Author:

Wei Li
State Key Laboratory of Software Development Environment
Beihang University
37 Xueyuan Road, Haidian District
Beijing 100191
China
e-mail: liwei@nlsde.buaa.edu.cn

2000 Mathematics Subject Classification: 83C05, 83C35, 58J35, 58J45, 58J05, 53C80

Library of Congress Control Number: 2009940118

Bibliographic information published by Die Deutsche Bibliothek.
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at http://dnb.ddb.de


This work is subject to copyright. All rights are reserved, whether the whole or part of the
material is concerned, specifically the rights of translation, reprinting, re-use of illustra-
tions, recitation, broadcasting, reproduction on microfilms or in other ways, and storage
in data banks. For any kind of use permission of the copyright owner must be obtained.

© 2010 Birkhäuser Verlag AG
Basel · Boston · Berlin
P.O. Box 133, CH-4010 Basel, Switzerland
Part of Springer Science+Business Media
Printed on acid-free paper produced from chlorine-free pulp. TCF∞
Printed in Germany


9 8 7 6 5 4 3 2 1  www.birkhauser.ch
Contents

Preface ix

Chapter 1 Syntax of First-Order Languages 1
  1.1 Symbols of first-order languages 4
  1.2 Terms 6
  1.3 Logical formulas 8
  1.4 Free variables and substitutions 9
  1.5 Gödel terms of formulas 13
  1.6 Proof by structural induction 15

Chapter 2 Models of First-Order Languages 19
  2.1 Domains and interpretations 22
  2.2 Assignments and models 24
  2.3 Semantics of terms 24
  2.4 Semantics of logical connective symbols 25
  2.5 Semantics of formulas 27
  2.6 Satisfiability and validity 30
  2.7 Valid formulas with ↔ 31
  2.8 Hintikka set 33
  2.9 Herbrand model 35
  2.10 Herbrand model with variables 38
  2.11 Substitution lemma 41
  2.12 Theorem of isomorphism 42

Chapter 3 Formal Inference Systems 45
  3.1 G inference system 49
  3.2 Inference trees, proof trees and provable sequents 52
  3.3 Soundness of the G inference system 57
  3.4 Compactness and consistency 61
  3.5 Completeness of the G inference system 63
  3.6 Some commonly used inference rules 66
  3.7 Proof theory and model theory 68

Chapter 4 Computability & Representability 71
  4.1 Formal theory 72
  4.2 Elementary arithmetic theory 74
  4.3 P-kernel on \( \mathbb{N} \) 76
  4.4 Church-Turing thesis 80
  4.5 Problem of representability 81
4.6 States of $P$-kernel ........................................ 82
4.7 Operational calculus of $P$-kernel ........................ 84
4.8 Representations of statements ............................ 86
4.9 Representability theorem ................................. 95

Chapter 5 Gödel Theorems .......................... 97
5.1 Self-referential proposition ............................... 98
5.2 Decidable sets ........................................ 100
5.3 Fixed point equation in $\Pi$ ............................ 104
5.4 Gödel’s incompleteness theorem ........................ 107
5.5 Gödel’s consistency theorem ........................... 109
5.6 Halting problem ..................................... 112

Chapter 6 Sequences of Formal Theories .................. 117
6.1 Two examples ....................................... 118
6.2 Sequences of formal theories ........................... 122
6.3 Proschema ......................................... 125
6.4 Resolvent sequences .................................. 128
6.5 Default expansion sequences ........................... 130
6.6 Forcing sequences .................................... 133
6.7 Discussions on proschemes ............................. 136

Chapter 7 Revision Calculus ............................... 139
7.1 Necessary antecedents of formal consequences ........... 140
7.2 New conjectures and new axioms ........................ 143
7.3 Refutation by facts and maximal contraction ............. 144
7.4 $R$-calculus ........................................ 146
7.5 Some examples ...................................... 153
7.6 Special theory of relativity .............................. 155
7.7 Darwin’s theory of evolution ............................ 156
7.8 Reachability of $R$-calculus ............................. 160
7.9 Soundness and completeness of $R$-calculus ............... 163
7.10 Basic theorem of testing ............................. 164

Chapter 8 Version Sequences .............................. 169
8.1 Versions and version sequences ......................... 171
8.2 The Proscheme OPEN ................................ 172
8.3 Convergence of the proscheme ......................... 176
8.4 Commutativity of the proscheme ....................... 178
8.5 Independence of the proscheme ......................... 180
8.6 Reliable proschemes .................................. 182
Preface

Classical mathematical logic is considered to be an important component of the foundation of mathematics. It is the study of mathematical methods, especially the properties of axiom systems and the structure of proofs. The core of mathematical logic consists of defining the syntax of first-order languages, studying their models, formalizing logical inference and proving its soundness and completeness. It also covers the theory of computability and Gödel’s incompleteness theorems. This process of abstraction started in the late 19th Century and was essentially completed by 1950.

In 1990, I began to give courses on mathematical logic. This teaching experience made me realize that, although deductive logic was well analyzed, the process of axiomatization had not been studied in depth. Several years later, I organized a series of seminars as an ensuing effort. The first five seminars covered classical mathematical logic and the rest were a preliminary outline of the formal theory of axiomatization.

As my understanding of mathematical logic became deeper, my desire to analyze and formalize the process of axiomatization became more intense. I also saw the influence of mathematical logic in information technology and scientific research. This inspired me to write a book for students living in the information society.

The computer was invented in the 1940’s and high-level programming languages were defined and implemented soon afterwards. Computer science has developed rapidly since then. This exerted a profound influence on mathematical logic, because its concepts and theories were extensively applied. However, the development of computer science has, in turn, made new demands on mathematical logic, which have been the focus of my research and the motivation for this book. This motivation is guided by two considerations.

Firstly, mathematical logic was originally a general theory about axiom systems and proofs in mathematics, but now, its concepts and theories have been adopted by computer science and have played a principal guiding role in the design and implementation of both software and hardware.

For example, the method of structural induction was invented to define the grammar of first-order languages, but it is now used to define programming languages. This suggests that the study of mathematical logic can be applied to many areas of computer science.

Another example is given by Peano’s theory of arithmetic. This is a formal theory in a first-order language, while the natural number system is a model of that theory. The distinction is essential in mathematical logic, because it is necessary in order to prove important theorems such as those of Gödel. However, many people outside this field find it hard to see the utility of making this distinction.

But in computer science, it is vital to differentiate between a high-level programming language and compiled executable codes. The difference between programs and their compiled executables is precisely the same as that made between first-order lan-
guages and their models, so the theorems of mathematical logic can be directly applied to study the properties and correctness of software systems.

These two examples show how mathematical logic is necessary to computer science, but we have also found the concepts of computer science helpful in understanding logic. For instance, students often find the process of Gödel coding difficult to grasp. To help them, we can make an analogy with computer science. In this, formulas are viewed as variable names in a programming language; the Gödel coding corresponds to the mechanism of assigning a pointer and the Gödel number corresponds to the address of the pointer, whose content is the Gödel term. This analogy helps students to understand and use these difficult concepts.

