..., a_n\}$ iff $g_{\mathcal{T}}(\{a_1, ..., a_n\}) \in \Sigma$ and either $\pi(g_{\mathcal{T}}(\{a_1, ..., a_n\})) = \{g_{\mathcal{O}}(a_1), ..., g_{\mathcal{O}}(a_n)\} \subset \mathbf{S}$
or $\pi(g_{\mathcal{T}}(\{a_1, ..., a_n\})) = \{g_{\mathcal{O}}(a_1), ..., g_{\mathcal{O}}(a_n)\} \subset \mathbf{Z}$;

(3.2). if $t = \{a_1, ..., a_n\}$ is a basic type, then $I \models t$ iff $g_T(t) \in \Sigma$ and either $\pi(g_T(t)) = \{g_O(a_1), ..., g_O(a_n)\} \subset \mathbf{S}$ or $\pi(g_T(t)) = \{g_O(a_1), ..., g_O(a_n)\} \subset \mathbf{Z}$;

(3.3). if $\{lb..rb\}$ is a basic type, then $I \models \{lb..rb\}$ iff $g_T(\{lb..rb\}) \in \Sigma$ and $\pi(g_T(\{lb..rb\})) = \{x \mid g_O(lb) \leq x \leq g_O(rb)\} \subset \mathbf{Z}$;

(3.4). if $t = \{lb..rb\}$ is a basic type, then $I \models t$ iff $g_T(t) \in \Sigma$ and $\pi(g_T(t)) = \{x : g_O(lb) \leq x \leq g_O(rb)\} \subset \mathbf{Z}$.

(4). For a representational type p:

(4.1). If p has a definitional property represented by $p : type\langle l \rightarrow q\rangle$, then $I \models p : type\langle l \rightarrow q\rangle$ iff $g_T(p) \in \Sigma, g_T(q) \in \Sigma, g_L(l) \in \Gamma/\{\gamma_{isa}\}$ and $\sigma(g_L(l))(\pi(g_T(p))) \subseteq \pi(g_T(q))$.

(4.2). If p has a factual property defined by $p : type(isa \rightarrow \{q_1, ..., q_m\})$, then $I \models p : type(isa \rightarrow \{q_1, ..., q_m\})$ iff $g_T(p) \in \pi(g_T(type)) \subseteq \Sigma$, $g_T(q_1) \in \pi(g_T(type)) \subseteq \Sigma$, ... $g_T(q_m) \in \pi(g_T(type)) \subseteq \Sigma$, $g_L(isa) = \gamma_{isa}$, and $g_L(isa)(g_T(p)) = \{g_T(q_1), ..., g_T(q_m)\}$, $\pi(g_T(p)) \subseteq \pi(g_T(q_i)) \subset \mathbf{O}$, for $1 \leq i \leq m$, and if $I \models q_i : type\langle l \rightarrow p_t\rangle$, then $I \models p : type\langle l \rightarrow p_t\rangle$.

(5). For the meta type *type* and built-in type *object*:

(5.1). $I \models type : type(isa \rightarrow \{type\})\langle isa \rightarrow set(type)\rangle$ iff $g_T(type) \in \Sigma, g_L(isa) = \gamma_{isa}$, $\sigma(g_L(isa))(\pi(g_T(type))) \subseteq 2^{\pi(g_T(type))}$, and $g_L(isa)(g_T(type)) = \{g_T(type)\}$;

(5.2). $I \models object : type(isa \rightarrow \{object\})$ iff $g_T(object) \in \pi(g_T(type)) \subseteq \Sigma$, $g_L(isa) = \gamma_{isa}$, and $g_L(isa)(g_T(object)) = \{g_T(object)\}$. □

By the definition, for every type p, all the definitional properties of its supertype are also its definitional properties. A type can have several definitional properties for the same attribute.

Definition 6.43 Two types p and q have a *subtype relation* based on some interpretation I denoted by $p \preceq q$, if $\pi(g_T(p)) \subseteq \pi(g_T(q))$. □

So two types have subtype relation if their classes have subset relation. Immediately, the following theorems hold:

Theorem 6.2 The subtype relation is a partial order on $\mathcal{T}$.

Proof: Direct from the definitions. □

Theorem 6.3 The subtype relationships between types are:

(1). If $\{a_1, ..., a_n\}$ is a basic type and $a_i \in \mathcal{Z}$ for $1 \leq i \leq n$, then $\{a_1, ..., a_n\} \preceq$ *integer*; if $\{a_1, ..., a_n\}$ is a basic type and $a_i \in \mathcal{S}$ for $1 \leq i \leq n$, then, $\{a_1, ..., a_n\} \preceq$ *string*.

(2). If $t = \{a_1, ..., a_n\}$ is a basic type and $a_i \in \mathcal{Z}$ for $1 \leq i \leq n$, then $t \preceq$ *integer*; if $t = \{a_1, ..., a_n\}$ is a basic type and $a_i \in \mathcal{S}$ for $1 \leq i \leq n$, then, $t \preceq$ *string*.

(3). If $\{lb..rb\}$ is a basic type, then $\{lb..rb\} \preceq$ *integer*.

(4). If $t = \{lb..rb\}$ is a basic type, then $t \preceq$ *integer*.

(5). If p is a representational type, then $p \preceq$ *object*.

(6). If $p \preceq q$, $p, q \in S$, then $set(p) \preceq set(q)$.

Proof: Direct from the definitions. □

Representational types have the following properties.

Theorem 6.4 Let p be a representation type with definitional properties: $\langle l \rightarrow p_1 \rangle$, ..., $\langle l \rightarrow p_n \rangle$. Then $\sigma(g_{\mathcal{L}}(l))(\pi(g_{\mathcal{T}}(p))) \subseteq \cap_{j=1}^{n} \pi(g_{\mathcal{T}}(p_j))$.

Proof: We have that $\sigma(g_{\mathcal{L}}(l))(\pi(g_{\mathcal{T}}(p))) \subseteq \pi(g_{\mathcal{T}}(p_i)), 1 \leq i \leq n$. So immediately, $\sigma(g_{\mathcal{L}}(l_s))(\pi(g_{\mathcal{T}}(p))) \subseteq \cap_{j=1}^{n} \pi(g_{\mathcal{T}}(p_j))$. □

Theorem 6.5 Let p be a representational type with $p : type(isa \rightarrow \{p_1, ..., p_m\})$ Then $p \preceq p_i, 1 \leq i \leq m$, i.e., p is a lower bound of $p_1, ..., p_m$ under the relation $\preceq$, and $\pi(g_{\mathcal{T}}(p)) \subseteq \cap_{j=1}^{m} \pi(g_{\mathcal{T}}(p_j))$.

Proof: Direct from the definitions. □

This theorem says that the class p is a subset of the intersection of the classes $p_1, ..., p_m$, or p is a lower bound of $p_1, ..., p_m$. For example, *workstudent* is a lower bound of *student* and *employee* in Example 5.3.

Theorem 6.6 Let $p_1 : type(isa \rightarrow \{.., p, ...\}), ..., p_m : type(isa \rightarrow \{..., p, ...\})$ are m representational types. Then $p_i \preceq p, 1 \leq i \leq m$, i.e., p is a upper bound of $p_1, ..., p_m$ under the relation $\preceq$, and $\cap_{j=1}^{m} \pi(g_{\mathcal{T}}(p_j)) \subseteq \pi(g_{\mathcal{T}}(p))$.

Proof: Direct from the definitions. □

This theorem says that the class p is a superset of the union of the classes $p_1, ..., p_m$, or p is an upper bound of $p_1, ..., p_m$. For example, *person* is an upper bound of *student* and *employee* in Example 5.3.

6.3.2 Satisfaction of Objects

Definition 6.44 Given an interpretation I, the satisfaction of an object o by I, denoted by $I \models o$ is defined by $g_{\mathcal{O}}$ as follows:

(1). For each object $o \in \mathcal{S}$, $I \models o$ iff $g_{\mathcal{O}}(o) = u \in \mathbf{S}$;
for each object $o \in \mathcal{Z}$, $I \models o$ iff $g_{\mathcal{O}}(o) = u \in \mathbf{Z}$;
for each representational object $o \in \mathcal{O}$, $I \models o$ iff $g_{\mathcal{O}}(o) = u \in \mathbf{O}$;

(2). For each set object $\{o_1, ..., o_m\}$, $I \models \{o_1, ..., o_m\}$ iff $I \models o_i, 1 \leq i \leq m$.

(3). For a representational object o,

(3.1). if o has a full factual property defined by $o : p(l \rightarrow o_t)$, then
$I \models o : p(l \rightarrow o_t)$ iff $g_{\mathcal{O}}(o) \in \pi(g_{\mathcal{T}}(p))$, and $\sigma(g_{\mathcal{L}}(l))(g_{\mathcal{O}}(o)) = g_{\mathcal{O}}(o_t)$;

(3.2). if o has a partial factual property defined by $o : p(l \rightarrow o'_t)$, then
$I \models o : p(l \rightarrow o'_t)$ iff $g_{\mathcal{O}}(o) \in \pi(g_{\mathcal{T}}(p))$, and $\sigma(g_{\mathcal{L}}(l))(g_{\mathcal{O}}(o)) \supseteq g_{\mathcal{O}}(o_t)$. □

By the definition, if an object has more that one different full factual properties, for example, if $mary : person(gender \rightarrow$ *"Female"*$)$ and $mary : person(gender \rightarrow$ *"Male"*$)$, then there is no interpretation which can satisfy both of them. However, an object can have several partial factual properties.

Definition 6.45 Let I be an interpretation. A representational object $o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow o_n)$ is *well-typed* under I iff there exist a representational type p with $p : type\langle l_1 \rightarrow p_1, ..., l_n \rightarrow p_n\rangle$ such that $g_{\mathcal{O}}(o_i) \in \pi(g_{\mathcal{T}}(p_i))$, $1 \leq i \leq n$. □

6.3.3 Satisfaction of Basic Terms and Basic Literals

Definition 6.46 A *variable assignment*, ν, is an assignment to each variable. It assigns an element in U to a variable and the variable is called an *individual object variable*, an element in 2^{U^+} to a variable and the variable is called a *set object variable*, a type in Σ to a variable and the variable is called a *type variable*, and a label in Γ to a variable and the variable is called 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 $\nu(o) = g_{\mathcal{O}}(o)$;
(2). if $l \in \mathcal{L}$, then $\nu(l) = g_{\mathcal{L}}(l)$;
(3). if $p \in \mathcal{T}$, then $\nu(p) = g_{\mathcal{T}}(p)$;
(4). if $f \in F$, then $\nu(f) = g_{\mathcal{F}}(f)$;
(5). $\nu(f(..., X, ...)) = g_{\mathcal{F}}(f)(..., \nu(X), ...)$;
(6). $\nu(\{o_1, ..., o_n\} = \{g_{\mathcal{O}}(o_1), ..., g_{\mathcal{O}}(o_n)\}$;
(7). if t is a literal or an arithmetic expression, then $\nu(t)$ results from t by applying ν to every object, label, type, variable, and object constructor of t. □

Definition 6.47 Given an interpretation I and a variable assignment ν, the satisfaction of a basic term ψ by I and ν, denoted by $I \models \nu(\psi)$, is defined as follows:

(1). For a basic term $X : p$, $I \models \nu(X : p)$ iff $\nu(X) \in \pi(g_T(p))$.

(2). For a basic term $\psi = X : p(l \rightarrow Y)$, $I \models \nu(\psi)$ iff
for all q such that $I \models p : type\langle l \rightarrow q \rangle$, $\nu(X) \in \pi(g_T(p)), \nu(Y) \in \pi(g_T(q))$, and $g_{\mathcal{L}}(l)(\nu(X) = \nu(Y)$.

(3). For a basic term $\psi = X : p(l \rightarrow Y'))$, $I \models \nu(\psi)$ iff
for all q such that $I \models p : type\langle l \rightarrow q \rangle$, $\nu(X) \in \pi(g_T(p)), \nu(Y) \in \pi(g_T(q))$, and $\sigma(g_{\mathcal{L}}(l))(\nu(X)) \supseteq \nu(Y)$.

(4). For a basic term $\psi = X : p(l_1 \rightarrow Y_1, ..., l_n \rightarrow Y_n'))$, $I \models \nu(\psi)$ iff
$I \models \nu(X : p(l_1 \rightarrow Y_1)), ..., I \models \nu(X : p(l_1 \rightarrow Y_1))$. □

Definition 6.48 Given an interpretation I and a variable assignment ν, the satisfaction of basic literals other than basic terms are defined as follows:

(1). $I \models \nu(\psi_1 = \psi_2)$ iff $\nu(\psi_1) = \nu(\psi_2)$.

(2). If ψ is a positive basic literal, $I \models \nu(\neg\psi)$ iff $I \not\models \nu(\psi)$.

(3). If ψ_1, ψ_2 are basic literals, $I \models \nu(\psi_1; \psi_2)$ iff $I \models \nu(\psi_1)$ or $I \models \nu(\psi_2)$.

(4). $I \models \nu(\delta_1 \theta \delta_2)$ iff $\nu(\delta_1), \nu(\delta_2) \in \mathbf{Z}$ and $\nu(\delta_1)\theta\nu(\delta_2)$, where $\theta \in \{\leq, \geq, <, >\}$.