So I aspired to write a book that not only studies mathematical logic but also enlightens those who are living in the information society and are doing scientific research. This is why this book tries to illustrate the concepts, theories and methods of mathematical logic with the practical use of computers, programming languages and software, so that we can see the close relationship between mathematical logic and computer science.

The second motivation for this book is that research in computer science and technology during the last 60 years has developed many valuable methods and theories that are not covered by classical mathematical logic. I have long cherished a hope that mathematical logic could be enriched and extended to include these concepts. This aim has guided my research into investigating the following basic problems:

1. Software version

   A software system is written in a programming language and its specification may be described by the formal theory of a first-order language. However, its implementation rarely completely satisfies the requirements of its designers or users. It can only be implemented through frequent exchange and close collaboration between the developers. This leads to a process of evolution through a series of versions.

   It is only by distinguishing the different versions of the software that the exchange and collaboration between developers can be managed. Therefore, mathematical logic needs to incorporate the concepts of a version of a formal theory and of a version sequence, so that the evolution of formal theories can be described and studied.

2. Testing and debugging

   Testing is crucial in software development. Software can only be released after it has passed rigorous tests. Many tools have been developed to assist this process. In spite of this, software testing still requires much manpower and it is a skilled craft that depends on the proficiency and experience of the testing personnel.

   On the whole, software testing has two parts: designing test cases and finding and correcting software errors. Both of these require logical analysis, but this is different from the logical inference used in mathematical proof. Since mathematical proof is formally defined, we can perform it with the aid of interactive software systems. In the same way, we would like to build software tools to locate errors and to revise existing versions. If the concepts of error correction can be expressed in mathematical logic, then the goal of ‘mechanization’ could be realized. This research should play a guiding role in improving the efficiency of software testing.
3. The methodology of software development

The quality of software products is determined by the methodology of their development. Generally speaking, this methodology mainly consists of rules and workflows, which are managed by software tools. We would like to study this methodology as an object in mathematical logic. In this way, we could define a programming-like language to formally describe different methodologies of software development and could study their properties and prove their reliability.

4. Meta-language environment

First-order languages and their models are defined and specified in the meta-language environment and, in addition, many important theorems are proved in this environment. This will inevitably impose requirements and restrictions on the meta-language environment, so mathematical logic must specify clearly the principles that the environment must obey.

In general, any theory of mathematics or natural science is formed by a kind of evolutionary process, which is manifested as a series of different versions at different stages of development. Scientific theories are developed over a long period of time because only a limited number of experts are involved. The scale of their principles and theorems is far smaller than that of software systems and the time needed for their development is much longer. Therefore, the different versions of the theory are not so obvious as in software development.

For this reason, classical mathematical logic only takes a particular version of an axiom system as its object of study and deduces the logical consequences within that version. However, problems such as managing versions and version sequences, revision of theories, selecting methodologies of scientific research and consideration of the meta-language environment are important in the process of development of all theories. So these are all problems which mathematical logic should now define and formally analyze.

The book consists of two parts, each containing five chapters. The first part presents the core ideas of classical mathematical logic, while the second part deals with the author’s work on formalizing axiomatization.

The second part includes a definition of versions of a formal theory, version sequences and their limits. It formalizes the revision of formal theories, defines the concept of proscheme, and uses it to describe a methodology for the evolution of formal theories. It goes on to study inductive inference and prescribes the principles of a meta-language environment. These are an extension and development of classical mathematical logic.

This book adopts the rigorous standards of classical mathematical logic: All concepts are strictly defined and illustrated with examples; all theorems are proved and details of proofs are provided if at all possible; all quoted conclusions and methods are referred to their original authors and sources. This book is intended to be a course book for postgraduate students of information science, but the first five chapters may be used as a textbook for undergraduate students.

Although several major revisions have been made of the draft of this book in the past few years, I do not claim that the present text is free of omissions or even errors. I would sincerely appreciate any criticisms or suggestions.
Many colleagues and students of mine read my manuscripts and contributed to the preparation of this book. Their comments and suggestions led to significant improvements in the content and presentation of the book. In particular, I would like to mention Jie Luo, Shengming Ma, Dongming Wang, and Yuping Zhang, who helped me considerably in preparing the English version, typesetting, proofreading and giving many useful suggestions. Jie Luo and Shengming Ma supplied a detailed proof of the theorem of representability in Appendix 3. My sincere thanks go to all of them for their generous support, help, and contribution. My heartfelt thanks also go to Bill Palmer for his passionate and professional efforts in language editing.

My wife Hua Meng was the first to advise me to distill my research and understanding of mathematical logic into a book. She and my daughter Xiaogeng Li looked on my writing as one of the most important events in my family. It is hard to tell how long the publication of this book would have been delayed without their loving care and constant support and encouragement. I dedicate this book to them with gratitude.

Wei Li

Beihang University, Beijing

September 2009
Chapter 1
Syntax of First-Order Languages

Programming languages such as BASIC, Pascal, and C are formal languages used for writing computer programs. A program usually implements an algorithm which describes the computational solution of a specific issue. This chapter introduces a different kind of formal language, known as a first-order language.

A first-order language is used to describe the properties and relationships between the objects in a specific domain. Usually, these domains are mathematical or scientific in nature. For example, the axioms, theorems, and corollaries in plane geometry, the properties of natural numbers, and the laws and principles in physics are objects that can be described by first-order languages.

We usually start describing a domain by defining the properties of its objects. Each property is described by one or more propositions.

For example, the following propositions describe aspects of number theory:

“1 is a natural number.”
“No two different natural numbers have the same successor.”
“If \( a > 1 \) and \( a \) cannot be divided by 2, then \( a \) is an odd number.”

And the following describe knowledge of physics:

“A photon is a rigid body.”
“The velocity of light does not depend on the velocity of the body emitting the light.”
“A rigid body will continue in a state of rest or of uniform motion in a straight line unless it is acted upon by a force.”

Lastly, the following describe relationships between people:

“Confucius is a human.”
“Zisi is a descendant of Confucius.”
“If \( A \) is a descendant of \( B \) and \( B \) is a descendant of \( C \), then \( A \) is a descendant of \( C \).”

It should be pointed out that assertions, statements or even specifications are used instead of propositions in some other books on mathematical logic. For the sake of simplicity and uniformity, we use propositions in this book to denote the properties of the objects in a domain.

Our knowledge of a domain is composed of propositions which describe the properties of and relationships between objects. The kernel of these propositions forms an axiom system such as the axioms of Euclidean geometry or the set of laws in classical mechanics. Specifications of functional requirements for software systems are also axiom systems that describe domain knowledge.
First-order languages are specifically useful to describe axiom systems because they allow us to reason from the axioms with a symbolic calculus, which can be implemented as computer software.

Computer programs use commands or statements to specify computations. The purpose of computation is to solve a problem algorithmically. In contrast, axiom systems use propositions to describe the properties of and relationships between objects in a domain. Logical inference rules are used to deduce the logical consequences of axioms in a mechanical way. They explore the logical structure of a domain, finding all propositions that are provable from the axioms.

What do we mean when we say that a programming language is a formal language? We mean that it is constructed from an alphabet which is a set of symbols. These symbols are used to define several kinds of syntactic objects such as program declarations and statements, and each syntactic object is strictly defined by a specific grammar, which is a set of syntactic rules. Only programs written in strict accordance with the grammar can convert algorithms into mechanical operations executable on computers.