(5). $I \models \nu(\delta_1 \theta \delta_2)$ iff $\nu(\delta_1)\theta\nu(\delta_2)$, where $\theta \in \{\subseteq, \supseteq, \subset, \supset\}$. □

Clearly, for a ground basic term ψ, i.e, a basic term without variables, its satisfaction is independent of a variable assignment, and it can be simply written as $I \models \psi$.

6.3.4 Satisfaction of Rules

Definition 6.49 Let I be an interpretation r be a rule of the form $A \Leftarrow L_1, ..., L_n$, $I \models r$ iff for each variable assignment ν, if $I \models \nu(A_i)$ for each $A_i, 1 \leq i \leq n$, then $I \models \nu(A)$, or for some variable assignment ν, not $I \models \nu(p_i)$ for some $p_i, 1 \leq i \leq n$.

□

Note here that there is a major difference between NLO and Horn-clause logic in their definitions of satisfaction of a rule. In Horn-clause logic, a rule is always satisfied by all interpretations, while in NLO, a rule may not be satisfied by any interpretation. There are two reasons for a rule in NLO not to be satisfied based on the above definition. One is that inferred factual attribute values do not confirm to the constraints of type definitions. The other is that an attribute of an object gets more than one full factual value.

6.3.5 Satisfaction of Programs

Definition 6.50 Let I be an interpretation and W be a set of type definitions, object definitions, terms or rules, $I \models W$ iff for each $W_i \in W$, $I \models W_i$. □

Definition 6.51 Let I be an interpretation and $P = \langle S, OB, R \rangle$ a program, $I \models P$ iff $I \models S \cup OB \cup R$. □

Definition 6.52 Let P be a program and I be an interpretation of P, I is called a *model* of P iff $\models_M P$. □

Theorem 6.7 Let $P = \langle S, OB, R \rangle$ be a program. If P has a model, then every representational object in OB is well-typed.

Proof: Direct from the definition. □

Definition 6.53 Let P be a program and F be an object. We say F is a *logical consequence* of P written as $P \models F$, if for every interpretation I of P, $I \models P$ implies that $I \models F$. □

Definition 6.54 Let P be a program, we say P is *unsatisfiable* if no interpretation of P is a model. □

Theorem 6.8 Let P be a program and F be an object. Then F is a logical consequence of P iff P is satisfiable and $P \cup \{\neg F\}$ is unsatisfiable.

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. □

6.3.6 Satisfaction of Typed Terms and Typed Literals

Definition 6.55 Given an interpretation I and a variable assignment ν, the satisfaction of a typed term ψ by I and ν, denoted by $I \models \nu(\psi)$, is defined as follows:

(1). For a typed term $\psi = P : type$, $I \models \nu(\psi)$ iff
$\nu(P) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma$.

(2). For a typed term $\psi = P : type(L \rightarrow \{Q\})$, $I \models \nu(\psi)$ iff
$\nu(L) = \gamma_{isa}$, $\nu(P) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma$, $\nu(Q) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma$, and
$\nu(Q) \in \sigma(\nu(L))(\nu(P))$.

(3). For a typed term $\psi = P : type(L \rightarrow \{q_1, ..., q_n\})$, $I \models \nu(\psi)$ iff
$\nu(L) = \gamma_{isa}$, $\nu(P) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma$, $g_{\mathcal{T}}(q_i) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma$ for $1 \leq i \leq n$,
and $\sigma(\nu(L))(\nu(P)) = \{g_{\mathcal{T}}(q_1), ..., g_{\mathcal{T}}(q_n)\}$.

(4). For a typed term $\psi = P : type(L \rightarrow \{q_1, ..., q_n\}')$, $I \models \nu(\psi)$ iff
$\nu(L) = \gamma_{isa}, \nu(P) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma, g_{\mathcal{T}}(q_i) \in \pi(g_{\mathcal{L}}(type)) \subseteq \Sigma$ for $1 \leq i \leq n$,
and $\sigma(\nu(L))(\nu(P)) \supseteq \{g_{\mathcal{T}}(q_1), ..., g_{\mathcal{T}}(q_n)\}$.

(5). For a typed term $\psi = P : type\langle L \rightarrow Q \rangle$, $I \models \nu(\psi)$ iff
$\nu(P) \in \pi(g_{\mathcal{T}}(type)) \subseteq \Sigma$, $\nu(Q) \in \pi(g_{\mathcal{T}}(type)) \subseteq \Sigma$, and
$\sigma(\nu(L))(\pi(\nu(P))) \subseteq \pi(\nu(Q))$.

(6). For a typed term $\psi = P : type\langle L_1 \rightarrow P_1, ..., L_n \rightarrow P_n\rangle$, $I \models \nu(\psi)$ iff
$I \models \nu(P : type\langle L_1 \rightarrow P_1\rangle), ..., I \models \nu(P : type\langle L_n \rightarrow P_1\rangle)$.

(7). For a typed term $\psi = X : P$, $I \models \nu(\psi)$ iff
$\nu(P) \in \pi(g_T(type)) \subseteq \Sigma$, and $\nu(X) \in \pi(\nu(P))$,

(8). For a typed term $\psi = X : P(L \rightarrow Y)$, $I \models \nu(\psi)$ iff
$\nu(P) \in \pi(g_T(type)) \subseteq \Sigma$, $\nu(X) \in \pi(\nu(P))$, and for all q such that $I \models \nu(P) : type\langle \nu(L) \rightarrow q\rangle$, $\nu(Y) \in \pi(\nu(q))$, $\sigma(\nu(L))(\nu(X)) = \nu(Y)$.

(9). For a typed term $\psi = X : P(L \rightarrow Y')$, $I \models \nu(\psi)$ iff
$\nu(P) \in \pi(g_T(type)) \subseteq \Sigma$, $\nu(X) \in \pi(\nu(P))$, and for all q such that $I \models \nu(P) : type\langle \nu(L) \rightarrow q\rangle$, $\nu(Y) \in \pi(\nu(q))$, $\sigma(\nu(L))(\nu(X)) \supseteq \nu(Y)$.

(10). For a typed term $\psi = X : P(L_1 \rightarrow Y_1, ..., L_n \rightarrow Y_n)$ $I \models \nu(\psi)$ iff
$I \models \nu(X : P(L_1 \rightarrow Y_1))$, ..., $I \models \nu(X : P(L_n \rightarrow Y_n))$ □

Definition 6.56 Given an interpretation I and a variable assignment ν, the satisfaction of typed literals other than typed terms are defined as follows:

(1). $I \models \nu(\delta_1 = \delta_2)$ iff $\nu(\delta_1) = \nu(\delta_2)$.

(2). If ψ is a typed literal, $I \models \nu(\neg\psi)$ iff $I \not\models \nu(\psi)$.

(3). If ψ_1, ψ_2 are typed literals, $I \models \nu(\psi_1; \psi_2)$ iff $I \models \nu(\psi_1)$ or $I \models \nu(\psi_2)$ □

6.3.7 Answers to Queries

Definition 6.57 Let P be a program, Q be a query and M be a model of P if it has, an *answer* to the query Q based on M is a variable assignment ν such that $M \models \nu(Q)$. □

So far, how a program can be satisfied and what answers to queries are have been discussed. However, what is its intended semantics is still not clear. In fact, the intended semantics of a program P is given by a special model M_P if it has which has the desired properties. This model will be discussed in the next chapter.

Chapter 7

Herbrand Interpretations

The semantics given in the last chapter is quite general. It tells how various symbols can be interpreted and how an interpretation can satisfy a program. A program may have many different models and therefore answers to a query may be different based on these models. However, it does not tell exactly what the intended semantics of a program is, and when given queries, what should be the right answers to them.

This chapter deals with these questions. It investigates a special kind of interpretation, as in traditional logic programming. This special kind of interpretation extends the Herbrand interpretation discussed in Chapter 2 and is also called a Herbrand interpretation in this chapter.

Definition 7.1 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). $U_0 = \mathcal{S} \cup \mathcal{Z} \cup \mathcal{O}$,
$U_i = U_{i-1} \cup \{f(o_1, ..., o_k) : f$ is a functor of arity k, and $o_j \in U_{i-1}, 1 \leq j \leq k\}$,
$U_* = \cup_{i=1}^{\infty} U_i$,
$U = U_* \cup 2^{U_*}$.

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

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

(4). $g_{\mathcal{T}}(p) = p$, for every $p \in \mathcal{T}$.

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

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

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

By the definition, the domains for objects, types and labels and functions of different Herbrand interpretations are the same, which are U, $\mathcal{T}$, $\mathcal{L}$ and $\mathcal{F}$. Also types, labels, objects and function symbols are interpreted as themselves in Herbrand interpretations. Only the classes and the mappings of labels may be interpreted differently. Since we are only interested in interpretations of a program, we can represent an interpretation by showing what the meta-class and normal classes are and how mappings of labels are defined and applied to objects.

Symbols such as *integer* and *string* have the normal fixed interpretation. For the simplicity of representation, example interpretations will not contain them.

Example 7.1 Consider the following definite program $P_1 = \langle S, OB, R \rangle$.

(a). Type System S

$p : type(isa \rightarrow \{object\})\langle f \rightarrow integer \rangle.$
$q : type(isa \rightarrow \{object\})\langle s \rightarrow set(integer) \rangle.$

(b). Object Base OB

$a_1 : p(f \rightarrow 1).$
$a_2 : p(f \rightarrow 2).$

(c). Rules R

$b : q(s \rightarrow \{X\}) \Leftarrow O : p(f \rightarrow X).$

A Herbrand interpretation for this program is

$$
\begin{aligned}
&I = I_{\mathcal{T}} \cup I_{\mathcal{O}} \\
&I_{\mathcal{T}} = \{\pi(type) = \{p, q, object, integer\}, \\
&\qquad \sigma(isa)(p) = \{object\}, \sigma(isa)(q) = \{object\}, \\
&\qquad \sigma(f)(\pi(p)) \subseteq \pi(integer), \sigma(s)(\pi(q)) \subseteq 2^{\pi(integer)}\}. \\
&I_{\mathcal{O}} = \{\pi(object) = \{a_1, a_2, b\}, \pi(p) = \{a_1, a_2\}, \pi(q) = \{b\}, \\
&\qquad \sigma(f)(a_1) = 1, \sigma(f)(a_2) = 2, \sigma(s)(b) = \{1, 2\}\}
\end{aligned}
$$

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

$$I_T = \{p : type(isa \rightarrow \{object\})\langle f \rightarrow integer\rangle,$$
$$q : type(isa \rightarrow \{object\})\langle f \rightarrow set(integer)\rangle\}.$$
$$I_O = \{a_1 : object, a_2 : object, b : object,$$
$$a_1 : p(f \rightarrow 1), a_2 : p(f \rightarrow 2), b : q(s \rightarrow \{1,2\})\}.$$ □

Later on, example interpretations will be represented by listing all classes and all mappings of labels in the representational object form.

Definition 7.2 Given a program P, a *Herbrand model* is a Herbrand interpretation which is a model for P. □

For the Example 7.1, the Herbrand interpretation I is obviously a Herbrand model for the given program.

Theorem 7.1 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. A Herbrand interpretation I' is defined as follows:

$$I' = I'_T \cup I'_O.$$
$$I'_T = \{s \mid s = p : type(isa \rightarrow \{q_1, ..., q_n\})\langle l \rightarrow q\rangle \text{ and } I \models s\}$$
$$I'_O = \{t \mid t = o : p(l_1 \rightarrow o_1, ..., l_n \rightarrow o'_n) \text{ and } I \models t\}$$

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

As the above theorem shows, Herbrand models as well as interpretations are the union of two sets, the set of type properties and the set of object properties. They do not contain negated information. Instead negated information can be inferred based on them. This is similar to traditional logic programming.

Theorem 7.2 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. □.

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

Given a program, it may have many interpretations and models, which means it can be interpreted differently.

Example 7.2 For the program in Example 7.1 again, following are some interpretations which are also models.

$$
\begin{aligned}
&I_1 = I_{\mathcal{T}_1} \cup I_{\mathcal{O}_1} \\
&I_{\mathcal{T}_1} = \{p : type(isa \rightarrow \{object\})\langle f \rightarrow integer\rangle, \\
&\qquad q : type(isa \rightarrow \{object\})\langle s \rightarrow set(integer)\rangle\} \\
&I_{\mathcal{O}_1} = \{a_1 : object, a_2 : object, a_3 : object, b : object, \\
&\qquad a_1 : p(f \rightarrow 1), a_2 : p(f \rightarrow 2), a_3 : p(f \rightarrow 3), b : q(s \rightarrow \{1,2,3\})\}. \\
&I_2 = I_{\mathcal{T}_2} \cup I_{\mathcal{O}_2} \\
&I_{\mathcal{T}_2} = \{p : type(isa \rightarrow \{object\})\langle f \rightarrow integer\rangle, \\
&\qquad q : type(isa \rightarrow \{object\})\langle f \rightarrow integer\rangle, s \rightarrow set(integer)\rangle\} \\
&I_{\mathcal{O}_2} = \{a_1 : object, a_2 : object, a_3 : object, b : object, \\
&\qquad a_1 : p(f \rightarrow 1), a_2 : p(f \rightarrow 2), a_3 : p(f \rightarrow 4), b : q(s \rightarrow \{1,2,4\})\}.
\end{aligned}
$$

In I_1, f maps a_3 to 3, and s maps b to $\{1,2,3\}$, while in I_2, f maps a_3 to 4, and s maps b to $\{1,2,4\}$. Besides, q has one more definitional property in I_2 than in I_1. □

Since there exist many interpretations and models, what is the exact semantics of the program? To answer this question, we first consider the simpler definite programs, then we consider normal programs.

7.1 Least Model Semantics for Definite Programs

This section first discusses the properties of definite programs.

Definition 7.3 Let P be a definite program, $I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$ and $I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$ be two interpretations of P. Then I_1 is a *sub-interpretation* of I_2, denoted by $I_1 \sqsubseteq I_2$ iff the following conditions hold:

(1). $\pi_1(type) \subseteq \pi_2(type)$.

(2). $\pi_1(p) \subseteq \pi_2(p)$, for every $p \in \pi_1(type)$.

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

(4). if $\sigma_1(l)$ is defined on $o \in U$, then $\sigma_2(l)$ is defined on $o \in U$ and $\sigma_1(l)(o) \subseteq \sigma_2(l)(o)$, for every set-valued label $l \in \Gamma_s$. □

Example 7.3 For the interpretations I, I_1, I_2 in examples 1 and 2, we have $I \sqsubseteq I_1$, $I \sqsubseteq I_2$ but neither $I_1 \sqsubseteq I_2$ nor $I_2 \sqsubseteq I_1$ because $\sigma_1(l)(a_3) \neq \sigma_2(l)(a_3)$ and neither $\sigma_1(s)(b) \subseteq \sigma_2(s)(b)$ nor $\sigma_1(s)(b) \supseteq \sigma_2(s)(b)$. □

Immediately, we have the following theorem.

Theorem 7.3 The sub-interpretation relation $\sqsubseteq$ over the set $\{I_i\}_{i \in N}$ of all possible interpretations of a given definite program is a partial order and $\{I_i\}_{i \in N}$ is a partially ordered set under $\sqsubseteq$.

Proof: Direct from the definition. □

Definition 7.4 Let P be a definite program and $I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$ and $I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$ be two interpretations of P. The *intersection* $I = \langle U, \Sigma, \Gamma, \pi, \sigma, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$, of I_1 and I_2, denoted by $I = I_1 \sqcap I_2$, is defined as follows:

(1). $\pi(type) = \pi_1(type) \cap \pi_2(type)$.

(2). $\pi(p) = \pi_1(p) \cap \pi_2(p)$, for every $p \in \pi(type)$.

(3). $\sigma(l)$ is defined on $o \in U$ if both $\sigma_1(l)$ and $\sigma_2(l)$ are 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 $l \in \Gamma_f$.

(4). $\sigma(l)$ 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 $l \in \Gamma_s$. □

Example 7.4 For the interpretations I, I_1, I_2 in examples 7.1 and 7.2, we have $I = I_1 \sqcap I_2$, based on the above definition. □

The intersection of interpretations of a definite program has the following properties.

Theorem 7.4 The relation $\sqcap$ over interpretations of a given program is commutative and idempotent i.e., $I_1 \sqcap I_2 = I_2 \sqcap I_1$ and $I_1 \sqcap I_1 = I_1$ for any two interpretations I_1 and I_2.

Proof: Direct from the definition. □

Theorem 7.5 The relation $\sqcap$ over interpretations of a given program is associative, i.e., $I_1 \sqcap (I_2 \sqcap I_3) = (I_1 \sqcap I_2) \sqcap I_3$ for any three interpretations I_1, I_2 and I_3.

Proof: Let

$$I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle,$$
$$I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle,$$
$$I_3 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle,$$
$$I_{23} = I_2 \sqcap I_3 = \langle U, \Sigma, \Gamma, \pi_{23}, \sigma_{23}, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle,$$
$$I_{12} = I_1 \sqcap I_2 = \langle U, \Sigma, \Gamma, \pi_{12}, \sigma_{12}, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle,$$
$$I_{123} = I_1 \sqcap I_{23} = \langle U, \Sigma, \Gamma, \pi_{123}, \sigma_{123}, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle,$$

$I'_{123} = I_{12} \sqcap I_3 = \langle U, \Sigma, \Gamma, \pi'_{123}, \sigma'_{123}, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$.

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

For I_{23} we have

$\pi_{23}(p) = \pi_2(p) \cap \pi_3(p)$,
$\sigma_{23}(l)(o) = \sigma_2(l)(o) = \sigma_3(l)(o)$, if $\sigma_2(l)(o) = \sigma_3(l)(o)$ for every $l \in \Gamma_0$,
$\sigma_{23}(l)(o) = \sigma_2(l)(o) \cap \sigma_3(l)(o)$, if $\sigma_2(l)$ and $\sigma_3(l)$ are defined on o for every $l \in \Gamma_i$.

So for I_{123} we have

$\pi_{123}(p) = \pi_1(p) \cap \pi_{23}(p) = \pi_1(p) \cap \pi_2(p) \cap \pi_3(p)$,
$\sigma_{123}(l)(o) = \sigma_1(l)(o) = \sigma_{23}(l)(o)$, if $\sigma_1(l)(o) = \sigma_{23}(l)(o)$ for every $l \in \Gamma_0$, 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 $l \in \Gamma_0$,
$\sigma_{123}(l)(o) = \sigma_1(l)(o) \cap \sigma_{23}(l)(o) = \sigma_1(l)(o) \cap \sigma_2(l)(o) \cap \sigma_3(l)(o)$ if
$\sigma_1(l)$, $\sigma_2(l)$ and $\sigma_3(l)$ are defined on o for every $l \in \Gamma_i$, $i \geq 1$.

Similarly for I'_{123} we have

$\pi'_{123}(p) = \pi_1(p) \cap \pi_2(p) \cap \pi_3(p)$,
$\sigma'_{123}(l)(o) = \sigma_1(l)(o)$ if $\sigma_1(l)(o) = \sigma_2(l)(o) = \sigma_3(l)(o)$ for every $l \in \Gamma_0$,
$\sigma'_{123}(l)(o) = \sigma_1(l)(o) \cap \sigma_2(l)(o) \cap \sigma_3(l)(o)$ if
$\sigma_1(l)$, $\sigma_2(l)$ and $\sigma_3(l)$ are defined on o for every $l \in \Gamma_i$, $i \geq 1$.

Therefore we have $\pi_{123} = \pi'_{123}$, $\sigma_{123} = \sigma'_{123}$. □

Theorem 7.6 (Model Intersection Property)

Let P be a definite program and $\{M_i\}_{i \in N}$ be a non-empty set of models for P. Then the intersection $\sqcap_{i \in N} M_i$ is also a model for P.

Proof: Let $M = \sqcap_{i \in N} M_i$. Clearly, M is an interpretation for P. If M is not a model for P, then either some representational types, representational 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 representational object t in P which is

$$o : p(l_1 \rightarrow o_1, l_2 \rightarrow \{o_{s_1}, ..., o_{s_n}\}, l_3 \rightarrow \{o_{t_1}, ..., o_{t_m}\}')$$

cannot be satisfied by M. Since $M_i, i \in N$ are models of P, then $M_i \models t$. That is, $o \in \pi_i(p)$, $\sigma_i(l_1)(o) = o_1$, $\sigma_i(l_2)(o) = \{o_{s_1}, ..., o_{s_n}\}$, $\sigma_i(l_3)(o) \supseteq \{o_{t_1}, ..., o_{t_n}\}$, for all $i \in N$. We have $o \in \cap_{i \in N}\pi_i(p)$, $\cap_{i \in N}\sigma_i(l_1)(o) = o_1$, $\cap_{i \in N}\sigma_i(l_2)(o) = \{o_{s_1}, ..., o_{s_n}\}$. $\cap_{i \in N}\sigma_i(l_3)(o) \supseteq \{o_{t_1}, ..., o_{t_n}\}$. Therefore $M \models t$, which is a contradiction.

Suppose a rule r in P which is

$$A \Leftarrow A_1, ..., A_n$$

cannot be satisfied by M. Now let us consider every ground substitution $\theta_j, j \in N$. If not every M_i is such that $M_i \models A_1\theta_j$, ..., $M_i \models A_n\theta_j$, then we do not have $M \models A_1\theta_j$, ..., $M \models A_n\theta_j$. Therefore we have $M \models r\theta_j$. That is, M cannot satisfy the body therefore satisfy the rule, which is a contradiction. Suppose for all M_i we have $M_i \models A_1\theta_j$, ..., $M_i \models A_n\theta_j$. Then $M \models A_1\theta_j$, ..., $M \models A_n\theta_j$. Since every M_i is a model of P, we have $M_i \models A\theta_j$. So $M \models A\theta_j$. Therefore $M \models r\theta_j$, which is still a contradiction. For all ground substitution $\theta_j, j \in N$, we have $M \models r\theta_j$, that is, $M \models r$, which is a contradiction. □.

The above theorem says that the NLO programs preserve the model intersection property of traditional logic programming, based on Definition 7.4 for the model intersection.

Definition 7.5 A model M of P is *minimal* iff for each model N of P, if $N \sqsubseteq M$ then $N = M$. □

Definition 7.6 A model M of P is *least* iff for each model N of P, $M \sqsubseteq N$. □

Theorem 7.7 If a definite program P has a model, then it has a unique least model which is the intersection of all possible models for P denoted by M_P.

Proof: Direct from the definition. □

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

Example 7.5 For the interpretations I, I_1, I_2 in the examples 7.1 and 7.2, the least model is I. 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 I_1 and I_2 contain $o_3 : p(f \rightarrow 3)$ and $o_3 : p(f \rightarrow 4)$. □

Theorem 7.8 The set $\{M_i\}_{i \in N}$ of all possible models of a given definite program is a partially ordered set under $\sqsubseteq$.

Proof: Direct from the definition. □

Theorem 7.9 All possible models $\langle \{M_i\}_{i \in N}, \sqsubseteq \rangle$ of a definite program form a meet-semilattice.

Proof: Direct from the definition. □

Theorem 7.10 Let P be a definite program. If P has a model, then $M_P = \{F \mid F$ 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 6.8.

iff $P \cup \{\neg F\}$ has no Herbrand models, by theorem 7.2.

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

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

iff $F \in M_P$. □

Definition 7.7 Given a definite program P, its declarative semantics is given by its unique least model M_P if it has. □

Unlike traditional logic programming in which every definite program has a unique least model. A definite NLO program may not have a model, let alone a unique least model.

Example 7.6 Consider the following definite program $P_2 = \langle S, OB, R \rangle$.

(a). Type System S

$person : type(isa \rightarrow \{object\})\langle gender \rightarrow \{\text{"Male"}, \text{"Female"}\}\rangle$.

(b). Object Base OB

$mary : person(gender \rightarrow \text{"Female"})$.
$john : person(gender \rightarrow \text{"Male"})$.

(c). Rules R

$mary : person(gender \rightarrow X) \Leftarrow john : person(gender \rightarrow X)$.

Following are two interpretations for the program.

$I_1 = \{person : type(isa \rightarrow \{object\})\langle gender \rightarrow \{\text{"Male"}, \text{"Female"}\}\rangle,$
$mary : person(gender \rightarrow \text{"Female"}),$
$john : person(gender \rightarrow \text{"Male"})\}.$

$I_2 = \{person : type(isa \rightarrow \{object\})\langle gender \rightarrow \{\text{"Male"}, \text{"Female"}\}\rangle,$
$mary : person(gender \rightarrow \text{"Male"}),$
$john : person(gender \rightarrow \text{"Male"})\}.$

But neither of them are models of the program. In fact, this program has no models because the single-valued attribute *gender* of *mary* has two different values. □

Example 7.7 Consider the following definite program P_3.

$$student : type\langle age \rightarrow \{15..35\}\rangle.$$
$$alan : student(age \rightarrow 36).$$

This program has no model because the object *alan* is not well-typed. □

Definition 7.8 Given a definite program P and a query ?- $L_1, ..., L_n$, a correct answer to it is a ground substitution θ such that $M_P \models L_1\theta$, ..., $M_P \models L_n\theta$. □

Example 7.8 For the definite program P_1 and its interpretation I in Example 7.1, we know that I is the least model based on the previous discussions. Therefore the semantics of P_1 is given by I. Given a query ?- $o : P(L \rightarrow Y)$, the only answer to it is $\theta = \{P/q, L/s, Y/\{1,2\}\}$. Given another query ?- $P : type\langle L \rightarrow Q\rangle$, there are two answers to it: $\theta_1 = \{P/p, L/f, Q/integer\}$, $\theta_2 = \{P/q, L/s, Q/set(integer)\}$. □

7.2 Bottom-up Computation for Definite Programs

Based on the above discussions, if a definite program has a model, then it has a least model. This least model is the intersection of all models and contains all logic consequence of the program. This section shows that this least model can be obtained by the operator defined as follows, which corresponds to the bottom-up computation of proof theory shown in Chapter 2.

Definition 7.9 Given a normal program $P = \langle S, OB, R\rangle$ and an interpretation I, then an operator T_P over I is defined as follows.

$$T_P(I) = \{A\theta \mid A \Leftarrow L_1, ..., L_n \in R \text{ and there exists a ground substitution } \theta \text{ such that } I \models L_1\theta, ..., I \models L_n\theta\}.$$ □

Note that a definite program is also a normal program.