In the same way, a first-order language is also a formal language. It is based upon a set of symbols and is composed of two kinds of syntactic objects. Each syntactic object has a specific syntactic structure and is defined by a set of rules. If an axiom system is defined in strict accordance with the syntactic rules of first-order languages, we can convert logical reasoning about a domain into symbolic calculus.

The difference between first-order languages and programming languages lies in the fact that the description of the knowledge of each specific domain requires a specific first-order language, while any computable problem can be solved by programs written in any programming language.

Let us discuss what sets of symbols and syntactic objects a first-order language should contain. The symbols used by each first-order language should be of two types. One of them is related to specific domain knowledge and these are special symbols used by this language and are called \textit{domain specific symbols}. The other consists of symbols common to the description of every domain, which are called \textit{logical symbols}.

Symbols related to specific domain knowledge may be further divided into two types. One type is used to describe constants and functions and consists of \textit{constant symbols} and \textit{function symbols}. The other type is used to describe relationships between concepts and the symbols are called \textit{predicate symbols}. The following are some examples of constant symbols, function symbols, and predicate symbols:

1. \textbf{Constant symbols:} 0, \(\pi\), and \(e\) are constants in mathematics. The acceleration of gravity \((g)\), universal gravitational constant \((G)\), and the velocity of light \((c)\) are constants in physics. Confucius and Zisi (the grandson of Confucius) are both constants describing a human relationship. Every constant of a domain is described by a specific constant symbol in a first-order language for the domain.

2. \textbf{Function symbols:} The successor \(\sigma\) of \(x\) defined by \(\sigma(x) = x + 1\) is a unary function, and addition and multiplication are binary functions in number theory. \(\sin x\), \(\cos x\), \(\ln x\), \(\exp x\) are functions used in physics. Each function of a domain is described by a specific function symbol in a first-order language for the domain.
 Predicate symbols: “is prime,” “is even,” and “is odd” are some of the basic properties of natural numbers, “=”, “<” are basic logical relations in number theory, “rigid body,” “velocity,” “force,” etc. are basic concepts in physics, and “descendant” is a basic relation between humans. In academic research, we use natural\( (x) \) to denote “\( x \) is a natural number,” even\( (y) \) to denote “\( y \) is an even number,” rigid\( (z) \) to denote “\( z \) is a rigid body,” \( P(x,y) \) to denote “\( x \) is the descendant of \( y \),” and so on. In general, a basic property in a domain is described by a specific predicate symbol in first-order languages for the domain.

There are three other kinds of symbols, namely, variables, logical connectives, and quantifiers, which are needed to specify logical statements about a domain. They are called logical symbols in first-order languages.

Symbols of the first kind are the variables occurring in functions and predicates, such as \( x, y, \) and \( z \) in the previous examples. Conceptually, they are the same as the variables defined in programs. In first-order languages they are also called variable symbols.

Symbols of the second kind denote logical connectives occurring in propositions. Each proposition in a domain is composed of basic statements combined by logical connectives. There are five commonly used logical connectives:

“negation of …,” “… and …,” “… or …,”
“If … then …,” “… if and only if …”

For example, in the proposition “if \( 1 < a \) and \( a \) cannot be divided by 2, then \( a \) is an odd number,” “cannot be divided by 2” is the negation of “can be divided by 2,” while “if \( 1 < a \) and \( a \) cannot be divided by 2” is connected by using “and.” Finally, the proposition takes the form “if … then ….” In fact, most other logical connectives may be expressed as combinations of these five logical connectives.

As in programming languages, special symbols are introduced in first-order languages to denote logical connectives:

- Logical connective: negation, and, or, if … then …, if and only if
- Special symbol: \( \neg \), \( \land \), \( \lor \), \( \rightarrow \), \( \iff \)

By using the above symbols, the proposition “if Kongrong is a descendant of Zisi and Zisi is a descendant of Confucius, then Kongrong is a descendant of Confucius” can be described as

\[
P(Kongrong, Zisi) \land P(Zisi, Confucius)) \rightarrow P(Kongrong, Confucius).
\]

A third kind of symbol is used to describe generality. These symbols are called quantifiers and are described in first-order languages by the following symbols:

- Quantifier: for all \( x \) …, there exists an \( x \) …
- Special symbol: \( \forall x \) …, \( \exists x \) …

They deal with the instantiation or universality of a concept. In other words whether a property holds for some object or all objects in a domain. Thus, the proposition “for all \( x \),
y, z, if x is a descendant of y and y is a descendant of z, then x is a descendant of z” can be expressed as
\[ \forall x \forall y \forall z ((P(x,y) \land P(y,z)) \rightarrow P(x,z)), \]
and read as: for all x, y, z, if P(x,y) holds and P(y,z) holds, then P(x,z) holds. In the above formula, parentheses are used to indicate the priority of logical connectives. If \((P(x,y) \land P(y,z))\) does not have outer parentheses, then it is not clear whether \(P(x,y) \land P(y,z) \rightarrow P(x,z)\) should be read as “P(x,y) holds and if P(y,z) holds, then P(x,z) holds,” or as “if P(x,y) holds and P(y,z) holds, then P(x,z) holds.” The outer parentheses of \((P(x,y) \land P(y,z))\) indicate that we mean the latter. Therefore, parentheses are indispensable in first-order languages.

A programming language may contain several kinds of syntactic objects, whereas a first-order language has only two kinds of syntactic objects, i.e., “terms” and “logical formulas.” Terms are used to describe constants, variables, and functions, while logical formulas are used to describe propositions. We shall see later that the “terms” and “logical formulas” of first-order languages are defined by different syntactic rules.

In summary, first-order languages are formal languages used to describe propositions about knowledge domains. The aim of introducing first-order languages is to convert logical reasoning into symbolic calculus. We have explained briefly the symbols and syntactic ingredients that a first-order language should have. The purpose of this chapter is to give formal definitions of these concepts. This chapter also discusses a specific method of defining first-order languages, which is called definition by structural induction and the powerful mathematical proof techniques that this method induces.

It should be noted that historically, first-order languages appeared earlier than programming languages. It was during the theoretical study of first-order languages that computability was thoroughly studied and formally defined, followed by the invention of computers. In order for computer to be programmed easily, scientists began to design high-level programming languages using the theory of formal languages, which had matured through the study of first-order languages. As a result of the popularization of computers, programming languages began to be taught at high schools. The fundamental ideas and methods of first-order languages have been widely accepted, if mostly unknowingly, through the use of programming languages. Therefore, first-order languages are presented in this book in comparison with daily-used programming languages so that they may become easier to understand and master.

### 1.1 Symbols of first-order languages

We have already pointed out that first-order languages are formal languages describing various knowledge domains, but in order to discuss their properties, we will introduce them at an abstract level. A first-order language is defined by the following symbol sets.