Theorem 7.11 Given a definite program P, T_P is monotonic, i.e, if $I_1 \sqsubseteq I_2$, then $T_P(I_1) \sqsubseteq T_P(I_2)$.

Proof: For every rule $A \Leftarrow A_1, ..., A_n \in R$, because of $I_1 \sqsubseteq I_2$, if $I_1 \models A_1\theta$, ..., $I_1 \models A_n\theta$, then $I_2 \models A_1\theta$, ..., $I_2 \models A_n\theta$. So $T_P(I_1) \sqsubseteq T_P(I_2)$. □

Theorem 7.12 Let P be a normal program and I be an interpretation of P. Then I is a model for P iff $T_P(I) \sqsubseteq I$.

Proof: I is a model for P iff for each rule $A \Leftarrow A_1, ..., A_n$ in P, we have $I \models A_1\theta$,..., $I \models A_n\theta$ implies $I \models A\theta$ iff $T_P(I) \sqsubseteq I$. □

Definition 7.10 Given a definite program $P = \langle S, OB, R \rangle$, the powers of the operator T_P is defined as follows:

$$T_P \uparrow 0 = S \cup OB$$
$$T_P \uparrow n = T_P(T_P \uparrow (n-1)) \cup S \cup OB$$
$$T_P \uparrow \omega = lub\{T_P \uparrow n \mid \omega \text{ denotes the first ordinal number and } n \in \omega\}$$ □

Example 7.9 For the definite program P_1 again. We have:

$$T_P \uparrow 0 = S \cup OB$$
$$T_P \uparrow 1 = T_P(T_P \uparrow 0) \cup T_P \uparrow 0$$
$$= S \cup OB \cup \{o : q(s \rightarrow \{1,2\})\}$$
$$T_P \uparrow \omega = ... = T_P \uparrow 3 = T_P \uparrow 2 = T_P \uparrow 1 = I$$ □

Theorem 7.13 The powers of the operator T_P have 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 $T_P \uparrow \alpha = lfp(T_P)$.

Proof: Direct from the definition. □

This theorem says that T_P is monotonic.

Theorem 7.14 Let $X = \{T_P \uparrow n \mid n \in \omega\}$. Then X is directed, i.e., every finite subset of X has an upper bound in X, and $\nu(\{B_1, ..., B_n\}) \subseteq lub(X)$ iff $\nu(\{B_1, ..., B_n\}) \subseteq I$, for some $I \in X$.

Proof: The first part of the theorem is straightforward. For the second part, it is trivial that $\nu(\{B_1, ..., B_n\}) \subseteq I$ implies $\nu(\{B_1, ..., B_n\} \subseteq lub(X)$.

Assume that $\nu(\{B_1, ..., B_n\}) \subseteq lub(X)$. Then for each $i, 1 \leq i \leq n$, we have $\nu(B_i) \in lub(X)$. If not $\nu(B_i) \in I$ for all $I \in X$, then not $\nu(B_i) \in lub(X)$, which is a contradiction to assumption. Therefore, for each $\nu(B_i)$, there is some $I_i \in X$ where $\nu(B_i) \in I_i$. 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 $\nu(\{B_1, ..., B_n\}) \subseteq I$. □

The above theorem is just used by the following theorem.

Theorem 7.15 Let $P = \langle S, OB, R \rangle$ and $X = \{T_P \uparrow n \mid n \in \omega\}$. 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

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

iff $A \Leftarrow A_1, ..., A_n \in R$ and $\nu(\{A_1, ..., A_n\}) \subseteq lub(X)$

iff $A \Leftarrow A_1, ..., A_n \in R$ and $\nu(\{A_1, ..., A_n\}) \subseteq I$, for some $I \in X$

by theorem 7.14

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

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

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

For the second part of the theorem, we have that

$T_P(T_P \uparrow \omega) = T_P(lub(X)) = lub(T_P(X))$

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

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

Now we come to the major result of the theory. This theorem provides a fixpoint characterization of the least model of a definite program, as in traditional logic programming.

Theorem 7.16 (Fixpoint Characterization of the Least Model)
Let $P = \langle S, OB, R\rangle$ be a program. If it has a model, then $T_P \uparrow \omega = M_P$. In other words, $T_P \uparrow \omega$ is a model of P and every model of P contains $T_P \uparrow \omega$.

Proof 1: $M_P = glb\{I \mid I$ is a model for $P\}$, by theorem 7.7

$= glb\{I \mid T_P(I) \sqsubseteq I\}$, by theorem 7.12

$= lfp(T_P)$, by theorem 6.1

$= T_P \uparrow \omega$, by theorem 7.15. □

Proof 2: (1). $T_P \uparrow \omega$ is model of P.

Assume $T_P \uparrow \omega$ is not a model. Then there is a rule in R of the form

$A \Leftarrow A_1, ..., A_n$.

and a ground substitution θ such that $T_P \uparrow \omega \models A_1\theta$, ... $T_P \uparrow \omega \models A_n\theta$, but not

$T_P \uparrow \omega \models A\theta$. For each i, there exists a j such that $T_P \uparrow j \models A_i\theta$. Let $\alpha(i)$ denote this j and m be $\max\{\alpha(i) \mid 1 \leq i \leq n\}$. Then for some $m' > m$, $T_P \uparrow m' \models A\theta$. By monotonicity of T_P, $T_P \uparrow \omega \models A\theta$, which is a contradiction.

(2). Every model of P contains $T_P \uparrow \omega$.

Let N be a model. We prove by induction that $T_P \uparrow i \sqsubseteq N$. The basis is obvious. Assume the claim holds for $T_P \uparrow i$, and consider $T_P \uparrow_{i+1}$. Since N is a model of P, $T_P(N) \sqsubseteq N$ by Theorem 7.12. Now that $T_P \uparrow i \sqsubseteq N$, we have $T_P \uparrow (i+1) = T_P(T_P \uparrow i) \cup S \cup OB \sqsubseteq T_P(N) \sqsubseteq N$. Hence also $T_P \uparrow (i+1) \sqsubseteq N$. □

Note that if a program has no model, then $T_P \uparrow \omega$ may not even be an interpretation of the program.

Example 7.10 Consider the definite program P_2 in Example 7.6 again. By the definition, we have

$$\begin{aligned} T_P \uparrow 0 &= S \cup OB \\ T_P \uparrow 1 &= T_P(T_P \uparrow 0) \cup T_P \uparrow 0 \\ &= S \cup OB \cup \{mary : person(gender \rightarrow \text{“}Male\text{”})\} \\ T_P \uparrow \omega &= ... = T_P \uparrow 3 = T_P \uparrow 2 = T_P \uparrow 1 = I_1 \cup I_2 \\ &\supset \{mary : person(gender \rightarrow \text{“}Male\text{”}), \\ &\quad\ mary : person(gender \rightarrow \text{“}Female\text{”})\} \end{aligned}$$

which is not an interpretation of the program. □

The reason for the problem is that the attribute *gender* of *mary* has two different values. The theorem 7.16 only says that if a program has a model, then it has a least model M_P which is equal to $T_P \uparrow \omega$.

7.3 Perfect Model Semantics for Normal Programs

As discussed above, for a definite program P, if it has a model, then it has following properties:

- T_P is monotonic,
- the intersection of two models of P is still a model of P, and
- P has a least model M_P.

The declarative semantics of P is given by M_P which equals $T_P \uparrow \omega$.

However, as in traditional logic programming, normal programs do not enjoy these properties . Let P be a normal program, that is, a program which has negation in the body of a rule or sets in both body and head of a rule. We have

- T_P need not be monotonic,
- the intersection of two models of P need not be a model of P, and
- P may have no least model.

Example 7.11 Consider the following normal program P_4

$$mary : person.$$
$$X : female \Leftarrow X : person, \neg X : male.$$

This program has two models $M_1 = \{mary : person, mary : female\}$ and $M_2 = \{mary : person, mary : male\}$. Their intersection is $\{mary : person\}$ which is not a model. Both M_1 and M_2 are minimal models, but there is no least model. We have $\{mary : person\} \sqsubseteq M_2$, $T_P(\{mary : person\}) = M_1$, and $T_P(M_2) = \{\}$. Since $M_1 \not\sqsubseteq \{\}$, T_P is not monotonic. □

Example 7.12 Consider another normal program P_5 whose type system is omitted.

$a_1 : p(f \rightarrow 1)$.
$b_1 : q(s_1 \rightarrow \{Y\}) \Leftarrow O : p(f \rightarrow Y)$.
$c_1 : r(s_2 \rightarrow \{Y\}) \Leftarrow O : q(s_1 \rightarrow Y)$.

Following are three models of the program.

$M_1 = \{a_1 : p(f \rightarrow 1), b_1 : q(s_1 \rightarrow \{1\}), c_1 : r(s_2 \rightarrow \{\{1\}\})\}$.
$M_2 = \{a_1 : p(f \rightarrow 1), a_2 : p(f \rightarrow 2), b_1 : q(s_1 \rightarrow \{1,2\}), c_1 : r(s_2 \rightarrow \{\{1,2\}\})\}$.
$M_3 = \{a_1 : p(f \rightarrow 1), a_2 : p(f \rightarrow 3), b_1 : q(s_1 \rightarrow \{1,3\}), c_1 : r(s_2 \rightarrow \{\{1,3\}\})\}$.

Their intersection is $\{a_1 : p(f \rightarrow 1), b_1 : q(s_1 \rightarrow \{1\}), c_1 : r\}$ which is not a model. Here M_1, M_2 and M_3 are all minimal models, but there is no least model. Let $I_1 = \{b_1 : q(s_1 \rightarrow \{1\})\}$ and $I_2 = \{b_1 : q(s_1 \rightarrow \{1,2\})\}$. We have $I_1 \sqsubseteq I_2$ $T_P(I_1) = \{c_1 : r(s_1 \rightarrow \{\{1\}\})\}$, and $T_P(I_2) = \{c_1 : r(s_1 \rightarrow \{\{1,2\}\})\}$. That is, $T_P(I_1) \not\sqsubseteq T_P(I_2)$. So T_P is nonmonotonic. Note that this program can have many minimal models. □

7.3.1 Stratified Programs

Similar to traditional logic programming, we restrict every normal program to be stratified so that a distinguished minimal model, if it has, can be selected in a very natural and intuitive way as the intended semantics of the program.

Definition 7.11 Let $P = \langle S, OB, R\rangle$ be a program. The *depends-on* relationship, denoted by $<$, between types and between attributes in R, and an auxiliary relation $\leq$ are defined as follows.

D1. Let p and q be two types. $p < q$, if there is a rule in R with p in the head and q in the body, and the term in which q occurs is negated, or either the term in which p occurs or the term in which q occurs involves sets. If $p < q$, then for every attribute att_p of p and every attribute att_q of q, we have $att_p < att_q$.

D2. Let p and q be two types. $p \leq q$, if there is a rule in R with p in the head and q in the body, and the terms in which p and q occur have no negations and do

not involve sets. If $p \leq q$, then for every attribute att_p of p and every attribute att_q of q, we have $att_p \leq att_q$.

D3. Let att_1 and att_2 be two attributes. $att_1 < att_2$, if they are the attributes of the same type and there is a rule in R with att_1 in the head and att_2 in the body, and the term in which att_2 occurs is negated, or either the term in which att_1 occurs or the term in which att_2 occurs involve sets.

D4. Let att_1 and att_2 be two attributes. $att_1 \leq att_2$, if they are the attributes of the same type and there is a rule in R with att_1 in the head and att_2 in the body, and the terms in which att_1 and att_2 occur have no negations and do not involve sets.

D5. If $A \leq B$ and $B \leq C$, then $A \leq C$.

D6. If $A < B$ and $B < C$, then $A < C$. □

Example 7.13 For the programs P_4 in Example 7.11 and P_5 in Example 7.12, we have $female < male$ by D1, $female \leq person$ by D2, $r < q$, $q < p$ by D1, and $r < q$ by D6. □

Example 7.14 Consider the following normal program P_6 which is an NLO version of the program in Example 2.4, the type system is obvious and omitted both in the program and in models.

$$
\begin{aligned}
&peter : person(in \rightarrow water).\\
&phil : person(can \rightarrow swim).\\
&X : person(could \rightarrow sink) \Leftarrow X : person(in \rightarrow water),\\
&\qquad\qquad\qquad\qquad \neg X : person(can \rightarrow swim).\\
&X : person(is \rightarrow happy) \Leftarrow X : person(in \rightarrow water),\\
&\qquad\qquad\qquad\qquad \neg X : person(could \rightarrow sink).
\end{aligned}
$$

We have $could \leq in$ and $is \leq in$ by D4, $could < can$ and $is < could$ by D3, $is < can$ by D5, etc.. □

Definition 7.12 Let $P = \langle S, OB, R \rangle$ be a normal program. A *definition* of a class p in P is the subset of P consisting of all rules in R such that p occurs in the head

of them, and all objects in OB in which p occurs. A *definition* of values of a factual attribute *att* in P, we mean the subset of P consisting of all rules in R such that *att* occurs in the head of them, and all objects in OB in which *att* occurs. □

Definition 7.13 A program $P = \langle S, OB, R \rangle$ is *stratified* if there is a partition $P = P_0 \cup P_1 ... P_n$ where $P_0 = OB \cup S$, such that the following conditions hold for $i = 0, ..., n$.