**Definition 1.1 (First-order languages).** The set of symbols of each first-order language is composed of two kinds of symbol sets. One is identified as logical symbol sets, whereas
1.1. Symbols of first-order languages

the other is named non-logical symbol sets or symbol sets for the knowledge domains. The logical symbol sets include the following.

\[ V \]: The set of variable symbols. It consists of countable (possibly empty) variable symbols: \( x_1, x_2, \ldots, x_n, \ldots \).

\[ C \]: The set of logical connective symbols. It is composed of the symbols of logical connectives \( \neg, \wedge, \vee, \rightarrow \) and \( \leftrightarrow \) that read as “not”, “and”, “or”, “if \ldots then \ldots”, and “if and only if” respectively.

\[ Q \]: The set of quantifier symbols. It includes \( \forall \) and \( \exists \) that read as “for all \ldots” and “there exists a(n) \ldots” respectively.

\[ E \]: The set containing the equality symbol \( = \).

\[ P \]: The set of parenthesis symbols. It encompasses “(” and “)” that read as “the left parenthesis” and “the right parenthesis”.

In particular, each first-order language has three kinds of non-logical symbol sets of its own.

\[ L_c \]: The set of constant symbols. It consists of countable (including zero) constant symbols: \( c_1, c_2, \ldots \).

\[ L_f \]: The set of function symbols. It is composed of countable (including zero) function symbols: \( f_1, f_2, \ldots \). We use \( f_{x_1 \cdots x_m} \) to denote an \( m \)-ary function symbol, in which \( m \geq 1 \) is the number of variable symbols as well as the number of arguments of the function.

\[ L_p \]: The set of predicate symbols. It consists of countable (including zero) predicate symbols that are represented by \( P_1, P_2, \ldots \). We use \( P_{x_1 \cdots x_m} \) to denote an \( m \)-ary predicate symbol, in which \( m \geq 1 \) is the number of variable symbols as well as the number of arguments of the predicate.

All first-order languages have the same sets of logical symbols, whereas different first-order languages have different non-logical symbol sets. We should point out that \( = \) is also a predicate symbol. Hereafter we shall use \( L \) to represent a first-order language. Since all first-order languages have the same sets of logical symbols, \( L \) essentially represents the sets of non-logical symbols of the first-order language.

In what follows, we shall give an example of a first-order language that describes elementary arithmetic. We will use this frequently in later chapters of the book.

**Example 1.1 (Elementary arithmetic language \( A \)).** The language of elementary arithmetic is a first-order language that will be denoted by \( A \) henceforth. Its constant symbol set, function symbol set, and predicate symbol set are \( \{0\} \), \( \{S, +, \cdot\} \) and \( \{<\} \) respectively.

The purpose of introducing \( S \) is to represent the successor function or “plus 1” function in arithmetic. The binary function symbols \( + \) and \( \cdot \) stand for the addition and multiplication in arithmetic respectively. The predicate symbol \( < \) denotes the “less than” relation between two natural numbers.
Although first-order languages and programming languages have a similar foundation, the way symbols are used in each is different. Firstly, a programming language is more general purpose; any algorithm can be expressed, and new constants and functions can be created for this purpose. A first-order language, on the other hand, is defined to describe one specific domain of knowledge and the constants, functions and predicates are determined by that domain.

Secondly, programming languages allow only a finite number of identifier symbols while first-order languages permit a countably infinite number of separate symbols. As we shall see, knowledge about some mathematical domains can only be captured using an infinite number of symbols.

For the convenience of description and usage, we prescribe that the symbols used in $V, \mathcal{L}_c, \mathcal{L}_f$ and $\mathcal{L}_p$ are different from one another in this book. We also use the lowercase letters $x, y, z, \ldots$ to denote variable symbols, $f, g, h, \ldots$ to denote function symbols and the uppercase letters $P, Q, R, \ldots$ to denote predicate symbols, and thus conform to the conventions of mathematics and knowledge domains. In addition, we will simply refer to the constant symbols and predicate symbols as constants and predicates hereafter if no misunderstanding of the context is incurred.

## 1.2 Terms

Terms are one of the two kinds of syntactic objects of first-order languages. The terms of first-order languages are defined by the same method as that of arithmetic expressions of programming languages, except that the former are more general and allow countably infinite constant, variable, and function symbols.

**Definition 1.2 (Terms).** The terms of a first-order language $\mathcal{L}$ are defined inductively by the following three rules.

- $T_1$: Each constant symbol is a term.
- $T_2$: Each variable symbol is a term.
- $T_3$: If $t_1, \ldots, t_n$ are terms and $f$ is an $n$-ary function symbol, then $ft_1 \cdots t_n$ is a term.

The rules in Definition 1.2 are named as $T$-rules. Henceforth we shall use $\mathcal{L}_T$ to denote the set of terms of the first-order language $\mathcal{L}$.

Definition 1.2 is a structural inductive definition that can also be represented in the following form:

$$ t ::= c \mid x \mid ft_1 \cdots t_n. $$

Here $|$ denotes “or” and ::= denotes “is inductively defined as”. The above form is called the Backus normal form [Backus, 1959].

**Example 1.2 (Terms of $\mathcal{A}$).** The symbol strings

$$ S0, Sx_1, +S0Sx, \cdot x_1 + Sx_1x_2, SS < $$
1.2. Terms

are all terms of $A$ except $SS<.$

For any finite string composed of symbols of $A,$ we can determine if it is a term of $A$ by invoking the $T$-rules in finite steps. In what follows, let us prove that $+S0SSx$ is a term.

(1) 0 is a term. (By $T_1$.)

(2) $x$ is a term. (By $T_2$.)

(3) $S0$ is a term. (By $T_3$ since $S$ is a unary function symbol and 0 is a term according to (1).)

(4) $Sx$ is a term. (By $T_3$ since $S$ is a unary function symbol and the variable $x$ is a term according to (2).)

(5) $SSx$ is a term. (By $T_3$ since $S$ is a unary function symbol and $Sx$ is a term according to (4).)

(6) $+S0SSx$ is a term. (By $T_3$ since $+$ is a binary function symbol and both $S0$ and $SSx$ are terms according to (3) and (5).)

The intention of introducing terms is to describe the constants, variables and functions in a knowledge domain. Definition 1.2 says that each term is just a symbol string whose symbols are from the symbol sets $V,$ $L_c$ and $L_f,$ and whose constructions are in strict accordance with the $T$-rules. Definition 1.2 does not concern the meaning of terms at all. When discussing the semantics of first-order languages in Chapter 2, we shall see that terms are interpreted as constants, variables or functions of a domain.

In Example 1.2, the function symbol $S$ can be used repeatedly. This is a characteristic of structural inductive definitions. Hereafter we write

$$S^00 \text{ for } 0 \text{ and } S^{n+1}0 \text{ for } S(S^n0), \text{ and thus } S^n0 \text{ for } \underbrace{SS \cdots SS}_n0.$$ $S^n0$ is only an abbreviation whose superscript $n$ stands for “making successor operations $n$ times.”

We should also point out that, except for the lack of parentheses, the representations of terms in first-order languages are basically the same as those of constants, variables and functions in programming languages and textbooks in mathematics. The representations for the terms $S0$ and $Sx_1$ are called prefix representations. They are usually written as $S(0)$ and $S(x_1)$ in programming languages and textbooks in mathematics. The conventional representations of $+Sx_1x_2$ and $\cdot x_1 + Sx_1x_2$ are $S(x_1) + x_2$ and $x_1 \cdot (S(x_1) + x_2)$ respectively. In this book, we shall use the conventional representations of these frequently used functions for the convenience of reading and understanding, provided that no misunderstandings of the context are incurred. Strictly speaking, these are no longer the terms of $A$ specified by Definition 1.2 but their aliases.
1.3 Logical formulas

The other kind of syntactic objects in first-order languages are logical formulas. They are the ‘first class’ syntactic objects of first-order languages and are defined by structural induction.

**Definition 1.3 (Logical formulas).** The logical formulas of a first-order language $L$, or called formulas for short, are represented by uppercase letters $A$, $B$, ... and are defined inductively by the following five $F$-rules.

- \( F_1 \) : If \( t_1 \) and \( t_2 \) are terms, then \( t_1 = t_2 \) is a formula.
- \( F_2 \) : If \( t_1, \ldots, t_n \) are \( n \) terms and \( R \) is an \( n \)-ary predicate, then \( Rt_1 \cdots t_n \) is a formula.
- \( F_3 \) : If \( A \) is a formula, then \( \neg A \) is a formula.
- \( F_4 \) : If \( A \) and \( B \) are formulas, then \( A \land B \), \( A \lor B \), \( A \rightarrow B \), \( A \leftrightarrow B \) are all formulas.
- \( F_5 \) : If \( A \) is a formula and \( x \) is a variable, then \( \forall x A \) and \( \exists x A \) are also formulas. In this case, \( x \) is called a bound variable.

The formulas defined by the rules \( F_1 \) and \( F_2 \) are identified as atomic formulas, whereas the formulas defined by the rules \( F_3 \), \( F_4 \) and \( F_5 \) are called composite formulas.

The formula \( \neg A \) reads as the “negation of formula \( A \)” or “not \( A \)”, \( A \land B \), \( A \lor B \), \( A \rightarrow B \) and \( A \leftrightarrow B \) read as the “conjunction of formulas \( A \) and \( B \)”, “disjunction of formulas \( A \) and \( B \)”, “\( A \) implies \( B \)” and “\( A \) is equivalent to \( B \)” respectively. \( \forall x A \) and \( \exists x A \) are identified as quantified formulas with \( A \) being the body of the formula. \( \forall x A \) and \( \exists x A \) read as “for all \( x \), \( A \)” and “there exists an \( x \) such that \( A \)” respectively. The redundant parentheses in the formula can be omitted without changing the meaning of a formula. Henceforth we use the notation \( L_F \) to denote the set of the formulas in a first-order language \( L \).

The Backus normal form defined by the above structural induction is:

\[
A ::= t_1 \uparrow t_2 \mid Rt_1 \cdots t_n \mid \neg A \mid A \land B \mid A \lor B \mid A \rightarrow B \mid A \leftrightarrow B \mid \forall x A \mid \exists x A.
\]

**Example 1.3 (The formulas of $\mathcal{A}$).** According to Definition 1.3, we can determine if the symbol strings

\[
\forall x \neg (Sx = 0) \quad \text{and} \quad \forall x \forall y (<xy \rightarrow (\exists z(y = +xz)))
\]

are formulas of $\mathcal{A}$. In what follows, we prove that the symbol string \( \forall x \neg (Sx = 0) \) is a formula of $\mathcal{A}$. 
1.4 Free variables and substitutions

(1) $Sx \neq 0$ is a formula. (By $F_1$ since both $Sx$ and 0 are terms according to Example 1.2.)

(2) $\neg(Sx \neq 0)$ is a formula. (By $F_3$ and (1).)

(3) $\forall x \neg (Sx \neq 0)$ is a formula. (By $F_5$ and (2).)

Similarly, we can prove that the symbol string $\forall x \forall y(<xy \rightarrow (\exists z(y \neq +xz)))$ is also a formula.

According to Definition 1.3, each logical formula is a finite symbol string constructed in strict accordance with the $F$-rules. Definition 1.3 tells how a formula is defined syntactically, but it does not concern its meaning. Logical formulas describe propositions about the domain of knowledge. For example, the formula $\forall x \neg (Sx \neq 0)$ denotes the proposition “each natural number is greater than or equal to 0” and the formula $\forall x \forall y(<xy \rightarrow (\exists z(y \neq +xz)))$ expresses the proposition “if $x < y$, then there must exist a natural number $z$ such that $y = x + z$.” In the next chapter, we will introduce a method of interpreting each symbolic formula by a semantic proposition about a domain.

An advantage of introducing first-order languages is that by symbolizing the constants, functions, equations, predicates, logical connectives and quantifiers in propositions of a domain, the logical structures implicit in propositions can be made explicit. In this way, the logical reasoning about the propositions can be converted into symbolic calculus.

1.4 Free variables and substitutions

Local variables are allowable in programming languages with each variable having a specific scope within which the variable is bound and available. In addition, programmers are allowed to use formal parameters in the declarations of procedures and functions. The formal parameters are a type of variable which are substituted by real parameters when the procedures and functions are called. The ideas of local variables, formal parameters, real parameters and substitutions in programming languages coincide with those of bound variables, free variables and substitutions in first-order languages. In first-order languages, variables may be bound by quantifier symbols. Let us look at an example first.

Example 1.4. Suppose that $x$, $y$, $z$ are three different variables in the formula

$$A : \exists x((P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z)),$$

where $P$, $R$, $Q$ are three binary predicates. The variable $x$ in $P$, $R$, $Q$ is bound by the outermost quantifier symbol $\exists$ with its scope being $((P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z))$. The variable $y$ in $R$ is bound by the quantifier $\forall$ with its scope being $R(x,y)$. Nonetheless, $y$ in $P$ and $z$ in $Q$ are not bound by any quantifiers in the formula and they occur free in the formula $A$; they are the free variables of $A$.

Definition 1.4 (Free variables of terms). Suppose that $t$ is a term of $\mathcal{L}$ with $FV(t)$ being the set of free variables of $t$. According to the syntactic structure of terms, $FV(t)$ is
According to Definitions 1.4 and 1.5, we can determine the free variables

\[ FV(x) = \{x\}, \quad x \text{ is a variable.} \]
\[ FV(c) = \emptyset, \quad c \text{ is a constant symbol.} \]
\[ FV(ft_1 \cdots t_n) = FV(t_1) \cup \cdots \cup FV(t_n). \]

If \( x \in FV(t) \), then we identify \( x \) as a free variable of \( t \) or say that \( x \) occurs free in \( t \).
If \( FV(t) = \emptyset \), then we call \( t \) a ground term (or closed term).

**Definition 1.5 (Free variables of formulas).** Suppose that \( A \) is a formula of \( \mathcal{L} \), \( FV(A) \) being the set of free variables of \( A \). We define the set of free variables of \( A \), \( FV(A) \), inductively as follows.

1. \( FV(t_1 \doteq t_2) = FV(t_1) \cup FV(t_2). \)
2. \( FV(Pt_1 \cdots t_n) = FV(t_1) \cup \cdots \cup FV(t_n). \)
3. \( FV(\neg A) = FV(A). \)
4. \( FV(A \ast B) = FV(A) \cup FV(B), \quad \text{where } \ast \text{ stands for any of } \land, \lor, \rightarrow, \leftrightarrow. \)
5. \( FV(\forall xA) = FV(A) - \{x\}. \)
6. \( FV(\exists xA) = FV(A) - \{x\}. \)

If \( x \in FV(A) \), then we identify \( x \) as a free variable of the formula \( A \) or say that \( x \) occurs free in \( A \). If \( FV(A) = \emptyset \), then we call \( A \) a sentence, which is a formula encompassing no free variables.