- if $A < B$, then the definition of B is contained within $\cup_{j<i} P_j$.
- if $A \leq B$, then the definition of B is contained within $\cup_{j \leq i} P_j$.

Each P_i is called a *stratum* of P. □

Definition 7.14 Let P be a normal program. The *dependency graph* G of P is the directed graph. The nodes of G consist of all types and attributes of P. There is an edge (p, q) in G iff $p < q$, or $p \leq q$. Furthermore, the edge (p, q) is marked with a $<$ or $\leq$ depending on $p < q$ or $p \leq q$. □

A dependency graph of P represents the relation *depend-on* between types and attributes of P.

Definition 7.15 A program P is stratified iff in its dependency graph there is no cycles containing a $<$ edge.

Proof: Essentially the same as the proof in [ABW88] □

Since a program has only finite types and attributes, The above theorem suggests a simple test whether a program is stratified. It is trivial to check that the programs P_4 in Example 7.11, P_5 in Example 7.12, and P_6 in Example 7.14 are stratified.

Example 7.15 Following is a program which is not stratified.

$$X : boredPerson \Leftarrow X : person(hobbies \rightarrow \{\}).$$
$$X : person(hobbies \rightarrow \{tv\}) \Leftarrow X : boredPerson.$$

We have both $boredPerson < person$ and $person < boredPerson$. That is, there is a cycle containing $<$ edges in its dependency graph. Therefore, this program is not stratified. □

Definition 7.16 Let P be a normal program and $I_1 = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$ and $I_2 = \langle U, \Sigma, \Gamma, \pi_2, \sigma_2, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$ be two interpretations of P. Then the *difference* of I_1 and I_2 is an interpretation $I = \langle U, \Sigma, \Gamma, \pi_1, \sigma_1, g_\mathcal{T}, g_\mathcal{L}, g_\mathcal{O}, g_\mathcal{F} \rangle$, denoted by $I = I_1 - I_2$ is defined as follows:

d1. $\pi(type) = \pi_1(type)$.

d2. $o \in \pi(p)$, if $o \in \pi_1(p)$ and $o \notin \pi_2(p)$ for every $p \in \pi(type)$;
$o \in \pi(p)$, if $o \in \pi_1(p)$ and $o \in \pi_2(p)$ and there is a label l such that $\sigma(l)$ is defined on o, for every $p \in \pi(type)$;

d3. if $\sigma_1(l)$ is defined on $o \in U$ and $\sigma_2(l)$ is not defined on $o \in U$, then then $\sigma(l)$ is defined on o and $\sigma(l)(o) = \sigma_1(l)(o)$;
if both $\sigma_1(l)$ and $\sigma_2(l)$ are defined on $o \in U$, and $\sigma_1(l)(o) \neq \sigma_2(l)(o)$, then $\sigma(l)$ is defined on o and $\sigma(l)(o) = \sigma(l)(o) - \sigma_2(l)(o)$;
if both $\sigma_1(l)$ and $\sigma_2(l)$ are defined on $o \in U$, and $\sigma_1(l)(o) = \sigma_2(l)(o)$, then $\sigma(l)$ is not defined on o. □

7.3.2 Perfect Model

Now that we have defined the depend-on relation and model difference, we are prepared to define the notion of a perfect model. It is our goal to *minimize* definitions of classes and factual attribute values which are depended on by others as much as possible, even at the expense of enlarging the set of definitions of classes and factual attribute values which depend on the minimized ones.

Definition 7.17 Let P be a normal program and suppose that M and N are two distinct models of P. Then N is said to be *preferable* to M, denoted by $N \ll M$, if for every object in $N - M$, in which A occurs, there is an object in $M - N$, in which B occurs, such that $A < B$. A model M of P is *perfect* if there are no models preferable to M. □

Theorem 7.17 Preferabilty is a transitive relation on models.

Proof: Assume $N \ll M \ll Q$. We show that $N \ll Q$. Since by Definition 7.17, $\ll$ is irreflexive, we first show $N \neq Q$, that is, they are distinct. Assume that $N = Q$. Then we have $Q \ll M \ll Q$. Since $M \ll Q$, $Q \sqsubseteq M$ is impossible. Similarly, $Q \ll M$, $M \sqsubseteq Q$ is impossible. Hence both $Q - M$ and $M - Q$ are nonempty. From $M \ll Q$, we have that for every A in $M - Q$, there exists a B in $Q - M$ such that $A < B$. Now from $Q \ll M$, we have a C in $M - Q$, such that $B < C$, and so on. Since $<$ is transitive and irreflexive on the finite set of types and attributes which appear in the program, it cannot have infinite decreasing chains, which is a contradiction. Thus, $N \neq Q$.

To show that $N \ll Q$, consider a A in $N - Q$. If A is also in M, then it is in $M - Q$, so we can deduce the existence of a B in $Q - M$ such that $A < B$. If B is not in N, we are done. If B is in N, then it is in $N - M$, which allows us to infer the existence of yet another C in $M - N$ such that $B < C$. If C is in Q, then it is in $Q - N$ and we are done. Otherwise, it is in $M - Q$, so we have the same situation as we previously had for A. Since there exist no infinite decreasing chains of $<$, a member in $Q - N$ must eventually be found. The case where A is not in M is handled similarly. □

Theorem 7.18 If $N \sqsubseteq M$, then $N \ll M$. In particular, the perfect model is minimal.

Proof: The first assertion is obvious and the second immediately follows from the first. □

The converse of the above theorem is, in general, not true. A normal program may have many minimal models as the program P_4 and P_5 show. This theorem says that a perfect model is just one of them. An advantage of the concepts of preferability and perfect model is that they are based on the depend-on relations between types and attributes, and are independent of the specific stratification chosen for a program.

Theorem 7.19 Let P be a definite program and supposed P has a model. $N \ll M$ if and only if $N \sqsubseteq M$. Consequently, a model of P is perfect if and only if it is the least model of P.

Proof: Since P is a definite program, the depend-on relation $<$ is empty. Consequently, $N \ll M$ iff $N \sqsubseteq M$. □

Theorem 7.20 In order to show that a model M is perfect it suffices to show that there are no minimal models preferable to M.

Proof: Suppose that there is no minimal models preferable to M and suppose that N is a model such that $N \ll M$. Let K be a minimal model such that $K \sqsubseteq N$. Then $K \ll M$, which is a contradiction. □

Example 7.16 Consider the program P_4 and its two models M_1 and M_2 in Example 7.11. Since $M_1 - M_2 = \{mary : female\}$ and $M_2 - M_1 = \{mary : male\}$ (by

d2) and we have $female < male$ based on Example 7.13, $M_1 \ll M_2$. Consequencely, M_1 is perfect. □

Example 7.17 For the program P_5 and its three minimal models M_1, M_2, M_3, we have $M_1 - M_2 = \{c_1 : r(s_2 \rightarrow \{\{1\}\})\}$ and $M_2 - M_1 = \{a_2 : p(f \rightarrow 1), b_1 : q(s_1 \rightarrow \{2\}, c_1 : r(s_2 \rightarrow \{\{1,2\}\})\}$ (by d3). Since $r < p$ based on Example 7.13, $M_1 \ll M_2$. Similarly we have $M_1 \ll M_3$. Consequencely, M_1 is the perfect model. □

Example 7.18 Consider the program P_6 in Example 7.14, following are its four models.

$$
\begin{aligned}
M_1 = \{&peter : person(in \rightarrow water), phil : person(can \rightarrow swim),\\
&peter : person(could \rightarrow sink)\}\\
M_2 = \{&peter : person(in \rightarrow water), phil : person(can \rightarrow swim),\\
&peter : person(can \rightarrow swim), peter : person(is \rightarrow happy),\}\\
M_3 = \{&peter : person(in \rightarrow water), phil : person(can \rightarrow swim),\\
&peter : person(can \rightarrow swim), peter : person(could \rightarrow sink)\}\\
M_4 = \{&peter : person(in \rightarrow water), phil : person(can \rightarrow swim),\\
&peter : person(can \rightarrow swim), phil : person(could \rightarrow sink),\\
&peter : person(could \rightarrow sink), phil : person(is \rightarrow happy),\\
&peter : person(is \rightarrow happy), phil : person(is \rightarrow happy)\}
\end{aligned}
$$

We have $M_1 - M_2 = \{peter : person(could \rightarrow sink)\}$ and $M_2 - M_1 = \{peter : person(can \rightarrow swim), peter : person(is \rightarrow happy)\}$. Since $could < can$, $M_1 \ll M_2$. Similarly, $M_1 \ll M_3$, $M_1 \ll M_4$, $M_2 \ll M_3$, $M_2 \ll M_4$, and $M_3 \ll M_4$. Consequently, M_1 is perfect. □

Definition 7.18 Given a normal program P, its declarative semantics is given by its unique perfect model M_P if it has. □

Definition 7.19 Given a normal program P and a query ?- $L_1, ..., L_n$, a correct answer to it is a ground substitution θ such that $M_P \models L_1\theta$, ..., $M_P \models L_1\theta$. □

7.4 Bottom-up Computation for Normal Programs

The last section gives a precise characterization of the semantics of a stratified normal program. The main results presented there are purely declarative. They do not immediately lead to any constructive method for evaluating the program and computing the answers to the query. This section shows that this perfect model can be obtained by bottom-up computation.

First, we re-define powers of an operator T_P of a program P.

Definition 7.20 Given a normal program $P = \langle S, OB, R \rangle$, the powers of the operator T_P is defined as follows:

$$T_P \uparrow 0(I) = I$$
$$T_P \uparrow n(I) = T_P(T_P \uparrow (n-1)(I)) \cup T_P(n-1)(I)$$
$$T_P \uparrow \omega(I) = lub\{T_P \uparrow n(I) \mid n \in \omega\}$$ □

Theorem 7.21 If T_P is monotonic, then

$$T_P \uparrow (n)(I) = T_P(T_P \uparrow (n-1)(I)) \cup I.$$

Proof: First we have

$$T_P \uparrow n(I) = T_P(T_P \uparrow (n-1)(I)) \cup T_P \uparrow (n-1)(I) \supseteq T_P \uparrow (n-1)(I).$$

Since T_P is monotonic, we have

$$T_P(T_P \uparrow n(I)) \supseteq T_P(T_P \uparrow (n-1)(I)). \quad (1)$$

Now we prove the theorem by induction on n. Since

$$T_P \uparrow 1(I) = T_P(T_P \uparrow 0(I)) \cup T_P \uparrow 0(I) = T_P(T_P \uparrow 0(I)) \cup I,$$

the theorem holds for $n = 1$. Assume it holds for $n = k$, that is,

$$T_P \uparrow k(I) = T_P(T_P \uparrow (k-1)(I)) \cup I \quad (2)$$

Consider $n = k + 1$, we have

$$\begin{aligned} T_P \uparrow (k+1)(I) &= T_P(T_P \uparrow k(I)) \cup T_P \uparrow k(I) \\ &= T_P(T_P \uparrow k(I)) \cup T_P(T_P \uparrow (k-1)(I)) \cup I \quad \text{by (2)} \\ &= T_P(T_P \uparrow k(I)) \cup I \quad \text{by (1)} \end{aligned}$$

Therefore, the theorem holds for all n. □

The above theorem says that Definition 7.20 generalizes Definition 7.10.

Definition 7.21 Let P be a normal program stratified by $P = P_0 \cup ... \cup P_n$. M_n is defined as follows.

$$\begin{aligned} &M_0 = S \cup OB, \\ &M_1 = T_{P_1} \uparrow \omega(M_0), \\ &M_2 = T_{P_2} \uparrow \omega(M_1), \\ &... \\ &M_n = T_{P_n} \uparrow \omega(M_{n-1}). \end{aligned}$$

□

Theorem 7.22 Let P be a normal program stratified by $P = P_0 \cup ... \cup P_n$. Then T_{P_i} is monotonically increasing, $1 \leq i \leq n$.

Proof: Directly from the definition. □

Let M_P be the unique perfect model of P. It is intended to show that M_n is a model of P and M_n is preferable to every other models of P, that is, $M_n = M_P$. We first consider a program which has only two strata.

Theorem 7.23 Let P be a program, stratified by $P = P_0 \cup P_1 \cup P_2$. If P has a model, then

(1). M_1 is a model of $P_0 \cup P_1$.
(2). M_2 is a model of P.
(3). If N is a model of P such that $N \sqcap M_1 = M_1$, then $M_2 \sqsubseteq N$.

Proof. (1): It follows directly from Theorem 7.16

(2): We first prove by induction on i that $T_{P_2} \uparrow i(M_1) \sqcap M_1 = M_1$. The claim is obviously true for $i = 0$. Since the program is stratified, applying T_{P_2} to $T_{P_2} \uparrow i(M_1)$ does not add any values to attributes and any objects to the classes in P_1. The claim follows.

Next, we show that M_2 is a model of P. Let r in P_2 be

$$A \Leftarrow L_1, ..., L_n \tag{1}$$

and assume that for some θ, $M_2 \models L_i\theta$, $1 \leq i \leq n$. Since $T_{P_2} \uparrow i(M_1)$ is monotonically increasing by Theorem 7.22 and $T_{P_2} \uparrow \omega(M_1)$ is its limit, For each i, there exist $\alpha(i)$ such that $T_{P_2} \uparrow \alpha(i) \models L_i\theta$, so for a sufficiently large q, we have $T_{P_2} \uparrow q \models L_k\theta$. For some $l > q$, r is one of the rules in P_2 applied to $T_{P_2} \uparrow l(M_1)$, to produce $T_{P_2} \uparrow (l+1)(M_1)$. It follows that $T_{P_2} \uparrow (l+1)(M_1) \models A\theta$; hence $T_{P_2} \uparrow \omega(M_1) \models A\theta$.