**Example 1.5.** According to Definitions 1.4 and 1.5, we can determine the free variables in \( \exists x((P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z)) \) as follows.

\[
FV(\exists x((P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z))) \\
= FV(((P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z)) - \{x\} \\
= (FV(P(x,y)) \cup \forall y R(x,y)) \cup \{x\} - \{x\} \\
= ((\{x,y\} \cup \{y\}) \cup \{x,z\} - \{x\} \\
= (\{x\} \cup \{y\} \cup \{x,z\}) - \{x\} \\
= (\{x,y,z\}) - \{x\} \\
= \{y,z\}.
\]

We should point out that \( y \) here is the variable \( y \) in \( P(x,y) \), not the one in \( R(x,y) \).

In programming languages, a formal parameter used in the declaration of a function can be substituted by a real parameter. In the same way, a free variable in a term or formula can be substituted by a term, creating a new instance of that expression. The following definition makes this procedure precise.
1.4. Free variables and substitutions

Definition 1.6 (Substitution of terms). Let $s$ and $t$ be terms. Denote by $s[t/x]$ the term obtained from $s$ by substituting the term $t$ for the free variable $x$ of $s$. According to the structure of terms, $s[t/x]$ is inductively defined as follows.

$$
y[t/x] = y, \quad \text{if } y \neq x.
$$
$$
y[t/x] = t, \quad \text{if } y = x.
$$
$$
c[t/x] = c, \quad c \text{ is a constant symbol.}
$$
$$
f t_1 \cdots t_n[t/x] = f t_1[t/x] \cdots t_n[t/x].
$$

Note that the equal sign $=$ in the above definition refers to the equality of elements in a set, which is different from $\doteqdot$ in first-order languages. The equality symbol $\doteqdot$ is a specific predicate symbol of first-order languages.

Definition 1.7 (Substitution of formulas). Let $A$ be a formula containing a free variable $x$. $A[t/x]$ stands for the formula obtained from $A$ by substituting the term $t$ for the free variable $x$ of $A$. It is abbreviated sometimes to $A[t]$. According to the syntactic structure of formulas, $A[t/x]$ is inductively defined as follows.

1. $((t_1 \doteqdot t_2)[t/x]) = (t_1[t/x] \doteqdot t_2[t/x])$.
2. $R t_1 \cdots t_n[t/x] = R t_1[t/x] \cdots t_n[t/x]$.
3. $(\neg A)[t/x] = \neg (A[t/x])$.
5. $(\forall x A)[t/x] = \forall x A$.
6. $(\exists x A)[t/x] = \exists x A$.
7. $(\forall y A)[t/x] = \forall y A[t/x]$, if $y \not\in FV(t)$.
8. $(\exists y A)[t/x] = \exists y A[t/x]$, if $y \not\in FV(t)$.
9. $(\forall y A)[t/x] = \forall z A[z/y][t/x]$, if $y \in FV(t)$, $z \not\in FV(t)$, $z$ does not occur in $A$.
10. $(\exists y A)[t/x] = \exists z A[z/y][t/x]$, if $y \in FV(t)$, $z \not\in FV(t)$, $z$ does not occur in $A$.

In rules (9) and (10), the conditions $z \not\in FV(t)$ and $z$ not occurring in $A$ indicate that the variable $z$ is a new variable with respect to $t$ and $A$, that is, $z$ is neither a free variable of $t$ nor a free variable or bound variable of $A$.

Example 1.6 (Substitution). Let $t = fc$ with $f$ being a unary function symbol and $c$ being a constant symbol. We substitute $t$ for the free variable $y$ of the formula in Example 1.5 as follows.

$$
(\exists x (P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z))[fc/y]
$$
$$
= \exists x (((P(x,y) \land \forall y R(x,y)) \rightarrow Q(x,z))[fc/y])
$$
$$
= \exists x ((P(x,y) \land \forall y R(x,y))[fc/y] \rightarrow Q(x,z)[fc/y])
$$
$$
= \exists x ((P(x,y)[fc/y] \land (\forall y R(x,y))[fc/y] \rightarrow Q(x,z))
$$
$$
= \exists x ((P(x,fc) \land \forall y R(x,y)) \rightarrow Q(x,z)).
$$
Definition 1.7 provides three groups of substitution rules for quantified formulas. The first group consists of rules (5) and (6), which prescribe that we can only substitute for the free variables in a quantified formula. The second group is composed of rules (7) and (8), which indicate that if the bound variable of a quantified formula is not a free variable of the term \( t \), then the substitution in the quantified formula amounts to a substitution in its body. The third group consists of rules (9) and (10), whose usage is demonstrated by the following example.

**Example 1.7.** Suppose that \( A = \exists y (y < x) \) and let \( t = y \). Consider a substitution \( A[t/x] \). Since \( x \neq y \), if we invoke the second group of rules, then we shall have
\[
(\exists y (y < x))[y/x] = \exists y (y < y),
\]
which does not coincide with our experience. In fact, if we interpret \( A \) as

“for any integer \( x \), there exists a \( y \) such that \( y < x \) holds”,

then the proposition is true for integers. We certainly hope that the proposition is still true after a substitution for \( x \). Nonetheless after the substitution the proposition becomes

“there exists an integer \( y \) such that \( y < y \) holds”,

which is false. The problem lies in the fact that \( y \) substituted for \( x \) is a bound variable of \( A \). Generally speaking, if the free variables of \( t \) are not the bound variables of \( A \), which is the condition \( y \notin FV(t) \) for the second group of rules, then we make the substitution according to the second group of rules. If a free variable of \( t \) is by any chance also a bound variable of \( A \), then the condition \( y \in FV(t) \) for the third group of rules holds for this variable.

In this case, if we still make a substitution according to the second group of rules, then we shall make the mistake as described in the above example. The solution is to introduce a new variable \( z \) that is neither a free variable of \( t \) nor a free variable or a bound variable of \( A \). When making the substitution, we first substitute \( z \) for the bound variable \( y \) of the quantifier such that the free variable of \( t \) is no longer the bound variable of \( A \). Then we make the substitution \( [t/x] \) according to the second group of rules and the mistake is avoided. This is the motivation for rules (9) and (10).

According to rule (10), the correct solution for the above example is
\[
(\exists y (y < x))[y/x] = (\exists z (y < x)[z/y])[y/x] \\
= \exists z (z < x)[y/x] \\
= \exists z (z < y).
\]

The result of the substitution can be interpreted as

“for any integer \( y \), there exists an integer \( z \) such that \( z < y \) holds,”

which lives up to our expectation.