(3): Let N be a model such that $N \sqcap M_1 = M_1$. We show by induction that $T_{P_2} \uparrow i(M_1) \sqsubseteq N$. For $i = 0$, the claim is trivial. Assume the claim holds for $T_{P_2} \uparrow i(M_1)$, and consider $T_{P_2} \uparrow (i+1)(M_1)$. Let r be a rule in P_2 of the form (1) above, whose application to $T_{P_2} \uparrow i(M_1)$ adds $A\theta$ to $T_{P_2} \uparrow (i+1)(M_1)$. Since $N \sqcap M_1 = M_1$, the body of the rule, under the substitution θ, is satisfied by N. Since N is a model of R, we have $N \models A\theta$. Thus, $T_{P_2} \uparrow (i+1)(M_1) \sqsubseteq N$. □

Theorem 7.24 Let P be a normal program stratified by $P = P_0 \cup P_1 \cup P_2$. If P has a model, then M_2 is the unique perfect model of P.

Proof: We just need to prove that M_2 is preferable to every model of P. Let N be any model of P that is different from M_2. If $N \sqcap M_1 = M_1$, then by Theorem 7.23, $M_2 \sqsubseteq N$; hence $M_1 \ll N$. If $N \sqcap M_1 \neq M_1$, then it must contain something that is

not in M_1, and hence also not in M_2. It follows that in this case also, $M_2 \ll N$. □

Now we are ready for the general case.

Theorem 7.25 Let P be a normal program stratified by $P = P_0 \cup ... \cup P_n$. Suppose that P has a model. Then M_n is a model of P and for each $i, 1 \leq i \leq n$, $M_n \sqcap M_i = M_i$.

Proof: We show, using induction on i, that M_i is a model of $P_0 \cup ... \cup P_i$, and for all j, $0 \leq j < i$, $M_i \sqcap M_j = M_j$. When $i = n$, we prove the theorem.

For the basis, $i = 0$, the claim is trivially true. Assume the claim hold for some $i \geq 0$. By Theorem 7.23, M_{i+1} is a model of $P_0 \cup ... \cup P_{i+1}$ and $M_{i+1} \sqcap M_j = M_j$, $0 \leq j < i+1$. □

Theorem 7.26 Let P be a normal program stratified by $P = P_0 \cup ... \cup P_n$. Suppose that P has a model. Then M_n is preferable to every other model of P. That is $M_P = M_n$.

Proof: Let N be a model of P and is different from M_n. Then the restriction of N to the classes and attributes in $P_0 \cup ... \cup P_i$ is a model of $P_0 \cup ... \cup P_i$. Denote this restriction by N_i. Now, let j be the smallest integer such that $M_j \neq N_j$. Then $M_i = N_i$ for all $i \leq j$. From Theorem 7.24, it follows that $M_j \sqsubseteq N_j$. Therefore, if $M_n - N \neq \{\}$, the definition of each A in this difference must belong to P_k, for some $k > j$. It follows that M_n is preferable to N. □

This theorem tells exactly how to compute the perfect model M_P of the program P. The stratum zero contains $OB \cup S$. Then, some further classes and attribute

values defined (perhaps recursively) without the use of negation and sets in stratum 1 are obtained. Next, some new classes and new attribute values defined in terms of the previous ones, possibly with the use of negation and sets in stratum are obtained. This process is iterated. Then M_n is the perfect model.

Theorem 7.27 Given a program P and a query Q with type or label variables, whether or not there is an answer to the query is decidable, based on the above bottom up computation.

Proof: For the given program P, the types and attribute labels are finite. Thus, the substitutions to the type and label variables in the query Q are finite. □

This theorem says that even though we have type and label variables in NLO which make it higher-order, it is still decidable.

Chapter 8

Transformation into Prolog

Chapters 5, 6 and 7 have shown that NLO has an expressive syntax and sound semantics. This chapter shows that NLO is also implementable in practice. It shows that satisfiable NLO programs and queries can be transformed into semantically equivalent Prolog programs and queries and get correct answers.

It is assumed that in Prolog, there are three built-in predicates: $integer(X)$ which is true if X is substituted by an integer, $string(X)$ which is true if X is substituted by a string, $setof(X, P, S)$ which is true if the set of all substitutions of X such that P is true is S. In Prolog, there is no real concept of sets, but lists. But sets are normally represented indirectly by lists. To simplify the presentation of the transformation, it is assumed that sets are directly representable in Prolog in standard set notation {...}. It is also assumed that predicates *subset, member, union, intersection* and *difference* over sets are pre-defined.

8.1 Transformation of Types

A type in NLO is a name which may have a factual property and a number of definitional properties which impose constraints on the factual properties which all objects possessing this type should have. Associated with a type is a class which is the set of all known objects possessing this type. Therefore, each type as well

as class of NLO is transformed into five predicates: *type, object_of, attribute, attribute_value*, and *class*. The predicate *type* is used for denoting the existence of a type. If p is a type, then $type(p)$ will be in the transformed program. The predicate *object_of* is used for denoting that an object is known to belong to a class. If o is an object in the class p, then $object\_of(o, p)$ will be in the transformed program. The predicate *attribute* is used for the definitional properties of types. If p has a definitional property $l \rightarrow q$, then $attribute(p, l, q)$ will be in the transformed program. The predicate *attribute_value* is used for the factual properties of types. If p has a single-valued factual property $l \rightarrow q$, then $attribute\_value(p, l, q)$ will be in the transformed program. If p has a set-valued factual property $l \rightarrow \{q_1, ..., q_n$, then $attribute\_value(p, l, q_1)$,...,$attribute\_value(p, l, q_n)$ will be in the transformed program. The predicate *class* is used for the class associated with a type. If p is a finite type, and $a_1, ..., a_n$ are all elements of this type, then $class(p, \{a_1, ..., a_n\})$ will be in the transformed program. If p is an infinite type, then it is impossible to list all its extension and $class(p, p)$ is used. The meta type *type* is used only for the semantics of NLO programs so that it is not needed to include it in the transformed program.

8.1.1 Transformation of Built-in Types

The following transformations of the three built-in types, *integer*, *string* and *object* are included in the transformed program of NLO if they are used.

$type(integer)$.
$object\_of(X, integer)$:- $integer(X)$.
$class(integer, integer)$.

$type(string)$.
$object\_of(X, string)$:- $string(X)$.
$class(string, string)$.

$type(object)$.
$attribute\_value(object, isa, \{object\})$.
$class(object, X)$:- $setof(Y, object\_of(Y, object), X)$.

The first group says that *integer* is a type, objects of *integer* are integers and the class associated with the type *integer* is *integer*. The second group says that *string* is a type, objects of *string* are strings and the class associated with the type *string* is *string*. The last group says that *object* is a type, *object* has a factual property called *isa* whose value is $\{object\}$ and the class *object* contains all objects of type p. In fact, the last rule can be generalized for all finite classes as follows.

$class(T, X)$:- $type(T), T \neq integer, T \neq string, setof(Y, object\_of(Y, T), X)$.

8.1.2 Transformation of Set Types

According to the semantics of NLO, if p is a type, then $set(p)$ is a set type. The objects of the set type $set(p)$ are subsets of the class p. The class $set(p)$ is $set(p)$ if p is infinite, otherwise, is the power set of class p. Based on the semantics, following is its transformation.

$type(set(P))$:- $type(P)$.
$object\_of(X, set(P))$:- $type(set(P))$, $subset(X, Y), class(P, Y)$.
$class(set(P), set(P))$:- $class(P, P)$.
$class(set(P), X)$:- $setof(Y, (class(P, Z), subset(Y, Z)), X)$.

The third rule above deals with *integer* and *string*. Based on it, we can infer $class(set(integer), set(integer))$.

8.1.3 Transformation of Basic Types

Since basic types are directly used in the program, rather than explicitly defined, whenever they occur in the program, the following transformation will be used.

If $\{a_1, ..., a_n\}$ is a basic type, then it is transformed into

$type(\{a_1, ..., a_n\}).$
$object\_of(X, \{a_1, ..., a_n\})$:- $(X = a_1; ...; X = a_n)$,
$(object\_of(X, string); object\_of(X, integer))$.

If $s = \{a_1, ..., a_n\}$ is a basic type, then it is transformed into

$type(s).$
$object\_of(X, s)$:- $(X = a_1; ...; X = a_n)$,
$(object\_of(X, string); object\_of(X, integer))$.

If $\{lb..rb\}$ is a basic type, then it is transformed into

$type(\{lb..rb\}).$
$object\_of(X, \{lb..rb\})$:- $object\_of(X, integer), X \geq lb, X \leq rb.$

If $s = \{lb..rb\}$ is a basic type, then it is transformed into

$type(s).$
$object\_of(X, s)$:- $object\_of(X, integer), X \geq lb, X \leq rb.$

Example 8.1 The transformations of the basic types $gender = \{$ *"Male", "Female"*$\}$ and $\{0..120\}$ are as follows:

$type(gender).$
$object\_of(X, gender)$:- $object\_of(X, string)$,
$(X =$ *"Male"*; $X =$ *"Female"*$).$

$type(\{0..120\}).$
$object\_of(X, \{0..120\})$:- $object\_of(X, integer), 0 \leq X, X \leq 120.$ □

8.1.4 Transformation of Representational Types

For each representational type p with a definitional property defined by

$p : type\langle l \rightarrow q \rangle$, its transformation is

$type(p).$
$attribute(p, l, q)$:- $type(q).$

For each representational type with a factual property defined by

$p : type(isa \rightarrow \{p_1, ..., p_m\})$, its transformation is

$type(p).$

$attribute\_value(p, isa, \{p_1, ..., p_n\})$:- $type(p_1), ..., type(p_n).$

$object\_of(X, p_1)$:- $object\_of(X, p).$
...
$object\_of(X, p_n)$:- $object\_of(X, p).$

$attribute(p, L, Q)$:- $attribute(p_1, L, Q).$
...
$attribute(p, L, Q)$:- $attribute(p_n, L, Q).$

Example 8.2 Consider the following two representational types of NLO:

$person : type(isa \rightarrow \{object\})$
$\langle sex \rightarrow gender,$
$age \rightarrow \{0..120\}\rangle.$

$student : type(isa \rightarrow \{person\})$
$\langle age \rightarrow \{15..35\},$
$studying\_in \rightarrow dept,$
$taking \rightarrow set(course)\rangle.$

Their transformations are

$type(person).$
$attribute\_value(student, isa, object)$:- $type(object).$
$object\_of(X, object)$:- $object\_of(X, person).$
$attribute(person, L, Q)$:- $attribute(object, L, Q).$
$attribute(person, sex, gender)$:- $type(gender).$
$attribute(person, age, \{0..120\})$:- $type(\{0..120\}).$

$type(student)$.
$attribute\_value(student, isa, person)$:- $type(person)$.
$object\_of(X, person)$:- $object\_of(X, student)$.
$attribute(student, L, Q)$:- $attribute(person, L, Q)$.
$attribute(student, age, \{15..35\})$:- $type(\{15..35\})$.
$attribute(student, studying\_in, dept)$:- $type(dept)$.
$attribute(student, taking, set(course))$:- $type(set(course))$. □

8.2 Transformation of Objects

For a representational object o with a full factual property represented by $o : p(l \rightarrow o_t)$, the transformation is as follows:

$object\_of(o, p)$.
$attribute\_value(o, l, o_t)$.

For a representational object o with a partial factual property represented by $o : p(l \rightarrow \{o_1, ..., o_n\}')$, an additional predicate *attribute_member* is used for every member of the set. The transformation is as follows:

$object\_of(o, p)$.
$attribute\_member(o, l, o_1)$
...
$attribute\_member(o, l, o_n)$
$attribute\_value(O, L, X)$:- $setof(Y, attribute\_member(O, L, Y), X)$

Note that the rule above is quite general and applicable to all objects with partial factual attribute values. Thus it should be included in the transformed program.

Example 8.3 Consider the following two representational objects:

$mary : person(sex \rightarrow$ *"Female"*,
$age \rightarrow 28)$.
$john : person(age \rightarrow 35,$
$studies\_in \rightarrow math,$

$takes \rightarrow \{m203, m321, cs213\}'$,
$borrows \rightarrow \{prolog, databases\}$).

Their transformations are

$object\_of(mary, person)$.
$attribute\_value(mary, sex,$ *"Female"*$)$.
$attribute\_value(mary, age, 28)$.

$object\_of(john, person)$.
$attribute\_value(john, age, 35)$,
$attribute\_value(john, studies\_in, math)$.
$attribute\_member(john, taking, m203)$.
$attribute\_member(john, taking, m321)$.
$attribute\_member(john, taking, cs213)$.
$attribute\_value(john, borrow, \{prolog, databases\})$. □

8.3 Transformation of Basic Terms

Basic terms are used in queries and in rules either in head or in body. The transformation shown here only applies to the basic terms used in queries and in bodies of rules.

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

$object\_of(X, p)$.