In summary, if \( A \) is the quantified formula \( \forall y B \) or \( \exists y B \), then there are two groups of rules for making the substitution \( A[t/x] \). If \( y \notin FV(t) \), then we say that \( t \) is free for \( A \) with respect to \( y \) and we use the rules (7) and (8). If \( y \in FV(t) \), then we say that \( t \) is bound by \( A \) with respect to \( y \) and we can only use rules (9) and (10).
1.5 Gödel terms of formulas

Although the terms and formulas of first-order languages are two different categories of syntactic objects, it is sometimes possible to convert one into the other. In this section, we show how this can be done in the language of elementary arithmetic. This method is called Gödel coding [Shoenfield, 1967]. The basic idea is to first code every formula \( A \) by a natural number \( \&A \), called the Gödel number of \( A \). Then the natural number \( \&A \) is made to correspond to a term \( S^{\&A}0 \), called the Gödel term of \( A \). The integration of these two steps represents each formula \( A \) in \( \mathcal{A} \) by a term \( S^{\&A}0 \) and the representation is bijective.

Gödel coding is analogous to the mechanism of indirect addressing in computer instructions and pointers in programming languages. Let us illustrate this analogy using the language C. Suppose that \( x \) is an integer variable and \( p \) is an integer pointer, with \( \&x \) denoting the address of \( x \). After the execution of the statement \( p = \&x \), \( p \) points to the address of \( x \) and \( *p \) represents the content stored in the address \( \&x \).

The analogy with Gödel coding amounts to regarding each formula \( A \) (a symbol string) in a first-order language as the name of a variable in C, whose Gödel number \( \&A \) is the storage address of the variable \( A \) and the Gödel term \( S^{\&A}0 \) is the content stored at the address \( \&A \).

Gödel coding is defined inductively. First, we define the concept of ordinal number in Gödel coding.

**Definition 1.8 (Ordinal number).** Suppose that \( a_1, a_2, \ldots, a_n \) are natural numbers. \( < a_1, a_2, \ldots, a_n > \) is called the ordinal number of \( a_1, a_2, \ldots, a_n \) and it represents the natural number \( p_1^{a_1+1} p_2^{a_2+1} \cdots p_n^{a_n+1} \) with \( p_1, \ldots, p_n \) being the first \( n \) prime numbers. Namely,

\[
< a_1, a_2, \ldots, a_n > = p_1^{a_1+1} p_2^{a_2+1} \cdots p_n^{a_n+1},
\]

where \( a_i \) (\( 0 < i \leq n \)) is called the \( i \)-th element of this ordinal number.

An ordinal number is a natural number. Any element \( a_i \) of an ordinal number will still be an ordinal number. Nonetheless, not every natural number is an ordinal number and, as an example, 0 is not an ordinal number.

**Definition 1.9 (Gödel coding).** The Gödel coding of \( \mathcal{A} \) is a map \( \& : \mathcal{A} \mapsto \mathbb{N} \). \& maps each symbol, term or formula of \( \mathcal{A} \) to a natural number. According to the syntactic structure of symbols, terms and formulas of \( \mathcal{A} \), \& is inductively defined as follows.

1. **Symbols**

   \[
   \begin{align*}
   \&() &= 1, \\
   \&(\neg) &= 13, \\
   \&(\lor) &= 15, \\
   \&(\land) &= 17, \\
   \&(\to) &= 19, \\
   \&(\equiv) &= 21, \\
   \&(\exists) &= 23.
   \end{align*}
   \]
(2) Variables

\[ \&(x_n) = 25 + 2 \cdot n, \quad n \in \mathbb{N}. \]

It should be noted that the number 25 could be replaced by any odd number greater than 23 to allow us to introduce more symbols. This will be seen in Chapter 5.

(3) Terms

\[ \& (S t) = < \& S, \& t >, \]
\[ \& (t_1 \cdot t_2) = < \& (\cdot), \& t_1, \& t_2 >, \quad \text{where } \cdot \text{ stands for any of } +, \cdot. \]

(4) Formulas

\[ \& (t_1 \cdot t_2) = < \& (\cdot), \& t_1, \& t_2 >, \quad \text{where } \cdot \text{ stands for any of } <, \div, \]
\[ \& (\neg A) = < \& (\neg), \& A >, \]
\[ \& (A \cdot B) = < \& (\cdot), \& A, \& B >, \quad \text{where } \cdot \text{ stands for any of } \land, \lor, \to, \leftrightarrow, \]
\[ \& (\forall x_n A) = < \& (\forall), \& (x_n), \& A >, \]
\[ \& (\exists x_n A) = < \& (\exists), \& (x_n), \& A >. \]

Example 1.8 (Gödel number). According to the rules of Gödel coding, we can determine effectively the Gödel number of each formula. For example, let \( A \) be a formula of the form

\[ \forall x_3 \exists x_1 x_3 = x_1 + x_2. \]

The Gödel number of \( A \) is

\[ \& (\forall x_3 \exists x_1 x_3 = x_1 + x_2) \]
\[ = < \& (\forall), \& (x_3), \& (\exists x_1 x_3 = x_1 + x_2) > \]
\[ = < 21, 31, \& (\exists x_1 x_3 = x_1 + x_2) > \]
\[ = < 21, 31, \langle 23, 27, \& (x_3 = x_1 + x_2) > \rangle \]
\[ = < 21, 31, \langle 23, 27, \langle 9, 31, \{5, 27, 29\} > \rangle > \]
\[ = 2^{21+1} \cdot 3^{31+1} \cdot 5^{(23, 27, \langle 9, 31, \{5, 27, 29\} > \rangle + 1} \]
\[ = 2^{21+1} \cdot 3^{31+1} \cdot 5^{2^{23+1} \cdot 3^{27+1} \cdot 1.5^{\langle 9, 31, \{5, 27, 29\} > \rangle + 1 + 1} \]
\[ = 2^{21+1} \cdot 3^{31+1} \cdot 5^{2^{23+1} \cdot 3^{27+1} \cdot 1.5^{\langle 9, 31, \{5, 27, 29\} > \rangle + 1 + 1 + 1 + 1}. \]

The following lemma indicates that Gödel coding establishes a one-to-one correspondence between \( \mathcal{A} \) and Gödel numbers.

**Lemma 1.1.** Gödel coding is a one-to-one map from \( \mathcal{A} \) to the set of Gödel numbers.
1.6. Proof by structural induction

Proof. The conclusion follows directly from the unique factorization theorem of prime numbers and the fact that the ordinal numbers are even so the odd variable codes will never coincide with them. □

Definition 1.10 (Gödel term). Let \( A \) be a formula of \( \mathcal{A} \) and the Gödel number of \( A \) is \&A. The Gödel term of \( A \) is \( S^{\&A}0 \).

Example 1.9 (Gödel term). The Gödel term of the formula \( \forall x_3 \exists x_1 x_3 \neq x_1 + x_2 \) is

\[
S^{2^{21}+1 \cdot 3^{31}+1 \cdot 5^{23}+1 \cdot 3^{27}+1 \cdot 5^{9}+1 \cdot 3^{31}+1 \cdot 5^{5}+1 \cdot 3^{27}+1 \cdot 5^{9}+1 + 1 + 1 + 1}.
\]

If \( \mathcal{L} \) is a first-order language extending \( \mathcal{A} \) which contains extra symbols, then we can still define their Gödel numbers and Gödel terms using the above method. We will see in Chapter 5 that the original intention of Gödel was to represent the self-referential statements in first-order languages so as to prove the incompleteness of formal theories. Nonetheless the idea of Gödel coding inspired the development of indirect addressing in computer hardware as well as the pointers in programming languages. In this sense, Gödel is the pioneer of these mechanisms.