For a basic term $X : p(l \rightarrow Y)$, where Y is either a variable, or an object, its transformation is

$object\_of(X, p)$,
$attribute\_value(X, l, Y)$.

For a basic term $X : p(l \rightarrow \{o_1, ..., o_n\}')$, its transformation is

$object\_of(X, p)$,
$attribute\_value(X, l, Y))$,
$member(o_1, Y)$,

...
$member(o_n, Y)$,

For a basic term $X : p(l \rightarrow \{Y\})$, where Y is a variable, its transformation is

$object\_of(X, p)$,
$attribute\_value(X, l, Z)$,
$member(Y, Z)$.

8.4 Transformation of Basic Literals

Similar to basic terms, the transformation shown here only applies to the basic literals used in queries and in bodies of rules.

Let ψ be a basic term and $trans(\psi)$ stand for the transformation of ψ. Then for arithmetic expressions, their transformation are straightforward:

$$trans(\psi_1 + \psi_2) = trans(\psi_1) + trans(\psi_2)$$

$$trans(\psi_1 - \psi_2) = trans(\psi_1) - trans(\psi_2)$$

$$trans(\psi_1 \times \psi_2) = trans(\psi_1) \times trans(\psi_2)$$

$$trans(\psi_1 \div \psi_2) = trans(\psi_1) \div trans(\psi_2)$$

For set expressions, their transformation are as follows.

$$trans(\psi_1 \cup \psi_2) = union(trans(\psi_1), trans(\psi_2))$$

$$trans(\psi_1 \cap \psi_2) = intersection(trans(\psi_1), trans(\psi_2)).$$

$$trans(\psi_1 \setminus \psi_2) = difference(trans(\psi_1), trans(\psi_2)).$$

For a negation of a basic term $\neg\psi$, its transformation consists of the negative sign followed by the transformation of the basic term without negation, i.e.

$$trans(\neg\psi) = \neg(trans(\psi)).$$

For a disjunctive basic term $\psi_1; \psi_2$, it transformation is

$trans(\psi_1; \psi_2) = trans(\psi_1);\ trans(\psi_2)$.

For arithmetic comparison expressions, their transformations are straightforward.

$trans(\psi_1 \leq \psi_2) = trans(\psi_1) \leq trans(\psi_2)$.

$trans(\psi_1 \geq \psi_2) = trans(\psi_1) \geq trans(\psi_2)$.

$trans(\psi_1 < \psi_2) = trans(\psi_1) < trans(\psi_2)$.

$trans(\psi_1 > \psi_2) = trans(\psi_1) > trans(\psi_2)$.

For set comparison expressions, their transformations are as follows.

$trans(\psi_1 \subset \psi_2) = subset(trans(\psi_1), trans(\psi_2)), trans(\psi_1) \neq trans(\psi_2)$.

$trans(\psi_1 \supset \psi_2) = subset(trans(\psi_2), trans(\psi_1)), trans(\psi_1) \neq trans(\psi_2)$.

$trans(\psi_1 \subseteq \psi_2) = (subset(trans(\psi_1), trans(\psi_2)); trans(\psi_1) = trans(\psi_2))$.

$trans(\psi_1 \supseteq \psi_2) = (subset(trans(\psi_2), trans(\psi_1)); trans(\psi_1) = trans(\psi_2))$.

8.5 Transformation of Rules

A rule consists of a head and a body of the form $A \Leftarrow body$. The body is a collection of basic literals $L_1, ..., L_n$, each of which is either a basic term, the negation of a basic term, a disjunctive basic term or an expression. The transformation of the body of a rule is just the conjunction of the transformation of the literals in the body. That is, $trans(body) = trans(L1), ..., trans(L_n)$.

The head of a rule is a basic term. But its transformation is different from the transformation of a basic term. It depends on the usage of the rule.

A rules can be used in two different ways. One is to deduce factual attribute values of existing representational objects. The other is to construct new representational objects and deduces their attribute values. In the later case, object constructors are

used.

Let $X : p(l \rightarrow Y) \Leftarrow body$ be a rule. If X is not an object constructor, then the rule only has the following transformation.

$attribute\_value(X, l, Y)$:- $trans(body)$.

Otherwise the rule has the following additional transformation.

$object\_of(p, X)$:- $type(p), trans(body)$.

Let $X : p(l \rightarrow \{o_1, ..., o_n\}') \Leftarrow body$ be rules. If X is not an object constructor, then the rule only has the following transformation.

$attribute\_member(X, l, o_1)$:- $trans(body)$.

...

$attribute\_member(X, l, o_n)$:- $trans(body)$.

Otherwise the rule has the following additional transformation.

$object\_of(p, X)$:- $type(p), trans(body)$.

Let $X : p(l \rightarrow \{Y\}) \Leftarrow body$ be a rule. If X is not an object constructor, then the rule only has the following transformation.

$attribute\_member(X, l, Y)$:- $trans(body)$.

Otherwise the rule has the following additional transformation.

$object\_of(p, X)$:- $type(p), trans(body)$.

Let r be a rule of the form:

$X : p(l_1 \rightarrow X_1, ..., l_n \rightarrow X_n) \Leftarrow body$.

If X is not an object constructor, then the rule r only has the following transformation.

$trans(X : p(l_1 \rightarrow X_1)$:- $(body))$.

...

$trans(X : p(l_1 \rightarrow X_n)$:- $(body))$.

Otherwise the rule has the following additional transformation.

$object\_of(p, X)$:- $type(p), trans(body)$.

Example 8.4 Consider the following two NLO rules which only deduce factual attribute values of existing objects.

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

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

Their transformations are:

(1) $attribute\_value(X, address, Y)$:- $A \leq 20$,
$object\_of(X, person)$,
$attribute\_value(X, age, A)$,
$attribute\_value(X, father, Z)$,
$object\_of(Z, person)$,
$attribute\_value(Z, address, Y)$.

(2) $attribute\_member(X, heading, Y)$:- $object\_of(Y, employee)$,
$attribute\_value(Y, works\_in, D)$,
$object\_of(D, dept)$,
$attribute\_value(D, head, X)$. □

Example 8.5 Consider the following rule which constructs new objects and deduce their factual attribute values.

$h(X, Y) : p(s_1 \rightarrow \{Z\}) \Leftarrow X : q(s_2 \rightarrow \{Y\}), Y : r(f \rightarrow Z)$.

The transformation is:

$object\_of(h(X, Y), p)$:- $object\_of(X, q)$,
$attribute\_member(X, s_2, Y)$,

$$object\_of(Y, r),$$
$$attribute\_value(Y, f, Z),$$

$$attribute\_member(f(h(X, Y), s_1, Z) \text{ :- } object\_of(X, q),$$
$$attribute\_member(X, s_2, Y),$$
$$object\_of(Y, r),$$
$$attribute\_value(Y, f, Z).$$ □

8.6 Transformation of Typed Terms and Literals

For a typed term $P : type$, its transformation is

$type(P).$

For a typed term $P : type(L \rightarrow \{Q\})$, where Q is a variable, its transformation is

$type(P),$
$attribute(P, L, Z),$
$member(Q, Z).$

For a typed term $P : type(L \rightarrow \{q_1, ..., q_n\})$, its transformation is

$type(P),$
$attribute(P, L, \{q_1, ..., q_n\}).$

For a typed term $P : type(L \rightarrow \{q_1, ..., q_n\}')$, Its transformation is

$type(P),$
$attribute(P, L, Q),$
$member(q_1, Q),$
...
$member(q_n, Q).$

For a typed term $P : type\langle L \rightarrow Q\rangle$, where Q is a variable, its transformation is

$type(P),$
$attribute(P, L, Q).$

For a typed term $P : type\langle L_1 \rightarrow P_1, ..., L_n \rightarrow P_n \rangle$, its transformation is

$trans(P : type\langle L_1, \rightarrow P_1 \rangle).$
...
$trans(P : type\langle L_n, \rightarrow P_n \rangle).$

For a typed term $X : P$, its transformation is

$type(P),$
$object\_of(X, P).$

For a typed term $X : P(L \rightarrow Y)$, its transformation is

$type(P),$
$object\_of(X, P),$
$attribute(X, L, Y).$

For a typed term $X : P(L \rightarrow Y')$, its transformation is

$type(P),$
$object\_of(X, P),$
$attribute(P, L, Z),$
$subset(Y, Z).$

For a typed term $X : P(L_1 \rightarrow Y_1, ..., L_n \rightarrow Y_n)$, its transformation is

$trans(X : P(L_1, \rightarrow Y_1)).$
...
$trans(X : P(L_n, \rightarrow Y_n)).$

For a typed literal $\delta_1 = \delta_2$, its transformation is

$trans(\delta_1) = trans(\delta_2).$

For the negation of a typed term $\neg\psi$, its transformation consists of the negative sign followed by the transformation of the typed term without negation, i.e.

$trans(\neg\psi) = \neg(trans(\psi)).$

For a disjunctive typed term $\psi_1; \psi_2$, it transformation is

$(trans(\psi_1); trans(\psi_2))$.

Example 8.6 Consider the following typed terms and literals:

(1) $workstudent : type(isa \rightarrow \{student, employee\})$.

(2) $person : type(L \rightarrow \{Q\})$.

(3) $\neg P : type(isa \rightarrow \{person\}')$.

(4) $P : type\langle L \rightarrow Q\rangle$.

Their transformations are as follows.

(1) $type(workstudent)$,
$attribute\_value(workstudent, isa, \{student, employee\})$.

(2) $type(person)$,
$attribute\_value(person, isa, Z)$,
$member(Q, Z)$.

(3) $\neg(type(P), attribute\_value(P, isa, Z), member(person, Z))$.

(4) $type(P)$,
$attribute(P, L, Q)$. □

8.7 Transformation of Queries

A query is a conjunction of literals of form ?- $L_1, ..., Ln$. Its transformation is just the conjunction of transformed literals, that is,

$trans(\ ?- L_1, ..., Ln) = ?- trans(L_1), ..., trans(L_n)$.

Example 8.7 Consider the following queries:

(1) ?- $X : person(children \rightarrow \{jenny\}')$

(2) ?- $mary : person(children \rightarrow \{X\}), X : person(children \rightarrow Y)$.

(3) ?- $P : type(L_1 \rightarrow Q_1)\langle L_2 \rightarrow Q_2\rangle$

(4) ?- $X : P(L \rightarrow Y)$.

Their transformations are

(1) ?- $object\_of(X, person)$,
$attribute\_value(X, children, Y)$,
$member(jenny, Y)$.

(2) ?- $object\_of(mary, person)$,
$attribute\_value(mary, children, X)$,
$object\_of(X, person)$,
$attribute\_value(X, children, Y)$.

(3) ?- $type(P)$,
$attribute\_value(P, L_1, Q_1)$,
$attribute(P, L_2, Q_2)$.

(4) ?- $type(P)$,

$object\_of(X, P)$,

$attribute\_value(X, L, Y)$. □

8.8 Correctness of Transformation

The previous sections have shown how to transform NLO programs and queries into Prolog programs and queries. The transformation is justified by the following theorems.

Theorem 8.1 Let P be an NLO program and assume that P is satisfiable and M_P is the intended semantics of P. Let P^* be the transformed program, that is, $trans(P) = P^*$ and M_{P^*} be the intended semantics of P^*. Then $trans(M_P) = M_{P^*}$.

Proof: Let $P = \langle S, OB, R \rangle$. Then $S \cup OB \sqsubseteq M_P$. It is straightforward to show that $trans(S) \cup trans(OB) \subseteq M_{P^*}$. Suppose $M_P \models o$ and o is not in $OB \cup S$, then

we can prove by induction on the stratum of P that $M_{P^*} \models trans(o)$, similar to the proof in Theorem 7.25 Therefore, $trans(M_P) \subseteq M_{P^*}$.

Now consider every fact in M_{P^*} which is one of the following forms: $type(p)$, $object\_of(o,p)$, $attribute(p,l,q)$, $attribute\_value(o,l,o_t)$, $attribute\_member(o,l,o_t)$, and $class(p,c)$. Then it is easy to show that

if $M_{P^*} \models type(p)$, then $M_P \models p : type$;

if $M_{P^*} \models object\_of(o,p)$, then $M_P \models o : p$;

if $M_{P^*} \models attribute(p,l,q)$, then $M_P \models p : type\langle l \rightarrow q \rangle$;

if $M_{P^*} \models attribute\_value(o,l,o_t)$, then $M_P \models o : p(l \rightarrow o_t)$ for some p;

if $M_{P^*} \models class(p,c)$, then $\pi(p) = c$;

if $M_{P^*} \models attribute\_member(o,l,o_t)$, then $M_P \models o : p(l \rightarrow \{o_t\}')$ for some p.

Therefore, $trans(M_P) = M_{P^*}$. □.

Theorem 8.2 Let P, M_P, P^* and M_{P^*} be the same as in the theorem above and Q be an arbitrary query and Q^* be its transformation. Then $M_P \models Q\theta$ iff $M_{P^*} \models Q^*\theta$.

Proof: Straightforward and omitted.

These two theorems say that satisfiable NLP programs and queries can be transformed into semantically equivalent Prolog programs and queries.

8.9 Summary

This chapter has shown how to transform NLO programs and queries into semantically equivalent Prolog programs and queries. Such transformation suggests that

NLO is fully implementable in practice, even though it is not intended to be implemented in this way.