1.6 Proof by structural induction

In the previous sections, the terms, formulas, free variables and substitutions of first-order languages are all defined by structural induction. In this section, we show how to use the inductive nature of these definitions to prove general properties of formulas in first-order languages.

Let’s take the definition of formulas as an example. By structural induction, we first define the atomic formulas, which are equations and predicates, and then define the composite formulas by three \( F \)-rules (actually seven rules). These rules tell us how a composite formula is constructed from its components. Each \( F \)-rule can be written in a mathematical form. For instance, the rule on the disjunction formula in \( F_3 \) is “if \( A \) and \( B \) are formulas, then \( A \lor B \) is a formula”, which can be written in the form of a ‘fraction’

\[
\frac{A \quad B}{A \lor B}.
\]

We should point out that \( A \) and \( B \) in the numerator of the fraction represent any logical formulas. Hence the above rule is a ‘schema’ to generate disjunction formulas. In general, each rule in a definition through structural induction can be written in the form of a ‘fraction’ as follows:

\[
\frac{X_1 \cdots X_n}{X},
\]

where the uppercase letters \( X_1, \ldots, X_n \), \( X \) represent well-formed objects. The objects \( X_1, \ldots, X_n \) in the numerator of the fraction are identified as the premise and the denominator \( X \) of the fraction is called the conclusion of the rule. This rule can be interpreted as: if the premise \( X_1, \ldots, X_n \) holds, then the conclusion \( X \) holds.
In mathematical investigations, we often need to prove that a class of objects possess a certain property, which is usually the most difficult part of the whole investigation. Nowadays there are still many mathematical conjectures with their rigorous proofs pending. Nonetheless, if an object is defined by structural induction, then the proof of its properties may become rather simple and even turn into a kind of routine schema. The reason is that it suffices to verify under such circumstances that the atomic objects possess the property and each composite object possesses the property, from which we can deduce that all objects possess the property.

The composite object is a conclusion of a certain rule according to the definition by structural induction. Thus it suffices to prove that, for every rule defining composite objects, the premises having the property implies the conclusion also has the property. This kind of proof method is called the proof by structural induction, or structural induction for short. It can be strictly stated as follows.

**Method 1.1 (Structural induction).** Suppose that a set $Z$ is defined by a group of rules. To prove that the set $Z$ possesses a property $\Psi$, we only need to prove the following.

$I_1$: Each atomic object that is directly defined possesses the property $\Psi$;

$I_2$: For each rule

$$
\frac{X_1 \cdots X_n}{X},
$$

if $X_1, \ldots, X_n$ all possess the property $\Psi$, then we can prove that $X$ also possesses the property $\Psi$.

$I_1$ is called the *induction basis*. The condition “if $X_1, \ldots, X_n$ all possess the property $\Psi$” specified in $I_2$ is identified as the *induction hypothesis*.

The proof method of structural induction can be applied to the proofs of terms and formulas, which can be summarized in the following proof schema.

**Method 1.2 (Proof that terms possess the property $\Psi$).** To prove that each term possesses the property $\Psi$, we only need to prove:

$T'_1$: Each variable possesses the property $\Psi$;

$T'_2$: Each constant possesses the property $\Psi$;

$T'_3$: If terms $t_1, \ldots, t_n$ all possess the property $\Psi$ and $f$ is an $n$-ary function symbol, then $ft_1 \cdots t_n$ also possesses the property $\Psi$.

**Method 1.3 (Proof that formulas possess the property $\Psi$).** To prove that each formula possesses the property $\Psi$, we only need to prove:

$F'_1$: $t_1 \doteq t_2$ possesses the property $\Psi$;

$F'_2$: For any $n$-ary predicate symbol $R$ and terms $t_1, \ldots, t_n$, $Rt_1 \cdots t_n$ possesses the property $\Psi$;

$F'_3$: If $A$ possesses the property $\Psi$, then so does ($\neg A$);
1.6. Proof by structural induction

$F'_4$: If $A$ and $B$ both possess the property $\Psi$, then so do $(A \wedge B)$, $(A \lor B)$, $(A \rightarrow B)$ and $(A \leftrightarrow B)$;

$F'_5$: If $A$ possesses the property $\Psi$, then so do both $\forall x A$ and $\exists x A$.

Let us look at the following example.

**Example 1.10.** For any given first-order language $\mathcal{L}$, every formula in $\mathcal{L}$ contains an equal number of left parentheses “(” and right parentheses “)”.

**Proof.** We prove the conclusion first on terms by structural induction.

$T'_1$: Every variable $x$ has no parenthesis and thus the conclusion holds.

$T'_2$: Every constant $c$ also has no parenthesis and thus the conclusion holds as well.

$T'_3$: For any term $ft_1 \cdots t_n$ with $f$ being an $n$-ary function symbol, every term $t_i$ ($i = 1, \ldots, n$) contains an equal number of left and right parentheses according to the assumption of structural induction. As per $T'_3$ of Method 1.2, no new parenthesis is added to the term $ft_1 \cdots t_n$ and the number of left or right parentheses contained in it equals the total number of left or right parentheses contained in $t_1, \ldots, t_n$. Thus the conclusion holds for terms.

The proof by structural induction on formulas proceeds as follows.

$F'_1$: The conclusion holds for $t_1 \equiv t_2$. Since $t_1$ and $t_2$ are terms, the conclusion holds for $t_1$ and $t_2$ according to the first part of the proof, and no new parenthesis is added to the formula $t_1 \equiv t_2$.

$F'_2$: The conclusion holds for $Rt_1 \cdots t_n$. The reason is that $t_1, \ldots, t_n$ are all terms and as per the first part of the proof, the conclusion holds for $t_1, \ldots, t_n$; $R$ is an $n$-ary predicate that contains no parenthesis in itself and thus no new parenthesis is added to the formula $Rt_1 \cdots t_n$.

$F'_3$: Suppose that $A$ is a formula that contains an equal number of left and right parentheses. According to Definition 1.3, $(\neg A)$ contains an equal number of left and right parentheses.

$F'_4$: Suppose that the conclusion holds for both the formulas $A$ and $B$. If we assume that $A$ contains $n$ left parentheses and $m$ right parentheses and $B$ contains $m$ left parentheses and $m$ right parentheses, then according to Definition 1.3 the formula $(A \wedge B)$ contains $n + m + 1$ left parentheses and $n + m + 1$ right parentheses. Thus the conclusion holds for $(A \wedge B)$. Similarly we can prove that the conclusion holds for $(A \lor B)$, $(A \rightarrow B)$ and $(A \leftrightarrow B)$ as well.

$F'_5$: Suppose that formula $A$ contains an equal number of left and right parentheses. According to the definition, the numbers of the left parentheses and right parentheses contained in $\forall x A$ or $\exists x A$ equal the numbers of those contained in $A$ respectively.

The conclusion is proved. □

In fact, any property that can be proved by structural induction can also be proved by mathematical induction. In this sense, we say that the proofs using the structural induction method are rational. The bridge connecting the structural induction method and the mathematical induction method is the rank of terms and formulas.