Substantial sample examples have been tested using NU-Prolog and Quintus Prolog and operate correctly.

Chapter 9

Conclusion and Further Work

Approaches to deductive databases are subject to 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.

This thesis has analyzed two significant problems inherent in deductive databases, namely complex object modeling and higher-order features. It has discussed what should be incorporated to extend deductive databases, based on the work of object-oriented programming languages and semantics and object-oriented data models. To model complex objects, we need proper notions to represent object identity, single-valued properties and set-valued properties, syntactical sets, types, classes and inheritance. To represent and manipulate schema and sets, we need variables not only for individuals, but also for nested syntactical sets, types and property names.

Several typical solutions to these two problems have been examined, and why they cannot naturally and directly satisfy the above requirements has been shown.

Based on the requirements and related work, this thesis has proposed a novel

Criteria	NLO
Object is viewed as	surrogate
Object Identity	surrogate
Object Generation	Yes
Single-Valued Factual Property	function formation
Set-Valued Factual Property	function formation
Syntactic Sets	homogeneous
Nested Sets	Yes
Set Variables	Yes
Separation of Classes and Objects	Yes
Uniformity of Terms and Atoms	Yes
Type Definitions	Yes
Subtypes of Basic types	Yes
Inheritance	Yes
Well-defined Semantics	simple
Semantic Properties of Program	Yes

Table 9.1 Summary of NLO

deductive database language NLO which can naturally and directly support object identity, object properties, syntactical sets, types, classes, inheritance, schemas, sets, and higher-order queries in a uniform way. Therefore, it solves the above two problems.

The semantics of NLO given here is quite simple, natural and direct, compared to other approaches. The syntactic and semantic properties of NLO programs have also been investigated and the precisely defined semantics of NLO programs are given in the way similar to the traditional logic programming.

Table 9.1 summarizes the features of NLO based on the same criteria as in Tables 4.1 and 4.2 in Chapter 4. Table 9.2 summarizes Tables 4.1, 4.2 and 9.1 in an

Criteria	LOGIN	O-Logic	R-Logic	F-Logic	LDL	L^2	COL	NLO
O-view	surr	surr	surr	surr	tuple	tuple	tuple	surr
O-Identity	surr	surr	surr	surr	rel	rel	rel	surr
O-Gen.	No	Yes	Yes	Yes	rel	rel	rel	Yes
Single Val.	func	func	func	func	tuple	tuple	tuple	func
Set Val.	prolog	No	func	func	tuple	tuple	tuple	func
Syntac-Sets	list	No	hete	homo	hete	hete	homo	hete
Nested Sets	No	No	No	No	Yes	Yes	Yes	Yes
Set Variable	prolog	No	No	No	Yes	Yes	Yes	Yes
Separation	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Uniformity	No	Yes	Yes	Yes	No	No	No	Yes
Type Def.	some	No	No	Yes	rel	rel	rel	Yes
Subtypes	Yes	No	No	some	No	No	No	Yes
Inheritance	Yes	No	No	Yes	No	No	No	Yes
Well-definded	No	No	complex	complex	Yes	Yes	Yes	simple
Sem. Prop.	No	No	No	No	Yes	Yes	Yes	Yes

Table 9.2 Summary of Comparison of NLO with Other Approaches

abbreviated form.

This thesis has also shown that NLO can be transformed into Prolog so that it is fully implementable in practice. NLO is intended as a real deductive database language, and how to implement it efficiently is a worthwhile topic for further research.

The rest of this chapter discusses two possible extensions to NLO. Section 1 discusses the update problem which is a very important aspect of database applications. Section 2 discusses the use of type and label variables in NLO to increase its expressive power.

9.1 Updates

There is another significant problem existing in deductive databases which is not dealt with in this thesis. This 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.

The notion of states is inherent in any notion of updates. The Dynamic Logic approach assigns state transition semantics to a logic program [NK88]. 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 [NK88] 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 given in that paper which can ensure this property.

The Dynamic Logic interpretation of updates [NK88] 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 [NTR87]. 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 [NTR87], 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 [Cle90]. 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.

It seems that it is possible to extend NLO based on the ideas of Starlog to solve the update problem, i.e, incorporating an explicit temporal dimension into NLO. However, more work is needed substantially.

9.2 Type and Label Variable in NLO Program

The theory developed in this thesis requires programs having no type and label variables. This requirement restricts the expressive power of NLO. By allowing type and label variables, all the built-in semantics of NLO are syntactically expressible. The following rules exemplify this:

(1) $P : type\langle L \rightarrow T\rangle \Leftarrow P : type(isa \rightarrow \{Q\}), Q : type\langle L \rightarrow T\rangle.$

(2) $O : Q \Leftarrow P : type(isa \rightarrow \{Q\}), O : P.$

(3) $set(P) : type \Leftarrow P : type.$

Here P, Q, L, T, O are all variables. The first and second rules are for the factual attribute *isa* of types. The first one says that if P is a subtype of Q, then all definitional properties of Q are also definitional properties of P. The second says if P is a subtype of Q, then all objects of P are also objects of Q. The last rule says that if P is a type, then $set(P)$ is also a type.

It seems possible that a theory of local stratification can be developed to deal with these requirements, similar to the local stratification theory in [Prz88]. However, unlike normal stratification, local stratification of a logic program cannot be statically checked but must be dynamically checked. Further work is needed to explore such semantics in NLO.

Bibliography

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. *Towards a Theory of Declarative Knowledge*, chapter 2, pages 89–148. In Minker [Min88], 1988.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Trans. on Database Systems*, 10(2):230–260, June 1985.

[AFOP88] A.Albano, F.Giannotti, R. Orsini, and D. Pedreschi. *The Types System of Galileo*, chapter 8. Springer-Verlag, 1988.

[AG88] S. Abiteboul and S. Grumbach. COL: A Logic-Based Language for Complex Objects. In Schmidt et al. [SCM88], pages 271–293.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Trans. on Database Systems*, 12(4):525–565, December 1987.

[AK89] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–173, 1989.

[AKN86] H. Ait-Kaci and R. Nasr. LOGIN: A Logic Programming Language with built-in Inheritance. *J. Logic Programming*, 3(3):198–215, October 1986.

[And91] T. Andrews. *Programming with Vbase*, chapter 9. In Gupta and Horowitz [GH91], 1991.

[Bee89] C. Beeri. Formal models for object-oriented databases. In Kim et al. [KNN89], pages 405–430.

[BJ89] G.B. Boolos and R.C. Jeffrey. *Computability and Logic.* Cambridge University Press, 3 edition, 1989.

[BMS84] M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors. *On Conceptual Modelling.* Springer-Verlag, 1984.

[BNST91] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Construction in a Logic Database Language. *J. Logic Programming*, 10(3,4):181–232, April/May 1991.

[Boc86] J. Bocca. On the Evaluation Strategy of EDUCE. In *Proc. ACM Syum. on Principles of Database Systems*, 1986.

[Bor88] A. Borgida. *Class Hierarchies in Information Systems: Sets, Types or Prototypes*, chapter 10. Springer-Verlag, 1988.

[Bra86] Ivan Bratko. *Prolog Programming For Artificial Intelligence.* Addison-Wesley, 1986.

[Bro84] M.L. Brodie. *On the Development of Data Models*, pages 19–48. In Brodie et al. [BMS84], 1984.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Proc. Intl. Symp. on Semantics of Data Types*, pages 51–67. Springer-Verlag Lecture Notes in Computer Science 173, June 1984.

[CGT90] S. Ceri, G. Gottlob, and T. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[CGW86] S. Ceri, G. Gottlob, and G. Wiederhold. Efficient Database Access through Prolog. *IEEE Trans. on Software Engineering*, Feb 1986.

[Cle90] J. G. Cleary. Colliding Pucks Solved Using a Temporal Logic . In *Proc. Conf. on Distributed Simulation*, San Diego, Jan 1990.

[Cod79] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems*, 4(4):297–434, December 1979.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, Dec 1985.

[DKV88] S. Danforth, S. Khoshafian, and P. Valduriez. FAD – A Database Programming Language. Technical report, MCC, 1988.

[GG83] D. Gabbay and F. Guenthner, editors. *Handbook of Philosophical Logic*, volume 1. D. Reidel Publishing, 1983.

[GG84] M.R. Genesereth and M.L. Ginsberg. Logic Programming. *Comm. ACM*, 28(9):933–941, September 1984.

[GH91] R. Gupta and E. Horowitz, editors. *Object-Oriented Databases with Applications to Case, Networks and VLSI CAD.* Prentice Hall, 1991.

[GM92] J. Grant and J. Minker. The Impact of Logic Programming on Databases. *Comm. ACM*, 35(3):67–81, March 1992.

[GMN84] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–186, June 1984.

[Gol81] W.D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Hat82] W.S. Hatecher. *The Logical Foundations of Mathematics.* Pergamon Press, 1982.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Trans. Database Systems*, 6(3):351–386, September 1981.

[Hue73] R. Huel. The Undecidability of Unification in Third-Order Logic. *Information and Control*, 22:257–267, 1973.

[KBC+87] W. Kim, J. Banerjee, H.T. Chou, J.F. Garza, and D. Woelk. Composite Object Support in an Object-Oriented Database System. In *OOPSLA '87 Proceedings*, 1987.

[KL89] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 134–146, 1989.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14, Dept of CS, SUNY at Stony Brook, 1990.

[KN88] R. Krishnamurthy and S. Naqvi. Towards a Real Horn Clause Language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 252–263, Los Angles, USA, 1988.

[KNN89] W. Kim, J.M. Nicolas, and S. Nishio, editors. *Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989. North-Holland.

[Kup87] G.M. Kuper. Logic Programming with Sets. In *Proc. ACM Syum. on Principles of Database Systems*, pages 11–20, 1987.

[KW89] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited). In *Proc. ACM Syum. on Principles of Database Systems*, pages 379–393, 1989.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2 edition, 1987.

[LR89] C. Lecluse and P. Richard. The O_2 Database Programming Language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 411–422, Amsterdam, The Netherlands, 1989.

[Mai86] D. Maier. A Logic for Objects. Technical Report CS/E-86-012, Oregon Graduate Center, Beaverton, Oregon, 1986.

[Mai87] D. Maier. Why Database Languages are a Bad Idea. In *Proc. Workshop on Database Programming Languages*, Roscoff, France, Sept 1987.

[MBW80] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Trans. on Database Systems*, 5(2):185–207, June 1980.

[Min88] J. Minker, editor. *Foundation of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, 1988.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of Object-Oriented DBMS. In *OOPSLA '86 Proceedings*, pages 472–482. ACM New York, 1986.

[NK88] S. Naqvi and R. Krishnamurthy. Database Updates in Logic Programming. In *Proc. ACM Syum. on Principles of Database Systems*, pages 261–272, 1988.

[NTR87] L. Naish, L.A. Thom, and K. Ramamohanarao. Concurrent Database Updates in Prolog. In *Proc. Intl. Conf. on Logic Programming*, pages 178–189, 1987.

[PM88] J. Peckham and F. Maryanski. Semantic Database Models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

[Prz88] T.C. Przmusinski. *On the Declarative Semantics of Deductive Databases and Logic Programs*, chapter 5, pages 193–216. In Minker [Min88], 1988.

[Rei84] R. Reiter. *Towards a Logical Reconstruction of Relational Database Theory*, pages 191–233. In Brodie et al. [BMS84], 1984.

[SCM88] J.W. Schmidt, S. Ceri, and M. Missikoff, editors. *Proceedings of International Conference on Extending Database Technology*, Venice, Italy, March 1988. Springer-Verlag Lecture Notes in Computer Science 303.

[She88] J.C. Shepherdson. *Negation in Logic Programming*, chapter 1, pages 19–88. In Minker [Min88], 1988.

[Shi79] D.W. Shipman. The Functional Extending the Database Relational Model to Capture More Meaning. *ACM Trans. on Database Systems*, 4(4):297–434, December 1979.

[SS77] J.M. Smith and D.C.P. Smith. Database Abstraction: Aggregation and Generalization. *ACM Trans. on Database Systems*, 2(2):105–133, June 1977.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[Su86] S.Y.W. Su. Modeling Integrated Manufacturing Data with SAM*. *IEEE Computer Society*, 19(1):34–49, January 1986.

[TL86] D.C Tsichritzis and F.H. Lochovsky. NU – Prolog Reference Manual. Technical Report 86/10, Dept of CS, Univ. of Melbourne, 1986.

[TZ86] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data Language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 33–41, Kyoto, Japan, 1986.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[vBD83] J. van Benthem and K. Doets. *Higher-Order Logic*, pages 275–329. Volume 1 of Gabbay and Guenthner [GG83], 1983.

[War82] D.H.D. Warren. Higher-Order Extensions to Prolog: Are they Needed? In J.E. Hayes, D. Michale, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood with John Willey and Sons, 1982.

[ZAKB+85] C. Zaniolo, H. Act-Kaci, H. Beech, D. Cammarata, and L. Kerschberg. Object-Oriented Database Systems and Knowledge Systems. Technical Report DB-038-85, MCC, 1985.

[Zal88] Edward N. Zalta, editor. *Intensional Logic and The Metaphysics of Intentionality.* The MIT Press, 1988.

[Zan89] Carlo Zaniolo. Object identity and inheritance in deductive database —an evolutionary approach. In Kim et al. [KNN89], pages 7–21.